intel

MCS® BASIC-52 USERS MANUAL

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 270010-003

CHAPTER 1
Introduction

1.1 INTRODUCTION TO MCS BASIC-52

Welcome to MCS® BASIC-52. This program functions as a BASIC interpreter occupying 8K of ROM in INTEL's 8052AH microcontroller. MCS BASIC-52 provides most of the features of "standard" BASICS, plus many additional features that apply to control environments and to the architecture of the 8052AH.

The design goal of MCS BASIC-52 was to develop a software program that would make it easy for a hardware/software designer to interact with the 8052 device; but, at the same time not limit the designer to the slow and sometimes awkward constructs of BASIC. This program is not a "toy" like many of the so called TINY BASICS. It is a powerful software tool that can significantly reduce the design time of many projects. MCS BASIC-52 is ideal for so called imbedded systems, where terminals are not attached to system, but the system controls and manipulates equipment and data.

MCS BASIC-52 offers many unique hardware and software features, including the ability to store and execute the user program out of an EPROM, the ability to process interrupts within the constructs of a BASIC program, plus an accurate real time clock. In addition, the arithmetic routines and I/O routines contained in MCS BASIC-52 can be accessed with assembly language CALL routines. This feature can be used to eliminate the need for the user to write these sometimes difficult and tedious programs.

All of the above are covered in this document. This is NOT a "How to Write Basic Programs" manual. Many excellent texts on this subject have been produced. Your local computer store can recommend many such texts.

The descriptions of many of the statements in this manual involve rather detailed discussions that relate to interfacing MCS BASIC-52 to assembly language programs. If the user is not interested in using assembly language with MCS BASIC-52 these discussions may be ignored. If you are only interested in programming the MCS BASIC-52 device in BASIC, you can treat all statements the same way they would be in any standard BASIC interpreter.

In reading this manual, you will find that some information may be repeated two or three times. This is not an accident. Years of experience have proven that one of the most frustrating experiences one encounters with manuals is trying to find a particular piece of information that the reader knows is in the manual, but can't remember where.

1.2 GETTING STARTED

If you are like most engineers, technicians, hobbyists and humans, and don't like to read manuals, this section is for you. The purpose of this section is to get you off on the right foot. If you are in the High Anxiety Mode and just want to see if the darn chip works, wire the device in the minimum hardware configuration as suggested in the Hardware Configuration chapter of this manual, apply power, and watch what happens. NOTHING! That's because after power is applied to the MCS BASIC-52 device, the program initializes the 8052AH hardware and goes into an AUTO-BAUD search routine. You must touch the space bar on the serial input device in order to get MCS BASIC-52 to SIGN ON. The message that will appear is *MCS-51 BASIC Vx.x*. If a space character is not the first character sent to the MCS BASIC-51 device after reset, you can spend a lot of time trying to figure out what went wrong. So do yourself a favor, read this section and touch the space bar before you call your local Intel Field Applications Engineer. We received a number of questions asking how the AUTO-BAUD search routine worked. As a result this routine is listed in Chapter 11 of this manual.

1.3 WHAT HAPPENS AFTER RESET?

After RESET, MCS BASIC-52:

1) Clears the INTERNAL 8052AH memory

2) Initializes the internal registers and pointers

3) Tests, clears, and sizes the EXTERNAL memory

BASIC then assigns the top of EXTERNAL RANDOM ACCESS MEMORY to the SYSTEM CONTROL VALUE — MTOP and uses this number as the random number seed. BASIC assigns the default crystal value, 11.0592 Mhz, to the SYSTEM CONTROL VALUE — XTAL and uses this default value to calculate all time dependent functions, such as the EPROM programming timer and the interrupt driven REAL TIME CLOCK. Finally, BASIC checks external memory location 8000H to see if the baud rate information is stored. If the baud rate is stored, MCS BASIC-52 initializes the baud rate generator (the 8052AH's SPECIAL FUNCTION REGISTER — T2CON) with this information and signs on. If it isn't stored, BASIC interrogates the serial port input and waits for a space character to be typed. This sounds like a lot, but on the 8052AH, it doesn't take much time.

1.3 WHAT HAPPENS AFTER RESET?

MCS BASIC-52 initializes the 8052AH's Special Function Registers, TMOD, TCON, and T2CON with the following values:

TCON – 244 (0F4H)

TMOD – 16 (10H)

T2CON – 52 (34H)

After Reset, the console device should display the following:

MCS-51(tm) BASIC Vx.x
READY

To see if everything is OK after Reset, type the following:

>PRINT XTAL, TMOD, TCON, T2CON
(BASIC should respond)
11059200 16 244 52

If it does, everything is working properly. If it does not make sure that the external memory, the serial port, and the oscillator are connected and working. Hardware debug begins here.

In the Appendix of this manual is a QUICK REFERENCE GUIDE. It provides a short description of all of the COMMANDS and STATEMENTS implemented in MCS BASIC-52. You might want to use this section to gain a quick understanding of the MCS BASIC-52 software package. Those of you who are familiar with the BASIC language will notice that most of the STATEMENTS and COMMANDS used in MCS BASIC-52 are "standard," so getting started should not be a problem.

1.4 DEFINITION OF TERMS:

COMMANDS:

MCS BASIC-52 operates in two modes, the COMMAND or direct mode and the interpreter or RUN mode. MCS BASIC-52 Commands can only be entered when the processor is in the COMMAND or direct mode. MCS BASIC-52 takes immediate action after a command has been entered. This document will use the terms RUN MODE and COMMAND MODE to refer to the two different modes of operation.

STATEMENTS:

A BASIC program is comprised of statements. Every statement begins with a line number, followed by the statement body, and terminated with a Carriage Return (cr), or a colon (:) in the case of multiple statements per line. Some statements can be executed in the COMMAND MODE, others cannot. The DESCRIPTION OF STATEMENTS section of this manual describes whether a statement can be executed in the COMMAND mode or only in the RUN mode.

There are three general types of statements in MCS BASIC-52: ASSIGNMENTS, INPUT/OUTPUT, and CONTROL. The DESCRIPTION OF STATEMENT section of this manual explains what type is associated with each statement.

- EVERY line in a program must have a statement line number ranging between 0 and 65535 inclusive.
- Statement numbers are used by BASIC to order the program statements sequentially.
- In any program, a statement number can be used only once.
- Statements need not be entered in numerical order, because BASIC will automatically order them in ascending order.
- A statement may contain no more than 72 characters in Version 1.0 and no more than 79 in Version 1.1.
- Blanks (spaces) are ignored by BASIC and BASIC automatically inserts blanks during LIST.
- More than one statement can be put on a line, if separated by a colon (:), but only one statement number is allowed per line.

FORMAT STATEMENTS:

Format Statements may only be used within the PRINT STATEMENT. The format statements include TAB([expr]), SPC([expr]), USING(special symbols), and CR (carriage return with no line feed). Details of the format statements are provided in the description of the PRINT STATEMENT section of this manual.

1.4 DEFINITION OF TERMS

DATA FORMAT:

The range of numbers that can be represented in MCS BASIC-52 is:

$\pm 1E-127$ to $\pm .99999999E+127$.

There are eight digits of significance in MCS BASIC-52. Numbers are internally rounded to fit this precision. Numbers may be entered and displayed in four formats: integer, decimal, hexadecimal, and exponential. EXAMPLE: 129, 34.98, OA6EH, 1.23456E+3

INTEGERS:

In MCS BASIC-52, integers are numbers that ranges from 0 to 65535 or 0FFFFH. All integers can be entered in either decimal or hexadecimal format and all hexadecimal numbers must begin with a valid digit (e.g. the number A000H must be entered 0A000H). When an operator, such as .AND. requires an integer, MCS BASIC-52 will truncate the fraction portion of number so it will fit the integer format. All line numbers used by MCS BASIC-52 are integers. This document will refer to integers and line numbers, respectively in the following manner:

[integer] — [ln num]

NOTE — Throughout this document the brackets [] are used only to indicate an integer, constant, etc. They are NOT entered when typing the actual number or variable.

CONSTANTS:

A constant is a real number that ranges from $\pm 1\ E-127$ to $\pm .99999999E+127$. A constant, of course, can be an integer. This document will refer to constants in the following manner:

[const]

OPERATORS:

An operator performs a pre-defined operation on variables and/or constants. Operators require either one or two operands. Typical two operand or dyadic operators include ADD (+), SUBTRACT (−), MULTIPLY (*), and DIVIDE (/). Operators that require only one operand are often referred to as UNARY OPERATORS. Some typical UNARY OPERATORS are SIN, COS, and ABS.

1.4 DEFINITION OF TERMS

VARIABLES:

In Version 1.0 of MCS BASIC-52 a variable could be defined as either a letter, (i.e. A, X, I), a letter followed by a number, (i.e. Q1, T7, L3), a letter followed by a ONE DIMENSIONED expression, (i.e. J(4), G(A+6), I(10*SIN(X))), or a letter followed by a number followed by a ONE DIMENSIONED expression (i.e. A1(8), P7(DBY(9)), W8(A+B). In Version 1.1 variables can be defined in the same manner as in Version 1.0, however variables may also contain up to 8 letters or numbers including the underline character. This permits the user to use a more descriptive name for a given variable. Examples of valid variables in Version 1.1 of MCS BASIC-52 are as follows:

FRED VOLTAGE1 I_I1 ARRAY(ELE_1)

When using the expanded variable names available in Version 1.1 of MCS BASIC-52 it is important to note that 1) It takes longer for MCS BASIC-52 to process these expanded variable names and 2) *The user may not use any keyword as part of a variable name* (i.e. the variables TABLE and DIET could not be used because TAB and IE are reserved words). BAD SYNTAX ERRORS will be generated if the user attempts to define a variable that contains a reserved word.

Variables that include a ONE DIMENSIONED expression [expr] are often referred to as DIMENSIONED or ARRAYED variables. Variables that only involve a letter or a letter and a number are called SCALAR variables. The details concerning DIMENSIONED variables are covered in the description of the STATEMENT ROUTINE DIM. This document will refer to VARIABLES as:

[var].

MCS BASIC-52 allocates variables in a "static" manner. That means each time a variable is used, BASIC allocates a portion of memory (8 bytes) specifically for that variable. This memory cannot be de-allocated on a variable by variable basis. That means if you execute a statement like Q=3, later on you cannot tell BASIC that the variable Q no longer exists so, please "free up" the 8 bytes of memory that belong to Q. Sorry, it doesn't work this way. The only way the user can clear the memory that is allocated to variables is to execute a CLEAR STATEMENT. This Statement "frees" all memory allocated to variables.

IMPORTANT NOTE:

Relative to a dimensioned variable, it takes MCS BASIC-52 a lot less time to find a scalar variable. That's because there is no expression to evaluate in a scalar variable. So, if you want to make a program run as fast as possible, use dimensioned variables only when you have to. Use scalars for intermediate variables, then assign the final result to a dimensioned variable.

EXPRESSIONS:

An expression is a logical mathematical formula that involves OPERATORS (both unary and dyadic), CONSTANTS, and VARIABLES. Expressions can be simple or quite complex, i.e. 12*EXP(A)/100, H(1)+55, or (SIN(A)*SIN(A)+COS(A)*COS(A))/2. A "stand alone" variable [var] or constant [const] is also considered an EXPRESSION. This document will refer to EXPRESSIONS as:

[expr].

1.4 DEFINITION OF TERMS

RELATIONAL EXPRESSIONS:

Relational expressions involve the operators EQUAL (=), NOT EQUAL (<>), GREATER THAN (>), LESS THAN (<), GREATER THAN OR EQUAL TO (>=) and LESS THAN OR EQUAL TO (<=). They are used in control statements to "test" a condition (i.e. IF A < 100 THEN . . .). Relational expressions ALWAYS REQUIRE TWO OPERANDS. This document will refer to RELATIONAL EXPRESSIONS as:

[rel expr].

SPECIAL FUNCTION OPERATORS:

Virtually all of the special function registers on the 8052AH can be accessed by using the special function operators. The exceptions are PORTS 0, 2 and 3 and non-I/O associated registers such as ACC, B, and PSW. Other SPECIAL FUNCTION OPERATORS are XTAL and TIME. Details of the SPECIAL FUNCTION OPERATORS are covered in the section SPECIAL FUNCTION OPERATORS.

SYSTEM CONTROL VALUES:

The system control values include the following: LEN (which returns the length of the program), FREE (which designates how many bytes of RAM are not used that are allocated to BASIC), and MTOP (which is the last memory location that is assigned to BASIC). Details of the system control values are covered in the section SYSTEM CONTROL VALUES.

1.4 DEFINITION OF TERMS

STACK STRUCTURE:

MCS BASIC-52 reserves the first 512 bytes of EXTERNAL DATA MEMORY to implement two "software" stacks. These are the control stack and the arithmetic stack or ARGUMENT STACK. Understanding how the stacks work in MCS BASIC-52 is NOT NECESSARY if the user wishes only to program in BASIC. However, understanding the stack structure is necessary if the user wishes to link MCS BASIC-52 to ASSEMBLY language routines. The details of how to link to assembly language are covered in the ASSEMBLY LANGUAGE LINKAGE section of this manual.

CONTROL STACK — The control stack occupies locations 96 (60H) through 254 (OFEH) in external ram memory. This memory is used to store all information associated with loop control (i.e. DO — WHILE, DO — UNTIL, and FOR — NEXT) and basic subroutines (GOSUB). The stack is initialized to 254 (OFEH) and "grows down."

ARGUMENT STACK — The ARGUMENT STACK occupies locations 301 (12DH) through 510 (1FEH) in external ram memory. This stack stores all constants that MCS BASIC-52 is currently using. Operations such as ADD, SUBTRACT, MULTIPLY, and DIVIDE always operate on the first two numbers on the ARGUMENT STACK and return the result to the ARGUMENT STACK. The argument stack is initialized to 510 (1FEH) and "grows down" as more values are placed on the ARGUMENT STACK. Each floating point number placed on the ARGUMENT STACK requires 6 BYTES of storage.

INTERNAL STACK — The stack pointer on the 8052AH (SPECIAL FUNCTION REGISTER, SP) is initialized to 77 (4DH). The 8052AH's stack pointer "grows up" as values are placed on the stack. In MCS BASIC-52 the user has the option of placing the 8052AH's STACK POINTER anywhere (above location 77) in internal memory. The details of how to do this are covered in the ASSEMBLY LANGUAGE LINKAGE section of this manual.

LINE EDITOR:

MCS BASIC-52 contains a minimum level line editor. Once a line is entered the user may not change the line without re-typing the line. However, it is possible to delete characters while a line is in the process of being entered. This is done by entering a RUBOUT or DELETE character (7FH). The RUBOUT character will cause the last character entered to be erased from the text input buffer. Additionally, a control-D will cause the entire line to be erased. In Version 1.1 of MCS BASIC-52, Control-Q (X-ON) and Control S (X-OFF) recognition have been added to the serial port. The user is cautioned not to accidently type a Control-S when entering information because the MCS BASIC-52 will no longer respond to the console device. Control-Q is used to bring the console device back to life after Control-S is typed.

NOTE — In this document a carriage return is indicated by the symbol (cr). The carriage return is the RETURN key on most keyboards.

1.5 WHAT'S THE DIFFERENCE BETWEEN V1.0 AND V1.1

Thanks to feedback from many of the users of MCS BASIC-52, a number of changes and additions have been made to Version 1.1. All of these changes and additions were made to enhance the usefulness of the product and yet retain 100% compatibility, well almost 100% compatibility with the original version. To make things simple, all of the changes will be mentioned here and a reference will be provided as to where the reader of this manual may obtain more information about the change or addition.

The only change that has been made to V1.1 that is not compatible with V1.0 is with the IF_THEN_ELSE STATEMENT when used with multiple statements per line. In V1.0, the following two examples would function in the same manner.

EXAMPLE 1:

```
10 IF A=B THEN C=A : A=A/2 : GOTO 100
20 PRINT A
```

EXAMPLE 2:

```
10 IF A=B THEN C=A
12 A=A/2
14 GOTO 100
20 PRINT A
```

They function in the same manner because V1.0 treats the delimiter (:) exactly the same as a carriage return (cr) in every case. However, V1.1 executes the remainder of the line if and only if the test A = B proves to be true. This means in EXAMPLE 1 IF A did equal B, V1.1 would then set C = A, then set A = A/2, then execute line 100. IF A did not equal B, V1.1 would then PRINT A and ignore the statements C = A : A = A/2 : GOTO 100. V1.1 will execute EXAMPLE 2 exactly the same way as V1.0. This same logical interpretation holds true for the ELSE statement as well. This example dictates a simple rule for maintaining IF_THEN_ELSE compatability between the two versions. IF THE DELIMITER (:) IS NOT USED IN AN IF_THEN_ELSE STATEMENT, V1.0 AND V1.1 WILL TREAT THE STATEMENTS IN THE SAME MANNER!!

This change was made because most users of MCS BASIC-52 felt that the V1.1 interpretation of this statement was more useful because fewer GOTO statements need be employed in a typical program.

Additionally, V1.1 accepts inputs in either lower or upper case, whereas V1.0 converted lower case to upper case. V1.1 will however, convert keywords from lower case to upper case during the LISTing of a program. Finally, MCS BASIC-52 V1.1 runs between 2% and 10% faster than V1.0. Typically, this should not cause any problems.

As far as the user is concerned, these are the only changes that may affect the operation of a typical program. Now, on to the additions.

1.5 WHAT'S THE DIFFERENCE BETWEEN V1.0 AND V1.1

ADDITIONS TO MCS BASIC-52 V1.1:

- X-ON (control Q) and X-OFF (control S) have been added. These permit the user to "stop" (control S) and start (control Q) the display of characters during a LIST or PRINT. This feature also permits synchronization with external I/O (input/output) devices. The X-OFF (control S) functions on a line by line basis, not on a character by character basis.

- Five new statements have been added. These include IDLE, LD@, ST@, PGM, and RROM. Details of these statements are listed under the DESCRIPTION OF STATEMENTS section of this manual.

- Six new RESET options have been provided. They permit the user to assign the top of memory (MTOP) during reset, and allow the user to write specific RESET programs in assembly language. Additionally, they provide an option where the memory WILL NOT be cleared during RESET. More information on the specific RESET OPTIONS is detailed in the DESCRIPTIONS OF EPROM FILE COMMANDS under PROG1, PROG2, PROG3, PROG4, PROG5, and PROG6 COMMANDS and in Chapter 11 of this manual.

- The Timing of the EPROM programming algorithm has been significantly relaxed between the various strobes required for the EPROM programming function. This relaxed timing permits the user to program devices such as the 8751H and the 8748/9 using the EPROM programming capabilities of the MCS BASIC-52 device. Details of the timing changes are in Chapter 10 of this manual.

- During EPROM programming, the INT0/DMA REQUEST pin of the MCS BASIC-52 device is treated as a ready input pin. This allows for a simple direct connection to EEPROM devices such as the 2817A. For normal EPROM programming, INT0 must be kept high or the programming hangs up. Details concerning the use of EEPROMS with the MCS BASIC-52 device are provided in Chapter 10 of this manual.

- A RUN TRAP option has been provided. This option traps the MCS BASIC-52 interpreter in the program RUN mode and will not permit the user to exit this mode. Details of this option are covered in Chapter 8.4 of this manual.

- A user STATEMENT/COMMAND expansion option has been provided. This permits the user to easily add new or custom STATEMENTS and COMMANDS to MCS BASIC-52. Details of this option are covered in Chapter 12 of this manual.

1.5 WHAT'S THE DIFFERENCE BETWEEN V1.0 AND V1.1

ADDITIONS TO MCS BASIC-52 V1.1:

- A number of new assembly language user OP BYTES have been added. These permit the user to make better use of the STATEMENT/COMMAND expansion option previously described. Details of these new OP BYTES are presented in Chapter 9.6 of this manual.

- The length of the input buffer has been increased from 72 characters to 79 characters and the ERROR: LINE TOO LONG has been eliminated. Instead, when the cursor reaches the 79th position a bell character will be echoed everytime the user attempts to enter another character.

- A new variation on the PRINT (including PH0. and PH1.) and LIST statements have been added. This new option is evoked with an @ character (EXAMPLE: PRINT@ or LIST@) and permits the user to write specific output drivers for these statements and commands. When the @ PRINT or LIST is evoked, MCS BASIC-52 CALLS external code memory location 403CH. The user must put the specific output driver in this location. More details of this option is in Description of Statements section of this manual.

- The control stack has been made more "forgiving." This means that the user can execute a GOSUB to a subroutine that contains a FOR-NEXT loop and return from the subroutine without completing the FOR-NEXT loop. Version 1.0 would yield a C-STACK ERROR under these circumstances, V1.1 yields no error.

- The question mark character ? is interpreted as a PRINT statement (EXAMPLE: (PRINT 10+20 is the same as ? 10+20). The symbols P. remains a shorthand notation for PRINT just as in V1.0.

- The FOR-NEXT statement can be executed in the direct mode. This lets the user write short routines in the DIRECT MODE to, for example, display a region of memory (EXAMPLE: FOR I=200H to 210H: PH0. XBY(I): NEXT I)

- Variables can be up to 8 characters in length, however, only the first character, the last character, and the total number of characters are of signifance. This lets the user better describe variables that are used in a program. Chapter 1.4 details the limitations on the expanded variables in Version 1.1.

1.5 WHAT'S THE DIFFERENCE BETWEEN V1.0 AND V1.1

ADDITIONS TO MCS BASIC-52 V1.1:

- The CALL statement vectors to locations 4100H through 41FFH if the CALL integer is between 0 and 7FH inclusive. This means that CALL 0 will vector to location 4100H, CALL 1 to location 4102H, CALL 2 to location 4104H, etc. This permits the user to easily generate assembly language CALL tables by using simple integers with the CALL statement. Anyway, CALL 0 through CALL 1FFFH was not too useful because these numbers vectored into the MCS BASIC-52 ROM.

- The error message anomaly for an invalid line number on a GOTO or GOSUB STATEMENT has been eliminated on V1.1 of MCS BASIC-52. The correct line number is now processed and displayed by the error processor.

- The FOR-TO-{STEP}-NEXT statement can be executed in the COMMAND MODE in version 1.1 of MCS BASIC-52. Additionally, the NEXT statement does not require a variable in version 1.1. Details of these features are covered in the Description of Statements section of this manual.

- The REM statement can be executed in the COMMAND MODE. If the user is employing some type of UPLOAD/DOWNLOAD routine with a computer, this lets the user insert REM statements, without line numbers in the text and not download them to the MCS BASIC-52 device. This helps to conserve memory.

- Version 1.1 is also a little less "crashable" than version 1.0. This is due to a more extensive "type checking" on control transfer routines (i.e. GOTO, GOSUB).

CHAPTER 2
Description of Commands

2.1 DESCRIPTION OF COMMANDS

COMMAND: RUN(cr)

ACTION TAKEN:

After RUN(cr) is typed all variables are set equal to zero, all BASIC evoked interrupts are cleared and program execution begins with the first line number of the selected program. The RUN command and the GOTO statement are the only way the user can place the MCS BASIC-52 interpreter into the RUN mode from the COMMAND mode. Program execution may be terminated at any time by typing a control-C on the console device.

VARIATIONS:

Unlike some Basic interpreters that allow a line number to follow the RUN command (i.e., RUN 100), MCS BASIC-52 does not permit such a variation on the RUN command. Execution always begins with the first line number. To obtain the same functionality as the RUN[ln num] command, use the GOTO[ln num] statement in the direct mode. SEE STATEMENT GOTO.

EXAMPLE:

```
>10 FOR I=1 TO 3.
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3

READY
>
```

2.2 DESCRIPTION OF COMMANDS:

COMMAND: CONT(cr)

ACTION TAKEN:

If a program is stopped by typing a control-C on the console device or by execution of a STOP statement, you can resume execution of the program by typing CONT(cr). Between the stopping and the re-starting of the program you may display the values of variables or change the values of variables. However, you may NOT CONTinue if the program is modified during the STOP or after an error.

VARIATIONS:

None.

EXAMPLE:

```
>10 FOR I=1 TO 10000
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3
 4
 5    - (TYPE CONTROL-C ON CONSOLE)

STOP - IN LINE 20

READY
>PRINT I
 6

>I=10

>CONT

 10
 11
 12
```

2.3 DESCRIPTION OF COMMANDS:

COMMAND: LIST(cr)

ACTION TAKEN:

The LIST(cr) command prints the program to the console device. Note that the list command "formats" the program in an easy to read manner. Spaces are inserted after the line number and before and after statements. This feature is designed to aid in the debugging of MCS BASIC-52 programs. The "listing" of a program may be terminated at anytime by typing a control-C on the console device.

VARIATIONS:

Two variations of the LIST COMMAND are possible with MCS BASIC-52. They are:

LIST [ln num] (cr) and

LIST [ln num] — [ln num] (cr)

The first variation causes the program to be printed from the designated line number (integer) to the end of the program. The second variation causes the program to be printed from the first line number (integer) to the second line number (integer). NOTE — the two line numbers MUST BE SEPARATED BY A DASH —.

EXAMPLE:

```
READY
>LIST
 10 PRINT "LOOP PROGRAM"
 20 FOR I=1 TO 3
 30 PRINT I
 40 NEXT I
 50 END

READY
>LIST 30
 30 PRINT I
 40 NEXT I
 50 END

READY
>LIST 20-40
 20 FOR I=1 TO 3
 30 PRINT I
 40 NEXT I
```

2.4 DESCRIPTION OF COMMANDS

COMMAND: LIST#(cr)

ACTION TAKEN:

The LIST#(cr) command prints the program to the LIST device. The BAUD rate to this device must be initialized by the STATEMENT — BAUD[expr]. All comments that apply to the LIST command apply to the LIST# command. The LIST#(cr) command is included to permit the user to make "hard copies" of a program. The output to the list device is on P1.7 of the MCS BASIC-52 device.

2.5 DESCRIPTION OF COMMANDS

COMMAND: LIST@(cr) (VERSION 1.1 ONLY)

ACTION TAKEN:

The LIST@ command does the same thing as the LIST command except that the output is directed to a user defined output driver. This command assumes that the user has placed an assembly language output routine in external code memory location 403CH. To enable the @ driver routine the user must SET BIT 27H (39D) in the internal memory of the MCS BASIC-52 device. BIT 27H (39D) is BIT 7 of internal memory location 24H (36D). This BIT can be set by the BASIC statement DBY(24H)=DBY(24H). OR.80H or by a user supplied assembly language routine. If the user evokes the @ driver routine and this bit is not set, the output will be directed to the console driver. The only reason this BIT must be set to enable the @ driver is that it adds a certain degree of protection from accidently typing LIST@ when no assembly language routine exist. The philosophy here is that if the user sets the bit, the user provides the driver or else!!!

When MCS BASIC-52 calls the user output driver routine at location 403CH, the byte to output is in the accumulator and R5 of register bank 0 (RB0). The user may modify the accumulator (A) and the data pointer (DPTR) in the assembly language output routine, but cannot modify any of the registers in RB0. This is intended to make it real easy for the user to implement a parallel or serial output driver without having to do a PUSH or a POP.

2.6 DESCRIPTION OF COMMANDS

COMMAND: NEW(cr)

ACTION TAKEN:

When NEW(cr) is entered, MCS BASIC-52 deletes the program that is currently stored in RAM memory. In addition, all variables are set equal to ZERO, all strings and all BASIC evoked interrupts are cleared. The REAL TIME CLOCK, string allocation, and the internal stack pointer value (location 3EH) are NOT effected. In general, NEW (cr) is used simply to erase a program and all variables.

2.7 DESCRIPTION OF COMMANDS

COMMAND: NULL [integer](cr)

ACTION TAKEN:

The NULL[integer] (cr) command determines how many NULL characters (00H) MCS BASIC-52 will output after a carriage return. After initialization NULL = 0. The NULL command was more important back in the days when a "pure" mechanical printer was the most common I/O device. Most modern printers contain some kind of RAM buffer that virtually eliminates the need to output NULL characters after a carriage return. NOTE — the NULL count used by MCS BASIC-52 is stored in internal RAM location 21 (15H). The NULL value can be changed dynamically in a program by using a DBY(21) = [expr] statement. The [expr] can be any value between 0 and 255 (0FFH) inclusive.

VARIATIONS:

None.

CHAPTER 3
Description of EPROM File Commands

DESCRIPTION OF EPROM FILE COMMANDS

One of the unique and powerful features of MCS BASIC-52 is that it has the ability to execute and SAVE programs in an EPROM. MCS BASIC-52 actually generates all of the timing signals needed to program most EPROM devices. Saving programs in EPROMS is a much more attractive and RELIABLE alternative relative to cassette tape, especially in control and/or noisy environments.

The hardware needed to permit MCS BASIC-52 to program an EPROM device is minimal, typically only one NAND gate, three or four transistors, and a few resistors are all that is required. Details of the hardware requirements are in the EPROM PROGRAMMING section of this manual.

MCS BASIC-52 can save more than one program in an EPROM. In fact, it can save as many programs as the size of the EPROM memory permits. The programs are stored sequentially in the EPROM and any program can be retrieved and executed. This sequential storing of programs is referred to as the EPROM FILE. The following commands permit the user to generate and manipulate the EPROM FILE.

3.1 DESCRIPTION OF EPROM FILE COMMANDS

COMMANDS: RAM(cr) and ROM [integer] (cr)

ACTION TAKEN:

These two commands tell the MCS BASIC-52 interpreter whether to select the current program (the current program is the one that will be displayed during a LIST command and executed when RUN is typed) out of RAM or EPROM. The RAM address is assumed to be 512 (200H) and the EPROM address begins at 32, 784 (8010H).

RAM

When RAM(cr) is entered MCS BASIC-52 selects the current program from RAM MEMORY. This is usually considered the "normal" mode of operation and is the mode that most users interact with the command interpreter.

ROM

When ROM [integer] (cr) is entered MCS BASIC-52 selects the current program out of EPROM memory. If no integer is typed after the ROM command (i.e. ROM (cr)) MCS BASIC-52 defaults to ROM 1. Since the programs are stored sequentially in EPROM the integer following the ROM command selects which program the user wants to run or list. If you attempt to select a program that does not exist (i.e. you type in ROM 8 and only 6 programs are stored in the EPROM) the message ERROR: PROM MODE will be displayed.

MCS BASIC-52 does not transfer the program from EPROM to RAM when the ROM mode is selected. So, you cannot EDIT a program in the ROM mode. If you attempt to edit a program in the ROM mode, by typing in a line number, the message ERROR: PROM MODE will be displayed. The following command to be described, XFER, permits one to transfer a program from EPROM to RAM for editing purposes.

Since the ROM command does NOT transfer a program to RAM, it is possible to have different programs in ROM and RAM simultaneously. The user can "flip" back and forth between the two modes at any time. Another added benefit of NOT transferring a program to RAM is that all of the RAM memory can be used for variable storage if the PROGRAM is stored in EPROM. The SYSTEM CONTROL VALUES — MTOP and FREE always refer to RAM not EPROM.

VARIATIONS:

None.

3.2 DESCRIPTION OF EPROM FILE COMMANDS

COMMAND: XFER(cr)

ACTION TAKEN:

The XFER (transfer) command transfers the current selected program in EPROM to RAM and then selects the RAM mode. If XFER is typed while MCS BASIC-52 is in the RAM mode, the program stored in RAM is transferred back into RAM and the RAM mode is selected. The net result is that nothing happens except that a few milli-seconds of CPU time is used to do a wasted move. After the XFER command is executed, the user may edit the program in the same manner any RAM program may be edited.

VARIATIONS:

None.

3.3 DESCRIPTION OF EPROM FILE COMMANDS

COMMAND: PROG(cr)

ACTION TAKEN:

The PROG COMMAND programs the resident EPROM with the current selected program. The current selected program may reside in either RAM or EPROM. This command assumes that the hardware is configured in the manner described in the EPROM PROGRAMMING section of this manual.

After PROG (cr) is typed, MCS BASIC-52 displays the number in the EPROM FILE the program will occupy.

EXAMPLE:

```
>LIST
10        FOR I=1 TO 10
20        PRINT I
30        NEXT I

READY
>PROG
 12

READY
>ROM 12

READY
>LIST
10        FOR I=1 TO 10
20        PRINT I
30        NEXT I

READY
>
```

In this example, the program just placed in the EPROM is the 12th program stored.

VARIATIONS:

None.

3.4 DESCRIPTION OF EPROM FILE COMMANDS

COMMANDS: PROG1(cr) and PROG2(cr)

ACTION TAKEN:

PROG1

Normally, after power is applied to the MCS BASIC-52 device, the user MUST type a "space" character to initialize the 8052AH's serial port. As a convenience, MCS BASIC-52 contains a PROG1 COMMAND. What this command does is program the resident EPROM with the BAUD RATE information. So, the next time the MCS BASIC-52 device is "powered up," i.e. RESET, the chip will read this information and initialize the serial port with the stored baud rate. The "sign-on" message will be sent to the console immediately after the MCS BASIC-52 device completes its reset sequence. The "space" character no longer needs to be typed. Of course, if the BAUD rate on the console device is changed a new EPROM must be programmed to make MCS BASIC-52 compatible with the new console.

PROG2

The PROG2 command does everything the PROG1 command does, but instead of "signing-on" and entering the COMMAND MODE, the MCS BASIC-52 device immediately begins executing the first program stored in the resident EPROM.

THIS IS AN IMPORTANT FEATURE!!

By using the PROG2 command it is possible to RUN a program from a RESET condition and NEVER connect the MCS BASIC-52 chip to a console. In essence, saving PROG2 information is equivalent to typing a ROM 1, RUN command sequence. This is ideal for control applications, where it is not always possible to have a terminal present. In addition, this feature permits the user to write a special initialization sequence in BASIC or ASSEMBLY LANGUAGE and generate a custom "sign-on" message for specific applications.

3.5 DESCRIPTION OF EPROM FILE COMMANDS

COMMANDS: FPROG(cr), FPROG1(cr), AND FPROG2(cr)

ACTION TAKEN:

FPROG(cr), FPROG1(cr), and FPROG2(cr) do exactly the same thing as PROG(cr), PROG1(cr), and PROG2(cr) respectively, except that the algorithm used to perform the programming function is the INTEL "INTELLIGENT" fast programming algorithm. The user MUST provide a way to increase VCC to the EPROM to 6 volts.

3.6 DESCRIPTION OF EPROM FILE COMMANDS

COMMANDS: PROG3(cr), PROG4(cr), FPROG3(cr), FPROG4(cr) (VERSION 1.1 ONLY)

ACTION TAKEN:

PROG3

The PROG3 COMMAND functions the same way as the PROG1 COMMAND previously described, except that PROG3 also saves the system control value, MTOP, when it is evoked. During a RESET or power-up sequence MCS BASIC-52 will only clear the external data memory up to the MTOP value that was saved when the PROG3 COMMAND was evoked. This permits the user to "protect" regions of memory from being cleared during a RESET or power-up condition. In typical use, the PROG3 COMMAND assumes that the user is saving some critical information if some type of battery-backed-up or non-volitle memory and does not want this information to be destroyed during a RESET or power-up sequence.

PROG4

The PROG4 COMMAND is a combination of the PROG2 and PROG3 COMMAND. PROG4 saves the same information as PROG3, but also executes the first program stored in the EPROM after a RESET or power-up condition.

FPROG3 and FPROG4

The FPROG3 and FPROG4 commands save the same information as the PROG3 and PROG4 commands respectively, except that the INTELligent™ algorithm is used to program the EPROM.

VARIATIONS:

None.

3.7 DESCRIPTION OF EPROM FILE COMMANDS

COMMANDS: PROG5(cr), PROG6(cr), FPROG5(cr), FPROG6(cr) (VERSION 1.1 ONLY)

ACTION TAKEN:

PROG5 & FPROG5

The PROG5 command saves both the baud rate information and the MTOP information, just like the PROG3 command previously described. However, during a RESET or power-up condition the MCS BASIC-52 device examines external data memory location 5FH (95 decimal). If the user has placed the value 0A5H (165 decimal) in this location, the MCS BASIC-52 device will **not clear the external memory** during a RESET or power-up condition. This permits the user to "save" programs in external memory, providing some type of battery back-up scheme has been employed.

Normally, when using the PROG5 command to establish the RESET or power-up condition, the MCS BASIC-52 device will enter the command mode after RESET or power-up. However, if the user wishes to execute the program stored in external memory, the character 34H (52 decimal) needs to be placed in external memory location 5EH (94 decimal). Placing a 34H in location 5EH causes MCS BASIC-52 to enter the "RUN TRAP MODE." Details of this mode are presented in chapter 8 of this manual.

PROG6 & FPROG6

Does the same thing as PROG5, but CALLS external program memory location 4039H during a RESET or power-up sequence. This option also requires the user to put the character 0A5H in external memory location 5FH to insure that external RAM will not be cleared during RESET or power-up. The user must put an assembly language initialization routine in external code memory location 4039H or else this RESET mode will crash. When the user returns from the customized assembly language RESET routine, three options exist:

OPTION 1 FOR PROG6

If the CARRY BIT is CLEARED (CARRY = 0) upon return from the user RESET routine MCS BASIC-52 will enter the auto-baud rate determining routine. The user must then type a space character (20H) on the terminal to complete the RESET routine and produce a RESET message on the terminal.

OPTION 2 FOR PROG6

If the CARRY BIT is SET (CARRY = 1) and BIT 0 of the ACCUMULATOR is CLEARED (ACC. 0 = 0) MCS BASIC-52 will produce the standard sign-on message upon return from the user supplied RESET routine. The baud rate will be the one that was saved when the PROG6 option was used.

OPTION 3 FOR PROG6

If the CARRY BIT is SET (CARRY = 1) and BIT 0 of the ACCUMULATOR is SET (ACC. 0 = 1), MCS BASIC-52 will execute the first program stored by the user in EPROM (starting address of the program is 8010H) upon return from the user supplied RESET routine.

CHAPTER 4
Description of Statements

4.1 DESCRIPTION OF STATEMENTS

STATEMENT: BAUD [expr]

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

The BAUD [expr] statement is used to set the baud rate for the software line printer port resident on the MCS BASIC-52 device. In order for this STATEMENT to properly calculate the baud rate, the crystal (special function operator — XTAL) must be correctly assigned (e.g. XTAL = 9000000). MCS BASIC-52 assumes a crystal value of 11.0592 MHz if no XTAL value is assigned. The software line printer port is P1.7 on the 8052AH device. The main purpose of the software line printer port is to let the user make a "hard copy" of program listings and/or data. The COMMAND LIST# and the STATEMENT PRINT# direct outputs to the software line printer port. If the BAUD [expr] STATEMENT is not executed before a LIST# or PRINT# command/statement is entered, the output to the software line printer port will be at about 1 BAUD and it will take A LONG TIME to output something. You may even think that BASIC has crashed, but it hasn't. It's just outputting at a VERY SLOW rate. So be sure to assign a BAUD rate to the software printer port BEFORE using LIST# or PRINT#. The maximum baud rate that can be assigned by the BAUD statement depends on the crystal. In general, 4800 is a reasonable maximum baud rate, however the user may want to experiment with different rates. The software serial transmits 8 data bits, 1 start bit, and two stop bits. No parity is transmitted.

EXAMPLE:

```
BAUD 1200

Will cause the line printer port to output data at 1200 BAUD.
```

VARIATIONS:

None.

4.2 DESCRIPTION OF STATEMENTS

STATEMENT: CALL [integer]

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

The CALL [integer] STATEMENT is used to call an assembly language program. The integer following CALL is the address where the user must provide the assembly language routine. To return to BASIC the user must execute an assembly language RET instruction. Examples of how to use the CALL [integer] instruction are given in the ASSEMBLY LANGUAGE LINKAGE section of this manual.

EXAMPLE:

```
CALL 9000H
```

Will cause the 8052AH to execute the assembly language program beginning at location 9000H (i.e. the program counter will be loaded with 9000H).

VARIATIONS: (VERSION 1.1 ONLY)

If the integer following the CALL statement is between 0 and 127 (7FH), Version 1.1 of MCS BASIC-52 will multiply the user integer by two, then add 4100H and vector to that location. This means that CALL 0 will call location 4100H, CALL 1 will call 4102H, CALL 2 — 4104H and so on. This permits the user to generate a simple table of assembly language routines without having to enter 4 digit hex integers after the CALL statement from the user supplied RESET routine.

4.3 DESCRIPTION OF STATEMENTS

STATEMENT: CLEAR

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

The CLEAR STATEMENT sets all variables equal to 0 and resets all BASIC evoked interrupts and stacks. This means that after the CLEAR statement is executed an ONEX1 or ONTIME statement must be executed before MCS BASIC-52 will acknowledge interrupts. ERROR trapping via the ONERR statement will also not occur until an ONERR[integer] STATEMENT is executed. The CLEAR STATEMENT does not affect the real time clock that is enabled by the CLOCK1 STATEMENT. CLEAR also does not reset the memory that has been allocated for STRINGS, so it is NOT necessary to enter the STRING [expr], [expr] STATEMENT to re-allocate memory for strings after the CLEAR STATEMENT is executed. In general, CLEAR is simply used to "erase" all variables.

VARIATIONS:

None.

4.4 DESCRIPTION OF STATEMENTS

STATEMENTS: CLEARI (clear interrupts)

CLEARS (clear stacks)

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

CLEARI

The CLEARI STATEMENT clears all of the BASIC evoked interrupts. Specifically, the ONTIME and ONEX1 interrupts are DISABLED after the CLEARI STATEMENT is executed. This is accomplished by clearing bits 2 and 3 of the 8052AH's special function register, IE and by clearing the status bits that determine whether MCS BASIC-52 or the user is controlling these interrupts. The real time clock which is enabled by the CLOCK1 STATEMENT is not affected by CLEARI. This statement can be used to selectively DISABLE interrupts during specific sections of the users BASIC program. The ONTIME and/or ONEX1 STATEMENTS MUST BE EXECUTED AGAIN before the specific interrupts will be enabled.

CLEARS

The CLEARS statement RESETS all of MCS BASIC-52's STACKS. The CONTROL and ARGUMENT STACKS are reset to their initialization value, 254 (OFEH) and 510 (1FEH) respectively. The INTERNAL STACK (the 8052AH's STACK POINTER, SPECIAL FUNCTION REGISTER-SP) is loaded with the value that is in INTERNAL RAM location 62 (3EH). This statement can be used to "purge" the stack should an error occur in a subroutine. In addition, this statement can be used to provide a "special" exit from a FOR-NEXT, DO-WHILE, or DO-UNTIL loop.

