BXC-51

BASIC-51 Cross-Compiler
Version 5.0

By TAVVE Software Co.
Written by Anthony V. Edwards

Technical Support

Technical support is provided by sending a fax to (508) 369-9549 or calling (508) 369-9556 and asking for technical support. Bug reports may be faxed directly to (919) 405-2131.

To receive technical support, you must provide the following information: the serial number of your software, the registered owner's name, and a summary of the problem. If the owner has not registered his/her software, then no support can be provided. To receive phone technical support, please call (508) 369-9556 and ask for BXC-51 Technical Support.

Guarantee

If you are not completely satisfied with the product, return the disk and manual in good condition within 30 days from the date of purchase to Binary Technology, Inc. and we will send to you a refund less our standard re-stocking fee.

BXC-51

Table of Contents

1. Introduction

Binary Technology, Inc. has been providing engineers and programmers with hardware and software solutions based on the Intel MCS-51 family of processors since 1982. It continues to be our goal to produce products that are reliable, easy-to-use, relevant, and affordable.

BXC-51 is a powerful software tool for reducing development time and costs. It's easy to use, as well.

BXC-51 Version 5.0 is the latest release of BXC-51, the first cross-compiler available for Intel's MCS BASIC-52 interpreted BASIC, Version 1.1 for the 8052/32 and DS5000 microcontrollers. It adheres to the Intel standard complete with interrupt support, I/O, and real-time clock. In-line assembly is supported. The BXC-51 compiler produces an Intel hex format file compatible with all PROM programmers, monitor-debuggers (such as Binary Technology, Inc.'s M/DP), or in-circuit emulators. BXC-51 produces an assembly language intermediate file allowing close examination or optimization by the user.

BXC-51 compiled code runs faster than the BASIC-52 interpreter, provides source-code security, and allows use of the less expensive 8031/8032 microcontrollers. The BXC-51 compiler provides flexibility of external memory, program, and I/O locations. BXC-51supports both integer, byte, floating point and string data types. BXC-51 generates code for the greater 8051 family of microcontrollers. Command line options directly support the 8051/31, 8052/32, and DS5000 but any 8051 derivative can be supported through configuration.

This manual assumes that the user is familiar with programming in BASIC. Familiarity with assembly language programming is not necessary. For additional information, we recommend "The Intel MCS BASIC-52 Users Manual" Intel Order No. 270010-003, and "BASIC-52 Programming", by Systronix (available from Binary Technology, Inc.). For beginning texts on BASIC, consult your local computer store or bookstore.

The final page of this documentation is available for you to make suggestions, criticisms, and comments on enhancements you think would be appropriate to our current products, as well as products you would like to see us develop.

Thank you for purchasing the BXC-51. We are confident that it will be a significant aid to you in your 8031/8051 (and family derivatives) development projects.

The BASIC Language

BASIC is a general-purpose, high level programming language originally designed by Thomas E. Kurtz and John G. Kemeny at Dartmouth College in 1964. BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. It was designed to bring programming to everyone. BASIC evolved originally from mainframes to minis to personal computers in the 1980s. With the IBM PC and compatible computers, BASIC was distrubuted to everyone.

Today, we have BASIC-52, the de facto BASIC standard for the 8051 microcontroller family from Intel, written by John Katausky of Intel in 1985. BASIC-52 is very much a subset of modern BASIC, but it retains the essential elements. It has English commands and functions which are easy to remember. It is line oriented with each line beginning with a line number. It's convenience makes it a popular tool for developing microcontroller applications. Using an interpreter, programs are easily entered, modified, and executed.

BXC-51 BASIC

BXC-51 V1.0 compiled existing BASIC-52 applications into complete, standalone Assembly programs. Since V1.0, BXC-51 has extended the BASIC-52 language in a number of ways. Most particularly, new variable data types and command extensions have been added.

BXC-51 has introduced integer, byte, bit, and dynamic string variables. Variables may be memory mapped to specific memory locations. Additional special variables have been added for additional special function registers and to improve error tracking.

BXC-51 interfaces with Assembly easily. The very BASIC lanugage itself may be extended by using BASIC Extension Libraries (BXL's). Using BXL's allows you to create command and function extensions to BXC-51 which seamlessly integrate into your BASIC code. Or you may be using a BXL provided by your local dealer to support special I/O devices. Or, simply, an assembly routine may be mapped to a BASIC keyword, using DEFASM, to reduce many CALL statements in code.

A number of smaller, but very useful, changes have been made as well. BASIC line numbers are now optional. Line labels may be used instead of line numbers. Greater debugging control is allowed with the TRACE1 command. Serial input can be buffered with SBUFFER. Bit shifting operators allow you to bit shift numbers left or right. BXC-51 brings BASIC-52 directly to the 8051/31 and DS5000 microcontrollers as well as derivative microcontrollers, such as the 8xC550 or 8xC552, indirectly through customization control.

System Requirements

Before you begin installation, make sure you have the minimum system requirements:

1. A PC-DOS or MS-DOS based machine.
2. 384K of RAM.
3. One floppy disk drive.
4. MS-DOS Version 2.0 or later.

Installation

Insert the *original* BXC-51 disk into your floppy drive. To begin the installation, type this command:

```
C>    A:INSTALL
```

If, however, you inserted the disk in your B: drive, type this command:

```
C>    B:INSTALL
```

The INSTALL program is very user-friendly. Follow all its instructions. It will first ask you for your name. Identify yourself, your company name, or both. Do not use initials. Spell out the words in the space provided.

Next, INSTALL will ask where you wish to install BXC-51. The default is in `C:\BXC`. Specify another directory if this default is not satisfactory.

INSTALL now procedes to copy the files off your disk into the target directory. Messages will be displayed, telling you which files are being read, written, or configured.

When INSTALL finishes, BXC-51 is completely installed.

Make a backup of your original disk and store it in a safe place.

Files Installed

A number of files are on the distribution disk. However, only three are required to run BXC-51. They are BXC51.EXE, BXC51.LIB, and SXA51.EXE. Additionally, HEX.EXE and RENUM.EXE are provided as useful utility programs.

BXC51.EXE - This is the BASIC-51 Cross-Compiler. This program takes a BASIC source code file and converts it to assembly. To generate assembly code, BXC51.EXE reads sections of the file named BXC51.LIB file. Once the assembly is generated, the SXA51.EXE assembler is invoked to generate an Intel formatted HEX file.

BXC51.LIB - This is the library for the BASIC system routines. It is a compressed and coded copy of source code. When compiling your program, only the needed routines will be extracted from BXC51.LIB.

SXA51.EXE - This is Binary Technology, Inc.'s 8051/8052 cross-assembler. The assembler is invoked once all the assembly code has been generated. This program will always compile without any errors. Any errors in your BASIC source code will be caught by BXC-51.

RENUM.EXE - This is a BASIC-52 line renumbering utility. Use it to renumber all or part of your BASIC-52 programs.

HEX.EXE - This is Binary Technology, Inc.'s Intel Hex file manipulation utility.

In addition to the program files, there are many BASIC examples in .BAS files. These files are:

```
     ARRAY.BAS         EXPR.BAS          LOAD_B.BAS        SEEBAUD.BAS
     ASM.BAS           FOR.BAS           LOAD_F.BAS        SHLR.BAS
     BAUD.BAS          GOSUB.BAS         LOAD_I.BAS        SIEVE_B.BAS
     BMARK_B.BAS       HELLO.BAS         LOGIC.BAS         SIEVE_F.BAS
     BMARK_F.BAS       IE.BAS            MEM.BAS           SIEVE_I.BAS
     CALL.BAS          IF.BAS            ONGO.BAS          SIN.BAS
     CLEAR.BAS         IF2.BAS           PRINT.BAS         STRING.BAS
     DBY.BAS           IFN.BAS           PUSH.BAS          TAB.BAS
     DEFS.BAS          INPUT.BAS         READ.BAS          TIMER.BAS
     DIM.BAS           LDST.BAS          READ2.BAS         USING.BAS
          DIV0.BAS          LET.BAS           REM.BAS
WHILE.BAS
```

The simplest of these examples is HELLO.BAS which outouts the message "Hello, World!" to the serial port and exits. This is a good place to start when compiling your first BASIC program.

The example files are *not* copied from your original BXC-51 diskette during installation. If you wish to copy them, you will need to type the DOS command:

```
     COPY A:*.BAS C:\BXC
```

If you are anxious to see your new compiler work, try compling HELLO.BAS by following the instructions in the next section.

Getting Started Fast

If you do not have time now to read this manual thoroughly and are anxious to compile your BASIC code, this section contains the minimum of what you need to know to begin.

1. Be sure that BXC51.EXE, BXC51.LIB, and SXA51.EXE are in the same directory.

2. To compile your BASIC file, type:
   `BXC51 myfile`
   at the DOS prompt. For example,
   `BXC51 HELLO`
   to compile HELLO.BAS

3. This will generate a file, `myfile.HEX`, which you can immediately download to your board. Your program has the following qualities:

   a. It starts in ROM (code memory) at 0H (use **-p***addr* to change it, see page 124)

   b. External RAM is assumed to start at 0H (use **-v***addr* to change it, see page 125)

   c. All contiguous RAM bytes will be cleared from 0 to E000H (use **-u***addr* to change it, see page 125)

   d. Set the baud rate by pressing the space bar when the program starts (use **-b***rate* to change it, see page 128)

e. Your program assumes you have an 8051/31 (use **-2**, **-5**, **-t***cpu* to change it, see page 126)

f. If an error occurs, the BASIC source code line number will be displayed (use **-l** to change it, see page 124)

g. Only arithmetic errors can be trapped (use **-e** to change it, see page 124)

h. When the program is finished, it will loop indefinitely (use **-x***addr* to change it, see page 128)

i. Your program is a complete, standalone assembly program (use **-2i** or **-sub** to change it, see page 126)

Notation Used in this Book

Different typefaces are used in this book to improve understanding.

`Typewriter`

Typewriter typeface displays text that is either seen or typed on your PC.

K E Y S

Keys that you press on your PC keyboard appear as key outlines, such as Ctrl C.

Syntax Descriptions

When describing command syntax, square brackets, [and], denote optional arguments. The vertical bar, |, denotes multiple options available as a command line parameter, such as ON | OFF. Text in *italics* represents text that you must provide, such as a hexadecimal address when you see *addr*. **Boldface** text specifies text you must literally type since it is required in the command syntax, such as the command name.

2. Basic Language Elements

Basic Symbols

The BASIC language is composed of a number of basic symbols which can be letters, digits, or special symbols. Letters may be from A to Z, including underscore, '_'. The letters may be in upper, lower, or mixed case. Case only matters inside "quotes". Digits may be 0 or 1 for binary numbers, 0 through 9 for decimal numbers, and 0 through 9 and A through F for hexadecimal numbers (upper case not required). Hexadecimal numbers must begin with a decimal digit, such as 0, and end with H, e.g. 0F00H. Binary numbers must end with B, e.g. 10100101B. Special symbols used by BASIC are: + - * / = < > ( ) { } . , : ; " @ # % $

Comments begin with either the keyword REM or a semi-colon, ';'.

Reserved Words

There are a number of reserved words which are used for BASIC commands, functions, and special variables. Do not use these reserved words for BASIC variables in your program.

```
ABS       DIM       LET       PWM       TAN
AND       DO        LOG       RAMORG    TCON
ASC       ELSE      MCON      RCAP2     THEN
ATN       END       MID$      READ      TIME
BAUD      ERRLINE   MTOP      REM       TIMER0
CALL      ERRVALUE  NEXT      RESTORE   TIMER1
CBY       EXP       NOT       RETI      TIMER2
CHR       FALSE     NULL      RETURN    TMOD
CHR$      FOR       ON        RIGHT$    TRACE0
CLEAR     FREE      ONERR     RND       TRACE1
CLEARI    GET       ONEX1     ROMORG    TRUE
CLEARS    GOSUB     ONTIME    SBUFFER   UI0
CLOCK0    GOTO      OR        SGN       UI1
CLOCK1    HIGH      PCON      SHL       UNTIL
COS       IDLE      PGM       SHR       UO0
CR        IE        PI        SIN       UO1
CTRLC0    IF        POP       SPC       USING
CTRLC1    INPUT     PORT0     SQR       VAL
DATA      INT       PORT1     STOP      WHILE
DBY       IP        PORT2     STR$      XBY
DEFASM    LEFT$     PORT3     STRING    XOR
DEFCTRL   LEN       PRINT     T2CON     XTAL
DEFVAR    LEN       PUSH      TAB
```

If you are using a BXL to extend the BASIC language, all the commands and functions defined in the BXL are additional reserved words. Similarly, when programming for 8051 derivatives, there may be additional reserved words. See BXL or derivative microcontroller documentation for list.

Program Lines

Every BASIC program is composed of a series of lines. Each line contains at least one complete statement. Multiple statements are separated by colons, ':'. The line may optionally begin with a line number or line label.

Using line numbers is the traditional form of referencing lines. Each line begins with its reference number at the left before the BASIC statements. When using line numbers before each line, the numbers must ascend from low numbers at the beginning of the program to high numbers at the end. Commands such as GOTO or GOSUB may be followed by the line number to redirect program control to the specified numbered line.

Instead of using a line number, you may use a line label. Line labels are always in braces (e.g., {LABEL}). They can be used anywhere that a line number is used such as at the beginning of a line or following a GOSUB or GOTO command. Unlike line numbers, line labels do not need to be in any order. However, like line numbers, each must be unique.

With the advent of BXC-51 V4.0, line numbers and labels are optional. If you do not need a line number or label at the start of a line, you may omit it. The following is a valid BXC-51 program:

```
REM this is the beginning
PRINT "Begin"
A=0
{AGAIN} PRINT "Testing..."
10 TRACE1
20 A=A+1
30 PRINT A
40 TRACE0
If A < 5 THEN {AGAIN}
PRINT "Done"
END
```

3. Data Types

There are six basic types of variables allowed in BXC-51: floating point variables, integer variables, byte variables, bit variables, static strings, and dynamic strings. Each of these types, except bit variables, can be used as an array. Each of these types, except bit and static string variables, may be memory mapped to a specific location in memory. Bit variables are not unique variables; they are a variation of integer and byte variables. Variable names for the different types and for arrays are all unique to their type, so I, I%, I#, I$, I(), I%(), I#(), and I$() represent eight different variables. However, special function register variables can be represented as either a floating point, integer, or byte and are not unique, e.g. PORT1 and PORT1#, TIMER0 and TIMER0%, etc.

Floating Point

Floating point numbers have a decimal place and range from ±1 E -127 to ±.99999999 E 127 (as well as 0). Eight digits, an exponent, and a sign represent a floating point value. Each floating point variable occupies 6 bytes of external RAM to store its value. Intel's BASIC-52 uses floating point variables exclusively, while BXC-51 Version 5.0 uses floating point in addition to the other types described below.

A valid floating point variable name is any sequence of letters or digits starting with a letter. An underscore, '_', is considered a letter. There is no limit to the number of letters a variable name may contain. A floating point array variable may be dimensioned with the DIM statement. Below are some example floating point variable names:

NO_POINTS	NP
X	T4
SAMPLE3	X34T

Integer

An integer is a number with no decimal place ranging from -32768 to 0 to +32767 (or 8000H to 0 to 7FFFH). An integer variable occupies 2 bytes of external RAM to store its value. Each integer may be converted to an unsigned (positive) integer by adding the value 65536 to it and storing the result in a floating point variable.

Overflow and underflow errors are not detected in integer arithmetic. All parts of an integer expression must remain in the integer range or the result is invalid. For example, if I%=200, then the result of 200*I%/50 will not be 800 as expected, but instead be -510 because 200*I% overflowed past 32767 into the negative number range.

A valid integer variable name is any sequence of letters or digits that starts with a letter and ends with a percent symbol, '%'. An underscore character, '_', is considered a letter. There is no limit to the number of letters an integer variable name may contain. The following are some example integer variable names:

NO_POINTS%	NP%
X%	T4%
SAMPLE3%	X34T%

Note, however, that some reserved integer variables (RCAP2%, TIMER0%, TIMER1%, TIMER2%) are not located in external RAM, but internal RAM. They are exceptions.

Byte

A byte is a number that has no decimal place and ranges from 0 to 255. In arithmetic expressions, a byte may be used anywhere an integer is used. A byte variable occupies up 1 byte of internal RAM to store its value. There is a limited amount of space for byte variables; 10 bytes on the 8031/51 or DS5000 and 51 bytes available on the 8032/52. Byte variables must be dimensioned with a constant rather than a run-time expression like other arrays. This is so the byte variable space can be allocated correctly before your program runs.

A valid byte variable name is any sequence of letters or digits that starts with a letter and ends with a sharp symbol, '#'. An underscore character, '_', is considered a letter. There is no limit to the number of letters a byte variable name may contain. Below are some example byte variable names:

NO_POINTS#	NP#
X#	T4#
SAMPLE3#	X34T#

Bits

A bit is a number with the value 0 or 1. In arithmetic expressions, a bit may be used anywhere a byte or integer is used.

A bit variable is not a unique variable containing only one bit; it is a bit of a byte or integer variable. To address one bit of a byte or integer variable, specify the variable's name and follow it with a dot, '.', and a number to represent the bit number. There are 8 bits in a byte, numbered 0 to 7. There are 16 bits in an integer, numbered 0 to 15. Bit 0 is the least significant bit (LSB). Bit variables may be used in expressions and assigned a value. The following are some example bit variable uses:

NO_POINTS#.3	NP%.10
X%.4	T4#.7
SAMPLE3%.15	DBY(34).0
PT#(3).4	IVT%(I%+3*J%).12

Note to Assembly programmers: Bit variables are *not* limited to bit addressable internal RAM locations. Any internal RAM or external integer may be a bit variable in BASIC.

Static String

A static string is a string of fixed length. Static strings are allocated by the STRING command which dictates the length of each string and the number of static strings. They cannot be used until after the STRING command. Static strings are accessed in an array manner.

Static string variable names begin with the dollar sign, '$', followed by an array subscript, from 0 to the highest string number. Below are some example static string variable names:

$(0)	$(5)
$(3)	$(10)
$(K)	$(I+3*J)

The MCS BASIC-52 interpreter supports static strings. It does not support dynamic strings.

Dynamic String

A dynamic string is very different from a static string. For dynamic string variables, the STRING command does *not* need to be executed and string lengths may vary between 0 and 255 bytes as needed. Each variable length string occupies 3 bytes of external RAM in addition to its value. The text of the string is stored below MTOP.

A valid dynamic string variable name is any sequence of letters or digits starting with a letter and ending with a dollar sign. An underscore character, '_' is considered a letter. There is no limit to the number of letters a string variable name may contain. Below are some example string variable names:

SITE_NAME$	TYPE$
X$	T4$
SAMPLE3$	X34T$

When using dynamic strings, the compiler sets aside the area from 200H to 2FFH (or location relative to **-v***addr* command line option) for a string buffer. This string buffer will corrupt any BASIC program in the interpreter. This is of particular concern when using the **-2i** command line option.

Dynamic strings take up an undetermined amount of external RAM, based upon their using in the BASIC program. As such, they tend to get re-arranged in memory to make best use of available RAM. This is called garbage collection. Programs that change dynamic variable values frequently may experience delays when garbage collection is performed.

Arrays

An array packs a data type into a series of variables accessible by index. Arrays may be of floating point, integer, byte, and dynamic string data types. The DIM command (see page 21) is used to dimension how many indices are needed for an array.

Array variables follow the same naming convention of their respective types, but are additionally followed by parenthesis with the index. The index may be a specific number or an expression which evaluates to the desired index. Below are some example array variable names for the different data types:

MOTOR(3)	TABLE2(I+3*J)
COUNT%(3)	IRTIME%(I%+3*J%)
PBS#(3)	AJE#(5)
PT$(3)	IVT$(I+3*J)

Memory Mapped Variables

By default, BXC-51 determines where in memory your variables are stored, typically just above 200H. However, you can override the compiler by using the DEFVAR command (see page 20) to specify where the variable should be. Floating point, integer, and dynamic string variables may be mapped to external RAM locations. Byte variables may be mapped to internal RAM locations (both internal RAM and control registers).

Arrays may also be memory mapped. Normal BASIC array variables occupy 3 bytes of external RAM which contain the number of indices and the address in memory where the array of numbers begin. Memory mapped array variables, on the other hand, do not require those 3 bytes. Memory mapped arrays refer to the actual memory address where the array of numbers begin. No index checking is performed on memory mapped array variables because they were not DIMensioned.

For example, to memory map REG1% to memory location 6000H and 6001H, use the statement

```
    DEFVAR REG1%@6000H
```

To refer to a series of integers beginning at 7434H, use the statement

```
    DEFVAR COMMON%()@7434H
```

To map a byte variable to a control register, use the statement

```
    DEFVAR ADCON#@0C5H
```

Memory mapped variables and arrays are used in expressions just like non-memory mapped variables.

4. Expressions

Expressions are formulas for calculating results. Expressions may appear in variable assignments, as parameters to BASIC commands, as parameters to BASIC functions, or as indices to variable arrays. Expressions consist of combinations of operands and operators. Operands are variables, constants, or functions. Operators combine operands to produce another operand. Operators have different orders of precedence which dictates which operator is used before which.

Expressions may contain parenthesis to explicitly specify which operators are used before others. There is no arbitrary limit on the number of parenthesis allowed.

BXC-51 has three basic types of expressions: floating point, integer, and string. Floating point expressions may contain floating point, integer, byte, and bit variables. Integer expressions may only contain integer, byte, and bit variables. String expressions may only contain string variables.

Operators

Both floating point and integer expressions use the following operators. Dynamic string expressions may not use the subtraction, multiplication, division, logical, or bit-shift operators.

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation (not allowed for integer expressions)
`.AND.`	logical AND
`.OR.`	logical OR
`.XOR.`	logical XOR
`.SHL.`	bit-shift left
`.SHR.`	bit-shift right
`=`	test for equality
`<`	test for less than
`>`	test for greater than
`<=`	test for less than or equal to
`>=`	test for greater than or equal to
`<>`	test for inequality

Operators fall into one of six priority levels when evaluated in an expression. The order of precedence is (where 1 is the highest):

1. Unary minus (negative sign before an operand)
2. Exponentiation: **
3. Multiplication and division: * /
4. Addition and subtraction: + -
5. Shift left and shift right: .SHL. .SHR.
6. Relational operators: < > <= >= = <>

7. Logical operators: .AND. .OR.

Unary Minus
A unary minus immediately precedes the operand it negates. In the case of parenthesis operand, the whole expression in the parenthesis is evaluated *before* the result is negated. Unary minus may be applied to any floating point or integer expression operand.

Exponentiation Operator
The exponentiation operator, **, may only be used in floating point expressions.

Multiplying Operators
The multiplication and division operators, * and /, may be used in floating point and integer expressions.

Adding Operators
The addition operator, +, may be used in any expression. The subtraction operator, -, may be used in floating point and integer expressions.

Bit-Shift Operators
The bit-shift operators, .SHL. and .SHR., may be used in floating point and integer expressions. In floating point expressions, the number to be bit shifted is first converted to an integer, then bit shifted, and then converted back to a floating point number, thus dropping any decimal component.

Relational Operators
The relational operators may be used in any expression.

Logical Operators
The logical operators, .AND., .OR., and .XOR., may be used in floating point and integer expressions. In floating point expressions, the numbers are first converted to an integer, then combined, and then converted back to a floating point number, thus dropping any decimal component.

Functions

BASIC allows a variety of functions to be used in expressions. Until you are familiar with all the functions, BASIC functions may look like array variables to you. When used in an expression, BASIC functions are followed by parenthesis that contain parameters for the functions.

Function names beginning with FN are user-defined functions (see the DEFFN on page 19) and may contain multiple parameters as well.

Functions may return floating point, integer, or dynamic string values. See the function documentation on page 44 for more information.

5. BASIC Commands

A BASIC statement is composed of a BASIC command and its parameters. BXC-51 can compile all MCS BASIC-52 program statements. Interpreter run-time commands such as LIST, RUN, RAM, ROM, etc. are inappropriate within a compiled program. The following is a list of these appropriate BASIC commands followed by an explanation of each. Check marked commands are available only in BXC-51, not in the interpreter.

	BAUD	Set printer port baud rate
	CALL	Call assembly routine by address
	CLEAR	Clear all variables, arrays, and interrupts
	CLEARI	Clear all interrupts
	CLEARS	Clear stack space
	CLOCK0	Turn real-time clock off
	CLOCK1	Turn real-time clock on
✓	CTRLC0	Disable control-C usage
✓	CTRLC1	Enable control-C usage
✓	DEFASM	Declare an assembly routine as a BASIC keyword
✓	DEFCTRL	Declare a control register as a byte variable
✓	DEFFN	Declare a user-defined function
✓	DEFVAR	Declare a variable at a specific memory address
	DIM	Dimension an array
✓	DISABLE *intr*	Disable derivative microcontroller interrupt
	DO...UNTIL	Loop until a certain condition arises
	DO...WHILE	Loop while a condition is true
✓	ENABLE *intr*	Enable derivative microcontroller interrupt
	END	Halt program execution normally
	FOR...NEXT	Loop with an index variable a finite number of times
	GOSUB...RETURN	Call a BASIC subroutine
	GOTO	Jump to another line of BASIC
	IDLE	Wait for an interrupt to occur
	IF...THEN...ELSE	Conditionally execute a statement
	INPUT	Input information from user
	LD@	Push a floating point value on the stack from memory
	LET	Assign an expression to a variable
	NULL	Configure NULs to be sent after carriage return
	ONERR	If a program error occurs, GOTO a BASIC subroutine
	ONEX1	If external interrupt 1 occurs, GOSUB a BASIC subroutine
	ON GOTO	On an index, GOTO a BASIC line
	ON GOSUB	On an index, GOSUB a BASIC line
	ONTIME	If a timer interrupt occurs, GOSUB a BASIC subroutine
✓	ON*intr*	On a derivative microcontroller interrupt, GOSUB a subroutine
	PGM	Program an EPROM
	PH0.	PRINT, outputting numbers in hexadecimal
	PH0.@	PH0 . to a user defined output driver

	PH0.#	PH0 . to the list device
	PH1.	PRINT, outputting numbers in hexadecimal with leading zeros
	PH1.@	PH1. to a user defined output driver
	PH1.#	PH1. to the list device
	POP	Pop value(s) off the top of the floating point argument stack
	PRINT	Output text, numbers, and strings to console device
	PRINT@	PRINT to a user defined output driver
	PRINT#	PRINT to the list device
	PUSH	Push a value on the floating point argument stack
	PWM	Pulse width modulation - sound generator
	READ...DATA	Read a value from a DATA statement with expression(s)
	REM or ;	A comment
	RESTORE	Mark all DATA as unread
	RETI	RETURN from ONTIME or ONEX1
✓	SBUFFER *size*	Specifiy serial buffer size
✓	SBUFFER OFF	Disable serial buffering
✓	SBUFFER ON	Enable serial buffering
✓	SBUFFER NOECHO	Disable user keystroke echo
✓	SBUFFER ECHO	Enable user keystroke echo
	ST@	Pop a value off floating point argument stack to memory
	STOP	Abort program execution with a message
	STRING	Allocate string storage space
✓	TRACE0	Turn off line number tracing
✓	TRACE1	Turn on line number tracing
	UI0	Turn off user defined console input routines
	UI1	Turn on user defined console input routines
	UO0	Turn off user defined console output routines
	UO1	Turn on user defined console output routines

These commands take different data type expressions as input. The following explains the syntax.

addr	A RAM address, whether a constant or Assembly variable
bit_addr	A bit address, from 0 to 7 for bytes, 0 to 15 for integers
byte_var	A byte variable name, including trailing #
expr	A floating point, integer, or byte expression
int_expr	An integer or byte expression
line	A line number or line label
name	A name used to identify a variable or function
string	A static or dynamic string variable or quoted string
string_var	A static string variable name, e.g. $(0)
text	Any free form text
TF_expr	A floating point, integer, or byte variable which evaluates to True or False
var	A variable name, including % or # (if integer or byte)

BAUD *expr*

Set the baud rate for the printer serial port to the value of *expr*. Use the PRINT# statement to output to the printer (also known as the list device). The printer port is only available on the 8052/32 at Port 1, bit 7; this statement is ignored on the 8051/31 and DS5000. PRINT# statements on the 8051/31 and DS5000 CPUs are redirected to the console serial port at the baud rate established by the **-b***addr* command line option.

If the BAUD statement is not executed before the first PRINT# statement, a very low baud rate will be assumed. This statement uses the crystal for timing. If the crystal value is different than the default (see XTAL on page 61), the baud will not be accurate.

Example:
```
     10 PRINT "This text is output to the serial port"
     20 BAUD 1200
     30 PRINT
     40 PRINT# "This text is output to the line printer"
     50 PRINT# "And more text to the line printer"
     60 PRINT "And lastly, this text goes to the serial port"
```

CALL *addr*

Use this routine to directly call an assembly routine at the address specified by *addr*. The Assembly routine must follow the guidelines set forth on page 97 in section "Library Routines". The Assembly routine must end with a RET instruction to return control to the BASIC program.

If *addr* is smaller than 80H, *addr* will be read as shorthand to call routines located above 4100H. To calculate the full address, multiply *addr* by 2 and add to 4100H. Hence, CALL 0 would be shorthand for CALL 4100H, CALL 1 for CALL 4102H, CALL 2 for CALL 4104H, etc. This affords you the convenience of having an assembly jump table starting at 4100H.

Example:
```
     10 PRINT "This program assumes there are assembly routines"
     20 PRINT "at 6000H, 6100H, and 6200H."
     30 CALL 6000H  ; call the first routine
     40 DBY(24)= 10: CALL 6100H  ; pass 10 in R0B3 to routine
     50 PRINT "Result: ", DBY(25)
     60 CALL 6200H  ; call last routine
```

CLEAR

Clear all variables, arrays, and interrupts. The entire variable space is reinitialized to 0H, but the static string space is left intact. All interrupts are disabled; ONTIME, ONEX1, ONERR, and ON*intr* are disabled. CLEAR does not affect the TIME variable. It does not imply a CLOCK0 statement or any DISABLE command.

Example:
```
     10 PRINT "Memory Clear test"
     20 a=5
     30 PRINT "Start:   A=", A, "  $(1)= ??"
     40 STRING 100,10
     50 $(1)= "Hello"
```

```
60 PRINT "Before:  A=", A, "  $(1)=", $(1)
70 a= 7
80 CLEAR
90 PRINT "After:   A=", A, "  $(1)=", $(1)
; the value of A changes from 5 to 0 to 0 (again)
; the value of $(1) changes from undefined to 'Hello' to ''
```

CLEARI
Clear all interrupts set by ONTIME, ONEX1, ONERR, and ON*intr* commands. CLEARI does not affect the TIME variable. It does not imply a CLOCK0 statement or DISABLE command.

Example:
```
10 ONEX1 100  ; setup to catch external interrupts
20 TIME=0: CLOCK1
30 DO: GOSUB 3000: UNTIL TIME>3
40 CLEARI     ; no more external interrupts
```

CLEARS
Clear the Control, Argument, and internal stacks. The Control Stack is used to keep track of all FOR loops, DO loops, and GOSUBs. The Argument Stack is used by floating point calculations, the PUSH and POP statements, and the LD@ and ST@ statements. The internal stack is reset to the value in DBY(3EH) which is important only to assembly language programmers who may want to change the internal stack space. The internal stack default value is 4EH. If no byte variables are used, this value may be changed to as high as D0H in an 8052/32 (or 60H in an 8051/31, DS5000, or derivative microcontroller) without encountering problems (although lower values are recomended for programs using complex expressions).

Example:
```
10 FOR I=1 TO 10
20 PUSH I*I   ; load the Argument Stack
30 NEXT
40 GOSUB 3000 ; process some of the Argument Stack
50 CLEARS     ; clear all stacks
```

CLOCK0
Turn off the real-time clock that was turned on by the CLOCK1 statement. The TIME variable will no longer be updated until the next CLOCK1 statement. Note that on the 8051/31, DS5000, and derivative microcontrollers, the PGM or PWM statements will interfere with the timing of the real-time clock.

Example:
```
10 TIME=0: CLOCK1
20 FOR I=1 TO 10
30 GOSUB 3000  ; do some processing
40 NEXT
50 CLOCK0
60 mins%= INT(TIME/60)
70 secs%= TIME-mins%*60
80 PRINT "Total time: ",mins%, "minutes, ", secs%, "seconds"
```

CLOCK1
Turn real-time clock on. Interrupts inside the processor will be enabled to update the TIME variable. The TIME variable will be periodically updated until a CLOCK0 statement is executed. The CLEAR and CLEARI statements do not perform CLOCK0. Note that on the 8051/31 or DS5000 CPUs, use of the PGM or PWM statements will interfere with the timing of the real-time clock.