EXAMPLE OF CLEARS:

```
>10 PRINT "MULTIPLICATION TEST, YOU HAVE 5 SECONDS"
>20 FOR I = 2 TO 9
>30 N = INT(RND*10) : A = N*I
>40 PRINT "WHAT IS ",N,"*",I,"?": CLOCK1
>50 TIME = 0 : ONTIME 5,200 : INPUT R: IF R<>A THEN 100
>60 PRINT "THAT'S RIGHT": TIME=0: NEXT I
>70 PRINT "YOU DID IT, GOOD JOB": END
>100 PRINT "WRONG, TRY AGAIN": GOTO 50
>200 REM WASTE CONTROL STACK, TOO MUCH TIME
>210 CLEARS: PRINT "YOU TOOK TOO LONG": GOTO 10
```

NOTE: When the CLEARS and CLEARI STATEMENTS are LISTED they will appear as CLEAR S and CLEAR I respectively. Don't be alarmed, that is the way it's supposed to work.

4.5 DESCRIPTION OF STATEMENTS

STATEMENTS: CLOCK1 and CLOCK0

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

CLOCK1

The CLOCK1 STATEMENT enables the REAL TIME CLOCK feature resident on the MCS BASIC-52 device. The special function operator TIME is incremented once every 5 milliseconds after the CLOCK1 STATEMENT has been executed. The CLOCK1 STATEMENT uses TIMER/COUNTER 0 in the 13-bit mode to generate an interrupt once every 5 milliseconds. Because of this, the special function operator TIME has a resolution of 5 milliseconds.

MCS BASIC-52 automatically calculates the proper reload value for TIMER/COUNTER 0 after the crystal value has been assigned (i.e. XTAL = value). If no crystal value is assigned, MCS BASIC-52 assumes a value of 11.0592 MHz. The special function operator TIME counts from 0 to 65535.995 seconds. After reaching a count of 65535.995 seconds TIME overflows back to a count of zero. Because the CLOCK1 STATEMENT uses the interrupts associated with TIMER/COUNTER 0 (the CLOCK1 statement sets bits 7 and 2 in the 8052AH's special function register, IE), the user may not use this interrupt in an assembly language routine if the CLOCK1 STATEMENT is executed in BASIC. The interrupts associated with the CLOCK1 STATEMENT cause MCS BASIC-52 programs to run at about 99.6% of normal speed. That means that the interrupt handling for the REAL TIME CLOCK feature only consumes about .4% of the total CPU time. This very small interrupt overhead is attributed to the very fast and effective interrupt handling of the 8052AH device.

CLOCK0

The CLOCK0 (zero) STATEMENT disables or "turns off" the REAL TIME CLOCK feature. This statement clears bit 2 in the 8052AH's special function register, IE. After CLOCK0 is executed, the special function operator TIME will no longer increment. The CLOCK0 STATEMENT also returns control of the interrupts associated with TIMER/COUNTER 0 back to the user, so this interrupt may be handled at the assembly language level. CLOCK0 is the only MCS BASIC-52 statement that can disable the REAL TIME CLOCK. CLEAR and CLEARI will NOT disable the REAL TIME CLOCK.

VARIATIONS:

None.

4.6 DESCRIPTION OF STATEMENTS

STATEMENTS: DATA — READ — RESTORE

MODE: RUN

TYPE: ASSIGNMENT

DATA

DATA specifies expressions that may be retrieved by a READ STATEMENT. If multiple expressions per line are used, they MUST be separated by a comma.

READ

READ retrieves the expressions that are specified in the DATA STATEMENT and assigns the value of the expression to the variable in the READ STATEMENT. The READ STATEMENT MUST ALWAYS be followed by one or more variables. If more than one variable follows a READ STATEMENT, they MUST be separated by a comma.

RESTORE

RESTORE "resets" the internal read pointer back to the beginning of the data so that it may be read again.

EXAMPLE:

```
>10 FOR I=1 TO 3
>20 READ A,B
>30 PRINT A,B
>40 NEXT I
>50 RESTORE
>60 READ A,B
>70 PRINT A,B
>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)
>RUN

  10   20
  5    10
  0   -1
  10   20
```

VARIATIONS:

None.

4.6 DESCRIPTION OF STATEMENTS

Explanation of previous example:

Everytime a READ STATEMENT is encountered the next consecutive expression in the DATA STATEMENT is evaluated and assigned to the variable in the READ STATEMENT. DATA STATEMENTS may be placed anywhere within a program, they will NOT be executed nor will they cause an error. DATA STATEMENTS are considered to be chained together and appear to be one BIG DATA STATEMENT. If at anytime all the DATA has been read and another READ STATEMENT is executed then the program is terminated and the message ERROR: NO DATA — IN LINE XX is printed to the console device.

4.7 DESCRIPTION OF STATEMENTS

STATEMENT: DIM

MODE: COMMAND AND/OR RUN

TYPE: ASSIGNMENT

DIM reserves storage for matrices. The storage area is first assumed to be zero. Matrices in MCS BASIC-52 may have only ONE DIMENSION and the size of the dimensioned array MAY NOT exceed 254 elements. Once a variable is dimensioned in a program it may not be re-dimensioned. An attempt to re-dimension an array will cause an ARRAY SIZE ERROR. If an arrayed variable is used that has NOT been dimensioned by the DIM STATEMENT, BASIC will assign a default value of 10 to the array size. All arrays are set equal to zero when the RUN COMMAND, NEW COMMAND, or the CLEAR STATEMENT is executed. The number of bytes allocated for an array is 6 times the (array size plus 1). So, the array A(100) would require 606 bytes of storage. Memory size usually limits the size of a dimensioned array.

VARIATIONS:

More than one variable can be dimensioned by a single DIM STATEMENT, i.e., DIM A(10), B(15), A1(20).

EXAMPLE:

```
DEFAULT ERROR ON ATTEMPT TO RE-DIMENSION ARRAY

>10 A(5)=10      - BASIC ASSIGNS DEFAULT OF 10 TO ARRAY SIZE HERE
>20 DIM A(5)     - ARRAY CANNOT BE RE-DIMENSIONED
>RUN

ERROR: ARRAY SIZE - IN LINE 20

20    DIM A(5)
----------X
```

4.8 DESCRIPTION OF STATEMENTS

STATEMENTS: DO — UNTIL [rel expr]

MODE: RUN

TYPE: CONTROL

The DO — UNTIL [rel expr] instruction provides a means of "loop control" within an MCS BASIC-52 program. All statements between the DO and the UNTIL [rel expr] will be executed until the relational expression following the UNTIL statement is TRUE. DO — UNTIL loops may be nested.

EXAMPLES:

```
SIMPLE DO-UNTIL

>10 A=0
>20 DO
>30 A=A+1
>40 PRINT A
>50 UNTIL A=4
>60 PRINT "DONE"
>RUN

 1
 2
 3
 4
DONE

READY
>
```

```
NESTED DO-UNTIL

>10 DO : A=A+1 : DO : B=B+1
>20 PRINT A, B, A*B
>30 UNTIL B=3
>40 B=0
>50 UNTIL A=3
>RUN

 1   1   1
 1   2   2
 1   3   3
 2   1   2
 2   2   4
 2   3   6
 3   1   3
 3   2   6
 3   3   9

READY
>
```

VARIATIONS:

None

4.9 DESCRIPTION OF STATEMENTS

STATEMENTS: DO — WHILE [rel expr]

MODE: RUN

TYPE: CONTROL

The DO — WHILE [rel expr] instruction provides a means of "loop control" within an MCS BASIC-52 program. This operation of this statement is similar to the DO -- UNTIL [rel expr] except that all statements between the DO and the WHILE [rel expr] will be executed as long as the relational expression following the WHILE statement is true. DO — WHILE and DO — UNTIL statements can be nested.

EXAMPLES:

```
SIMPLE DO-WHILE

>10 DO
>20 A=A+1
>30 PRINT A
>40 WHILE A<4
>50 PRINT "DONE"
>RUN

 1
 2
 3
 4
DONE

READY
>
```

```
NESTED DO-WHILE - DO-UNTIL

>10 DO : A=A+1 : B=B+1
>20 PRINT A, B, A*B
>30 WHILE B<>3
>40 B=0
>50 UNTIL A=3
>RUN

 1  1  1
 1  2  2
 1  3  3
 2  1  2
 2  2  4
 2  3  6
 3  1  3
 3  2  6
 3  3  9

READY
>
```

VARIATIONS:

None

4.10 DESCRIPTION OF STATEMENTS

STATEMENT: END

MODE: RUN

TYPE: CONTROL

The END STATEMENT terminates program execution. The continue command, CONT will not operate if the END STATEMENT is used to terminate execution (i.e., a CAN'T CONTINUE ERROR will be printed to the console). The last statement in an MCS BASIC-52 program will automatically terminate program execution if no END STATEMENT is used.

EXAMPLES:

```
LAST STATEMENT TERMINATION

>10 FOR I=1 TO 4
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3
 4

READY
>
```

```
END STATEMENT TERMINATION

>10 FOR I=1 TO 4
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I
>110 RETURN
>RUN

 1
 2
 3
 4

READY
>
```

VARIATIONS:

None

4.11 DESCRIPTION OF STATEMENTS

STATEMENTS: FOR — TO — {STEP} — NEXT

MODE: RUN VERSION 1.0 (COMMAND AND/OR RUN in Version 1.1)

TYPE: CONTROL

The FOR — TO — {STEP} — NEXT STATEMENTS are used to set up and control loops.

EXAMPLE:

```
10 FOR A=B TO C STEP D
20 PRINT A
30 NEXT A
```

If B = 0, C = 10, and D = 2, the PRINT STATEMENT at line 20 will be executed 6 times. The values of "A" that will be printed are 0, 2, 4, 6, 8, 10. "A" represents the name of the index or loop counter. The value of "B" is the starting value of the index, the value of "C" is the limit value of the index, and the value of "D" is the increment to the index. If the STEP STATEMENT and the value "D" are omitted, the increment value defaults to 1, therefore, STEP is an optional statement. The NEXT STATEMENT causes the value of "D" to be added to the index. The index is then compared to the value of "C," the limit. If the index is less than or equal to the limit, control will be transferred back to the statement after the FOR STATEMENT. Stepping "backwards" (i.e. FOR I = 100 TO 1 STEP-1) is permitted in MCS BASIC-52. Unlike some BASICS, the index MAY NOT be omitted from the NEXT STATEMENT in MCS BASIC-52 (i.e. the NEXT statement MUST always be followed by the appropriate variable).

EXAMPLES:

```
>10 FOR I=1 TO 4
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3
 4

READY
>
```

```
>10 FOR I=0 TO 8 STEP 2
>20 PRINT I
>30 NEXT I
>RUN

 0
 2
 4
 6
 8

READY
>
```

4.11 DESCRIPTION OF STATEMENTS

In Version 1.1 of MCS BASIC-52 it is possible execute the FOR-TO-{STEP}-NEXT statement in the Command Mode. This makes it possible for the user to do things like display regions of memory by writing a short program like FOR I=512 TO 560: PH0. XBY(I),: NEXT I. It may also have other uses, but they haven't been thought of.

Also Version 1.1 allows the NEXT statement to be used without a variable following the statement. This means that programs like:

EXAMPLE:

```
10 FOR I = 1 TO 100
20 PRINT I
30 NEXT
```

Are permitted in Version 1.1 of MCS BASIC-52. The variable associated with the NEXT is always assumed to be the variable used in the last FOR statement.

4.12 DESCRIPTION OF STATEMENTS

STATEMENTS: GOSUB[ln num] — RETURN

MODE: RUN

TYPE: CONTROL

GOSUB

The GOSUB [ln num] STATEMENT will cause MCS BASIC-52 to transfer control of the program directly to the line number ([ln num]) following the GOSUB STATEMENT. In addition, the GOSUB STATEMENT saves the location of the STATEMENT following GOSUB on the control stack so that a RETURN STATEMENT can be performed to return control.

RETURN

This statement is used to "return" control back to the STATEMENT following the most recently executed GOSUB STATEMENT. The GOSUB-RETURN sequence can be "nested" meaning that a subroutine called by the GOSUB STATEMENT can call another subroutine with another GOSUB STATEMENT.

EXAMPLES:

```
SIMPLE SUBROUTINE

>10 FOR I=1 TO 5
>20 GOSUB 100
>30 NEXT I
>100 PRINT I
>110 RETURN
>RUN

 1
 2
 3
 4
 5

READY
>
```

```
NESTED SUBROUTINES

 >10 FOR I=1 TO 3
 >20 GOSUB 100
 >30 NEXT I
 >40 END
 >100 PRINT I,
 >110 GOSUB 200
 >120 RETURN
 >200 PRINT I*I
 >210 RETURN
 >RUN

  1  1
  2  4
  3  9

  READY
  >
```

4.12 DESCRIPTION OF STATEMENTS

NOTE — The Control Stack on Version 1.1 permits a graceful exit from incompleted control loops, given the following example:

EXAMPLE:

```
.
50      GOSUB 1000
.
.
1000    FOR I = 1 TO 10
1010    IF X = I THEN 1040
1020    PRINT I*X
1030    NEXT I
1040    RETURN
```

Version 1.1 would permit the programmer to exit the subroutine even though the FOR-NEXT loop might not be allowed to complete if X did equal I. Version 1.0 of MCS BASIC-52 would yield a C-STACK error if the FOR-NEXT loop was not allowed to complete before the RETURN statement was executed.

4.13 DESCRIPTION OF STATEMENTS

STATEMENT: GOTO [ln num]

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

The GOTO [ln num] STATEMENT will cause BASIC to transfer control directly to the line number ([ln num]) following the GOTO STATEMENT.

EXAMPLE:

```
50 GOTO 100
```

Will, if line 100 exists, cause execution of the program to resume at line 100. If line number 100 does not exist the message ERROR: INVALID LINE NUMBER will be printed to the console device.

Unlike the RUN COMMAND the GOTO STATEMENT, if executed in the COMMAND MODE, does not CLEAR the variable storage space or interrupts. However, if the GOTO STATEMENT is executed in the COMMAND MODE after a line has been edited, MCS BASIC-52 will CLEAR the variable storage space and all BASIC evoked interrupts. This is a necessity because the variable storage and the BASIC program reside in the same RAM memory. So editing a program can destroy variables.

NOTE — (Version 1.0 only)

Because of the way MCS BASIC-52's text interpreter processes a line, when an INVALID LINE NUMBER ERROR occurs on the GOTO, GOSUB, ON GOTO, and ON GOSUB STATEMENTS the line AFTER the GOTO or GOSUB STATEMENT will be printed out in the error message. This may be confusing, but it was a trade-off between execution speed, code size, and error handling. Error handling lost.

EXAMPLE:

```
>10 GOTO 100
>20 PRINT X
>RUN

ERROR: INVALID LINE NUMBER - IN LINE 20

20    PRINT X
------------X
```

Version 1.1 does not exhibit this particular anomaly.

4.14 DESCRIPTION OF STATEMENTS

STATEMENTS: ON [expr] GOTO[ln num], [ln num], . . . [ln num]

ON [expr] GOSUB[ln num], [ln num], . . . [ln num]

MODE: RUN

TYPE: CONTROL

The value of the expression following the ON statement is the number in the line list that control will be transferred to.

EXAMPLE:

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

If Q was equal to 0, control would be transferred to line number 100. If Q was equal to 1, control would be transferred to line number 200. If Q was equal to 2, GOTO line 300, etc. All comments that apply to GOTO and GOSUB apply to the ON STATEMENT. If Q is less than ZERO a BAD ARGUMENT ERROR will be generated. If Q is greater than the line number list following the GOTO or GOSUB STATEMENT, a BAD SYNTAX ERROR will be generated. The ON STATEMENT provides "conditional branching" options within the constructs of an MCS BASIC-52 program.

4.15 DESCRIPTION OF STATEMENTS

STATEMENTS: IF — THEN — ELSE

MODE: RUN

TYPE: CONTROL

The IF statement sets up a conditional test. The generalized form of the IF — THEN — ELSE statement is as follows:

[ln num] IF [rel expr] THEN valid STATEMENT ELSE valid STATEMENT

A specific example is as follows:

```
>10 IF A=100 THEN A=0 ELSE A=A+1
```

Upon execution of line 10 IF A is equal to 100, THEN A would be assigned a value of 0. IF A does not equal 100, A would be assigned a value of A + 1. If it is desired to transfer control to different line numbers using the IF statement, the GOTO statement may be omitted. The following examples would yield the same results:

```
>20 IF INT(A)< 10 THEN GOTO 100 ELSE GOTO 200
>20 IF INT(A)< 10 THEN 100 ELSE 200
```

Additionally, the THEN statement can be replaced by any valid MCS BASIC-52 statement, as shown below:

```
>30 IF A<>10 THEN PRINT A ELSE 10
>30 IF A<>10 PRINT A ELSE 10
```

The ELSE statement may be omitted. If it is, control will pass to the next statement.

EXAMPLE:

```
>20 IF A=10 THEN 40
>30 PRINT A
```

In this example, IF A equals 10 then control would be passed to line number 40. If A does not equal 10 line number 30 would be executed.

4.15 DESCRIPTION OF STATEMENTS

COMMENTS ON IF-THEN-ELSE-

Version 1.1 is not compatible with V1.0 when the IF_THEN_ELSE STATEMENT is used with multiple statements per line. In V1.0, the following two examples would function in the same manner.

EXAMPLE 1:

```
10 IF A=B THEN C=A : A=A/2 : GOTO 100
20 PRINT A
```

EXAMPLE 2:

```
10 IF A=B THEN C=A
12 A=A/2
14 GOTO 100
20 PRINT A
```

They function in the same manner because V1.0 treats the delimiter (:) exactly the same as a carriage return (cr) in every case. However, V1.1 executes the remainder of the line if and only if the test A = B proves to be true. This means in EXAMPLE 1 IF A did equal B, V1.1 would then set C = A, then set A = A/2, then execute line 100. IF A did not equal B, V1.1 would then PRINT A and ignore the statements C = A : A = A/2 : GOTO 100. V1.1 will execute EXAMPLE 2 exactly the same way as V1.0. This same logical interpretation holds true for the ELSE statement as well. This example dictates a simple rule for maintaining IF_THEN_ELSE compatibility between the two versions. IF THE DELIMITER (:) IS NOT USED IN AN IF_THEN_ELSE STATEMENT, V1.0 AND V1.1 WILL TREAT THE STATEMENTS IN THE SAME MANNER!!

This change was made because most users of MCS BASIC-52 felt that the V1.1 interpretation of this statement was more useful because fewer GOTO statements need be employed in a typical program.

4.16 DESCRIPTION OF STATEMENTS

STATEMENTS: INPUT

MODE: RUN

TYPE: INPUT/OUTPUT

The INPUT statement allows users to enter data from the console during program execution. One or more variables may be assigned data with a single input statement. The variables must be separated by a comma.

EXAMPLE:

```
INPUT A, B
```

Would cause the printing of a question mark (?) on the console device as a prompt to the operator to input two numbers separated by a comma. If the operator does not enter enough data, then MCS BASIC-52 responds by outputting the message TRY AGAIN to the console device.

EXAMPLE:

```
>10 INPUT A, B
>20 PRINT A, B
>RUN

?1

TRY AGAIN

?1, 2
 1   2

READY
```

The INPUT statement may be written so that a descriptive prompt is printed to tell the user what to type. The message to be printed is placed in quotes after the INPUT statement. If a comma appears before the first variable on the input list, the question mark prompt character will not be displayed.

EXAMPLES:

```
>10 INPUT"ENTER A NUMBER"A
>20 PRINT SQR(A)
>RUN

ENTER A NUMBER
?100
 10
```

```
>10 INPUT"ENTER A NUMBER-", A
>20 PRINT SQR(A)
>RUN

ENTER A NUMBER-100
 10
```

4.16 DESCRIPTION OF STATEMENTS

Strings can also be assigned with an INPUT statement. Strings are always terminated with a carriage return (cr). So, if more than one string input is requested with a single INPUT statement, MCS BASIC-52 will prompt the user with a question mark.

EXAMPLES:

```
>10 STRING 110,10
>20 INPUT "NAME: ",$(1)
>30 PRINT "HI ",$(1)
>RUN

NAME: SUSAN
HI SUSAN

READY
```

```
>10 STRING 110,10
>20 INPUT "NAMES: ",$(1),$(2)
>30 PRINT "HI ",$(1)," AND ",$(2)
>RUN

NAMES: BILL
?ANN
HI BILL AND ANN

READY
```

Additionally, strings and variables can be assigned with a single INPUT statement.

EXAMPLE:

```
>10 STRING 100,10
>20 INPUT"NAME(CR), AGE - ",$(1),A
>30 PRINT "HELLO ",$(1),", YOU ARE ",A,"YEARS OLD"
>RUN

NAME(CR), AGE - FRED
?15
HELLO FRED, YOU ARE 15 YEARS OLD

READY
>
```

4.17 DESCRIPTION OF STATEMENTS

STATEMENT: LET

MODE: COMMAND AND/OR RUN

TYPE: ASSIGNMENT

The LET statement is used to assign a variable to the value of an expression. The generalized form of LET is:

LET [var] = [expr]

EXAMPLES:

```
LET A = 10*SIN(B)/100 or
LET A = A + 1
```

Note that the = sign used in the LET statement is not equality operator, but rather a "replacement" operator and that the statement should be read A is replaced by A plus one. THE WORD LET IS ALWAYS OPTIONAL, i.e.

```
LET A = 2 is the same as A = 2
```

When LET is omitted the LET statement is called an IMPLIED LET. This document will use the word LET to refer to both the LET statement and the IMPLIED LET statement.

The LET statement is also used to assign the string variables, i.e:

```
LET $(1)="THIS IS A STRING" or
LET $(2)=$(1)
```

Before Strings can be assigned the STRING [expr], [expr] STATEMENT MUST be executed, or else a MEMORY ALLOCATION ERROR will occur.

SPECIAL FUNCTION VALUES can also be assigned by the LET statement, i.e.:

```
LET IE = 82H or
LET XBYTE(2000H)=5AH or
LET DBYTE(25)=XBYTE(1000)
```

4.18 DESCRIPTION OF STATEMENTS

STATEMENT: ONERR[ln num]

MODE: RUN

TYPE: CONTROL

The ONERR[ln num] statement lets the programmer handle arithmetic errors, should they occur, during program execution. Only ARITH. OVERFLOW, ARITH. UNDERFLOW, DIVIDE BY ZERO, and BAD ARGUMENT errors can be "trapped" by the ONERR statement, all other errors are not. If an arithmetic error occurs after the ONERR statement is executed, the MCS BASIC-52 interpreter will pass control to the line number following the ONERR[ln num] statement. The programmer can handle the error condition in any manner suitable to the particular application. Typically, the ONERR[ln num] statement should be viewed as an easy way to handle errors that occur when the user provides inappropriate data to an INPUT statement.

With the ONERR[ln num] statement, the programmer has the option of determining what type of error occurred. This is done by examining external memory location 257 (101H) after the error condition is trapped. The error codes are as follows:

```
ERROR CODE = 10 - DIVIDE BY ZERO
ERROR CODE = 20 - ARITH. OVERFLOW
ERROR CODE = 30 - ARITH. UNDERFLOW
ERROR CODE = 40 - BAD ARGUMENT
```

This location may be examined by using an XBY(257) statement.

4.19 DESCRIPTION OF STATEMENTS

STATEMENT: ONEX1 [ln num]

MODE: RUN

TYPE: CONTROL

The ONEX1 [ln num] statement lets the user handle interrupts on the 8052AH's INT1 pin with a BASIC program. The line number following the ONEX1 statement tells the MCS BASIC-52 interpreter which line to pass control to when an interrupt occurs. In essence, the ONEX1 statement "forces" a GOSUB to the line number following the ONEX1 statement when the INT1 pin on the 8052AH is pulled low. The programmer must execute a RETI statement to exit from the ONEX1 interrupt routine. If this is not done all future interrupts on the INT1 pin will be "locked out" and ignored until a RETI is executed.

The ONEX1 statement sets bits 7 and 2 of the 8052AH's interrupt enable register IE. Before an interrupt can be processed, the MCS BASIC-52 interpreter must complete execution of the staement it is currently processing. Because of this, interrupt latency can vary from microseconds to tens of milliseconds. The ONTIME [expr], [ln num] interrupt has priority over the ONEX1 interrupt. So, the ONTIME interrupt can interrupt the ONEX1 interrupt routine.

4.20 DESCRIPTION OF STATEMENTS

STATEMENT: ONTIME [expr], [ln num]

MODE: RUN

TYPE: CONTROL

Since MCS BASIC-52 processes a line in the millisecond time frame and the timer/counters on the 8052AH operate in the micro-second time frame, there is an inherent incompatibility between the timer/counters on the 8052AH and MCS BASIC-52. To help solve this situation the ONTIME [expr], [ln num] statement was devised. What ONTIME does is generate an interrupt everytime the SPECIAL FUNCTION OPERATOR, TIME, is equal to or greater than the expression following the ONTIME statement. Actually, only the integer portion of TIME is compared to the integer portion of the expression. The interrupt forces a GOSUB to the line number ([ln num]) following the expression ([expr]) in the ONTIME statement.

Since the ONTIME statement uses the SPECIAL FUNCTION OPERATOR, TIME, the CLOCK1 statement must be executed in order for ONTIME to operate. If CLOCK1 is not executed the SPECIAL FUNCTION OPERATOR, TIME, will never increment and not much will happen.

Since the ONTIME statement generates an interrupt when TIME is greater than or equal to the expression following the ONTIME statement, how can periodic interrupts be generated? That's easy, the ONTIME statement must be executed again in the interrupt routine:

EXAMPLE:

```
>10 TIME=0 : CLOCK1 : ONTIME 2,100 : DO
>20 WHILE TIME<10 : END
>100 PRINT "TIMER INTERRUPT AT -",TIME,"SECONDS"
>110 ONTIME TIME+2,100 : RETI
>RUN

TIMER INTERRUPT AT - 2.045 SECONDS
TIMER INTERRUPT AT - 4.045 SECONDS
TIMER INTERRUPT AT - 6.045 SECONDS
TIMER INTERRUPT AT - 8.045 SECONDS
TIMER INTERRUPT AT - 10.045 SECONDS

READY
```

You may wonder why the TIME that was printed out was 45 milliseconds greater than the time that the interrupt was supposed to be generated. That's because the terminal used in this example was running at 4800 BAUD and it takes about 45 milliseconds to print the message TIMER INTERRUPT AT -" ".

4.20 DESCRIPTION OF STATEMENTS

If the programmer does not want this delay, a variable should be assigned to the SPECIAL FUNCTION OPERATOR, TIME, at the beginning of the interrupt routine.

EXAMPLE:

```
>10 TIME=0 : CLOCK1 : ONTIME 2,100: DO
>20 WHILE TIME<10 : END
>100 A=TIME
>110 PRINT "TIMER INTERRUPT AT -",A,"SECONDS"
>120 ONTIME A+2,100 : RETI
>RUN

TIMER INTERRUPT AT - 2 SECONDS
TIMER INTERRUPT AT - 4 SECONDS
TIMER INTERRUPT AT - 6 SECONDS
TIMER INTERRUPT AT - 8 SECONDS
TIMER INTERRUPT AT - 10 SECONDS

READY
```

Like the ONEX1 statement, the ONTIME interrupt routine *must be exited with a RETI statement*. Failure to do this will "lock-out" all future interrrupts.

The ONTIME interrupt has priority over the ONEX1 interrupt. This means that the ONTIME interrupt can interrupt the ONEX1 interrupt routine. This priority was established because time related functions in control applications were viewed as critical routines. If the user does not want the ONEX1 routine to be interrupted by the ONTIME interrupt, a CLOCK0 or a CLEARI statement should be executed at the beginning of the ONEX1 routine. The interrupts would have to be re-enabled before the end of the ONEX1 routine. The ONEX1 interrupt cannot interrupt an ONTIME routine.

The ONTIME statement in MCS BASIC-52 is unique, relative to most BASICS. This powerful statement eliminates the need for the user to "test" the value of the TIME operator periodically throughout the BASIC program.

4.21 DESCRIPTION OF STATEMENTS

STATEMENT: PRINT or P. (? VERSION 1.1 ONLY)

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

The PRINT statement directs MCS BASIC-52 to output to the console device. The value of expressions, strings, literal values, variables or test strings may be printed out. The various forms may be combined in the print list by separating them with commas. If the list is terminated with a comma, the carriage return/ line feed will be suppressed. P. is a "shorthand" notation for PRINT. In Version 1.1 ? is also "shorthand" notation for PRINT.

EXAMPLES:

```
>PRINT 10*10,3*3     >PRINT "MCS-51"        >PRINT 5,1E3
 100  9              MCS-51                  5  1000
```

Values are printed next to one another with two intervening blanks. A PRINT statement with no arguments causes a carriage return/line feed sequence to be sent to the console device.

SPECIAL PRINT FORMATTING STATEMENTS

TAB([expr])

The TAB([expr]) function is used in the PRINT statement to cause data to be printed out in exact locations on the output device. TAB([expr]) tells MCS BASIC-52 which position to begin printing the next value in the print list. If the printhead or cursor is on or beyond the specified TAB position, MCS BASIC-52 will ignore the TAB function.

EXAMPLE:

```
>PRINT TAB(5),"X",TAB(10),"Y"
     X     Y
```

SPC([expr])

The SPC([expr]) function is used in the PRINT statement to cause MCS BASIC-52 to output the number of spaces in the SPC argument.

EXAMPLE:

```
>PRINT A,SPC(5),B
```

may be used to place an additional 5 spaces between the A and B over and above the two that would normally be printed.

4.21 DESCRIPTION OF STATEMENTS

CR

The CR function is interesting and unique to MCS BASIC-52. When CR is used in a PRINT statement it will force a carriage return, but no line feed. This can be used to create one line on a CRT device that is repeatedly updated.

EXAMPLE:

```
>10 FOR I=1 TO 1000
>20 PRINT I,CR,
>30 NEXT I
```

will cause the output to remain only on one line. No line feed will ever be sent to the console device.

USING(special characters)

The USING function is used to tell MCS BASIC-52 what format to display the values that are printed. MCS BASIC-52 "stores" the desired format after the USING statement is executed. So, all outputs following a USING statement will be in the format evoked by the last USING statement executed. The USING statement need not be executed within every PRINT statement unless the programmer wants to change the format. U. is a "shorthand" notation for USING. The options for USING are as follows:

USING(Fx) — This will force MCS BASIC-52 to output all numbers using the floating point format. The value of x determines how many significant digits will be printed. If x equals 0, MCS BASIC-52 will not output any trailing zeros, so the number of digits will vary depending upon the number. MCS BASIC-52 will always output at least 3 significant digits even if x is 1 or 2. The maximum value for x is 8.

EXAMPLE:

```
>10 PRINT USING(F3),1,2,3
>20 PRINT USING(F4),1,2,3
>30 PRINT USING(F5),1,2,3
>40 FOR I=10 TO 40 STEP 10
>50 PRINT I
>60 NEXT I
>RUN

 1.00 E 0   2.00 E 0   3.00 E 0
 1.000 E 0   2.000 E 0   3.000 E 0
 1.0000 E 0   2.0000 E 0   3.0000 E 0
 1.0000 E+1
 2.0000 E+1
 3.0000 E+1
 4.0000 E+1

 READY
```

4.21 DESCRIPTION OF STATEMENTS

USING(#.#) — This will force MCS BASIC-52 to output all numbers using an integer and/or fraction format. The number of "#" 's before the decimal point represents the number of significant integer digits that will be printed in the fraction. The decimal point may be omitted, in which case only integers will be printed. USING may be abbreviated U. USING (###.###), USING(######) and USING(######.##) are all valid in MCS BASIC-52. The maximum number of "#" characters is 8. If MCS BASIC-52 cannot output the value in the desired format (usually because the value is too large) a question mark (?) will be printed to console device, then BASIC will output the number in the FREE FORMAT described below.

EXAMPLE:

```
>10 PRINT USING(##.##),1,2,3
>20 FOR I=1 TO 120 STEP 20
>30 PRINT I
>40 NEXT I
>RUN

  1.00     2.00     3.00
  1.00
 21.00
 41.00
 61.00
 81.00
? 101

READY
```

NOTE: The USING(Fx) and the USING(#.#) formats will always "align" the decimal points when printing a number. This feature makes displayed columns of numbers easy to read.

USING(0) — This argument lets MCS BASIC-52 determine what format to use. The rules are simple, if the number is between ±99999999 and ±.1, BASIC will display integers and fractions. If it is out of this range, BASIC will use the USING(F0) format. Leading and trailing zeros will always be suppressed. After reset, MCS BASIC-52 is placed in the USING(0) format.

4.22 DESCRIPTION OF STATEMENTS

STATEMENT: PRINT# or P.# (?# VERSION 1.1 ONLY)

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

The PRINT#, P.#, and ?# (in Version 1.1 only) statement does the same thing as the PRINT, P. and ? (in Version 1.1 only) statement except that the output is directed to the list device instead of the console device. The BAUD rate to the list device must be initialized by the STATEMENT — BAUD[expr] before the PRINT#, P.#, or, ?# statement is used. All comments that apply to the PRINT, P. or, ? statement apply to the PRINT#, P.#, or ? statement. P.# and ?# (in Version 1.1 only) are "shorthand" notations for PRINT#.

4.23 DESCRIPTION OF STATEMENTS

STATEMENTS: PH0., PH1., PH0.#, PH1.#

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

The PH0. and PH1. statements do the same thing as the PRINT statement except that the values are printed out in a hexadecimal format. The PH0. statement suppresses two leading zeros if the number to be printed is less than 255 (0FFH). The PH1. statement always prints out four hexadecimal digits. The character "H" is always printed after the number when PH0. or PH1. is used to direct an output. The values printed are always truncated integers. If the number to be printed is not within the range of valid integer (i.e. between 0 and 65535 (0FFFFH) inclusive), MCS BASIC-52 will default to the normal mode of print. If this happens no "H" will be printed out after the value. Since integers can be entered in either decimal or hexadecimal form the statements PRINT, PH0., and PH1. can be used to perform decimal to hexadecimal and hexadecimal to decimal conversion. All comments that apply to the PRINT statement apply to the PH0. and PH1. statements. PH0.# and PH1.# do the same thing as PH0. and PH1. respectively, except that the output is directed to the list device instead of the console device.

EXAMPLES:

```
>PH0. 2*2          >PH1. 2*2          >PRINT 99H         >PH0. 100
 04H                0004H              153                64H

>PH0. 1000         >PH1. 1000         >P. 3E8H           >PH0. PI
 3E8H               03E8H              1000               03H
```

4.24 DESCRIPTION OF STATEMENTS

STATEMENT: PRINT@, PH0.@, PH1.@ (VERSION 1.1 ONLY)

MODE: COMMAND AND/OR RUN

TYPE: INPUT/OUTPUT

The PRINT@ (P.@ OR ?@), PH0.@, and PH1.@ statements do the same thing as the PRINT (P.@ or ?@), PH0., and PH1. statements respectively except that the output is directed to a user defined output driver. These statements assume that the user has placed an assembly language output routine in external code memory location 403CH. To enable the @ driver routine the user must SET BIT 27H (39D) in the internal memory of the MCS BASIC-52 device. BIT 27H (39D) is BIT 7 of internal memory location 24H (36D). This BIT can be set by the BASIC statement DBY(24H) = DBY(24H).OR. 80H or by a user supplied assembly language routine. If the user evokes the @ driver routine and this bit is not set, the output will be directed to the console driver. The only reason this BIT must be set to enable the @ driver is that it adds a certain degree of protection from accidently typing LIST@ when no assembly language routine exist. The philosophy here is that if the user sets the bit, the user provides the driver or else!!!

When MCS BASIC-52 calls the user output driver routine at location 403CH, the byte to output is in the accumulator and R5 of register bank 0 (RB0). The user may modify the accumulator (A) and the data pointer (DPTR) in the assembly language output routine, but cannot modify any of the registers in RB0. This is intended to make it real easy for the user to implement a parallel or serial output driver without having to do a PUSH or a POP.

4.25 DESCRIPTION OF STATEMENTS

STATEMENT: PUSH[expr]

MODE: COMMAND AND / OR RUN

TYPE: ASSIGNMENT

The arithmetic expression, or expressions following the PUSH statement are evaluated and then sequentially placed on MCS BASIC-52's ARGUMENT STACK. This statement, in conjunction with the POP statement provide a simple means of passing parameters to assembly language routines. In addition, the PUSH and POP statements can be used to pass parameters to BASIC subroutines and to "SWAP" variables. The last value PUSHED onto the ARGUMENT STACK will be the first value POPPED off the ARGUMENT STACK.

VARIATIONS:

More than one expression can be pushed onto the ARGUMENT stack with a single PUSH statement. The expressions are simply followed by a comma: PUSH[expr],[expr],.....[expr]. The last value PUSHED onto the ARGUMENT STACK will be the last expression [expr] encountered in the PUSH STATEMENT.

EXAMPLES:

```
SWAPPING                SUBROUTINE
VARIABLES               PASSING

>10 A=10                >10 PUSH 1,3,2
>20 B=20                >20 GOSUB 100
>30 PRINT A,B           >30 POP R1,R2
>40 PUSH A,B            >40 PRINT R1,R2
>50 POP A,B             >50 END
>60 PRINT A,B           >100 REM QUADRATIC A=2,B=3,C=1 IN EXAMPLE
>RUN                    >110 POP A,B,C
                        >120 PUSH (-B+SQR(B*B-4*A*C))/(2*A)
10  20                  >130 PUSH (-B-SQR(B*B-4*A*C))/(2*A)
20  10                  >140 RETURN
                        >RUN
READY
>                       -1 -.5

                        READY
                        >
```

4.26 DESCRIPTION OF STATEMENTS

STATEMENT: POP[var]

MODE: COMMAND AND / OR RUN

TYPE: ASSIGNMENT

The top of the ARGUMENT STACK is assigned to the variable following the POP statement and the ARGUMENT STACK is "POPPED" (i.e. incremented by 6). Values can be placed on the stack by either the PUSH statement or by assembly language CALLS. NOTE — If a POP statement is executed and no number is on the ARGUMENT STACK, an A-STACK ERROR will occur.

VARIATIONS:

More than one variable can be popped off the ARGUMENT stack with a single POP statement. The variables are simply followed by a comma (i.e. POP [var],[var], ... [var]).

EXAMPLES:

```
See PUSH statement.
```

COMMENT:

The PUSH and POP statements are unique to MCS BASIC-52. These powerful statements can be used to "get around" the GLOBAL variable problems so often encountered in BASIC PROGRAMS. This problem arises because in BASIC the "main" program and all subroutines used by the main program are required to use the same variable names (i.e. GLOBAL VARIABLES). It is not always convenient to use the same variables in a subroutine as in the main program and you often see programs re-assign a number of variables (i.e. A = Q) before a GOSUB STATEMENT is executed. If the user reserves some variable names JUST for subroutines (i.e. S1, S2) and passes variables on the stack as shown in the previous example, you will avoid any GLOBAL variable problems in MCS BASIC-52.

4.27 DESCRIPTION OF STATEMENTS

STATEMENT: PWM [expr], [expr], [expr]

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

PWM stands for PULSE WIDTH MODULATION. What it does is generate a user defined pulse sequence on P1.2 (bit 2 of I/O PORT 1) of the MCS BASIC-52 device. The first expression following the PWM statement is the number of clock cycles the pulse will remain high. A clock cycle is equal to 12/XTAL, which is 1.085 microseconds at 11.0592 MHz. The second expression is the number of clock cycles the pulse will remain low and the third expression is the total number of cycles the user wishes to output. All expressions in the PWM statement *must be valid integers* (i.e. between 0 and 65535 (0FFFFH) inclusive). Additionally, *the minimum value for the first two expressions in the PWM statement is 25.*

The PWM statement can be used to create "audiable" feedback in a system. In addition, just for fun, the programmer can play music using the PWM statement. More details about using the PWM statement are in the appendix.

EXAMPLE:

```
>PWM 100, 100, 1000
```

At 11.0592 MHz would generate 1000 cycles of a square wave that has a period of 217 microseconds (4608 Hz) on P1.2.

4.28 DESCRIPTION OF STATEMENTS

STATEMENT: REM

MODE: RUN (Version 1.0) COMMAND AND/OR RUN (Version 1.1)

TYPE: CONTROL — PERFORMS NO OPERATION

REM is short for REMark. It does nothing, but allows the user to add comments to a program. Comments are usually needed to make a program a little easier to understand. Once a REM statement appears on a line the entire line is assumed to be a remark, so a REM statement may not be terminated by a colon (:), however, it may be placed after a colon. This can be used to allow the programmer to place a comment on each line.

EXAMPLES:

```
>10 REM INPUT ONE VARIABLE
>20 INPUT A
>30 REM INPUT ANOTHER VARIABLE
>40 INPUT B
>50 REM MULTIPLY THE TWO
>60 Z=A*B
>70 REM PRINT THE ANSWER
>80 PRINT Z


>10 INPUT A : REM INPUT ONE VARIABLE
>20 INPUT B : REM INPUT ANOTHER VARIABLE
>30 Z=A*B : REM MULTIPLY THE TWO
>40 PRINT Z :REM PRINT THE ANSWER
```

The following will NOT work because the entire line would be interpreted as a REMark, so the PRINT statement would not be executed:

```
>10 REM PRINT THE NUMBER : PRINT A
```

NOTE — The reason the REM statement was made executable in the command mode in Version 1.1 of MCS BASIC-52 is that if the user is employing some type of UPLOAD/DOWNLOAD routine with a computer, this lets the user insert REM statements, without line numbers in the text and not download them to the MCS BASIC-52 device. This helps to conserve memory.

4.29 DESCRIPTION OF STATEMENTS

STATEMENT: RETI

MODE: RUN

TYPE: CONTROL

The RETI statement is used to exit from interrupts that are handled by an MCS BASIC-52 program. Specifically, the ONTIME and the ONEX1 interrupts. The RETI statement does the same thing as the RETURN statement except that it also clears a software interrupt flags so interrupts can again be acknowledged. If the user fails to execute the RETI statement in the interrupt procedure, all future interrupts will be ignored.

4.30 DESCRIPTION OF STATEMENTS

STATEMENT: STOP

MODE: RUN

TYPE: CONTROL

The STOP statement allows the programmer to break program execution at specific points in a program. After a program is STOPped variables can be displayed and/or modified. Program execution may be resumed with a CONTinue command. The purpose of the STOP statement is to allow for easy program "debugging." More details of the STOP-CONT sequence are covered in the DESCRIPTION OF COMMAND — CONT section of this manual.

EXAMPLE:

```
>10 FOR I=1 TO 100
>20 PRINT I
>30 STOP
>40 NEXT I
>RUN

 1
STOP - IN LINE 40

READY
>CONT

 2
```

Note that the line number printed out after the STOP statement is executed is the line number following the STOP statement, NOT the line number that contains the STOP statement!!!

4.31 DESCRIPTION OF STATEMENTS

STATEMENT: STRING [expr], [expr]

MODE: COMMAND and/or RUN

TYPE: CONTROL

The STRING [expr],[expr] statement allocates memory for strings. Initially, no memory is allocated for strings. If the user attempts to define a string with a statement such as LET $(1)="HELLO" before memory has been allocated for strings, a MEMORY ALLOCATION ERROR will be generated. The first expression in the STRING [expr],[expr] statement is the total number of bytes the user wishes to allocate for string storage. The second expression denotes the maximum number of bytes that are in each string. These two numbers determine the total number of defined string variables.