Example:
    (see CLOCK0 example above)

CTRLC0
Disable Ctrl-C usage. Normally, a user may press Ctrl-C to abort execution while a BXC-51 compiled program is running. After disabling Ctrl-C, the user will not be able to interrupt the program.

Example:
```
     10 CTLRC0  ; do not allow Ctrl-C during initialization
     20 FOR motor%=6000H TO 6040H STEP 10H
     30 GOSUB 3000  ; initialize this motor
     40 NEXT
     50 CTRLC1  ; okay to press Ctrl-C now
```

CTRLC1
Enable Ctrl-C usage. Allow the user to press Ctrl-C to interrupt program execution. When a BXC-51 compiled program starts, the user may press Ctrl-C until the CTRLC0 statement is executed. Use CTRLC1 to allow the user to press Ctrl-C once again.

Example:
    (see CTRLC0 example above)

DATA *expr* [,*expr* [,...]]
A DATA statement contains a series of expressions that can be assigned to variables using the READ statement. Each datum must evaluate to a number. The READ command reads datum in succession starting with the first datum of the first DATA command. When all datum has been read, the next attempt to use READ will cause an OUT OF DATA error unless a RESTORE command is used first.

It is valid to use expressions that refer to variables or use functions. However, strings are not permitted.

DATA statements may appear anywhere in your program. The DATA statements are scanned from the beginning of your program to the end.

Example:
```
     10 PRINT "Testing Read & Data statements"
     11 READ a, b, c : REM A= 5, B= 10, C= 15
     12 PRINT a, b, c
     13 DATA 5,10,15,20 : REM first three ready by line 11
```

```
14 DATA a*2
15 READ a, b : REM A= 20, B= A*2 = 40
16 PRINT a, b
```

DEFASM *name* @ *addr* [, *name* @ *addr* [,...]]

The DEFASM statement allows you to extend the BASIC language by associating a command name with an Assembly routine at *addr* in effect creating your own BASIC command that requires no parameters. To create BASIC commands that have parameters, you will need to create a BXL file; see page 104 for details. *addr* must be an integer constant (in decimal or hexadecimal form) or it must be an Assembly label. This command eliminates the need to use the CALL command every time you want to perform a function from an Assembly routine. Place DEFASM statements at the beginning of your BASIC program before *name* is used. This statement is not executed at run-time and does not generate code; it has special meaning to the compiler to make BASIC and Assembly coding easier.

Once the DEFASM statement has been processed, you may use *name* as if it were a BASIC command with no parameters. To pass parameters to assembly routines, (1) use the PUSH or LD@ command to put values on the argument stack, (2) store values in internal memory between 24 and 31 for the assembly routine to read, (3) pass the parameters via specific memory locations defined by DEFVAR, or (4) extend the BASIC language using a BXL (see page 104).

Example:

```
10 DEFASM CHECK @ 5000H, MOVEIT @ 5059H, SHUTDOWN @ MX13
100 FOR I%= 1 TO 20 : CHECK
110 IF DBY(1FH) = 15 THEN MOVEIT
120 NEXT I%
130 SHUTDOWN
140 GOTO 150
$ASM
MX13:   MOV     DPTR,#06600H
        MOV     A,#0
        MOVX    @DPTR,A
        INC     DPTR
        INC     DPTR
        MOVX    @DPTR,A
        RET
$BASIC
150 REM
```

DEFFN *name*[$][%][#](*param1* [, *param2* [, ...]])=*expr*

The DEFFN statement allows you to create a user-defined function. The function may return a floating point, integer, byte, or string value. Any parameters specified must be of the same data type as the function. Any number of parameters may be specified. The value for each parameter is substituted in the appropriate places where mentioned in the expression. For example,

```
DEFFN IOREG%(OFFSET%)=DBY(0F020H+OFFSET%)
```

will substitute the function's parameter, OFFSET%, into the DBY() function each time the function user function FNIOREG%() is used. For example,

```
      PRINT FNIOREG%(3)
```

will, in effect, perform DBY(0F020H+3) and print the result.

To undefine a function declare it with no parameters and no expression. For example,

```
      DEFFN IOREG%()=
```

Using the wrong number of parameters to the function causes a PARAMETER MISMATCH error at run-time. Attempting to use a function that has not yet been defined (or that has been undefined) will cause the run-time error NO FN DEF.

Example:
```
      10 DEFFN F(X)=X**2/10: DEFFN R(Y)=SQR(Y*10)
      20 GOSUB 4000
      30 END
      4000 REM Output simple graph
      4010 FOR I=10 TO 1 STEP -1
      4020 ? I, TAB(5), "*", TAB(FN R(I)+5), "o"
      4030 NEXT
      4040 ? "    ***********"
      4050 ? "    0         10"
      4060 RETURN
```

DEFCTRL *byte_var* @ *addr* [, *byte_var* @ *addr* [,...]]
The DEFCTRL statement allows you to declare special function registers (microcontroller control registers) located at 80H and up with the byte variable name *byte_var*. Remember that byte variables end with the # symbol (e.g., TCON#). Place DEFCTRL statements at the beginning of your BASIC program before *byte_var* is used. This statement is not executed at run-time and does not generate code; it has special meaning to the compiler to make special function register in derivative microcontrollers easier to use. When using the **-t***cpu* command line option, a number of registers will automatically be generated. See derivative microcontroller documentation for more information.

Once the DEFCTRL statement is processed, you may use *byte_var* as if it was any other byte variable.

Example:
```
      10 DEFCTRL ADCON#@0C5H, ADAT#@0C6H
      20 ADCON#.3= 1  ; start A/D conversion
      30 DO: WHILE ADCON#.4=0  ; wait for conversion to complete
      40 PRINT "Level: ", ADAT#
```

DEFVAR *var* @ *addr* [, *var* @ *addr* [,...]]
The DEFVAR statement allows you to control where variables are located in memory. Each variable name specified, *var*, will be located at memory address *addr*. Integer and floating point variables and arrays are located in external RAM. Byte variables and arrays are located in internal RAM. Place DEFVAR statements at the beginning of your BASIC program before *var* is used. This statement is not executed at run-time and does not generate output; it has special meaning to

the compiler to allow control over variable memory locations. Once the DEFVAR statement is processed, you may use *var* as if it was a regular variable.

To specify an array, do not specify its size (e.g., DEFVAR A() @ 5000H). Note that the starting address of an array is the starting address of it's first value e.g. A(0). It is inappropriate to DIMension this kind of array.

Example:
```
     10 DEFVAR PARAM1#@24, PARAM2#@25
     20 DEFVAR TABLE%()@6002H, TSIZE%@6000H
     30 PARAM1#= 13: PARAM2#= 12  ; parameters to assembly
     routine
     40 CALL 05F20H  ; collect data for a while
     50 FOR I%= 1 TO TSIZE%
     60 PRINT I%, ")", TABLE%(I%)
     70 NEXT
```

DIM *var(expr1)* [, *var(expr2)* [, ...]]
Dimension an array variable to a specific size. The expression in parenthesis is the maximum size for the array. Once dimensioned, an array can be indexed from 0 to the maximum size. Arrays are limited to one dimension. They cannot be larger than 254, and they cannot be redimensioned unless a CLEAR statement is first executed. A CLEAR statement removes the previous array dimension. Arrays that are not dimensioned before they are used are automatically dimensioned to a maximum size of 10 and cannot be redimensioned.

An attempt to redimension an array will cause an ARRAY SIZE error. Floating point, integer, and dynamic string arrays may be dimensioned using the result of a calculation (e.g., DIM F(N*2+1)). Byte arrays must be dimensioned with a constant (e.g., DIM F%(10)). Be aware that byte arrays compete with byte variables for internal RAM; there are only 10 bytes available in an 8031/51, DS5000, and derivative microcontrollers; and 51 bytes available in an 8032/52.

Example:
```
     10 DIM TABLE(100)
     20 FOR I= 1 TO 100
     30 TABLE(I)= GET
     40 IF TABLE(I) = 0 THEN 30
     50 IF TABLE(I) = 4 THEN I=100 ; exit loop
     60 NEXT
```

DISABLE *intr*
When using the **-t***cpu* command line option to specify a derivative microcontroller, additional interrupts may be available. Consult additional documentation for that microcontroller to determine what interrupts are available and what functions they provide (for example, see 8xC550 documentation on page 88 or 8xC552 documentation on page 89). For each interrupt available, the ON*intr* command handles the interrupt and ENABLE allows those interrupts to occur. Use this command to disable those interrupts.

Example:
```
     10 ONCM1 1000  ; where to go when compare true
```

```
20 CM0%= 0A080H  ; set the compare value
30 ENABLE CM0
40 DO : UNTIL DONE%
50 END
1000 REM Handle compare interrupt
1010 GOSUB 4000 ; do appropriate action
1020 COMP_COUNT%= COMP_COUNT%+1
1030 IF COMP_COUNT%>=100 THEN DISABLE CM0
1040 RETI
```

DO

The DO statement is the first part of two statements that form a loop. The body of the loop contains BASIC statements between DO and WHILE or UNTIL. A loop contains a series of statements that executed a number of times until a certain condition arises. See the WHILE and UNTIL statements. A loop is executed at leaset once. Too many embedded DO loops or other control structures will cause a C-STACK error at run-time.

Example:

```
; wait until external interrupt 1
DO
  COUNT=COUNT+1
UNTIL TCON#.3
```

ENABLE *intr*

When using the **-t***cpu* command line option to specify a derivative microcontroller, additional interrupts may be available. Consult additional documentation for that microcontroller to determine what interrupts are available and what functions they provide (for example, see 8xC550 documentation on page 88 or 8xC552 documentation on page 89). For each interrupt available, the ON*intr* command handles the interrupt. The ENABLE command allows those interrupts to occur.

Example:

```
10 ONTIME 1,2000 ; interrupt at 1 second
20 ONAD 1000  ; where to go when A/D converstion done
30 TIME= 0: CLOCK1
40 DO : UNTIL DONE%
50 END
1000 REM Handle A/D conversion complete interrupt
1010 PRINT "A/D conversion data: ", ADAT#
1020 RETI
2000 REM Handle ONTIME interrupt
2010 ENABLE AD   ; start the A/D conversion
2020 ONTIME TIME+1, 2000  ; setup next conversion time
2030 RETI
```

END

Halt program execution normally and without an error message. If the **-x***addr* command line option is specified, it will direct execution elsewhere. Otherwise the program will begin an infinite loop. When a program ends, all BASIC interrupts are disabled.

Example:
```
     10 PRINT "Hello, World!"
     20 END
     30 PRINT "The text on this line is never output."
```

FOR *var* = *expr1* TO *expr2* [STEP *expr3*]
The FOR statement is the first part of two statements that form a loop. The body of the loop is contained between the FOR and NEXT statements. A FOR statement will execute a finite number of times advancing from the value of *expr1* to *expr2* in increments of 1 unless a STEP value, *expr3*, is given. The STEP value may be negative in which case the loop will decrease from *expr1* to *expr2*. The index variable, *var*, must be a floating point variable, integer variable, or byte variable. Strings and bit variables are not allowed. Integer index variables only range from -32768 to +32767; for greater range than that, use a floating point index variable. Byte index variables range from 0 to 255. Byte variables cannot have a negative STEP value. Too many embedded FOR loops or other control structures will cause a C-STACK error at run-time.

Example:
```
     10 FOR I%=10 TO -10 STEP -1
     20 PRINT I%
     30 NEXT I%
```

GOSUB *line*
The GOSUB statement allows program control to temporarily switch to another part of the program to a BASIC subroutine. Program control will change to the line number or label specified. If *line* does not exist, a compiler error will be reported. The original location is resumed after executing a RETURN. Using the GOSUB statement allows you to execute a series of BASIC statements repeatedly without retyping them; each time you need to execute the series of statements, just perform a GOSUB.

When a GOSUB is in progress, another GOSUB may be used. BASIC keeps track of nested GOSUBs. Too many GOSUBs, however, or other control structures will cause a C-STACK error at run-time.

Example:
```
     5 GOSUB {collect_input}
     10 GOSUB 3000  ; process the input
     20 GOSUB {output}
     30 END
     {collect_input} INPUT "Enter a number", A: RETURN
     3000 A= SQR(A): RETURN
     {output} PRINT "Square root is", A: RETURN
```

GOTO *line*
The GOTO statement switches control from one part of the program to another. Program execution is switched to the line number or label specified.

Example:
```
     10 ? "First": GOTO 100
     {alabel} ? "Third": GOTO {blabel}
```

```
100 ? "Second": GOTO {alabel}
{blabel} ? "Fourth": GOTO 200
200 ? "Done": END
```

IDLE

Wait for an interrupt to occur; wait for either an ONTIME or an ONEX1 interrupt to occur and serviced it. Program execution will be halted until the interrupt occurs. The IDL bit of the PCON register (bit 0) is set so the microcontroller goes into low power mode (if supported). After the interrupt is serviced, program execution continues following the IDLE statement.

For assembly programmers writing interrupt handlers, the IDLE statement can be terminated by setting bit 21H in the interrupt routine. The IDLE statement can be detected because it clears Port 1, bit 6 when it starts and sets it when it when finished. On the DS5000 microcontroller, PCON.0 is set to enable Idle Mode.

Example:

```
10 GOSUB 2000  ; initialize
20 IDLE  ; wait for an interruption
30 GOSUB 3000  ; process/update
40 GOTO 20  ; repeat
```

IF *TF_expr* THEN *statements* [ELSE *statements*]

Single line IF...THEN. If a particular expression *TF_expr* is true, execute the statement(s) following THEN. If not, execute the statement(s) following ELSE if ELSE is specified. If a line number or label follows THEN or ELSE, program execution will switch to the line specified as though a GOTO had been executed. An IF... THEN...ELSE statement is restricted to a single BASIC program line and cannot be spread over several lines the way the DO and FOR loops can.

Example:

```
10 INPUT "Guess my number", A
20 IF A=3 THEN ? "You guessed my number!": END
30 IF A=0 THEN ? "Bye.": END
40 IF A>=2 .AND. A<=4 THEN ? "Getting warmer!"
50 IF A<3 THEN ? "Too Low.": ELSE ? "Too high."
60 ? "I'm thinking of a different number.  Try again."
70 GOTO 10
```

IF *TF_expr* THEN
statements
[ELSE IF *TF_expr* THEN
statements2]
[ELSE
statements3]
ENDIF

Multiple line IF...THEN. When using the multiple line form of the IF...THEN command, the *statements* part of the command may include multiple statements spread over several lines. There are several permulations of this command allowed: IF...THEN...ENDIF, IF...THEN...ELSE...ENDIF, and IF...THEN...ELSEIF...ELSEIF...ENDIF (where the second ELSEIF really means as many ELSEIF's as necessary and ELSE may be substituted, too). The key to making an IF...THEN statement a multiple IF...THEN statement is the use of THEN or ELSE as the last keyword on a line. If a statement follows the THEN or ELSE, the compiler assumes it is a single line IF...THEN, not a multiple line one.

There is no limit to the number of lines in each *statements* block. However, multiple line IF...THEN statments can only be nested 40 levels deep. The ENDIF is required for multiple line IF...THEN statements; the compiler will report errors when they are missing.

Example:
```
10 INPUT "Guess my number", A
20 IF A=3 THEN
21    ? "You guessed my number!"
30 ELSE IF A=0 THEN
31    ? "Bye."
40 ELSE
41    IF A>=2 .AND. A<=4 THEN ? "Getting warmer!"
50    IF A<3 THEN
51       ? "Too Low."
52    ELSE
53       ? "Too high."
54    ENDIF
60    ? "I'm thinking of a different number.  Try again."
70    GOTO 10
80 ENDIF
90 END
```

INPUT ["*text*" [,]] *var1* [, *var2* [, ...]]

Halt execution of the program until the user has entered some data. If any text appears in double quotes, the *prompt_string* message will be displayed before the user is asked for data. If a comma is present while specifying the prompt string, no question mark will appear and the user will be prompted on the same line that the prompt string appears. When no comma is present, a question mark will be placed on the line below the prompt string. The text the user types will be assigned to the variable specified. If the INPUT statement specifies multiple variables, the user must enter data to match the exact amount of variables, separated by commas. If not, a TRY AGAIN message appears and the user will be prompted again. An EXTRA IGNORED message displays if the user specifies too much information. However, the user will not have to reenter the information. The variables may be floating point, integer or byte variables or strings (both static or dynamic). Everything on the rest of the line will be stored in the string if a string is requested. If multiple strings are specified, multiple prompts will appear until each string has a value.

Example:
```
10 STRING 100,20
20 INPUT "Your name?", $(0)
30 INPUT "Your age?", AGE
40 ? "Hello, ",$(0),".  You have seen", AGE/4, "leapyears."
```

LD@ ***expr***

The LD@ statement fetches a floating point number from external memory and pushes it onto the floating point Argument Stack. The address *expr* specifies the last byte of the 6 bytes that are used to contain the floating point number. For example, use the

```
     LD@ 6A05H
```

statement to fetch a floating point number that has been stored beginning at 6A00H. Use the ST@ command to store the value to another memory location or POP to store it into a floating point variable. When the Argument Stack is full, attempting to fetch a number onto it causes an A-STACK error.

Example:

```
     10 FOR I%= 06A00H TO 06A24H STEP 6
     20 LD@ I%+5
     30 NEXT
     40 GOSUB 4000  ; process the 7 numbers on stack
     50 POP ANSWER  ; get the resulting answer
```

LET ***var=expr***

The LET statement allows you to assign a value to a variable. The expression must be the same data type as the variable. Variables may be floating point, integer (with %), byte, (with #), bit (with %. or #.), static string (as $(*n*)), or dynamic string data types. Special assignments can also be made to DBY(), XBY(), and ASC().

1. `LET` *`var = expr`*

Assign the result of an expression to a variable of the same type. A floating point or integer expression can be assigned to a floating point variable. A floating point expression can not be assigned to an integer variable or byte variable, only to integer expressions. An error will not occur if a negative value or positive value larger than 255 is assigned to a byte variable unless the **-g** command line flag was specified. A static string may be assigned to a dynamic string and vice versa, but only dynamic strings may use string expressions to calculate a value; static strings have restrictions (see below).

2. `LET` *`string_var = string`*

string may be double quoted text, static string variable, or a dynamic string variable. String expressions cannot be assigned to a static string variable. Dynamic string variables, on the other hand, can be assigned values from string expressions including static or dynamic string variables, string functions, and quoted text added together.

3. `LET ASC(`*`string_var,expr1`*`) = `*`expr2`*

A character at position *expr1* in a static string can be changed when *expr2* is the ASCII code number of the new character. This does not apply to dynamic strings. For dynamic strings, this must be done using the LEFT$(), CHR$(), and MID$() functions. For example:

```
LET A$=LEFT$(A$,pos-1)+CHR$(expr2)+MID$(A$,pos+1,255)
```

4. `LET DBY(`*`expr1`*`) = `*`expr2`*

Assign a value from a floating point or integer expression, *expr2*, to an internal RAM location. A bit address may be specified, e.g. DBY(expr).3.

5. `LET XBY(`*`expr1`*`) = `*`expr2`*

Assign a value from a floating point or integer expression, *expr2*, to an external RAM location.

6. `LET `*`var.bit_addr`*` = `*`expr2`*

Assign a bit value to a bit in a byte or integer variable. The bit address may range from 0 to 7 for byte variables and from 0 to 15 for integer variables.

The keyword LET is extraneous. The short form of this statement is to drop the 'LET' and only specify the assignment.

Example:
```
LET HALFPI = PI/2
KTABLE(34) = SQRT(HALFPI)
COUNT% = 34+XBY(1204)
LET BIN%(12) = BIN%(12)+1
$(1) = "Var Bind"
SITE$ = "Factor 1209"
EQUIP$(12) = "34-12 Lifter"
ASC($(3),5) = 65
DBY(34H) = 0
XBY(200H) = XBY(0FF00H+I)
```

NEXT [*var*]

The NEXT statement is the second part of two statements that form a FOR loop. The body of the loop is contained between the FOR and NEXT statements. The NEXT statement may optionally specify the variable's name. If the variable name is given, the loop will always make sure your NEXT statement loops back to the correct FOR, reporting a C-STACK error if they do not match. When no variable is specified, NEXT simply loops back to the last executed FOR statement. After the FOR statement has advanced to its final value, program execution will resume after the NEXT statement.

If *var* is specified, make sure it is identical to the index variable in the matching FOR statement. It must have identical spelling, and if it is an integer or byte variable, it must be followed by '%' or '#', as appropriate.

Example:
```
10 FOR I= 1 TO COUNT
20 FOR J= I+1 TO COUNT
30 IF A(J) < A(I) THEN A=A(J): A(J)= A(I): A(I)= A
40 NEXT J
50 NEXT I
```

ON *expr* GOSUB *line1* [, *line2* [, ...]]

The ON...GOSUB statement is used as a multi-branching GOSUB statement. Conditional upon the value of *expr*, the first, second, or third, etc., line number or label will be GOSUBed. If *expr* is 0, a GOSUB *line1* is performed; if *expr* is 1, a GOSUB *line2* is performed; etc. When the respective RETURN statement is executed, the program will continue execution after the end of the list of line numbers.

Example:
```
     10 PRINT "1. Collect data"
     20 PRINT "2. Process data"
     30 PRINT "3. Output data"
     40 PRINT "0. Quit"
     50 INPUT "Selection? ", A
     60 ON A GOSUB 100, 200, 300, 400: GOTO 50
     100 END: REM Quit
     200 REM Collect data
     210 RETURN
     300 REM Process data
     310 RETURN
     400 REM Output data
     410 RETURN
```

ON *expr* GOTO *line1* [, *line2* [, ...]]

The ON...GOTO statement is used as a multi-branching GOTO statement. Conditional upon the value of *expr*, the first, second, third, etc. line number or label will be GOTOed. If *expr* is 0, then a GOTO *line1* is performed; if *expr* is 1, then a GOTO *line2* is performed; etc.

Example:
```
     10 PRINT "1. Instructions"
     20 PRINT "2. Proceed with operation"
     30 PRINT "3. Abort operation now"
     40 INPUT "Selection? ", A
     50 ON A GOSUB 40, 100, 200, 300: GOTO 40
     100 REM Instructions
     200 REM Proceed with operation
     300 REM Abort operation now
```

NULL *expr*

Specify the number of NUL characters to output after each carriage return. Older printers which do not have line buffers may require a series of NUL characters to be output after each carriage return (CR) that is printed to the serial port. This command configures the number of NUL characters to output.

Example:
```
     10 NULL 8
     20 PRINT "Hello, World!"
```

ONERR *line*

Specify the program line to switch to in the event of an arithmetic error. Should an error occur, a GOTO will be induced to *line*. The error number will be stored in the variable ERRVALUE%

and the line number where the error occurred will be stored in the variable ERRLINE% (see page 52). The CLEAR and CLEARI statements will remove the ONERR statement's error trapping ability.

If the **-e** command line option was specified to trap all errors (not just arithmetic errors), then any error will cause a GOTO to *line*. If you have an error in your error handling routine, your program will loop forever.

Example:
```
     10 ONERR 1000
     20 A= 20
     30 B= 0
     40 C= A/B: REM this causes /0 error
     50 PRINT "Answer: ",C
     60 END
     1000 PRINT "Error", ERRVALUE%, "in line", ERRLINE%
     1010 REM take corrective action
```

ONEX1 *line*
Specify the subroutine to execute when external interrupt 1 is detected. Should an external interrupt 1 occur, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of the usual RETURN. When an external interrupt 1 occurs, the normal program flow is temporarily suspended while the interrupt is being processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONEX1 statement's interrupt processing ability until the next ONEX1 statement is encountered. Without the ONEX1 statement, BXC-51 generates code that ignores external interrupt 1 (except to vector it to 4013H or wherever the **-c***addr* command line option specifies).

Note that the EX1 interrupt is polled which causes latentcy between when the interrupt occurs and when the BASIC subroutine services it. This may be a concern for applications that need immediate interrupt service.

Example:
```
     10 ONEX1 1000
     20 DONE = FALSE
     30 DO
     40 GOSUB 4000  ; collect information
     50 GOSUB 5000  ; process information
     60 UNTIL DONE
     70 END
     1000 REM EX1 interrupt
     1010 DONE = TRUE
     1020 GOSUB 6000 ; output information
     1030 RETURN
```

ONTIME *expr, line*
Specify the subroutine to be executed when the TIME variable is greater than or equal to *expr*. For this comparison, only the seconds are compared rather than milliseconds or smaller. If the TIME variable becomes greater than or equal to *expr*, a GOSUB will be induced to *line*. To

return from the TIME subroutine, use a RETI instruction instead of a RETURN. When a TIME interrupt occurs, program flow is only temporarily interrupted. The ONTIME interrupt is cleared. Therefore, if you want another TIME interrupt, you must execute another ONTIME statement for the next timer interval. No TIME interrupts will occur unless a CLOCK1 has been executed to enable the real-time clock which updates the TIME variable. Note that on the 8051/31, DS5000, and derivative microcontrollers, the PGM and PWM statements will interfere with the timing of the real-time clock.

Note that the timer interrupt is polled which causes latentcy between when the time matches and when the BASIC subroutine services it. This may be a concern for applications that need immediate precise timing.

Example:
```
10 TIME=0  ; reset time
20 CLOCK1 ; enable clock
30 ONTIME 1,1000  ; interrupt at 1 second
40 DO
50 WHILE TIME<9: END  ; stop at 9 seconds
1000 REM ONTIME Interrupt
1010 a= TIME
1020 PRINT "Timer interrupt at -",a,"seconds"
1030 ONTIME a+2,100  ; interrupt 2 seconds from now
1040 RETI
```

ON*intr line*

When using the **-t***cpu* command line option to specify a derivative microcontroller, additional interrupts may be available. Consult additional documentation for that microcontroller to determine what interrupts are available and what functions they provide (for example, see 8xC550 documentation on page 88 or 8xC552 documentation on page 89). For each interrupt available, an ON*intr* command may be used, where *intr* is the name of the interrupt.

The ON*intr* command specifies the subroutine to execute when the interrupt occurs. When the interrupt occurs, a GOSUB will be induced to *line*. To return from the subroutine, use a RETI instruction instead of a RETURN. When the interrupt occurs, program flow is only temporarily interrupted. No interrupts will occur unless an ENABLE statement has been executed to enable *intr*. Use the DISABLE statement to disable the interrupt.

Example:
```
10 ONAD 1000  ; when A/D conversion completes
20 ENABLE AD  ; begin A/D conversion
30 AD_READY%= FALSE
40 DO : UNTIL AD_READY%
50 PRINT "A/D data:", ADAT#
60 END
1000 AD_READY%= TRUE
1010 RETI
```

PGM

The PGM statement programs a block of RAM into an EPROM. The RAM starting address, EPROM starting address, number of bytes, and programming pulse must be stored in internal RAM (see diagram below). The RAM starting address must be put in DBY(1BH) (high byte) and DBY(19H) (low byte). The EPROM starting address must be put in DBY(1AH) (high byte) and DBY(18H) (low byte). The number of bytes to be programmed must be put in DBY(1FH) (high byte) and DBY(1EH) (low byte). The width of the EPROM programming pulse must be put in DBY(41H) (high byte) and DBY(40H) (low byte). Calculate the width value by using the formula $65536 - \frac{(width \times XTAL)}{12}$ where *width* is the EPROM pulse width in seconds. For example, use a width of 0.001 for 1 millisecond. Finally, bit 26H.3 must be set to select the INTELligent programming and cleared to select the normal 1 millisecond algorithm. Setting the bit can be performed by

```
DBY(26H).3 = 1
```

or cleared by performing

```
DBY(26H).3 = 0
```

If no errors occur, DBY(1FH) and DBY(1EH) will be zero. If the values are non-zero, the operation was unsuccessful. The PGM statement can only be used when run from *internal* ROM, such as the 8751, 8752, mask programmed, or Dallas Semiconductor's 'DS' series microcontrollers. Note that on the 8051 (2 counter/timer) microcontrollers, use of the PGM or PWM statements will interfere with the timing of the real-time clock.

Purpose of value	Location (high)	Location (low)
Start of block in RAM (source)	DBY(1BH)	DBY(19H)
Number of bytes in block	DBY(1FH)	DBY(1EH)
Start of block in EPROM (target)	DBY(1AH)	DBY(18H)
Programming pulse	DBY(41H)	DBY(40H)

Example:

```
     10 ADDRRAM=1060H: BLOCK_LEN= 400H
     20 ADDRROM= 8000H: PULSE= .075 ; 75 milliseconds
     30 GOSUB 5000 ; program it
     40 END
     5000 RELOAD= 65536 - PULSE * XTAL/12
     5010 DBY(41H)= HIGH(RELOAD): DBY(40H)= LOW(RELOAD)
     5020 DBY(1BH)= HIGH(ADDRRAM): DBY(19H)= LOW(ADDRRAM)
     5030 DBY(1FH)= HIGH(BLOCK_LEN): DBY(1EH)= LOW(BLOCK_LEN)
     5040 DBY(1AH)= HIGH(ADDRROM): DBY(18H)= LOW(ADDROM)
     5050 PGM
     5060 IF DBY(1FH) <> 0 .OR. DBY(1EH) <> 0 THEN
     5070   PRINT "Range Program Failed."
     5080 ENDIF
     5090 RETURN
```

PH0. ***print_items***

The PH0. command is identical to the PRINT command in every respect except for how it outputs its numbers. The PH0. command displays numbers smaller than 100H in hexadecimal without leading zeros. The character 'H' will follow the number to indicate that the number is hexadecimal.

Example:

```
     10 ONTIME 1,100
     20 DEFVAR OTADDR%@126H
     30 PH0. "Interrupt routine @", OTADDR%
```

PH0.@ ***print_items***

The PH0.@ command is the same as the PH0. command except that it outputs the value to a user defined output driver (at 403CH or wherever the **-c***addr* command line option specifies). To enable PH0.@ to use the user defined output driver, DBY(24H).7 must be set.

PH0.# ***print_items***

The PH0.# command is identical to the PH0. command except that it outputs the value to the list device (see the BAUD command on page 16). On the 8051/31, DS5000, and derivative microcontrollers, this command does not write to the list device, instead it uses the serial port.

PH1. ***print_items***

The PH1. command is the same as the PRINT command in every respect except how it outputs its numbers. The PH1. command always displays its numbers as four hexadecimal digits, with leading zeros as necessary. The character 'H' will be displayed following the number to clearly delineate the number as a hexadecimal number.

Example:

```
     10 INPUT "Enter a floating point number:", A
     20 PUSH A
     30 ST@ 12
     40 PRINT "Hex representation: ",
     50 FOR I%= 7 TO 12
     60 PH1. XBY(I%),
     70 NEXT
     80 PRINT
```

PH1.@ ***print_items***

The PH1.@ command is identical to the PH1. command except that it outputs the value to a user defined output driver (at 403CH or wherever the **-c***addr* command line option specifies). To enable PH1.@ to use the user defined output driver, DBY(24H).7 must be set.

PH1.# ***print_items***

The PH1.# command is identical to the PH1. command except that it outputs the value to the list device (see the BAUD command on page 16). On the 8051/31, this statement does not write to the list device, but uses the serial port instead.

POP *var1* [, *var2* [, ...]]

The POP command is used to remove values from the top of the floating point Argument Stack and store them in the specified variable(s). Use the PUSH or LD@ commands to put values on the floating point Argument Stack. If the Argument Stack is empty, attepting to remove a value from it causes an A-STACK error.

The specified variables may be floating point, integer, or byte variables.

Some programmers use the POP command inside subroutines to receive data PUSHed before the GOSUB was performed, storing the values in the subroutine's special variables.

Example:

```
10 INPUT "Leg lengths in a right triangle: ", L1, L2
20 PUSH L1, L2
30 GOSUB 2000
40 POP H
50 PRINT "Hypotenuse: ", H
60 END
2000 POP A, B
2010 PUSH SQR(A*A+B*B)  ; calculate hypotenuse
2020 RETURN
```

PRINT *print_item1* [, *print_item2* [, ...]] [,]

The PRINT command is a general purpose output statement to the serial port. Follow the PRINT command with a list of print items, each separated by a comma. Normally, a CR-LF sequence follows each printed line. To suppress this, place a comma at the end of the line.

In a PRINT statement, *print_item* can be:

1. A floating point, integer, byte, or bit expression. It will be displayed in a general form unless a USING command has changed the default numeric output format.

2. A double quoted string, string variable or string expression.

3. TAB(*expr*) will print spaces up to the column that *expr* specifies. If the cursor or printhead is already beyond the tab location, the request is ignored.

4. SPC(*expr*) will print the specified number of spaces.

5. CHR(*expr*) will output the ASCII character for the code *expr*. CHR(*string*, *expr*) will output a specific character at position *expr* from inside a static string variable.