You might think that the total number of defined strings would be equal to the first expression in the STRING [expr],[expr] statement divided by the second expression. Ha,ha, do not be so presumptuous. MCS BASIC-52 requires one additional byte for each string, plus one additional byte overall. This means that the statement STRING 100,10 would allocate enough memory for 9 string variables, ranging from $(0) to $(8) and all of the 100 allocated bytes would be used. Note that $(0) is a valid string in MCS BASIC-52.

After memory is allocated for string storage, neither commands, such as NEW nor statements, such as CLEAR, will "de-allocate" this memory. The only way memory can be de-allocated is to execute a STRING 0,0 statement. STRING 0,0 will allocate no memory to string variables.

IMPORTANT NOTE

Every time the STRING [expr],[expr] statement is executed, MCS BASIC-52 executes the equivalent of a CLEAR statement. This is a necessity because string variables and numeric variables occupy the same external memory space. So, after the STRING statement is executed, all variables are "wiped-out." Because of this, string memory allocation should be performed early in a program (like the first statement or so) and string memory should never be "re-allocated" unless the programmer is willing to destroy all defined variables.

4.32 DESCRIPTION OF STATEMENTS

STATEMENTS: UI1 and UI0 (USER INPUT)

MODE: COMMAND and/or RUN

TYPE: CONTROL

UI1

The UI1 statement permits the user to write specific console input drivers for MCS BASIC-52. After UI1 is executed BASIC will call external program memory location 4033H when a console input is requested. The user must provide a JUMP instruction to an ASSEMBLY LANGUAGE INPUT ROUTINE at this location. The appropriate ASCII input from this routine is placed in the 8052AH's accumulator and the user input routine returns back to BASIC by executing an ASSEMBLY LANGUAGE RET instruction. The user must NOT modify any of the 8052AH's registers in the assembly language program with the exception of the MEMORY and REGISTER BANK allocated to the USER. THE ASSEMBLY LANGUAGE LINKAGE section of this manual explains what memory MCS BASIC-52 allocates to the user and how the user may allocate additional memory if needed.

In addition to providing the INPUT driver routine for the UI1 statement, the user must also provide a CONSOLE STATUS CHECK routine. This routine checks to see if the CONSOLE DEVICE has a character ready for MCS BASIC-52 to read. BASIC CALLS external memory location 4036H to check the CONSOLE STATUS. The CONSOLE STATUS ROUTINE sets the CARRY BIT to 1 (C = 1) if a character is ready for BASIC to read and CLEARS the CARRY BIT (C = 0) if no character is ready. Again, the contents of the REGISTERS must not be changed. MCS BASIC-52 uses the CONSOLE STATUS CHECK routine to examine the keyboard for a control-C character during program execution and during a program LISTING. This routine is also used to perform the GET operation.

UI0

The UI0 statement assigns the console input console routine back to the software drivers resident on the MCS BASIC-52 device. UI0 and UI1 may be placed anywhere within a program. This allows the BASIC program to accept inputs from different devices at different times.

NOTE: The UI0 and UI1 function is controlled by BIT 30 (1EH) in the 8052AH's internal memory. BIT 30 is in internal memory location 35.6 (23.6H) i.e. the sixth bit in internal memory location 35 (23H). When BIT 30 is SET (BIT 30 = 1), the user routine will be called. When BIT 30 is CLEARED (BIT 30 = 0), the MCS BASIC-52 input driver routine will be used. The assembly language programmer can use this information to change the input device selection in assembly language.

4.33 DESCRIPTION OF STATEMENTS

STATEMENTS: UO1 and UO0 (USER OUTPUT)

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

UO1

The UO1 STATEMENT permits the user to write specific console output drivers for MCS BASIC-52. After UO1 is executed BASIC will call external program memory location 4030H when a console output is requested. The user must provide a JUMP instruction to an ASSEMBLY LANGUAGE OUTPUT ROUTINE at this location. MCS BASIC-52 places the output character in REGISTER 5 (R5) of REGISTER BANK 0 (RB0). The user returns back to BASIC executing an assembly language RET instruction. The user must NOT modify any of the 8052AH's REGISTERS, including the ACCUMULATOR during the user output procedure with the exception of the MEMORY and REGISTER BANK allocated to the user. UO1 gives the user the freedom to write custom output routines for MCS BASIC-52.

UO0

UO0 STATEMENT assigns the console output routine back to the software drivers resident on the MCS BASIC-52 device. UO0 and UO1 may be placed anywhere within a program. This allows the BASIC program to output characters to different devices at different times.

NOTE: The UO0 and UO1 function is controlled by BIT 28 (1CH) in the 8052AH's internal memory. BIT 28 is in the internal memory location 35.4 (23.4H), i.e. the fourth bit in the internal memory location 35 (28H). When BIT 28 is SET (BIT 28 = 1), the user routines will be called. When BIT 28 is cleared, (BIT 28 = 0), the MCS BASIC-52 output drivers will be used. The assembly language programmer can use this information to change the output device selection in assembly language.

4.34 DESCRIPTION OF STATEMENTS

STATEMENT: IDLE (VERSION 1.1 ONLY)

MODE: RUN

TYPE: CONTROL

The IDLE statement forces the MCS BASIC-52 device into a "wait until interrupt mode." Execution of statements is halted until either an ONTIME [expr], [ln num] or an ONEX1 [ln num] interrupt is received. The user must make sure that one or both of these interrupts have been enabled before executing the IDLE instruction or else the MCS BASIC-52 device will enter a "wait forever mode" and for all practical purposes the system will have crashed.

When an ONTIME [expr], [ln num] or an ONEX1 [ln num] is received while in the IDLE mode, the MCS BASIC-52 device will execute the interrupt routine, then execute the statement following the IDLE instruction. Hence, the execution of the IDLE instruction is terminated when an interrupt is received.

While in the IDLE mode, the MCS BASIC-52 device asserts the /DMA ACKNOWLEDGE pin (PORT 1, BIT 6 = 0) to indicate that the IDLE instruction is active and that no external bus activity will occur. This PIN is physically pin 7 on the MCS BASIC-52 device. When the MCS BASIC-52 device exits from the IDLE mode, this pin is placed back into the logically 1 (non-active) state.

The user may also exit from the IDLE mode with an assembly language interrupt routine. This is accomplished by setting BIT 33 (21H) (which is in Bit addressable RAM location 36.1) when returning from the assembly language interrupt routine. If this bit is not set by the user, the MCS BASIC-52 device will remain in the IDLE mode when the user assembly language routine returns to BASIC.

An attempt to execute the IDLE statement in the direct mode will yield a BAD SYNTAX ERROR.

4.35 DESCRIPTION OF STATEMENTS

STATEMENT: RROM [interger] (VERSION 1.1 ONLY)

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

RROM stands for RUN ROM. What it does is select a program in the EPROM file, then execute the program. The integer after the RROM statement selects what program in the EPROM file is to be executed. In the COMMAND mode RROM 2 would be equivalent to typing ROM 2, then RUN. But, notice that RROM [integer] is a statement. This means that a program that is already executing can actually force the execution of a completely different program that is in the EPROM file. This gives the user the ability to "change programs" on the fly.

If the user executes a RROM [integer] statement and an invalid integer is entered (say 6 programs are contained in the EPROM file and the user enters RROM 8, or no EPROM is in the system), no error will be generated and MCS BASIC-52 will execute the statement following the RROM [integer] statement.

NOTE — Every time the RROM [integer] statement is executed, all variables and strings are set equal to zero, so variables and strings CANNOT be passed from one program to another by using the RROM [integer] statement. Additionally, all MCS BASIC-52 evoked interrupts are cleared.

4.36 DESCRIPTION OF STATEMENTS

STATEMENTS: LD@ [expr] and ST@ [expr] (VERSION 1.1 ONLY)

MODE: COMMAND AND/OR RUN

TYPE: INPUT/OUTPUT

ST@

The ST@ [expr] statement lets the user specify where MCS BASIC-52 floating point numbers are to be stored. The expression [expr] following the ST@ statement specifies the address of where the number is to be stored and **the number is assumed to be on the argument stack.** The ST@ [expr] statement is designed to be used in conjunction with the LD@ [expr] statement. The purpose of these two statements is to allow the user to save floating point numbers anywhere in memory with the assumption that the user will employ some type of battery back-up or non-volatile scheme with this memory.

LD@

The LD@ [expr] statement lets the user retrieve floating point numbers that were saved with the ST@ [expr] statement. The expression [expr] following the LD@ statement specifies where the number is stored and after executing the LD@ [expr] statement, **the number is placed on the argument stack.**

EXAMPLE: Saving and retrieving a ten element array at location array at location 0F000H

```
10    REM *** ARRAY SAVE ***
20    FOR I = 0 TO 9
30    PUSH A(I) : REM PUT ARRAY VALUE ON STACK
40    ST@ 0F005H+6*I : REM STORE IT, SIX BYTES PER NUMBER
50    NEXT I
60    REM *** GET ARRAY ***
70    FOR I = 0 TO 9
80    LD@ 0F005H+6*I
90    POP B(I)
100   NEXT I
```

Remember that each floating point number requires 6 bytes of storage. Also note that expression in the ST@ [expr] and LD@ [expr] statements point to the most significant byte of the stored number. Hence, ST@ (0F005H) would save the number in locations 0F005H, 0F004H, 0F003H, 0F002H, 0F001H, and 0F000H.

4.37 DESCRIPTION OF STATEMENTS

STATEMENT: PGM

MODE: COMMAND AND/OR RUN

TYPE: INPUT/OUTPUT

The PGM statement gives the user the ability to program an EPROM or EEPROM while executing a BASIC program. **The PGM statement requires that the user set up internal memory locations 18H (24D), 19H (25D), 1AH (26D), 1BH (27H), 1EH (30D) and 1GH (31D).** Note that these internal memory locations are normally reserved for the user!!

The User must initialize these internal memory locations with the following:

EXAMPLE:

```
LOCATION          CONTENTS

  1BH: 19H          THE ADDRESS OF THE SOURCE INFORMATION THAT IS TO BE
 (27D: 25D)         PROGRAMMED INTO THE EPROM , LOCATION 19H IS THE LOW
                    BYTE AND LOCATION 1BH IS THE HIGH BYTE

  1AH: 18H          THE ADDRESS - 1 OF THE EPROM LOCATION(S) THAT ARE TO
 (26D: 24D)         BE PROGRAMMED, LOCATION 18H IS TH LOW BYTE AND
                    LOCATION 1AH IS THE HIGH BYTE

  1FH: 1EH          THE NUMBER OF BYTES THAT THE USER WANTS TO PROGRAM
 (31D: 30D)         LOCATION 1EH IS THE LOW BYTE AND LOCATION 1FH IS
                    THE HIGH BYTE
```

The user must also initialize the width of the desired EPROM programming pulse and store the value in internal memory locations 40H (64D) (high byte) and 41H (65D) (low byte). The reload for a 50 millisecond EPROM programming pulse is calculated as follows:

```
10      REM  R = RELOAD VALUE, W = WIDTH IN SECONDS (50 MILLISECONDS)
20      W = .05
30      R = 65536 - W * XTAL/12
40      DBY(40H) = R/256
50      DBY(41H) = R .AND. OFFH
```

In addition, the user must also SET or CLEAR BIT 38.3 (26.3H) to select the INTELligent EPROM programming algorithm. The Bit is SET to select INTELligent programming and CLEARED to select the normal 50 millisecond algorithm. To SET the BIT, execute a DBY(38) = DBY(38) .OR. 8H Statement, to CLEAR the BIT, execute a DBY(38) = DBY(38) .AND. 0F7H instruction.

4.3?7 DESCRIPTION OF STATEMENTS

IMPORTANT NOTE!

When executed in the RUN mode, The PGM statement will not generate an error if the EPROM fails to program properly. Instead, the control of the program will be passed back to the user just as if the EPROM programmed properly. The user must then examine locations 1EH and 1FH. If the contents of locations 1EH and 1FH both equal zero, then the EPROM programmed properly. If they do not, then an ERROR occurred during the programming process. The user can then examine locations 1AH:18H to determine what location in the EPROM failed to program.

Well, this sounds like a lot to do just to program an EPROM, but it's not so bad. The following program is an example of a universal EPROM/EEPROM programmer built around MCS BASIC-52. This program can program a block of RAM into an EPROM or EEPROM that is addressed at 8000H or above.

EXAMPLE:

```
10    PRINT "UNIVERSAL PROM PROGRAMMER" :  PRINT "WHAT TYPE OF DEVICE ?"
20    PRINT  :  PRINT "1 = EEPROM" :  PRINT "2 = INTELLIGENT EPROM"
30    PRINT "3 = NORMAL (50 MS) EPROM" :  PRINT  :  INPUT "TYPE (1,2,3) - ",T
40    ON (T-1) GOSUB 340,350,360
50    REM this sets up intelligent programming if needed
60    IF W=.001 THEN DBY(26)=DBY(26).OR.8 ELSE DBY(26)=DBY(26).AND.0F7H
70    REM calculate pulse width and save it
80    PUSH (65536-(W*XTAL/12)) :  GOSUB 380
90    POP G1 : DBY(40H)=G1 :  POP G1 : DBY(41H)=G1 :  PRINT
100   INPUT " STARTING DATA ADDRESS - ",S :  IF S<512.OR.S>0FFFFH THEN 100
110   PRINT  :  INPUT " ENDING DATA ADDRESS - ",E
120   IF E<S.OR.E>0FFFFH THEN 110
130   PRINT  :  INPUT " PROM ADDRESS - ",P :  IF P<8000H.OR.P>0FFFFH THEN 130
140   REM calculate the number of bytes to program
150   PUSH (E-S)+1 :  GOSUB 380 :  POP G1 : DBY(31)=G1 :  POP G1 : DBY(30)=G1
160   REM set up the eprom address
170   PUSH (P-1) :  GOSUB 380 :  POP G1 : DBY(26)=G1 :  POP G1 : DBY(24)=G1
180   REM set up the source address
190   PUSH S :  GOSUB 380 :  POP G1 : DBY(27)=G1 :  POP G1 : DBY(25)=G1
200   PRINT  :  PRINT "TYPE A 'CR' ON THE KEYBOARD WHEN READY TO PROGRAM"
210   REM wait for a 'cr' then program the eprom
220  X=GET :  IF X<>0DH THEN 220
230   REM program the eprom
240   PGM
250   REM see if any errors
260   IF (DBY(30).OR.DBY(31))=0 THEN  PRINT "PROGRAMMING COMPLETE" :  END
270   PRINT  :  PRINT "***ERROR***ERROR***ERROR***" :  PRINT
280   REM these routines calculate the address of the source and
290   REM eprom location that failed to program
300  S1=DBY(25)+256*DBY(27) : S1=S1-1 : D1=DBY(24)+256*DBY(26)
310   PH0. "THE VALUE ",XBY(S1), :  PH1. " WAS READ AT LOCATION ",S1 :  PRINT
320   PH0. "THE EPROM READ ",XBY(D1), :  PH1. " AT LOCATION ",D1 :  END
330   REM these subroutines set up the pulse width
340  W=.0005 :  RETURN
350  W=.001 :  RETURN
360  W=.05 :  RETURN
370   REM this routine takes the top of stack and returns high, low bytes
380   POP G1 :  PUSH (G1.AND.0FFH) :  PUSH (INT(G1/256)) :  RETURN
```

CHAPTER 5
Description of Arithmetic/Logic Operators and Expressions

5.1 DUAL OPERAND OPERATORS

MCS BASIC-52 contains a complete set of arithmetical and logical operators. Operators are divided into two groups, dual operand or dyadic operators and single operand or unary operators. The generalized form of all dual operand instructions is as follows:

[expr] OP [expr], where OP is one of the following operators:

+ ADDITION OPERATOR

EXAMPLE:

```
PRINT 3+2
5
```

/ DIVISION OPERATOR

EXAMPLE:

```
PRINT 100/5
20
```

** EXPONENTIATION OPERATOR

Raises the first expression to the power of the second expression. The power any number can be raised to is limited to 255. The notation ** was chosen instead of the sometimes used ↑ symbol because the "up arrow" symbol appears different on various terminals. To eliminate confusion the ** notation was chosen.

EXAMPLE:

```
PRINT 2**3
8
```

* MULTIPLICATION OPERATOR

EXAMPLE:

```
PRINT 3*3
9
```

– SUBTRACTION OPERATOR

EXAMPLE:

```
PRINT 9-6
3
```

5.1 DUAL OPERAND OPERATIONS

.AND. LOGICAL AND OPERATOR

EXAMPLE:

```
PRINT 3. AND. 2
2
```

.OR. LOGICAL OR OPERATOR

EXAMPLE:

```
PRINT 1. OR. 4
5.
```

.XOR. LOGICAL EXCLUSIVE OR OPERATOR

EXAMPLE:

```
PRINT 7. XOR. 6
1
```

COMMENTS ON LOGICAL OPERATORS .AND., .OR., and .XOR.

These operators perform a BIT-WISE logical function on valid INTEGERS. That means both arguments for these operators must be between 0 and 65535 (0FFFFH) inclusive. If they are not, MCS BASIC-52 will generate a BAD ARGUMENT ERROR. All non-integer values are truncated, NOT rounded.

You may wonder why the notation .OP. was chosen for the logical functions. The only reason for this is that MCS BASIC-52 eliminates ALL spaces when it processes a user line and inserts spaces before and after STATEMENTS when it LISTS a user program. MCS BASIC-52 does not insert spaces before and after operators. So, if the user types in a line such as 10 A = 10 * 10, this line will be listed as 10 A=10*10. All spaces entered by the user before and after the operator will be eliminated. The .OP. notation was chosen for the logical operators because a line entered as 10 B = A AND B would be listed as 10 B=AANDB. This just looked confusing, so the dots were added to the logical instructions and the previous example would be listed as 10 B=A.AND.B, which is easier to read.

5.2.1 UNARY OPERATORS — GENERAL PURPOSE

ABS([expr])

Returns the ABSOLUTE VALUE of the expression.

EXAMPLES:

```
PRINT ABS(5)          PRINT ABS(-5)
5                     5
```

NOT([expr])

Returns a 16 bit one's complement of the expression. The expression must be a valid *integer* (i.e. between 0 and 65535 (0FFFFH) inclusive). Non-integers will be truncated, not rounded.

EXAMPLES:

```
PRINT NOT(65000)      PRINT NOT(0)
535                   65535
```

INT([expr])

Returns the integer portion of the expression.

EXAMPLES:

```
PRINT INT(3.7)        PRINT INT(100.876)
3                     100
```

SGN([expr])

Will return a value of +1 if the argument is greater than zero, zero if the argument is equal to zero, and −1 if the argument is less than zero.

EXAMPLES:

```
PRINT SGN(52)         PRINT SGN(0)          PRINT SGN(-8)
1                     0                     -1
```

5.2.1 UNARY OPERATORS — GENERAL PURPOSE

SQR([expr])

Returns the square root of the argument. The argument may not be less than zero. The result returned will be accurate to within +/− a value of 5 on the least significant digit.

EXAMPLES:

```
PRINT SQR(9)          PRINT SQR(45)         PRINT SQR(100)
3                     6.7082035             10
```

RND

Returns a pseudo-random number in the range between 0 and 1 inclusive. The RND operator uses a 16-bit binary seed and generates 65536 pseudo-random numbers before repeating the sequence. The numbers generated are specifically between 0/65535 and 65535/65535 inclusive. Unlike most BASICS, the RND operator in MCS BASIC-52 does not require an argument or a dummy argument. In fact, if an argument is placed after the RND operator, a BAD SYNTAX error will occur.

EXAMPLES:

```
PRINT RND
 30278477
```

PI

PI is not really an operator, it is a stored constant. In MCS BASIC-52, PI is stored as 3.1415926. Math experts will notice that PI is actually closer to 3.141592653, so proper rounding for PI should yield the number 3.1415927. The reason MCS BASIC-52 uses a 6 instead of a 7 for the last digit is that errors in the SIN, COS and TAN operators were found to be greater when the 7 was used instead of 6. This is because the number PI/2 is needed for these calculations and it is desireable, for the sake of accuracy to have the equation PI/2 + PI/2 = PI hold true. This cannot be done if the last digit in PI is an odd number, so the last digit of PI was rounded to 6 instead of 7 to make these calculations more accurate.

5.2.2 UNARY OPERATORS — LOG FUNCTIONS

LOG([expr])

Returns the natural logarithm of the argument. The argument must be greater than 0. This calculation is carried out to 7 significant digits.

EXAMPLES:

```
PRINT LOG(12)          PRINT LOG(EXP(1))
2.484906               1
```

EXP([expr])

This function raises the number "e" (2.7182818) to the power of the argument.

EXAMPLES:

```
PRINT EXP(1)           PRINT EXP(LOG(2))
2.7182818              2
```

5.2.3 UNARY OPERATORS — TRIG FUNCTIONS

SIN([expr])

Returns the SIN of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between ±200000.

EXAMPLES:

```
PRINT SIN(PI/4)        PRINT SIN(0)
 7071067               0
```

COS([expr])

Returns the COS of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between ±200000.

EXAMPLES:

```
PRINT COS(PI/4)        PRINT COS(0)
 7071067               1
```

5.2.3 UNARY OPERATORS — TRIG FUNCTIONS

TAN([expr])

Returns the TAN of the argument. The argument is expressed in radians. The argument must be between ±200000.

EXAMPLES:

```
PRINT TAN(PI/4)     PRINT TAN(0)
1                   0
```

ATN([expr])

Returns the ARCTANGENT of the argument. The result is in radians. Calculations are carried out to 7 significant digits. The ATN operator returns a result between −PI/2 (3.1415926/2) and PI/2.

EXAMPLES:

```
PRINT ATN(PI)       PRINT ATN(1)
1.2626272            78539804
```

COMMENTS ON TRIG FUNCTIONS

The SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. This reduction is accomplished by the following equation:

REDUCED ARGUMENT = **(user arg/PI − INT(user arg/PI)) * PI**

The REDUCED ARGUMENT, from the above equation, will be between 0 and PI. The REDUCED ARGUMENT is then tested to see if it is greater than PI/2. If it is, then it is subtracted from PI to yield the final value. If it isn't, then the REDUCED ARGUMENT is the final value.

Although this method of angle reduction provides a simple and economical means of generating the appropriate arguments for a Taylor series, there is an accuracy problem associated with this technique. The accuracy problem is noticed when the user argument is large (i.e. greater than 1000). That is because significant digits, in the decimal (fraction) portion of REDUCED ARGUMENT are lost in the (user arg/PI − INT(user arg/PI)) expression. As a general rule, try to keep the arguments for the TRIG functions as small as possible!

5.3 UNDERSTANDING PRECEDENCE OF OPERATORS

The hierarchy of mathematics dictates that some operations are carried out before others. If you understand the hierarchy of mathematics, it is possible to write complex expressions using only a minimum amount of parentheses. It's easy to illustrate what precedence is all about, examine the following equation:

4+3*2 = ?

Should you add (4+3) then multiply seven by 2, or should you multiply (3*2) then add 4? Well, the hierarchy of mathematics says that multiplication has precedence over addition, so you would multiply (3*2) first then add 4. So,

4+3*2 = 10

The rules for the hierarchy of math are simple. When an expression is scanned from left to right an operation is not performed until an operator of lower or equal precedence is encountered. In the example addition could not be performed because multiplication has higher precedence. The precedence of operators from highest to lowest in MCS BASIC-52 is as follows:

1)OPERATORS THAT USE PARENTHESES ( )

2)EXPONENTATION (**)

3)NEGATION (−)

4)MULTIPLICATION (*) AND DIVISION (/)

5)ADDITION (+) AND SUBTRACTION (−)

6)RELATIONAL EXPRESSIONS (=, <>, >, >=, <, <=)

7)LOGICAL AND (.AND.)

8)LOGICAL OR (.OR.)

9)LOGICAL XOR (.XOR.)

Relative to operator precedence, the rule of thumb should always be; when in doubt, use parentheses.

5.4 HOW RELATIONAL EXPRESSIONS WORK

Relational expressions involve the operators =, <>, >, >=, <, and <=. These operators are typically used to "test" a condition. In MCS BASIC-52 relational operators return a result of 65535 (0FFFFH) if the relational expression is true, and a result of 0, if the relation expression is false. But, where is the result returned? It is returned to the argument stack. Because of this, it's possible to actually display the result of a relational expression.

EXAMPLES:

```
PRINT 1=0      PRINT 1>0      PRINT A<>A      PRINT A=A
0              65535          0               65535
```

It may seem strange to have a relational expression actually return a result, but it offers a unique benefit in that relational expressions can actually be "chained" together using the logical operators .AND., .OR., and .XOR.. This makes it possible to test a rather complex condition with ONE statement.

EXAMPLE:

```
>10 IF A<B. AND. A>C. OR. A>D THEN. . . . . .
```

Additionally, the NOT([expr]) operator can be used.

EXAMPLE:

```
>10 IF NOT(A>B). AND. A<C THEN. . . . . .
```

By "chaining" together relational expressions with logical operators, it is possible to test very particular conditions with one statement. When using logical operators to link together relational expressions, it is very important that the programmer pay careful attention to the precedence of operators. The logical operators were assigned lower precedence, relative to relational expressions, just to make the linking of relational expressions possible without using parentheses.

CHAPTER 6
Description of String Operators

6.1 WHAT ARE STRINGS?

A string is a character or a bunch of characters that are stored in memory. Usually, the characters stored in a string make up a word or a sentence. Strings are handy because they allow the programmer to deal with words instead of numbers. This is useful because it allows one to write "friendly" programs, where individuals can be referred to by their names instead of a number.

MCS BASIC-52 contains ONE dimensioned string variable, $([expr]). The dimension of the string variable (the [expr] value) ranges from 0 to 254. This means that 255 different strings can be defined and manipulated in MCS BASIC-52. Initially, NO memory is allocated for strings. Memory is allocated by the STRING [expr], [expr] STATEMENT. The details of this statement are covered in the DESCRIPTION OF STATEMENTS chapter of this manual.

In MCS BASIC-52, strings can be defined in two ways, with the LET STATEMENT and with the INPUT STATEMENT.

EXAMPLE:

```
>10 STRING 100,20
>20 $(1)="THIS IS A STRING, "
>30 INPUT "WHAT'S YOUR NAME? - ",$(2)
>40 PRINT $(1),$(2)
>RUN

WHAT'S YOUR NAME? - FRED

THIS IS A STRING, FRED
```

STRINGS can also be assigned to each other with a LET statement.

EXAMPLE:

```
$(2)=$(1)
```

Would assign the STRING value in $(1) to the STRING $(2).

6.2 THE ASC OPERATOR

In MCS BASIC-52, two operators manipulate STRINGS. These operators are ASC( ) and CHR( ). Admittedly, the string operators contained in MCS BASIC-52 are not quite as powerful as the string operators contained in some BASICS. But surprisingly enough, by using the string operators available in MCS BASIC-52 it is possible to manipulate strings in almost any way imaginable. This in itself is a commendable feat since MCS BASIC-52 was designed primarily to be a sophisticated BASIC language oriented controller, not a string manipulator. The string operators available in MCS BASIC-52 are as follows:

ASC( )

The ASC( ) operator returns the integer value of the ASCII character placed in the parentheses.

EXAMPLE:

```
>PRINT ASC(A)
 65
```

65 is the decimal representation for the ASCII character "A." In addition, individual characters in a predefined ASCII string can be evaluated with the ASC( ) operator.

EXAMPLE:

```
>10 $(1)="THIS IS A STRING"
>20 PRINT $(1)
>30 PRINT ASC($(1),1)
>RUN

 THIS IS A STRING
 84
```

When the ASC( ) operator is used in the manner shown above, the $([expr]) denotes what string is being accessed and the expression after the comma "picks out" an individual character in the string. In the above example, the first character in the string was picked out and 84 is the decimal representation for the ASCII character "T."

6.2 THE ASC OPERATOR

EXAMPLE:

```
>10 $(1)="ABCDEFGHIJKL"
>20 FOR X=1 TO 12
>30 PRINT ASC($(1),X),
>40 NEXT X
>RUN

 65   66   67   68   69   70   71   72   73   74   75   76
```

The numbers printed in the previous example are the values that represent the ASCII characters A,B,C, . . . L.

Additionally, the ASC( ) operator can be used to change individual characters in a defined string.

EXAMPLE:

```
>10 $(1)="ABCDEFGHIJKL"
>20 PRINT $(1)
>30 ASC($(1),1)=75
>40 PRINT $(1)
>50 ASC($(1),2)=ASC($(1),3)
>60 PRINT $(1)
>RUN

ABCDEFGHIJKL
KBCDEFGHIJKL
KCCDEFGHIJKL
```

In general, the ASC( ) operator lets the programmer manipulate individual characters in a string. A simple program can determine if two strings are identical.

EXAMPLE:

```
>10 $(1)="SECRET" : REM SECRET IS THE PASSWORD
>20 INPUT "WHAT'S THE PASSWORD - ",$(2)
>30 FOR I=1 TO 6
>40 IF ASC($(1),I)=ASC($(2),I) THEN NEXT I ELSE 70
>50 PRINT "YOU GUESSED IT!"
>60 END
>70 PRINT "WRONG, TRY AGAIN" : GOTO 20
>RUN

WHAT'S THE PASSWORD - SECURE
WRONG, TRY AGAIN
WHAT'S THE PASSWORD - SECRET
YOU GUESSED IT
```

6.3 THE CHR OPERATOR

CHR( )

The CHR( ) operator is the converse of the ASC( ) operator. It converts a numeric expression to an ASCII character.

EXAMPLE:

```
>PRINT CHR(65)
A
```

Like the ASC( ) operator, the CHR( ) operator can also "pick out" individual characters in a defined ASCII string.

EXAMPLE:

```
>10 $(1)="MCS BASIC-52"
>20 FOR I=1 TO 12 : PRINT CHR($(1),I), : NEXT I
>30 PRINT : FOR I=12 TO 1 STEP -1
>40 PRINT CHR($(1),I), : NEXT I
>RUN

MCS BASIC-52
25-CISAB SCM
```

In the above example, the expressions contained within the parentheses, following the CHR operator have the same meaning as the expressions in the ASC( ) operator.

Unlike the ASC( ) operator, the CHR( ) operator CANNOT be assigned a value. A statement such as CHR($(1),1) = H, is INVALID and will generate a BAD SYNTAX ERROR. Use the ASC( ) operator to change a value in a string. The CHR( ) operator can only be used within a print statement!

CHAPTER 7
Special Operators

7.1 SPECIAL FUNCTION OPERATORS

SPECIAL FUNCTION OPERATORS are called SPECIAL FUNCTION OPERATORS because they directly manipulate the I/O hardware and the memory addresses on the 8052AH device. All SPECIAL FUNCTION OPERATORS, with the exception of CBY([expr]) and GET, can be placed on either side of the replacement operator (=) in a LET STATEMENT.

EXAMPLES:

```
A = DBY(100) and DBY(100) = A+2
```

Both of the above are valid statements in MCS BASIC-52. The SPECIAL FUNCTION OPERATORS in MCS BASIC-52 include the following:

CBY([expr])

The CBY([expr]) operator is used to retrieve data from the PROGRAM or CODE MEMORY address space of the 8052AH. Since CODE memory cannot be written into on the 8052AH, the CBY([expr]) operator cannot be assigned a value. It can only be read.

EXAMPLE: A = CBY(1000) Causes the value in code memory space 1000 to be assigned to the variable A. The argument for the CBY([expr]) operator MUST be a valid integer (i.e. between 0 and 65535 (0FFFFH)). If it is not, a BAD ARGUMENT ERROR will occur.

DBY([expr])

The DBY([expr]) operator is used to retrieve or assign a value to the 8052AH's internal data memory. Both the value and argument in the DBY operator must be between 0 and 255 inclusive. This is because there are only 256 internal memory locations in the 8052AH and one byte can only represent a quantity between 0 and 255 inclusive.

EXAMPLES:

```
A=DBY(B) and DBY(250) = CBY(1000)
```

The first example would assign variable A the value that is in internal memory location B. B would have to be between 0 and 255. The second example would load internal memory location 250 with the same value that is in program memory location 1000.

7.1 SPECIAL FUNCTION OPERATORS

XBY([expr])

The XBY([expr]) operator is used to retrieve or assign a value to the 8052AH's external data memory. The argument in the XBY([expr]) operator must be a valid integer (i.e. between 0 and 65535 (0FFFFH)). The value assigned to the XBY([expr]) operator must be between 0 and 255. If it is not a BAD ARGUMENT ERROR will occur.

EXAMPLES:

```
XBY(4000H)=DBY(100) and A=XBY(0F000H)
```

The first example would load external memory location 4000H with the same value that was in internal memory location 100. The second example would make the variable A equal to the value in external memory location 0F000H.

GET

The GET operator only produces a meaningful result when used in the RUN mode. It will always return a result of zero in the command mode. What GET does is read the console input device. Actually, it takes a "snapshot" of the console input device. If a character is available from the console device, the value of the character will be assigned to GET. After GET is read in the program, GET will be assigned the value of zero until another character is sent from the console device. The following example will print the decimal representation of any character sent from the console:

EXAMPLE:

```
>10 A=GET
>20 IF A<>0 THEN PRINT A
>30 GOTO 10
>RUN

 65      (TYPE "A" ON CONSOLE)
 49      (TYPE "1" ON CONSOLE)
 24      (TYPE "CONTROL-X" ON CONSOLE)
 50      (TYPE "2" ON CONSOLE)
```

The reason the GET operator can be read only once before it is assigned a value of zero is that this implementation guarantees that the first character entered will always be read, independent of where the GET operator is placed in the program.

7.1 SPECIAL FUNCTION OPERATORS

The following operators directly manipulate the 8052AH's special function registers. Specific details of the operation of these registers is in the MICROCONTROLLER USERS HANDBOOK, available from INTEL.

IE

The IE operator is used to retrieve or assign a value to the 8052AH's special function register IE. Since the IE register on the 8052AH is a BYTE register, the value assigned to IE must be between 0 and 255. The IE register on the 8052AH contains an unused bit, BIT IE.6. Since this bit is "undefined," it may be read as a random one or zero, so the user may want to mask this bit when reading the IE register. This can be done with a statement like A = IE.AND.OBFH. The only statements in MCS BASIC-52 that write to the IE register are the CLOCK0, CLOCK1, ONEX1, CLEAR, and CLEARI statements.

EXAMPLES:

```
IE = 81H  and  A = IE. AND. OBFH
```

IP

The IP operator is used to retrieve or assign a value to the 8052AH's special function register IP. Since the IP register on the 8052AH is a BYTE register, the value assigned to IP must be between 0 and 255. The IP register on the 8052AH contains two unused bits, BIT IP.6 and IP.7. Since these bits are "undefined," they may be read as a random 1 or 0, so the user may want to mask these bits when reading the IP register. This can be done with a statement such as B = IP.AND.3FH. MCS BASIC-52 does not write to the IP register during initialization, so user can establish whatever interrupt priorities are required in a given application.

EXAMPLES:

```
IP = 3  and  A = IP. AND. 3FH
```

PORT1

The PORT1 operator is used to retrieve or assign a value to the 8052AH's P1 I/O port. Since P1 on the 8052AH is a BYTE wide register, the value assigned to P1 must be between 0 and 255 inclusive. Certain bits on P1 have pre-defined functions. If the user does not implement any of the hardware associated with these pre-defined functions, The PORT1 instruction can be used in any manner appropriate in the application.

7.1 SPECIAL FUNCTION OPERATORS

PCON

The PCON operator is used to retrieve or assign a value to the 8052AH's PCON register. In the 8052AH, only the most significant bit of the PCON register is used, all other bits are undefined. Setting this bit will double the baud rate if TIMER/COUNTER 1 is used as the baud rate generator for the serial port. PCON is a byte register.

RCAP2

The RCAP2 operator is used to retrieve and/or assign a value to the 8052AH's special function registers RCAP2H and RCAP2L. This operator treats RCAP2H and RCAP2L as a 16-bit register pair. RCAP2H is the high byte and RCAP2L is the low byte. The RCAP2H and RCAP2L registers are the reload/capture registers for TIMER2. The user must use caution when writing to RCAP2 register because RCAP2 controls the BAUD rate of the serial port on the MCS BASIC-52 device. The following can be used to determine what BAUD rate the MCS BASIC-52 device is operating at:

```
BAUD = XTAL/(32*(65536-RCAP2) )
```

T2CON

The T2CON operator is used to retrieve and/or assign a value to the 8052AH's special function register T2CON. The T2CON is a byte register that controls TIMER2's mode of operation and determines which timer (TIMER1 or TIMER2) is used as the 8052AH's baud rate generator. MCS BASIC-52 initializes T2CON with the value 52 (34H) and assumes that its value is never changed. Randomly changing the value of T2CON, without knowing what you are doing can "crash" the serial port on the 8052AH. Beware!

7.1 SPECIAL FUNCTION OPERATORS

TCON

The TCON operator is used to retrieve and/or assign value to the 8052AH's special function register TCON. TCON is a byte register that is used to enable or disable TIMER0 and TIMER1, plus the interrupts that are associated with these timers. Additionally, TCON determines whether the external interrupt pins on the 8052AH are operating in a level sensitive or edge-triggered mode. MCS BASIC-52 initializes TCON with the value 244 (0F4H) and assumes that it is never changed. The value 244 (0F4H) places both TIMER0 and TIMER1 in the run (enabled) mode. If the user disables the operation of TIMER0, by clearing BIT 4 in the TCON register, the REAL TIME CLOCK will NOT work. If the user disables the operation of TIMER1, by clearing BIT 6 in the TCON register, the EPROM programming routines, the software serial port, and the PWM statement will NOT work. Use caution when changing TCON!!!

TMOD

The TMOD operator is used to retrieve and/or assign a value to the 8052AH's special function register TMOD. TMOD is a byte register that controls TIMER0 and TIMER1's mode of operation. MCS BASIC-52 initializes the TCON register with a value of 16 (10H). The value 16 (10H) places TIMER0 in mode 0, which is a 13-bit counter mode and TIMER1 in mode 1, which is a 16-bit counter mode. MCS BASIC-52 assumes that the modes of these two timer/counters are never changed. If the user changes the mode of TIMER0, the REAL TIME CLOCK will not operate properly. If the user changes the mode of TIMER1, EPROM programming, the software serial port, and the PWM statement will not work properly. If the user does not use these features available in MCS BASIC-52, either timer/counter can be placed in any mode required by the specific application.

7.1 SPECIAL FUNCTION OPERATORS

TIME

The TIME operator is used to retrieve and/or assign a value to the REAL TIME CLOCK resident in MCS BASIC-52. After reset, TIME is equal to 0. The CLOCK1 statement enables the REAL TIME CLOCK. When the REAL TIME CLOCK is enabled, the SPECIAL FUNCTION OPERATOR, TIME will increment once every 5 milliseconds. The TIME operator uses TIMER0 and the interrupts associated with TIMER0 on the 8052AH. The unit of TIME is seconds and the appropriate XTAL value must be assigned to insure that the TIME operator is accurate.

When TIME is assigned a value with a LET statement (i.e. TIME = 100), only the integer portion of TIME will be changed.

EXAMPLE:

```
>CLOCK1          (enable REAL TIME CLOCK)

>CLOCK0          (disable REAL TIME CLOCK)

>PRINT TIME      (display TIME)
 3.315

>TIME = 0        (set TIME = 0)

>PRINT TIME      (display TIME)
 .315            (only the integer is changed)

The "fraction" portion of TIME can be changed by
manipulating the contents of internal memory
location 71 (47H). This is accomplished by a DBY(71)
statement. Note that each count in internal memory
location 71 (47H) represents 5 milliseconds of TIME.
Continuing with the EXAMPLE;

>DBY(71) = 0     (fraction of TIME = 0)

>PRINT TIME
 0

>DBY(71) = 3     (fraction of TIME = 3, 15 ms)

>PRINT TIME
 1.5 E-2
```

7.1 SPECIAL FUNCTION OPERATORS

The reason only the integer portion of TIME is changed when assigned a value is that it allows the user to generate accurate time intervals. For instance, let's say you want to create an accurate 12 hour clock. There are 43200 seconds in a 12 hour period, so an ONTIME 43200,[ln num] statement is used. Now, when the TIME interrupt occurs the statement TIME = 0 is executed, but the millisecond counter is not re-assigned a value so if interrupt latency happens to exceed 5 milliseconds, the clock will still remain accurate.

TIMER0

The TIMER0 operator is used to retrieve or assign a value to the 8052AH's special function registers TH0 and TL0. This operator treats the byte registers TH0 and TL0 as a 16-bit register pair. TH0 is the high byte and TL0 is the low byte. MCS BASIC-52 uses TH0 and TL0 to implement the REAL TIME CLOCK function. If the user does not implement the REAL TIME CLOCK function (i.e. does not use the statement CLOCK1) in the BASIC program TH0 and TL0 may be used in any manner suitable to the particular application.

TIMER1

The TIMER1 operator is used to retrieve or assign a value to the 8052AH's special function registers TH1 and TL1. This operator treats the byte registers TH1 and TL1 as a 16-bit register pair. TH1 is the high byte and TL1 is the low byte. MCS BASIC-52 uses TH1 and TL1 to implement the timings for the software serial port, the EPROM programming feature, and the PWM statement. If the user does not use any of these features TH1 and TL1 may be used in any manner suitable to the particular application.

TIMER2

The TIMER2 operator is used to retrieve or assign a value to the 8052AH's special function registers TH2 and TL2. This operator treats the byte registers TH2 and TL2 as a 16-bit register pair. TH2 is the high byte and TL2 is the low byte. MCS BASIC-52 uses TH2 and TL2 to generate the baud rate for the serial port. If the user does not use TIMER2 to clock the serial port, TH2 and TL2 may be used in any manner suitable to the particular application.

7.1 SPECIAL FUNCTION OPERATORS

XTAL

The XTAL operator tells MCS BASIC-52 what frequency the system is operating at. The XTAL operator is used by MCS BASIC-52 to calculate the REAL TIME CLOCK reload value, the PROM programming timing, and the software serial port baud rate generation. The XTAL value is expressed in Hz. So,

XTAL = 9000000

would set the XTAL value to 9 MHz.