6. CR will output a single CR without an LF. Follow with a comma to suppress another CR-LF.

7. USING(*codes*) specifies a new format for floating point numbers to use until the next USING statement is executed. The *codes* for the USING statement come in two varieties; fraction and combined specifications. For the fraction specification, use 'F' followed by the number of digits in the fraction that should be displayed. For

example, USING(F3) will output all numbers to 3 decimal places with trailing zeros if necessary. The number of digits in the fraction can be from 3 to 8 (1 and 2 are promoted to 3).

If USING(0) is specified, the number is output in a general form which is the default when your program first starts.

The combined fraction and integer specification allows you to specify the number of integer digits to display and the number of fraction digits to display. Do this by using pound signs, '#', separated by a decimal point. For example, USING(###.##) would output numbers to two decimal places, allowing up to three digits for the integer part. If you do not want to display any fraction part, the decimal is not required. A maximum of eight pound signs are allowed (the compiler will complain if you try to use more).

Example:
```
     10 INPUT "Any number: "A
     20 ? "You typed:", A
     30 IF A < 0 THEN I$="i": A= ABS(A)
     40 PRINT "Square root of it is", SQR(A), I$
     50 PRINT "Square root to 2 decimal places is", USING(#.##),
     SQR(A), I$
     60 PRINT TAB(13), "3 decimal places is", USING(#.###),
     SQR(A), I$
     70 PRINT SPC(13), "4 decimal places is", USING(#.####),
     SQR(A), I$

     10 PRINT "Example cursor flips:"
     20 DEF FN EVEN%(N%)=NOT(N%/2-N%/2)  ; nifty test for even
     30 DELAY= 100
     40 C(0)= ASC(-)
     50 C(1)= ASC(\)
     60 C(2)= ASC(|)
     70 C(3)= ASC(/)
     80 FOR FLIPS%=1 TO 20  ; 10 rotations
     90 FOR I%= 0 TO 3
     100 PRINT CHR(C(I%)), CR,
     110 FOR X=1 TO DELAY: NEXT  ; pause
     120 NEXT
     130 IF FN EVEN%(FLIPS%) THEN PRINT CHR(7), CR,  ; chirp
     140 NEXT
     150 PRINT
```

PRINT@ *print_items*

The PRINT@ command is identical to the PRINT command except that it outputs the value to a user defined output driver (located at 403CH or wherever the **-c***addr* command line option specifies). To enable PRINT@ to use the user defined output driver, DBY(24H).7 must be set.

Example:
```
     10 buffer%= 8000H  ; location of character buffer
```

```
20 DBY(24H).7= 1   ; enable @ output
30 PRINT@ "Hello, World!" ; store message starting at 8000H
40 END
$ASM
HERE  EQU    $
      ORG    403CH      ; create console output handling
      PUSH   DPL        ; save DPTR
      PUSH   DPH
      MOV    DPTR,#IV_BUFFER
      CALL   LDPTRI     ; DPTR = BUFFER%
      MOVX   @DPTR,A    ; save character
      INC    DPTR       ; update pointer
      MOV    R1,DPL
      MOV    R3,DPH
      MOV    DPTR,#IV_BUFFER
      CALL   IPUTVAR
      POP    DPH    ; restore DPTR
      POP    DPL
      RET
      ORG    HERE
$BASIC
```

PRINT# *print_items*
The PRINT# command is the same as the PRINT command except that it outputs the value to the list device (see the BAUD command on page 16). On the 8051/31, DS5000, and derivative microcontrollers, this statement does not write to the list device; it uses the serial port instead.

Example:
```
10 BAUD 1200
20 PRINT# "Report of collected data"
30 PRINT# "========================"
40 COUNT%=1
50 FOR I%=6000H TO 6FFFH
60 LD@ I%+5
70 POP N
80 PRINT# COUNT%, ")", N
90 COUNT%= COUNT% + 1
100 NEXT
```

PUSH *expr1* [, *expr2* [, ...]]
The PUSH command is used to put values calculated from expressions on the floating point Argument Stack. Use the POP or ST@ command to remove values from the floating point Argument Stack. When the Argument Stack is full, attempting to PUSH a value causes an A-STACK error.

Some programmers use the PUSH command before subroutines to pass data that will be POPed and stored in variables specific to the subroutine.

Example:
```
10 REM Copy data to non-volatile memory
30 memaddr%= 6000H
```

```
20 DO
30 READ n
40 PUSH n
50 ST@ memaddr%+5
60 memaddr%= memaddr%+6
70 UNTIL n=0
80 DATA SIN(PI/6), SIN(PI/3), SIN(PI/2), SIN(PI*2/3),
SIN(PI*5/6), 0
```

PWM *expr1, expr2, expr3*

The PWM statement generates a variable duty cycle pulse train on Port 1, bit 2 (P1.2) that can be used for many purposes. The first number, *expr1*, is the number of clock cycles that the pulse will remain high. The second number, *expr2*, is the number of clock cycles that the pulse will remain low. The third number, *expr3*, is the total number of desired iterations. Each of these three expressions must be integers between 0 and 65535. However, the first two expressions must be larger than 24. The default length of a clock cycle is 12.0/XTAL seconds, which is 1.085 microseconds for 11.0592 MHz.

Note that on the 8051/31, DS5000, and derivative microcontrollers, the PGM and PWM commands interfere with the timing of the real-time clock. Refer to the CLOCK1 statement on page 18.

Example:

```
10 K=.2304147  ; assuming clock of 11.0592 MHz
20 INPUT "Microseconds high:" hms
30 INPUT "Microseconds low:" lms
40 INPUT "How many cycles", cycles%
50 PWM hms*K, lms*K, cycles
```

READ *var1* [, *var2* [, ...]]

The READ statement will assign one value per variable specified collected from a DATA statement. The values for these variables are obtained by scanning the DATA statements starting at the top of your program. Once a DATA value is read, READ moves on to the next DATA value until it runs out of DATA. The RESTORE statement can be used to start reading DATA from the first item of DATA. If DATA statements contain variables, READing them at different times may generate different values. The specified variables may be floating point, integer, or byte variables. An attempt to read data when all DATA statements have been read will cause an OUT OF DATA error.

Example:

```
10 CLOCK1
20 READ INTERVAL
30 IF INTERVAL = 0 THEN END
40 GOSUB {START_PROCESS}
50 TIME= 0
60 DO : WHILE TIME<INTERVAL  ; time the process
70 GOSUB {STOP_PROCESS}
80 GOSUB {NEXT_PROCESS}
90 GOTO 20  ; handle next interval
100 DATA 3.9, 2.4, 1.2, .79, .43, .25, .15, .1, .08, .05, 0
```

REM *text*
; *text*

The REM statement causes the text that follows it to be ignored. BXC-51 will not generate any code if the **-l** command line option is specified. The REM statement allows the programmer to put helpful commentary text into a program to help clarify it. A REM statement on a line by itself without a line number will not generate any code. The semicolon ';' may be used in place of the keyword 'REM'. Use many REM statements to keep your source code well documented.

Example:
```
     REM Example Hello World program
     10 PRINT "Hello, World!"   ; output the message
     20 REM That's it
```

RESTORE

The RESTORE statement will make the next READ statement fetch values from the first DATA statement in the program. RESTORE allows the same DATA to be read multiple times.

Example:
```
     10 FOR COUNT=1 TO 5
     20 RESTORE  ; start reading data from beginning
     30 FOR I=2 TO I: READ A: NEXT ; skip some
     40 FOR I=COUNT TO 5
     50 READ SUM(I-COUNT)
     60 NEXT
     70 NEXT COUNT
     80 DATA 1, 1/2, 1/4, 1/8, 1/16
     90 FOR COUNT=0 TO 4
     100 PRINT SUM(COUNT)
     110 NEXT
```

RETURN

Use the RETURN statement to finish a subroutine and return control to to just after the GOSUB to the subroutine. If there is no matching GOSUB, a C-STACK error occurs. Do not use this statement to return from an ONTIME, ONEX1, or ON*intr* interrupt subroutine - use RETI instead. Doing so will prevent another interrupt from being processed.

You may RETURN from inside a FOR or DO loop. The loops will be terminated upon RETURN (however, any loops used outside the subroutine will still be active).

Example:
```
     10 FOR I%=0 TO 16
     20 GOSUB 4000
     30 NEXT
     40 END
     4000 PH1. I%
     4010 RETURN

     10 DIM A(100)
     20 GOSUB 20000 ; get data to fill array A()
     30 FOR I%=1 TO 100 STEP 10
     40 POS1%= I%: POS2%= I%+9
     50 GOSUB {OUTPUT}
     60 NEXT
     70 END
     {OUTPUT} REM Output routine
     4000 PRINT POS1, ":",
     4010 FOR POS%=1 TO 100
     4020 IF POS% >= POS1% THEN PRINT A(POS%),
     4030 IF POS2%=POS% THEN PRINT: RETURN
     4040 NEXT
     4050 PRINT
     4060 RETURN
```

RETI
Use the RETI statement to complete an interrupt subroutine and return control to the section of the program that was executing before the interrupt occurred. It will perform a RETURN as well as allowing new interrupts to occur.

Example:
```
     10 TIME=0  ; reset time
     20 CLOCK1  ; enable clock
     30 ONTIME 1,1000  ; interrupt at 1 second
     40 DO
     50 WHILE TIME<9: END  ; stop at 9 seconds
     1000 REM ONTIME Interrupt
     1010 PRINT "Timer interrupt!"
     1020 RETI
```

SBUFFER *size* | OFF | ON | NOECHO | ECHO
This command controls how BXC-51 buffers input from the serial port. Normally, BXC-51 does not buffer its input (like MCS BASIC-52). Use this statement to allow buffering in your program. The buffer size must be a constant and must be between 1 and 253. This command should appear near the top of your program. Only specify the buffer size once; the buffer cannot dynamically change. A buffer size of 0 means no buffering. Once your program starts, all input will be buffered (for the GET function and the INPUT command). If the serial buffer becomes full, additional input is ignored until GET or INPUT fetches characters from the buffer.

Example:
```
     10 SBUFFER 100   ; allow 100 characters to be buffered
     20 ? "Type text now and see it appear in 3 seconds."
     30 CLOCK1: TIME= 0
     40 DO : UNTIL TIME>3  ; wait 3 seconds
     50 INPUT "Code: ", PASSWORD$
```

SBUFFER OFF
Use SBUFFER OFF to switch off serial line buffering temporarily. Any input currently in the serial buffer is lost.

Example:
```
     10 SBUFFER 100   ; allow 100 characters to be buffered
     20 ? "Type text now and see it ignored in 3 seconds."
```

```
30 CLOCK1: TIME= 0
40 DO : UNTIL TIME>3  ; wait 3 seconds
50 SBUFFER OFF : SBUFFER ON  ; flushes buffer
60 INPUT "Code: ", PASSWORD$
```

SBUFFER ON
Use SBUFFER ON to switch serial line buffering back on.

SBUFFER NOECHO
Use SBUFFER NOECHO to turn off the echo feature of the INPUT statement. Normally, as the user enters data to an INPUT command, BXC-51 echoes the characters back out to the serial line. Use NOECHO to turn off that echo temporarily.

Example:
```
10 SBUFFER 100   ; allow 100 characters to be buffered
30 INPUT "Login name: ", NAME$
40 SBUFFER NOECHO
50 INPUT "Password: ", PASSWORD$
60 SBUFFER ECHO
```

SBUFFER ECHO
Use SBUFFER ECHO to turn character echo back on for the INPUT command.

ST@ *expr*
Take the value from the top of the floating point Argument Stack and store it in external memory. For the six bytes of memory that a floating point number occupies, *expr* must be the address of the last byte. For example, use the

```
ST@ 6A05H
```

statement to store the value found on the top of the floating point argument stack in memory beginning at 6A00H. Use LD@ or PUSH commands to place values on the Argument Stack. If there is no value on the Argument Stack, an A-STACK error occurs.

Example:
```
10 FOR I%= 0 TO 20
20 PUSH INFO(I%)
30 ST@ 6A00H + I%*6 + 5
40 NEXT
```

STOP
Halt program execution, print out a STOP message, and output the current line number (unless the **-l** command line option was specified). If the **-x***addr* command line option was not specified, your program loops forever. Otherwise, the **-x***addr* command line option specifies where to continue execution. When a program ends, all BASIC interrupts are disabled.

Example:
```
10 PRINT "At any time, press 'Q' to quit."
20 GOSUB 1000  ; collect data
25 IF GET=ASC(Q) THEN STOP
```

```
      30 GOSUB 2000  ; process data
      35 IF GET=ASC(Q) THEN STOP
      40 GOSUB 3000  ; output data
      45 IF GET=ASC(Q) THEN STOP
      50 GOTO 20
```

STRING *expr1, expr2*

The STRING command allocates or reallocates storage space for static strings. The second expression is the number of characters per string, while the first expression is the total amount of string space to be allocated. Each string is one byte larger than *expr2* since it requires a terminating character, CR, to mark the end of the string. One extra byte is required to keep track of the number of strings. (*expr1*-1)/(*expr2*+1) rounded down, is the total number of strings allowed. The maximum number of strings is 254 and the lowest numbered string is $(0). Each time the STRING command is executed, a CLEAR is performed. This is for compatability with MCS BASIC-52. Additionally, the CLEAR statement does not de-allocate the STRING space. Only the command

```
      STRING 0,0
```

will do this. The value of *expr1* must not exceed free external RAM space available otherwise a MEMORY ALLOCATION error occurs.

Example:

```
      10 STRING 405, 100  ; 4 strings of 100 bytes each
      20 $(0)= "Summary Report of Operations"
      30 $(1)= "================================"
      40 $(2)= "Report of Operations by Device"
      50 $(3)= "========================="
      60 PRINT $(0)
      70 PRINT $(1)
      80 GOSUB 1000  ; output report
      90 PRINT $(1), CHR(12),
      100 PRINT $(2)
      110 PRINT $(3)
      120 GOSUB 2000  ; output report
      130 PRINT $(3)
```

TRACE0

The TRACE0 statement disables line number tracing started by the TRACE1 statement. After TRACE0, line numbers will no longer appear for each BASIC line (or statement).

Example:

```
      10 PRINT "Test program"
      20 PRINT "0. Exit"
      30 PRINT "1. Begin tracing"
      40 PRINT "2. Stop tracing"
      50 INPUT "Choice? ", A%
      60 IF A%=0 THEN END
      70 IF A%=1 THEN TRACE1
      80 IF A%=2 THEN TRACE0
      90 GOTO 50
```

TRACE1
The TRACE1 statement enables line number tracing for program debugging: As the program execution flows from one line to the next (by normal progression or by GOTO, NEXT, etc.), the new line number will appear. The line number is printed inside of square brackets, [ ], before executing your code on that line. If the **-g** command line option is specified, then the line number and statement number displays. It appears after that statement is executed (e.g., [200-1] is displayed after the first statement on line 200 has been executed). The TRACE1 statement is disabled if the **-l** command line option is specified.

The TRACE1 statement is extremely useful to help debug BASIC programs. It lets you see what the program is doing step-by-step. If your program crashes or executes the wrong way, TRACE1 helps show you where the problem lies.

See TRACE0 for example.

UI0
The UI0 statement changes the console input routines to the defaults, turning off the user-defined console input routines.

See UI1 for example.

UI1
The UI1 statement changes the console input routines to the user-provided assembly routines. The "get character" routine will be CALLed at 4033H (or wherever the **-c***addr* command line option specifies) and the character is expected to be returned in the accumulator. The "console status check" routine is CALLed at 4036H and must set the Carry flag high when a character is waiting. If not, it must be cleared. The user provided Assembly routine must use RB3 and must return the default bank to RB0 before RETurning to the main program.

This statement sets bit 1EH and the UI0 command clears it. Without the UI1 statement, BXC-51 does not generate the code required to allow user console input.

Example:
```
     10 buffer%= 8000H  ; location of input buffer
     20 UI1: INPUT "" CMD$: UI0  ; get console input
     30 IF CMD$="" THEN 100
     40 GOSUB 1000  ; process the command
     50 GOTO 20     ; get next command
     100 UI0  ; done with console input
     110 PRINT "Results:"
     120 GOSUB 2000 ; output results
     130 END
     $ASM
     HERE  EQU    $
           ORG    4033H     ; create console input handling
           JMP    XX_GETCHAR
           ORG    4036H     ; console status check
           SET    C         ; always ready
           RET
```

```
XX_GETCHAR:
      MOV    DPTR,#IV_BUFFER
      CALL   LDPTRI     ; DPTR = BUFFER%
      MOVX   A,@DPTR    ; get character
      MOV    R0,A       ; remember it
      INC    DPTR       ; update pointer
      MOV    R1,DPL
      MOV    R3,DPH
      MOV    DPTR,#IV_BUFFER
      CALL   IPUTVAR
      MOV    A,R0       ; recover character
      RET
      ORG    HERE
$BASIC
```

UNTIL *TF_expr*

The UNTIL statement is the second part of two statements that form a DO loop. The body of the loop contains BASIC statements between the DO and UNTIL pair. When this statement is executed, the true/false expression is evaluated. If it is false (0), program execution will resume following the matching DO statement. If the expression is true (non 0), program execution will resume after the UNTIL statement. Either way, the loop is executed at least once.

Example:

```
10 PRINT "Press any character"
20 TIME=0
30 CLOCK1
30 DO
40 UNTIL GET<>0
50 CLOCK0
60 PRINT TIME,"seconds elapsed before you pressed the key"
```

UO0

The UO0 statement changes the console output routines to the defaults, turning off the user-defined console output routines. After UO0, PRINT statements resume outputting to the serial port.

See UO1 for example.

UO1

The UO1 statement changes the console output routine to the user-provided assembly routine. The print character routine will be CALLed at 4030H (or wherever the **-c***addr* command line option specifies) with the character in the R5 of RB0. The user provided Assembly routine must only use RB3 and must return the default bank to RB0 before RETurning to the main program. Also, it must restore any modified registers.

This statement sets bit 1CH and the UO0 command clears it. Without the UO0 statement, BXC-51 ignores user console output.

Example:

```
10 buffer%= 8000H  ; location of character buffer
```

```
20 UO1   ; enable console output
30 PRINT "Hello, World!" ; store message starting at 8000H
40 END
$ASM
HERE  EQU    $
      ORG    4030H     ; create console output handling
      PUSH   DPL       ; save DPTR
      PUSH   DPH
      MOV    DPTR,#IV_BUFFER
      CALL   LDPTRI    ; DPTR = BUFFER%
      MOVX   @DPTR,A   ; save character
      INC    DPTR      ; update pointer
      MOV    R1,DPL
      MOV    R3,DPH
      MOV    DPTR,#IV_BUFFER
      CALL   IPUTVAR
      POP    DPH    ; restore DPTR
      POP    DPL
      RET
      ORG    HERE
$BASIC
```

WHILE *TF_expr*

The WHILE statement is the second part of two statements that form a DO loop. The body of the loop contains BASIC statements between the DO and WHILE pair. When this statement is executed, the true/false expression is evaluated. If true (non 0), program execution resumes following the matching DO statement. If false (0), program execution resumes after the WHILE statement. Either way, the loop is executed at least once.

Example:

```
10 PRINT "You must press SPACE four times to exit"
20 COUNT= 0
30 DO
40 IF GET=ASC( ) THEN COUNT= COUNT+1
50 WHILE COUNT<4
```

6. BASIC Functions

BXC-51 supports all MCS BASIC-52 functions. The following is a summary list of all these functions, plus the additional ones with check marks featured only in BXC-51.

	ABS(*x*)	Return the absolute value of *x*
	ASC(*c*)	Return the ASCII code for character *c*
	ASC(*s*)	Return the ASCII code at beginning of string *s*
	ASC($(*n*),*x*)	Return the ASCII code for a character in string *n*
	ATN(*x*)	Return the arctangent of *x*
	CBY(*x*)	Return byte value from program memory (ROM)
✓	CHR$(*c*)	Return string of ASCII code *c*
	COS(*x*)	Return the cosine of *x*
	DBY(*x*)	Return/Set contents of internal memory
	EXP(*x*)	Return the value of **e** raised to the *x*
✓	HIGH(*x*)	Return high byte value of *x*
✓	INT(*x*)	Return integer part of *x*
✓	LEFT$(*s,n*)	Return left most n characters of string
✓	LEN(s)	Return length of string s
	LOG(*x*)	Return natural logarithm of *x*
✓	LOW(*x*)	Return low byte value of *x*
✓	MID$(*s,n,m*)	Return range of characters in middle of string s
	NOT(*x*)	Return the logical NOT of *x* (1's complement)
✓	RIGHT$(*s,n*)	Return rightmost characters of string s
	SGN(*x*)	Return sign of *x*
	SIN(*x*)	Return sine of *x*
✓	STR$(*n*)	Convert a number n to a dynamic string
	SQR(*x*)	Return square root of *x*
	TAN(*x*)	Return tangent of *x*
✓	VAL(*s*)	Convert a string to a number
	XBY(*x*)	Return/Set contents of external memory (RAM)

These functions take different data type expressions as input. The following explains the syntax:

expr	A floating point, integer, or byte expression
character	A single character (letter, number, punctuation, etc.)
string	A static string variable
strexpr	A dynamic string expression

ABS(*expr*)

Return the absolute value of the expression in parenthesis. This function may be used in a floating point or integer expression.

Example:

```
10 INPUT "Enter any number: ", N
20 PRINT "Square root:", SQR(ABS(N)),
    30 IF N<0 THEN PRINT "i" ELSE PRINT
```

ASC(*character*)
Return the ASCII character code for the character in parenthesis. For example, ASC(A) is 65, ASC(:) is 58. Note: MCS BASIC-52 returns a different value than BXC-51 for ASC(*), ASC(+), ASC(-), and ASC(/) because MCS BASIC-52 incorrectly returns the token code for the symbols, not the ASCII code. This function may be used in a floating point or integer expression.

Example:
```
    10 PRINT "Press X to Exit"
    20 PRINT "Press C to Collect data"
    30 PRINT "Press O to Output data"
    40 DO: A=GET: WHILE A=0
    50 IF A=ASC(X) .OR. A=ASC(x) THEN END
    60 IF A=ASC(C) .OR. A=ASC(c) THEN GOSUB 1000 ; collect
    70 IF A=ASC(O) .OR. A=ASC(o) THEN GOSUB 2000 ; outout
    80 GOTO 40
```

ASC(*strexpr*)
ASC(*strexpr,expr*)
Return the ASCII character code for the first (or *expr*th) character in of the dynamic string expression. This function may be used in a floating point or integer expression.

Example:
```
    10 INPUT "Message: ", MESG$
    20 PRINT "Codes: ",
    30 FOR I%= 1 TO LEN(MESG$)
    40 PRINT ASC(MID$(MESG$, I%, 1)),
    50 NEXT
    60 PRINT
```

ASC(*string*, *expr*)
Return the ASCII character code for a character inside a static string. The expression determines which character is returned. If the expression is 1, the first character of the static string is returned; if it is 2, the second character is returned; etc. This function may be used in a floating point or integer expression. This function is different than most because it can be used to change the character in the string at the location specified (see LET on page 26).

Example:
```
    10 STRING 102,100
    20 INPUT "Message: ", $(0)
    30 PRINT "Codes: ",
    40 FOR I%= 1 TO 100
    50 PRINT ASC($(0), I%),
    60 IF ASC($(0), I%)=13 THEN I%=100 ; exit loop
    60 NEXT
    70 PRINT
```

ATN(*expr*)
Return the arctangent, tan^{-1}, of the expression. The value is returned in radians. This function may only be used in a floating point expression.

Example:
```
     10 PRINT "Inside the right triangle, measure the leg
     lengths"
     20 INPUT "Leg 1's length:", L1
     30 INPUT "Leg 2's length:", L2
     40 PRINT "Angle between leg 1 and hypotenuse:", ATN(L1/L2)
     50 PRINT "Angle between leg 1 and hypotenuse:", ATN(L2/L1)
```

CBY(*expr*), CBY#(*expr*)
Return the byte value from program memory (ROM). The expression *expr* is the ROM address that to read.

Example:
```
     10 PRINT "This program really begins at",
     20 PH0. CBY(ROMORG%+1)*256+CBY(ROMORG%+2)
```

CHR$(*expr*)
Convert the value of *expr* to a 1 character dynamic string for the ASCII value *expr*. Values between 0 and 255 (including 13) are valid.

Note that using this function will corrupt any program in the BASIC-52 interpreter when using the **-2i** command line option since the string buffer is located at 200H through 300H. To avoid this, use the ASC()= command with static strings.

Example:
```
     10 PRINT "Printable characters:"
     20 FOR I%=32 TO 127 STEP 32
     30 FOR J%=I% TO I%+31
     40 PRINT CHR$(J%),
     50 NEXT
     60 PRINT
     70 NEXT
```

COS(*expr*)
Return the cosine of the expression. The expression evaluated as radians. This function may only be used in a floating point expression.

Example:
```
     10 INPUT "Angle, in radians: ", A
     20 INPUT "Radius of circle: ", R
     30 PRINT "Given a circle at (0,0), the point is at ",
     40 PRINT "(", R*COS(A), ",", R*SIN(A), ")"
```

DBY(*expr*), DBY#(*expr*)
Return/Set contents of internal memory. On an 8051/31, DS5000, and some derivative microcontrollers, internal memory ranges from 0 to 127, or 0H to 7FH. This function is different

than most because it can be used to get the internal memory value or set internal memory. This function is bit addressable (followed by dot and bit position from 0 to 7) to return a specific bit. If this function appears on the left side in an assignment statement, that internal memory location will be altered (see the LET command on page 26). For example,

```
DBY(3EH)= 0A0H
```

will change the internal stack home position to begin at A0H when CLEARS is executed.

You may wish to use byte variables (see page) instead of using the DBY() because the name of the byte variable may be more descriptive than a specific address such as 3EH.

Example:
```
10 DEFVAR STK0#@3EH
20 DBY(3EH)= 60H
30 PRINT "Stack now at", STK0#
40 STK0#= 50H
50 PRINT "Stack now at", STK0#
```

EXP(*expr*)
Return the value of **e** (2.7182818) raised to the power of the value of the expression. This function may only be used in a floating point expression.

Example:
```
10 INPUT "Power of e: ", X
20 PRINT "e to that power is", EXP(X)
```

HIGH(*expr*)
Return the value of the high (or most significant) byte of an integer expression. This function is useful in both floating point and integer expressions. As a floating point function, a valid value is still returned for the range 32768 through 65535 which directly maps to the high byte integer values for -32768 through -1. For example, the integer number 770 has an upper byte of 3 and lower byte of 2 because 3*256+2=770; HIGH(770) yields 3.

Example:
```
10 A%= 34E1H
20 XBY(5000H)= HIGH(A%)  ; 34H
30 XBY(5001H)= LOW(A%)  ; E1H
```

INT(*expr*)
Return integer part of the floating point expression. This neither rounds down nor rounds up, it just truncates the decimal part. For example, INT(4.3) returns 4 while INT(-5.2) returns -5.

This function is required to convert a floating point expression to an integer value inside integer expressions. This function may be used in a floating point or integer expression.

Example:
```
10 INPUT "Radians: ", R
20 DEGREES%= INT(180*R/PI)
```

```
      30 PRINT "Degrees: ", DEGREES%
```

LEFT$(*strexpr,expr*)
Return the leftmost characters of a dynamic string expression, *strexpr*. *strexpr* is any valid string expression and *expr* is any integer greater than or equal to 0. Note, if *expr* is larger than the length of the string expression, the whole string expression is returned. A negative length causes a BAD ARGUMENT error.

Note that use of this function will corrupt any program in the BASIC-52 interpreter when using the **-2i** command line option since the string buffer is located at 200H through 300H.

Example:
```
      10 DIM A(20)
      20 GOSUB 2000 ; collect data into array A()
      30 PRINT "Bar chart of data" : PRINT
      40 DOTS$="********************" ; 20
      50 FOR I%=1 TO 20
      60 PRINT I%, TAB(5), "|", LEFT$(DOTS$, A(I%))
      70 NEXT
```

LEN(*strexpr*)
Return the length (in characters) of a dynamic string expression, *strexpr*. This value will always be from 0 to 255.

Example:
```
      10 WORD$= "honey"
      20 PRINT "Strip the first and last letter of ", WORD$, ": ",
      30 PRINT MID$(WORD$, 2, LEN(WORD$)-2)
```

LOG(*expr*)
Return the natural logarithm of the expression. Remember that logarithms are only valid for non-zero, positive numbers. LOG(0) or the LOG() of a negative number will cause a BAD ARGUMENT error. This function can only be used in a floating point expression.

Example:
```
      10 INPUT "Enter a number: ", A
      20 PRINT "Natural log:", LOG(A)
      30 PRINT "Log base 10:", LOG(A)/LOG(10)
```

LOW(*expr*)
Return the value of the low (or least significant) byte of an integer expression. The integer number 767 has an upper byte of 2 and lower byte of 255 because 2*256+255=767; LOW(767) yields 255. This function may be used in both floating point and integer expressions.

Example:
```
      10 A%= 02FFH
      20 XBY(5000H)= HIGH(A%)  ; 02H
      30 XBY(5001H)= LOW(A%)  ; FFH
```

MID$(*strexpr,expr1,expr2*)
Return a range of characters in the middle of a dynamic string expression *strexpr*, starting at character position *expr1* for a total length of *expr2* characters. The start of the range is 1 or higher. If the starting point is beyond the number of characters actually in the string, an empty string is returned. The length may be from 0 to 255. If start position plus length is larger than the length of the string expression, then the whole string from start position onward is returned. Specify a length of 255 to get the whole string from the starting position onward. A negative start position or length causes a BAD ARGUMENT error.

Note that use of this function will corrupt any program in the BASIC-52 interpreter when using the **-2i** command line option since the string buffer is located at 200H through 300H.

Example:
```
     10 MON$="JanFebMarAprMayJunJulAugSepOctNovDec"
     20 INPUT "Enter date (MM/DD/YY): ", MONTH, DAY, YEAR
     30 PRINT MID$(MON$, MONTH*3+1, 3), DAY, ", ", YEAR
```

NOT(*expr*)
Return the logical NOT of the expression (1's complement). Any decimal part of a floating point number is ignored. This function may be used in a floating point or integer expression.

Example:
```
     10 LED%= XBY(0E000H)  ; get the LED status
     20 XBY(0E000H)= NOT(LED%)  ; invert the LEDs
     30 TIME=0: CLOCK1
     40 DO : WHILE TIME<.5  ; wait .5 seconds
     50 XBY(0E000H)= NOT(LED%)  ; revert the LEDs to previous
```

RIGHT$(*strexpr,expr*)
Return the rightmost characters of a string expression, *strexpr*. *strexpr* is any valid dynamic string expression and the length, *expr*, is any integer greater than or equal to 0. If *expr* is larger than the length of the string expression, the whole string is returned. A negative length causes a BAD ARGUMENT error.

Note that use of this function will corrupt any program in the BASIC-52 interpreter when using the **-2i** command line option since the string buffer is located at 200H through 300H.

Example:
```
     10 PRINT "Each part name should begin with three letters"
     20 PRINT "and end with 4 numbers, e.g. ABC-1234"
     30 INPUT "Part name: ", PART$
     40 PRINT "Part type: ", LEFT$(PART$, 3)
     50 PRINT "Part number: ", RIGHT$(PART$, 4)
```

SGN(*expr*)
Return the sign of the expression. The return values are -1, 0, or 1 if the expression is negative, zero, or, positive, respectively.

Example:

```
10 INPUT "A negative or positive number: ", A
20 PRINT "Pushing it 10 further from 0: ", A+SGN(A)*10
```

SIN(*expr*)
Return the sine of the expression. The expression evaluated as radians. This function can only be used in a floating point expression.

Example:
```
10 PRINT "Bar chart of sine" : PRINT
20 DOTS$="********************" ; 20
30 FOR ANGLE=0 TO PI STEP PI/19
40 PRINT USING(#.###), ANGLE, TAB(8),
50 PRINT "|", LEFT$(DOTS$, SIN(ANGLE))
60 NEXT
```

SQR(*expr*)
Return the square root of the expression. If the expression is negative, a BAD ARGUMENT error will be reported. This function can only be used in a floating point expression, however INT(SQR(*expr*)) can be used in an integer expression.

Example:
```
10 INPUT "Two legs of a right triangle: ", L1, L2
20 PRINT "Hypotenuse:", SQR(L1*L1+L2*L2)
```

STR$(*expr*)
Convert the numeric value *expr* into a string. The current output format set by the last USING statement determines the format of the string returned. The returned string will not have any leading or trailing spaces in it.

Note that use of this function will corrupt any program in the BASIC-52 interpreter when using the **-2i** command line option since the string buffer is located at 200H through 300H.

Example:
```
10 PRINT "Enter sample height: ", H
20 MESG$= "Height("+STR$(H)+")"
30 PRINT MESG$  ; prints Height(3) if H=3
```

TAN(*expr*)
Return the tangent of the expression. The expression evaulated as radians. This function will report a DIVIDE BY ZERO if the expression evaluates to π/2 or -π/2. This function can only be used in a floating point expression.

Example:
```
10 INPUT "Enter angle: ", A
20 PRINT "Slope of line is: ", TAN(A)
```

VAL(*strexpr*)
Convert a string expression to a numeric value. The beginning of the string must have a numeric value otherwise 0 is returned. All non-numeric text is ignored from the first non-numeric character onward. Any number following non-numeric text is ignored.

Example:
```
     10 PRINT "Follow time with units of time using"
     20 PRINT "s for seconds or m for minutes, e.g. 10s"
     30 INPUT "Time to wait: ", DELAY$
     40 DELAY=VAL(DELAY$)
     50 IF RIGHT$(DELAY$, 1)="m" THEN DELAY= DELAY*60
     60 TIME=0: CLOCK1
     70 DO : WHILE TIME<=DELAY
```

XBY(*expr*), XBY#(*expr*)
Return/Set contents of external memory (RAM). *expr* is the address of the external memory location. This function is different than most because it can be used to get the external memory value or set external memory. If this function appears on the left side of an assignment, the external memory location will be altered (see the LET command on page 26).

Example:
```
     10 OT%= XBY(126H)*256+XBY(127H)
     20 IF OT%=0 THEN
     30    PRINT "ONTIME is not setup"
     40 ELSE
     50    PH0. "ONTIME is setup at", OT%
     60 ENDIF
```

7. BASIC Special Variables

BXC-51 supports all MCS BASIC-52 special variables. The following is a summary list of all the special variables, plus the additional ones with check marks featured only in BXC-51.

✓	ERRLINE%	Line number of last error
✓	ERRVALUE%	Error code of last error
✓	FALSE	Return logical false, *0*
	FREE	Return the amount of RAM left
	GET	Return current character on console
	IE	Return/Set value of interrupt enable register
	IP	Return/Set value of interrupt priority register
	LEN	Return length of program
✓	MCON#	Return/set value of memory control reg.
	MTOP	Return/Set the top of memory
	PCON	Return/Set the power control register
	PI	Return the value of π
✓	PORT0#	Return/Set value of P0 I/O port
	PORT1	Return/Set value of P1 I/O port
✓	PORT2#	Return/Set value of P2 I/O port
✓	PORT3#	Return/Set value of P3 I/O port
✓	RAMORG	Return starting location of RAM
	RCAP2	Return/Set value for timer 2's reload/capture registers
	RND	Return a random number
✓	ROMORG	Return starting location of program (ROM)
	T2CON	Return/Set value of timer/counter 2 control register
	TCON	Return/Set value of timer/counter control register
	TIME	Return/Set value of real-time clock
	TIMER0	Return/Set value of timer/counter 0
	TIMER1	Return/Set value of timer/counter 1
	TIMER2	Return/Set value of timer/counter 2
	TMOD	Return/Set timer/counter mode control register
✓	TRUE	Return logical true, *-1*
	XTAL	Return/Set value of system clock speed, in Hz

Most of these variables may be used in floating point, integer, and byte expressions. Most can have expressions assigned to them. Some of these variables require the integer or byte variable syntax of a trailing % or #. Be careful to use it. For variable names that optionally appear as integer or byte variables, this is indicated below.

`ERRLINE%`

Return/set the value of line number of the last error. This information is most helpful when writing an ONERR subroutine which is handling an error. Initially, this value is 0.

Example:

```
10 ONERR 1000
20 READ A  ; without any DATA, this is an error
30 PRINT A
40 END
1000 PRINT "Error on line", ERRLINE%
1010 PRINT "Beginning clean-up procedures"
1020 GOSUB 4000 ; clean up
1030 END
```

ERRVALUE%

Return the value of the last error that occurred. Initially, this value is 0. After an error, the possible value of ERRVALUE% is:

0	No error
10	Divide by zero
20	Arithmetic overflow
30	Arithmetic underflow
40	Bad argument
250	Non-arithmetic error

Example:

```
10 ONERR 1000
20 PRINT A/0  ; /0 is an error
30 END
1000 PRINT "Error", ERRVALUE%, "on line", ERRLINE%
1010 PRINT "Beginning clean-up procedures"
1020 GOSUB 4000 ; clean up
1030 END
```

FALSE, FALSE%

Return the value of logical false, which is 0. This special variable is useful to make code more readable (e.g., DONE=FALSE). FALSE is a constant; a value cannot be assigned to it.

Example:

```
10 DONE%= FALSE: CLOCK1: TIME=0
20 DO
30 INPUT "Enter activity number (0=Exit): ", AN
40 IF AN=0 THEN DONE%= TRUE
50 IF AN<0 THEN GOSUB 2000
60 IF AN>0 THEN GOSUB 3000
70 PRINT "Time", TIME
80 UNTIL DONE%
```

FREE, FREE%

Return the amount of RAM extra left for use by program. This value will only change when an array is allocated, when the STRING statement is executed or when MTOP changes (see page 55). In MCS BASIC-52, FREE decreases every time a new variable is referenced. In BXC-51, all the variables are allocated at compile-time and cannot be deallocated.

Unlike the MCS BASIC-52 interpreter, FREE does not equal MTOP - LEN - 511 because the program space, LEN, is not in the same block of memory as RAM variables. A value cannot be assigned to FREE.

When using the integer variation, FREE%, the value may be negative if the amount of free memory is greater than 32K. The real value of free memory is FREE%+65536.

Example:
```
    10 REM Allocate as many 100 byte strings will fit in 1/4 of
    11 REM the free memory
    20 STRING FREE%/4, 100
    30 DEFVAR VARTOP%@104H
    30 NS%= XBY(VARTOP%)
    40 PRINT NS%, "strings allocated"
```

GET, GET#

Return the current pressed character on console. If no character has been received, GET returns 0. If a character is pressed and waiting on the console, the ASCII value of that character will be returned. After using GET, the character on the console is cleared. When a second character arrives before the first one is read, the second character is discarded. See the SBUFFER statement on page 38 to allow GET to buffer characters so none are lost. A value cannot be assigned to GET.

Example:
```
    10 PRINT "Press Q to quit"
    20 DO
    30 KEY%= GET
    40 UNTIL KEY%=ASC(Q) .OR. KEY%=ASC(q)
    50 PRINT "All done"
```

IE, IE#

Return/Set value of Interrupt Enable register. If this special variable appears on the left side of an assignment statement, IE is altered. Otherwise, IE's present value is returned. The only other BASIC statements that alter IE are: CLEAR, CLEARI, CLOCK0, CLOCK1, and ONEX1. See page 113 for a summary of this Special Function Register.

Example:
```
    10 PRINT "Interrupt enable:"
    20 IF IE#.0 THEN PRINT "EX0 enabled"
    30 IF IE#.1 THEN PRINT "Timer 0 enabled"
    40 IF IE#.2 THEN PRINT "EX1 enabled"
    50 IF IE#.3 THEN PRINT "Timer 1 enabled"
    60 IF IE#.4 THEN PRINT "Serial enabled"
    70 IF IE#.0 THEN PRINT "Timer 2 enabled"
```

IP, IP#

Return/Set value of interrupt priority register. If used alone, this special variable returns the current settings of the interrupt priority register. If used in an assignment statement, it alters the register. See page 113 for a summary of this special function register.

Example:
```
     10 PRINT "Interrupt priority:"
     20 IF IE#.0 THEN PRINT "EX0 high"
     30 IF IE#.1 THEN PRINT "Timer 0 high"
     40 IF IE#.2 THEN PRINT "EX1 high"
     50 IF IE#.3 THEN PRINT "Timer 1 high"
     60 IF IE#.4 THEN PRINT "Serial high"
     70 IF IE#.0 THEN PRINT "Timer 2 high"
```

LEN, LEN%
Return the length of your program. This value is fixed because your program is compiled. A value cannot be assigne to LEN.

If your program is larger than 32K, then LEN% will be negative; the real length is LEN%+65536.

Example:
```
     10 PRINT "This program is", LEN, "bytes long."
     20 PRINT "Starting at", ROMORG%
     30 PRINT "Ending at", ROMORG%+LEN-1
```

MCON#
Return the Memory Control register. This variable only has meaning on a DS5000 CPU. A new value cannot be assigned to MCON#, however, it can be altered by using this Assembly code:

```
     MOV     C7,#0AAH
     MOV     C7,#055H
     MOV     BV_MCON,#newvalue
```

Example:
```
     10 PRINT "This DS5000's configuration:"
     20 IF MCON#.3 THEN SIZE%=8000H ELSE SIZE%=2000H
     30 SPLIT%= 800H*(MCON# .SHR. 4)
     40 IF (MCON# .SHR. 4)=15 THEN SPLIT%=8000H
     50 PH0. "ROM: ", 0, "to", SPLIT%-1
     60 PH0. "RAM: ", SPLIT%, "to", SIZE%-1
```

MTOP, MTOP%
Return/Set the top of external RAM memory. The special variable contains the address of the highest addressable RAM location usable by BASIC when your program starts. If the **-u***addr* command line option is specified, MTOP is initially set to that address (unless the **-2i** or **-sub** command line options are present in which case MTOP is not set at all at program startup.) If you wish to alter this value, assign it a new value. If the **-u***addr* command line option was specified, MTOP is initially set to it.

If your program needs to alter MTOP, it should do so at the beginning. The value of MTOP is used to determine where dynamic strings may be stored (just below MTOP).

If this value is larger than 7FFFH then MTOP% will be negative; the real memory top is MTOP%+65536.

MTOP is used to determine where to place static strings (seee STRING command on page 40) and where to store dynamic strings.

Example:
```
     10 REM Leave 8000H and above for other programs
     20 IF MTOP>=8000H THEN MTOP=7FFFH
```

PCON, PCON#
Return/Set the power control register. See page 112 for a summary of this special function register.

Example:
```
     10 PRINT "Power Control register:"
     20 PRINT "Value: ", PCON#
     30 IF PCON#.7 THEN PRINT "Baud rate doubled"
```

PI
Return the value of π, 3.1415926. This special variable can only be used in a floating point expression. PI is a constant. A new value cannot be assigned to it.

Example:
```
     10 INPUT "How many points on a circle? ", PTS%
     20 INPUT "Radius of circle? ", R
     30 FOR I= 0 TO 2*PI STEP 2*PI/(PTS%-1)
     40 PRINT USING(#.###),I, "= (", R*COS(I), ",", R*SIN(I), ")"
     50 NEXT
```

PORT0#
Return/Set value of P0 I/O port. This variable provides dubious value on boards designed with external RAM because PORT0 is used for the address and data bus. However, on the DS5000 and derivative microcontrollers which use on-chip 'external' RAM, this variable provides access to Port 0. When assigning values to a particular bit of the port, remember that bit assignments read the port, change the bit, then write the port. When assigning values to a particular bit of the port, remember that bit assignments read the port, change the bit, then write the port.

Example:
```
     10 PRINT "Port 0:"
     20 PRINT "PORT0.0 =", PORT0#.0
     30 PRINT "PORT0.1 =", PORT0#.1
     40 PRINT "PORT0.2 =", PORT0#.2
     50 PRINT "PORT0.3 =", PORT0#.3
     60 PRINT "PORT0.4 =", PORT0#.4
     70 PRINT "PORT0.5 =", PORT0#.5
     80 PRINT "PORT0.6 =", PORT0#.6
     90 PRINT "PORT0.7 =", PORT0#.7
```

PORT1, PORT1#
Return/Set value of P1 I/O port.

Example:

```
10 PRINT "Port 1:"
20 PRINT "PORT1.0/T2 =", PORT1#.0
30 PRINT "PORT1.1/T2EX =", PORT1#.1
40 PRINT "PORT1.2 =", PORT1#.2
50 PRINT "PORT1.3 =", PORT1#.3
60 PRINT "PORT1.4 =", PORT1#.4
70 PRINT "PORT1.5 =", PORT1#.5
80 PRINT "PORT1.6 =", PORT1#.6
90 PRINT "PORT1.7 =", PORT1#.7
```

PORT2#

Return/Set value of P2 I/O port. This port is used for addressing external RAM unless the **-5** command line option is specified (for DS5000 CPU). This variable provides dubious value on boards designed with external RAM because PORT0 is used for the address bus. However, on the DS5000 and derivative microcontrollers which use on -chip 'external' RAM, this variable provides access to Port 2. When assigning values to a particular bit of the port, remember that bit assignments read the port, change the bit, then write the port.

Example:
```
10 PRINT "Port 2:"
20 PRINT "PORT2.0 =", PORT2#.0
30 PRINT "PORT2.1 =", PORT2#.1
40 PRINT "PORT2.2 =", PORT2#.2
50 PRINT "PORT2.3 =", PORT2#.3
60 PRINT "PORT2.4 =", PORT2#.4
70 PRINT "PORT2.5 =", PORT2#.5
80 PRINT "PORT2.6 =", PORT2#.6
90 PRINT "PORT2.7 =", PORT2#.7
```

PORT3#

Return/Set value of P3 I/O port.

Example:
```
10 PRINT "Port 3:"
20 PRINT "PORT3.0/RXD =", PORT3#.0
30 PRINT "PORT3.1/TXD =", PORT3#.1
40 PRINT "PORT3.2/INT0 =", PORT3#.2
50 PRINT "PORT3.3/INT1 =", PORT3#.3
60 PRINT "PORT3.4/T0 =", PORT3#.4
70 PRINT "PORT3.5/T1 =", PORT3#.5
80 PRINT "PORT3.6/WR =", PORT3#.6
90 PRINT "PORT3.7/RD =", PORT3#.7
```

RAMORG, RAMORG%

Return starting location of the external RAM system variables. The BASIC program variables start at RAMORG+200H. If the **-v***addr* compiler option is specified, RAMORG will be that address. Otherwise, RAMORG will be 0.

If this value is larger than 7FFFH, RAMORG% will be negative. The real starting location is RAMORG%+65536. RAMORG is fixed at compile-time; a value cannot be assigned to it.

Example:
```
     10 PRINT "RAM usage:"
     20 PH0. "BASIC system RAM:", RAMORG%, "to", RAMORG%+200H
     30 PH0. "Top of BASIC RAM:", MTOP
     40 PH0. "Total BASIC RAM:", MTOP-RAMORG%
     50 PH0. "Total RAM free:", FREE
```

RCAP2, RCAP2%
Return/Set value for Timer 2's Reload/Capture register. This register is responsible for generating the baud rate of the serial port on the 8052/8032 microcontrollers. Therefore, altering this variable may change the baud rate. This register is only available on the 8052/32.

Note that RCAP2% refers to the RCAP2 Special Function Register, not an external RAM integer.

Example:
```
     10 PRINT "RCAP2 register:", RCAP2
```

RND
Return a random number between 0.0 and 1.0. The random number is generated as an integer from 0 to 65535 and it is divided by 65535 to reduce it to the 0 to 1 range. This special variable can only be used in a floating point expression however INT(RND*MAX) may be used in integer expressions. Values cannot be assigned to RND.

Example:
```
     10 REM Wait for 1 to 10 seconds
     20 ONTIME INT(RND*9.99999)+1, 1000
     30 TIME=0: CLOCK1
     40 IDLE
     50 END
     1000 CLOCK0
     1010 PRINT "Random time waited:", TIME, "seconds"
```

ROMORG, ROMORG%
Return the starting location of your program (ROM). If the **-p***addr* compiler option was specified, ROMORG is that address; otherwise, ROMORG is 0. ROMORG is fixed at compile-time; a value cannot be assigned to it.

If this value is larger than 7FFFH, ROMORG% will be negative. The real starting location is ROMORG%+65536.

Example:
```
     10 PH0. "This program starts at", ROMORG,
     20 PH0. "in memory and ends at", ROMORG+LEN-1
```

T2CON, T2CON#
Return/Set value of Timer/Counter 2 control register. This special variable controls whether the serial port baud rate timer is controlled by Timer 1 or Timer 2, which should not be changed. This register is available only on the 8052/32 microcontroller. See page 113 for a summary of this special function register.

Example:
```
     10 PRINT "T2CON: ", T2CON
     20 PRINT "T2CON.0/CPRL2 =", T2CON#.0
     30 PRINT "T2CON.1/CT2 =", T2CON#.1
     40 PRINT "T2CON.2/TR2 =", T2CON#.2
     50 PRINT "T2CON.3/EXEN2 =", T2CON#.3
     60 PRINT "T2CON.4/TCLK =", T2CON#.4
     70 PRINT "T2CON.5/RCLK =", T2CON#.5
     80 PRINT "T2CON.6/EXF2 =", T2CON#.6
     90 PRINT "T2CON.7/TF2 =", T2CON#.7
```

TCON, TCON#

Return/Set value of Timer/Counter Control register. This special variable is used to control Timer 0, Timer 1, External Interrupt 0, and External Interrupt 1.

Timer 0 is used by the CLOCK0/CLOCK1 commands. On the 8051/31, DS5000, and derivative microcontrollers, Timer 0 also controls the PGM and PWM commands. Because of this, care must be taken not to use the time clock simultaneously with the PGM or PWM commands. Timer 1 is used for the baud rate on the 8051/31, DS5000, and derivative microcontrollers. On the 8052 microcontroller, Timer 1 is used by the PGM and PWM commands.

See page 113 for a summary of this special function register.

Example:
```
     10 PRINT "Timer 0:"
     20 IF TCON#.4 THEN PRINT "Timer 0 is running."
     30 IF TCON#.5 THEN PRINT "Timer 0 interupt pending."
     40 PRINT "Timer 1:"
     50 IF TCON#.6 THEN PRINT "Timer 1 is running."
     60 IF TCON#.7 THEN PRINT "Timer 1 interupt pending."
     70 PRINT "External Interrupt 0:"
     80 IF TCON#.0 THEN PRINT "Falling edge triggered"
     90 IF TCON#.0=0 THEN PRINT "Low level triggered"
     100 IF TCON#.1 THEN PRINT "Interupt pending."
     110 PRINT "External Interrupt 1:"
     120 IF TCON#.2 THEN PRINT "Falling edge triggered"
     130 IF TCON#.2=0 THEN PRINT "Low level triggered"
     140 IF TCON#.3 THEN PRINT "Interupt pending."
```

TIME

Return/Set value of real-time clock in seconds. This special variable will change if a CLOCK1 statement has been executed and will stay constant following a CLOCK0 statement. This special variable can only be used in a floating point expression, however INT(TIME) may be used in integer and byte expressions. After the CLOCK1 statement, the TIME variable increases with the passage of time. Before the CLOCK1 statement, it is customary to set TIME to 0 for timing. You may set TIME to the number of seconds since midnight. However, TIME will not reset to 0 at the next midnight. Timing accuracy is roughly .025 seconds.

Example:
```
     10 PRINT "Timing program routines"
     20 CLOCK1
     30 TIME=0: GOSUB 1000  ; routine to be timed
     40 A=TIME
     50 PRINT "Routine 1000 took", A, "seconds"
     60 TIME=0: GOSUB 2000  ; routine to be timed
     70 A=TIME
     50 PRINT "Routine 2000 took", A, "seconds"
```

TIMER0, TIMER0%

Return/Set value of Timer/Counter 0. This special variable is used to control updating the TIME special variable after a CLOCK1 statement. On the 8051/31, DS5000, and derivative microcontrollers, the PGM and PWM statements use Timer 0.

If this value is larger than 32767, TIMER0% will be negative; the real value is TIMER0%+65536.

Note that TIMER0% refers to the TIMER0 special function register, not an external RAM integer.

Example:
```
     10 PRINT "Timer 0:", TIMER0
```

TIMER1, TIMER1%

Return/Set value of timer/counter 1. This special variable is used to control the baud rate on an 8051/31 or DS5000. On the 8052/32, it is also used by the PGM, PRINT#, and PWM statements. If this value is larger than 32767, TIMER1% will be negative; the real value is TIMER1%+65536.

Note that TIMER1% refers to the TIMER1 Special Function Register, not an external RAM integer.

Example:
```
     10 PRINT "Timer 1:", TIMER1
```

TIMER2, TIMER2%

Return/Set value of Timer/Counter 2. This variable is not available on the 8051/31, DS5000, and derivative microcontrollers. This special variable is used to control the baud rate on an 8052/32.

If this value is larger than 32767, TIMER2% will be negative; the real value is TIMER2%+65536.

Note that TIMER2% refers to the TIMER2 special function register, not an external RAM integer.

Example:
```
     10 PRINT "Timer 2:", TIMER2
```

TMOD, TMOD#

Return/Set Timer/Counter Mode Control register. It controls the modes of Timer 0 and Timer 1. See page 113 for a summary of this special function register.

Example:

```
     10 PRINT "Timer 0:"
     20 IF TMOD#.2 THEN ? "Counter mode" ELSE ? "Timer mode"
     30 PRINT "Mode: ", TMOD#.0+2*TMOD#.1
     40 IF TMOD#.3 THEN PRINT "Gate enabled."
     50 PRINT "Timer 1:"
     60 IF TMOD#.6 THEN ? " Counter mode" ELSE ? "Timer mode"
     70 PRINT "Mode: ", TMOD#.4+2*TMOD#.5
     80 IF TMOD#.7 THEN PRINT "Gate enabled."
```

TRUE, TRUE%

Returns the value of logical true which is -1 or FFFFH. This special variable is useful to make code more readable (e.g., IF CONDITION=TRUE THEN 500). TRUE is a constant; a value may not be assigned to it.

Example:

```
     10 KEEP_GOING%= TRUE
     20 DO
     30 GOSUB 1000 ; determine fluid level
     40 IF LEVEL<.1 THEN KEEP_GOING%= FALSE
     50 WHILE KEEP_GOING%
```

XTAL

Return/Set value of system clock speed in Hz. The default value is 11,059,200 Hz or 11.0592 MHz. Altering this variable will not alter the baud rate unless RCAP2 (or TIMER1) is also changed. The baud rate is determined by

$$BAUD = \frac{XTAL}{32\times(65536-RCAP2)}$$

for the 8052/32 microcontroller, and

$$BAUD = \frac{XTAL}{192\times(256-INT(TIMER1/256))}$$

for the 8051/31, DS5000, and derivative microcontrollers. This special variable can only be used in floating point expressions.

8. Getting Started

This section assumes you are familiar with BASIC-52 programming, but unfamiliar with compiling using BXC-51. This section will demonstrate how to use the compiler to compile and run your BASIC program. After performing the steps outlined in this section, you should explore other parts of this manual, depending upon your needs.

Creating Source Code

Before you can use BXC-51, you must have BASIC source code. There are two ways to do so, by using the interpreter or using an editor. The result of either way must be that your created source must be in a file on your PC. By convention, BASIC source code files use the .BAS file extension.

Many engineers prefer to develop their program in the BASIC-52 interpreter before compiling. This has the advantage that you can test your program and modify it quickly and easily. This has the disadvantage that you cannot take advantage of the BXC-51 extensions to BASIC. Once you have developed the code to your satisfaction, upload the source to a file on your PC. The BASIC Toolkit (BTK) product from Binary Technology, Inc. is a useful tool for this stage of development.

Many engineers also prefer to develop their program using a text editor on the PC, such as Infinitor (contact Binary Technology, Inc. Use the text editor to write your BASIC source code. Quit the editor and run the compiler. If the compiler generates any errors, re-edit your BASIC source code to correct the errors. This is a very typical cycle of development. The BXC-51 IDE product (available from Binary Technology, Inc. is an integrated development environment which allows you to edit your BASIC source, compile it, download it, and run it all without exiting to DOS.

Regardless of your development environment, you must create a BASIC source code file on your PC.

The Default Compiler Options

Before running BXC51.EXE, you must consider how you are using memory on your target system, what baud to use for the serial port, which microcontroller, and other issues. By default, BXC-51 assumes the following options when no command line options are present:

a. It starts in ROM (code memory) at 0H. If you need to locate it somewhere else in ROM, use **-p***addr* command line option to change it. For example, if your target system has the BASIC-52 interpreter present, the lowest available ROM address is 2000H.

b. External RAM is assumed to start at 0H. If you do not have RAM configured there or you desire to have it configured elsewhere, use the **-v***addr* command line option to change it.

c. All contiguous RAM bytes will be cleared from 0 to E000H. If you used **-v***addr*, then all contiguous RAM bytes will be cleared from that address to E000H. To specify an upper bounds on memory clearing and use of memory by your BXC-51 compiled program, use the **-u***addr* command line option to change it.

d. Set the baud rate by pressing the space bar when the program starts. This is normal when using BASIC-52 interpreter, however you may specify a baud rate ahead of time by using the **-b***rate* command line option to change it.

e. Your program assumes your target microcontroller is an 8051/31. For a 8052 target microcontroller, use the **-2** command line option. For a DS5000 target microcontroller, use the **-5** command line option. For a derivative microcontroller target (such as the 8xC550 or 8xC552), use the **-t***cpu* command line option.

f. If an error occurs, the BASIC source code line number will be displayed. Each line number generates code. To hide the BASIC source code numbers, use **-l** command line option to compile without them.

g. Only arithmetic errors can be trapped. To trap other errors such as NO DATA, use the **-e** command line option and ONERR BASIC command.

h. When the program finishes, it will loop indefinitely. To exit to a specific ROM address to run other programs, use the **-x***addr* command line option. This is handy for jumping into monitor code, if you have a monitor on your board (such as M/DP available from Binary Technology, Inc.) or restarting the program.

i. The compiled code assumes no interpreter code in ROM, so it generates a completely standalone program. If you have the BASIC-52 interpreter present and enabled, your program can use some of that code when you specify the **-2i** command line option. This greatly reduced program code size. To create an Assembly subroutine, use the **-sub** command line option.

A complete description of all available command line options appears on page 124.

Commonly Used Compiler Options

If you have not yet formed an opinion about which command line options are best for you, you may wish to know what other engineers prefer to use.

Most engineers who communicate with their board via serial port use one baud rate all the time. By specifying **-b***rate* on the command line, the baud rate is fixed and the program can start running immediately without having to wait for the user to send a space for auto-baud detection.

The **-v***addr* and **-u***addr* command line options are very common for specifying what block of RAM (the lower and upper limit) is allowed to be used for variables and the BASIC environment.

The **-x***addr* command line option tells your program where to go when it is done. use **-x0** to have your program automatically restart when it exits (assming your program starts at 0H). Many engineers have on-board monitors that they jump to as well.

Many engineers use the **-l** command line option to reduce program code size. This comes at a price, however, because **-l** removes all line numbering information which may be useful for debugging.

Customizing Compiler Options

When you establish a set of command line options which you prefer to use, you can customize BXC-51 to always assume them. Do this by setting the DOS environment variable called BXCFLAGS. If you SET this in your `AUTOEXEC.BAT`, then those defaults will always be used. Of course, any command line options setup in `BXCFLAGS` may be overridden on the command line. When `BXCFLAGS` is setup, BXC-51 will echo them each time the compiler starts. For example, if you have this line in your `AUTOEXEC.BAT` file:

```
SET BXCFLAGS=-v5000 -u5FFF -b9600
```

and you invoke the compiler with this command

```
BXC51 -m HELLO.BAS
```

then you first see this line output before the compiler works on your source code:

```
BXC51 -v5000 -u5FFF -b9600 -m HELLO.BAS
```

This is BXC-51's way of informing you of your setup defaults. In the case of repeated command line options (such as multiple **-b***rate* options), the rightmost option is used.

Compiling

To compile your BASIC source code, type the BXC51 command followed by the desired command line options, followed by the BASIC source file. For example, to compile `HELLO.BAS` to run at 9600 baud, type

```
BXC51 -b9600 HELLO.BAS
```

The `.BAS` file extension is optional unless you use a different extention.

If any errors are reported, you will need to alter your source code and re-compile before going further.

Upon success, a .HEX file will be created. The .HEX file is your compiled program in the Intel HEX file format. To also produce an Assembly listing, use the **-a** command line optoin which generates a .LST file. For the above example, the file `HELLO.HEX` would be generated. It is now ready to be downloaded or programmed.

Downloading & Programming

This step varies from setup to setup. The goal of this step is to get the `.HEX` file onto a chip on the target system.

If the target system has external RAM and ROM shared address space and a means of downloading to the board (using a monitor on the board and a communications package on the PC), that may be easiest. For the DS5000, the bootstrap loader does this. For other microcontrollers, you will need to obtain a monitor (such as M/DP) and a communcations package (such as Kermit or QComm). Contact Binary Technology, Inc. if you need them.

Another popular technique is to program an EEPROM with a programmer. See your programmer's instructions for further details.

Running

Once the `.HEX` file is on the target system, power on the unit and reset it. If you located your code at 0H (the default), your program runs immediately. If it does not, consult the next section, **Troubleshooting**, for possible causes. If you located your program elsewhere, you will need to use your monitor to run your code or otherwise trigger it to start.

Upon completion, your compiled program will output the message

```
Program terminated.
```

to the serial port.

When running your program over and over again to debug it into perfection, you may find it useful to use a simulator such as BXC-51 Simulator (available from Binary Technology, Inc. A simulator allows you to see all the details of how your program executes and provides an interactive means of interrogating and changing your environment as you run and re-run your program.

Troubleshooting

If your BXC-51 compiled program does not run, consult this checklist.

1. Did you specify the **-b***rate* command line option when you compiled your program. If not, your program is waiting for you to press the spacebar to determine the baud rate.

2. Do you have external RAM where you specified in the **-v***addr* command line option. Without RAM, your program cannot run.

3. Is your program in the same contiguous RAM space as your variables? If your program is located above your variable space and you did not use the **-u***addr* command line

option to set the upper limit of RAM, your program will CLEAR itself. This is not a problem when working with a PROM.

4. If your program is in RAM, is the RAM chip configured as program memory space. Assembly programs can only be executed out of the 'ROM' or program memory space.

5. Is your program located at 0H? On the 8052 microcontroller, make sure your $\overline{EA}$ pin is tied low to disable the internal ROM for BASIC-52 interpreter.

6. When mixing assembly language and BASIC, ensure that you have not relocated the stack pointer into the byte variable storage space. Also make sure that the PSW is set to RB0 when returning from your subroutine.

7. When your program starts, does it immediately output MEMORY ALLOCATION? If so, there is either no or not enough external RAM on your board. There must be 200H bytes plus enough for BASIC variables. If using the **-u***addr* command line option, make sure memory goes that high.

9. BASIC Program as Assembly Subroutine

Use the **-sub** command line option to instruct BXC-51 to create an assembly subroutine that will coexist with other assembly subroutines. Use the **-v***addr* and **-u***addr* command line options to limit the area of RAM that the BXC-51 subroutine may use. When using the **-p***addr* command line option to specify where the subroutine starts, be careful to route the appropriate interrupt vectors up to the subroutines: if your subroutine uses ONEX1, the X1 interrupt needs routing; if your subroutine uses the real time clock, the T0 interrupt needs routing (see the explanation of the **-p***addr* command line option on page 124).

To use the subroutine, CALL the address specified by the **-p***addr* command line option. When the BXC-51 code is finished running, it will RETurn back to the CALL. The stack may be severely altered by the use of byte variables so only one level of RETurn addresses can be guaranteed.

Before CALLing the subroutine, the BASIC environtment must already be initialized starting at the address specified by the **-b***addr* command line option. This includes initializing MTOP.

The **-x***addr* command line option is ignored. Whether the BXC-51 subroutine completes with an error or not, it will RETurn to the address that CALLed it.

If your BXC-51 subroutine is CALLed from the MCS BASIC-52 interpreter, you can also specify the **-2i** command line option. This will reduce the size of the BXC-51 subroutine. Read the section "BXC-51 Programs Coexisting with MCS BASIC-52" for additional comments.

Some BASIC command are not well suited for subroutines. If possible avoid command dependent upon interrupts such as CLOCK1 and ONEX1. Try to avoid commands that dynamically allocate memory: DIM, dynamic strings, and STRING. Byte variables may overlap with other subroutine's use of the stack; use DEFVAR to locate byte variables safely away from the internal stack.

Initializing a Minimum BASIC Environment

When a BASIC program compiled as a subroutine runs, it expects the BASIC environment to be already initialized. Make sure the following minimum has been initialized:

1. Clear all external RAM from RAMORG to RAMORG+200H. Store a 0 in each byte.
2. Clear all internal RAM from 8H to 4DH.
3. Set MTOP (10AH and 10BH) to the upper limit of external RAM that the subroutine may use. Copy this value to VARTOP (104H and 105H), and ST_ALL (106H and 107H).

4. If using DIM, set MT_ALL (108H and 109H) to the first byte following your allocated BASIC variables. In the Assembly listing, the label PSTART holds this address. Also set DBY(42H)=6.

5. If using any commands dependent upon the crystal value, initialize XTAL.

6. Setup the Argument Stack with DBY(9)=0FEH.

7. Setup the Control Stack with DBY(11H)=0FEH.

8. If using READ and DATA, place RESTORE at the top of the subroutine to initialize READ. (Otherwise, READ will cause unpredictable results.)