7.2 EXAMPLES OF MANIPULATING SPECIAL FUNCTION VALUES

Using the logical operators available in MCS BASIC-52, it is possible to write to or read from any byte of the special function registers that MCS BASIC-52 treats as a register pair:

EXAMPLE:

```
WRITING TO THE HIGH BYTE
>TIMER0 = (TIMER0 .AND. 00FFH)+ INT(256*(USER BYTE))
```

EXAMPLE:

```
WRITING TO THE LOW BYTE
>TIMER0 = (TIMER0 .AND. 0FF00H) + (USER BYTE)
```

EXAMPLE:

```
READING HIGH BYTE
>PH0. INT(TIMER0/256)
```

EXAMPLE:

```
READING LOW BYTE
>PH0. TIMER0 .AND. 0FFH
```

TIMER1 can function as the baud rate generator for MCS BASIC-52. To assign TIMER1 as the baud rate generator, the following instructions must be executed:

```
>TMOD = 32       -       TIMER1 in auto reload mode
>TIMER1 = 256*(256-(65536-RCAP2)/12) - load TIMER1
>T2CON = 0       -       use TIMER1 as baud rate gen
```

This sequence of instructions can be executed in either the direct mode or as part of a program. When TIMER1 is used as the baud rate generator, TIMER2 can be used in anyway suitable to the application. The PROG, FPROG, LIST#, PRINT# and PWM commands/statements cannot be used when TIMER1 functions as the baud rate generator for the MCS BASIC-52 device. Certain crystals may not be able to use TIMER1 as the baud rate generator, especially at high (above 2400) baud rates.

7.3 SYSTEM CONTROL VALUES

The SYSTEM CONTROL VALUES determine or reveal how memory is allocated by MCS BASIC-52.

MTOP

After reset, MCS BASIC-52 sizes the external memory and assigns the last valid memory address to the SYSTEM CONTROL VALUE, MTOP. MCS BASIC-52 will not use any external RAM memory beyond the value assigned to MTOP. If the user wishes to allocate some external memory for an assembly language routine the LET statement can be used (e.g. MTOP = USER ADDRESS). If the user assigns a value to MTOP that is greater than the last valid memory address, a MEMORY ALLOCATION ERROR will be generated.

EXAMPLES:

```
>PRINT MTOP
 2047

>MTOP=2000

>PRINT MTOP
 2000
```

LEN

The SYSTEM CONTROL VALUE, LEN, tells the user how many bytes of memory the current selected program occupies. Obviously, LEN cannot be assigned a value, it can only be read. A NULL program (i.e. no program) will return a LEN of 1. The 1 represents the end of program file character.

FREE

The SYSTEM CONTROL VALUE, FREE, tells the user how many bytes of RAM memory are available to the user. When the current selected is in RAM memory, the following relationship will always hold true.

FREE = MTOP − LEN − 511

NOTE: Unlike some BASICS, MCS BASIC-52 does not require any "dummy" arguments for the SYSTEM CONTROL VALUES.

CHAPTER 8
Error Messages, Bells, Whistles, and Anomalies

8.1 ERROR MESSAGES

MCS BASIC-52 has a relatively sophisticated ERROR processor. When BASIC is in the RUN mode the generalized form of the ERROR message is as follows:

```
ERROR: XXX - IN LINE YYY

YYY BASIC STATEMENT
-----------X
```

Where XXX is the ERROR TYPE and YYY is the line number of the program in which the error occurred. A specific example is:

```
ERROR: BAD SYNTAX - IN LINE 10

10 PRINT 34*21*
--------------X
```

The X signifies approximately where the ERROR occurred in the line number. The specific location of the X may be off by one or two characters or expressions depending on the type of error and where the error occurred in the program. If an ERROR occurs in the COMMAND MODE only the ERROR TYPE will be printed out NOT the Line number. This makes sense, because there are no line numbers in the COMMAND MODE. The ERROR TYPES are as follows:

BAD SYNTAX

A BAD SYNTAX error means that either an invalid MCS BASIC-52 COMMAND, STATEMENT, or OPERATOR was entered and BASIC cannot process the entry. The user should check and make sure that everything was typed in correctly. In Version 1.1 of MCS BASIC-52 a BAD SYNTAX ERROR is also generated if the programmer attempts to use a reserved keyword as part of a variable.

BAD ARGUMENT

When the argument of an operator is not within the limits of the operator a BAD ARGUMENT ERROR will be generated. For instance, DBY(257) would generate a BAD ARGUMENT ERROR because the argument for the DBY operator is limited to the range 0 to 255. Similarly, XBY(5000H) = −1 would generate a BAD ARGUMENT ERROR because the value of the XBY operator is limited to the range 0 to 255.

8.1 ERROR MESSAGES

ARITH. UNDERFLOW

If the result of an arithmetic operation exceeds the lower limit of an MCS BASIC-52 floating point number, an ARITH. UNDERFLOW ERROR will occur. The smallest floating point number in MCS BASIC-52 is ± 1E−127. For instance, 1E−80/1E+80 would cause an ARITH. UNDERFLOW ERROR.

ARITH. OVERFLOW

If the result of an arithmetic operation exceeds the upper limit of an MCS BASIC-52 floating point number, an ARITH. OVERFLOW ERROR will occur. The largest floating point number in MCS BASIC-52 is ±.99999999E+127. For instance, 1E+70*1E+70 would cause an ARITH. OVERFLOW ERROR.

DIVIDE BY ZERO

A division by ZERO was attempted i.e. 12/0, will cause a DIVIDE BY ZERO ERROR.

ILLEGAL DIRECT (VERSION 1.0 ONLY)

Some statements, such as IF-THEN and DATA cannot be executed while the MCS BASIC-52 device is in the COMMAND MODE. If you attempt to execute one of these statements the message ERROR: ILLEGAL DIRECT will be printed to the console device. The ILLEGAL DIRECT ERROR is not trapped in Version 1.1 of MCS BASIC-52. ILLEGAL DIRECT ERRORS return a BAD SYNTAX ERROR in Version 1.1.

LINE TOO LONG (VERSION 1.0 ONLY)

If you type in a line that contains more than 73 characters the message ERROR: LINE TOO LONG will be printed to the console device. MCS BASIC-52's input buffer can only handle up to 73 characters.

NOTE

This error does not exist in Version 1.1. Instead the input buffer has been increased to 79 characters and MCS BASIC-52 will echo a bell character to the user terminal if too many characters are entered into the input buffer.

NO DATA

If a READ STATEMENT is executed and no DATA STATEMENT exists or all DATA has been read and a RESTORE instruction was not executed the message ERROR: NO DATA — IN LINE XXX will be printed to the console device.

8.1 ERROR MESSAGES

CAN'T CONTINUE

Program execution can be halted by either typing in a control-C to the console device or by executing a STOP STATEMENT. Normally, program execution can be resumed by typing in the CONT command. However, if the user edits the program after halting execution and then enters the CONT command, a CAN'T CONTINUE ERROR will be generated. A control-C must be typed during program execution or a STOP STATEMENT must be executed before the CONT command will work.

PROGRAMMING

If an error occurs while the MCS BASIC-52 device is programming an EPROM, a PROGRAMMING ERROR will be generated. An error encountered during programming destroys the EPROM FILE STRUCTURE, so the user cannot save any more programs on that particular EPROM once a PROGRAMMING ERROR occurs.

A-STACK

An A-STACK (ARGUMENT STACK) error occurs when the argument stack pointer is forced "out of bounds." This can happen if the user overflows the argument stack by PUSHing too many expressions onto the stack, or by attempting to POP data off the stack when no data is present.

C-STACK

A C-STACK (CONTROL STACK) error will occur if the control stack pointer is forced "out of bounds." 158 bytes of external memory are allocated for the control stack, FOR — NEXT loops require 17 bytes of control stack DO — UNTIL, DO — WHILE, and GOSUB require 3 bytes of control stack. This means that 9 nested FOR — NEXT loops is the maximum that MCS BASIC-52 can handle because 9 times 17 equals 153. If the user attempts to use more control stack than is available in MCS BASIC-52 a C-STACK error will be generated. In addition, C-STACK errors will occur if a RETURN is executed before a GOSUB, a WHILE or UNTIL before a DO, or a NEXT before a FOR.

8.1 ERROR MESSAGES

I-STACK

An I-STACK (INTERNAL STACK) error occurs when MCS BASIC-52 does not have enough stack space to evaluate an expression. Normally, I-STACK errors will not occur unless insufficient memory has been allocated to the 8052AH's stack pointer. Details of how to allocate memory to the stack pointer are covered in the ASSEMBLY LANGUAGE LINKAGE section of this manual.

ARRAY SIZE

If an array is dimensioned by a DIM statement and then you attempt to access a variable that is outside of the dimensioned bounds, an ARRAY SIZE error will be generated.

EXAMPLE:

```
>DIM A(10)
>PRINT A(11)

ERROR: ARRAY SIZE
READY
```

MEMORY ALLOCATION

MEMORY ALLOCATION ERRORS are generated when user attempts to access STRINGS that are "outside" the defined string limits. Additionally, if the SYSTEM CONTROL VALUE, MTOP is assigned a value that does not contain any RAM memory, a MEMORY ALLOCATION ERROR will occur.

8.2 DISABLING CONTROL-C

In some applications, it may be desirable or even a requirement that program execution not accidentally be halted. Under "normal" operation the execution of any MCS BASIC-52 program can be terminated by typing a "control-C" on the console device. However, it is possible to disable the "control-C" break function in MCS BASIC-52. This is accomplished by setting BIT 48 (30H) to a one. BIT 48 is located in internal memory location 38.0 (26.0H). This BIT may be set by executing the following statement in an MCS BASIC-52 program:

DBY(38) = DBY(38).OR.01H

Once this BIT is set to a one, the control-C break function, for both LIST and RUN operations will be disabled. The user has the option to create a custom break character or string of characters by using the GET operator. The following is an example of how to implement a custom break character:

EXAMPLE:

```
>10 STRING 100,10: A=1: REM INITIALIZE STRINGS
>20 $(1) = "BREAK" : REM "BREAK" IS THE PASSWORD
>30 DBY(38)=DBY(38).OR.1 : REM DISABLE CONTROL-C
>40 FOR I=1 TO 1000 : REM DUMMY LOOP
>50 J=SIN(I)
>60 K=GET : IF K<>0 THEN 100 ELSE NEXT I
>70 END
>100 IF K=ASC($(1),A) THEN A=A+1 ELSE A=1
>110 REM TEST FOR MATCH
>120 IF A=1 THEN NEXT I
>130 IF A=6 THEN 200 ELSE NEXT I
>140 END
>200 PRINT "BREAK"
>210 DBY(38)=DBY(38).AND.0FEH : REM ENABLE CONTROL-C
```

In this example, typing the word BREAK will stop program execution. In other words, BREAK is a password.

8.3 IMPLEMENTING "FAKE DMA"

The MCS BASIC-52 device does not contain any hardware mechanism that supports Direct Memory Access (DMA). However, the DMA function is supported in software by MCS BASIC-52. During DMA operation MCS BASIC-52 guarantees that no external memory access will be performed. To enable the DMA function, the following must be performed:

1) BIT 49, which is located in internal memory location 38.1 (26.1H) must be set to a one. This can be accomplished in BASIC by using the statement — DBY(38) = DBY(38).OR.02H

2) BIT 0 and BIT 7 of the SPECIAL FUNCTION REGISTER, IE (Interrupt enable) must be set to a one. This can be accomplished in BASIC by using the statement — IE = IE.OR.81H

After the three BITS mentioned above are set to a one, external interrupt zero ($\overline{INT0}$) acts as a DMA input pin. $\overline{INT0}$ is pin 12 on the 8052AH. Whenever $\overline{INT0}$ is pulled low (to a logical zero state), the MCS BASIC-52 device will enter the DMA mode and no accesses will be made to external memory. To acknowledge that MCS BASIC-52 has entered the DMA mode, MCS BASIC-52 outputs a zero on pin 7 (P1.6). In essence, PORT 1.6 is the $\overline{DMA\ ACK}$ pin of the MCS BASIC-52 device. In most applications, this pin would be used to disable three-state buffers that would be placed on PORT2, PORT0, and the address latch of the MCS BASIC-52 system. After the user pulls the $\overline{INT0}$ pin high, MCS BASIC-52 will output a one on P1.6 and normal program execution will continue. During this "fake DMA" cycle, the MCS BASIC-52 program does nothing except wait for the $\overline{INT0}$ pin to be pulled high. So, program execution is halted.

It should be noted that although this "fake DMA" operation does provide the same functionality as a normal DMA hardware mechanism, it also takes substantially longer for the normal DMA REQUEST — DMA ACKNOWLEDGE cycle to be performed. That is because MCS BASIC-52 uses interrupts to implement the DMA operation, instead of dedicated hardware. As a general rule, cycle stealing DMA is not an option with MCS BASIC-52's "fake" DMA. Only "burst mode" DMA cycles can be implemented without a significant time penalty. When "fake DMA" is implemented, the user must provide three-state buffers on the PORT2, PORT0, and the address latch of the MCS BASIC-52 system.

8.4 RUN TRAP OPTION (Version 1.1 Only)

Version 1.1 of MCS BASIC-52 permits the user to trap the interpreter in the RUN MODE. This option is evoked by putting a 34H (52D) in external data memory location 5EH (94D). After a 34H (52D) is placed in external data memory location 5EH (94D) the MCS BASIC-52 interpreter will be trapped in the RUN mode forever or until the contents of external data memory location is changed to something other than 34H (52D). If no program is present when a 34H (52D) is placed in location 5EH (94D), MCS BASIC-52 will print the READY message forever and it will be time to RESET the device. The RUN TRAP option can be employed with the other RESET options to permit the user to execute a program from RAM on a RESET or power-up condition when some type of battery back-up memory scheme is employed.

8.5 ANOMALIES

Most dictionaries define an anomaly as a deviation from the normal or common order or as an irregularity. Anomalies to an extreme become "BUGS" or something that is wrong with the program. Like all programs, MCS BASIC-52 contains some anomalies, hopefully, no bugs. The purpose of mentioning the known anomalies here is that it may save the programmer some time, should strange things happen during program execution. The known anomalies deal mainly with the way MCS BASIC-52 compacts or tokenizes the BASIC program. The known anomalies and cautions are as follows:

1) When using the variable H after a line number, make sure you put a space between the line number and the H, or else BASIC will assume that the line number is a HEX number.

EXAMPLES:

```
>20H=10   (WRONG)          >20 H=10    (RIGHT)
>LIST                      >LIST
32    =10                  20     H=10
```

2) When using the variable I before an ELSE statement, make sure you put a space between the I and the ELSE statement, or else BASIC will assume that the IE portion of IELSE is the special function operator IE.

EXAMPLES:

```
>20 IF I>10 THEN PRINT IELSE 100
>LIST
20     IF I>10 THEN  PRINT IELSE 100  (WRONG)

>20 IF I>10 THEN PRINT I ELSE 100
>LIST
20     IF I>10 THEN  PRINT I ELSE 100 (RIGHT)
```

3) A Space character may not be placed inside the ASC( ) operator. In other words, a statement like PRINT ASC( ) will yield a BAD SYNTAX ERROR. Spaces may be placed in strings however, so a statement like LET $(1) = "HELLO, HOW ARE YOU" will work properly. The reason ASC( ) yields an error is because MCS BASIC-52 eliminates all spaces when a line is processed, so ASC( ) will be stored as ASC( ) and MCS BASIC-52 interprets this as an error.

CHAPTER 9
Assembly Language Linkage

9.1 OVERVIEW

NOTE: This section assumes that the designer has an understanding of the architecture and assembly language of the MCS-51 Microcontroller family!!!

MCS BASIC-52 contains a complete library of routines that can easily be accessed with assembly language CALL instructions. The advantage of using assembly language is that it offers a significant improvement in execution speed relative to interpreted BASIC. In order to successfully interface MCS BASIC-52 with an assembly language program, the software designer must be aware of a few simple facts.

READ THIS CAREFULLY!!!

1. MCS BASIC-52 uses REGISTER BANKS 0, 1, and 2 (RB0, RB1, and RB2). REGISTER BANK 3 (RB3) is never used except during a PGM statement. RB3 is designated the USER REGISTER BANK and the users can do whatever they want to with REGISTER BANK 3 (RB3) and MCS BASIC-52 will never alter the contents of this REGISTER BANK except during the execution of a PGM statement. The contents of REGISTER BANK 3 (RB3) can be changed by executing a DBY ([expr]) = [0 to 255] statement. Where the [expr] evaluates to a number between 24 (18H) and 31 (1FH) inclusive. In addition, INTERNAL MEMORY LOCATIONS 32 (20H) and 33 (21H) are also NEVER used by MCS BASIC-52. These two BIT and/or BYTE addressable locations are specifically reserved for assembly language programs.

2. MCS BASIC-52 uses REGISTER BANK 0 (RB0) as the WORKING REGISTER FILE. Whenever assembly language is used to access MCS BASIC-52's routines, the WORKING REGISTER FILE, REGISTER BANK 0 (RB0) MUST BE SELECTED!!! This means that the USER MUST MAKE SURE THAT REGISTER BANK 0 (RB0) IS SELECTED BEFORE CALLING ANY OF MCS BASIC-52's ROUTINES. This is done simply by setting BITS 3 and 4 in the PSW equal to ZERO. If this is not done, MCS BASIC-52 will "KICK OUT" the USER and NO operation will be performed. When an ASSEMBLY LANGUAGE program is accessed by using the MCS BASIC-52's CALL instruction, REGISTER BANK 0 (RB0) will always be selected. So unless the user selects REGISTER BANK 3 (RB3) in assembly language, it is NOT NECESSARY to change the designated REGISTER BANK.

3. ALWAYS ASSUME THAT MCS BASIC-52 DESTROYS THE CONTENTS OF THE WORKING REGISTER FILE AND THE DPTR, UNLESS OTHERWISE STATED IN FOLLOWING DOCUMENTATION.

4. Certain routines in MCS BASIC-52 require that REGISTERS be initialized BEFORE the user CALLS that specific ROUTINE. These registers are ALWAYS in the WORKING REGISTER FILE, REGISTER BANK 0 (RB0).

5. Certain routines in MCS BASIC-52 return the result of an operation in a register or registers. The result registers are ALWAYS in the WORKING REGISTER FILE, REGISTER BANK 0, (RB0).

9.1 OVERVIEW

READ THIS CAREFULLY!!!

6. MCS BASIC-52 loads the INTERNAL STACK POINTER (SPECIAL FUNCTION REGISTER- SP) with the value that is in INTERNAL MEMORY LOCATION 62 (3EH). MCS BASIC-52 initializes INTERNAL MEMORY LOCATION 62 (3EH) by writing a 77 (4DH) to this location after a hardware RESET. MCS BASIC-52 does NOT use any memory beyond 77 (4DH) for anything EXCEPT STACK SPACE. If the user wants to ALLOCATE some additional internal memory for their application, this is done by changing the contents of INTERNAL MEMORY LOCATION 62 (3EH) to a value that is GREATER than 77 (4DH). This will allocate the INTERNAL MEMORY LOCATIONS from 77 (4DH) to the value that is placed in INTERNAL MEMORY LOCATION 62 (3EH) to the user. As a guideline, it is a good idea to allow at least 48 (30H) bytes of stack space for MCS BASIC-52. The bad news about reducing the stack space is that it will reduce the amount of nested parentheses that MCS BASIC-52 can evaluate in an expression [expr]. This will either cause a I-STACK ERROR or will cause a fatal CPU "crash." Use caution and DON'T allocate more memory than you need.

EXAMPLE OF THE EFFECTS OF ALTERING THE STACK ALLOCATION:

```
                          COMMENTS

>PRINT DBY(62)            AFTER RESET INTERNAL MEMORY LOCATION 62
 77                       CONTAINS A 77

>PRINT (1*(2*(3)))        BASIC HAS NO PROBLEM EVALUATING 3 LEVELS
 6                        OF NESTED PARENTHESIS

>DBY(62)=230              NOW ALLOCATE 255-230 = 25 BYTES OF STACK
                          SPACE TO BASIC, REMEMBER, THE STACK ON
>PRINT (1*(2*(3)))        THE 8052AH GROWS "UP"

ERROR: I-STACK            BASIC CANNOT EVALUATE THIS EXPRESSION
READY                     BECAUSE IT DOES NOT HAVE ENOUGH STACK
>DBY(62)=210              NOW ALLOCATE 255-210 = 45 BYTES OF STACK
                          SPACE TO BASIC
>PRINT (1*(2*(3)))        THE I-STACK ERROR GOES AWAY
 6
```

7. Throughout this section a 16-BIT REGISTER PAIR is designated-Rx:Ry, where Rx is the most significant byte and Ry is the least significant byte.

EXAMPLE:

```
R3:R1 - R3=MOST SIGNIFICANT BYTE, R1=LEAST SIGNIFICANT BYTE
```

9.2 GENERAL PURPOSE ROUTINES

Accessing MCS BASIC-52 routines with assembly language is easy. The user just loads the ACCUMULATOR with a specific value and CALLS LOCATION 48 (30H). The value placed in the ACCUMULATOR determines what operation will be performed. Unless otherwise stated, the CONTENTS of the DPTR and REGISTER BANK 0 (RB0) will ALWAYS be altered when calling these routines. The generalized form for accessing MCS BASIC-52's routines is as follows:

```
ANL   PSW,#11100111B   ; make sure
                         RB0 is
                         selected
MOV   A,#OPBYTE        ; load the
                         instruction
CALL  30H              ; execute the
                         instruction
```

The value of OPBYTE determines what operation will be performed. The following operations can be performed:

OPBYTE = 0 (00H) RETURN TO COMMAND MODE

This instruction causes MCS BASIC-52 to enter the COMMAND MODE. Control of the CPU is handed back to the MCS BASIC-52 interpreter and BASIC will respond by outputting a READY and a PROMPT character (>).

OPBYTE = 1 (01H) POP ARGUMENT STACK AND PUT VALUE IN R3:R1

This instruction converts the value that is on the ARGUMENT STACK to a 16 BIT BINARY INTEGER and returns the BINARY INTEGER in registers R3 (high byte) and R1 (low byte) of REGISTER BANK 0 (RB0). The ARGUMENT STACK gets popped after this instruction is executed. If the value on the ARGUMENT STACK cannot be represented by a 16-BIT BINARY NUMBER (i.e. it is NOT between 0 and 65535 (0FFFFH) inclusive), BASIC WILL TRAP THE ERROR and print a BAD ARGUMENT ERROR MESSAGE. The BINARY VALUE returned is TRUNCATED, NOT ROUNDED.

EXAMPLE:

```
BASIC PROGRAM -                 10  PUSH  260
                                20  CALL  5000H

ASSEMBLY LANGUAGE PROGRAM -     ORG 5000H
                                MOV  A,#01H     ; load opbyte
                                CALL   30H      ; RB0 still selected
                                ;
                                ; at this point R3 (of RB0) = 01H
                                ; and R1 (of RB0) = 04H
                                ; so, R3:R1 = 260, which was the value
                                ; that was placed on the ARGUMENT STACK
```

9.2 GENERAL PURPOSE ROUTINES

COMMENTS ON THE NEXT TWO INSTRUCTIONS:

The next two instructions permit the user to transfer floating point numbers between an assembly language program and MCS BASIC-52. The user provides the address where a floating point number is stored or will be stored in a 16-bit REGISTER PAIR. Depending on the operation requested, the floating point numbers are either PUSHED ON or POPPED OFF MCS BASIC-52's ARGUMENT STACK. This instruction permits the user to keep track of variables in assembly language and bypass the relatively slow procedure BASIC must follow.

As mentioned earlier, when a floating point number is PUSHED onto the ARGUMENT STACK, the ARGUMENT STACK POINTER is decremented by a count of 6. This is because a floating point number requires 6 bytes of RAM storage. Although it may seem confusing to the novice, the LAST value placed on the ARGUMENT STACK is referred to as the value on the TOP of the ARGUMENT STACK, even though it is on the BOTTOM of the STACK relative to the sequential numbering of memory addresses. No one knows why this is so.

The ARGUMENT STACK resides in EXTERNAL RAM MEMORY LOCATIONS 301 (12DH) through 510 (1FEH). The lower BYTE of the ARGUMENT STACK POINTER resides in INTERNAL MEMORY LOCATION 9 (09H). MCS BASIC-52 always assumes that the upper BYTE (higher order address) of the ARGUMENT STACK POINTER is 1 (01H). The software designer can use this information, along with the next two instructions to perform operations like copying the stack.

OPBYTE = 2 (02H) PUSH THE FLOATING POINT NUMBER ADDRESSED BY REGISTER PAIR R2:R0 ONTO THE ARGUMENT STACK.

R2 and R0 (in REGISTER BANK 0, RB0) contain the ADDRESS (R2 = high byte, R0 = low byte) of the location where the floating point number is stored. After this instruction is executed the floating point number that the REGISTER PAIR R2:R0 points to is PUSHED onto the TOP of the ARGUMENT STACK. The ARGUMENT STACK POINTER automatically gets DECREMENTED, by a count of 6, when the value is placed on the stack. A floating point number in MCS BASIC-52 requires 6 BYTES of RAM storage. The register Pair R2:R0 points to the MOST SIGNIFICANT BYTE of the floating point number and is DECREMENTED BY 6 after the CALL instruction is executed. So, if R2:R0 = 7F18H before this instruction was executed, it would equal 7F12H after this instruction was executed.

9.2 GENERAL PURPOSE ROUTINES

OPBYTE = 3 (03H) POP THE ARGUMENT STACK AND SAVE THE FLOATING POINT NUMBER IN THE LOCATION ADDRESSED BY R3:R1.

The TOP of the ARGUMENT STACK is moved to the location pointed to by the REGISTER PAIR R3:R1(R3 = high byte, R1 = low byte, in REGISTER BANK 0, RB0). The ARGUMENT STACK POINTER is automatically INCREMENTED BY 6. Just as in the previous PUSH instruction, the REGISTER PAIR R3:R1 points to the MOST SIGNIFICANT BYTE of the floating point number and is DECREMENTED BY 6 after the CALL instruction is executed.

EXAMPLE OF USER PUSH AND POP:

```
BASIC PROGRAM:      >5  REM PUT 100 AND 200 ON THE ARGUMENT STACK
                    >10 PUSH 100,200
                    >15 REM CALL THE USER ROUTINE TO SAVE NUMBERS
                    >20 CALL 5000H
                    >25 REM CLEAR THE STACK
                    >30 CLEARS
                    >35 REM USE ASM TO PUT NUMBERS BACK ON STACK
                    >40 CALL 5010H
                    >50 POP A,B
                    >60 PRINT A,B
                    >RUN

                     100 200

                    READY

ASM PROGRAM:        ORG   5000H
                    MOV   R3,#HIGH USER_SAVE   ;LOAD POINTERS TO WHERE
                    MOV   R1,#LOW USER_SAVE    ;NUMBERS WILL BE SAVED
                    MOV   A,#03H               ;LOAD OPBYTE
                    CALL  30H                  ;SAVE ONE NUMBER
                    MOV   A,#03H               ;LOAD OPBYTE AGAIN
                    CALL  30H                  ;SAVE ANOTHER NUMBER
                    RET                        ;BACK TO BASIC
                    ;
                    ORG   5010H
                    MOV   R2,#HIGH USER_SAVE   ;LOAD ADDRESS WHERE
                    MOV   R0,#LOW USER_SAVE    ;NUMBERS ARE STORED
                    MOV   A,#02H               ;LOAD OPBYTE
                    CALL  30H                  ;PUT ONE NUMBER ON STACK
                    MOV   A,#02H               ;LOAD OPBYTE
                    CALL  30H                  ;NEXT NUMBER ON STACK
                    RET                        ;BACK TO BASIC
```

9.2 GENERAL PURPOSE ROUTINES

OPBYTE = 4 (04H) PROGRAM A PROM USING R3:R1 AS THE SOURCE ADDRESS POINTER, R2:R0 AS THE DESTINATION (PROM) ADDRESS POINTER, AND R7:R6 AS THE BYTE COUNTER.

This instruction assumes that the DATA to be programmed into a PROM is stored in external memory and that the REGISTER PAIR R3:R1 (in RB0) contains the address of this external memory. REGISTER PAIR R7:R6 contains the total number of bytes that are to be programmed. The PROM is programmed sequentially and everytime a byte is programmed the REGISTER PAIR R7:R6 is decremented and the REGISTER PAIRS R3:R1 and R2:R0 are incremented. Programming continues until R7:R6 equals ZERO. The REGISTER PAIR R2:R0 must be initialized with the desired ADDRESS of the EPROM to be programmed MINUS 1. This may sound strange, but that is the way it works. So, if you wanted to program an EPROM starting at address 9000H, with the data stored in address 0D00H and you wanted to program 500 BYTES, then the registers would be set up as follows: R2:R0 = 8FFFH, R3:R1 = 0D00H, and R7:R6 = 01F4H (500 decimal). You would then put a 4 (04H) in the ACC and CALL location 30H.

NOTE: In Version 1.0, if an ERROR OCCURS DURING PROGRAMMING, MCS BASIC-52 WILL TRAP THE ERROR AND ENTER THE COMMAND MODE. The user cannot handle errors that occur during the EPROM programming operation!!!!

In Version 1.1, programming errors will only be trapped if the MCS BASIC-52 device is in the COMMAND MODE. If the MCS BASIC-52 device is in the run mode, control will be passed back to the user. The user must then examine registers R6 and R7. If R6 = R7 = 0, then the programming operation was successfully completed, if these registers do not equal zero then registers R2:R0 contain the address of the EPROM location that failed to program. This feature in Version 1.1 permits the user to program EPROMS in embedded applications and manage errors, should they occur in the programming process, without trapping to the command mode.

In addition to setting up the pointers previously described, the user must also initialize the INTERNAL MEMORY locations that control the width of the programming pulse. This gives the user complete control over this critical prom programming parameter. The internal memory locations that must be initialized with this information are 64 (40H) and 65 (41H). These locations are treated as a 16 bit register pair with location 64 (40H) designated as the most significant byte and location 65 (41H) as the least significant byte. Locations 64 (40H) and 65 (41H) are loaded into TH1 (TIMER 1 high byte) and TL1 (TIMER 1 low byte) respectively. The width of the programming pulse, in microseconds is determined by the following equation:

WIDTH = (65536 — 256*DBY(64) + DBY(65))*12/XTAL microseconds

conversely;

DBY(64):DBY(65) = 65536 — WIDTH*XTAL/12

The proper values for the "normal" 50 millisecond programming algorithm and the 1 millisecond "INTELligent" algorithm are calculated and stored by MCS BASIC-52 in external memory locations 296:297 (128H:129H) and 298:299 (12AH:12BH) respectively. If the user wants to use the pre-calculated values the statements DBY(64) = XBY(296) and DBY(65) = XBY(297) may be used to initialize the prom programming width for the normal algorithm and the statements DBY(64) = XBY(298) and DBY(65) = XBY(299) can be used to initialize for the INTELligent algorithm.

9.2 GENERAL PURPOSE ROUTINES

To select the "INTELLIGENT" EPROM PROGRAMMING algorithm the directly addressable BIT 51 (33H) MUST be set to 1 before the EPROM PROGRAMMING routine is called. The "STANDARD" 50 ms EPROM PROGRAMMING algorithm is selected by CLEARING BIT 51 (33H) (i.e. BIT 51 = 0) before calling the EPROM PROGRAMMING routine. The directly addressable BIT 51 is located in internal memory location 38.3 (26 3H) (BIT 3 of BYTE 38 (26H) in internal memory). This BIT can be SET or CLEARED by the BASIC STATEMENTS DBY(38) = DBY(38).OR.08H to SET and DBY(38) = DBY(38).AND.0F7H to CLEAR. Of course, the user can set or clear this bit in assembly language with a SETB 51 or CLR 51 instruction.

The user must also turn on the EPROM PROGRAMMING voltage BEFORE calling the EPROM PROGRAMMING routine. This is done by CLEARING BIT P1.5, the fifth BIT on PORT 1. This too can be done in BASIC with a PORT1 = PORT1.AND.0DFH instruction or in assembly language with a CLR P1.5 instruction. The user must also set this bit when the PROM PROGRAMMING procedure is complete.

This instruction assumes that the hardware surrounding the MCS BASIC-52 device is the same as the suggestions in the EPROM PROGRAMMING chapter of this manual.

9.2 GENERAL PURPOSE ROUTINES

OPBYTE = 5 (05H) INPUT A STRING OF CHARACTERS AND STORE IN THE BASIC INPUT BUFFER.

This instruction inputs a line of text from the console device and saves the information in the MCS BASIC-52's input buffer. MCS BASIC-52's input buffer begins at EXTERNAL MEMORY LOCATION 7 (0007H). All of the line editing features available in MCS BASIC-52 are implemented in this instruction. If a control-C is typed during the input process, MCS BASIC-52 will trap back into the command mode. A carriage return (cr) terminates the input procedure.

OPBYTE = 6 (06H) OUTPUT THE STRING OF CHARACTERS POINTED TO BY THE REGISTER PAIR R3:R1 TO THE CONSOLE DEVICE.

This instruction is used to OUTPUT a string of characters to the console device. R3:R1 contains the initial address of this string. The string can either be stored in PROGRAM MEMORY or EXTERNAL DATA MEMORY. If BIT 52 (34H) (which is BIT 4 of internal RAM location 38 (26H)) is set, the output will be from PROGRAM MEMORY. If BIT 52 is cleared, the output will be from EXTERNAL DATA MEMORY. The DATA stored in MEMORY is sent out to the console device one byte at a time and the memory pointer is incremented. The output is stopped when a termination character is read. The termination character for PROGRAM MEMORY and EXTERNAL DATA MEMORY are different. The termination character for EXTERNAL DATA MEMORY is a (cr) 0DH. The termination character for PROGRAM MEMORY is a " or 22H.

OPBYTE = 7 (07H) OUTPUT A CARRIAGE RETURN-LINE FEED SEQUENCE TO THE CONSOLE DEVICE.

Enough said.

OPBYTE = 128 (80H) OUTPUT THE CHARACTER IN R5 (REGISTER BANK 0) TO THE CONSOLE DEVICE.

This routine takes the character that is in R5 (register bank 0) and directs it to the console device. Any console device may be selected (i.e. U0 or U1 or the software serial port).

9.2 GENERAL PURPOSE ROUTINES

OPBYTE = 144 (90H) OUTPUT THE NUMBER ON THE TOP OF ARGUMENT STACK TO THE CONSOLE DEVICE.

The floating point number that is on the top of the argument stack is outputted to the console device. The FORMAT is determined by the USING statement. The argument stack is POPPED after the output operation.

OPBYTE = 154 (9AH) THE 16 BIT NUMBER REPRESENTED BY REGISTER PAIR R2:R0 IS PUSHED ON THE ARGUMENT STACK.

This instruction converts the 16 bit register pair R2:R0 to a floating point number and pushes this number onto the argument stack. This instruction is the converse of the OPBYTE = 1 instruction.

9.3 UNARY OPERATORS

The next group of instructions perform an operation on the number that is on the TOP of the ARGUMENT STACK. If the TOP of the ARGUMENT STACK is represented by the symbol [TOS], then the following instructions would take the general form:

[TOS] < OP [TOS]

Where OP is one of the following operators:

OPBYTE = 24 (18H) — ABSOLUTE VALUE

[TOS] < ABS([TOS]). The [TOS] is replaced by the absolute value of [TOS].

OPBYTE = 25 (19H) — INTEGER

[TOS] < INT([TOS]). The [TOS] is replaced by the integer portion of [TOS].

OPBYTE = 26 (1AH) — SIGN

[TOS] < SGN([TOS]). If [TOS] > 0 then [TOS] = 1, if [TOS] = 0 then [TOS] = 0, and if [TOS] < 0 then [TOS] = −1.

OPBYTE = 27 (1BH) — ONE'S COMPLEMENT

[TOS] < NOT([TOS]). [TOS] must be a valid integer.

OPBYTE = 28 (1CH) — COSINE OPERATOR

[TOS] < COS([TOS]). [TOS] must be between ±200000.

OPBYTE = 29 (1DH) — TANGENT OPERATOR

[TOS] < TAN([TOS]). [TOS] must be between ±200000 and [TOS] cannot equal PI/2, 3*PI/2, 5*PI/2, ... (2*N + 1)*PI/2.

9.3 UNARY OPERATORS

OPBYTE = 30 (1EH) — SINE OPERATOR

[TOS] < SIN([TOS]). [TOS] must be between ±200000.

OPBYTE = 31 (1FH) — SQUARE ROOT

[TOS] < SQR ([TOS]). [TOS] must be >= 0.

OPBYTE = 32 (20H) — CBY OPERATOR

[TOS] < CBY ([TOS]). [TOS] must be a valid integer.

OPBYTE = 33 (21H) — E TO THE [TOS] OPERATOR

[TOS] < e(2.7182818)**[TOS]. e is raised to the [TOS] power.

OPBYTE = 34 (22H) — ATN OPERATOR

[TOS] < ATN([TOS]). Arctangent, the value returned is between ± PI/2.

OPBYTE = 35 (23H) — LOG OPERATOR (natural LOG)

[TOS] < LOG([TOS]) — [TOS] must be > 0.

OPBYTE = 36 (24H) — DBY OPERATOR

[TOS] < DBY([TOS]). [TOS] must be between 0 and 255 inclusive.

OPBYTE = 37 (25H) — XBY OPERATOR

[TOS] < XBY([TOS]). [TOS] must be a valid integer.

9.4 SPECIAL OPERATORS

The next group of instructions place a value on the stack. The value placed on the stack is as follows:

OPBYTE = 38 (26H) — PI

[TOS] = PI. PI (3.1415926) is placed on the [TOS].

OPBYTE = 39 (27H) — RND

[TOS] = RND. A random number is placed on the [TOS].

OPBYTE = 40 (28H) — GET

[TOS] = GET. The value of the SPECIAL FUNCTION OPERATOR, GET is put on the [TOS].

OPBYTE = 41 (29H) — FREE

[TOS] = FREE. The value of the SYSTEM CONTROL VALUE, FREE is put on the [TOS].

OPBYTE = 42 (2AH) — LEN

[TOS] = LEN. The value of the SYSTEM CONTROL VALUE, LEN is put on the [TOS].

OPBYTE = 43 (2BH) — XTAL

[TOS] = XTAL. The value of the SPECIAL FUNCTION OPERATOR, XTAL is put on the [TOS].

OPBYTE = 44 (2CH) — MTOP

[TOS] = MTOP. The value of the SYSTEM CONTROL VALUE, MTOP is put on the [TOS].

9.4 SPECIAL OPERATORS

OPBYTE = 45 (2DH) — TIME

[TOS] = TIME. The value of the SPECIAL FUNCTION OPERATOR, TIME is put on the [TOS].

OPBYTE = 46 (2EH) — IE

[TOS] = IE. The value of the IE register is put on the [TOS].

OPBYTE = 47 (2FH) — IP

[TOS] = IP. The value of the IP register is put on the [TOS].

OPBYTE = 48 (30H) — TIMER0

[TOS] = TIMER0. The value of TIMER0 (TH0:TL0) is put on the [TOS].

OPBYTE = 49 (31H) — TIMER1

[TOS] = TIMER1. The value of TIMER1 (TH1:TL1) is put on the [TOS].

OPBYTE = 50 (32H) — TIMER2

[TOS] = TIMER2. The value of TIMER2 (TH2:TL2) is put on the [TOS].

OPBYTE = 51 (33H) — T2CON

[TOS] = T2CON. The value of the T2CON register is put on the [TOS].

OPBYTE = 52 (34H) — TCON

[TOS] = TCON. The value of the TCON register is put on the [TOS].

9.4 SPECIAL OPERATORS

OPBYTE = 53 (35H) — TMOD

[TOS] = TMOD. The value of the TMOD register is put on the [TOS].

OPBYTE = 54 (36H) — RCAP2

[TOS] = RCAP2. The value of the RCAP2 registers (RCAP2H:RCAP2L) is put on the [TOS].

OPBYTE = 55 (37H) — PORT1

[TOS] = PORT1. The value of the PORT1 (P1) pins is placed on the [TOS].

OPBYTE = 56 (38H) — PCON

[TOS] = PCON. The value of the PCON register is put on the [TOS].

9.5 DUAL OPERAND OPERATORS

The next group of instructions assume that TWO values are on the ARGUMENT STACK. If number on the TOP of the ARGUMENT STACK is represented by the symbol [TOS] and the number NEXT to TOP of the ARGUMENT STACK is represented by the symbol [NxTOS] and the ARGUMENT STACK POINTER is represented by the symbol AGSP, then the following instructions would take the general form:

```
  TEMP1 = [TOS]
  TEMP2 = [NxTOS]
   AGSP  < AGSP + 6
RESULT = TEMP2 OP TEMP1
  [TOS] = RESULT
```

Where OP is one of the following operators to be described. NOTE that the group of instructions ALWAYS POP the ARGUMENT STACK by one FLOATING POINT NUMBER SIZE (i.e. 6 BYTES).

ERRORS can be handled in two different ways with the ADD, SUBTRACT, MULTIPLY, and DIVIDE routines. One option is to let MCS BASIC-52 trap ERRORS, should they occur during the operation. With this option MCS BASIC-52 will print the appropriate error message to the console device. The other option passes a STATUS CODE to the user. After the operation the Accumulator contains the status code information. The Status information is as follows:

ACC.0 — ARITHMETIC UNDERFLOW
ACC.1 — ARITHMETIC OVERFLOW
ACC.2 — RESULT WAS ZERO (not an error, just a condition)
ACC.3 — DIVIDE BY ZERO
ACC.4 — NOT USED, ZERO RETURNED
ACC.5 — NOT USED, ZERO RETURNED
ACC.6 — NOT USED, ZERO RETURNED
ACC.7 — NOT USED, ZERO RETURNED

If an ARITH. OVERFLOW or a DIVIDE BY ZERO ERROR occurs and the user is handling the error condition, the floating point processor will return a result of ± 99999999E + 127 to the argument stack. The user can do what they want to with this result (i.e. use it or waste it). An ARITH. UNDERFLOW ERROR will return to the argument stack a result of 0 (zero).

9.5 DUAL OPERAND OPERATORS

MCS BASIC-52 can perform the following DUAL OPERAND OPERATIONS:

OPBYTE = 9 (09H) EXPONENTIATION — The [NxTOS] value is raised to the [TOS] power. RESULT = [NxTOS] ** [TOS]. NOTE — [TOS] MUST BE LESS THAN 256.

OPBYTE = 10 (0AH) MULTIPLY

RESULT = [NxTOS] * [TOS]. If an ERROR occurs during this operation (i.e. ARITH. OVERFLOW or UNDERFLOW) MCS BASIC-52 will trap the error and print the error message to the console device.

OPBYTE = 136 (88H) MULTIPLY

RESULT = [NxTOS] * [TOS]. If an error occurs during this operation, the status byte previously discussed will be returned to the user.

OPBYTE = 11 (0BH) ADD

RESULT = [NxTOS] + [TOS]. BASIC handles errors.

OPBYTE = 130 (82H) ADD

RESULT = [NxTOS] + [TOS]. User handles errors.

OPBYTE = 12 (0CH) DIVIDE

RESULT = [NxTOS] / [TOS]. BASIC handles errors.

OPBYTE = 138 (8AH) DIVIDE

RESULT = [NxTOS] / [TOS]. User handles errors.