9. If using buffered serial input, place

```
SBUFFER OFF: SBUFFER ON
```

at the top of the subroutine.

You only need to initialize the subroutine once. If multiple subroutines share the same common external RAM range, then that external RAM range need only be initialized once.

Once the subroutine starts, use the CLEAR command to reset all BASIC variables. Otherwise, the BASIC variables will have the previous (or if no previous, random) values.

Converting Your Program Into a Subroutine

Using the **-sub** command line option generates only slightly different code than normal. The internal and external memory are not initialized. The board is assumed to be already initialized compatible with MCS BASIC-52. To properly initialize them, either start up the BASIC-52 interpreter, run a standalone BXC-51 compiled program, or initialize memory as outline in the section "Initializing a Minimum BASIC Environment" above. Each performs BXC-51's normal board initialization (setting up special registers, clearing memory, and system RAM initialization).

The other code difference is what your program does when it starts and when it exits. Normally, DBY(03EH) holds the internal stack pointer's holding register and typically has a value around 04DH. This value is the stack's home position and is used to prevent a runaway stack. The called subroutine needs its own stack space and a higher home position to preserve the calling program's stack (which requires additional internal stack space). So when the BXC-51 subroutine starts, it copies (onto the system stack) the old value at 03EH and replaces it with the current stack position. Upon exiting, the value of 03E is recovered and the stack position is restored so a RET will successfully return to the calling program. Because the stack's home position is not reliably known, BXC-51 subroutines should not use byte variables unless those variables are declared at specific addresses with DEFVAR.

For other variables such as floating point or integer variables, use the DEFVAR command (page 20) to share them with other subroutines or Assembly code.

When writing BASIC code which will become a subroutine, try to keep the following points in mind:

- If you want all your variables initialized to 0 when your subroutine starts, make sure you use the CLEAR command. Otherwise, they persist from CALL to CALL.

- Do NOT end your program with the RETURN statement. It will cause errors. End your program with END or STOP as your normally do. Note that if a run-time error occurs in your program, it will still return to the assembly code that called it (unless you trap errors with ONERR and use the **-e** command line option).

- Try to use ONTIME, ONEX1, and ONERR inside only one program/subroutine. It gets messy if multiple Assembly routines set up an ONTIME statement because the results are unpredictable. If both subroutines share the same variable space (have the same **-v***addr* command line option), then it will work. Note that the interrupts set up by the interpreter are different than the BXC-51 interrupts, so the ONTIME, ONEX1, and ONERR commands should not be used in BXC-51 subroutines when executing a program running in the interpreter. This is because the interpter record the line number to GOTO upon interrupt whereas the compiled code remembers the Assembly address.

- Byte variables typically start at 04EH in internal RAM. If you have multiple BXC-51 programs using byte variables, there will be overlap. If you have assembly code which uses 04EH and up, it too will overlap. To correct for this, use DEFVAR to declare byte variables at specific locations. The area above 07FH on the 8052 is ideal for this purpose.

- Floating point and integer variables must be declared in precisely the same order if two BXC-51 programs/subroutines share the same variable space (using the same **-v***addr* command line option). This can be done by initializing them at the top of your program (do not GOSUB/GOTO an initialization routine). As the variables are used, the compiler sets aside space for them. All floating point, integer, string, and array variables take up space in separate blocks in RAM. If multiple variable types are used and a different amount of one type of variable is declared in one program than the other, then the variables will not overlap properly. To verify that your variables do overlap properly, use the **-m** command line option to create a map for both programs' variables.

- You should determine how much RAM your subroutine requires and use the **-v***addr* and **-u***addr* command line options to keep your subroutine contained in that RAM space.

Remember that each compiled BXC-51 subroutine uses as much of the BXC-51 library code as it requires. If you have multiple BXC-51 subroutines, you will have multiple copies of parts of the library. If this is a problem for you, inquire about the BXC-51 Library Toolkit available from Binary Technology, Inc.

Coexisting with BASIC-52 interpreter

You can compile BXC-51 programs that safely coexist with MCS BASIC-52 if you are careful how you instruct BXC-51 to behave. Use the **-2i** command line option to instruct BXC-51 to use assembly routines out of the MCS BASIC-52 ROM as much as possible. Here are some points to look out for:

- BXC-51 assumes your board is only partially initialized and proceeds to fully initialize it. The CLEAR statement is performed. This may clear any MCS BASIC-52 program in memory. To avoid this, use the **-v***addr* and **-u***addr* command line options to limit the segment of memory being cleared. The CLEAR is necessary to guarantee that your variables are initialized to 0.

- Interrupt handling is not performed by MCS BASIC-52 while your BXC-51 program is running. Any ONEX1 or ONTIME statements executed by an MCS BASIC-52 program will cause the BXC-51 program to crash if the interrupt occurs. You may, however, use ONEX1 or ONTIME in your BXC-51 program, without problems.

- Similarly, ONERR should be used carefully. Any ONERR statement executed by an MCS BASIC-52 program will cause the BXC-51program to crash if a non-arithmetic error occurs when using **-e** command line option. An ONERR statement in your BXC-51 program will trap most, but not all errors since BXC-51 uses all the MCS BASIC-52 floating point arithmetic routines, all floating point errors are handled by MCS BASIC-52.

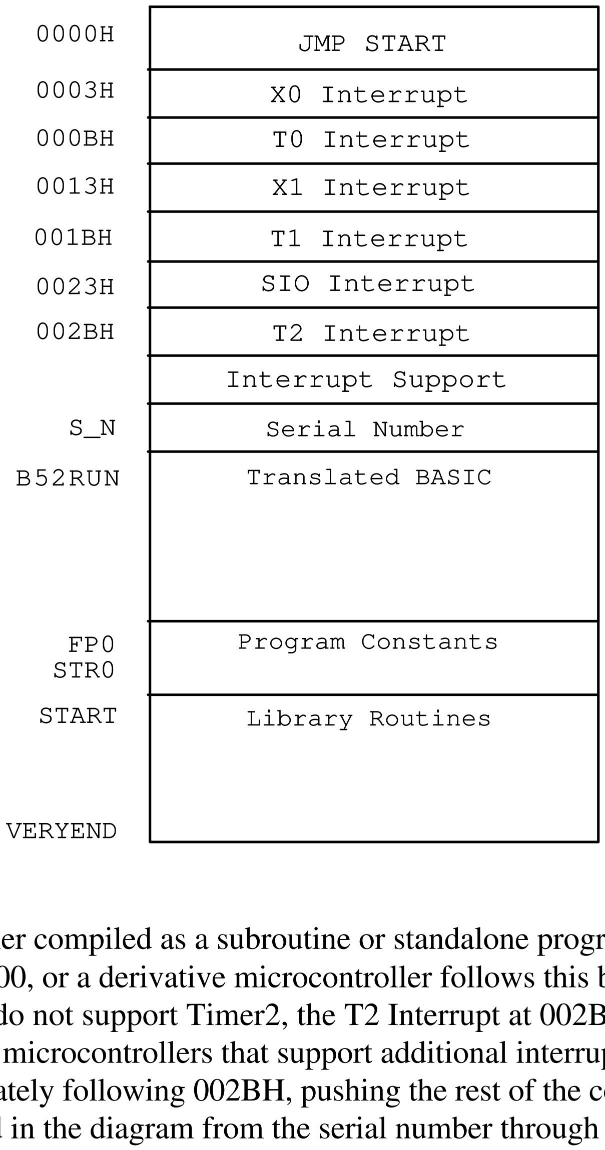
Otherwise, BXC-51 uses nearly all of the system variables in external RAM from 0H to 200H for the same purposes as MCS BASIC-52. Some non-run-time variables used by MCS BASIC-52 have been given other uses in BXC-51. See the next section for system variable architecture.

10. Compiled Program Structure

BXC-51 compiled programs use internal RAM and external RAM highly similar to MCS BASIC-52 interpreter. However, the program code is completely different. In the interpreter, the program is located in RAM and stored as a linked list of tokens evaluated and re-evaluated at run-time. The compiled code is 100% Assembly code. In the following sections, the architecture of the Assembly code, internal RAM, and external RAM is described.

Code Memory Architecture

The following diagram shows the basic architecture of your compiled code. The diagram assumes that the beginning of program code (ROMORG) is set to 0000H. If **-p***addr* was specified, add *addr* to the offset constants at left.

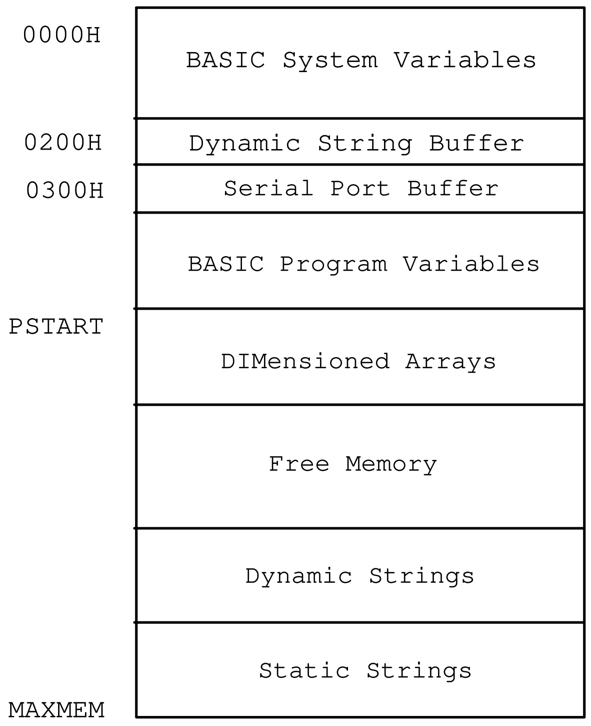


Every program, whether compiled as a subroutine or standalone program, whether compiled for the 8052, 8051, DS5000, or a derivative microcontroller follows this basic architecture. In microcontrollers that do not support Timer2, the T2 Interrupt at 002BH is not present. On the other hand, derivative microcontrollers that support additional interrupts insert those interrupt service hooks immediately following 002BH, pushing the rest of the code further up in memory. Constants are not used in the diagram from the serial number through to the library routines because the address of them varies depending upon command line options and BASIC source code. Instead, the diagram shows the Assembly labels which begin at each address. These labels may be used by in-line Assembly code.

If your code starts at a memory location other than 0H, the interrupt handlers will not be active unless you route stray interrupts upto your program. If you use ONEX1, external interrupt 1 (X1) must be routed up to ROMORG+13H. If you use the real time clock (CLOCK1/CLOCK0), timer overflow 0 (T0) interrupt must be routed up to ROMORG+1BH. If you are use derivative microcontroller ON*intr* commands, the respecitive interrupt must be routed as well. Interrupt rerouting is not necessary if these interrupt causing commands are not used.

External Memory Architecture

The BASIC environment occupies the first 200H bytes of external RAM. By default, the beginning of external RAM (RAMORG) is 0000H, however it may be relocated using the **-v***addr* command line option. Here's a basic diagram of external RAM architecure:



External RAM is designed as a stack of memory blocks. When one memory block is not needed, such as the Serial Port Buffer or Dynamic String Buffer, they occupy 0 bytes of external RAM, leaving room for the other memory blocks. Programs that use neither dynamic strings or serial port buffering have BASIC program variables starting at 200H. All the blocks at and above PSTART are created dynamically at run-time, as needed. The MEMORY ALLOCATION error typically displays when the blocks around the Free Memory block attempt to push into it so much that there is no more free memory left (by dimensioning an array, creating too many static strings, or storing too much text in dynamic strings.) The MAXMEM Assembly label is only created when the **-u***addr* command line option was specified. Otherwise, the value of MTOP is used (which is determined at program startup).

Note that if the **-2i** command line option is used, the BASIC System Variables are locked in at 0000H while the **-v***addr* command line option dictates where the Dynamic String Buffer begins, which is typically not 200H.

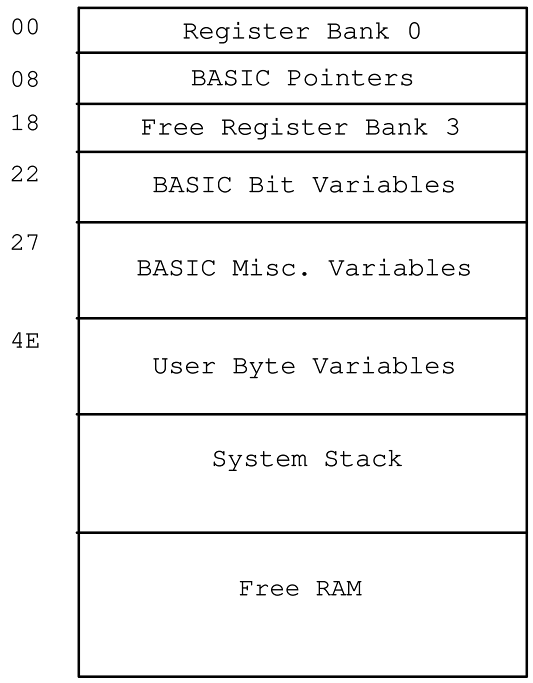
The BASIC System Variables are required for maintaining the BASIC environment which includes the Argument and Control stacks. Here is a detailed memory map of it:

Address (Hex)	Assembly Label	Description
000-001		Reserved for future use
002-003	ERRLN	Line number where the last error occurred
004	IBCNT	The length of input line
005-006	IBLN	Currently executing line number
007-050	IBUF	Input buffer
051-05D	CONVT	Floating point to integer conversion scratch space
05E		Reserved for future use
05F		Warm restart flag, if set to A5H (see **-w** command line option, page 123)
060-0FB		Control Strack (C-STACK)
0FF		Reserved for future use
100	GTB	Last character received, next char for GET. When a serial port buffer is on, this location contains the number of characters presently buffered.
101	ERRLOC	The error number of the last error to occur (see ONERR on page 28)
102-103	ERRNUM	The Assembly address of where to go for ONERR
104-105	VARTOP	Beginning of static string memory (end of dynamic strings)
106-107	ST_ALL	Bottom of dynamic string memory (end of free RAM)
108-109	MT_ALL	Top of DIMensioned memory (beginning of free RAM)
10A-10B	MEMTOP	Topmost address in RAM that BASIC may use (MTOP)
10C-10D	RCELL	Random number generator seed
10E-113	CXTAL-5	Crystal value (see XTAL on page 61)
114-11F		Scratch area for two temporary floating point numbers
120-121	INTLOC	The Assembly address of where to go for ONEX1
122-123	STR_AL	Size of allocated static string space
124-125	SPV	Serial port baud rate timer information for printer output
126-127	TIV	Timer interrupt vector - where to go for ONTIME
128-129	PROGS	Regular time-out for PROM programmer
12A-12B	IPROGS	Intelligent time-out for PROM programmer
12C		Reserved for future use
12D-1F8		Argument Stack (A-STACK)
1F9	VCOUNT	Count of pending interrupts (for derivative microcontrollers)

1FA-1FB	VTABLEP	Address of interrupt table for derivative microcontrollers
1FC-1FF		Reserved for future use

Internal Memory Architecture

The BASIC environment is contained partly in internal memory as well. Register banks 1 and 2 as well as memory locations 22H through 4DH are part of the BASIC environment. Register bank 0 is used as the primary register bank for subroutine parameters and temporary results. Register bank 3 is reserved for user Assembly applications. The following diagram demonstrates the basic architecture of internal RAM.



The System Stack is located immediately after the BASIC byte variables. If no byte variables are used, the System Stack begins at 4EH. Free RAM is all the memory above the System Stack. In general, the System Stack should have at least 28H bytes of RAM to use.

The following is a detailed description of how BASIC uses internal memory.

Address (Hex)	Assembly Label	Description
00-07	R0-R7	Register Bank 0, the default register bank
08	TXAL	Control Stack jump address, low
09	ASTKA	Top of Argument Stack position
0A	TXAH	Control Stack jump address, high
0B-0F	TEMP1-TEMP5	5 user-defined bytes
10	RTXAL	DATA item pointer, used by READ command, low byte

Address (Hex)	Assembly Label	Description
11	CSTKA	Top of Control Stack position
12	RTXAH	DATA item pointer, high
13-14		Reserved for interpreter - beginning of BASIC program
15	NULLCT	Count of NULs to follow carriage return
16	PHEAD	Print head position of output, used for TABbing
17	FORMAT	Floating point number output format (set by USING)
18-1F		Register Bank 3 - not used (reserved for user)
20-21		Not used (reserved for user)
22H.0	OTS	Flag: ONTIME vector setup
22H.1	INPROG	Flag: ONEX1 interrupt in progress
22H.2	INTBIT	Flag: Interrupts pending
22H.3	ON_ERR	Flag: ONERR executing
22H.4	OTI	Flag: ONTIME interrupt routine in progress
22H.5	LINEB	Reserved for future use
22H.6	INTPEN	Flag: External interrupt 1 detected
22H.7	CONB	Reserved for future use
23H.0	GTRD	Flag: Set when GET character ready
23H.1	LPB	Flag: Servicing PRINT@
23H.2	CKS_B	Flag: Trap timer1 interrupts, send them to 2088H
23H.3	COB	Flag: Console output to line printer
23H.4	COUB	Flag: Console output is user defined
23H.5	MUL_LIMIT_ CASE	Flag: Floating point underflow/overflow at limit detected
23H.6	CIUB	Flag: Console input is user defined
23H.7	SPINT	Flag: Trap serial port interrupt, send them to 2050H
24H.0	STOPBIT	Reserved for future use
24H.1	U_IDL	Flag: User idle bit to quit IDLE
24H.2	INP_B	Flag: INPUT command being executed
24H.3	FPTRAP	Flag: Trap floating point output (used by STR$())
24H.4	ARGF	Reserved for future use
24H.5	RETBIT	Flag: RETI command being executed
24H.6	I_T0	Flag: Trap external interrupt 0, send them to 2040H
24H.7	UPB	Flag: When set, PRINT@ will call 4030H
25H.0	TRONBIT	Flag: Set when TRACE1 executed for tracing
25H.1	BUFOFF	Flag: Set when SBUFFER is disabled
25H.2	UBIT	Flag: Used by DIM command
25H.3	ISAV	Flag: Temporary bit for interrupt status
25H.4	BO	Flag: When set, use output routine at 2040H
25H.5	XBIT	Reserved for future use
25H.6	C_BIT	Flag: Set when CLOCK1 executed for updating TIME
25H.7	SBUFSIL	Flag: Set for no echo on serial input
26H.0	NO_C	Flag: Set to ignore Ctrl-C input as user interrupt

Address (Hex)	Assembly Label	Description
26H.1	DRQ	Flag: Set to enable fake DMA
26H.2	BI	Flag: When set, use character input routine at 2068H
26H.3	INTELB	Flag: Set for intelligent PROM programming
26H.4	C0ORX1	Flag: When set, text from ROM being printed (not RAM)
26H.5	CNT_S	Flag: Set when Ctrl-S received (suspends output)
26H.6	ZSURP	Flag: Set to suppress leading zeroes during hex output
26H.7	HMODE	Flag: Set for hexadecimal output of numbers
27		Reserved for future use
28-3D		Floating point calculation scratch area
3E	SPSAV	System Stack home position (when empty)
3F	S_LEN	Static string length for all static strings
40-41	T_HH,T_LL	Timer 1 reload value
42	ARRSIZ	Used by DIM for unit size
43	XP2	Used on DS5000 as temporary holding of P2
44		Reserved for future use
45-46	MT1,MT2	Scratch area for transcendental functions
47	MILLIV	Millisecond counter for TIME
48-49	TVH,TVL	Seconds counter for TIME
4A	SAVE_T	Timer 0 reload value for TIME
4B-4C	SP_H,SP_L	Interrupt time value which causes ONTIME interrupt
4D		Reserved for future use
4E-FF		User Byte Variables System Stack

Program Initialization

This section is useful if you are using the **-i** or **-r** command line option. It will aid you in understanding what is initialized by the time your assembly routine is called. The following initialization is not performed when the **-sub** command line option is specified.

1. When your program first starts up, it immediately initializes the SCON, TMOD, TCON, and T2CON registers. On the 8051/31, DS5000, and derivative microcontrollers, the registers are initialized to:

   SCON - 5AH
   TMOD - 21H
   TCON - 84H
   T2CON - 00H

   On the 8052/32, the special function registers are initialized to:

   SCON - 5AH
   TMOD - 10H
   TCON - 54H

T2CON - 34H

See the table below for timer/counter usage by the 8051/31, DS5000 (**-5** option), and 8052/32 (**-2** option) initializations. Derivative microcontrollers use the timer/counters the same as the 8051/31.

Resource	8051/31	DS5000	8052/32
Console baud rate	TIMER1	TIMER1	TIMER2
Printer baud rate	N/A	N/A	TIMER1
CLOCK0/CLOCK1	TIMER0	TIMER0	TIMER0
PGM	TIMER0	TIMER0	TIMER1
PWM	TIMER0	TIMER0	TIMER1

Table 1. Timer/Counter Usage

2. If the **-r** command line option is specified, the location 2001H in code memory is compared to the value AAH. If they are equal, a CALL to 2090H occurs.

3. The internal memory is cleared. On the 8051/31 DS5000, and derivative microcontrollers, internal memory is from 0H to 7FH. On the 8052/32, internal memory is from 0H to FFH. If the **-w** command line option is specified, the internal memory is not cleared.

4. The internal stack pointer is set to 4EH (or higher if you have byte variables). To change this in BASIC, use DBY(3EH) to set the new pointer value and execute the CLEARS statement.

5. If the **-b***rate* option is specified, the serial port baud rate is initialized by setting up the appropriate reload value.

6. If the **-i***addr* command line option is specified, the location XBY(RAMORG+5FH) is tested. If the value found is A5H, the address specified after the **-i***addr* option is CALLed. If you want your initialization routine to start the BASIC program, execute a JMP to B52RUN.

7. Initialize external memory starting at the beginning of the variable space, probing to see how far it extends. If the **-u***addr* command line option is specified, probing will not be performed (but will be CLEARed later). If the upper limit of RAM is lower than the value the address specified with the **-u***addr* value given, a MEMORY ALLOCATION error will result during step (8) next. MTOP is set to the determined upper limit of RAM (or address from **-u***addr*).

8. Set up the floating point Argument Stack and initialize the default crystal value of XTAL to 11.0592 MHz.

9. Set the baud rate. Wait for the user to hit the space character to determine the baud. This step will be ignored if the **-b***rate* command line option was specified.

10. If the **-g** command line option was specified, then this message displays:

    BXC-51 Compiled program *filename*

    where *filename* is your source file's name.

11. The CLEAR command is executed to clear external RAM up to MTOP.

12. Any BXL initialization routines present are CALLed.

13. Execution of the first line of BASIC program begins.

Program Termination

This section descibes what occurs when a program terminates due to the END command, STOP command, or run-time error.

1. Any BXL termination routines present are CALLed.

2. The following message displays:

```
Program Terminated.
```

3. Any interrupts enabled by the compiled program are disabled. This includes the real time clock, external interrupt 1, and any derivative microcontroller interrupts enabled using the ENABLE command.

4. An Assembly JMP instruction jumps the the exit address specified by the **-x***addr* command line option. If no **-x***addr* command line option is specified, an infinite loop begins.

11. BXC-51 Programming

Although programming in MCS BASIC-52 is highly simiar to programming with BXC-51 there are extra considerations when working with the compiler. The following sections cover programming topics specific to the compiler.

Reducing Program Code Size

When developing a program, it may be permissible to have large programs, but finished products or memory constraints may necessitate reducing your code size. Here are some suggestions:

- do not use the **-g** command line option. Although handy for debugging, it generates extra code.

- use the **-l** command line option. Without the burden of keeping track of your source code lines, this can save a lot of space in long BASIC programs.

- use the **-b***rate* command line option if possible. On the 8031/51, DS5000, and derivative microcontrollers, a lot of code goes into determining the baud rate; on the 8032/8052, only a little code is generated. However, if you know the baud rate, no extra code is generated.

- use the **-c0** command line option if you are not using the UI1, U01, or PRINT@ commands, or if you don't care about user console I/O and stray interrupts. Extra code is generated to accommodate the vectors starting at 4000H.

- use BXC-51 Version 3 or later. Starting with Version 3, BXC-51 puts in extra effort to generate only code for what you need and still provide the benefits of the BASIC programming environment (such as Ctrl-C, Ctrl-S, source line tracking, and error reporting) thus keeping programs small.

- if you have an 8052 with MCS BASIC-52 enabled, use the **-2i** command line option. This greatly reduces the size of the library that your BXC-51 program depends upon.

- if you are using a DS5000 CPU and you are not using any on-chip 'external RAM', do not bother with the **-5** command line option, use the **-1** (one, not L) command line option.

Speeding Up Run-Time Execution

Follow these tips to speed up your program at run-time.

- use integer array indices, not floating point variables. Floating point calculations are much slower than integer calculations.

- avoid using FOR, DO; implement them using GOTO/IF. For example:

```
10 REM DO
20   A=A+1
30 IF A<10 THEN 10: REM WHILE A<10
```

- unlike the intepreter, GOTO and GOSUB are fast. There is no need to place subroutines at the beginning of your program for speed. Your compiled program will GOSUB the first line of code just as fast as the last line.

- do not use dynamic strings. Although convenient, handy, and powerful for text manipulation, they are very slow because they use memory so dynamically.

- use **-l** (L, not one) command line option; or, alternatively, do not use line number except where needed. Each line number generates code to note the line number. Without line numbers, no time is wasted recording line numbers.

- initialize an array instead of using READ/DATA. Assign the values directly to the array elements instead of using READ to initialize them. The source code will be bulkier, but faster.

- avoid floating point calculation; use integer calculations as much as possible. For example, if you need to keep track of voltage levels from .05 to 5.00, consider using a fixed decimal number as an integer variable with values from 5 yo 600. For output pruposes, just divide by 100.

Optional Integer Expressions

In many places in BASIC-52, the compiler expects an integer expression, but a floating point expression is acceptable. These are optional integer expressions. If you provide an expression with only integer data type variables, values, and functions, BXC-51 will calculate it fast. However if there are any floating point data type variables, values, or functions used, then the whole expression is treated as a floating point expression and then converted to integer. Optional integer expressions are used as:

- array and string indexing (e.g., A(I%) is faster than A(I) because I% is processed faster than I.)

- parameters to functions: TAB(), SPC(), CHR(), ASC(), DBY(), CBY(), XBY(). The parameters to these functions are integers.

- parameters to commands: LD@, ST@, IF-THEN, ONTIME, PWM, STRING, WHILE, UNTIL. As necessary, each of these commands reduces its parameter to an integer before executing.

- value for special variable assignments: IE, IP, TCON, and other special variables, except for XTAL and TIME.

One of the most common optional integer expressions that slows calculation is the use of a floating point variable.

Optimized Integer Expressions

Starting in BXC-51 Version 3, integer expressions are optimized for speed and code size. The following strategy is used:

- constant expressions are calculated at compile time (e.g., `A%= 2 + 5` is converted to `A%= 7` before compiling).
- expressions in parentheses are calculated first. This reduces integer stack clutter.
- terms are rearranged to reduce register transfers (e.g., `A%+ B%* C%` will be performed by `B%* C%+ A%`).

Cross-Reference Information

Cross-reference information can tell you which BASIC lines refer to other lines as well as which BASIC variables are used where in your program. The cross-reference information is produced by the assembler. Compile your program using the normal command line options with this new option: **-a-c**. This option tells the compiler to add **-c** as a command line option to the assembler. When the assembler finishes, you will have a .LST file with cross-reference information at the end.

The cross-reference information will be the very last section of the .LST file. The information is displayed like this:

```
ln30 = 40AA       88   140
```

Which displays the label name, its address, and the assembly listing lines where the label is referenced. In this case, we see that line number 30 is located at 40AAH in memory and it is referenced at line 88 and line 140 of the listing (the lines in the listing are numbered). One of those two lines is where LN30 is defined and the other is where another line performed either a GOTO or GOSUB to the line.

In the assembly listing, all labels beginning with "`ln`" refer to line numbers with the line number as the rest of the label. All lables beginning with "`ll_`" are line labels with the BASIC label name as the rest of the assembly label, for example `ll_lift` is the assembly label for the BASIC `{LIFT}` line label.

Similarly, BASIC variables can be cross-referenced when you know how to identify their assembly label equivalents. Here is a list of assembly label prefixes and the respective BASIC variable type:

Prefix	BASIC Variable Type

V_	Floating point variable
IV_	Integer variable
BV_	Byte variable
SV_	Dynamic string variable
AV_	Floating point array variable
IAV_	Integer array variable
BAV_	Byte array variable
SAV_	Dynamic string array variable

For best understanding, you should not use the **-l** command line option when generating a cross-reference listing since it strips out of the listing file the comments which show the BASIC source text.

Line Renumbering

A line renumbering utility is provided on your BXC-51 disk to help you maintain your BASIC code and make it easier for you to share sections of code between your BASIC programs by only renumbering certain ranges of lines.

In its simplest use, RENUM will renumber all the lines of your BXC-51 program. You may specify the desired starting line number and the increment between lines. For example, if you want the first line of BASIC code to be line 1000 and all successive lines to be incremented by 10, and your program file is PROGRAM.BAS, type:

```
RENUM PROGRAM.BAS 1000 10
```

at the DOS prompt. Your file will be completely renumbered. Any references to the old line numbers will be appropriately changed. If you do not specify a starting line or increment, each defaults to 10.

In a more advanced use, you can choose to renumber just a portion of your program. If you have a range of lines between 1000 and 1100 and you want them to be renumbered starting at 2000 and incremented by 5, type:

```
RENUM PROGRAM.BAS 1000-1100 2000 5
```

RENUM does not let you move lines of BASIC around to different parts of the file. It keeps your file in order and only renumbers some lines. If you attempt to renumber the range of lines to an existing line number, RENUM reports an error and stops.

National Language Support

Although BXC-51 outputs English text by default, it is not limited to English. Upon start up, BXC-51 attempts to read the file `BXC51.LSF`. If absent, the default English messages are used. If present, the messages in that file are used instead of English ones. Each message has a number. When BXC-51 needs to output a message, it looks up the message by number. If the `BXC51.LSF` file is present, BXC-51 seeks the message number. The text following the found message number will be displayed instead of the English text. (If the sought message number is missing, the default English text is used.)

The following is the default contents of `BXC51.LSF` if such a file was needed for an English version of the BASIC Compiler. Translate each English message to your language, store them in `BXC51.LSF`, and see BXC-51 output messages which you do not need to translate. Explanations of the error messages begins on page 134.

```
1 Variable space adjusted to
2 Unknown command line option
3 You must first INSTALL BXC51 before running it
4 bytes of RAM is insufficient space
5 Use the -p flag to specify a ROM address above 2000H when using -2i
6 BASIC file must be last parameter
7 Cannot open
8 Insufficient memory
9 Cannot open BXC51.LIB for library routines
10 Cannot open %s for output
11 Out of memory
12 Invalid BASIC line
13 Unable to complete assembly
14 lines successfully compiled
15 Cannot write memory map file
16 defaulting byte array
17 Line label
18 Unknown line numbers:
19 Unknown line labels:
20 Your version of BXC51.LIB does not match BXC51.EXE
21 Please update BXC51.LIB before proceeding.
22 Undefined symbols:
23 Warning: TRACE is useless when -l option is on
24 Warning: vectored CALL statement to 100H+
25 Warning: CALL's made to BCS-52 Interpreter ROM are not valid
26 Warning: Serial buffer cannot exceed 253 characters.  Truncated.
27 Warning: variable already DEFined
28 Warning:  byte variables below 4EH overlap with BASIC system variables
100 8052-specific operation has no effect on 8051
101 BAUD refers to printer serial port which is not available on 8051
102 PI is not an integer constant, using 3
103 XBY%/CBY%/DBY%: integer array name might be confused with special byte
   function XBY#/DBY#/CBY#
120 ELSE without a preceding IF
121 Illegal FOR index variable
122 You cannot change ROM
123 Too many parenthesis or expression too complicated
124 RND is not an integer function
125 XTAL is not an integer variable
126 TIME is not an integer variable
127 bit address too high (after .)
128 string not allowed here, expecting a number variable
129 out of byte variable memory space
130 byte array redimensioned to different size
131 byte variable defined out of byte space (0...127)
132 control register not in correct range (128...255)
133 memory mapped variables cannot be DIMensioned
134 variable is not bit addressable
135 Expecting a line number or label
136 Unrecognizable command
137 Expecting an integer expression
138 ) expected
139 Integer constant expected
```

```
140 ( expected
141 Variable expected
142 , expected
143 DATA item expected
144 Array variable expected
145 H expected for hexadecimal constant
146 Expression expected
147 = expected
148 TO expected
149 GOTO or GOSUB expected
150 # expected
151 0 or 1 expected
152 Another expression term expected
153 String expression expected
154 : expected
155 @ expected
156 Invalid DEF type
157 Identifier/Name expected
158 String expression expected
159 + expected
160 $ expected
161 } expected
162 Duplicate line label
163 Duplicate line number
164 Too many embedded IFs
165 ENDIF without a preceding IF
166 Expecting THEN
167 Exponentiation not allowed in integer/byte expression
168 % expected
169 Interrupt name expected (verify correct -t command line option)
200 Internal error %d\n
```

Text notes such as `%s` or `%d` are placeholders for where a string or integer is inserted in the message. The compiler inserts a string or integer there to make the message more meaningful.