OPBYTE = 13 (0DH) SUBTRACT

RESULT = [NxTOS] − [TOS]. BASIC handles errors.

9.5 DUAL OPERAND OPERATORS

OPBYTE = 132 (84H) SUBTRACT

RESULT = [NxTOS] − [TOS]. User handles errors.

OPBYTE = 14 (0EH) EXCLUSIVE OR

RESULT = [NxTOS] XOR [TOS], both values must be GREATER THAN OR EQUAL TO ZERO and LESS THAN OR EQUAL TO 65535 (0FFFFH).

OPBYTE = 15 (0FH) LOGICAL AND

RESULT = [NxTOS] and [TOS], both values must be GREATER THAN OR EQUAL TO ZERO and LESS THAN OR EQUAL TO 65535 (0FFFFH).

OPBYTE = 16 (10H) LOGICAL OR

RESULT = [NxTOS] OR [TOS], both values must be GREATER THAN OR EQUAL TO ZERO and LESS THAN OR EQUAL TO 65535 (0FFFFH).

OPBYTE = 18 (12H) TEST FOR EQUALITY

IF [NxTOS] = [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

OPBYTE = 19 (13H) TEST FOR GREATER THAN OR EQUAL

IF [NxTOS] >= [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

OPBYTE = 20 (14H) TEST FOR LESS THAN OR EQUAL

IF [NxTOS] <= [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

9.5 DUAL OPERAND OPERATORS

OPBYTE = 21 (15H) TEST FOR NOT EQUAL

IF [NxTOS] <> [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

OPBYTE = 22 (16H) TEST FOR LESS THAN

IF [NxTOS] < [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

OPBYTE = 23 (17H) TEST FOR GREATER THAN

IF [NxTOS] > [TOS] then, RESULT = 65535 (0FFFFH), else RESULT = 0.

9.6 ADDED LINK ROUTINES TO VERSION 1.1

Version 1.1 of MCS BASIC-52 contains a number of useful assembly language link routines that were not available in Version 1.0. Most of these routines were designed to be used in conjunction with the new Command/Statement extensions that are described in Chapter 11 of this manual. The added link routines are as follows:

OPBYTE = 57 (39H) EVALUATE AN EXPRESSION WITHIN THE BASIC TEXT STRING AND PLACE THE RESULT ON THE ARGUMENT STACK

This routine permits the user to evaluate a BASIC expression [expr] containing variables, operators and constants. The result of the evaluated expression is placed on the floating point argument stack. This lets the user evaluate expressions in "customized" statements and commands. An example of use of this OPBYTE is given at the end of this section.

OPBYTE = 58 (3AH) PERFORM CRYSTAL DEPENDENT CALCULATIONS WITH THE VALUE THAT IS ON THE ARGUMENT STACK

This routine is provided mainly to let the user write an assembly language RESET routine and perform all of the crystal dependent calculations that are required by MCS BASIC-52. An example of a customized RESET routine that uses this OPBYTE is presented in Chapter 11 of this manual.

OPBYTE = 63 (3FH) GET A CHARACTER OUT OF THE BASIC TEXT STRING

This routine permits the user to "pick" a character out of the BASIC program. For instance, in BASIC the user could have the following:

```
10    CALL 1000H A
```

If the user executed the following in assembly language at 1000H:

```
MOV     A, #63
LCALL   30H
```

The character A would be returned in the accumulator. The Basic text pointer is located in location 8 (8H) (low byte) and 10 (0AH) (high byte) of the internal ram on the MCS BASIC-52 device. If the user were to implement the above function, the basic text pointer must be advanced to the carriage return at the end of the statement before returning back to Basic. Failure to do this will cause a BAD SYNTAX ERROR when the user returns back to Basic. The following OPBYTE can be used to advance the Basic Text pointer.

9.6 ADDED LINK ROUTINES TO VERSION 1.1

OPBYTE = 64 (40H) GET CHARACTER, THEN INCREMENT TEXT POINTER

This OPBYTE does the same thing as the previous one described, except that the BASIC text pointer is INCREMENTED AFTER the character is read. An example of this OPBYTE is presented at the end of this section.

OPBYTE = 65 (41H) INPUT A CHARACTER FROM THE CONSOLE DEVICE, PUT IT IN THE ACCUMULATOR, THEN RETURN

This OPBYTE permits the user to input characters from MCS BASIC-52's console input routine. The character is placed in the accumulator upon return.

OPBYTE = 66 (42H) ENTER THE RUN MODE

This OPBYTE permits the user to start the execution of an MCS BASIC-52 program from assembly language. The user need only insure that locations 19 (13H) and 20 (14H) of internal data memory contain the start address (high byte, low byte respectively) of the BASIC program.

OPBYTE = 129 (81H) INPUT AN ASCII FLOATING POINT NUMBER AND PLACE IT ON THE ARGUMENT STACK. THE DPTR POINTS TO THE EXTERNAL RAM LOCATION, WHERE THE ASCII TEXT STRING IS STORED

This routine assumes that the user has placed an ASCII text string somewhere in memory and that this ASCII text string represents a valid floating point number. The user then puts the DPTR to the starting address of this text string. After this OPBYTE is executed the text string will be converted to a valid MCS BASIC-52 floating point number and placed on the argument stack and the DPTR will be advanced to the end of the floating point number. If the DPTR does not point to a text string that contains a valid floating point number, the accumulator will contain an 0FFH upon return.

OPBYTE = 152 (98) OUTPUT, IN HEX, TO THE CONSOLE OUTPUT DRIVER, THE CONTENTS OF R3:R1

This routine is used to display HEX numbers, assuming that they are in registers R3:R1. If R3 = 0, leading zeros can be supressed by setting BIT 54 (36H) before calling this routine. If BIT 54 (36H) is cleared when this routine is called, the driver will always output four hex digits followed by the character H. This routine always outputs a space character (20H) to the console device, before any hex digits are output. BIT 54 (36H) is bit 6 of internal RAM location 38.

9.6 ADDED LINK ROUTINES TO VERSION 1.1

EXAMPLE:

```
MCS-51 MACRO ASSEMBLER

ISIS-II MCS-51 MACRO ASSEMBLER V1.0
OBJECT MODULE PLACED IN :F4:DEMO.HEX
ASSEMBLER INVOKED BY:  ASM51 :F4:DEMO

LOC  OBJ            LINE     SOURCE

                        1     ;*************************************************************************
                        2     ;
                        3     ; The following is an example of a program that uses the new OPBYTES
                        4     ; available in version 1.1 of MCS BASIC-52.  This code is by no means
                        5     ; optimized, but it is meant to demonstrate how the user can define
                        6     ; "customized" commands and statements in version 1.1 of MCS BASIC-52.
                        7     ;
                        8     ; The new command defined here is DISPLAY.  What it does is display a
                        9     ; region of external data memory to the console device.  The syntax
                       10     ; for this statement is:
                       11     ;
                       12     ;                       DISPLAY [expr], [expr]
                       13     ;
                       14     ; Where the first expression is the starting address and the last
                       15     ; expression is the ending address.  In this example the DISPLAY is
                       16     ; treated like a command which means that it cannot be executed in
                       17     ; RUN mode.
                       18     ;
                       19     ; The output for the DISPLAY command is as follows:
                       20     ;
                       21     ; ADDRESS then 16 Bytes of Characters i.e.
                       22     ;
                       23     ; 1000H 00H 22H 33H 27H ..........
                       24     ;
                       25     ; Now, on to the program.
                       26     ;*************************************************************************
```

9.6 ADDED LINK ROUTINES TO VERSION 1.1

```
LOC  OBJ          LINE  SOURCE

                    27
                    28          ORG     2002H
                    29
2002                30
2002 5A             31          DB      5AH         ; Tell basic that expansion option is
                    32                              ; present
2048                33          ORG     204BH
                    34
2048 D22D           35          SETB    45          ; Set the bit that says so
204A 22             36          RET
                    37
2070                38          ORG     2070H       ; Set up DPTR to jump table
                    39
2070 90207C         40          MOV     DPTR,#VECTOR_TABLE
2073 22             41          RET
                    42
2078                43          ORG     2078H       ; Set up DPTR to expansion table
                    44
2078 90207E         45          MOV     DPTR,#USER_TABLE
2078 22             46          RET
                    47  VECTOR_TABLE:
                    48
207C 2087           49          DW      DO_DISPLAY  ; This is the address of DISPLAY
                    50
                    51  USER_TABLE:
                    52
207E 10             53          DB      10H         ; Token for Display
207F 44495350       54          DB      'DISPLAY'   ; ASCII for display
2083 4C4159
2086 FF             55          DB      0FFH        ; End of table (short table)
                    56
                    57
                    58
                    59  DO_DISPLAY:
                    60
2087 302F63         61          JNB     47,DUMMY    ; make sure that MCS BASIC-52 is in
                    62                              ; the command mode. Bit 47 is set
                    63                              ; if it is.
                    64
208A 7439           65          MOV     A,#57       ; Evaluate the first expression after
208C 120030         66          LCALL   30H         ; the keyword display, MCS BASIC-52
                    67                              ; will handle any errors. The value
                    68                              ; of the expression will be on the
                    69                              ; Argument Stack
                    70
```

```
LOC  OBJ            LINE     SOURCE

208F 7440             71             MOV     A,#64           ; Get the character after the expression
2091 120030           72             LCALL   30H             ; and bump the BASIC text pointer
                      73
2094 B42C6A           74             CJNE    A,#',',C_ERROR  ; Make sure it is a comma, if not do an
                      75                                     ; error
                      76
2097 7439             77             MOV     A,#57           ; Evaluate the next expression (the
2099 120030           78             LCALL   30H             ; ending address) and put it on the
                      79                                     ; Argument Stack
                      80
209C 7401             81             MOV     A,#1            ; Convert the last expression (the
209E 120030           82             LCALL   30H             ; ending address) on the stack to
                      83                                     ; an integer and put it in R3:R1
                      84
20A1 8918             85             MOV     18H,R1          ; Save the ending address in the user
20A3 8B19             86             MOV     19H,R3          ; reserved locations 18H and 19H. This
                      87                                     ; is reserved as register bank 3
                      88
20A5 7401             89             MOV     A,#1            ; Convert the first expression (the
20A7 120030           90             LCALL   30H             ; starting address) on the stack to
                      91                                     ; an integer and put it in R3:R1
                      92
20AA 891A             93             MOV     1AH,R1          ; Save the starting address in the user
20AC 8B1B             94             MOV     1BH,R3          ; reserved locations 1AH and 1BH
                      95
                      96                                     ; Now everything is set up to loop
                      97
20AE C3               98     LOOP1:  CLR     C               ; Check to make sure that the starting
20AF E518             99             MOV     A,18H           ; or current address is <= the ending
20B1 951A            100             SUBB    A,1AH           ; address
20B3 E519            101             MOV     A,19H
20B5 951B            102             SUBB    A,1BH
20B7 5004            103             JNC     LOOP2           ; If the carry is set, it's over
20B9 E4              104             CLR     A               ; Go to the command mode

20BA 020030          105             LJMP    30H             ; (if display was a statement instead
                     106                                     ; of a command, this routine would
                     107                                     ; exit with a RET
                     108
20BD 7407            109     LOOP2:  MOV     A,#7            ; Do a carriage return, line feed
20BF 120030          110             LCALL   30H
                     111
20C2 A91A            112             MOV     R1,1AH          ; Output the Starting address
20C4 AB1B            113             MOV     R3,1BH
                     114
20C6 C236            115             CLR     36H             ; Don't supress leading zeros
```

```
LOC  OBJ              LINE     SOURCE

20C8 7498              116              MOV     A,#98H
20CA 120030            117              LCALL   30H
                       118
20CD 851A82            119     LOOP3:   MOV     DPL,1AH          ; Now, set up to read 16 bytes
20D0 851B83            120              MOV     DPH,1BH          ; put address in DPTR
                       121
20D3 E0                122              MOVX    A,@DPTR          ; Read the byte in external RAM
20D4 A3                123              INC     DPTR             ; Bump to the next location
                       124
20D5 85821A            125              MOV     1AH,DPL          ; Save the Address
20D8 85831B            126              MOV     1BH,DPH
                       127
20DB F9                128              MOV     R1,A             ; Output the byte
20DC 7B00              129              MOV     R3,#0            ; The high byte is always zero
20DE D236              130              SETB    36H              ; Supress leading Zeros
20E0 7498              131              MOV     A,#98H
20E2 120030            132              LCALL   30H
                       133
20E5 E51A              134              MOV     A,1AH            ; Check to see if on a 16 byte boundry
20E7 540F              135              ANL     A,#0FH
20E9 70E2              136              JNZ     LOOP3            ; Loop until on a 16 Byte Boundry
20EB 80C1              137              SJMP    LOOP1
                       138
                       139     DUMMY:
                       140
20ED 7407              141              MOV     A,#7             ; Do a carriage return-line feed
20EF 120030            142              LCALL   30H
                       143
20F2 7B21              144              MOV     R3,#HIGH D_MSG   ; Display the error message
20F4 7915              145              MOV     R1,#LOW D_MSG
20F6 D234              146              SETB    52               ; Print from ROM
20F8 7406              147              MOV     A,#6
20FA 120030            148              LCALL   30H
20FD E4                149              CLR     A                ; Go back to the command mode
20FE 020030            150              LJMP    30H
                       151
                       152     C_ERROR:
                       153
2101 7407              154              MOV     A,#7             ; Do what we did before
2103 120030            155              LCALL   30H
                       156
2106 7B21              157              MOV     R3,#HIGH C_MSG
2108 793B              158              MOV     R1,#LOW C_MSG
210A D234              159              SETB    52
210C 7406              160              MOV     A,#6
210E 120030            161              LCALL   30H
```

9.6 ADDED LINK ROUTINES TO VERSION 1.1

```
LOC  OBJ                LINE    SOURCE

2111 E4                  162             CLR     A
2112 020030              163             LJMP    30H
                         164
2115 44495350            165     D_MSG:  DB      'DISPLAY IS A COMMAND, NOT A STATEMENT"'
2119 4C415920
211D 49532041
2121 20434F4D
2125 4D414E44
2129 2C204E4F
212D 54204120
2131 53544154
2135 454D454E
2139 5422
                         166
213B 594F5520            167     C_MSG:  DB      'YOU NEED A COMMA TO MAKE DISPLAY WORK"'
213F 4E454544
2143 20412043
2147 4F4D4D41
214B 20544F20
214F 4D414B45
2153 20444953
2157 504C4159
215B 20574F52
215F 4B22
                         168
                         169             END

ASSEMBLY COMPLETE, NO ERRORS FOUND
  (that's all it takes)
```

9.7 INTERRUPTS

Interrupts can be handled by MCS BASIC-52 in two distinct ways. The first, which has already been discussed, allows statements in an MCS BASIC-52 program to perform the required interrupt routine. The ONTIME and ONEX1 statements enable this particular interrupt mode. Additionally, setting BIT 26.1H permits EXTERNAL INTERRUPT 0 to act as a "fake" DMA input and the details of this feature are in the BELLS, WHISTLES, and ANOMALIES section of this manual. The second method of handling interrupts in MCS BASIC-52 allows the programmer to write assembly language routines to perform the interrupt task. This method yields a much faster interrupt response time, but, the programmer must exercise some caution.

All interrupt vectors on the MCS BASIC-52 device are "mirrored" to external PROGRAM MEMORY LOCATIONS 4003H through 402BH inclusive. The only MCS BASIC-52 STATEMENTS that enable the interrupts on the 8052AH are the CLOCK1 and the ONEX1 STATEMENTS. If interrupts are NOT enabled by these STATEMENTS, BASIC assumes that the USER is providing the interrupt routine in assembly language. The vectors for the various interrupts are as follows:

LOCATION---INTERRUPT

4003H------EXTERNAL INTERRUPT 0

400BH------TIMER 0 OVERFLOW

4013H------EXTERNAL INTERRUPT 1

401BH------TIMER 1 OVERFLOW

4023H------SERIAL PORT

402BH------TIMER 2 OVERFLOW/EXTERNAL INTERRUPT 2

The programmer can enable interrupts in MCS BASIC-52 by using the statement IE = IE.OR.XXH, where XX enables the appropriate interrupts. The bits in the interrupt register (IE) on the 8052AH are defined as follows:

BIT	7	6	5	4	3	2	1	0
	EA	X	ET2	ES	ET1	EX1	ET0	EX0
	ENABLE ALL	UNDE-FINED	TIMER 2	SERIAL PORT	TIMER 1	EXT 1	TIMER 0	EXT 0

9.7 INTERRUPTS

Interrupts are enabled when the appropriate BITS in the IE register are set to a one. Details of the 8052AH interrupt structure are available in the MICROCONTROLLER USERS MANUAL available from INTEL.

IMPORTANT NOTE!!

Before MCS BASIC-52 vectors to the USER interrupt locations just described, the PROCESSOR STATUS WORD (PSW) is PUSHED onto the STACK. So, the USER does not have to save the PSW in the assembly language interrupt routine!!! HOWEVER, *THE USER MUST POP THE PSW BEFORE RETURNING FROM THE INTERRUPT.*

VERY IMPORTANT NOTE!!!

If the user is running some interrupt driven "background" routine while MCS BASIC-52 is running a program, the user *MUST NOT* CALL any of the assembly language routines available in the MCS BASIC-52 device. The only way the routines in the MCS BASIC-52 device can be accessed is when the CALL statement in MCS BASIC-52 is used to transfer control to the users assembly language program. The reason for this is that the MCS BASIC-52 interpreter must be in a "known" state before the user can call the routines available in the MCS BASIC-52 device and a "random" interrupt does not guarantee that the interpreter is in this known state. The user should use REGISTER BANK 3 to handle interrupt routines in assembly language.

9.8 RESOURCE ALLOCATION

Specific statements in MCS BASIC-52 require the use of certain hardware features on the device. If the user wants to use these hardware features for interrupt driven routines, conflicts between BASIC and the assembly language routine may occur. To avoid these potential conflicts, the programmer needs to know what hardware features are used by MCS BASIC-52. The following is a list of the COMMANDS and/or STATEMENTS that use the hardware features on the 8052AH.

CLOCK1 — uses TIMER/COUNTER 0 in the 13 bit 8048 mode.

PWM — uses TIMER/COUNTER 1 in the 16 bit mode

LIST# — uses TIMER/COUNTER 1 to generate baud rate in 16 bit mode

PRINT# — same as LIST#

PROG — uses TIMER/COUNTER 1 for programming pulse

ONEX1 — uses EXTERNAL INTERRUPT 1

In addition, TIMER/COUNTER 2 is used to generate the baud rate for the serial port. What the preceding list means is that if CLOCK1, PWM, ONEX1, LIST#, PRINT#, and PROG commands/statements are used by the programmer, the user *MAY NOT* use the associated TIMER/COUNTER or EXTERNAL INTERRUPT pin for an assembly language routine.

MCS BASIC-52 initializes the TIMER/COUNTER modes by writing a 244 (0F4H), 16 (10H), and 52 (34H) to the TCON, TMOD, and T2CON registers respectively. These registers are initialized only during the RESET initialization sequence, and MCS BASIC-52 assumes that these registers are NEVER changed. So, if the user changes the contents of TCON, TMOD, or T2CON, something funny and/or disastrous is bound to happen if the Statements/Commands listed above are executed. If the user does not execute any of the previously mentioned Statements or Commands, the user is free to use the interrupts in any way suitable to the application.

CHAPTER 10
System Configuration

10.1 MEMORY/HARDWARE CONFIGURATION

MCS BASIC-52 always requires at least 1K bytes of external memory. After reset, MCS BASIC-52 sizes the external memory. If less than 1K bytes of external memory are available, MCS BASIC-52 will not "sign-on," in fact, it will internally loop forever. This obviously is not too exciting, so it is wise to hang some external memory on the MCS BASIC-52 device.

MCS BASIC-52 sizes consecutative external memory locations from 0000H until a memory failure is detected. The sizing operation is performed simply by writing a 5AH to an external memory location, then testing the location. If the particular memory location passes this test, BASIC then writes a 00H to the location, then again, checks the location. MCS BASIC-52 only sizes the external memory from locations 0 through 0DFFFH. Memory locations 0E000H through 0FFFFH are reserved for user I/O and/or assembly language programs.

The MCS BASIC-52 program resides in the 8K of ROM available in INTEL's 8052AH device and as a result requires that external memory be "partitioned" in a specific manner. The architecture of the 8052AH is NOT Von Neumann. This means that Data and Program Memory do not reside in the same physical address space on the 8052AH. Specifically, the $\overline{RD}$ (pin 17) and $\overline{WR}$ (pin 16) pins on the 8052AH are used to enable DATA memory and $\overline{PSEN}$ (pin 29) pin is used to enable PROGRAM memory. Depending on the hardware configuration, MCS BASIC-52 operates in two distinct "memory" modes.

RAM ONLY MODE

In this mode of operation, Read/write memory is connected to the MCS BASIC-52 device starting at memory address 0000H. Memory can be placed up to location 0FFFFH. In this mode of operation the decoded addresses are used to generate the CHIP SELECT ($\overline{CS}$) signal for the RAM devices. The $\overline{RD}$ pin on the 8052AH is used to generate the OUTPUT ENABLE ($\overline{OE}$) strobe and the $\overline{WR}$ pin generates the WRITE ENABLE ($\overline{WE}$ or $\overline{WR}$) strobe. $\overline{PSEN}$ is not used in the RAM only mode of operation. The RAM only mode of operation offers the simplest hardware configuration available for the MCS BASIC-52 device. An example of this configuration is shown in Figure 1. Since $\overline{PSEN}$ is not used in the RAM only mode, the user may not CALL assembly language routines. The RAM only also does not support EPROM programming. In general, the RAM only mode will be used only to "check out" the device during the initial system development stage.

10.1 MEMORY/HARDWARE CONFIGURATION

RAM/EPROM MODE

The RAM/EPROM mode of operation allows for the complete system implementation of MCS BASIC-52. This mode of operation requires that external memory be mapped in a certain manner. The RAM/EPROM memory configuration is as follows:

1) The $\overline{RD}$ and the $\overline{WR}$ pins on the MCS BASIC-52 device are used to enable RAM memory that is addressed from 0000H to 7FFFH. Addresses are used to decode the chip select ($\overline{CS}$) for the RAM devices and $\overline{RD}$ and $\overline{WR}$ are used to enable the $\overline{OE}$ and $\overline{WE}$ or ($\overline{WR}$) pins respectively.

2) The $\overline{PSEN}$ pin on the MCS BASIC-52 device is used to enable EPROM memory that is addressed from 2000H to 7FFFH. Addresses are used to decode the chip select ($\overline{CS}$) for the EPROM devices and $\overline{PSEN}$ is used to enable the $\overline{OE}$ pin.

3) For addresses between 8000H and 0FFFFH both the $\overline{RD}$ and the $\overline{PSEN}$ pin on the MCS BASIC-52 device are used to enable the memory. Either EPROM or RAM devices can be placed in this address space. To permit both the $\overline{RD}$ and the $\overline{PSEN}$ pins to enable addresses in this address space, $\overline{RD}$ and $\overline{PSEN}$ must be logically "ANDED" together. This can be accomplished with a simple TTL gate such as a 74LS08. The $\overline{WR}$ pin on the MCS BASIC-52 device is used to write to RAM memory in this same address space. The $\overline{PSEN}$ and $\overline{RD}$ signals do not have to be anded beyond address 7FFFH to enable MCS BASIC-52 to program an EPROM. This is only a suggestion since it will permit the user to execute assembly language routines as well as MCS BASIC-52 programs that are located in this address space.

10.1 MEMORY/HARDWARE CONFIGURATION

This scheme of memory addressing actually permits MCS BASIC-52 to address 96K bytes of memory, 32K of RAM devices, 32K of EPROM/ROM devices and 32K of combined RAM/EPROM/ROM devices. Since $\overline{RD}$ and $\overline{PSEN}$ are ANDED for addresses from 8000H through 0FFFFH, the 8052AH "looks like" a Von Neumann machine in this address space. The XBY and CBY special function operators will yield the same value when their arguments are between 8000H and 0FFFFH.

When the EPROM programming feature in MCS BASIC-52 is used, BASIC assumes that the EPROM to be programmed is addressed starting at location 8000H. MCS BASIC-52 can only program EPROMS addressed between 8000H and 0FFFFH. When the PROG command is used for the first time, on an erased EPROM, MCS BASIC-52 stores this program beginning at address 8010H. Locations 8000H through 800FH are used to save the baud rate information, plus configuration information. Some suggestions for implementation of the RAM/EPROM mode are shown in figure 2.

10.2 EPROM PROGRAMMING CONFIGURATION/TIMING

With the proper hardware, the MCS BASIC-52 device can program just about any EPROM or EEPROM device. The only requirement for EPROM programming is that the EPROM to be programmed is addressed starting at location 8000H. MCS BASIC-52 requires very little external hardware to programs EPROMS. All of the critical EPROM programming timings are generated by threeI/O port pins on the MCS BASIC-52 device. These pins provide the following signals:

P1.3 — ALE DISABLE

PORT 1, BIT 3 (pin 4 on the 8052AH) is used to DISABLE the ALE signal to the external latched required by the 8052AH when external memory is addressed. This pin should be logically ANDED with ALE. A simple TTL gate, such as a 74LS08 can be used to perform the ANDING function. Under normal operation, P1.3 is in a logical high state (1). ONLY DURING EPROM PROGRAMMING IS P1.3 PLACED IN A LOGICAL LOW STATE (0). Disabling the ALE signal to the external latch is required to program EPROMS because of the way MCS BASIC-52 carries out the EPROM programming process.

During programming, MCS BASIC-52 treats I/O PORT 0 and I/O PORT 2 as I/O ports, not as address/data ports. MCS BASIC-52 first writes the low order address to be programmed to PORT 0. The data in PORT 0 is then latched into the external address latch and then MCS BASIC-52 disables the ALE signal to the latch by clearing bit P1.3. Thus, the low order address is "permanently" stored in the external latch. MCS BASIC-52 then writes the high order address to PORT 2 and the DATA to be programmed to PORT 0. So, the external address latch contains the low order address, PORT 2 contains the high order address, and PORT 0 contains the DATA when EPROM programming occurs.

IMPORTANT NOTES

When PORT 0 on the 8052AH is used as an I/O port, the output structure is an "open drain" configuration. This requires that "pull-up" resistors be placed on PORT 0 to permit MCS BASIC-52 to program EPROMS. Experimentally, 10K ohm pull-ups resistors on PORT 0 have yielded satisfactory results.

In Version 1.1, INT0 must be kept high when programming EPROMs.

10.2 EPROM PROGRAMMING CONFIGURATION/TIMING

P1.4 — PROGRAM PULSE WIDTH

PORT 1, BIT 4 (pin 5 on the 8052AH) is used to provide the 50 millisecond or the 1 millisecond programming pulse. The length of the programming pulse is determined by whether the "normal" or the "INTELligent" EPROM programming mode is selected. MCS BASIC-52 calculates the length of the programming pulse from the assigned crystal value. So, be sure the proper XTAL has been assigned. The accuracy of this pulse is within 10 CPU clock cycles. This pin is normally in a logical high (1) state. It is asserted low (0) to program the EPROMS. Depending on the EPROM to be programmed this signal will be used in different ways. More about this later.

P1.5 — ENABLE PROGRAM VOLTAGE

PORT 1, BIT 5 (pin 6 on the 8052AH) is used to enable the EPROM programming voltage. This pin is normally in a logical high (1) state. Prior to the EPROM programming operation, this pin is brought to a logical low (0) state. This pin is used to turn on or off the high voltage (12.5 volts to 25 volts, depending on the EPROM) required to program the EPROMS.

The timing for the EPROM programming pins is shown in figure 3. The hardware required to program different devices is shown in figure 4. Note that with very little external hardware the MCS BASIC-52 device can program virtually all commercially available EPROMS. Additionally, figure 5 suggests a circuit using an INTEL 2816A EEPROM. This circuit also features a push button erase option.

IMPORTANT NOTE

MCS BASIC-52 calculates the programming pulse width when the XTAL value is assigned. To insure proper programming, make sure XTAL is assigned the proper value. MCS BASIC-52 performs the programming pulse width calculation to within 5 clock cycles, so the accuracy of the programming pulse is well within the limits of any EPROM device.

10.3 SERIAL PORT IMPLEMENTATION

The serial port I/O signals on the 8052AH are TTL compatible signals. They are typically not compatible with most terminals. Figure 6 suggests hardware options for making the serial interface compatible with terminals. The serial port is initialized by MCS BASIC-52 to the 8-bit uart mode. In this mode 8 data bits, plus one start and one stop bit are transmitted. Parity is not used.

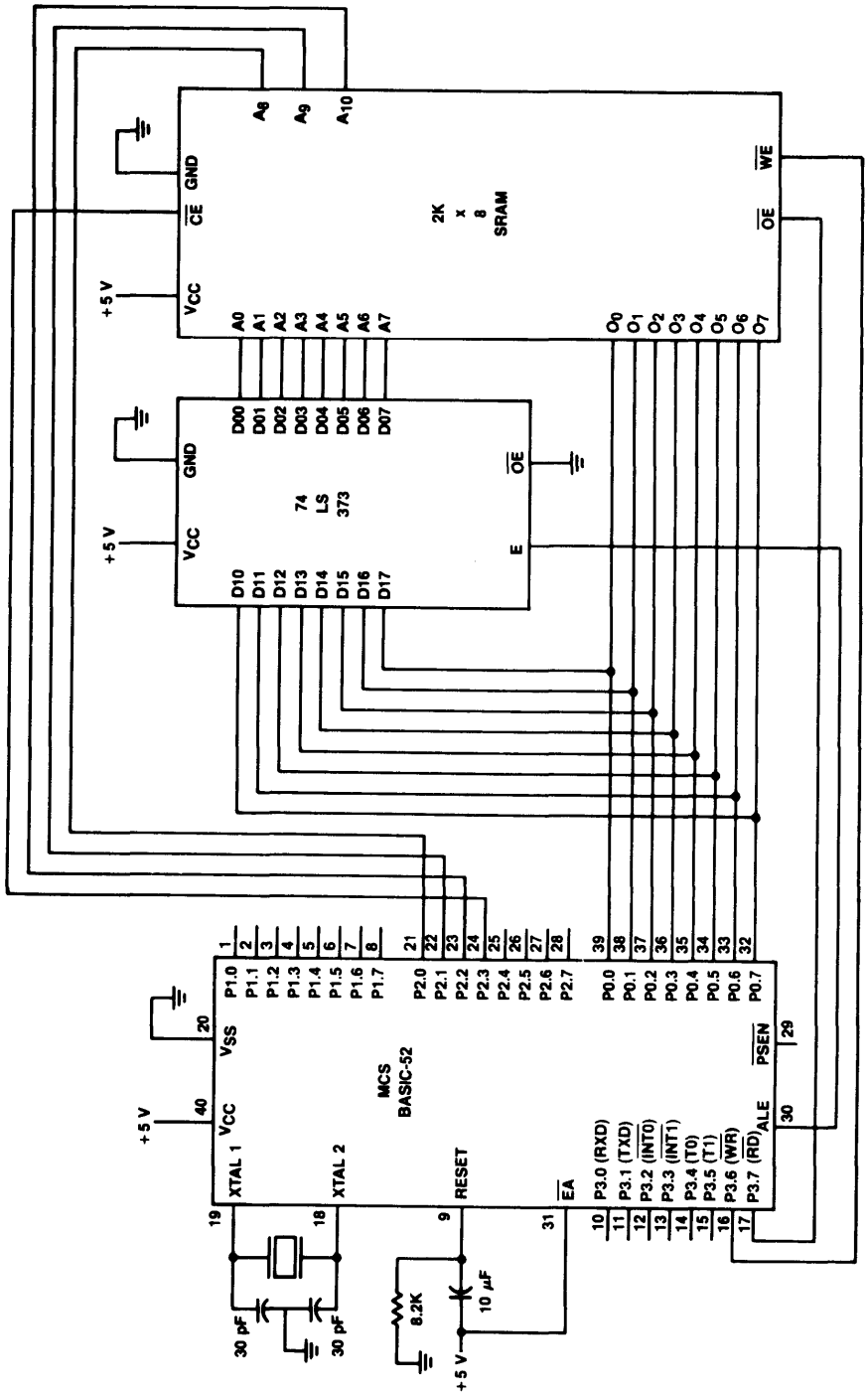


Figure 1. Interface to 2K x 8 Static RAM

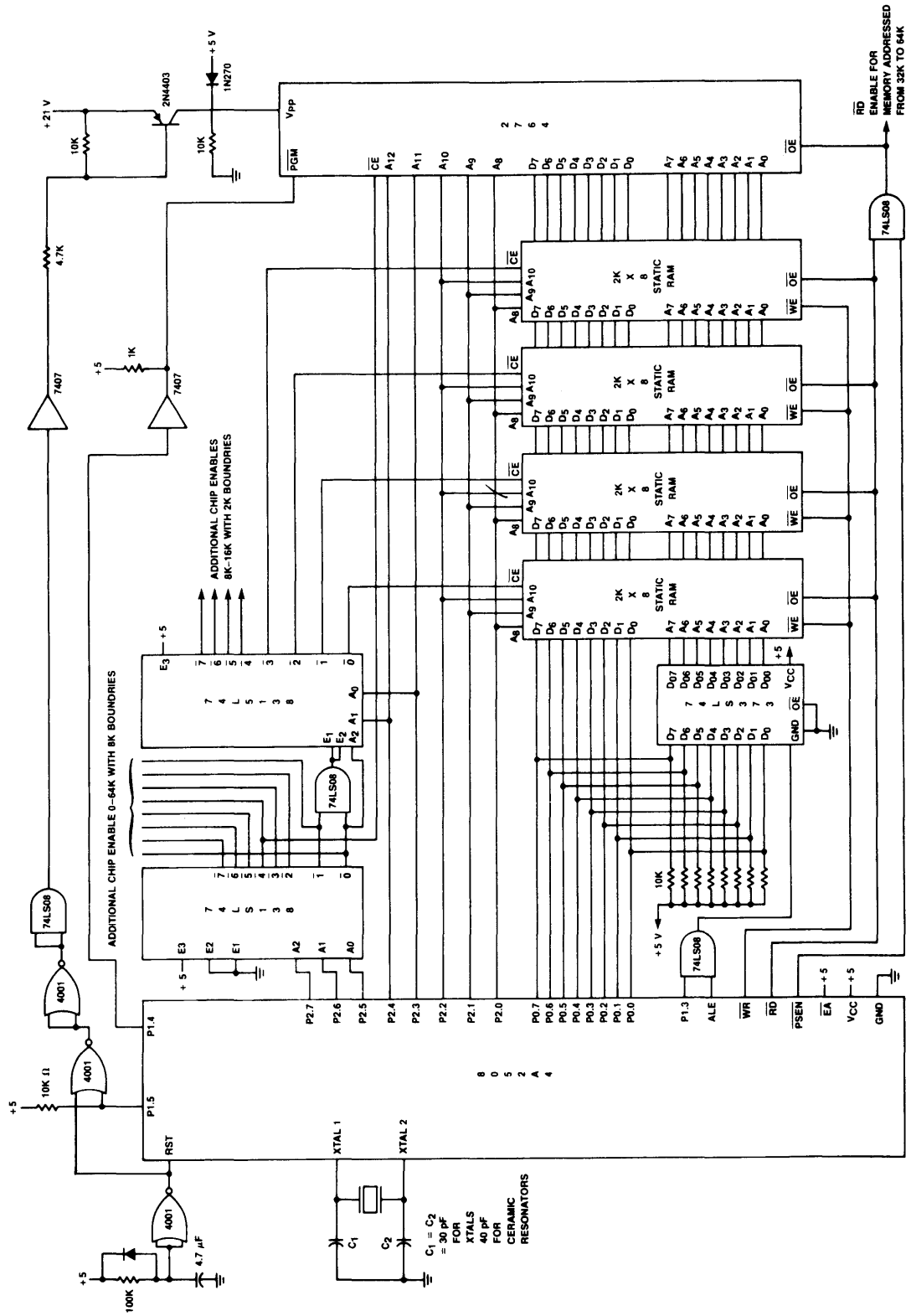


Figure 2A. Full system with EPROM power on protection (no DMA)

This system will decode: RAM from 0 to 16K on 2K boundaries, EPROM from 0 to 32K on 8K boundaries, RAM/EPROM from 32K to 64K on 8K boundaries

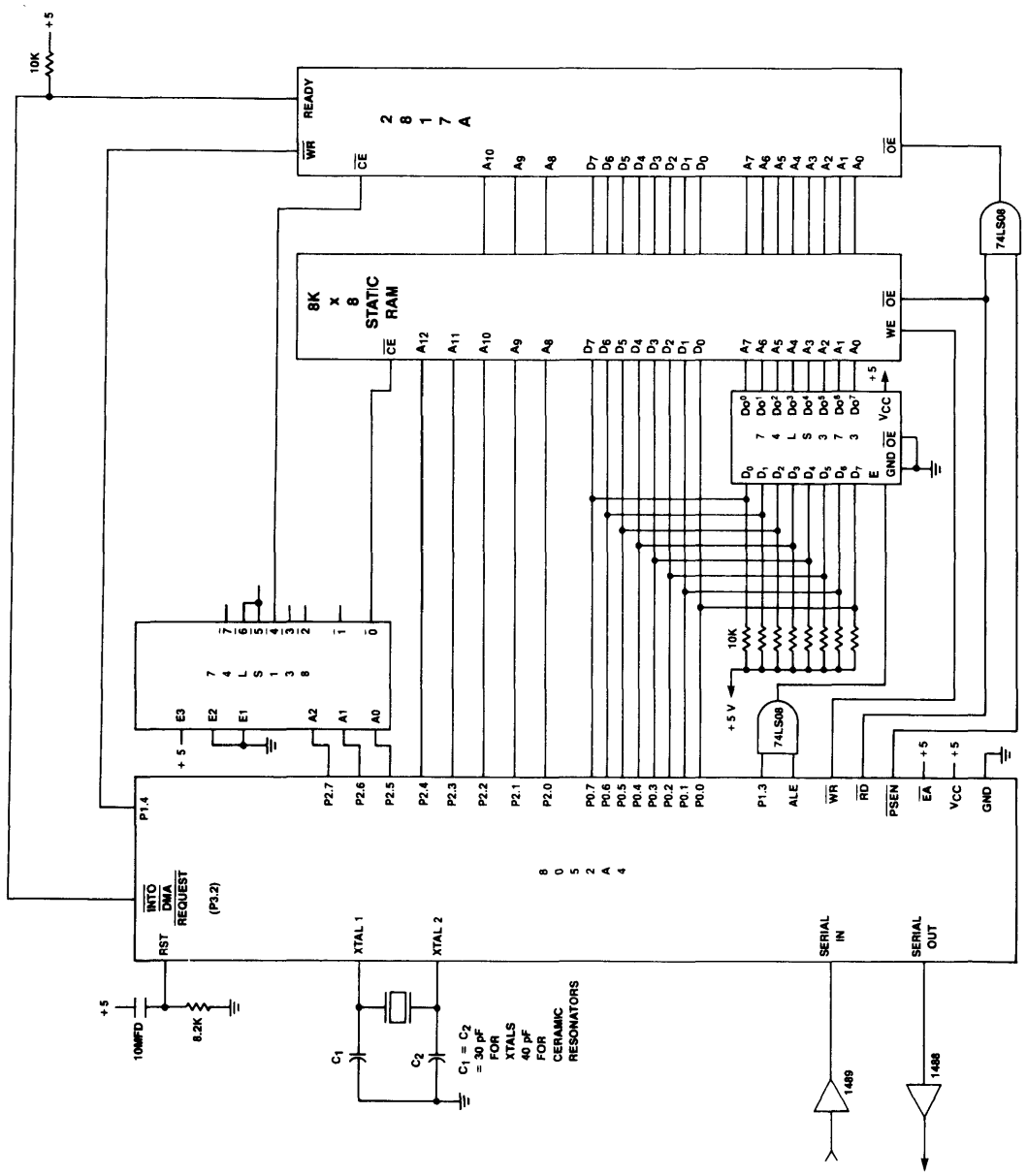


Figure 2B. Programming 2817A's with Version 1.1 of MCS BASIC-52

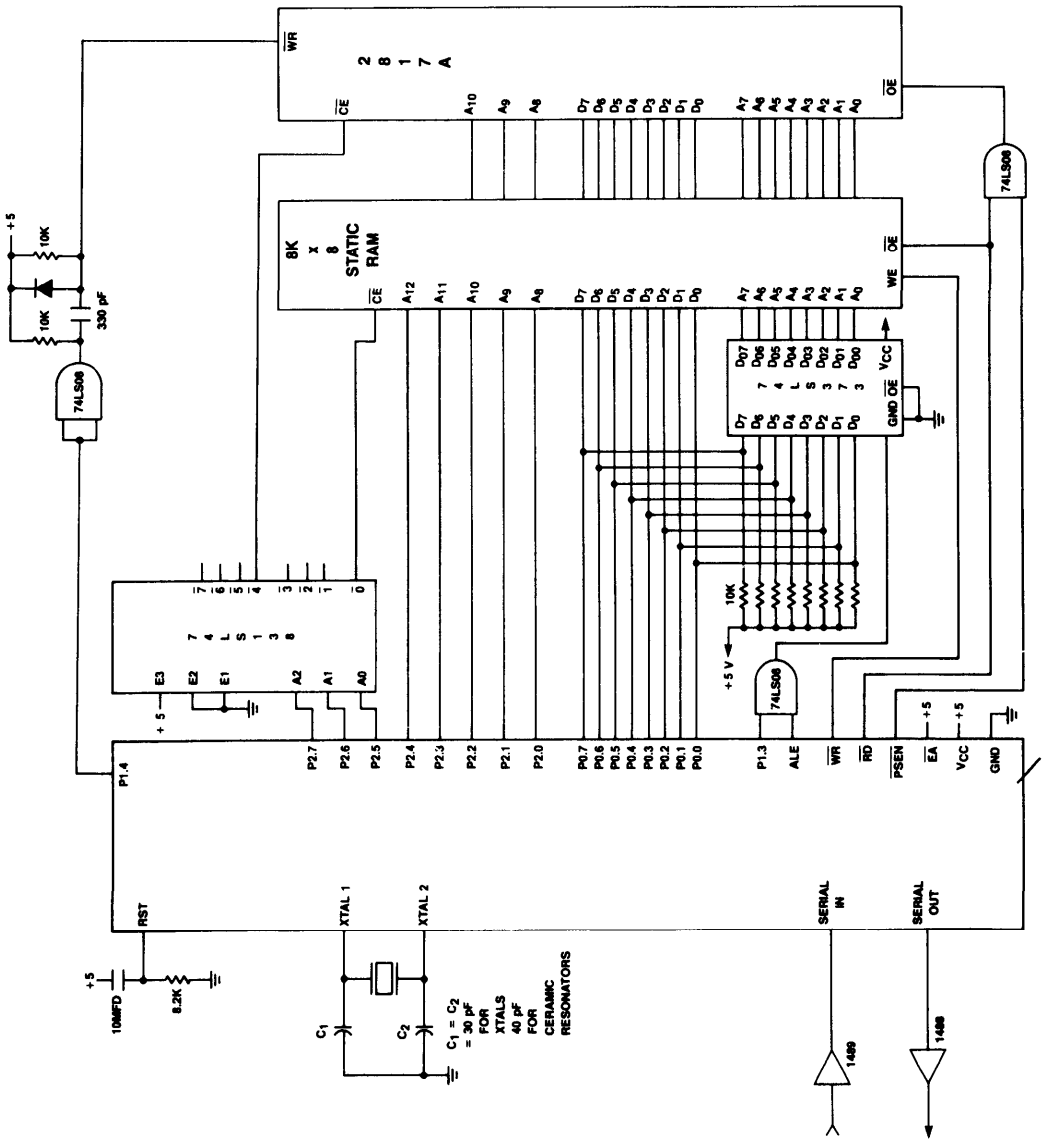


Figure 2C. Programming 2817A's with Version 1.0 of MCS BASIC-52

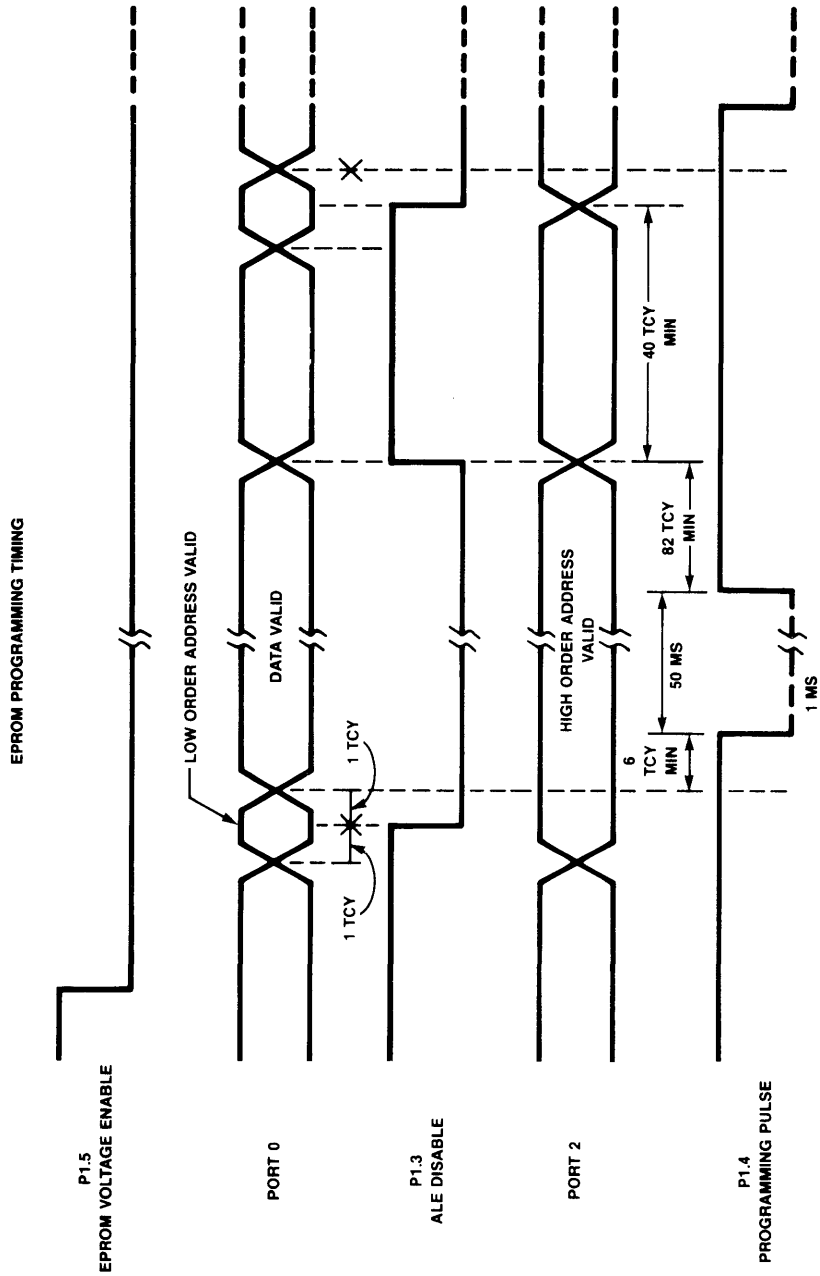
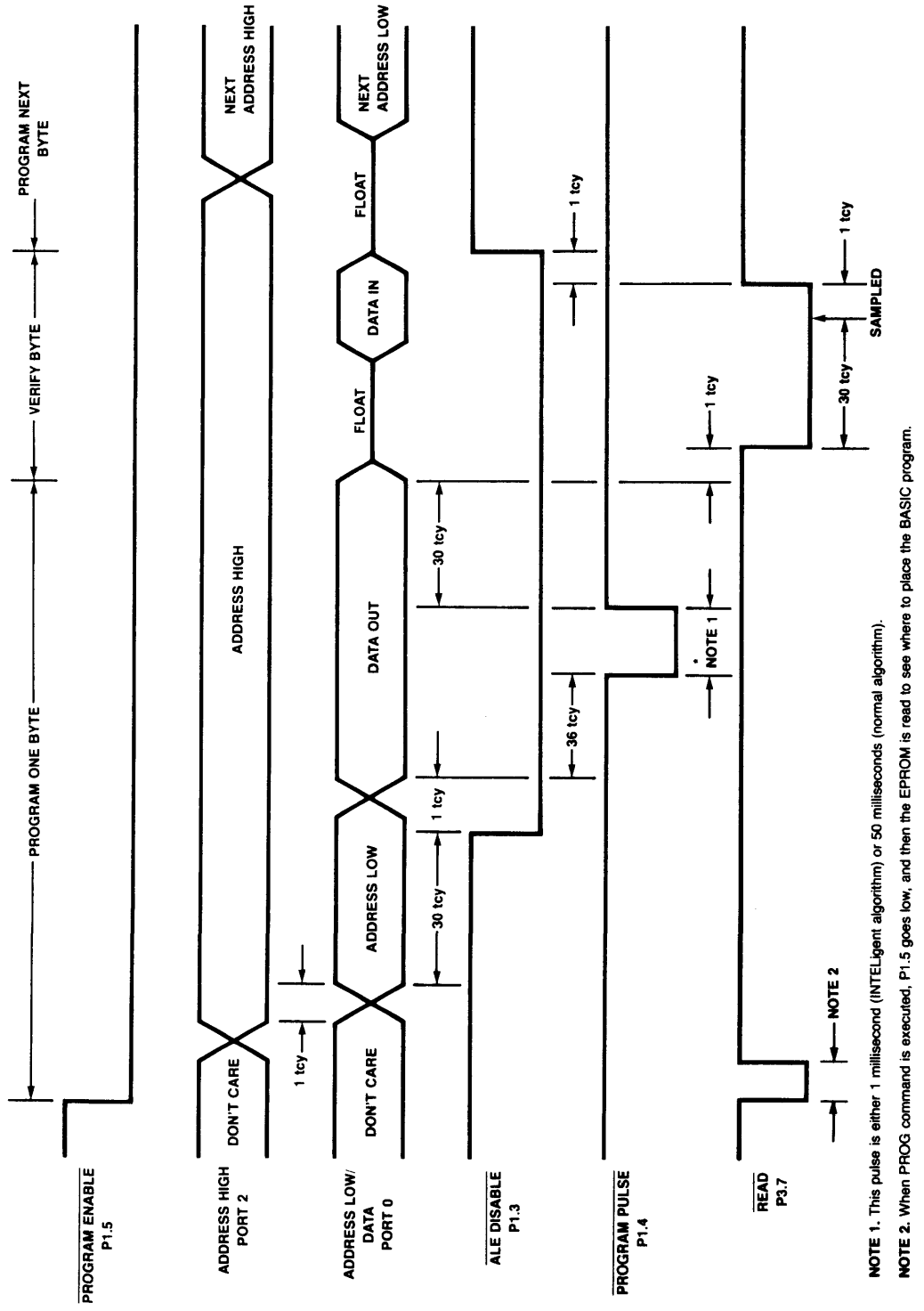


Figure 3A. EPROM Programming Timing Version 1.0



NOTE 1. This pulse is either 1 millisecond (INTELigent algorithm) or 50 milliseconds (normal algorithm).

NOTE 2. When PROG command is executed, P1.5 goes low, and then the EPROM is read to see where to place the BASIC program.

Figure 3B. EPROM programming timing for Version 1.1

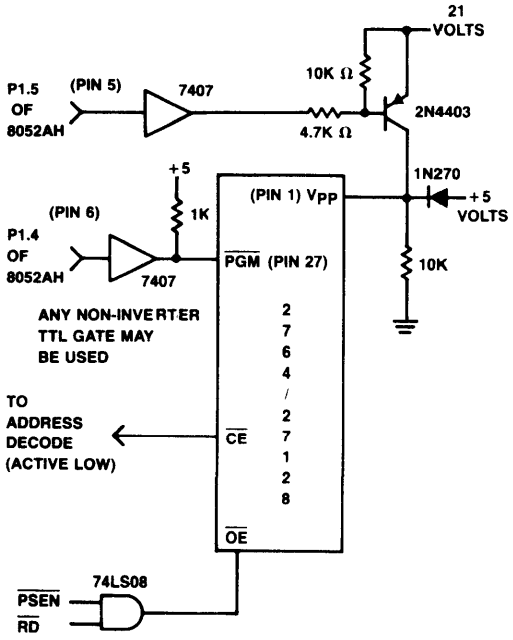


Figure 4A. Programming 2764's/27128's

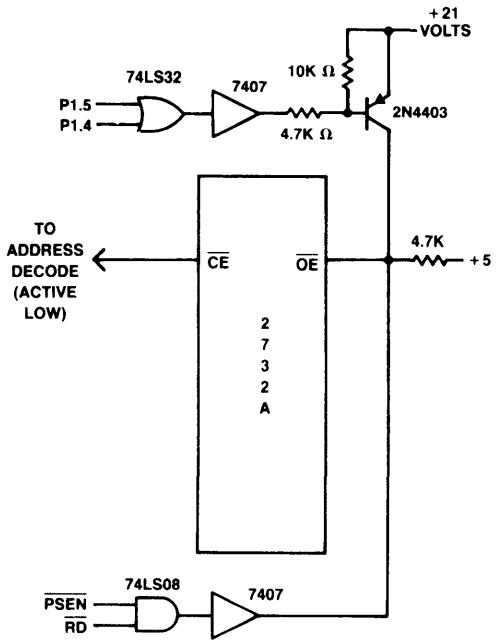


Figure 4B. Programming 2732A's

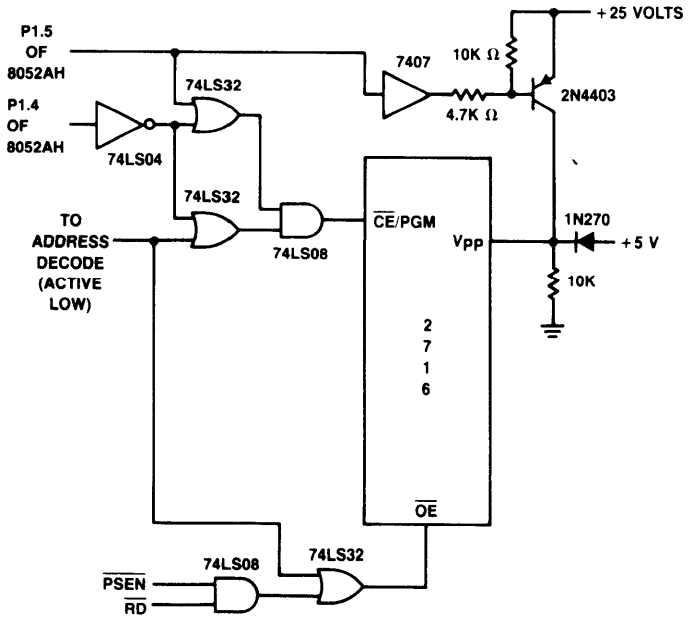


Figure 4C. Programming 2716's

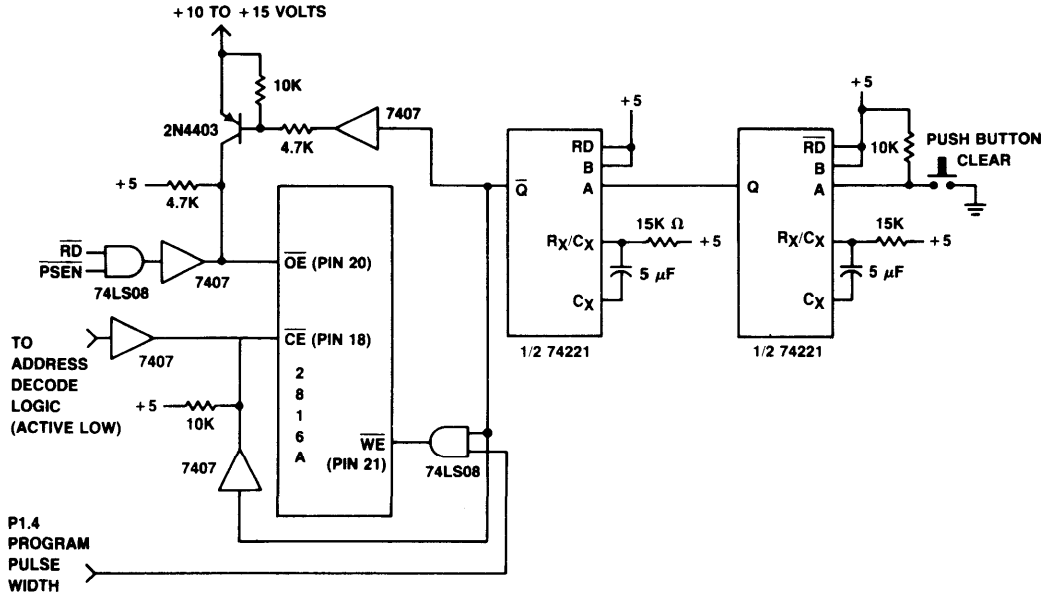


Figure 5. 2816A Circuit with Push Button Erase.

(Basic-52 should be "Idle" in the command mode when the Erase Button is pushed.)

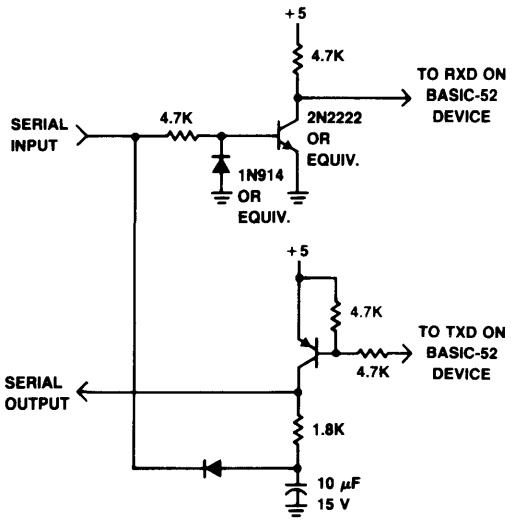


Figure 6A.
TWO TRANSISTORS TO IMPLEMENT RS-232. THE "NEGATIVE" SUPPLY FOR THE SERIAL OUTPUT LINE IS TAKEN FROM THE SERIAL INPUT LINE. NO ±12 VOLT SUPPLY IS REQUIRED.

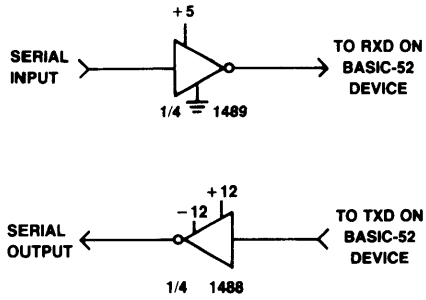


Figure 6B.
USING THE STANDARD 1489 AND 1488 LINE RECEIVERS AND DRIVERS, ±12 VOLTS IS NEEDED WITH THIS IMPLEMENTATION.

CHAPTER 11
Reset Options (Version 1.1 Only)

Version 1.1 of MCS BASIC-52 contains numerous RESET options that were not available in Version 1.0. They are discussed in detail in chapters 3.2 through 3.5 of this manual. Briefly, they are as follows:

PROG1

Saves only the serial port baud rate for a power-up or RESET condition.

PROG2

Saves the serial port baud rate and automatically runs the first program that is saved in EPROM on a power-up or RESET condition.

PROG3

Saves the serial port baud rate plus the assigned MTOP value. If RAM is available beyond the assigned MTOP value, it will not be cleared during a power-up or RESET condition.

PROG4

Saves the serial port baud rate plus the assigned MTOP value, just like PROG3, but also automatically runs the first program that is saved in EPROM on a power-up or RESET condition.

PROG5

Does the same thing as PROG4, however, if external memory location 5FH contains the character 0A5H on a power-up or RESET condition, external memory will not be cleared. This mode assumes that the user has employed some type of memory back-up.

RESET OPTIONS (VERSION 1.1 ONLY)

PROG6

Does the same thing as PROG5, but CALLS external program memory location 4039H during a RESET or power-up sequence. This option also requires the user to put the character 0A5H in external memory location 5FH to insure that external RAM will not be cleared during RESET or power-up. The user must put an assembly language RESET routine in external memory location 4039H or else this RESET mode will crash. When the user returns from the customized assembly language RESET routine, three options exist:

OPTION 1 FOR PROG6

If the CARRY BIT is CLEARED (CARRY = 0) upon return from the user RESET routine MCS BASIC-52 will enter the auto-baud rate determining routine. The user must then type a space character (20H) on the terminal to complete the RESET routine and produce a RESET message on the terminal.

OPTION 2 FOR PROG6

If the CARRY BIT is SET (CARRY = 1) and BIT 0 of the ACCUMULATOR is CLEARED (ACC. 0 = 0) MCS BASIC-52 will produce the standard sign-on message upon return from the user supplied RESET routine. The baud rate will be the one that was saved when the PROG6 option was used.

OPTION 3 FOR PROG6

If the CARRY BIT is SET (CARRY = 1) and BIT 0 of the ACCUMULATOR is SET (ACC. 0 = 1), MCS BASIC-52 will execute the first program stored by the user in EPROM (starting address of the program is 8010H) upon return from the user supplied RESET routine.

RESET OPTIONS (VERSION 1.1 ONLY)

If these options are still not sufficient to address the needs of a specific application, one other option exists and it functions as follows:

After RESET, MCS BASIC-52 initializes the SPECIAL FUNCTION REGISTERS SCON, TMOD, TCON, and T2CON with the following respective values, 5AH, 10H, 54H, and 34H. If the user places the character 0AAH in external CODE MEMORY location 2001H (remember CODE MEMORY is enabled by $\overline{\text{PSEN}}$), MCS BASIC-52 will CALL external CODE MEMORY location 2090H immediately after these special function registers are initialized. No other registers or memory locations will be altered except that the ACCUMULATOR will contain a 0AAH and the DPTR will contain a 2001H.

Since MCS BASIC-52 does not write to the above mentioned Special Function Registers at any time except during the RESET or power-up sequence the user has the option of modifying any of the Special Function Registers with this RESET option. Upon returning from this RESET mode, the MCS BASIC-52 software package will clear the internal memory of the 8052AH and proceed with the RESET routine. The PROG1 through PROG6 options will function as usual.

Now, suppose the user does not want to enter the normal RESET routines, or the user wants to implement some type of "warm" start-up routine. This can be accomplished simple by initializing the necessary Special Function Registers and then jumping back into either MCS BASIC-52's COMMAND mode or RUN MODE. For a warm start-up or RESET (warm means that the MCS BASIC-52 device was RESET, but power was not removed — i.e. the user hit the RESET button) the following must be initialized:

SCON, TMOD, TCON, T2CON, if the user does not want to use the values that MCS BASIC-52 supplies.

RCAP2H and RCAP2L must be loaded with the proper baud rate values. If the user has programmed an EPROM with one of the PROG1 through PROG6 options, the proper baud rate value will be stored in external DATA MEMORY locations 8001H (RCAP2H) and 8002H (RCAP2L).

The STACK POINTER (Special Function Register SP) must be initialized with the contents of the STACK POINTER SAVE location, which is in internal DATA MEMORY location 3EH. A MOV SP, 3EH assembly language instruction will accomplishment the STACK POINTER initialization.

RESET OPTIONS (VERSION 1.1 ONLY)

After the above are initialized by the user supplied RESET routine, the user may enter MCS BASIC-52's command mode by executing the following:

```
CLR     A
LJMP    30H
```

Now, it is important to remember that the previous description applies only to a "warm" RESET with power remaining to the MCS BASIC-52 system. This means that the user must also provide some way of detecting the difference between a warm RESET and a power-on RESET. This usually involves some type of flip-flop getting set with a power-on-clear signal from the users power supply. The details of implementating this RESET detection mechanism will not be discussed here as the possible hardware options vary depending upon the design.

The user may also implement a "cold start" reset option with the previously described reset mode. The following code details what is necessary to implement a cold start option.

EXAMPLE:

```
        ORG     2001H
        ;
        DB      0AAH            ; TELL BASIC THAT RESET IS EXTERNAL
        ;
        ORG     2090H           ; LOCATION BASIC WILL CALL FOR RESET
        ;
                                ; AT THIS POINT BASIC HAS PLACED A 5AH IN
                                ; SCON, A 10H IN TMOD, A 54H IN TCON AND
                                ; A 34H IN T2CON
        ;
                                ; FIRST CLEAR THE INTERNAL MEMORY
        MOV     R0,#0FFH        ; LOAD R0 WITH THE TOP OF INTERNAL MEMORY
        CLR     A               ; SET ACCUMULATOR = 0
        ;
RESET1: MOV     @R0,A           ; LOOP UNTIL ALL THE INTERNAL RAM IS CLEARED
        DJNZ    R0,RESET1
        ;
                                ; NOW SET UP THE STACK POINTER AND THE STACK
                                ; POINTER HOLDING REGISTER
        MOV     SP,#4DH         ; 4DH IS THE INITIALIZED VALUE OF THE STACK
        MOV     3EH,#4DH        ; THIS IS THE SP HOLDING REGISTER
        ;
                                ; NOW CLEAR THE EXTERNAL RAM, IN THIS
                                ; EXAMPLE ASSUME THAT 1FFFH BYTES OF RAM
                                ; ARE AVAILABLE
                                ; THE USER MUST CLEAR AT LEAST THE FIRST 512
                                ; BYTES OF RAM FOR A COLD START RESET
        ;
        MOV     R3,#HIGH 1FFFH
        MOV     R1,#LOW 1FFFH
        MOV     DPTR,#0FFFFH
        ;
```

RESET OPTIONS (VERSION 1.1 ONLY)

```
RESET2: INC     DPTR            ; DPTR = 0 THE FIRST TIME THRU
        CLR     A
        MOVX    @DPTR,A         ; CLEAR THE RAM, A MEMORY TEST PROGRAM COULD
                                ; BE IN THIS LOOP
        MOV     A,R3            ; NOW TEST FOR THE MEMORY LIMITS
        CJNE    A,DPH,RESET2
        MOV     A,R1
        CJNE    A,DPL,RESET2
        ;
        ; WHEN YOU GET HERE, YOU ARE DONE
        ;
        ; NOW SET UP THE MEMORY POINTERS, FIRST MTOP
        ;
        MOV     DPTR,#10AH      ; LOCATION OF MTOP IN EXTERNAL RAM
        MOV     A,#HIGH 1FFFH; SAVE MTOP
        MOVX    @DPTR,A
        INC     DPTR            ; NOW, SAVE THE LOW BYTE
        MOV     A,#LOW 1FFFH
        MOVX    @DPTR,A
        ;
        ; NOW SET UP THE VARTOP POINTER, WITH NO STRINGS,
        ; VARTOP = MEMTOP
        ;
        MOV     DPTR,#104H      ; LOCATION OF VARTOP IN EXTERNAL RAM
        MOV     A,#HIGH 1FFFH
        MOVX    @DPTR,A
        INC     DPTR
        MOV     A,#LOW 1FFFH
        MOVX    @DPTR,A
        ;
        ;
        ; NOW SAVE THE MATRIX POINTER "DIMUSE", THIS POINTER IS
        ; DESCRIBED IN THE APPENDIX, WITH NO PROGRAM IN RAM,
        ; DIMUSE = 528 AFTER RESET
        ;
        MOV     DPTR,#108H      ; LOCATION OF DIMUSE IN EXTERNAL RAM
        MOV     A,#HIGH 528
        MOVX    @DPTR,A
        INC     DPTR
        MOV     A,#LOW 528
        MOVX    @DPTR,A
        ;
        ; NOW SAVE THE VARIABLE POINTER "VARUSE" THIS POINTER IS
        ; ALSO DESCRIBED IN THE APPENDIX, AFTER RESET VARUSE = VARTOP
        ;
        MOV     DPTR,#106H      ; LOCATION OF VARUSE IN EXTERNAL RAM
        MOV     A,#HIGH 1FFFH
        MOVX    @DPTR,A
        INC     DPTR
        MOV     A,#LOW 1FFFH
        MOVX    @DPTR,A
        ;
        ; NOW SETUP BASICS CONTROL STACK AND ARGUMENT STACK
        ;
        MOV     9H,#0FEH        ; THIS INITIALIZES THE ARGUMENT STACK
        MOV     11H,#0FEH       ; THIS INITIALIZES THE CONTROL STACK
        ;
        ; NOW TELL BASIC THAT NO PROGRAM IS IN RAM, THIS IS NOT NEEDED
        ; IF THE USER HAS A PROGRAM IN RAM
        ;
        MOV     DPTR,#512       ; LOCATION OF THE START OF A USER PROGRAM
        MOV     A,#01H          ; END OF FILE CHARACTER
        MOVX    @DPTR,A
```

RESET OPTIONS (VERSION 1.1 ONLY)

```
        ;
        ; NOW PUSH THE CRYSTAL VALUE ON TO THE STACK AND LET BASIC
        ; CALCULATE ALL CRYSTAL DEPENDENT PARAMETERS
        ;
        SJMP    RESET3
        ;
XTAL:   DB      88H             ; THIS IS THE FLOATING POINT VALUE
        DB      00H             ; FOR AN 11.0592 MHZ CRYSTAL
        DB      00H
        DB      92H
        DB      05H
        DB      11H
        ;
        ;
RESET3: MOV     DPTR,#XTAL      ; SET UP TO PUSH CRYSTAL VALUE
        MOV     A,9             ; GET THE ARG STACK
        CLR     C
        SUBB    A,#6            ; DECREMENT ARG STACK BY ONE FP NUMBER
        MOV     9,A
        MOV     R0,A            ; SAVE THE CALCULATED ADDRESS IN R0
        MOV     P2,#1           ; THIS IS THE ARG STACK PAGE ADDRESS
        MOV     R1,#6           ; NUMBER OF BYTES TO TRANSFER
        ;
RESET4: CLR     A               ; TRANSFER ROM CRYSTAL VALUE TO THE
        MOVC    A,@A+DPTR       ; ARGUMENT STACK OF BASIC
        MOVX    @R0,A
        INC     DPTR            ; BUMP THE POINTERS
        DEC     R0
        DJNZ    R1,RESET4       ; LOOP UNTIL THE TRANSFER IS COMPLETE
        ;
        ; NOW CALL BASIC TO DO ALL THE CRYSTAL CACULATIONS
        ;
        MOV     A,#58           ; OPBYTE FOR CRYSTAL CALCULATION
        LCALL   30H             ; DO THE CALCULATION
        ;
        ; NOW TELL BASIC WHERE START OF THE USER BASIC PROGRAM IS
        ; BY LOADING THE START ADDRESS, IF THE PROGRAM IS IN EPROM
        ; 13H WOULD = HIGH 8011H AND 14H = LOW 8011H, ANYWAY
        ; ADDRESS 13H:14H MUST POINT TO THE START OF THE BASIC
        ; PROGRAM
        ;
        MOV     13H,#HIGH 512; THIS TELLS BASIC THAT THE START OF
        MOV     14H,#LOW 512 ; THE PROGRAM IS IN LOCATION 512
        ;
        ; NOW THE SERIAL PORT MUST BE INITIALIZED, THE USER
        ; CAN SET UP THE SERIAL PORT TO ANY DESIRED CONFIGURATION
        ; HOWEVER, THIS DEMO CODE WILL SHOW THE AUTO BAUD
        ; ROUTINE
        ;
        MOV     R3,#00H         ; INITIALIZE THE AUTO BAUD COUNTERS
        MOV     R1,#00H
        MOV     R0,#04H
        JB      RXD,$           ; LOOP UNTIL A START BIT IS RECEIVED
        ;
```

RESET OPTIONS (VERSION 1.1 ONLY)

```
;
RESET5: DJNZ    R0,$            ; WASTE 8 CLOCKS INITIALLY, SIX CLOCKS
                                ; IN THE LOOP (16) TOTAL
        CLR     C               ; 1 CLOCK (1)
        MOV     A,R1            ; 1 CLOCK (2)
        SUBB    A,#1            ; 1 CLOCK (3)
        MOV     R1,A            ; 1 CLOCK (4)
        MOV     A,R3            ; 1 CLOCK (5)
        SUBB    A,#00H          ; 1 CLOCK -- R3:R1 = R3:R1 - 1 (6)
        MOV     R3,A            ; 1 CLOCK (7)
        MOV     R0,#3           ; 1 CLOCK (8)
        JNB     RXD,RESET5      ; 2 CLOCKS (10), WAIT FOR END OF SPACE
        JB      RXD,$           ; WAIT FOR THE SPACE TO END (20H)
        JNB     RXD,$           ; WAIT FOR THE STOP BIT
        MOV     RCAP2H,R3       ; LOAD THE TIMER 2 HOLDING REGISTERS
        MOV     RCAP2L,R1
        ;
        ; NOW YOU CAN ADD A CUSTOM SIGN ON MESSAGE
        ;
        MOV     R3,#HIGH MSG ; PUT ADDRESS OF MESSAGE IN R3:R1
        MOV     R1,#LOW MSG
        SETB    52              ; PRINT FROM ROM
        MOV     A,#6            ; OP BYTE TO PRINT TEXT STRING
        LCALL   30H
        ;
        ; NOW OUTPUT A CR LF
        ;
        MOV     A,#7            ; OP BYTE FOR CRLF
        LCALL   30H
        ;
        ; GO TO THE COMMAND MODE
        ;
        CLR     A
        JMP     30H
        ;
MSG:    DB      'CUSTOM SIGN ON MESSAGE'
        DB      22H             ; TERMINATES MESSAGE
        ;
        END
```

RESET OPTIONS (VERSION 1.1 ONLY)

To Summarize what the user must do to successfully implement a "COLD START" RESET:

1) The user must clear the internal RAM of the MCS BASIC-52 device and at least the first 512 bytes of external RAM memory.

2) The user must initialize the stack pointer (special function register — SP) and the stack pointer holding register (internal RAM location 3EH) with a value that is between 4DH and 0E0H. 4DH gives MCS BASIC-52 the maximum stack size.

3) The user must initialize the following pointers in external RAM. MTOP at location 10AH (high byte) and 10BH (low byte). VARTOP at locations 104H (high byte) and 105H (low byte). DIMUSE at locations 108H (high byte) and 109H (low byte). VARUSE at locations 106H (high byte) and 107H (low byte). Details of what needs to be in these locations are presented in appendix 1.7 of this manual.

4) The Control stack pointer (location 11H in internal memory) and the Argument stack pointer (location 09H in internal memory) must also be initialized with the value 0FEH. If the user is not going to assign the XTAL (crystal) value in BASIC, then the XTAL value must be pushed onto the argument stack and the user must to an OPBYTE 58 call to MCS BASIC-52.

5) The User must also initialize the start address of a program. The start address is in locations 13H (high byte) and 14H (low byte) of internal data memory. If the user BASIC program is in RAM, then 13H: 14H = 512, if the user program is the the first program in EPROM, then 13H: 14H = 8011H.

6) The user must finally initialize the serial port. Any scheme can be used (as long as it works!!)

The added reset options should go a long way toward making MCS BASIC-52 configurable to any custom application.

CHAPTER 12
Command/Statement Extensions (Version 1.1 Only)

MCS BASIC-52 V1.1 provides a simple, but yet effective way for the user to add COMMANDS and/or STATEMENTS to the ones that are provided on the chip. All the user must do is write a few simple programs that will reside in external code memory. The step by step approach is as follows:

STEP 1

The user must first inform the MCS BASIC-52 device that the expansion options are available. This is done by putting the character 5AH in CODE memory location 2002H. When MCS BASIC-52 enters the command mode it will examine CODE memory location 2002H. If a 5AH is in this location, MCS BASIC-52 will CALL external CODE memory location 2048H. The user must then write a short routine to SET BIT 45 (2DH), which is bit 5 of internal memory location 37 (decimal) and place this routine at code memory location 2048H. Setting BIT 45 tells MCS BASIC-52 that the expansion option is available. The following simple code will accomplish all that is stated above:

```
ORG   2002H
DB    5AH
;
OG    2048H
SETB  45
RET
```

STEP 2

With BIT 45 SET, MCS BASIC-52 will CALL external CODE memory location 2078H everytime it attempts to tokenize a line that has been entered. At location 2078H, the user must load the DPTR (Data Pointer) with the address of the user supplied lookup table, complete with tokens.

COMMAND/STATEMENT EXTENSIONS (VERSION 1.1 ONLY)

STEP 3

The user needs the following information to generate a user token table:

1) THE USER TOKENS ARE THE NUMBRES 10H THROUGH 1FH (16 TOKENS AVAILABLE)

2) THE USER TOKEN TABLE BEGINS WITH THE TOKEN, FOLLOWED BY THE ASCII TEXT THAT IS TO BE REPRESENTED BY THAT TOKEN, FOLLOWED BY A ZERO (00H) INDICATING THE END OF THE ASCII, FOLLOWED BY THE NEXT TOKEN.

3) THE TABLE IS TERMINATED WITH THE CHARACTER 0FFH.

EXAMPLE:

```
            ORG     2078H
            ;
            MOV     DPTR,#USER_TABLE
            RET
            ;
            ORG     2200H           ;THIS DOES NOT NEED TO BE
            ;                       ;IN THIS LOCATION
USER_TABLE:
            ;
            DB      10H             ;FIRST TOKEN
            DB      'DISPLAY'       ;USER KEYWORD
            DB      00H             ;KEYWORD TERMINATOR
            ;
            DB      11H             ;SECOND TOKEN
            DB      'TRANSFER'      ;SECOND USER KEYWORD
            DB      00H             ;KEYWORD TERMINATOR
            ;
            DB      12H             ;THIRD TOKEN (UP TO 16)
            DB      'ROTATE'        ;THIRD USER KEYWORD
            DB      0FFH            ;END OF USER TABLE
```

This same user table is used when MCS BASIC-52 "de-tokenizes" a line during a LIST.

COMMAND/STATEMENT EXTENSIONS (VERSION 1.1 ONLY)

STEP 4

Step 3 tokenizes the user keyword, this means that MCS BASIC-52 translates the user keyword into the user token. So, in the preceding example, the keyword TRANSFER would be replaced with the token 11H. When MCS BASIC-52 attempts to execute the user token, it first makes sure that the user expansion option BIT is set (BIT 45), then CALLS location 2070H to get the address of the user vector table. This address is placed in the DPTR. The user vector table consist of series of Data Words that define the address of the user assembly language routines.

EXAMPLE:

```
                ORG     2070H           ;LOCATION BASIC CALLS TO
                                        ;GET USER LOOKUP
                ;
                MOV     DPTR,#VECTOR_TABLE
                RET
                ;
VECTOR_TABLE:
                ;
                DW      RUN_DISPLAY     ;ADDRESS OF DISPLAY
                                        ;ROUTINE, TOKEN (10H)
                DW      RUN_TRANSFER    ;ADDRESS OF TRANSFER
                                        ;ROUTINE, TOKEN (11H)
                DW      RUN_ROTATE      ;ADDRESS OF ROTATE
                                        ;ROUTINE, TOKEN (12H)
                ;
                ORG     2300H           ;AGAIN, THESE ROUTINES
                                        ;MAY BE PLACED ANYWHERE
                ;
 RUN_DISPLAY:
                ;
                ; USER ASM CODE FOR DISPLAY GOES HERE
                ;
RUN_TRANSFER:
                ;
                ; USER ASM CODE FOR TRANSFER GOES HERE
                ;
  RUN_ROTATE:
                ;
                ;USER ASM CODE FOR ROTATE GOES HERE
                ;
```

COMMAND/STATEMENT EXTENSIONS (VERSION 1.1 ONLY)

Note that the ordinal position of the DATA WORDS in the user vector table must correspond to the token, so the user statement with the token 10H must be the first DW entry in the vector table, 11H, the second, 12H, the third, and so on. The order of the tokens in the user table is not important!! The following user lookup table would function properly with the previous example:

EXAMPLE:

```
;
USER_TABLE:
;
          DB      13H             ; THE TOKENS DO NOT HAVE
          DB      'ROTATE'        ; TO BE IN ORDER IN THE
          DB      00H             ; USER LOOKUP TABLE
;
          DB      10H
          DB      'DISPLAY'
          DB      00H
;
          DB      12H
          DB      'TRANSFER'
          DB      0FFH            ; END OF TABLE
```

COMMAND/STATEMENT EXTENSIONS (VERSION 1.1 ONLY)

The user may also use the command/statement extension option to re-define the syntax of MCS BASIC-52. This is done simply by placing your own syntax in the user table and placing the appropriate BASIC token in front of your re-defined keyword. A complete listing of all MCS BASIC-52 tokens and keywords are provided in the back of this chapter. MCS BASIC-52 will always list out the program using the user defined systax, but it will still accept the standard keyword as a valid instruction. As an example, suppose that the user would like to substitute the keyword HEXOUT for PH1., then the user would generate the following entry in the user table:

EXAMPLE:

```
USER_TABLE:
            ;
            DB      8FH             ;TOKEN FOR PH1.
            DB      'HEXOUT'        ;TO BE IN ORDER IN THE
            DB      00H             ;USER LOOKUP TABLE
            ;
            DB      10H
            DB      'DISPLAY'
            DB      00H
            ;
            ;       REST OF USER_TABLE
            ;
            DB      0FFH            ;END OF TABLE
```

MCS BASIC-52 will now accept the keyword HEXOUT and it will function in a manner identical to PH1. PH1. will still function correctly, however HEXOUT will be displayed when the user LIST a program.