8051 Derivative Microcontroller Support

With BXC-51 Version 5.0, derivative microcontrollers of the 8051 family are supported. As shipped, these microcontrollers are immediately supported: 8xC550 and 8xC552. Other microncontrollers are supported, too, but they require some configuration on your part.

To specify a derivative microntroller, use the **-t***cpu* command line option. Specify the microcontroller name as *cpu*. For example, **-t550** for the 8xC550.

Support for derivative microcontrollers comes in these forms:

- support for additional or different interrupts
- support for byte special function registers
- support for 2 byte special function registers
- support for internal RAM size configuration (128 or 256 bytes)
- support for an external library of commands and initialization code (via BXL)
- support for using DPTR exclusively for external RAM references (not Port 2)

For each derivative microcontroller, BXC-51 needs to know how the derivative microcontroller differs from the 8051 family with respect to the items listed above. To find that configuration, BXC-51 first looks in the file named *cpu*.CPU, e.g. `550.CPU`. If no such file exists, BXC-51 then checks in the file `BXC51.CPU`. In both case, the compiler searches for a line beginning with `$`*cpu*. Once found, the compiler reads the text following it to understand the configuration. Alternate nicknames of the derivative microcontroller may be used as well, for example:

```
$550 C550
```

The example allows a user to specify either **-t550** or **-tC550** as a command line option. Any number of nicknames may be used. All text up to the next line that begins with `$` will be read as configuation for the microcontroller. The details of configuration are explained in the next section.

Once configured, additional commands and new special function variables are available in your BASIC program. For each interrupt configured, three BASIC commands appear:

> ON*intr* *line*
> ENABLE *intr*
> DISABLE *intr*

Where *intr* is the name of the interrupt, e.g. T2, AD, WD, etc. The ON*intr* command is like the ONEX1 command. When the interrupt occurs, a GOSUB to the subroutine at *line* happens. That subroutine must return using RETI. The ENABLE command is like the CLOCK1 command and the DISABLE command is like the CLOCK0 command. No interrupts will occur until they are enabled. No interrupts will occur when disabled.

For every special function register configured, a new byte and integer variable is created using its name. For example, if the special function register ADCON is configured, then ADCON# and ADCON% may be used with meaning inside your BASIC source code. The configuration determines whether ADCON is a one or two byte value, but it may be used as both an integer and byte variable.

As an example of additional commands and special variables, see the "Support for 8xC550 section" below.

Derivative Microcontroller Configuration Commands

Configure support for additional derivative microcontrollers by creating a *cpu*.CPU file or adding to the exsiting BXC51.CPU file. Begin the configuration section by placing the microcontroller name (perfably short like 552) preceded by a dollar sign. For example:

```
$552
```

Multiple names may be specified if they are separated by spaces. Configuration continues until the next configuration section that begins with a microcontroller name. In the file *cpu*.CPU, there can be only one section and it must begin that microcontroller's name:

```
$cpu
```

The configuration section is line oriented; one configuration command per line. Parameters to a configuration command are separated by spaces or tabs. Configuration commands include: EQU, INT, SFR, SFR2, RAM, DPTR, and BXL. Commands may be specified in upper or lower case. Comment lines begin with the pound sign, #, are ignored.

EQU *address name*

Declare an Assembly equate. The text immediately generates this Assembly code:

```
name EQU  0addressH
```

Equates are particularly handy so that other configuration commands can refer to *name* instead of a hexadecimal address. For example, an interrupt bit of ADCON:3 is more meaningful than 0C5H:3 when ADCON has been equated to C5H.

Any number of equates may be setup here. Typically, an equate is setup for each special function register referred to by the INT configuration command. These equates are *not* accessible to BASIC source code; they only support this derivative microcontroller at the Assembly level.

INT *address name control-bit intr-bit* [*vec-addr*]

Configure an interrupt. By default, BXC-51 assumes the following configuration:

```
INT  03  X0   N/A     IE.0
INT  0B  T0   TCON.4  IE.1
INT  13  X1   N/A     IE.2
INT  1B  T1   TCON.6  IE.3
INT  23  SIO  N/A     IE.4
```

For the 8052 microcontroller, this is additionally assumed:

```
INT  2B  T2   T2CON.2 IE.5
```

These defaults may be overridden. However, overriding SIO and T0 may disable functionality needed in the BASIC environment.

The address of the interrupt, *vec-addr*, is specified in hexadecimal. The name of the interrupt, *name*, is used to create three new BASIC commands - ON*name* (see page 30), ENABLE *name* (see page 22), and DISABLE *name* (see page 21) - for allowing a BASIC surboutine to handle the interrupt.

The control bit, *control-bit*, is what enables and diables the function that causes the interrupt. For example, the TCON.4 bit is the run control bit for Timer 0. If no control bit (or no single bit controls the function), then use `N/A` instead. This signals the compiler to not allow the ENABLE or DISABLE commands for this interrupt. If the control bit belongs to a register that is not bit addressable, then use a colon instead of a period between the register name and the bit position. For example, ADCON:3 will set bit 3 of ADCON, but it will not use the SETB Assembly instruction; instead it will use ORL. For example, T2CON.2 generates

```
SETB T2CON.2
```

to enable the interrupt. For ADCON:3, the

```
      ORL  ADCON,#8
```

instruction is used.

The interrupt bit, *intr-bit*, is used to enable or disable the interrupt. For example, IE.1 enables the Timer 0 overflow interrupt. If you do not wish the user to handle interrupts in BASIC, then set this bit to N/A. This signals the compiler to not allow the ON*name* command. Like the control bit, if the interrupt bit belongs to a register that is not bit addressable, then use a colon, instead of a period between the register name and the bit position.

The address to vector to when the interrupt occurs, *vec-addr*, is optional. If present, the compiler does not allow the ON*name* command. Instead, the compiler will JMP to the vector address provided. The vector address may be any valid Assembly address, whether it is an absolute address or label. If you are providing BASIC extensions through a BXL, you may specify a label in that BXL's initialization section.

SFR *address name*
Specify a special function register that may be queried and set in BASIC source code. The address is in internal RAM. The address is assumed to be between 80H and FFH, but it may be any value from 0 to FFH. A byte variable and integer variable are created by the name provided for use in the BASIC source code as *name*# and *name%*. This is similar to the DEFCTRL command for byte variables (see page 20). The BASIC source code may query the register contents or assign a new value to it. For example, to allow the user access to the ADCON special function register, use this configuration command:

```
      SFR  C5  ADCON
```

Consequently, this BASIC source uses it:

```
      10 ADCON#.3 = 1
      20 PRINT ADCON#
```

SFR2 *addressH addressL name*
Specify a special function register pair that may be queried and set in BASIC source code. The high and low byte address of the pair must be specified. A byte variable and integer variable are created by the name provided for use in the BASIC source code as *name*# and *name%*. The BASIC source code may query the register contents or assign a new value to it. For example, to allow the user access to the CT0 special function register, use this configuration command:

```
      SFR2  CC  AC  CT0
```

Consequently, this BASIC source code uses it:

```
      10 PRINT CT0%
      20 CT0%= 56BH
```

`RAM` *`size`*

The 8051 microcontroller contains only 128 bytes of internal RAM. All derivative microcontrollers are assumed to have the same unless this command is specified. The only two sizes supported are 128 and 256. When more internal RAM is available, the compiler allows the BASIC program to have more byte variables.

`DPTR [ONLY]`

The 8051 microcontroller uses both the DPTR and P2 for accessing external RAM. Some microcontrollers, however, have on-chip 'external' RAM which can only be accessed via DPTR. This command tells the compiler that all external RAM should be accessed via DPTR. Without this command, the generated code will access both the on-chip and external RAM at different times. The running program will become confused by this. The keyword ONLY is optional.

`BXL` *`filename`*`.BXL`

Specify the BXL to automatically include. By including a BXL, you may provide microcontroller initialization code, program termination code, and setup interrupt handlers with vector addresses in the BXL. Additionally, you may make extensions to the BASIC language specifically supporting the derivative microcontroller.

For example, setting up and feeding a watchdog timer requires accurate Assembly timing which cannot be achieved with a regular BASIC program. The commands to support this could be coded by you (or may have been coded by your dealer) so they are easily accessible in BASIC source code through a new command.

Support for 8xC550

As shipped, BXC-51 supports the 8xC550 microcontroller. When the **-t550** or **-tC550** command line options is specified, the following extensions are made to BASIC and may be used in your program:

`ONAD` *`line`*

Specify the subroutine to execute upon completion of A/D conversion (when A/D interrupt is detected). When an A/D interrupt occurs, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When an A/D interrupt occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONAD statement's interrupt processing ability. Without the ONAD statement, BXC-51 ignores the A/D conversion complete interrupt.

`DISABLE AD`

This command attempts to reset the A/D conversion process, but it cannot be reset by software. This command is useless and should not be used.

`ENABLE AD`

Start an A/D conversion. Upon completion, an A/D interrupt is generated (see the ONAD command above). Each time this command is executed, A/D conversion begins again. If A/D conversion is in progress, the command is ignored (it does not restart the A/D conversion process). Upon completion the A/D converter is automatically disabled, making it superfluous to use the DISABLE AD command.

ADAT#
Use this byte variable to query or change the value of the A/D Data register. This register contains the output of the A/D conversion. Use as a special byte variable.

ADCON#
Use this byte variable to query or change the value of the A/D Control register. This register controls the A/D conversion. Use as a special byte variable.

WDCON#
Use this byte variable to set or query the value of the Watchdog Control register. Use as a special byte variable.

WDL#
Use this byte to set or query the Watchdog Reload register value. Use as a special byte variable.

WFEED1#
Use this byte to query the Watchdog Timer Feed 1 register. To feed the Watchdog, a very precise series of assembly instructions must be executed where timing is critical:

```
CLR  EA
MOV  BV_WFEED1,#0A5H
MOV  BV_WFEED2,#05AH
SETB EA
```

WFEED2#
Use this byte to query the Watchdog Timer Feed 2 register. Use as a special byte variable.

Support for 8xC552

As shipped, BXC-51 supports the 8xC552 microcontroller. When the **-t552** or **-tC552** command line options is specified, the following extensions are made to BASIC and may be used in your program:

ONADC *line*
Specify the subroutine to execute upon completion of A/D Conversion (when A/D interrupt is detected). When an A/D interrupt occurs, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When an A/D conversion complete interrupt occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONADC statement's interrupt processing ability. Without the ONADC statement, BXC-51 ignores the A/D conversion complete interrupt.

DISABLE ADC
This command attempts to reset the A/D conversion process, but it cannot be reset by software. This command is useless and should not be used.

ENABLE ADC
Start an A/D conversion. Upon completion, an A/D interrupt is generated (see the ONADC command above). Each time this command is executed, A/D conversion begins again. If A/D conversion is in progress, the command is ignored (it does not restart the A/D conversion process). Upon completion the A/D converter is automatically disabled, making it superfluous to use the DISABLE ADC command.

ONCT*n* *line*
Specify the subroutine to execute when capture interrupt triggered for capture register *n*. There are 4 capture registers, ranging from 0 to 3. Should a capture interrupt occur, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When a capture interrupt occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONCT*n* statement's interrupt processing ability. Without the ONCT*n* statement, BXC-51 ignores capture interrupts for capture register *n*.

DISABLE CT*n*
Disable capture interrupt for capture register *n*, where *n* may range from 0 to 3, for a rising edge. The falling edge interrupt is not affected. The appropriate CTCON register bit will need to be cleared to also clear the falling edge interrupt. (See note in ENABLE CT*n* command below.)

ENABLE CTn
Enable capture interrupt for capture register *n*, where *n* may range from 0 to 3, when a rising edge on CT0I is detected. When the interrupt occurs, it will generate a capture interrupt for register *n* which can be serviced by the BASIC routine setup using ONCT*n* command above.

Note that this command only controls the *rising* edge interrupt for a capture (by setting the CTP*n* bit in CTCON). If you wish to control the *falling* edge interrupt, you may set the CTN*n* bit in CTCON yourself, e.g. CTCON#.1=1, or you may change the compiler default. To change the compiler default, you will need to alter the `BXC51.CPU` file. Find the section beginning with $552, find the interrupt for CT*n* and change the bit listed from CTCON:*m* to CTCON:*o* where *m*=*n**2 and *o*=*n**2+1. See documentation for CTCON for an explanation of the bits. Once the default is changed, the ENABLE and DISABLE commands will control the falling edge interrupt rather than the rising edge interrupt.

ONCM*n* *line*
Specify the subroutine to execute when compare interrupt triggered for compare register *n*. There are 3 compare registers, ranging from 0 to 2. A compare interrupt occurs when the compare register matches the value of Timer 2 (and the interrupt is enabled). Should a compare interrupt occur, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When a compare interrupt occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONCN*n* statement's interrupt processing ability. Without the ONCN*n* statement, BXC-51 ignores compare interrupts for compare register *n*.

Note that this is a very inefficient way of timing in BASIC. Due to the overhead of BASIC and the relatively infrequent checks for interrupts, only slow timer rates are feasible. To be handled properly, Assembly lanugage should be used to service the interrupt immediately. You will need to reconfigure `BXC51.CPU` to specify your Assembly routine (see page 85).

DISABLE CM*n*
Disable compare interrupt for compare register *n*, where *n* may range from 0 to 2.

ENABLE CM*n*
Enable compare interrupt for compare register *n*, where *n* may range from 0 to 2, when a match between CM*n* and Timer 2 occurs. When the interrupt occurs, it will generate a compare interrupt for register *n* which can be serviced by the BASIC routine setup using ONCM*n* command above.

ONT2 *line*
Specify the subroutine to execute when Timer 2 overflow interrupt is detected. Should a Timer 2 overflow interrupt 1 occur, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When a Timer 2 overflow interrupt occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONT2 statement's interrupt processing ability. Without the ONT2 statement, BXC-51 ignores Timer 2 overflow interrupt.

DISABLE T2
Disable the Timer 2 overflow interrupt. After executing this command, Timer 2 overflows will no longer generate interrupts.

ENABLE T2
Enable Timer 2 overflow interrupts. Once enabled, a Timer 2 overflow will generate an interrupt which can be serviced by the BASIC routine setup using ONT2 command above.

ONS1 *line*
Specify the subroutine to execute when serial interrupt 1 is detected. Should a serial interrupt 1 occur, a GOSUB will be induced to *line*. To return from the subroutine, a RETI instruction must be used in place of a normal RETURN. When a serial interrupt 1 occurs, the normal program flow is temporarily suspended while the interrupt is processed. The program will then resume where it left off. The CLEAR and CLEARI statements will remove the ONS1 statement's interrupt processing ability. Without the ONS1 statement, BXC-51 ignores serial interrupt 1.

Note that this is a very inefficient way of gathering input from serial port 1. Due to the overhead of BASIC and the relatively infrequent checks for interrupts, only slow baud rates are feasible. To be handled properly, Assembly lanugage should be used to service the interrupt immediately. You will need to reconfigure `BXC51.CPU` to specify your Assembly routine (see page 85).

DISABLE S1
Disable serial port 1. The ENS1 bit of S1CON is cleared.

ENABLE S1
Enable serial port 1. Once enabled, as each character arrives, it will generate a serial port 1 interrupt which can be serviced by the BASIC routine setup using ONS1 command above.

ADCH#
The A/D Converter High register. Use as a special byte variable.

ADCON#
The ADC Control register. Use as a special byte variable.

CTCON#
The Capture Control register. Use as a special byte variable.

CT*n*%, CTn#
This is Capture register *n*, where *n* ranges from 0 to 3. Use as a special variable. Its value ranges from -32768 to 0 to +32767 (8000H to 0 to 7FFFH).

CM*n*%, CM*n*#
This is Compare register *n*, where *n* ranges from 0 to 2. Use as a special variable. Its value ranges from -32768 to 0 to +32767 (8000H to 0 to 7FFFH).

IEN0#
Interrupt Enable register 0. This is identical to the regular IE# variable in BXC-51. Use as a special byte variable.

IEN1#
Timer T2 Interrupt Enable register. Use as a special byte variable.

IP0#
Interrupt Priority register 0. This is identical to the regular IP# variable in BXC-51. Use as a special byte variable.

IP1#
Interrupt Timer T2 Interrupt Priority register. Use as a special byte variable.

PORT4#
The Port 4 register. Use as a special byte variable.

PORT5#
The Port 5 register. Use as a special byte variable.

PWMP#
The Pulse Width Modulation Prescalar register. Use as a special byte variable.

PWM0#
The Pulse Width Modulation ratio register 0. Use as a special byte variable.

PWM1#
The Pulse Width Modulation ratio register 1. Use as a special byte variable.

RTE#
The Reset/Toggle Enable Register. Use as a special byte variable.

S0CON#
The Serial port 0 Control register. This is the same as the SCON# byte variable. Use as a special byte variable.

S1ADR#
The Serial port 1 Address register. Use as a special byte variable.

S1DAT#
The Serial port 1 Data register. Use as a special byte variable.

S1STA#
S1CON#
The Serial port 1 control register. Use as a special byte variable.

STE#
The Set Enable Register. Use as a special byte variable.

TM2%, TM2#
T2%, T2#
The Timer 2 register. The current value of Timer 2. Use as a special variable. Its value ranges from -32768 to 0 to +32767 (8000H to 0 to 7FFFH).

TM2CON#
The T2 Control register. Use as a special byte variable.

TM2IR#
The Interrupt Flag Register register. Use as a special byte variable.

T3#
The Timer 3 register. The current value of Timer 3. Use as a special byte variable.

12. BASIC/Assembly Linkage

BASIC is a versatile, general purpose language. Although many programs and subroutines can be written entirely in BASIC, there are some operations that require the speed of Assembly. The following section is for the BASIC/Assembly programmer who wishes to add Assembly support to a BASIC program.

In-Line Assembly

Use the $ASM compiler directive to insert Assembly language code directly into your BASIC source code file. Use the $BASIC compiler directive to continue on with more of your BASIC source code after your assembly code. The following is an example taken from the ASM.BAS file:

```
5 REM This program demonstrates Assembly combined with BASIC
$ASM
        MOV     DPTR,#mymsg       ; address of text
        CALL    ROM_P             ; print text
        CALL    CRLF              ; output CR and LF
        SJMP    OVER1             ; skip over data
MYMSG:  DB      'This is my message!"'
OVER1:
$BASIC
10 PRINT "And this is BASIC's message"
```

If you wish to include in-line Assembly into your BASIC source code and you wish to place it at a specific memory address, you will need to use a trick. By changing the current memory address with the ORG command, you change the location where the BASIC support library begins. Using ORG separates your translated BASIC from your support library in ROM. To avoid this, use this trick in your BASIC source code:

```
10 PRINT "Hello, World!"
$ASM
HERE EQU  $    ; remember the current memory address
     ORG  7700H  ; our specific memory address
$include "mycode.asm"
     ORG  HERE   ; recover previous memory address
$BASIC
20 PRINT "That's all!"
```

This example demonstrates how to divert the assembler so you can place your Assembly code up at 7700H, but not to interrupt the code organization and flow of the BASIC program. If you must interrupt your BASIC code in serveral places like this, you cannot re-use the Assembly label `HERE`. In this case, use HERE1, HERE2, and so on.

Handling Interrupts

If your program code is located at 0H, the interrupt handling performs in exactly the same way as MCS BASIC-52 interpreter. In general, the PSW is pushed on the stack and an LCALL is performed to an interrupt table at 4000H to handle the interrupt. The interrupt table at 4000H must be provided by the programmer and is identical to the table at 0H (i.e., 4003H is external interrupt 0, 400BH is the Timer 0 overflow interrupt, etc.). This table may be located elsewhere if the **-c***addr* command line option is specified.

Address	Routine
4003H	External Interrupt 0
400BH	Timer 0 Overflow
4013H	External Interrupt 1
401BH	Timer 1 Overflow
4023H	Serial I/O
402BH	Timer 2 Overflow
4030H	User Output
4033H	User Input
4036H	User Console Status
4039H	User Reset
403CH	User PRINT@

This is the code generated when your BASIC program is located at 0H:

```
; Interrupt control
; external interrupt 0
ORG  03H  ; X0
JB   DRQ,STQ        ; DRQ set if DMA is set
PUSH PSW
JMP  4003H
RETI

; timer 0 overflow
ORG  0BH  ; T0
PUSH PSW
JB   C_BIT,STJ      ; C_BIT set for real time clock
JMP  400BH

; external interrupt 1
ORG  013H ; X1
JB   INTBIT,STK     ; INTBIT set for ONEX1
JMP  4013H
RETI

; timer 1 overflow
```

```
      ORG  01BH ; T1
      PUSH PSW
      JMP  CKS_I
STJ:  JMP  I_DR

      ; serial port interrupt
      ORG  023H ; SIO
      JMP  BSPI     ; only used when SBUFFER ON

      ; fake DMA code
      LJMP 2040H
STQ:  JB   I_T0,$-3
      CLR  DACK
      JNB  P3.2,$
      SETB DACK
      RETI
      ; make note that ONEX1 interrupt pending
STK:  SETB INTPEN
      RETI
      ; End of interrupt control
```

The only interrupts that must be trapped by to your BXC-51 compiled program are external interrupt 1, if you are using the ONEX1 statement, the timer0 overflow interrupt, if you are using the built-in real time clock (CLOCK1/CLOCK0) and serial I/O interrupt, if you are using SBUFFER ON. BXC-51 does not utilize the other interrupts.

If your program is not located at 0H and BASIC-52 interpreter is not present and enabled, you must route three interrupts to the program code area. To make your interrupt handler smarter, route only the external interrupt 1 to ROMORG + 13H when INTBIT (bit 18) is set; only route timer overflow 1 interrupt to ROMORG + 0BH when (bit 46) is set; and only routine serial I/O interrupt to ROMORG+23H when BUFOFF (bit 41) is clear.

The interrupt table generated for your code when your code is not at 0H has dummy entries for all the interrupts that it does not use immediately by performing a RETI instruction. This is the code generated:

```
ROMORG EQU 4000H
      ; Interrupt control
      ORG  ROMORG+03H     ; X0
      JB   DRQ,STQ        ; DRQ set if DMA is set
      RETI

      ORG  ROMORG+0BH     ; T0
      PUSH PSW
      JB   C_BIT,STJ      ; C_BIT set for real time clock
      POP  PSW
      RETI

      ORG  ROMORG+013H    ; X1
      JB   INTBIT,STK     ; INTBIT set for ONEX1
      RETI
```

```
     ORG  ROMORG+01BH    ; T1
     PUSH PSW
     JMP  CKS_I
STJ: JMP  I_DR

     ORG  ROMORG+023H    ; SIO
     JMP  BSPI

     ORG  ROMORG+02BH    ; T2
     RETI

     ; fake DMA code
     LJMP 2040H
STQ: JB   I_T0,$-3
     CLR  DACK
     JNB  P3.2,$
     SETB DACK
     RETI
     ; make note that ONEX1 interrupt pending
STK: SETB INTPEN
     RETI
     ; End of interrupt control
```

Library Routines

This section provides a supplement to engineers who combine assembly code with BASIC. It explains assembly routines that BASIC commonly uses and that you may wish to use as well. This is *not* intended to be a complete reference to BASIC/Assembly programming. However, a complete reference can be found as part of the BXC-51 Library Toolkit (available from Binary Technology, Inc.).

When using these routines, assume that they use all the registers in RB0, RB1, and RB2. The registers in RB3 are available for your assembly program to use. Any input parameters are expected in RB0. Before calling any of these routines, make sure the PSW is set to RB0 as the default bank.

Floating Point Variable Fetch/Store

A floating point variable occupies six bytes of data. When pushing the value on the floating point argument stack or popping it off, the address that you provide should be the sixth byte of data rather than the first. To push the variable MYVAR onto the Argument Stack, use the routine PUSHAS. For example,

```
     MOV     R0,#LOW(V_MYVAR+5)
     MOV     R2,#HIGH(V_MYVAR+5)
     CALL    PUSHAS
```

To pop the value off the Argument Stack and store it in a floating point variable, use the POPAS routine:

```
        MOV     R2,#HIGH(V_MYVAR+5)
        MOV     R0,#LOW(V_MYVAR+5)
        CALL    POPAS
```

BXC-51 floating point variable labels are prefixed by the compiler with V_. For example, the floating point variable A uses the label V_A.

Floating Point Operators/Functions

A number of floating point functions are provided in the BASIC environment to perform floating point operations. Here is the list.

Routine	Function
AADD	Add two numbers
ASUB	Subtract two numbers
AMUL	Multiply two numbers
ADIV	Divide a number by another
AEXP	Raise one number to the power of another
AXRL	Logically XOR two numbers together
AANL	Logically AND two numbers together
AORL	Logically OR two numbers together
ANEG	Negate the number on the top of the stack
AEQ	Compare two numbers for equality
ANE	Compare two numbers for inequality
AGE	Compare two numbers for >=
ALE	Compare two numbers for <=
ALT	Compare two numbers for <
AGT	Compare two numbers for >
AABS	Take absolute value
AINT	Truncate the number down to an integer
ASGN	Determine sign of number (1, 0, -1)
ANOT	Logically NOT a number
ASIN	Sine of number
ACOS	Cosine of number
ATAN	Tangent of number
AATAN	Arctangent of number
ASQR	Square root of number
ALN	Natural logarithm of a number
AETOX	Raise **e** to the power of number
ARND	A random number

All of these routines operate on the values on the floating point Argument Stack. The results from each of these routines will be left on the floating point Argument Stack.

Integer Variable Fetch/Store

An integer variable requires two bytes of data. To fetch the variable MYVAR% from external memory into R2:R0, put the address of the integer variable into DPTR and use the routine IGETVAR. For example,

```
MOV     DPTR,#IV_MYVAR
CALL    IGETVAR
; value of variable now in R2:R0
```

To store a value into an integer variable, load the value into R3:R1, load the address of the variable into DPTR, and use the IPUTVAR routine. For example,

```
MOV     DPTR,#IV_MYVAR
MOV     R1,#LOW(VALUE)
MOV     R3,#HIGH(VALUE)
CALL    IPUTVAR
```

BXC-51 integer variable labels are prefixed by the compiler with IV_. For example, the integer variable A% uses the label IV_A.

Integer Operators/Functions

A number of integer functions are provided in the BASIC environment to perform integer arithmetic. Here is the list.

Routine	Function
IADD	Add R3:R1 with R2:R0 and store in R3:R1
ISUB	Subtract R2:R0 from R3:R1 and store in R3:R1
IMUL	Multiple R2:R0 by R3:R1 and store in R3:R1
IDIV	Divide R3:R1 by R2:R0 and store in R3:R1
IEQ	Set R3:R1 equal to the result of R3:R1 = R2:R0
IGE	Set R3:R1 equal to the result of R3:R1 >= R2:R0
IGT	Set R3:R1 equal to the result of R3:R1 > R2:R0
ILE	Set R3:R1 equal to the result of R3:R1 <= R2:R0
ILT	Set R3:R1 equal to the result of R3:R1 < R2:R0
INE	Set R3:R1 equal to the result of R3:R1 <> R2:R0
IANL	Logically AND R3:R1 and R2:R0, store in R3:R1
IORL	Logically OR R3:R1 and R2:R0, store in R3:R1
IXRL	Logically XOR R3:R1 and R2:R0, store in R3:R1
ISHL	Left-shift R3:R1 by R2:R0, store in R3:R1
ISHR	Right-shift R3:R1 by R2:R0, store in R3:R1
IABS	Absolute value of R2:R0
ISGN	Determine sign of R2:R0 (-1, 0, 1)
INEG	Negate value of R2:R0

Byte Variable Fetch/Store

A byte variable occupies one byte of internal RAM. To fetch the variable MYVAR# into R2:R0, use this Assembly code:

```
MOV     R0,#BV_MYVAR
MOV     A,@R0
MOV     R0,A
MOV     R2,#0
```

If the byte variable is mapped to a control register (or if the byte variable's address is lower than 80H), use this code instead:

```
MOV     R0,BV_MYVAR
MOV     R2,#0
```

To store a value into a byte variable, use this Assembly code:

```
MOV     R0,#BV_MYVAR
MOV     A,#VALUE
MOV     @R0,A
```

If the byte variable is mapped to a control register (or if the byte variable's address is lower than 80H), use this code instead:

```
MOV     BV_MYVAR,#VALUE
```

BXC-51 byte variable labels are prefixed by the compiler with BV_. For example, the byte variable A# uses the label BV_A.

Static String Variable Fetch/Store

To obtain the address of a static string variable (assuming that the STRING statement has already allocated string space), use the IST_CAL routine. For example,

```
MOV     R3,#0
MOV     R1,#5
CALL    IST_CAL
; the string address of $(5) now in R3:R1
```

To store a string from ROM into a static string, put the address of the text to store into DPTR, put the address of the static string variable in R3:R1, and call TQ2STR. For example,

```
MOV     R3,#0    ; $(5)
MOV     R1,#5
CALL    IST_CAL
MOV     DPTR,ROMSTR
CALL    TQ2STR
```

Dynamic String Variable Fetch/Store

To obtain the address of a dynamic string's text, use the GETSTRADDR routine. For example,

```
MOV     DPTR,#SV_A+2
CALL    GETSTRADDR
; R3:R1 now contains address and R5 contains length
```

Be careful about changing the text of a string variable in memory. Do not increase the string's length - it may overwrite portions of memory that manage dynamic strings which could cause a crash.

To assign a new value to a string, store the length of the string at STRBUF and copy the text to STRBUF+1, then call PUTSTR. For example,

```
; store length at STRBUF
MOV     DPTR,#STRBUF
MOV     A,#5    ; length
MOVX    @DPTR,A
; copy string to STRBUF+1
...
; assign value
MOV     DPTR,#SV_MYVAR
CALL    PUTSTR
```

BXC-51 dynamic string variable labels are prefixed by the compiler with SV_. For example, the dynamic string variable A$ uses the label SV_A.

Dynamic String Operators/Functions

A number of dynamic string functions are provided in the BASIC environment to perform string operations. Here is the list.

Routine	Function
SCONCAT	Concatenate a string at R5;R3:R1 to STRBUF
SRIGHT	Perform RIGHT$ on string at R5;R3:R1 starting at R2:R0
SLEFT	Perform LEFT$ on string at R5;R3:R1 starting at R2:R0
SMID	Perform LEFT$ on string at R5;R3:R1 starting at R7:R6 for R2:R0 characters
SCHR	Create 1 character string for character in R1
STRSTR	Convert floating point value on A-Stack to string
STRCMP	Compare string R4;R2:R0 to R5;R3:R1, result in R3:R1

For each of these string operations, the resulting string is put in R5;R3:R1.

Array Variable Fetch/Store

An array variable occupies 3 bytes which contains the array size and the memory address where the array begins. To determine the address where an array index begins, put the variable size in ARRSIZ (6 for floating point, 3 for dynamic string, 2 for integer), put the third byte of the array variable's address in DPTR, the array index in R3:R1, and call AROFS. For example,

```
MOV     ARRSIZ,#6
MOV     DPTR,#AV_MYVAR+2
MOV     R1,#7
MOV     R3,#0   ; always 0
CALL    AROFS
; floating point variables must be adjusted
CALL    ADJARR
```

```
; variable location now in R2:R0
```

To fetch or store a floating point variable, use the PUSHAS or POPAS routine described above as "Floating Point Fetch/Store." To fetch or store an integer variable, use the IGETVAR and IPUTVAR routine described above as "Integer Fetch/Store." For integer and dynamic string variables, make sure ARRSIZ returns to 6 after calling AROFS. ARRSIZ is expected to be 6.

If the array has not been dimensioned yet, it will be dimensioned to 10.

Text Output to Serial Port

There are several routines for sending text to the serial port. How to output a floating point, an integer, a static string, a dynamic string, a constant ROM string, and a single character is explained below. In each case, the output goes to the currently defined output device, whether it is the console, list device, or user-defined.

To print a floating point number, fetch the value onto the Argument Stack and call PRINT_FP. This routine will output the top of the Argument Stack. For example,

```
MOV     R0,#LOW(V_MYVAR+5)
MOV     R2,#HIGH(V_MYVAR+5)
CALL    PUSHAS
CALL    PRINT_FP
```

To print an integer number, place the integer in R2:R0, push it onto the A-Stack with TWO_EYS, and then call PRINT_FP. The integer will be output as an signed integer from -32768 to 0 to 32767. For example, to print 565,

```
MOV     R0,#LOW(565)
MOV     R2,#HIGH(565)
CALL    TWO_EYS
CALL    PRINT_FP
```

To print a byte, place the byte in R0, place 0 in R2, push it onto the A-Stack with TWO_EYS, and then call PRINT_FP. For example, to print 34,

```
MOV     R0,#34
MOV     R2,#0
CALL    TWO_EYS
CALL    PRINT_FP
```

To print a static string, put the address of the string in R3:R1 and call UPRNT. This routine will output a series of characters up to a carriage return. For example, to print out $(1),

```
MOV     R1,#1
MOV     R3,#0
MOV     A,R1
CALL    IST_CAL
CALL    UPRNT
```

To print a dynamic string, put the address of the string in R5;R3:R1 and call PRINT_DSTR. This routine will output the characters located at R3:R1 for the length specified in R5. For example, to print out A$,

```
          MOV    DPTR,#SV_A+2
          CALL   GETSTRADDR
          CALL   PRINT_DSTR
```

To print a string of text from ROM, move the address of beginning of text to DPTR and call ROM_P. This routine will output a series of characters up to a double quote character, ". For example,

```
          MOV    DPTR,#MSG
          CALL      ROM_P
          CALL                CRLF
          ...
     MSG: DB     'Hello, World!"'
```

To output a single character, put the character in R5 and call the TEROT routine. The character in R5 will be output. For example, to output the letter A,

```
          MOV                 R5,#'A'
          CALL   TEROT
```

Text Input from Serial Port

Either a character or a line of text may be input from the serial port. Call the INCHAR routine to wait for a character to be received. It will be stored in the accumulator when it arrives.

```
          CALL   INCHAR
 ; wait for character
          MOV    DPTR,#100
          MOVX   @DPTR,A  ; store at 100
```

Call the SINPUT routine to wait for a line of text (up to a carriage return) to be received. The text line is stored in the BASIC input buffer starting at 0007H.

```
     CALL    SINPUT   ; wait for a line
          MOV    DPTR,#4
          MOVX   A,@DPTR  ; get line length
          MOV    DPTR,#7  ; input buffer
          ...
```

Simple Custom BASIC Command

BXC-51 V3.0 introduced a quick and simple way to link your Assembly code to your BXC-51 program using the DEFASM command. The DEFASM command allows you to specify a routine at a particular address in memory. For example,

```
     DEFASM CheckMeter@7700H
```

specifies the Assembly routine for `CheckMeter` is located at 7700H in ROM. Use the command `CheckMeter` instead of using `CALL 7700H` in your program. This is very handy when you know the locations of your Assembly routines and those routine require no parameters passed by BASIC. Parameters are typically setup by using XBY() and DBY() or using memory mapped variables.

BXC-51 V5.0 allows the DEFASM command to specify the address as a label instead of a hexadecimal specific address. This allows you to embed the Assembly code in your BASIC program and not worry about getting its address correct. For example:

```
10 DEFASM CheckMeter@AR_CHECKMETER
20 GOTO 30      ; skip over Assembly code for time being
$ASM
AR_CHECKMETER:
     MOV       DPTR,#0F00H    ; address of the meter
     MOVX      A,@DPTR        ; get the current meter value
     ANL       A,#01FH        ; strip off irrelevant bits
     MOV       BV_SETTING,A   ; store in SETTING#
     RET
$BASIC
30 CheckMeter  ; call the Assembly routine
40 PRINT SETTING#
```

This example demonstrates using DEFASM to setup a subroutine at a label, adding the Assembly subroutine in-line, and communicating the result back to the BASIC program by storing the result in the BASIC byte variable SETTING#. When using in-line Assembly code for subroutines, remember to place a GOTO statement before it so the subroutine is only called when desired. Some engineers prefer to place all their Assembly code at the end of the program to keep it out of the way of the BASIC code. In this case, use an END statement before the Assembly code. Placement of in-line Assembly code is up to you.

BASIC Extensions through BXLs

As of BXC-51 V5.0, you may extend the BASIC command and function set similarly to the extensions of the interpreter. But even better. Each BXL specifies code to execute at program initialization and termination. Each BXL specifies new commands and new functions for BASIC. Command and function code is only linked into the program if the command or function is used.

BXL means BASIC eXtension Library and refers to the file extension, .BXL. When the user specifies **-b***filename* on the command line, the compiler opens the file named *filename*.BXL and reads its BASIC extensions.

The BXL file contains mostly Assembly code to support the commands and functions it defines. However, it also contains special keywords such as COMMAND and FUNCTION to clue the compiler in to the extensions defined. Each COMMAND or FUNCTION defined has three sections: initialization, termination, and execution. The initialization section is CALLed when the BASIC program begins. The termination section is CALLed when the BASIC program exits (whether exiting due to END, STOP, or error). The execution section is CALLed every time the

BASIC source code uses the command or function. The initialization, termination, and execution sections must RETurn to the BASIC program. A maximum of 16 parameters may be passed to a command or function. Passed parameters may be floating point, integer, byte, bit, or string values, however there are some limitations due to the number of registers that can contain information.

If a BXL file defines an AUTO command, that code is not used as a BASIC extension. Instead, the initialization and termination code will automatically be generated to the output file. This is handy for initializing buffers or memory used by the commands and functions in the BXL. This is also handy for starting and stopping I/O devices accessed by the defined commands. If the BXL is used for a derivative microcontroller, the AUTO command is ideal for setting up and coding interrupt handlers. For example,

```
COMMAND AUTO

INIT:
     ; just some EQU's for the commands
switch_addr  EQU     0FFC1H

     ; we MUST return, otherwise program will hang
     ; upon startup
     RET
END
```

All text in the BXL upto the first COMMAND or FUNCTION keyword is ignored. Each defined COMMAND and FUNCTION contains three sections which each begin with a special Assembly label: INIT, *name*, and EXIT (where *name* is the name of the command or function). A section ends when another section, COMMAND, or FUNCTION begins. All text inside a section is passed straight through to the Assembly output of the program. No syntax checking is performed. However, CALL and JMP statements are reviewed to see if they refer to a BASIC support library routine. If so, the compiler makes sure that routine is included when the BASIC support library code is generated.

If you share your BXL files with other engineers, you should verify that all your defined commands and functions can be compiled and run successfully. To be thorough, you should also make sure that each command and function may be compiled and run successfully, separately. If you devise a particularly clever or useful BXL, you may wish to sell it. If so and you wish to encrypt your BXL, please contact your dealer for BXL-Encrypt, the BXL file encryption tool.

BASIC Command Extentions

Inside a BXL, each defined BASIC command begins with the keyword COMMAND as the first word on the line. Following the COMMAND keyword is the name of the command and the parameters (if any). The command name may not include any spaces in it. Each parameter is specified as a name and type. Parameters are separated by spaces even though BASIC source code will have commas separating them. The name has no functional use, but it is handy for documentation purposes. The type can be any one of:

Type	BASIC Type	Parameter passing method
fp	floating point	The input parameter is passed on the floating point Argument Stack
int31	integer	The input parameter is passed in R3 (high byte) and R1 (low byte)
int20	integer	The input parameter is passed in R2 (high byte) and R0 (low byte)
int76	integer	The input parameter is passed in R7 (high byte) and R6 (low byte)
byte1	byte	The input parameter is passed in R1
byte0	byte	The input parameter is passed in R0
byte6	byte	The input parameter is passed in R6
bitc	bit	The input parameter is passed in the Carry flag (PSW.7)
string314	dynamic string	The input parameter is passed in R3 (high byte of string address), R1 (low byte of string address), and R4 (length of string). The string address is in external RAM
string205	dynamic string	The input parameter is passed in R2 (high byte of string address), R0 (low byte of string address), and R5 (length of string). The string address is in external RAM

The name is separated from the type by a colon. For example, this line begins the definition for a command which takes two integers:

```
COMMAND FLASH light:int31 duration:int20
```

When the FLASH command is encountered in the BASIC source code, the compiler will expect two integer parameters to follow. Anything different causes an error. For example, this would be valid:

```
FLASH 3,length%*2
```

The parameters would be passed to the BXL code according to their types. The first parameter is of type int31, so R3 would become 0 and R1 would become 3. Similarly, since the second parameter is of type int20, the compiler would set R2 equal to HIGH(length%*2) and R0 equal to LOW(length%*2). The immediate limitation that arises is that there are only 3 integer types, so only three integer parameters may be present. Two integers going to the same type will cause one of the integers to be clobbered. To circumvent this limitation, specify the fp type and use the IFIX routine to pop a floating point value off A-Stack to return it in R3:R1.

Stay alert when working with floating point parameters. The parameters are pushed on the stack in the order they are evaluated. So, the last floating point value pushed is the first one to be popped off the stack. Parameter evaluation proceeds from left to right. So, the rightmost floating point parameter is popped first.

Immediately following the command should be three sections of code:

```
INIT:
     ; initialization code
name:
     ; code which executes the command
EXIT:
     ; termination code
```

The INIT and EXIT sections are optional. The text between these section headers is passed straight through to the Assembly output along with any syntax errors that may be present. Use the END keyword to end a section so the text between END and the next section is not copies through to the Assembly output.

Here is an example COMMAND definition.

```
COMMAND LED number:byte1

; The single digit parameter to this command will
; be displayed on the 7 segment LED connected to
; Port 1.  If -1 (or any number over 9) is passed
; as a parameter, the display is blanked.
;
; Example:
;    LED 5
; will display a '5' on the 7 segment display

init:
     ; at startup, blank out the LED
     MOV  P1,#led_blank
     RET

led:
     MOV  A,R1
     CJNE A,#10,$+3
     JC   led_digit
     ; input not in range of 0-9, so blank display
     MOV  P1,#led_blank
     RET

led_digit:
     MOV  DPTR,#led_table
     MOVC A,@A+DPTR
     MOV  P1,A
     RET

led_table:
     DB   00010000B ; 0
     DB   11110100B ; 1
     DB   01000001B ; 2
     DB   11000000B ; 3
     DB   10100100B ; 4
     DB   10000010B ; 5
     DB   00000010B ; 6
     DB   11110000B ; 7
     DB   00000000B ; 8
     DB   10100000B ; 9
led_blank EQU  0FFh ; blank

exit:
     ; at exit, blank out the LED
     MOV  P1,#led_blank
     RET

END  ; end of command
```

BASIC Function Extentions

A function interfaces with the compiler to receive its input parameters identially to the way a command does (see above). However, a function is required to return a value as well. Inside a BXL, each defined BASIC function begins with the keyword FUNCTION as the first word on the line. Following the FUNCTION keyword is the name and type of the function and the parameters (if any). The command name may not include any spaces in it. Each parameter is specified as a name and type. Parameters are separated by spaces even though BASIC source code will have commas separating them. The parameter name has no functional use, but it is handy for documentation purposes. The parameter type can be any one of fp, int31, int20, int76, byte1, byte0, byte6, bitc, string314, and string205 (see the above section for type explanations). The function type is a subset of parameter types. A function can return a value as type fp, int31, byte1, bitc, or string315. The return type tells the compiler where to expect the result.

For example, to declare a function which takes two floating point values and returns an integer, it would look like this:

```
FUNCTION BINS:int31 total:fp step:fp
```

When the BINS function is encountered in the BASIC source code, the compiler will expect two floating point parameters to follow. Anything different causes an error. For example, this would be valid:

```
a%= BINS(PI,.6)/4
```

The parameters would be passed to the BXL code according to their types. In this case, both parameters would be pushed on the floating point stack, first PI and then .6. After the appropriate code has analyzed the two parametes, a result in this example is expected in R3 and R1. The compiler then generates code to divide that result by 4 before assigning it to the integer variable `a%`.

Other than returning a result, a function is identical to a command. See the Basic Command Extension section above for more information.

Here is an example FUNCTION definition.

```
FUNCTION SWITCH:bitc  switch_no:byte1
; This function reads the state of a switch
; located at FFC1.  Each bit of the byte read
; represents one of 8 switches at this address.
; This function reads the current setting for
; one of the switches and returns that value
; (1 or 0).  A valid switch number may be from
; 0 to 7.

SWITCH:
     ; the switch number is in R1
     MOV  A,R1       ; the switch #
     CJNE A,#8,$+3   ; make sure valid switch #
     JC   $+3
     RET             ; switch # invalid, return 0
     MOV     DPTR,#switch_addr ; location of switch
     MOVX    A,@DPTR ; get it's current state
     INC     R1      ; shift right until bit
     RRC     A
     DJNZ    R1,$-1  ; bit 0 in accumulator
     RET             ; our return value is in Carry
END
```

Derivative Microcontroller Extentions

A BXL is an ideal way to support functionality specific to a derivative microcontroller. Place any interrupt handlers in the COMMAND AUTO section. Any special commands or functions are declared individually. All of this is provided to the engineering using the compiler without the engineer specifying it with a command line option. The engineer specifies the microcontroller type with the **-t***cpu* command line option which reads the configuration for the microcontroller which configures this BXL to be included.

The COMMAND AUTO section can do more than initialize the microcontorller's registers, it can define the interrupt handlers. For example:

```
COMMAND AUTO
INIT:
     ; initialize microcontroller registers
     RET
ivec_wd:
     ; handle the interrupt for wd
     PUSH PSW
     ...
     POP  PSW
     RETI
ivec_xyz:
```

```
          ; handle the interrupt for xyz
          PUSH PSW
          ...
          POP  PSW
          RETI
     EXIT:
          ; de-initialize microcontroller registers
          RET
     END
```

For this example, the labels `ivec_wd` and `ivec_xyz` would be specified in the .CPU file with the INT command (see page 86).

For commands and functions that are specific to the derivative microcontroller, see the "Basic Command Extension" and "Basic Function Extension" sections above.

BXL Programming Tips

When writing BXL code, keep these tips in mind:

- Use unique label prefixes for all temporary or local labels. Assembly labels may be up to 32 characters long. Use three or more letters as a prefix, followed by underscore, to be safe, for example, FLASH_JUMP:, FLASH_LOOP:, FLASH_EXIT:, etc.

- Avoid using ORG. If you use ORG, remember the current program counter location and recover it.

- If the parameters are really integers and you have less than 4 integer parameters, use the integer type. It generates faster code. Popping the floating point Argument Stack and converting to an integer is a waste of time. Let the compiler determine if the value is already an integer or if it needs to be converted to an integer.

- Avoid referring to BASIC variables by label. You cannot be assured that the BASIC source code uses the variable referenced. If the variable is not referenced in BASIC, the assembler will report it as an undefined label.

- Feel free to use register bank 0 and 3 as a scratch area. Feel free to use the accumulator, B register, and DPTR without saving their contents.

- If your commands or functions use common EQUates, place the EQUates in a COMMAND AUTO section. Do not forget to RETurn from the INIT section, though.

- If your commands or functions use common Assembly routines, specify those routines in the COMMAND AUTO section (after RETurn from INIT).

- BXL functions do not use %, #, or $ to designate their type. Whether a function returns a floating point, integer, byte, bit, or string, the function call is the same. For example, MYFUNC(PARAM1, PARAM2).

13. Microcontroller Summary

For your reference, this section provides a summary of the 8051/8052 microcontroller.

Special Function Registers

Symbol	BASIC Variable	Name	Address
ACC*		Accumulator	0E0H
B*		B Register	0F0H
PSW*		Program Status Word	0D0H
SP		Stack Pointer	81H
DPTR		Data Pointer (2 bytes)	
DPL		Low byte	82H
DPH		High byte	83H
P0*	PORT0#	Port 0	80H
P1*	PORT1#	Port 1	90H
P2*	PORT2#	Port 2	0A0H
P3*	PORT3#	Port 3	0B0H
IP*	IP#	Interupt Priority Control	0B8H
IE*	IE#	Interupt Enable Control	0A8H
TMOD	TMOD#	Timer/Counter Mode Control	89H
TCON*	TCON#	Timer/Counter Control	88H
T2CON+*	T2CON#	Timer/Counter 2 Control	0C8H
TH0	TIMER0%	Timer/Counter 0 High Byte	8CH
TL0		Timer/Counter 0 Low Byte	8AH
TH1	TIMER1%	Timer/Counter 1 High Byte	8DH
TL1		Timer/Counter 1 Low Byte	8BH
TH2+	TIMER2%	Timer/Counter 2 High Byte	0CDH
TL2+		Timer/Counter 2 Low Byte	0CCH
RCAP2H+	RCAP2%	Timer/Counter 2 Capture Register High Byte	0CBH
RCAP2L+		Timer/Counter 2 Capture Register Low Byte	0CAH
SCON*		Serial Control	98H
SBUF		Serial Data Buffer	99H
PCON		Power Control	87H

* Bit addressable
+ 8052 only

PSW: Program Status Word (0D0H)

CY	AC	F0	RS1	RS0	OV	-	P

CY	PSW.7	Carry flag
AC	PSW.6	Auxilary Carry flag
F0	PSW.5	General purpose Flag 0
RS1	PSW.4	Register Bank selector bit 1
RS0	PSW.3	Register Bank selector bit 0
OV	PSW.2	Overflow flag
-	PSW.1	User definable flag
P	PSW.0	Parity flag

PCON: Power Control Register (87H)

SMOD	-	-	-	GF1	GF0	PD	IDL

SMOD	Double baud rate bit
-	Reserved for future use
-	Reserved for future use
-	Reserved for future use
GF1	General purpose flag
GF0	General purpose flag
PD	Power down bit

IDL		Idle Mode bit

IE: Interrupt Enable Register (0A8H)

EA	-	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disable all interrupts if 0, enable individual interrupts if 1
-	IE.6	Reserved for future use
ET2	IE.5	Enable Timer 2 overflow or capture interrupt (on 8052)
ES	IE.4	Enable serial port interrupt
ET1	IE.3	Enable Timer 1 overflow interrupt
EX1	IE.2	Enable External Interrupt 1
ET0	IE.1	Enable Timer 0 overflow interrupt
EX0	IE.0	Enable External Interupt 0

IP: Interrupt Priority Register (0B8H)

-	-	PT2	PS	PT1	PX1	PT0	PX0

-	IP.7	Reserved for future use
-	IP.6	Reserved for future use
PT2	IP.5	Timer 2 interrupt priority level (on 8052)
PS	IP.4	Serial port interrupt priority level
PT1	IP.3	Timer 1 interrupt priority level
PX1	IP.2	External interrupt 1 priority level
PT0	IP.1	Timer 0 interrupt priority level
PX0	IP.0	External interrupt 0 priority level

TCON: Timer/Counter Control Register (88H)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine
TR1	TCON.6	Timer 1 run control bit. Set/Cleared by software to turn Timer/Counter 1 on/off
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the interrupt service routine
TR0	TCON.4	Timer 0 run control bit. Set/Cleared by software to turn Timer/Counter 0 on/off
IE1	TCON.3	External Interrupt 1 edge flag. Set by hardware when External Interrupt edge is detected. Cleared by hardware when interrupt is processed
IT1	TCON.2	External Interrupt 1 type control bit. Set/Cleared by software to specify falling edge/low level triggered external interrupt
IE0	TCON.1	External Interrupt 0 edge flag. Set by hardware when External Interrupt edge is detected. Cleared by hardware when interrupt is processed
IT0	TCON.0	External Interrupt 0 type control bit. Set/Cleared by software to specify falling edge/low level triggered external interrupt

TMOD: Timer/Counter Mode Control Register (89H)

GATE	C/T	M1	M0	GATE	C/T	M1	M0

GATE		When TR*x* (in TCON) is set and GATE=1, Timer/Counter*x* will run only while INT*x* pin is high (hardware control). When GATE=0, Timer/Counter*x* will run only while TR*x*=1 (software control)
C/T		Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from T*x* input pin)

M1, M0		Mode=00, 5-bit timer prescaler (MCS-48 compatible)
		Mode=01, 16-bit timer/counter
		Mode=10, 8-bit auto-reload timer/counter
		Mode=11, (Timer 0) TL0 is an 8-bit timer/counter controlled by the standard Timer 0 control bits, TH0 is an 8-bit timer and is controlled by Timer 1 control bits
		Mode=11, (Timer 1) Timer/Counter 1 stopped

T2CON: Timer/Counter 2 Control Register (0C8H)

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

TF2	T2CON.7	Timer 2 overflow flag set by hardware and cleared by software. TF2 cannot be set when either RCLK=1 or CLK=1

EXF2	T2CON.6	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX when EXEN2=1. When Timer 2 interrupt is enabled, EXF2=1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software
RCLK	T2CON.5	Receive clock flag. When set, causes the Serial Port to use Timer 2 overflow pulses for its receive clock in modes 1 & 3. RCLK=0 causes Timer 1 overflow to be used for the receive clock.
TCLK	T2CON.4	Transmit clock flag. When set, causes the Serial Port to use Timer 2 overflow pulses for its transmit clock in modes 1 & 3. TCLK=0 causes Timer 1 overflows to be used for the transmit clock.
EXEN2	T2CON.3	Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of negative transition on T2EX if Timer 2 is not being used to clock the Serial Port. EXEN2=0 causes Timer 2 to ignore events at T2EX
TR2	T2CON.2	Software start/stop control for Timer 2. A logic 1 starts the timer
C/T2	T2CON.1	Timer(0) or Counter(1) select
CP/RL2	T2CON.0	Capture/Reload flag. When set, captures will occur on negative transitions at T2EX if EXEN2=1. When cleared, auto-reloads will occur either with Timer 2 overflows or this bit is ignored and the Timer is forced to auto-reload on Timer 2 overflow

SCON: Serial Port Control Register (98H)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0 SM1	SCON.7 SCON.6	Mode=0, Shift Register, Fosc/12 baud
		Mode=1, 8-bit UART, Variable baud
		Mode=2, 9-bit UART, Fosc/64 or Fosc/32
		Mode=3, 9-bit UART, Variable

SM2	SCON.5	Enables the multiprocessor communication feature in modes 2 & 3. In mode 2 or 3, if SM2 is set to 1 then RI will not be activitated if the received 9th data bit (RB8) is 0. In mode 1, if SM2=1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0.
REN	SCON.4	Set/Cleared by software to enable/disable reception
TB8	SCON.3	The 9th bit that will be transmitted in modes 2 & 3. Set/Cleared by software
RB8	SCON.2	In modes 2 & 3, is the 9th data bit that was received. In mode 1, if SM2=0, RB8 is the stop bit that was received. In mode 0, RB8 is not used
TI	SCON.1	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in other modes. Must be cleared by software
RI	SCON.0	Receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in other modes (except see SM2). Must be cleared by software

Instruction Set Summary

Mnemonic Operation	Flags P OV AC C	Function
ACALL addr11 $(PC) \leftarrow (PC) + 2$ $(SP) \leftarrow (SP) + 1$ $((SP)) \leftarrow (PC_{7-0})$ $(SP) \leftarrow (SP) + 1$ $((SP) \leftarrow (PC_{15-8})$ $(PC_{10-0}) \leftarrow$ page addr		Absolute Call
ADD A,Rn $(A) \leftarrow (A) + (Rn)$	P OV AC C	Add
ADD A,direct $(A) \leftarrow (A) + (direct)$	P OV AC C	Add
ADD A,@Ri $(A) \leftarrow (A) + ((R_i))$	P OV AC C	Add
ADD A,#data (A) ← (A) + #data	P OV AC C	Add
ADDC A,Rn $(A) \leftarrow (A) + (C) + (Rn)$	P OV AC C	Add with Carry
ADDC A,direct $(A) \leftarrow (A) + (C) + (direct)$	P OV AC C	Add with Carry

Mnemonic Operation	Flags P OV AC C	Function
ADDC A,@Ri (A) ← (A) + (C) + ((R_i))	P OV AC C	Add with Carry
ADDC A,#data (A) ← (A) + (C) + #data	P OV AC C	Add with Carry
AJMP addr11 (PC) ← (PC) + 2 ($PC_{10\text{-}0}$) ← page addr		Absolute Jump
ANL A,Rn (A) ← (A) ^ (Rn)	P	Logical-AND for byte variables
ANL A,direct (A) ← (A) ^ (direct)	P	Logical-AND for byte variables
ANL A,@Ri (A) ← (A) ^ ((R_i))	P	Logical-AND for byte variables
ANL A,#data (A) ← (A) ^ #data	P	Logical-AND for byte variables
ANL direct,A (direct) ← (direct) ^ (A)		Logical-AND for byte variables
ANL C,bit (C) ← (C) ^ (bit)	C	Logical-AND for bit variables
ANL C,/bit (C) ← (C) ^ /(bit)	C	Logical-AND for bit variables

CJNE A,direct,rel
(PC) ← (PC) + 3
IF (A) <> (direct)
THEN
  (PC) ← (PC) + rel
IF (A) < (direct)
THEN
  (C) ← 1
ELSE
  (C) ← 0
	C	Compare and Jump if Not Equal

CJNE A,#data,rel
(PC) ← (PC) + 3
IF (A) <> data
THEN
  (PC) ← (PC) + rel
IF (A) < data
THEN
  (C) ← 1
ELSE
  (C) ← 0
	C	Compare and Jump if Not Equal

CJNE Rn,#data,rel
(PC) ← (PC) + 3
IF (Rn) <> data
THEN
  (PC) ← (PC) + rel
IF (Rn) < data
THEN
  (C) ← 1
ELSE
  (C) ← 0
	C	Compare and Jump if Not Equal

CJNE @Ri,#data,rel
(PC) ← (PC) + 3
IF ((R_i)) <> data
THEN
  (PC) ← (PC) + rel
IF ((R_i)) < data
THEN
  (C) ← 1
ELSE
  (C) ← 0
	C	Compare and Jump if Not Equal

Mnemonic Operation	Flags				Function
	P	OV	AC	C	
CLR A (A) ← 0	P				Clear Accumulator
CLR C (C) ← 0				C	Clear Carry Flag
CLR bit (bit) ← 0					Clear Bit
CPL A (A) ← ~(A)	P				Complement Accumulator
CPL C (C) ← ~(C)				C	Complement Carry Flag
CPL bit (bit) ← ~(bit)					Complement Bit

DA A
- ACC contents BCD
IF [[(A_{3-0}) > 9] V [(AC) = 1]]
THEN (A_{3-0})←(A_{3-0})+6
AND
IF [[(A_{7-4}) > 9] V [(C) = 1]]
THEN (A_{7-4})←(A_{7-4})+6
	P			C	Decimal-adjust Accumulator for Addition

DEC A (A) ← (A) - 1	P				Decrement Accumulator
DEC Rn (Rn) ← (Rn) - 1					Decrement Register
DEC direct (direct) ← (direct) - 1					Decrement
DEC @Ri (Ri) ← (Ri) - 1					Decrement
DIV AB (A) ← (A)/(B) (B) ← (A) mod (B)	P	OV		C	Divide
DJNZ Rn,rel (PC) ← (PC) + 2 (Rn) ← (Rn) - 1 IF (Rn) > 0 or (Rn) < 0 THEN (PC) ← (PC) + rel					Decrement and Jump if Not Zero
DJNZ direct,rel (PC) ← (PC) + 2 (Rn) ← (Rn) - 1 IF (direct) > 0 or (direct) < 0 THEN (PC) ← (PC) + rel					Decrement and Jump if Not Zero
INC A (A) ← (A) + 1	P				Increment Accumulator
INC Rn (Rn) ← (Rn) + 1					Increment Register
INC direct (direct) ← (direct) + 1					Increment
INC @Ri (Ri) ← (Ri) + 1					Increment
INC DPTR (DPTR) ← (DPTR) + 1					Increment Data Pointer
JB bit,rel (PC) ← (PC) + 3 IF (bit) = 1 THEN (PC) ← (PC) + rel					Jump if Bit Set

Mnemonic	Flags				Function
Operation	P	OV	AC	C	
JBC bit,rel					Jump if Bit is Set and Clear Bit
(PC) ← (PC) + 3					
IF (bit) = 1					
THEN					
(bit) ← 0					
(PC) ← (PC) + rel					
JC rel					Jump if Carry Set
(PC) ← (PC) + 2					
IF (C) = 1					
THEN					
(PC) ← (PC) + rel					
JMP @A+DPTR					Jump Indirect
(PC) ← (A) + (DPTR)					
JNB bit,rel					Jump if Bit Not Set
(PC) ← (PC) + 3					
IF (bit) = 0					
THEN					
(PC) ← (PC) + rel					
JNC rel					Jump if Carry Not Set
(PC) ← (PC) + 2					
IF (C) = 0					
THEN					
(PC) ← (PC) + rel					
JNZ rel					Jump if Accumulator Not Zero
(PC) ← (PC) + 2					
IF (A) <> 0					
THEN					
(PC) ← (PC) + rel					
JZ rel					Jump if Accumulator Zero
(PC) ← (PC) + 2					
IF (A) = 0					
THEN					
(PC) ← (PC) + rel					
LCALL addr16					Long Call
(PC) ← (PC) + 3					
(SP) ← (SP) + 1					
((SP)) ← (PC_{7-0})					
(SP) ← (SP) + 1					
((SP)) ← (PC_{15-8})					
(PC) ← $addr_{15-0}$					
LJMP addr16					Long Jump
(PC) ← $addr_{15-0}$					
MOV A,Rn	P				Move Byte
(A) ← (Rn)					
MOV A,direct	P				Move Byte
(A) ← (direct)					
MOV A,@Ri	P				Move Byte
(A) ← ((Ri))					
MOV A,#data	P				Move Byte
(A) ← #data					
MOV Rn,A					Move Byte
(Rn) ← (A)					
MOV Rn,direct					Move Byte
(Rn) ← (direct)					
MOV Rn,#data					Move Byte
(Rn) ← #data					
MOV direct,A					Move Byte
(direct) ← (A)					

Mnemonic Operation	Flags				Function
	P	OV	AC	C	
MOV direct,Rn (Rn) $\leftarrow$ (direct)					Move Byte
MOV direct,direct (direct) $\leftarrow$ (direct)					Move Byte
MOV direct,@Ri (direct) $\leftarrow$ ((Ri))					Move Byte
MOV direct,#data (direct) $\leftarrow$ #data					Move Byte
MOV @Ri,A ((Ri)) $\leftarrow$ (A)					Move Byte
MOV @Ri,direct ((Ri)) $\leftarrow$ (direct)					Move Byte
MOV @Ri,#data ((Ri)) $\leftarrow$ #data					Move Byte
MOV C,bit (C) $\leftarrow$ (bit)				C	Move Bit to Carry
MOV bit,C (bit) $\leftarrow$ (C)					Move Carry to Bit
MOV DPTR,#data (DPTR) $\leftarrow$ #data					Move Value to Data Pointer
MOVC A,@A+DPTR (A) $\leftarrow$ ((A) + (DPTR))	P				Move Code Byte
MOVC A,@A+PC (A) $\leftarrow$ ((A) + (PC))	P				Move Code Byte
MOVX A,@Ri (A) $\leftarrow$ ((Ri))	P				Move External Byte
MOVX A,@DPTR (A) $\leftarrow$ ((DPTR))	P				Move External Byte
MOVX @Ri,A ((Ri)) $\leftarrow$ (A)					Move External Byte
MOVX @DPTR,A ((DPTR)) $\leftarrow$ (A)					Move External Byte
MUL AB $(A)_{7\text{-}0}$ $(B)_{15\text{-}8}$ $\leftarrow$ (A)X(B)	P	OV		C	Multiply
NOP (PC) $\leftarrow$ (PC) + 1					No Operation
ORL A,Rn (A) $\leftarrow$ (A) V (Rn)	P				Logical-OR a Byte
ORL A,direct (A) $\leftarrow$ (A) V (direct)	P				Logical-OR a Byte
ORL A,@Ri (A) $\leftarrow$ (A) V ((Ri))	P				Logical-OR a Byte
ORL A,#data (A) $\leftarrow$ (A) V #data	P				Logical-OR a Byte
ORL direct,A (direct) $\leftarrow$ (direct) V (A)					Logical-OR a Byte
ORL direct,#data (direct) $\leftarrow$ (direct) V #data					Logical-OR a Byte
ORL C,bit (A) $\leftarrow$ (A) V (bit)				C	Logical-OR a Bit
ORL C,/bit (A) $\leftarrow$ (A) V /(bit)				C	Logical-OR a Bit
POP direct (direct) $\leftarrow$ ((SP)) (SP $\leftarrow$ (SP) - 1					Pop from Stack