COMMAND/STATEMENT EXTENSIONS (VERSION 1.1 ONLY)

TOKEN	KEYWORD	TOKEN	KEYWORD	TOKEN	KEYWORD
80H	LET	0B0H	ABS	0ECH	<=
81H	CLEAR	0B1H	INT	0EDH	<>
82H	PUSH	0B2H	SGN	0EEH	<
83H	GOTO	0B3H	NOT	0EFH	>
84H	PWM	0B4H	COS	0F0H	RUN
85H	PHO.	0B5H	TAN	0F1H	LIST
86H	UI	0B6H	SIN	0F2H	NULL
87H	UO	0B7H	SQR	0F3H	NEW
88H	POP	0B8H	CBY	0F4H	CONT
89H	PRINT	0B9H	EXP	0F5H	PROG
89H	P.	0BAH	ATN	0F6H	XFER
89H	? (V1.1 ONLY)	0BBH	LOG	0F7H	RAM
8AH	CALL	0BCH	DBY	0F8H	ROM
8BH	DIM	0BDH	XBY	0F9H	FPROG
8CH	STRING	0BEH	PI	0FAH-0FFH	NOT USED
8DH	BAUD	0BFH	RND		
8EH	CLOCK	0C0H	GET		
8FH	PH1.	0C1H	FREE		
90H	STOP	0C2H	LEN		
91H	ONTIME	0C3H	XTAL		
92H	ONEX1	0C4H	MTOP		
93H	RETI	0C5H	TIME		
94H	DO	0C6H	IE		
95H	RESTORE	0C7H	IP		
96H	REM	0C8H	TIMER0		
97H	NEXT	0C9H	TIMER1		
98H	ONERR	0CAH	TIMER2		
99H	ON	0CBH	T2CON		
9AH	INPUT	0CCH	TCON		
9BH	READ	0CDH	TMOD		
9CH	DATA	0CEH	RCAP2		
9DH	RETURN	0CFH	PORT1		
9EH	IF	0D0H	PCON		
9FH	GOSUB	0D1H	ASC(		
0A0H	FOR	0D2H	USING(		
0A1H	WHILE	0D2H	U. (		
0A2H	UNTIL	0D3H	CHR(		
0A3H	END	0D4H-0DFH	NOT USED		
0A4H	TAB	0E0H	(		
0A5H	THEN	0E1H	**		
0A6H	TO	0E2H	*		
0A7H	STEP	0E3H	+		
0A8H	ELSE	0E4H	/		
0A9H	SPC	0E5H	-		
0AAH	CR	0E6H	.XOR.		
0ABH	IDLE	0E7H	.AND.		
0ACH	ST@ (V1.1 ONLY)	0E8H	.OR.		
0ADH	LD@ (V1.1 ONLY)	0E9H	- (NEGATE)		
0AEH	PGM (V1.1 ONLY)	0EAH	=		
0AFH	RROM(V1.1 ONLY)	0EBH	>=		

CHAPTER 13
Mapping User Code Memory

You might have noticed by now that some of external CODE memory locations that MCS BASIC-52 calls and uses are located around 2000H and some of the locations are located around 4000H. Specifically, they are as follows:

LOCATION	FUNCTION
2001H	ON RESET, MCS BASIC-52 LOOKS FOR A 0AAH IN THIS LOCATION, IF PRESENT, CALLS LOCATION 2090H
2002H	MCS BASIC-52 EXAMINES THIS LOCATION TO SEE IF THE USER WANTS TO IMPLEMENT THE COMMAND/STATEMENT EXTENSION OPTION, A 05AH IS TO BE PLACED IN THIS LOCATION TO EVOKE THE COMMAND/EXTENSION OPTION
2048H	MCS BASIC-52 CALLS THE LOCATION IF THE USER WANTS TO IMPLEMENT THE COMMAND/STATEMENT EXTENSION OPTION. THE USER WILL USUALLY SET BIT 45 THEN RETURN.
2070H	MCS BASIC-52 CALLS THIS LOCATION TO GET THE USER VECTOR TABLE ADDRESS WHEN THE COMMAND/STATEMENT EXTENSION OPTION IS EVOKED. THE ADDRESS OF THE VECTOR TABLE IS PUT IN THE DPTR BY THE USER.
2078H	MCS BASIC-52 CALLS THIS LOCATION TO GET THE USER LOOKUP TABLE ADDRESS WHEN THE COMMAND/STATEMENT EXTENSION OPTION IS EVOKED. THE ADDRESS OF THE LOOKUP TABLE IS PUT IN THE DPTR BY THE USER.
2090H	MCS BASIC-52 CALLS THIS LOCATION WHEN THE USER EVOKES THE ASSEMBLY LANGUAGE RESET OPTION
4003H	EXTERNAL INTERRUPT 0
400BH	TIMER O INTERRUPT
4013H	EXTERNAL INTERRUPT 1
401BH	TIMER 0 INTERRUPT
4023H	SERIAL PORT INTERRUPT
402BH	TIMER 2 INTERRUPT
4030H	USER CONSOLE OUTPUT
4033H	USER CONSOLE INPUT
4036H	USER CONSOLE STATUS
403CH	USER PRINT@ OR LIST@ VECTOR
4100H–41FFH	USER CALLS FORM 0 TO 7FH

MAPPING USER CODE MEMORY

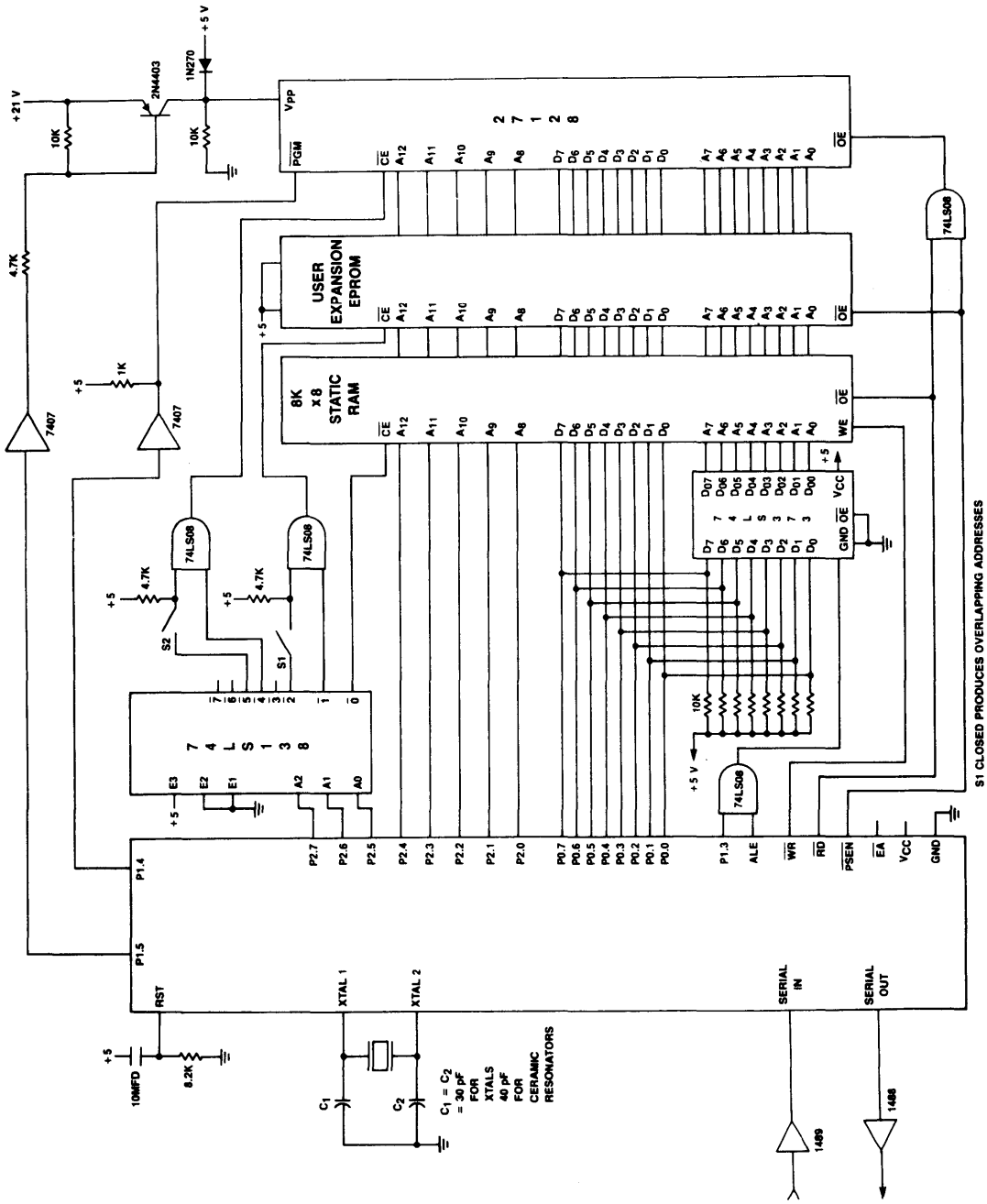
Other vectors between 2040H and 2090H also exist, but they are mainly for testing purposes, but for your information they are:

LOCATION	FUNCTION
2040H	TRAP LOCATION FOR EXTERNAL INTERRUPT 0 IF BIT 26H OF INTERNAL RAM IS SET AND THE DMA OPTION IS EVOKED. PSW IS NOT PUSHED ONTO STACK. INTERRUPTS OF COURSE, MUST BE ENABLED. ALSO, THIS LOCATION WILL BE CALLED FOR CONSOLE OUTPUT IF BIT 2CH OF INTERNAL RAM IS SET.
2050H	TRAP LOCATION FOR SERIAL PORT INTERRUPT IF BIT 1FH OF INTERNAL RAM IS SET. PSW IS PUSHED ONTO THE STACK.
2060H	CALLED FOR CONSOLE INPUT IF BIT 32H OF INTERNAL RAM IS SET.
2068H	CALLED FOR CONSOLE STATUS CHECK IF BIT 32H OF INTERNAL RAM IS SET.
2088H	TIMER 1 INTERRUPT TRAP IF BIT 1AH OF INTERNAL RAM IS SET. PSW IS PUSHED ONTO THE STACK.

Contrary to popular belief, these vectors were not chosen to force the user to buy bigger EPROMS. They are chosen so that addresses 2000H and 4000H can be overlayed and create no conflicts. The Overlayed addresses would appear as 2001H, 2002H, 4003H, 400BH, 4013H, 401BH, 4023H, 402BH, 4030H, 4033H, 4036H, 4039H, 2040H, 2048H, 2050H, 2060H, 2068H, 2070H, 2078H, 2088H, 2090H, and 4100H thru 41FFH. The diagram on the next page illustrates how to implement overlapping addresses for 2000H and 4000H. By using overlapping addresses, the user can implement all MCS BASIC-52 user expansion options with only a few hundred bytes of EPROM.

The reason this type of addressing scheme was chosen is that it permits the designer to offer custom versions of MCS BASIC-52, by using the vector locations in the 2000H region. And give the designers OEM the ability to take advantage of the I/O vectors located in the 4000H region.

As an added note, the MCS-51 instruction set is object relocatable on 2K boundaries if no LCALL or LJMP instructions are used. This means that it is possible for the designer to ORG a program for 2000H and actually execute the program at 2800H, 3000H, 3800H, etc. If the user does not use the LCALL or LJMP instructions.



Overlapping user EPROM address space

S1 CLOSED PRODUCES OVERLAPPING ADDRESSES
S2 CLOSED PERMITS 27128 TO BE AT ADDRESS 8000H. A13 MUST BE CONNECTED ON 27128

APPENDIX A

1.1 MEMORY USAGE (Version 1.0)

The following list specifies what locations in internal and external memory MCS BASIC-52 uses, and what these locations are used for. This information can largely be regarded as "for your information," but it can be used to do things like alter the pulse width of a EPROM programming pulse, etc.

INTERNAL MEMORY ALLOCATION:

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
00H THRU 07H	"WORKING REGISTER BANK"
08H	BASIC TEXT POINTER — LOW BYTE
09H	ARGUMENT STACK POINTER
0AH	BASIC TEXT POINTER — HIGH BYTE
0BH THRU 0FH	TEMPORARY BASIC STORAGE
10H	READ TEXT POINTER — LOW BYTE
11H	CONTROL STACK POINTER
12H	READ TEXT POINTER — HIGH BYTE
13H	START ADDRESS OF BASIC PROGRAM — HIGH BYTE
14H	START ADDRESS OF BASIC PROGRAM — LOW BYTE
15H	NULL COUNT
16H	PRINT HEAD POSITION FOR OUTPUT
17H	FLOATING POINT OUTPUT FORMAT TYPE
18H THRU 21H	NOT USED — RESERVED FOR USER
22H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 22.0H	SET WHEN "ONTIME" STATEMENT IS EXECUTED
BIT 22.1H	SET WHEN BASIC INTERRUPT IN PROGRESS
BIT 22.2H	SET WHEN "ONEX1" STATEMENT IS EXECUTED
BIT 22.3H	SET WHEN "ONERR" STATEMENT IS EXECUTED
BIT 22.4H	SET WHEN "ONTIME" INTERRUPT IS IN PROGRESS
BIT 22.5H	SET WHEN A LINE IS EDITED
BIT 22.6H	SET WHEN EXTERNAL INTERRUPT IS PENDING
BIT 22.7H	WHEN SET, CONT COMMAND WILL WORK
23H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 23.0H	USED AS FLAG FOR "GET" OPERATOR
BIT 23.1H	SET WHEN INVALID INTEGER FOUND IN TEXT
BIT 23.2H	TEMPORARY BIT LOCATION
BIT 23.3H	CONSOLE OUTPUT CONTROL, 1 = LINE PRINTER
BIT 23.4H	CONSOLE OUTPUT CONTROL, 1 = USER DEFINED
BIT 23.5H	BASIC ARRAY INITIALIZATION BIT
BIT 23.6H	CONSOLE INPUT CONTROL, 1 = USER DEFINED
BIT 23.7H	RESERVED

1.1 MEMORY USAGE

INTERNAL MEMORY ALLOCATION:

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
24H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 24.0H	STOP STATEMENT OR CONTROL-C ENCOUNTERED
BIT 24.1H	0 = HEX INPUT, 1 = FP INPUT
BIT 24.2H	0 = RAM MODE, 1 = ROM MODE
BIT 24.3H	ZERO FLAG FOR DOUBLE BYTE COMPARE
BIT 24.4H	SET WHEN ARGUMENT STACK HAS A VALUE
BIT 24.5H	RETI INSTRUCTION EXECUTED
BIT 24.6H	RESERVED
BIT 24.7H	RESERVED
25H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 25.0H	RESERVED, SOFTWARE TRAP TEST
BIT 25.1H	FIND THE END OF PROGRAM, IF SET
BIT 25.2H	RESERVED
BIT 25.3H	INTERRUPT STATUS SAVE BIT
BIT 25.4H	SET WHEN PROGRAM EXECUTION IS COMPLETE
BIT 25.5H	RESERVED, EXTERNAL TRAP TEST
BIT 25.6H	SET WHEN CLOCK1 EXECUTED, ELSE CLEARED
BIT 25.7H	SET WHEN BASIC IS IN THE COMMAND MODE
26H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 26.0H	SET TO DISABLE CONTROL-C
BIT 26.1H	SET TO ENABLE "FAKE" DMA
BIT 26.2H	RESERVED
BIT 26.3H	SET TO EVOKE "INTELLIGENT" PROM PROGRAMMING
BIT 26.4H	SET TO PRINT TEXT STRING FROM ROM
BIT 26.5H	RESERVED
BIT 26.6H	SET TO SUPPRESS ZEROS IN HEX MODE PRINT
BIT 26.7H	SET TO EVOKE HEX MODE PRINT

1.1 MEMORY USAGE

INTERNAL MEMORY ALLOCATION:

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
27H	"BIT" ADDRESSABLE BYTE COUNTER
28H THRU 3DH	BIT AND BYTE FLOATING POINT WORKING SPACE
3EH	INTERNAL STACK POINTER HOLDING REGISTER
3FH	LENGTH OF USER DEFINED STRING — $
40H	TIMER 1 RELOAD LOCATION — HIGH BYTE
41H	TIMER 1 RELOAD LOCATION — LOW BYTE
42H	BASIC TEXT POINTER SAVE LOCATION — HIGH BYTE
43H	BASIC TEXT POINTER SAVE LOCATION — LOW BYTE
44H	RESERVED
45H	TRANSCENDENTAL FUNCTION TEMP STORAGE
46H	TRANSCENDENTAL FUNCTION TEMP STORAGE
47H	MILLI-SECOND COUNTER FOR REAL TIME CLOCK
48H	SECOND COUNTER FOR REAL TIME CLOCK — HIGH BYTE
49H	SECOND COUNTER FOR REAL TIME CLOCK — LOW BYTE
4AH	TIMER 0 RELOAD FOR REAL TIME CLOCK
4BH	SOFTWARE SERIAL PORT BAUD RATE — HIGH BYTE
4CH	SOFTWARE SERIAL PORT BAUD RATE — LOW BYTE
4DH THRU 0FFH	8052AH STACK SPACE AND USER WORKING SPACE

1.1 MEMORY USAGE

EXTERNAL MEMORY ALLOCATION

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
00H AND 01H	"LAST" END OF FILE ADDRESS FOR RAM FILE (H-L)
02H AND 03H	CURRENT END OR FILE ADDRESS FOR RAM FILE (H-L)
04H	LENGTH OF THE CURRENT EDITED LINE
05H AND 06H	LN NUM IN BINARY OF CURRENT EDITED LINE (H-L)
07H THRU 49H	BASIC INPUT BUFFER
50H THRU 5FH	FLOATING POINT OUTPUT TEMP
60H THRU 0FEH	CONTROL STACK
0FFH	CONTROL STACK OVERFLOW
100H	LOCATION TO SAVE "GET" CHARACTER
101H	LOCATION TO SAVE ERROR CHARACTER CODE
102H AND 103H	LOCATION TO GO TO ON USER "ONERR" (H-L)
104H AND 105H	TOP OF VARIABLE STORAGE (H-L)
106H AND 107H	FP STORAGE ALLOCATION (H-L)
108H AND 109H	MEMORY ALLOCATED FOR MATRICES (H-L)
10AH AND 10BH	TOP OF MEMORY ASSIGNED TO BASIC (H-L)
10CH AND 10DH	RANDOM NUMBER SEED (H-L)
10EH THRU 113H	CRYSTAL VALUE
114H THRU 11FH	FLOATING POINT TEMPS
120H AND 121H	LOCATION TO GO TO ON ONEX1 INTERRUPT (H-L)
122H AND 123H	NUMBER OF BYTES ALLOCATED FOR STRINGS (H-L)
124H THRU 127H	ONTIME INTERRUPT AND LINE NUMBER (H-L)
128H AND 129H	"NORMAL" PROM PROGRAMMER TIME OUT (H-L)
12AH AND 12BH	"INTELLIGENT" PROM PROGRAMMER TIME OUT (H-L)
12CH	RESERVED
12DH THRU 1FEH	ARGUMENT STACK

NOTE: (H-L) means HIGH BYTE — LOW BYTE, in external memory all 16 bit binary numbers are stored with the HIGH BYTE in the first (lower order) address and the LOW BYTE in the next sequential address.

1.1 MEMORY USAGE (VERSION 1.1)

The following list specifies what locations in internal and external memory locations are used by Version 1.1 of MCS BASIC-52. Any differences between V1.0 and V1.1 are **in bold face type.**

INTERNAL MEMORY ALLOCATION: (VERSION 1.1)

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
00H THRU 07H	"WORKING REGISTER BANK"
08H	BASIC TEXT POINTER — LOW BYTE
09H	ARGUMENT STACK POINTER
0AH	BASIC TEXT POINTER — HIGH BYTE
0BH THRU 0FH	TEMPORARY BASIC STORAGE (Available to user in BASIC CALLS to ASM routines)
10H	READ TEXT POINTER — LOW BYTE
11H	CONTROL STACK POINTER
12H	READ TEXT POINTER — HIGH BYTE
13H	START ADDRESS OF BASIC PROGRAM — HIGH BYTE
14H	START ADDRESS OF BASIC PROGRAM — LOW BYTE
15H	NULL COUNT
16H	PRINT HEAD POSITION FOR OUTPUT
17H	FLOATING POINT OUTPUT FORMAT TYPE
18H THRU 21H	NOT USED — RESERVED FOR USER
22H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 22.0H	SET WHEN "ONTIME" STATEMENT IS EXECUTED
BIT 22.1H	SET WHEN BASIC INTERRUPT IN PROGRESS
BIT 22.2H	SET WHEN "ONEX1" STATEMENT IS EXECUTED
BIT 22.3H	SET WHEN "ONERR" STATEMENT IS EXECUTED
BIT 22.4H	SET WHEN "ONTIME" INTERRUPT IS IN PROGRESS
BIT 22.5H	SET WHEN A LINE IS EDITED
BIT 22.6H	SET WHEN EXTERNAL INTERRUPT IS PENDING
BIT 22.7H	WHEN SET, CONT COMMAND WILL WORK
23H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 23.0H	USED AS FLAG FOR "GET" OPERATOR
BIT 23.1H	**SET WHEN PRINT@ OR LIST@ IS EVOKED**
BIT 23.2H	**RESERVED, TRAPS TIMER 1 INTERRUPT**
BIT 23.3H	CONSOLE OUTPUT CONTROL, 1 = LINE PRINTER
BIT 23.4H	CONSOLE OUTPUT CONTROL, 1 = USER DEFINED
BIT 23.5H	BASIC ARRAY INITIALIZATION BIT
BIT 23.6H	CONSOLE INPUT CONTROL, 1 = USER DEFINED
BIT 23.7H	**RESERVED, USED TO TRAP SERIAL PORT INTERRUPT**

INTERNAL MEMORY ALLOCATION (VERSION 1.1)

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
24H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 24.0H	STOP STATEMENT OR CONTROL-C ENCOUNTERED
BIT 24.1H	**USER IDLE BREAK BIT**
BIT 24.2H	**SET DURING AN INPUT INSTRUCTION**
BIT 24.3H	**RESERVED**
BIT 24.4H	SET WHEN ARGUMENT STACK HAS A VALUE
BIT 24.5H	RETI INSTRUCTION EXECUTED
BIT 24.6H	**RESERVED, TRAPS EXTERNAL INTERRUPT 0**
BIT 24.7H	**SET BY USER TO SIGNIFY THAT A VALID LIST@ OR PRINT@ DRIVER IS PRESENT**
25H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 25.0H	RESERVED, SOFTWARE TRAP TEST
BIT 25.1H	FIND THE END OF PROGRAM, IF SET
BIT 25.2H	**SET DURING A DIM STATEMENT**
BIT 25.3H	INTERRUPT STATUS SAVE BIT
BIT 25.4H	**RESERVED, INPUT TRAP**
BIT 25.5H	**SET TO SIGNIFY EXPANSION IS PRESENT**
BIT 25.6H	SET WHEN CLOCK1 EXECUTED, ELSE CLEARED
BIT 25.7H	SET WHEN BASIC IS IN THE COMMAND MODE
26H	BITS USED SPECIFICALLY AS FOLLOWS
BIT 26.0H	SET TO DISABLE CONTROL-C
BIT 26.1H	SET TO ENABLE "FAKE" DMA
BIT 26.2H	**RESERVED, OUTPUT TRAP**
BIT 26.3H	SET TO EVOKE "INTELLIGENT" PROM PROGRAMMING
BIT 26.4H	SET TO PRINT TEXT STRING FROM ROM
BIT 26.5H	**SET WHEN CONTROL-S ENCOUNTERED**
BIT 26.6H	SET TO SUPPRESS ZEROS IN HEX MODE PRINT
BIT 26.7H	SET EVOKE HEX MODE PRINT

INTERNAL MEMORY ALLOCATION (VERSION 1.1)

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
27H	"BIT" ADDRESSABLE BYTE COUNTER
28H THRU 3DH	BIT AND BYTE FLOATING POINT WORKING SPACE
3EH	INTERNAL STACK POINTER HOLDING REGISTER
3FH	LENGTH OF USER DEFINED STRING — $
40H	TIMER 1 RELOAD LOCATION — HIGH BYTE
41H	TIMER 1 RELOAD LOCATION — LOW BYTE
42H	BASIC TEXT POINTER SAVE LOCATION — HIGH BYTE
43H	BASIC TEXT POINTER SAVE LOCATION — LOW BYTE
44H	RESERVED
45H	TRANCENDENTAL FUNCTION TEMP STORAGE
46H	TRANCENDENTAL FUNCTION TEMP STORAGE
47H	MILLI-SECOND COUNTER FOR REAL TIME CLOCK
48H	SECOND COUNTER FOR REAL TIME CLOCK — HIGH BYTE
49H	SECOND COUNTER FOR REAL TIME CLOCK — LOW BYTE
4AH	TIMER 0 RELOAD FOR REAL TIME CLOCK
4BH	**USER ARGUMENT FOR ONTIME — HIGH BYTE**
4CH	**USER ARGUMENT FOR ONTIME — LOW BYTE**
4DH THRU 0FFH	8052AH STACK SPACE AND USER WORKING SPACE

EXTERNAL MEMORY ALLOCATION (VERSION 1.1)

LOCATION(S) IN HEX	MCS BASIC-52 USAGE
00H THRU 03H	**NOT USED, RESERVED**
04H	LENGTH OF THE CURRENT EDITED LINE
05H AND 06H	LN NUM IN BINARY OF CURRENT EDITED LINE (H–L)
07H THRU 56H	**BASIC INPUT BUFFER**
56H THRU 5DH	**BINARY TO INTEGER TEMP**
5EH	**USED FOR RUN TRAP MODE (= 34H)**
5FH	**USED FOR POWER-UP TRAP (= 0A5H)**
60H THRU 0FEH	CONTROL STACK
00FH	CONTROL STACK OVERFLOW
100H	LOCATION TO SAVE "GET" CHARACTER
101H	LOCATION TO SAVE ERROR CHARACTER CODE
102H AND 103H	LOCATION TO GO TO ON USER "ONERR" (H–L)
104H AND 105H	TOP OF VARIABLE STORAGE (H–L)
106H AND 107H	FP STORAGE ALLOCATION (H–L)
108H AND 109H	MEMORY ALLOCATED FOR MATRICIES (H–L)
10AH AND 10BH	TOP OF MEMORY ASSIGNED TO BASIC (H–L)
10CH AND 10DH	RANDOM NUMBER SEED (H–L)
10EH THRU 113H	CRYSTAL VALUE
114H THRU 11FH	FLOATING POINT TEMPS
120H AND 121H	LOCATION TO GO TO ON ONEX1 INTERRUPT (H–L)
122H AND 123H	NUMBER OF BYTES ALLOCATED FOR STRINGS (H–L)
124H AND 125H	**SOFTWARE SERIAL PORT BAUD RATE (H–L)**
126H AND 127H	**LINE NUMBER FOR ONTIME INTERRUPT (H–L)**
128H AND 129H	"NORMAL" PROM PROGRAMMER TIME OUT (H–L)
12AH AND 12BH	"INTELLIGENT" PROM PROGRAMMER TIME OUT (H–L)
12CH	RESERVED
12DH THRU 1FEH	ARGUMENT STACK

NOTE: (H–L) still means HIGH BYTE — LOW BYTE, in external memory all 16 bit binary numbers are stored with the HIGH BYTE in the first (lower order) address and the LOW BYTE in the next sequential address.

1.2 USING THE PWM STATEMENT

The PWM statement can be used to generate quite accurate frequencies. The following table lists the reload values 8 octaves of an equal tempered chromatic scale. The reload values are for the first two arguments of the PWM statement, so it is assumed that a square wave is being generated. The reload values assume a 11.0592 MHz crystal.

NOTE	OCTAVE	IDEAL FREQUENCY	ACTUAL FREQUENCY	RELOAD	HEX RELOAD
C	1	32.703	32.704	14090	370AH
C#	1	34.648	34.649	13299	33F3H
D	1	36.708	36.708	12553	3109H
D#	1	38.891	38.889	11849	2E49H
E	1	41.203	41.202	11184	2BB0H
F	1	43.654	43.653	10556	293CH
F#	1	46.246	46.215	9963	26EBH
G	1	48.999	49.000	9404	24BCH
G#	1	51.913	51.915	8876	22ACH
A	1	55.000	55.001	8378	20BAH
A#	1	58.270	58.270	7908	1EE4H
B	1	61.735	61.736	7464	1D28H
C	2	65.406	65.408	7045	1B85H
C#	2	69.296	69.293	6650	19FAH
D	2	73.416	73.411	6277	1885H
D#	2	77.782	77.785	5924	1724H
E	2	82.406	82.403	5592	15D8H
F	2	87.308	87.306	5278	149EH
F#	2	92.498	92.493	4982	1376H
G	2	97.998	98.000	4702	125EH
G#	2	103.826	103.830	4438	1156H
A	2	110.000	110.002	4189	105DH
A#	2	116.540	116.540	3954	0F72H
B	2	123.470	123.472	3732	0E94H
C	3	130.812	130.798	3523	0DC3H
C#	3	138.592	138.586	3325	0CFDH
D	3	146.832	146.845	3138	0C42H
D#	3	155.564	155.570	2962	0B92H
E	3	164.812	164.807	2796	0AECH
F	3	174.616	174.612	2639	0A4FH
F#	3	184.996	184.986	2491	09BBH
G	3	195.996	196.001	2351	092FH
G#	3	207.652	207.661	2219	08ABH
A	3	220.000	219.952	2095	082FH
A#	3	233.080	233.080	1977	07B9H
B	3	246.940	246.946	1866	074AH

1.2 USING THE PWM STATEMENT

NOTE	OCTAVE	IDEAL FREQUENCY	ACTUAL FREQUENCY	RELOAD	HEX RELOAD
C	4	261.624	261.669	1761	06E1H
C#	4	277.184	277.256	1662	067EH
D	4	293.664	293.690	1569	0621H
D#	4	311.128	311.141	1481	05C9H
E	4	329.624	329.614	1398	0576H
F	4	349.232	349.355	1319	0527H
F#	4	369.992	370.120	1245	04DDH
G	4	391.992	391.836	1176	0498H
G#	4	415.304	415.135	1110	0456H
A	4	440.000	440.114	1047	0417H
A#	4	466.160	465.925	989	03DDH
B	4	493.880	493.890	933	03A5H
C	5	523.248	523.042	881	0371H
C#	5	554.368	554.512	831	033FH
D	5	587.238	587.006	785	0311H
D#	5	622.256	621.862	741	02E5H
E	5	659.248	659.228	699	02BBH
F	5	698.464	698.182	660	0294H
F#	5	739.984	739.647	623	026FH
G	5	783.984	783.674	588	024CH
G#	5	830.608	830.270	555	022BH
A	5	880.000	879.389	524	020CH
A#	5	932.320	932.793	494	01EEH
B	5	987.760	986.724	467	01D3H
C	6	1046.496	1047.272	440	01B8H
C#	6	1108.736	1107.692	416	01A0H
D	6	1174.656	1175.510	392	0188H
D#	6	1244.512	1245.405	370	0172H
E	6	1318.496	1320.343	349	015DH
F	6	1396.928	1396.364	330	014AH
F#	6	1479.968	1481.672	311	0137H
G	6	1567.968	1567.347	294	0126H
G#	6	1661.216	1663.538	277	0115H
A	6	1760.000	1758.779	262	0106H
A#	6	1864.640	1865.587	247	00F7H
B	6	1975.520	1977.682	233	00E9H

1.2 USING THE PWM STATEMENT

NOTE	OCTAVE	IDEAL FREQUENCY	ACTUAL FREQUENCY	RELOAD	HEX RELOAD
C	7	2092.992	2094.545	220	00DCH
C#	7	2217.472	2215.385	208	00D0H
D	7	2349.312	2351.020	196	00C4H
D#	7	2489.024	2490.811	185	00B9H
E	7	2636.992	2633.143	175	00AFH
F	7	2793.856	2792.727	165	00A5H
F#	7	2959.936	2953.846	156	009CH
G	7	3135.936	3134.694	147	0093H
G#	7	3322.432	3315.108	139	008BH
A	7	3520.000	3517.557	131	0083H
A#	7	3729.280	3716.129	124	007CH
B	7	3951.040	3938.362	117	0075H
C	8	4185.984	4189.091	110	006EH
C#	8	4434.944	4430.770	104	0068H
D	8	4698.624	4702.041	98	0062H
D#	8	4987.048	5008.695	92	005CH
E	8	5273.984	5296.552	87	0057H
F	8	5587.712	5619.512	82	0052H
F#	8	5919.872	5907.692	78	004EH
G	8	6217.872	6227.027	74	004AH
G#	8	6644.864	6678.261	69	0045H
A	8	7040.000	7089.231	65	0041H
A#	8	7458.560	7432.258	62	003EH
B	8	7902.080	7944.827	58	003AH

1.2 USING THE PWM STATEMENT

The following program generates the appropriate reload values for the PWM statement, using any crystal. The user enters the desired frequency and the crystal and the program determined the reload values and errors.

```
>10 INPUT "ENTER CRYSTAL FREQUENCY - ",X
>20 T=12/X
>30 INPUT "ENTER DESIRED FREQUENCY FOR PWM - ",F
>40 F1=1/F
>50 C=(F1/T)/2 : REM CALCULATE RELOAD VALUE
>60 IF C<20 THEN 30
>70 C1=C-INT(C) : REM CALCULATE FRACTION
>80 IF C1<.5 THEN 90 : C=C+1
>90 PRINT : PRINT "THE DESIRED FREQUENCY IS - ",X,"HZ"
>100 C=INT(C) : PRINT
>110 PRINT "THE ACTUAL FREQUENCY IS - ",1/(2*C*T),"HZ"
>120 PRINT
>130 PRINT "THE RELOAD VALUE FOR PWM IS - ",C," IN HEX - ",: PH1.C
>140 INPUT "ANOTHER FREQUENCY, 1=YES, 0=NO - ",Q
>150 IF Q=1 THEN 20
```

1.3 BAUD RATES AND CRYSTALS

The 16 bit auto-reload timer/counter (TIMER2) that is used to generate baud rates for the MCS BASIC-52 device is capable of generating accurate baud rates with a number of crystals. The following is a list of crystals that will accurately generate 9600 baud on the MCS BASIC-52 device. Additionally, the crystal values on the left hand side of the table will accurately generate 19200 baud.

XTAL	RCAP2 RELOAD	XTAL	RCAP2 RELOAD
3680400	65524	3993600	65523
4300800	65522	4608000	65521
4915200	65520	5222400	65519
5529600	65518	5836800	65517
6144000	65516	6451200	65515
6758400	65514	7065600	65513
7372800	65512	7680000	65511
7987200	65510	8294400	65509
8601600	65508	8908800	65507
9216000	65506	9523200	65505
9830400	65504	10137600	65503
10444800	65502	10752000	65501
11059200	65500	11366400	65499
11673600	65498	11980800	65497

With the crystals listed above, the accuracy of the baud rate generator and the REAL TIME CLOCK will depend ONLY on the absolute accuracy of the crystal. Note that the baud rate generator for the 8052AH is so accurate that any crystal above 10 MHz will generate 9600 baud to within 1.5% accuracy.

1.3 BAUD RATES AND CRYSTALS

The following program generates the appropriate TIMER2 reload values for a given baud rate. The user supplies the system clock frequency and the desired baud rate and the program calculates the proper TIMER2 reload value. Additionally, percent error, for both the baud rate generator and MCS BASIC-52's REAL TIME CLOCK are calculated and displayed.

```
>10 INPUT"ENTER CRYSTAL - ",X
>20 INPUT"ENTER BAUD RATE - ",B
>30 R=X/(32*B):T=X/76800
>40 R1=R-INT(R):T1=T-INT(T)
>50 IF R1<.5 THEN 80
>60 R1=1-R1
>70 R=R+1
>80 IF T1<.5 THEN 110
>90 T1=1-T1
>100 T=T+1
>110 PRINT "TIMER2 RELOAD VALUE IS - ",USING(#######),INT(65536-R)
>120 PRINT "BAUD RATE ERROR IS - ",USING(##.###),(R1/R)*100,"%"
>130 PRINT "REAL TIME CLOCK ERROR IS - "(T1/T)*100,"%"
```

1.4 QUICK REFERENCE

COMMANDS:

COMMAND	FUNCTION	EXAMPLE(S)
RUN	Execute a program	RUN
CONT	CONTinue after a STOP or control-C	CONT
LIST	LIST program to the console device	LIST LIST 10-50
LIST#	LIST program to serial printer	LIST# LIST# 50
LIST@	LIST program to user driver (version 1.1 only)	LIST@ LIST@ 50
NEW	erase the program stored in RAM	NEW
NULL	set NULL count after carriage return-line feed	NULL NULL 4
RAM	evoke RAM mode, current program in READ/WRITE memory	RAM
ROM	evoke ROM mode, current program in ROM/EPROM memory	ROM ROM 3
XFER	transfer a program from ROM/EPROM to RAM	XFER
PROG	save the current program in EPROM	PROG
PROG1	save baud rate information in EPROM	PROG1
PROG2	save baud rate information in EPROM and execute program after RESET	PROG2
PROG3	save baud rate and MTOP information in EPROM (version 1.1 only)	PROG3
PROG4	save baud rate and MTOP information in EPROM and execute program after RESET (version 1.1 only)	PROG4

1.4 QUICK REFERENCE

COMMANDS:

COMMAND	FUNCTION	EXAMPLE(S)
PROG5	same as PROG4 except that external RAM is not cleared on RESET or power up if external RAM contains a 0A5H in location 5EH (version 1.1 only)	PROG5
PROG6	same as PROG6 except that external code location 4039H is CALLED after RESET (version 1.1 only)	PROG6
FPROG	save the current program in EPROM using the INTELligent algorithm	FPROG
FPROG1	save baud rate information in EPROM using the INTELligent algorithm	FPROG1
FPROG2	save baud rate information in EPROM and execute program after RESET, use INTELligent algorithm	FPROG2
FPROG3	same as PROG3, except INTELligent programming algorithm is used (version 1.1 only)	FPROG3
FPROG4	same as PROG4, except INTELligent programming algorithm is used (version 1.1 only)	FPROG4
FPROG5	same as PROG5, except INTELligent programming algorithm is used (version 1.1 only)	FPROG5
FPROG6	same as PROG6, except INTELligent programming algorithm is used (version 1.1 only)	FPROG6

1.4 QUICK REFERENCE

STATEMENTS:

STATEMENT	FUNCTION	EXAMPLE(S)
BAUD	set baud rate for line printer port	BAUD 1200
CALL	CALL assembly language program	CALL 9000H
CLEAR	CLEAR variables, interrupts and Strings	CLEAR
CLEARS	CLEAR Stacks	CLEARS
CLEARI	CLEAR Interrupts	CLEARI
CLOCK1	enable REAL TIME CLOCK	CLOCK1
CLOCK0	disable REAL TIME CLOCK	CLOCK0
DATA	DATA to be read by READ statement	DATA 100
READ	READ data in DATA statement	READ A
RESTORE	RESTORE READ pointer	RESTORE
DIM	allocate memory for arrayed variables	DIM A(20)
DO	set up loop for WHILE or UNTIL	DO
UNTIL	test DO loop condition (loop if false)	UNTIL A=10
WHILE	test DO loop condition (loop if true)	WHILE A=B
END	terminate program execution	END
FOR-TO-{STEP}	set up FOR-NEXT loop	FOR A=1 TO 5
NEXT	test FOR-NEXT loop condition	NEXT A

1.4 QUICK REFERENCE

STATEMENTS:

STATEMENT	FUNCTION	EXAMPLE(S)
GOSUB	execute subroutine	GOSUB 1000
RETURN	RETURN from subroutine	RETURN
GOTO	GOTO program line number	GOTO 500
ON GOTO	conditional GOTO	ON A GOTO 5, 20
ON GOSUB	conditional GOSUB	ON A GOSUB 2, 6
IF-THEN-{ELSE}	conditional test	IF A<B THEN A=0
INPUT	INPUT a string or variable	INPUT A
LET	assign a variable or string a value (LET is optional)	LET A=10
ONERR	ONERRor GOTO line number	ONERR 1000
ONTIME	generate an interrupt when TIME is equal to or greater than ONTIME argument-line number is after comma	ONTIME 10, 1000
ONEX1	GOSUB to line number following ONEX1 when INT1 pin is pulled low	ONEX1 1000
PRINT	PRINT variables, strings or literals P. is shorthand for PRINT	PRINT A
PRINT#	PRINT to software serial port	PRINT# A
PH0.	PRINT HEX mode with zero suppression	PH0. A
PH1.	PRINT HEX mode with no zero suppression	PH1. A
PH0.#	PH0. to line printer	PH0.# A
PH1.#	PH1.# to line printer	PH1.# A

1.4 QUICK REFERENCE

STATEMENTS:

STATEMENT	FUNCTION	EXAMPLE(S)
PRINT@	PRINT to user defined driver (version 1.1 only)	PRINT@ 5*5
PH0.@	PH0. to user defined driver (version 1.1 only)	PH0. @ XBY(5EH)
PH1.@	PH1. to user defined driver (version 1.1 only)	PH1.@ A
PGM	Program an EPROM (version 1.1 only)	PGM
PUSH	PUSH expressions on argument stack	PUSH 10, A
POP	POP argument stack to variables	POP A, B, C
PWM	PULSE WIDTH MODULATION	PWM 50, 50, 100
REM	REMark	REM DONE
RETI	RETurn from Interrupt	RETI
STOP	break program execution	STOP
STRING	allocate memory for STRINGs	STRING 50, 10
UI1	evoke User console Input routine	UI1
UI0	evoke BASIC console Input routine	UI0
UO1	evoke User console Output routine	UO1
UO0	evoke BASIC console Output routine	UO0
ST@	store top of stack at user specified location (version 1.1 only)	ST@ 1000H ST@ A
LD@	load top of stack from user specified location (version 1.1 only)	LD@ 1000H LD@ A
IDLE	wait for interrupt (version 1.1 only)	IDLE
RROM	run a program in EP(ROM) (version 1.1 only)	RROM 3

1.4 QUICK REFERENCE

OPERATORS — DUAL OPERAND:

OPERATOR	FUNCTION	EXAMPLE(S)
+	ADDITION	1+1
/	DIVISION	10/2
**	EXPONENTATION	2**4
*	MULTIPLICATION	4*4
–	SUBTRACTION	8–4
.AND.	LOGICAL AND	10.AND.5
.OR.	LOGICAL OR	2.OR.1
.XOR.	LOGICAL EXCLUSIVE OR	3.XOR.2

OPERATORS — SINGLE OPERAND:

ABS( )	ABSOLUTE VALUE	ABS(–3)
NOT( )	ONES COMPLEMENT	NOT(0)
INT( )	INTEGER	INT(3.2)
SGN( )	SIGN	SGN(–5)
SQR( )	SQUARE ROOT	SQR(100)
RND	RANDOM NUMBER	RND
LOG( )	NATURAL LOG	LOG(10)
EXP( )	"e" (2.7182818) TO THE X	EXP(10)
SIN( )	RETURNS THE SINE OF ARGUMENT	SIN(3.14)
COS( )	RETURNS THE COSINE OF ARGUMENT	COS(0)
TAN( )	RETURNS THE TANGENT OF ARGUMENT	TAN(.707)
ATN( )	RETURNS ARCTANGENT OF ARGUMENT	ATN(1)

1.4 QUICK REFERENCE

OPERATORS — SPECIAL FUNCTION:

CBY( )	READ PROGRAM MEMORY	P. CBY(4000)
DBY( )	READ/ASSIGN INTERNAL DATA MEMORY	DBY(99) = 10
XBY( )	READ/ASSIGN EXTERNAL DATA MEMORY	P. XBY(10)
GET	READ CONSOLE	P. GET
IE	READ/ASSIGN IE REGISTER	IE = 82H
IP	READ/ASSIGN IP REGISTER	IP = 0
PORT1	READ/ASSIGN I/O PORT 1 (P1)	PORT1 = 0FFH
PCON	READ/ASSIGN PCON REGISTER	PCON = 0
RCAP2	READ/ASSIGN RCAP2 (RCAP2H:RCAP2L)	RCAP2 = 100
T2CON	READ/ASSIGN T2CON REGISTER	P. T2CON
TCON	READ/ASSIGN TCON REGISTER	TCON = 10H
TMOD	READ/ASSIGN TMOD REGISTER	P. TMOD
TIME	READ/ASSIGN THE REAL TIME CLOCK	P. TIME
TIMER0	READ/ASSIGN TIMER0 (TH0: TL0)	TIMER0 = 0
TIMER1	READ/ASSIGN TIMER1 (TH1: TL1)	P. TIMER1
TIMER2	READ/ASSIGN TIMER2 (TH2: TL2)	TIMER2 = 0FFH

STORED CONSTANT:

PI	PI – 3.1415926	PI

1.5 INSTRUCTION SET SUMMARY

COMMANDS
RUN
CONT
LIST
LIST#
LIST@ (V1.1)
NEW
NULL
RAM
ROM
XFER
PROG
PROG1
PROG2
PROG3 (V1.1)
PROG4 (V1.1)
PROG5 (V1.1)
PROG6 (V1.1)
FPROG
FPROG1
FPROG2
FPROG3 (V1.1)
FPROG4 (V1.1)
FPROG5 (V1.1)
FPROG6 (V1.1)

STATEMENTS
BAUD
CALL
CLEAR
CLEAR(S&I)
CLOCK(1&0)
DATA
READ
RESTORE
DIM
DO-WHILE
DO-UNTIL
END
FOR-TO-STEP
NEXT
GOSUB
RETURN
GOTO
ON-GOTO
ON-GOSUB
IF-THEN-ELSE
INPUT
LET
ONERR
ONEX1
ONTIME
PRINT
PRINT#
PRINT@ (V1.1)
PH0.
PH0.#
PH0.@ (V1.1)
PH1.
PH1.#
PH1.@ (V1.1)
PGM (V1.1)
PUSH
POP
PWM
REM
RETI
STOP
STRING
UI(1&0)
U0(1&0)
LD@ (V1.1)
ST@ (V1.1)
IDLE (V1.1)
RROM (V1.1)

OPERATORS
ADD (+)
DIVIDE (/)
EXPONENTIATION (**)
MULTIPLY (*)
SUBTRACT (−)
LOGICAL AND (.AND.)
LOGICAL OR (.OR.)
LOGICAL X-OR (.XOR.)
LOGICAL NOT (.OR.)
ABS( )
INT( )
SGN( )
SQR( )
RND
LOG( )
EXP( )
SIN( )
COS( )
TAN( )
ATN( )
=, >, >=, <, <=, <>
ASC( )
CHR( )
CBY( )
DBY( )
XBY( )
GET
IE
IP
PORT1
PCON
RCAP2
T2CON
TCON
TMOD
TIME
TIMER0
TIMER1
TIMER2
XTAL
MTOP
LEN
FREE
PI

1.6 FLOATING POINT FORMAT

MCS BASIC-52 stores all floating point numbers in a normalized packed BCD format with an offset binary exponent. The simplest way to demonstrate the floating point format is to use an example. If the number PI (3.1415926) was stored in location X, the following would appear in memory.