Mnemonic Operation	Flags	Function
	P OV AC C	
PUSH direct		Push to Stack
$(SP) \leftarrow (SP) + 1$		
$((SP)) \leftarrow (direct)$		
RET		Return from Subroutine
$(PC_{15\text{-}8}) \leftarrow ((SP))$		
$(SP) \leftarrow (SP) - 1$		
$(PC_{7\text{-}0}) \leftarrow ((SP))$		
$(SP) \leftarrow (SP) - 1$		
RETI		Return from Interrupt Subroutine
$(PC_{15\text{-}8}) \leftarrow ((SP))$		
$(SP) \leftarrow (SP) - 1$		
$(PC_{7\text{-}0}) \leftarrow ((SP))$		
$(SP) \leftarrow (SP) - 1$		
RL A		Rotate Accumulator Left
$(A_n+1) \leftarrow (A_n)\ \forall\ n=0\text{-}6$		
$(A_0) \leftarrow (A_7)$		
RLC A	P C	Rotate Accumulator Left Through the Carry Flag
$(A_n+1) \leftarrow (A_n)\ \forall\ n=0\text{-}6$		
$(A_0) \leftarrow (C)$		
$(C) \leftarrow (A_7)$		
RR A		Rotate Accumulator Right
$(A_n) \leftarrow (A_n+1)\ \forall\ n=0\text{-}6$		
$(A_7) \leftarrow (A_0)$		
RRC A	P C	Rotate Accumulator Right Through the Carry Flag
$(A_n) \leftarrow (A_n+1)\ \forall\ n=0\text{-}6$		
$(A_7) \leftarrow (C)$		
$(C) \leftarrow (A_0)$		
SETB C		Set Carry Bit
$(C) \leftarrow 1$		
SETB bit		Set Bit
$(bit) \leftarrow 1$		
SJMP rel		Short Jump
$(PC) \leftarrow (PC) + 2$		
$(PC) \leftarrow (PC) + rel$		
SUBB A,Rn	P OV AC C	Subtract with Borrow
$(A) \leftarrow (A) - (C) - (Rn)$		
SUBB A,direct	P OV AC C	Subtract with Borrow
$(A) \leftarrow (A) - (C) - (direct)$		
SUBB A,@Ri	P OV AC C	Subtract with Borrow
$(A) \leftarrow (A) - (C) - ((Ri))$		
SUBB A,#data	P OV AC C	Subtract with Borrow
$(A) \leftarrow (A) - (C) - \#data$		
SWAP A		Swap Nibbles Within Accumulator
$(A_{3\text{-}0}) \longleftrightarrow (A_{7\text{-}4})$		
XCH A,Rn	P	Exchange Accumulator with Byte
$(A) \longleftrightarrow (Rn)$		
XCH A,direct	P	Exchange Accumulator with Byte
$(A) \longleftrightarrow (direct)$		
XCH A,@Ri	P	Exchange Accumulator with Byte
$(A) \longleftrightarrow ((Ri))$		
XCHD A,@Ri	P	Exchange Digit
$(A_{3\text{-}0}) \longleftrightarrow ((Ri)_{3\text{-}0})$		
XRL A,Rn	P	Logical Exclusive-OR Byte
$(A) \leftarrow (A) \oplus (Rn)$		
XRL A,direct	P	Logical Exclusive-OR Byte
$(A) \leftarrow (A) \oplus (direct)$		

Mnemonic Operation	Flags				Function
	P	OV	AC	C	
XRL A,@Ri $(A) \leftarrow (A) \oplus ((Ri))$	P				Logical Exclusive-OR Byte
XRL A,#data $(A) \leftarrow (A) \oplus$ #data	P				Logical Exclusive-OR Byte
XRL direct,A $(direct) \leftarrow (direct) \oplus (A)$					Logical Exclusive-OR Byte
XRL direct,#data $(direct) \leftarrow (direct) \oplus$ #data					Logical Exclusive-OR Byte

14. BASIC Lanuguage Summary

BASIC Commands Summary

	BAUD	Set printer port baud rate
	CALL	Call assembly routine by address
	CLEAR	Clear all variables, arrays, and interrupts
	CLEARI	Clear all interrupts
	CLEARS	Clear stack space
	CLOCK0	Turn real-time clock off
	CLOCK1	Turn real-time clock on
✓	CTRLC0	Disable control-C usage
✓	CTRLC1	Enable control-C usage
✓	DEFASM	Declare an assembly routine as a BASIC keyword
✓	DEFCTRL	Declare a control register as a byte variable
✓	DEFFN	Declare a user-defined function
✓	DEFVAR	Declare a variable at a specific memory address
	DIM	Dimension an array
✓	DISABLE *intr*	Disable derivative microcontroller interrupt
	DO...UNTIL	Loop until a certain condition arises
	DO...WHILE	Loop while a condition is true
✓	ENABLE *intr*	Enable derivative microcontroller interrupt
	END	Halt program execution normally
	FOR...NEXT	Loop with an index variable a finite number of times
	GOSUB...RETURN	Call a BASIC subroutine
	GOTO	Jump to another line of BASIC
	IDLE	Wait for an interrupt to occur
	IF...THEN...ELSE	Conditionally execute a statement
	INPUT	Input information from user
	LD@	Push a floating point value on the stack from memory
	LET	Assign an expression to a variable
	NULL	Configure NULs to be sent after carriage return
	ONERR	If a program error occurs, GOTO a BASIC subroutine
	ONEX1	If external interrupt 1 occurs, GOSUB a BASIC subroutine

	ON GOTO	On an index, GOTO a BASIC line
	ON GOSUB	On an index, GOSUB a BASIC line
	ONTIME	If a timer interrupt occurs, GOSUB a BASIC subroutine
√	ON*intr*	On a derivative microcontroller interrupt, GOSUB a subroutine
	PGM	Program an EPROM
	PH0.	PRINT, outputting numbers in hexadecimal
	PH0.@	PH0 . to a user defined output driver
	PH0.#	PH0 . to the list device
	PH1.	PRINT, outputting numbers in hexadecimal with leading zeros
	PH1.@	PH1. to a user defined output driver
	PH1.#	PH1. to the list device
	POP	Pop value(s) off the top of the floating point argument stack
	PRINT	Output text, numbers, and strings to console device
	PRINT@	PRINT to a user defined output driver
	PRINT#	PRINT to the list device
	PUSH	Push a value on the floating point argument stack
	PWM	Pulse width modulation - sound generator
	READ...DATA	Read a value from a DATA statement with expression(s)
	REM or ;	A comment
	RESTORE	Mark all DATA as unread
	RETI	RETURN from ONTIME or ONEX1
√	SBUFFER *size*	Specifiy serial buffer size
√	SBUFFER OFF	Disable serial buffering
√	SBUFFER ON	Enable serial buffering
√	SBUFFER NOECHO	Disable user keystroke echo
√	SBUFFER ECHO	Enable user keystroke echo
	ST@	Pop a value off floating point argument stack to memory
	STOP	Abort program execution with a message
	STRING	Allocate string storage space
√	TRACE0	Turn off line number tracing
√	TRACE1	Turn on line number tracing
	UI0	Turn off user defined console input routines
	UI1	Turn on user defined console input routines
	UO0	Turn off user defined console output routines
	UO1	Turn on user defined console output routines

BASIC Functions Summary

	ABS(*x*)	Return the absolute value of *x*
	ASC(*c*)	Return the ASCII code for character *c*
	ASC(*s*)	Return the ASCII code at beginning of string *s*
	ASC($(*n*),*x*)	Return the ASCII code for a character in string *n*
	ATN(*x*)	Return the arctangent of *x*
	CBY(*x*)	Return byte value from program memory (ROM)
✓	CHR$(*c*)	Return string of ASCII code *c*
	COS(*x*)	Return the cosine of *x*
	DBY(*x*)	Return/Set contents of internal memory
	EXP(*x*)	Return the value of **e** raised to the *x*
✓	HIGH(*x*)	Return high byte value of *x*
✓	INT(*x*)	Return integer part of *x*
✓	LEFT$(*s,n*)	Return left most n characters of string
✓	LEN(s)	Return length of string s
	LOG(*x*)	Return natural logarithm of *x*
✓	LOW(*x*)	Return low byte value of *x*
✓	MID$(*s,n,m*)	Return range of characters in middle of string s
	NOT(*x*)	Return the logical NOT of *x* (1's complement)
✓	RIGHT$(*s,n*)	Return rightmost characters of string s
	SGN(*x*)	Return sign of *x*
	SIN(*x*)	Return sine of *x*
✓	STR$(*n*)	Convert a number n to a dynamic string
	SQR(*x*)	Return square root of *x*
	TAN(*x*)	Return tangent of *x*
✓	VAL(*s*)	Convert a string to a number
	XBY(*x*)	Return/Set contents of external memory (RAM)

BASIC Special Variables Summary

✓	ERRLINE%	Line number of last error
✓	ERRVALUE%	Error code of last error
✓	FALSE	Return logical false, *0*
	FREE	Return the amount of RAM left
	GET	Return current character on console
	IE	Return/Set value of interrupt enable register
	IP	Return/Set value of interrupt priority register
	LEN	Return length of program

✓	MCON#	Return/set value of memory control reg.
	MTOP	Return/Set the top of memory
	PCON	Return/Set the power control register
	PI	Return the value of π
✓	PORT0#	Return/Set value of P0 I/O port
	PORT1	Return/Set value of P1 I/O port
✓	PORT2#	Return/Set value of P2 I/O port
✓	PORT3#	Return/Set value of P3 I/O port
✓	RAMORG	Return starting location of RAM
	RCAP2	Return/Set value for timer 2's reload/capture registers
	RND	Return a random number
✓	ROMORG	Return starting location of program (ROM)
	T2CON	Return/Set value of timer/counter 2 control register
	TCON	Return/Set value of timer/counter control register
	TIME	Return/Set value of real-time clock
	TIMER0	Return/Set value of timer/counter 0
	TIMER1	Return/Set value of timer/counter 1
	TIMER2	Return/Set value of timer/counter 2
	TMOD	Return/Set timer/counter mode control register
✓	TRUE	Return logical true, *-1*
	XTAL	Return/Set value of system clock speed, in Hz

Operator Summary

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation (not allowed for integer expressions)
`.AND.`	logical AND
`.OR.`	logical OR
`.XOR.`	logical XOR
`.SHL.`	bit-shift left
`.SHR.`	bit-shift right
=	test for equality
<	test for less than
>	test for greater than
<=	test for less than or equal to
>=	test for greater than or equal to
<>	test for inequality

15. Command Line Options

All the command line options for BXC-51 are optional. To deviate from the compiler's defaults, use as many of the options listed below as needed.

Make sure all options precede the BASIC file name. If an option is accompanied by a hex address *addr*, a file name *name*, a baud rate *rate*, or microcontroller name *cpu*, make sure there is no space between the option letter and the item; for example, **-b1200** for 1200 baud.

Debugging On

-g Debugging is normally off. Certain run-time errors will be reported if you specify the **-g** option. For example, if byte values are expected and a larger value is provided (such as an array or string index), the upper byte will be ignored. However, if the **-g** option is specified, a BAD ARGUMENT error will occur. Code generated with the **-g** option will be larger than code without it. If you use the TRACE1 command, the **-g** option will allow you to trace each statement rather than just line changes.

Error Trapping On

-e By default, error trapping via ONERR traps only arithmetic errors. Specify the **-e** option to trap all errors. This traps A-STACK, C-STACK, NO DATA, I-STACK, MEMORY ALLOCATION, errors etc. Use carefully! See the ONERR statement (on page 28) for more information. When used in conjunction with the -2 command line option, not all errors can be trapped. When an error is detected in interpreter code, the compiler cannot intercept it.

Line Numbers Off

-l When compiling your code, the BASIC source code line numbers are encoded into the assembly to facilitate error tracking. This generates extra bytes of code. To save space in final versions of code, specify the **-l** option. The TRACE command does not work with the **-l** option.

Specify the Beginning of Code (ROM)

-p*addr* Without this option, code is generated beginning at address 0H in code memory. With **-p** and no value specified, 0H is assumed. With this option, your code may be relocated to another location, *addr*. At 0H, interrupt handling must be performed; elsewhere, a dummy interrupt space is coded except for external interrupt 1 (at *addr*+13H) and timer overflow 0 (at *addr*+1BH). *If your program uses the real-time clock or ONEX1 and your program does not start at 0H, you must route the appropriate vectors up to your BASIC program so they can be properly handled.* (*addr* is specified in hexadecimal). There are very few reasons (except for debugging with a monitor) for locating program code anywhere but address 0. The microcontroller always begins execution at address 0 upon reset.

Specify the Beginning of Variables (RAM)

-v*addr* Without the **-v***addr* option, variables are assumed to start at 0H. With **-v** and no value specified, 0 is assumed. Use this option to change the starting location of the bottom of RAM usable by this program. This RAM space contains the system parameters, Control Stack, floating point Argument Stack, and program variables. The system variable space starts at *addr*, the user variable space typically starts at *addr*+200H. *addr* is specified in hexadecimal.

If you specify the **-2i** option as well, the system variable space starts at 0H, regardless of the value of *addr*. If *addr* is not at 0H, the *addr* specifies the beginning of user variable space (not *addr* + 200H).

Specify the Upper Limit of Variables (RAM)

-u*addr* With no upper limit, your program tries to determine the size of RAM by clearing RAM until it hits unsocketed RAM or E000H, whichever comes first. With **-u***addr* and no value specified, E000H is the default. If your RAM is always a fixed size or if you wish higher RAM to remain untouched, use the **-u***addr* option to set the limit. The limit is inclusive to the RAM size. If **-u6000** is specified, BASIC will never initialize locations 6001H and up. *addr* is specified in hexadecimal. MTOP is set to this value when the program starts (unless the **-2i** or **-sub** option is present.)

Specify User Console I/O and Stray Interrupts

-c*addr* Without the **-c***addr* option (or with **-c** and no value specified), user console I/O and stray interrupts are assumed to start at 4000H. With this option, you specify where these routines start instead. If *addr* is 0H, then user console I/O and stray interrupts are ignored. This affects the UI1, UI0, UO0, and PRINT@ commands. The **-c0** option reduces final code size. For large programs, you may wish to use this option so the stray interrupt code does not overlap with your BASIC code located around 4000H.

User console, I/O and stray interrupt locations:

Location	Function
addr+03H	Stray EXT0 interrupt
addr+0BH	Stray Timer 0 overflow interrupt
addr+13H	Stray EXT1 interrupt
addr+1BH	Timer1 overflow interrupt
addr+23H	Serial Port interrupt
addr+2BH	Timer2 overflow interrupt

addr+30H
User console output character routine (see UO1, page 42)
addr+33H
User console input character routine (see UI1, page 41)
addr+36H
User console input ready routine (see UI1, page 41)
addr+39H
User reset routine
addr+3CH
User PRINT@ routine (see PRINT@, page 33)

Compile with BASIC Extensions (BXL)

-b*bxl* When writing BASIC source code that depends upon BASIC extensions defined in a BXL (see page 104), inform the compiler with this command line option. Without this command line option, command extensions become syntax errors and function extensions become array variables. Multiple BXL's may be specified in multiple command line options. Note that if the BASIC program does not use any extension defined in the BXL, the compiler extracts no code from the BXL.

Compile into Subroutine

-sub Normally BXC-51 creates a complete assembly program that includes complete microcontroller initialization. To create just an assembly routine which may be CALLed from the MCS BASIC-52 interpreter or from another program, use the **-sub** option. See the section "Converting Your BASIC Program Into a Subroutine" on page 67 for more information.

Generate Code for 8051/31

-1 This is a default command line option. The target program assumes your hardware has the capabilities of an 8051/31 with no interpreter present. It assumes that the board is not initialized and RAM must be cleared. Because the 8051/31 microcontrollers have only two counter/timers, the PGM and PWM statements may conflict with use of the real-time clock (CLOCK0/CLOCK1). See the command explanations for more details. Alternate options are **-2**, **-2i**, **-5**, or **-t***cpu*.

Generate Code for 8052/32

-2 BXC-51 assumes your hardware has the capabilities of an 8051/31. If your program uses the 8052/32 microcontrollers, you must specify this option. Trying to use 8052/32 capabilities without the **-2** option will cause BXC-51 to generate errors and warnings. This option does *not* assume that the BASIC-52 interpreter is present and enabled, so an Assembly support library will be included in your program. Because the 8051/31 microcontrollers have only two counter/timers, the PGM and PWM statements may conflict with use of the real-time clock (CLOCK0/CLOCK1). See the command explanations for more details.

Generate Code for Use with MCS BASIC-52 Interpreter

-2i Normally BXC-51 generates all the assembly code for the 8051/31 as a complete standalone program. However, if you have an 8052 microcontroller with the MCS BASIC-52 interpreter active, BXC-51 generates much less code by utilizing the interpreter's ROM by specifying **-2i**. For important additional information about using the **-2i** command line option, see the section "BXC-51 programs coexisting with MCS BASIC-52" on page 70. You may use the **-sub** command line option to create a subroutine to CALL your BXC-51 program from a BASIC program running in the interpreter. The **-2i** option forces other options to be ignored by BXC-51 (**-b***rate*, **-w**, **-i***addr*, and **-c***addr*) because the interpreter overrides them.

Generate Code for DS5000

-5 If your target microcontroller has a Dallas Semiconductor DS5000 microcontroller, specify this option. This option forces BXC-51 to do all external RAM references through DPTR, leaving Port 2 alone. Additionally, the MCON# variable is defined.

Generate Code for Derivative Microcontroller

-t*cpu* If your target microcontroller is a derivative microcontroller specially configured in the file *cpu*.CPU or BXC51.CPU, specify this option with the appropriate cpu name. Microcontrollers in the 8051 family are supported through special configuration. This support allows BASIC source code to reference additional special variables, handle additional interrupts, and possibly use command and function extensions to BASIC that are specific to *cpu*. As shipped, BXC-51 supports the 8xC550 and 8xC552. See the section "8051 Microcontroller Configuration Commands" on page 84 for details.

Specifying a Different Output Filename

-o*name* When BXC-51 compiles your BASIC (.BAS) file, it generates two output files: one with the file extension .ASM and the other with a .HEX file extension. The base filename is the same as the source code file. However, if you want your .ASM and .HEX files to have a different base filename, use the **-o***name* option. For example, if your program name is SEEBAUD.BAS, you will have SEEBAUD.ASM and SEEBAUD.HEX after running BXC-51. By specifying **-oOUT**, BXC-51 generates the two files OUT.ASM and OUT.HEX.

Code Generation Only

-s To make BXC-51 generate Assembly code but not assemble it, use the **-s** option. Normally, BXC-51 will invoke SXA51 to assemble your code upon successful compilation. This option is useful for examining or modifying the generated code prior to assembly.

Long Assembly Listing

-a To make SXA51 generate a long Assembly list file in a .LST file, specify the **-a** option. This option instructs BXC-51 to pass the **-l** flag to SXA51. This option does not work with the **-s** option.

Additional Assembler Options

-a*opt* To pass additional options to the assembler, use this option. Any text (without spaces) specified as *opt* will be passed to the assembly as a command line option. This option may be specified multiple times.

Invoke Simulator Upon Successful Compile

-sim To invoke a simulator after successfully assembling the BASIC source code, specify this option. The assembler SXA51 invokes the simulator using the command in the DOS environment variable SIM51. If SIM51 is undefined, BXCSIM is assumed (for the BXC-51 Simulator available from Binary Technology, Inc.) Any program can be specified, not just a simulator. The program is invoked with the .HEX or .LST file as the last command line parameter.

Generate Memory Map

-m*name* By default, no memory map is generated. Use this option to create a .MAP file that lists all of your variables, their memory locations, their sizes, and their types of memory (XRAM for external RAM, IRAM for internal RAM). If *name* is not specified, it defaults to the output file's name with the file extension .MAP.

Allow User Reset (at 2090H)

-r When MCS BASIC-52 interpreter first starts up, location 2001H is checked for the value AAH. If the value is present, a user reset occurs by CALLing 2090H. Since this is not always desirable in a program, it is an option. The user reset is called after the SCON, TMOD, TCON, and T2CON registers have been initialized to their defaults and before anything else (such as memory clear) has occurred.

Automatically Setting The Baud

-b*rate* When creating a standalone program, it needs to know the baud rate. Normally, it waits for the user to hit the space bar to calculate the baud rate. If you specify a baud rate with this option, the rate will automatically be set and you will not need to hit space. With -b and no value specified, 1200 baud is the default. This is a substitute for the BASIC-52 interpreter's PROG1 or PROG2 command. No error checking is performed on your rate so preposterous rates could cause communications problems. **-b** also allows you the freedom of using any baud rate that your host machine might have. With the **-2i** option, the baud rate is assumed to have been set by the interpreter.

User Initialization Routine

-i*addr* Once the complete BASIC environment is set up (with the exception of the baud rate if it is waiting for the space character), a user initialization routine may be called if this option is specified. If no address is specified, 4039H is assumed (as with the PROG6 command). This is NOT conditional: it will be called every time your program starts. The default is not to call the initialization routine. *addr* is specified in hexadecimal. With the **-2i** option, this option is ignored.

Exit Address

-x*addr* Normally, upon completion of your program (by error, STOP, or END), the microcontroller loops indefinitely, waiting for a system reset. To exit to a particular address, specify the address with the **-x**addr option. With **-x** and no value specified, 210EH is the default address (the entry point for M/DP V3). This is an added convenience if you are running a monitor such as M/DP. If you are working with the M/DP monitor, you can return to it by using the **-x210e** for version 3.0 and **-x5** for version 2.0. This option is ignored if **-sub** option is used.

Warm Start

-w This option is not available when the **-u***addr* command line option is specified. When creating a standalone program, your program will clear all external RAM when started. Using the **-w** option suppresses this initialization, allowing your program to warm restart. This does not suppress the autobaud option (**-b*rate***). Warm restart is only performed if the value A5H is stored in external RAM memory address RAMORG+5FH. If the value A5H is present, your program will only clear memory up to MTOP and not beyond. Use this flag to protect the contents of memory above MTOP during a warm restart.

Example 1

```
bxc51 -p4000 -v6000 -u6400 test
```

will compile the BASIC source `TEST.BAS` to begin execution at 4000H with variables stored in RAM beginning at 6000H through 6400H.

Example 2

```
BXC51 -p4000 -2i -sub -v1F00H test2
```

will compile the BASIC source TEST2.BAS into a subroutine at 4000H CALLable by a BASIC program running in the interpreter. For variables starting at 1F00H you will need MTOP = 1F00H in your MCS BASIC-52 interpreter program to reserve room for your BXC-51 compiled program variables. Otherwise, the MCS BASIC-52 interpreter variables may overlap with the BXC-51 compiled variables, which may cause undesirable results.

If you find that you are using certain compiler command line options repeatedly, you can save yourself some typing by using the DOS environment variable BXCFLAGS. If you always use the compiler options of example 1 above, you could type

```
SET BXCFLAGS=-p4000 -v6000 -u6400
```

when you start working (no spaces on either side of the =). Thereafter, whenever you type

```
BXC51 test
```

BXC-51 will automatically assume the two options **-p4000** and **-v6000**. To verify the `BXCFLAGS options you specified, each time BXC51 is invoked, it displays the full invocation with BXCFLAGS options included. Add` `this SET command to your AUTOEXEC.BAT` file so the flags are always the default. Typed command line options override the environment variable's options.

16. Compiler Error Messages

When compiling your program, errors are bound to arise. The compiler marks the location of the error with a caret and displays the reason for the error. The following explains those errors and possible causes that BXC-51 reports. For example,

```
     640 FOR A$=1 to 10
             ^ Illegal FOR index variable
```

`ELSE without preceding IF`
An ELSE statement must be on the same line as its matching IF. For multiple line IF statements, an ELSE statement must be matched to a pervious IF statement. Check your embedded IF statements for unmatched IF. See page 24.

`Illegal FOR index variable`
Only floating point, integer, and byte variables may be used as the index variable of a FOR loop. See page 23.

`You cannot change ROM`
The CBY() function is read-only. You cannot assign values to it. If you use ROM addressable RAM, use the XBY() function instead. See CBY function on page 46.

`Too many parentheses or expression too complicated`
Only 20 levels of parentheses nesting are supported. [This error is no longer appears.]

`RND is not an integer function`
This keyword does not belong in integer expressions. See page 58.

`XTAL is not an integer variable`
This key word does not belong in integer expressions. See page 61.

`TIME is not an integer variable`
This key word does not belong in integer expressions. See page 59.

`Bit address too high`
Only 0-7 are allowed for byte variables and 0-15 for integer variables. See page 9 for details on bit data type.

`String not allowed here, expecting a number variable`
The READ and POP commands are for numerical variables only.

`Out of byte variable memory space`
There is very limited byte variable space on the 8051. There are only 10 bytes available on 8051/31, DS5000, and some derivative microcontrollers; and 51 bytes on 8052/32 microcontroller. If you are using byte arrays, dimension them smaller. This error may appear when using byte variables asscociated with a derivative microcontroller when the **-t***cpu* command line option is missing or incorrect.

`Byte array redimensioned to different size`
You may not redimension a byte array, particularly to a different size.

`Control register not in correct range (128...255)`
All control registers defined by DEFCTRL must have an address from 80H to FFH. See page 20.

`Memory mapped variables cannot be DIMensioned`
The dimension size of a memory mapped variable is irrelevant. Correct syntax is DEFVAR A()@2000H. See page 20 for DEFVAR command, page 21 for DIM command.

`Expecting a line number or label`
A line number or label is expected where the caret is pointing. GOTO and GOSUB require it. See page 23.

`Unrecognizable command`
BXC-51 has tried to determine what you mean, but a syntax error is preventing it from deciphering the command.

`Expecting an integer expression`
A numerical expression (formula) is expected which evaluates to an integer. Floating point variables and functions are not allowed unless contained within INT() function (see page 47).

`) expected`
You are missing a matching right parenthesis to a previous open parenthesis or you have provided more information within the parentheses than is expected.

`Integer constant expected`
You are not allowed to put an integer expression here at the caret. The integer value must be specified exactly as a decimal, hexadecimal, or binary number.

`( expected`
You have used a function and it requires a parenthetical parameter.

`Variable expected`
The command in question requires you to provide a variable name. This variable may be altered.

`, expected`
There are less parameters that are expected. Insert a comma and provide the parameters following it.

`Data item expected`
A DATA statement can only have numerical expressions. Strings are not allowed. See page 18.

`Array variable expected`
The DIM command only operates on array variables with dimension sizes (e.g., A(15).) See page 21.

`H expected for hexadecimal constant`
Either you have a variable name that starts with a digit or you are missing the 'H' which is required at the end of a hexadecimal constant.

`Expression expected`
A floating point, integer, or byte expression is expected at the caret.

`= expected`
BXC-51 thinks you are trying to assign a value to a variable, but the = is missing. You may have extraneous text at the end of your variable name.

`TO expected`
A FOR statement requires a TO component; it is expected at the caret. See page 23.

`GOTO or GOSUB expected`
The ON statement must specify either GOTO or GOSUB at the caret. You may have extraneous text which should be removed.

`# expected`
USING only allows '#' and '.' as format characters.

`0 or 1 expected`
Enter either a 0 or 1 after the CLOCK, TRACE, and CTRLC commands. The appropriate commands are CLOCK0, CLOCK1, TRACE0, TRACE1, CTRLC0, and CTRLC1.

`Another expression term expected`
You have an operator (such as +) which requires a valid expression term on the right side of it, e.g. A*, 3+, and A%/ require another term such as A*B, 3+X, A%/2.

`String expression expected`
Only a string expression is valid at the caret.

`: expected`
There is extraneous text at the end of the command; a colon is expected to separate commands.

`@ expected`
To specify the address of a DEFVAR, DEFASM, and DEFCTRL command, @ must precede the address. There may be extraneous text where @ is expected. See page 19.

`Invalid DEF type`
Only the commands DEFVAR, DEFASM, DEFCTRL, and DEFFN are allowed as DEF commands. See page 19.

`Identifier/name expected`
You must specify a valid name for the Assembly routine's name in DEFASM. The name should only contain alphanumeric characters. See page 19.

`+ expected`
The only valid operator for strings is +.

`$ expected`
The functions that return strings must have a $(i.e., MID$(), LEFT$(), etc.).

`} expected`
A line label may only be alphanumeric characters surrounded by braces (i.e., {LABEL}; a closing brace is expected at caret.

`Duplicate line label`
You have already used this line label on a previous line. Use a different name.

`Duplicate line number`
You have already used this line number on a previous line. Use a different number.

`Too many embedded IFs`
Only a maximum of 40 IF-THEN statements can be embedded within each other. You are probably missing many ENDIF statements. See page 24.

`ENDIF without a preceding IF`
You have specified ENDIF and the compiler cannot find the immediately preceeding IF. Make sure you only have one ENDIF per IF statement. Also make sure that the ENDIF is near the IF statement. The ENDIF may not be inside some subroutine. See page 24.

`Expecting THEN`
You have started an IF-THEN statement and the THEN keyword is missing. The THEN keyword must appear on the same line with the IF keyword. See page 24.

`Exponentiation not allowed in integer/byte expression`
The ** operator is not allowed for integer and byte epxressions. You will have to enumerate the operation. For example, use A%*A% instead of A%**2, use A%*A%*A% instead of A%**3, etc. For fractional exponents, use INT(A%**.5).

`% expected`
The per cent symbol is expected here.

`Interrupt name expected (verify correct -t command line option)`
Either ON*intr*, ENABLE *intr*, or DISABLE *intr* command is being used and *intr* is unknown. Make sure that you have specified the correct derivative microcontroller with the **-t***cpu* command line option. You may need to check the .CPU file to get the exact spelling for the interrupt name. (Upper/Lower case is fine.)

17. Run-Time Error Messages

When your compiled program is running, any number of error conditions may be detected and an error reported. The following section lists those errors and describes why they occurred.

`ARITH. UNDERFLOW`
When performing floating point operations, if a temporary result is smaller than the smallest number that BASIC can represent, an arithmetic underflow error is reported. Resulting numbers smaller than 1E-128 will be in error.

`ARITH. OVERFLOW`
When performing floating point operations, if a temporary result is larger than the largest number that BASIC can represent, an arithmetic overflow error is reported. Resulting numbers equal to or larger than 1E+128 will be in error.

`ARRAY SIZE`
When accessing an array index that is out of the array bounds (and the **-g** command line option was specified), then this error appears. When attempting to re-dimension an array, this error appears. You may DIMension a variable only once. The variable cannot be used before it is DIMensioned otherwise the implied DIMension is 10.

`A-STACK`
This error appears when the Argument Stack gets too large or when a ST@ or POP command attempts to take a value of an empty Argument Stack. Avoid using LD@ and PUSH so much because they are using too much stack space.

`BAD ARGUMENT`
This error appears when a value is used outside an acceptable range. Typically, a value is expected in the range of 0 to 255 and a larger (or smaller) value is given.

`C-STACK`
This error appears when too many control structures are on the Control Stack. This can happen if too many GOSUBs have occurred or if you GOTO out of FOR or DO loops so the loops never finish. If this error happens at the very beginning of your program, it could mean that the BASIC environment is not initialized correctly.

`DIVIDE BY ZERO`
This error appears if you attempt to divide by the number 0 in a floating point, integer, or byte expression.

`I-STACK`
This error appears if an expression is too complicated and forces the internal system stack to overflow.

`NO DATA`
This error appears when your program attempts to READ more data than present. You may need to execute RESTORE to begin reading the DATA all over again.

`NO FN DEF`
This message appears if you are attempting to use a user defined function which has not yet been defined. All user defined functions begin with FN and appear as FN*name*(). Array variables that begin with FN must be renamed to avoid confusion.

`ARG MISMATCH`
This message appears when the wrong number of arguments are passed to a user defined function. If a user defined function is defined to take 3 parameters, any other number of parameters causes this error.

`MEMORY ALLOCATION`
An attempt was made to use some free memory, but there is no free memory left. At startup, this error appears if upper and lower bounds were incorrectly set with the **-v***addr* and **-u***addr* command line options and a part of the range is missing a RAM chip. If this error occurs when manipulating dynamic strings it may be an indication of either (a) an erroneous or changed MTOP value or (b) out of memory.

`PROGRAMMING`
An error occurred during the PGM command when it was attempting to program the EPROM.

`STRING TOO LARGE`
When constructing a string, if the string length exceeds 255 characters, this error arises.

We are constantly looking for ways to improve our products and would appreciate your comments and suggestions. Please send them to:

Binary Technology, Inc.
P.O. Box 541
Carlisle, MA 01741 USA

or send a fax to

(508) 369-9549

Below is a list of other products mentioned in this manual which are available from Binary Technology, Inc.

"The Intel MCS BASIC-52 Users Manual" Intel Order #270010-003

"BASIC-52 Programming" by Systronix

BASIC Toolkit (BTK) by Binary Technology, Inc.

Infinitor by TAVVE Software Co.

BXC-51 IDE by TAVVE Software Co.

M/DP by Binary Technology, Inc.

Kermit by New York University

QComm by TAVVE Software Co.

BXC-51 Simulator by TAVVE Software Co.

BXC-51 Library Toolkit by TAVVE Software Co.

BXL-Encode by TAVVE Software Co.