LOCATION	VALUE	DESCRIPTION
X	81H	EXPONENT — 81H = 10**1, 82H = 10**2, 80H = 10**0, 7FH = 10**−1 etc. THE NUMBER ZERO IS REPRESENTED WITH A ZERO EXPONENT
X-1	00H	SIGN BIT — 00H = POSITIVE, 01H = NEGATIVE OTHER BITS ARE USED AS TEMPS ONLY DURING A CALCULATION
X-2	26H	LEAST SIGNIFICANT TWO DIGITS
X-3	59H	NEXT LEAST SIGNIFICANT TWO DIGITS
X-4	41H	NEXT MOST SIGNIFICANT TWO DIGITS
X-5	31H	MOST SIGNIFICANT TWO DIGITS

Because MCS BASIC-52 normalizes all numbers, the most significant digit is never a zero unless the number is zero.

1.7 STORAGE ALLOCATION

This section is intended to answer the question — where does MCS BASIC-52 store its variables and strings?

Two 16 bit pointers stored in external memory control the allocation of strings and variables and an additional two pointers control the allocation of scalar variables and dimensioned variables. These pointers are located and defined as follows:

LOCATION (H–L)	NAME	DESCRIPTION
10AH–10BH	MTOP	THE TOP OF RAM THAT IS ASSIGNED TO BASIC
104H–105H	VARTOP	VARTOP = MTOP – (THE NUMBER OF BYTES OF MEMORY THAT THE USER HAS ALLOCATED FOR STRINGS). IF STRINGS ARE NOT USED, VARTOP = MTOP
106H–107H	VARUSE	AFTER A NEW, CLEAR, OR RUN IS EXECUTED, VARUSE = VARTOP, EVERYTIME THE USER ASSIGNS OR USES A VARIABLE VARUSE IS DECREMENTED BY A COUNT OF 8.
108H–109H	DIMUSE	AFTER A NEW, CLEAR, OR RUN IS EXECUTED, DIMUSE = [LENGTH OF THE USER PROGRAM THAT IS IN RAM MEMORY + STARTING ADDRESS OF THE USER PROGRAM IN RAM (512) + THE LENGTH OF ONE FLOATING POINT NUMBER (6)]. IF NO PROGRAM IS IN RAM MEMORY, DIMUSE = 518 AFTER A CLEAR IS EXECUTED

MCS BASIC-52 stores string variables between VARTOP and MTOP. $(0) is stored from VARTOP to VARTOP + (user defined string length + 1), $(1) is stored from VARTOP + (user defined string length + 1) + 1 to VARTOP + 2 * (user defined string length + 1) etc. If MCS BASIC-52 attempts to access a string that is outside the bounds established by MTOP , a MEMORY ALLOCATION ERROR is generated.

Now, Scalar variables are stored from VARTOP "down" and Dimensioned variables are stored from DIMUSE "up." When the user dimensions a variable either implicity or explicity the value of DIMUSE increases by the number of bytes required to store that dimensioned variable. For example, if the user executes a DIM A(10) statement, DIMUSE would increase by 66. This is because the user is requesting storage for 11 numbers (A(0) through A(10)) and each number requires 6 bytes for storage and 6 * 11 = 66.

1.7 STORAGE ALLOCATION

As mentioned in the previous example, everytime the user defines a new variable the VARUSE pointer decrements by a count of 8. Six of the eight counts are due to the memory required to store a floating point number and the other two counts are the storage required for the variable name (i.e. A1, B7, etc). The variable B7 would be stored as follows:

LOCATION	VALUE	DESCRIPTION
X	37H	THE ASCII VALUE — 7, IF B7 WAS A DIMENSIONED VARIABLE THE MOST SIGNIFICANT BIT OF THIS LOCATION WOULD BE SET. IN VERSION 1.1 THIS LOCATION ALWAYS CONTAINS THE ASCII VALUE FOR THE LAST CHARACTER USED TO DEFINE A VARIABLE
X-1	42H	THE ASCII VALUE — B, IN VERSION 1.1 OF MCS BASIC-52 THIS LOCATION CONTAINS THE ASCII VALUE OF THE FIRST CHARACTER USED TO DEFINE A VARIABLE PLUS 26 * THE NUMBER OF CHARACTERS USED TO DEFINE A VARIABLE, IF THE VARIABLE CONTAINS MORE THAN 2 CHARACTERS.
X-2 THRU X-7	??	THE NEXT SIX LOCATIONS WOULD CONTAIN THE FLOATING POINT NUMBER THAT THE VARIABLE IS ASSIGNED TO, IF THE VARIABLE WAS A SCALAR VARIABLE. IF THE VARIABLE WAS DIMENSIONED, X-2 WOULD CONTAIN THE LIMIT OF THE DIMENSION (I.E. THE MAX. NUMBER OF ELEMENTS IN THE ARRAY) AND X-3: X-4 WOULD CONTAIN THE BASE ADDRESS OF THE ARRAY. THIS ADDRESS IS EQUAL TO THE OLD VALUE OF THE DIMUSE POINTER BEFORE THE ARRAY WAS CREATED

Whenever a new scalar or dimensioned variable is used in a program, MCS BASIC-52 checks both the DIMUSE nd VARUSE pointers to make sure that VARUSE > DIMUSE. If the relationship is not true, a MEMORY ALLOCATION ERROR is generated.

1.7 STORAGE ALLOCATION

To Summarize:

Strings are stored from VARTOP to MTOP.

Scalar variables are stored from VARTOP "down" and VARUSE points to the next available scalar location.

Dimensioned variables are stored from the end of the user program in RAM "up." If no program is in RAM this location is 518. DIMUSE keeps track of the number of bytes the user has allocated for dimensioned variables.

If DIMUSE >= VARUSE a MEMORY ALLOCATION ERROR is generated

1.8 FORMAT OF AN MCS BASIC-52 PROGRAM

This section answers the question "How does MCS BASIC-52 store a program?"

LINE FORMAT

Each line of MCS BASIC-52 text consists of tokens and ASCII characters, plus 4 bytes of overhead. Three of these four bytes are stored at the beginning of every line. The first byte contains the length of a line in binary and the second two bytes are the line number in binary. The fourth byte is stored at the end of the line and this byte is always a 0DH or a carriage return in ASCII. An example of a typical line is shown below, assume that this is the first line of a program in RAM.

10 FOR I = 1 TO 10 : PRINT I : NEXT I

LOCATION	BYTE	DESCRIPTION
512	11H	THE LENGTH OF THE LINE IN BINARY (17D BYTES)
513	00H	HIGH BYTE OF THE LINE NUMBER
514	0AH	LOW BYTE OF THE LINE NUMBER
515	0A0H	THE TOKEN FOR "FOR"
516	49H	THE ASCII CHARACTER "I"
517	0EAH	THE TOKEN FOR "="
518	31H	THE ASCII FOR "1"
519	0A6H	THE TOKEN FOR "TO"
520	31H	THE ASCII FOR "1"
521	30H	THE ASCII FOR "0"
522	3AH	THE ASCII FOR ":"
523	89H	THE TOKEN FOR "PRINT"
524	49H	THE ASCII FOR "I"
525	3AH	THE ASCII FOR ":"
526	97H	THE TOKEN FOR "NEXT"
527	49H	THE ASCII FOR "I"
528	0DH	END OF LINE (CARRIAGE RETURN)

TO FIND THE LOCATION OF THE NEXT LINE, THE LENGTH OF THE LINE IS ADDED TO THE LOCATION WHERE THE LENGTH OF THE LINE IS STORED. IN THIS EXAMPLE, 512 + 17D = 529, WHICH IS WHERE THE NEXT LINE IS STORED.

The END of a program is designated by the value 01H. So, in the previous example if line 10 was the only line in the program, location 529 would contain the value 01H. A program simply consists of a number of lines packed together in one continuous block with the last line ending in a 0DH, 01H sequence.

1.8 FORMAT OF AN MCS BASIC-52 PROGRAM

EPROM FILE FORMAT

The EPROM FILE format consists of the same line and program format, previously described except that each program in the EPROM file begins with the value 55H. The value 55H is only used by MCS BASIC-52 to determine if a valid program is present. If the user types ROM 6, MCS BASIC-52 actually goes through the first program stored in EPROM line by line until the END of PROGRAM (01H) is found, then it examines the next location to see if a 55H is stored in that location. It then goes through that program line by line. This process is repeated 6 times. If the character 55H is not found after the end of a program, MCS BASIC-52 will return with the PROM MODE error message. This would mean that less than six programs were stored in that EPROM.

The first program stored in EPROM (ROM 1) always begins at location 8010H and this location will always contain a 55H. The actual user program will begin at location 8011H.

EPROM locations 8000H through 800FH are reserved by MCS BASIC-52. These locations contain initialization information when the PROGX options are used. Version 1.0 of MCS BASIC-52 only used the first three bytes of this reserved EPROM area. The information stored in these bytes is as follows:

LOCATION	DESCRIPTION
8000H	CONTAINED A 31H IF PROG 1 WAS USED, CONTAINED A 32H IF PROG 2 WAS USED
8001H	BAUD RATE (RCAP2H)
8002H	BAUD RATE (RCAP2L)

Version 1.1 of MCS BASIC-52 uses the same locations as Version 1.0, but additionally locations 8003H and 8004H (high byte, low byte) are used to store the MTOP information for the PROG 3, 4, 5, 6 options.

IMPORTANT NOTE —

The PROG X options simply store ASCII character following the PROG command in loction 8000H. That is why PROG 1 stores a 31H in location 8000H, PROG 2 a 32H, PROG 3 (Version 1.1 only) a 33H etc. If the user employs the user defined reset option defined in Chapter 11 of this manual, it would be possible for the user to create unique PROG options. For example, PROG A would store a 41H in location 8000H and upon RESET the user could examine this location with an assembly language routine and generate a unique PROG A reset routine for that particular application.

1.9 ANSWERS TO A FEW QUESTIONS

QUESTION

Why can't MCS BASIC-52 access the 8052's SPECIAL FUNCTION REGISTER SCON?

ANSWER

The only time the user would likely change the contents of SCON is if the user is writing custom I/O drivers in assembly language. If the user is writing assembly language I/O drivers, then the user can change the contents of SCON in assembly language. Changing the contents of SCON can cause MCS BASIC-52's console routines to crash.

QUESTION

I have written an upload/download routine using my computer, but when I download a program, MCS BASIC-52 misses characters, why?

ANSWER

MCS BASIC-52 is actually capable of accepting characters at 38,400 baud. The problem is that after MCS BASIC-52 receives a carriage return (cr), it tokenizes the line of text that was just entered. Depending on how complicated and how long the line is, MCS BASIC-52 can take up to a couple of hundred milliseconds to tokenize the line. If the user keeps stuffing characters into the serial port while MCS BASIC-52 is tokenizing the line, the characters will be lost. What the user must do in the download routine is wait until MCS BASIC-52 responds with the prompt character (>) after a carriage return is sent to the MCS BASIC-52 device. The prompt (>) informs the user that MCS BASIC-52 is ready to receive characters from the console device.

QUESTION

I am writing in assembly language and I notice that the 8052AH has no decrement DPTR instruction. What is the easiest, shortest or simplest way to decrement the DPTR?

ANSWER

The shortest one we know is:

```
        XCH     A, DPL      ; SWAP A<>DPL
        JNZ     DECDP       ; DPH = DPH-1 IF DPL = 0
        DEC     DPH
DECDP:  DEC     A           ; DPL = DPL-1
        XCH     A, DPL
```

This routine affects no flags or registers (except the DPTR) either!

1.9 ANSWERS TO A FEW QUESTIONS

QUESTION

After RESET or power-up, MCS BASIC-52 does not return the proper value for MTOP, what's the problem?

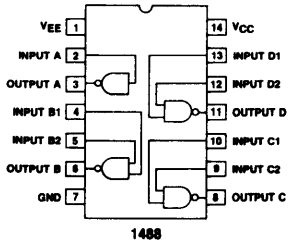
ANSWER

Virtually everytime this problem occurs it is because something is wrong with the decoding circuitry in the system or one or more of the address lines to the RAM are open or shorted. The user should make sure that all of the address lines to the system RAM are connected properly!

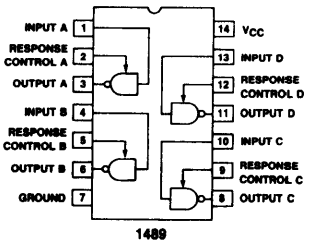
A simple memory test can be implemented in the COMMAND MODE to verify the addressing to the RAM. First set XBY(1000H) = 55, then walk ones across the address (i.e. P. XBY(1001H) – P. XBY(1002H) – P. XBY(1004H) – P. XBY(1008H) P. XBY(1010H)) until all locations are tested. If for instance, P. XBY(1008H) returns a result of 55, then address line 3 (A3) would probably be open or shorted.

1.10 PIN-OUT LIST

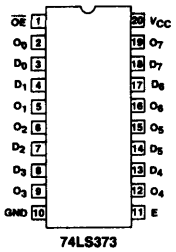
The following is a pin-out list of the most common devices found in an MCS BASIC-52 system:



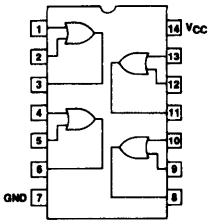
1488



1489

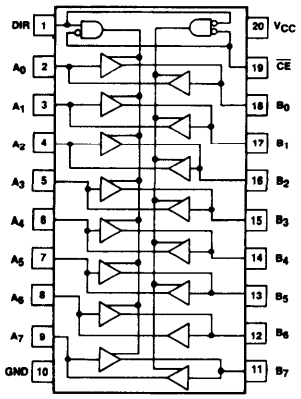


74LS373



74LS32

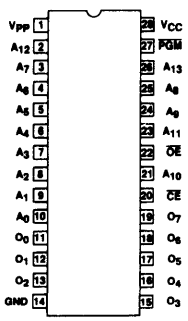
74LS138



74LS245

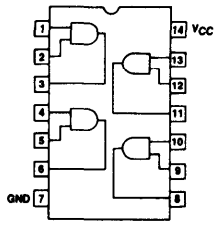
27256	2764A	2732A
VPP	VPP	
A12	A12	
A7	A7	A7
A6	A6	A6
A5	A5	A5
A4	A4	A4
A3	A3	A3
A2	A2	A2
A1	A1	A1
A0	A0	A0
O0	O0	O0
O1	O1	O1
O2	O2	O2
GND	GND	GND

P27128A

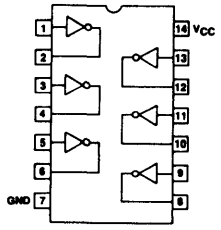


2732A	2764A	27256
	VCC	VCC
	PGM	A14
VCC	N.C.	A13
A9	A8	A8
A9	A9	A9
A11	A11	A11
OE/VPP	OE	OE
A10	A10*	A10
CE	CE	CE
O7	O7	O7
O6	O6	O6
O5	O5	O5
O4	O4	O4
O3	O3	O3

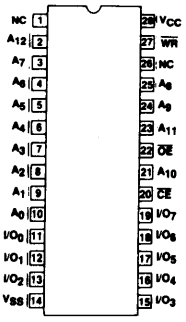
EPROMS



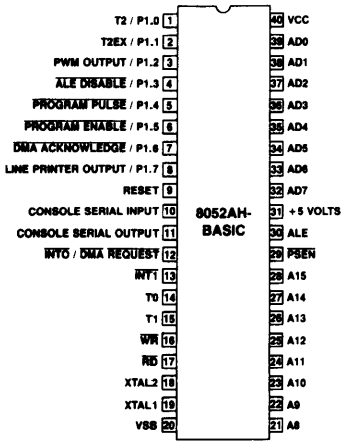
74LS08
74LS00
INVERTING

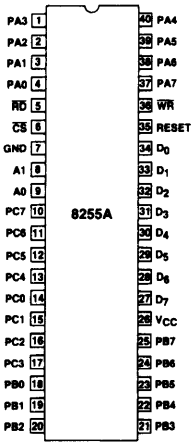


7406/LS04/LS05
7407 NON-INVERTING



8K x 8 SRAM





1.11 8052AH SPECIAL FUNCTION REGISTERS

The following details the operation of the special function registers on the 8052AH:

SYMBOL NAME	NAME	ADDRESS	MCS BASIC-52
ACC	Accumulator	0E0H	NOT ADDRESSABLE
B	B Register	0F0H	NOT ADDRESSABLE
PSW	Program Status Word	0D0H	NOT ADDRESSABLE
SP	Stack Pointer	81H	NOT ADDRESSABLE
DPTR	Data Pointer 2 Bytes:		
DPH	Low Byte	82H	NOT ADDRESSABLE
DPL	High Byte	83H	NOT ADDRESSABLE
P0	Port 0	80H	NOT ADDRESSABLE
P1	Port 1	90H	PORT1
P2	Port 2	0A0H	NOT ADDRESSABLE
P3	Port 3	0B0H	NOT ADDRESSABLE
IP	Interrupt Priority Control	0B8H	IP
IE	Interrupt Enable Control	0A8H	IE
TMOD	Timer/Counter Mode Control	89H	TMOD
TCON	Timer/Counter Control	88H	TCON
T2CON	Timer/Counter 2 Control	0C8H	T2CON
TH0	Timer/Counter 0 High Byte	8CH	
			} TIMER0
TL0	Timer/Counter 0 Low Byte	8AH	
TH1	Timer/Counter 1 High Byte	8DH	
			} TIMER1
TL1	Timer/Counter 1 Low Byte	8BH	
TH2	Timer/Counter 2 High Byte	0CDH	
			} TIMER2
TL2	Timer/Counter 2 Low Byte	0CCH	
RCAP2H	T/C 2 Capture Reg. High Byte	0CBH	
			} RCAP2
RCAP2L	T/C 2 Capture Reg. Low Byte	0CAH	
SCON	Serial Control	98H	NOT ADDRESSABLE
SBUF	Serial Data Buffer	99H	NOT ADDRESSABLE
PCON	Power Control	87H	NOT ADDRESSABLE

1.11 8052AH SPECIAL FUNCTION REGISTERS

PSW: PROGRAM STATUS WORD. ADDRESS 0D0H

CY	AC	F0	RS1	RS0	OV	—	P

CY PSW.7 Carry Flag.

AC PSW.6 Auxiliary Carry Flag.

F0 PSW.5 Flag 0 available to the user for general purpose.

RS1 PSW.4 Register Bank selector bit 1.

RS0 PSW.3 Register Bank selector bit 0.

OV PSW.2 Overflow Flag.

— PSW.1 RESERVED FOR FUTURE USE.

P PSW.0 PARITY FLAG.

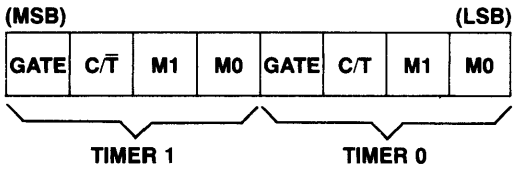
PCON: POWER CONTROL REGISTER. NOT BIT ADDRESSABLE.

SMOD	—	—	—	—	—	—	—

SMOD Doubles the baud rate when TIMER 1 is used to generate the baud rate for the serial port.

The remaining bits of PCON are not implemented on the MCS BASIC-52 device.

1.11 8052AH SPECIAL FUNCTION REGISTERS



GATE Gating control When set. Timer/Counter "x" is enabled only while "INTx" pin is high and "TRx" control pin is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set

C/T Timer or Counter Selector Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).

M1	M0	Operating Mode
0	0	MCS-48 Timer "TLx" serves as five-bit prescaler.
0	1	16 bit Timer/Counter "THx" and "TLX" are cascaded; there is no prescaler
1	0	8-bit auto-reload timer-counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an eight-bit timer counter-controlled by the standard Timer 0 control bits TH0 is an eight-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer-counter 1 stopped.

TMOD: Timer/Counter Mode Control Register

1.11 8052AH SPECIAL FUNCTION REGISTERS

(MSB) (LSB)

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RLS2}$

Symbol	Position	Name and Significance
TF2	T2CON.7	Timer 2 overflow flag set by a Timer 2 overflow and must be cleared by software. TF2 will not be set when either RCLK = 1 or TCLK = 1.
EXF2	T2CON.6	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and 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 and 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 and 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 a 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	Start/stop control for Timer 2. A logic 1 starts the timer.
C/$\overline{T2}$	T2CON.1	Timer or counter select. (Timer 2) 0 = Internal timer (OSC/12) 1 = External event counter (falling edge triggered).
CP/$\overline{RL2}$	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 negative transitions at T2EX when EXEN 2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.

Timer/Counter 2 Control Register

1.11 8052AH SPECIAL FUNCTION REGISTERS

(MSB)							(LSB)
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	$f_{osc.}/12$
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	$f_{osc.}/64$ or $f_{osc.}/32$
1	1	3	9-bit UART	variable

- **SM2** enables the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3, if SM2 is set to 1 then RI will not be activated 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** enables serial reception. Set by software to enable reception. Clear by software to disable reception.
- **TB8** is the 9th data bit that will be transmitted in modes 2 and 3. Set or clear by software as desired.
- **RB8** In modes 2 and 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** is 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 the other modes, in any serial transmission. Must be cleared by software.
- **RI** is 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 the other modes, in any serial reception (except see SM2). Must be cleared by software.

SCON: Serial Port Control Register

(MSB)							(LSB)
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
0	1	0	1	0	1	0	0

Symbol	Position	Name and Significance	Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on timer/counter overflow. Cleared by hardware when processor vectors to interrupt routine.	IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn timer/counter on/off.	IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on timer/counter overflow. Cleared by hardware when processor vectors to interrupt routine.	IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn timer/counter on/off.	IT0	TCON.0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

TCON: Timer/Counter Control Register

1.11 8052AH SPECIAL FUNCTION REGISTERS

(MSB)							LSB)
X	X	PT2	PS	PT1	PX1	PT0	PX0

Symbol	Position	Function
—	IP.7	reserved
—	IP.6	reserved
PT2	IP.5	defines the Timer 2 interrupt priority level. PT2 = 1 programs it to the higher priority level.
PS	IP.4	defines the Serial Port interrupt priority level. PS = 1 programs it to the higher priority level.
PT1	IP.3	defines the Timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level.
PX1	IP.2	defines the External Interrupt 1 priority level. PX1 = 1 programs it to the higher priority level.
PT0	IP.1	defines the Timer 0 interrupt priority level. PT0 = 1 programs it to the higher priority level.
PX0	IP.0	defines the External Interrupt 0 priority level. PX0 = 1 programs it to the higher priority level.

IP: Interrupt Priority Register

(MSB)							LSB
EA	X	ET2	ES	ET1	EX1	ET0	EX0

Symbol	Position	Function
EA	IE.7	disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
—	IE.6	reserved
ET2	IE.5	enables or disables the Timer 2 overflow or capture interrupt. If ET2 = 0, the Timer 2 interrupt is disabled.
ES	IE.4	enables or disables the Serial Port interrupt. If ES = 0, the Serial Port interrupt is disabled.
ET1	IE.3	enables or disables the Timer 1 Overflow interrupt. If ET1 = 0, the Timer 1 interrupt is disabled.
EX1	IE.2	enables or disables External Interrupt 1. If EX1 = 0, External Interrupt 1 is disabled.
ET0	IE.1	enables or disables the Timer 0 Overflow interrupt. If ET0 = 0, the Timer 0 Interrupt is disabled.
EX0	IE.0	enables or disables External Interrupt 0. If EX0 = 0, External Interrupt 0 is disabled.

IE: Interrupt Enable Register

1.12 REFERENCES

REFERENCES

J. Sack and J. Meadows, *Entering BASIC*, Science Research Associates, 1973.

C. Pegels, BASIC: *A Computer Programming Language*, Holden-Day, Inc., 1973.

J. Kemeny and T. Kurtz, *BASIC Programming*, People Computer Company, 1967.

Albrecht, Finkle, and Brown, *BASIC*, People Computer Company, 1973.

T. Dwyer, *A Guided Tour of Computer Programming in BASIC*, Houghton Mifflin Co., 1973.

Eugene H. Barnett, *Programming Time Shared Computers in BASIC*, Wiley-Interscience, L/C 72-175789.

Programming Language #2, Digital Equipment Corp., Maynard, Mass. 01754.

101 BASIC Computer Games, Digital Equipment Corp., Maynard, Mass. 01754.

What to do After You Hit Return. People Computer Company.

BASIC-80 REFERENCE MANUAL, Intel Corp., Santa Clara, Calif.

APPENDIX B

INSTRUCTION SET SUMMARY

This appendix contains two tables (see tables B-1 and B-2): the first identifies all of the 8052's instructions in alphabetical order; the second table lists the instructions according to their hexadecimal opcodes and lists the assembly language instructions that produced that opcode.

The alphabetical listing also includes documentation of the bit pattern, flags affected, number of machine cycles per execution and a description of the instructions operation and function. The list below defines the conventions used to identify operation and bit patterns.

ABBREVIATIONS AND NOTATIONS USED

A	Accumulator	*llllllll*	One byte of a 16-bit address encoded in operand byte
AB	Register Pair		
B	Multiplication Register		
bit address	8052 bit address	*mmmmmmmm*	Data address encoded in operand byte
page address	11-bit code address within 2K page	*oooooooo*	Relative offset encoded in operand byte
relative offset	8-bit 2's complement offset	*r* or *rrr*	Register identifier encoded in operand byte
C	Carry Flag		
code address	Absolute code address		
data	Immediate data	AND	Logical AND
data address	On-chip 8-bit RAM address	NOT	Logical complement
DPTR	Data pointer	OR	Logical OR
PC	Program Counter	XOR	Logical exclusive OR
R*r*	Register (*r* = 0–7)	+	Plus
SP	Stack pointer	−	Minus
high	High order byte	/	Divide
low	Low order byte	•	Multiply
i–j	Bits i through j	(X)	The contents of X
.n	Bit n	((X))	The memory location addressed by (X) (The contents of X)
aaa aaaaaaaa	Absolute page address encoded in instruction and operand byte		
		=	Is equal to
bbbbbbbb	Bit address encoded in operand byte	< >	Is not equal to
		<	Is less than
dddddddd	Immediate data encoded in operand byte	>	Is greater than
		←	Is replaced by

Table B-1. Instruction Set Summary

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
ACALL *code addr* (PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP)) ← (PC) *low* (SP) ← (SP) + 1 ((SP)) ← (PC) *high* (PC) 0–10 ← *page address*	2	a a a 1 0 0 0 1 a a a a a a a a		Push PC on stack, and replace low order 11 bits with low order 11 bits of code address.
ADD A,*#data* (A) ← (A) + *data*	1	0 0 1 0 0 1 0 0 d d d d d d d d	P OV AC C	Add immediate data to A.
ADD A,@R*r* (A) ← (A) + ((R*r*))	1	0 0 1 0 0 1 1 r	P OV AC C	Add contents of indirect address to A.
ADD A,R*r* (A) ← (A) + (R*r*)	1	0 0 1 0 1 r r r	P OV AC C	Add register to A.
ADD A,*data addr* (A) ← (A) + (*data address*)	1	0 0 1 0 0 1 0 1 m m m m m m m m	P OV AC C	Add contents of data address to A.
ADDC A,*#data* (A) ← (A) + (C) + *data*	1	0 0 1 1 0 1 0 0 d d d d d d d d	P OV AC C	Add C and immediate data to A.
ADDC A,@R*r* (A) ← (A) + (C) + ((R*r*))	1	0 0 1 1 0 1 1 r	P OV AC C	Add C and contents of indirect address to A.
ADDC A,R*r* (A) ← (A) + (C) + (R*r*)	1	0 0 1 1 1 r r r	P OV AC C	Add C and register to A.
ADDC A,*data addr* (A) ← (A) + (C) + (*data address*)	1	0 0 1 1 0 1 0 1 m m m m m m m m	P OV AC C	Add C and contents of data address to A.
AJMP *code addr* (PC) 0–10 ← *code address*	2	a a a 0 0 0 0 1 a a a a a a a a		Replace low order 11 bits of PC with low order 11 bits code address.
ANL A,*#data* (A) ← (A) AND *data*	1	0 1 0 1 0 1 0 0 d d d d d d d d	P	Logical AND immediate data to A.
ANL A,@R*r* (A) ← (A) AND ((R*r*))	1	0 1 0 1 0 1 1 r	P	Logical AND contents of indirect address to A.
ANL A,R*r* (A) ← (A) AND (R*r*)	1	0 1 0 1 1 r r r	P	Logical AND register to A.
ANL A,*data addr* (A) ← (A) AND (*data address*)	1	0 1 0 1 0 1 0 1 m m m m m m m m	P	Logical AND contents of data address to A.
ANL C,*bit addr* (C) ← (C) AND (*bit address*)	2	1 0 0 0 0 0 1 0 b b b b b b b b	C	Logical AND bit to C.
ANL C,/*bit addr* (C) ← (C) AND NOT (*bit address*)	2	1 0 1 1 0 0 0 0 b b b b b b b b	C	Logical AND complement of bit to C.
ANL *data addr, #data* (*data address*) ← (*data address*) AND *data*	2	0 1 0 1 0 0 1 1 m m m m m m m m d d d d d d d d		Logical AND immediate data to contents of data address
ANL *data addr*,A (*data address*) ← (*data address*) AND A	1	0 1 0 1 0 0 1 0 m m m m m m m m		Logical AND A to contents of data address.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function

CJNE @Rr,#data,code addr
(PC) ← (PC) + 3
IF ((Rr)) < > data
THEN
(PC) ← (PC) + relative offset
IF ((Rr)) <data
THEN (C) ← 1
ELSE (C) ← 0
	2	1 0 1 1 0 1 1 r
		d d d d d d d d
		o o o o o o o o
			C	If immediate data and contents of indirect address are not equal, jump to code address.

CJNE A,#data,code addr
(PC) ← (PC) + 3
IF (A) < > data
THEN
(PC) ← (PC) + relative offset
IF (A) <data
THEN (C) ← 1
ELSE (C) ← 0
	2	1 0 1 1 0 1 0 0
		d d d d d d d d
		o o o o o o o o
			C	If immediate data and A are not equal, jump to code address.

CJNE A,data addr,code addr
(PC) ← (PC) + 3
IF (A) < > (data address)
THEN
(PC) ← (PC) + relative offset
IF (A) < (data address)
THEN (C) ← 1
ELSE (C) ← 0
	2	1 0 1 1 0 1 0 1
		m m m m m m m m
		o o o o o o o o
			C	If contents of data address and A are not equal, jump to code address.

CJNE Rr,#data,code addr (PC) ← (PC) + 3 'F (Rr) < > data THEN (PC) ← (PC) + relative offset IF (Rr) < data THEN (C) ← 1 ELSE (C) ← 0	2	1 0 1 1 1 r r r d d d d d d d d o o o o o o o o	C	If immediate data and register are not equal, jump to code address.
CLR A (A) ← 0	1	1 1 1 0 0 1 0 0	P	Set A to zero (0).
CLR C (C) ← 0	1	1 1 0 0 0 0 1 1	C	Set C to zero (0).
CLR bit addr (bit address) ← 0	1	1 1 0 0 0 0 1 0 b b b b b b b b		Set bit to zero (0).
CPL A (A) ← NOT (A)	1	1 1 1 1 0 1 0 0	P	Complements each bit in A.
CPL C (C) ← NOT (C)	1	1 0 1 1 0 0 1 1	C	Complement C.
CPL bit addr (bit address) ← NOT (bit address)	1	1 0 1 1 0 0 1 0 b b b b b b b b		Complement bit.
DA A	1	1 1 0 1 0 1 0 0	P C	Adjust A after a BCD add.
DEC @Rr ((Rr)) ← ((Rr)) − 1	1	0 0 0 1 0 1 1 r		Decrement contents of indirect address.
DEC A (A) ← (A) − 1	1	0 0 0 1 0 1 0 0	P	Decrement A.
DEC Rr (Rr) ← (Rr) − 1	1	0 0 0 1 1 r r r		Decrement register.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
DEC *data addr* (*data address*) ← (*data address*) − 1	1	0 0 0 1 0 1 0 1 mmmmmmmm		Decrement contents of data address.
DIV AB (AB) ← (A)/(B)	4	1 0 0 0 0 1 0 0	P OV C	Divide A by B (multiplication register).
DJNZ R*r*,*code addr* (PC) ← (PC) + 2 (R*r*) ← (R*r*) − 1 IF (R*r*) < > 0 THEN (PC) ← (PC) + *relative offset*	2	1 1 0 1 1 r r r o o o o o o o o		Decrement register, if not zero (0), then jump to code address.
DJNZ *data addr,code addr* (PC), ← (PC) + 3 (*data address*) ← (*data address*) − 1 IF (*data address*) < > 0 THEN (PC) ← (PC) + *relative offset*	2	1 1 0 1 0 1 0 1 mmmmmmmm o o o o o o o o		Decrement data address, if zero (0), then jump to code address.
INC @ R*r* ((R*r*)) ← ((R*r*)) + 1	1	0 0 0 0 0 1 1 r		Increment contents of indirect address.
INC A (A) ← (A) + 1	1	0 0 0 0 0 1 0 0	P	Increment A.
INC DPTR (DPTR) ← (DPTR) + 1	1	1 0 1 0 0 0 1 1		Increment 16-bit data pointer.
INC R*r* ((R) ← (R*r*) + 1	1	0 0 0 0 1 r r r		Increment register.
INC *data addr* (*data address*) ← (*data address*) + 1	2	0 0 0 0 0 1 0 1 mmmmmmmm		Increment contents of data address.
JB *bit addr,code addr* (PC) ← (PC) + 3 IF (*bit address*) = 1 THEN (PC) ← (PC) + *relative offset*	2	0 0 1 0 0 0 0 0 b b b b b b b b o o o o o o o o		If bit is one, n jump to code address.
JBC *bit addr,code addr* (PC) ← (PC) + 3 IF (*bit address*) = 1 THEN (*bit address*) ← 0 (PC) ← (PC) + *relative offset*	2	0 0 0 1 0 0 0 0 b b b b b b b b o o o o o o o o		If bit is one, n clear bit and jump to code address.
JC *code addr* (PC) ← (PC) + 2 IF (C) = 1 THEN (PC) ← (PC) + *relative offset*	2	0 1 0 0 0 0 0 0 o o o o o o o o		If C is one, then jump to code address.
JMP @ A + DPTR (PC) ← (A) + (DPTR)	2	0 1 1 1 0 0 1 1		Add A to data pointer and jump to that code address.
JNB *bit addr,code addr* (PC) ← (PC) + 3 IF (*bit address*) = 0 THEN (PC) ← (PC) + *relative offset*	2	0 0 1 1 0 0 0 0 b b b b b b b b o o o o o o o o		If bit is zero, n jump to code address.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
JNC *code addr* (PC) + (PC) + 2 IF (C) = 0 THEN (PC) ← (PC) + *relative offset*	2	0 1 0 1 0 0 0 0 o o o o o o o o		If C is zero (0), n jump to code address.
JNZ *code addr* (PC) ← (PC) + 2 IF (A) < > 0 THEN (PC) ← (PC) + *relative offset*	2	0 1 1 1 0 0 0 0 o o o o o o o o		If A is not zero (0), n jump to code address.
JZ *code addr* (PC) ← (PC) + 2 IF (A) = 0 THEN (PC) ← (PC) + *relative offset*	2	0 1 1 0 0 0 0 0 o o o o o o o o		If A is zero (0), then jump to code address.
LCALL *code addr* (PC) ← (PC) + 3 (SP) ← (SP) + 1 ((SP)) ← ((PC)) *low* (SP) ← (SP) + 1 ((SP)) ← (PC) *high* (PC) ← *code address*	2	0 0 0 1 0 0 1 0 l l l l l l l l† l l l l l l l l†		Push PC on stack and replace entire PC value with code address.
LJMP *code addr* (PC) ← *code address*	2	0 0 0 0 0 0 1 0 l l l l l l l l† l l l l l l l l†		Jump to code address.
MOV @R*r*,#*data* ((R*r*)) ← *data*	1	0 1 1 1 0 1 1 r d d d d d d d d		Move immediate data to indirect address.
MOV @R*r*,A ((R*r*)) ← (A)	1	1 1 1 1 0 1 1 r		Move A to indirect address.
MOV @R*r*,*data addr* ((R*r*)) ← (*data address*)	2	1 0 1 0 0 1 1 r mmmmmmmm		Move contents of data address to indirect address.
MOV A,#*data* (A) ← *data*	1	0 1 1 1 0 1 0 0 d d d d d d d d	P	Move immediate data to A.
MOV A,@R*r* (A) ← ((R*r*))	1	1 1 1 0 0 1 1 r	P	Move contents of indirect address to A.
MOV A,R*r* (A) ← (R*r*)	1	1 1 1 0 1 r r r	P	Move register to A.
MOV A,*data addr* (A) ← (*data address*)	1	1 1 1 0 0 1 0 1 mmmmmmmm	P	Move contents of data address to A.
MOV C,*bit addr* (C) ← (*bit address*)	1	1 0 1 0 0 0 1 0 b b b b b b b b	C	Move bit to C.
MOV DPTR,#*data* (DPTR) ← *data*	2	1 0 0 1 0 0 0 0 d d d d d d d d† d d d d d d d d†		Move two bytes of immediate data pointer.
MOV R*r*,#*data* (R*r*) ← *data*	1	0 1 1 1 1 r r r d d d d d d d d		Move immediate data to register.
MOV R*r*,A (R*r*) ← (A)	1	1 1 1 1 1 r r r		Move A to register.

† The high order byte of the 16-bit operand is in the first byte following the opcode. The low order byte is in the second byte following the opcode.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
MOV Rr,*data addr* (Rr) ← (*data address*)	2	1 0 1 0 1 r r r mmmmmmmm		Move contents of data address to register.
MOV *bit addr*,C (*bit address*) ← (C)	2	1 0 0 1 0 0 1 0 b b b b b b b b		Move C to bit.
MOV *data addr*,#*data* (*data address*) ← *data*	2	0 1 1 1 0 1 0 1 mmmmmmmm d d d d d d d d		Move immediate data to data address.
MOV *data addr*,@Rr (*data address*) ← ((Rr))	2	1 0 0 0 0 1 1 r mmmmmmmm		Move contents of indirect address to data address.
MOV *data addr*,A (*data address*) ← (A)	1	1 1 1 1 0 1 0 1 mmmmmmmm		Move A to data address.
MOV *data addr*,Rr (*data address*) ← (Rr)	2	1 0 0 0 1 r r r mmmmmmmm		Move register to data address.
MOV *data addr1,data addr2* (*data address1*) ← (*data address2*)	2	1 0 0 0 0 1 0 1 mmmmmmmm* mmmmmmmm*		Move contents of second data address to first data address.
MOVC A,@A + DPTR (PC) ← (PC) + 1 (A) ← ((A) + (DPTR))	2	1 0 0 1 0 0 1 1	P	Add A to DPTR and move contents of that code address with A.
MOVC A,@A + PC (A) ← ((A) + (PC))	2	1 0 0 0 0 0 1 1	P	Add A to PC and move contents of that code address with A.
MOVX @DPTR,A ((DPTR)) ← (A)	2	1 1 1 1 0 0 0 0		Move A to external data location addressed by DPTR.
MOVX @Rr,A ((Rr)) ← (A)	2	1 1 1 1 0 0 1 r		Move A to external data location addressed by register.
MOVX A,@DPTR (A) ← ((DPTR))	2	1 1 1 0 0 0 0 0	P	Move contents of external data location addressed by DPTR to A.
MOVX A,@Rr (A) ← ((Rr))	2	1 1 1 0 0 0 1 r	P	Move contents of external data location addressed by register to A.
MUL AB (AB) ← (A) * (B)	4	1 0 1 0 0 1 0 0	P OV C	Multiply A by B (multiplication register).
NOP	1	0 0 0 0 0 0 0 0		Do nothing.
ORL A,#*data* (A) ← (A) OR *data*	1	0 1 0 0 0 1 0 0 d d d d d d d d	P	Logical OR immediate data to A.
ORL A,@Rr (A) ← (A) OR ((Rr))	1	0 1 0 0 0 1 1 r	P	Logical OR contents of indirect address to A.
ORL A,Rr (A) ← (A) OR (Rr)	1	0 1 0 0 1 r r r	P	Logical OR register to A.
ORL A,*data addr* (A) ← (A) OR (*data address*)	1	0 1 0 0 0 1 0 1 mmmmmmmm	P	Logical OR contents of data address to A.
ORL C,*bit addr* (C) ← (C) OR (*bit address*)	2	0 1 1 1 0 0 1 0 b b b b b b b b	C	Logical OR bit to C.

* The source data address (second data address) is encoded in the first byte following the opcode. The destination data address is encoded in the second byte following the opcode.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
ORL C,/*bit addr* (C) ← (C) OR NOT (*bit address*)	2	1 0 1 0 0 0 0 0 b b b b b b b b	C	Logical OR complement of bit to C.
ORL *data addr*,#*data* (*data address*) ← (*data address*) OR *data*	2	0 1 0 0 0 0 1 1 m m m m m m m m d d d d d d d d		Logical OR immediate data to data address.
ORL *data addr*,A (*data address*) ← (*data address*) OR A	1	0 1 0 0 0 0 1 0 m m m m m m m m		Logical OR A to data address.
POP *data addr* (*data address*) ← ((SP)) (SP) ← (SP) − 1	2	1 1 0 1 0 0 0 0 m m m m m m m m		Place top of stack at data address and decrement SP.
PUSH *data addr* (SP) ← (SP) + 1 ((SP)) ← (*data address*)	2	1 1 0 0 0 0 0 0 m m m m m m m m		Increment SP and place contents of data address at top of stack.
RET (PC)*high* ← ((SP)) (SP) ← (SP) − 1 (PC)*low* ← ((SP)) (SP) ← (SP) − 1	2	0 0 1 0 0 0 1 0		Return from subroutine call.
RETI (PC)*high* ← ((SP)) (SP) ← (SP) − 1 (PC)*low* ← ((SP)) (SP) ← (SP) − 1	2	0 0 1 1 0 0 1 0		Return from interrupt routine.
RL A	1	0 0 1 0 0 0 1 1		Rotate A left one position.
RLC A	1	0 0 1 1 0 0 1 1	P C	Rotate A through C left one position.
RR A	1	0 0 0 0 0 0 1 1		Rotate A right one position.
RRC A	1	0 0 0 1 0 0 1 1	P C	Rotate A through C right one position.
SETB C (C) ← 1	1	1 1 0 1 0 0 1 1	C	Set C to one (1).
SETB *bit addr* (*bit address*) ← 1	1	1 1 0 1 0 0 1 0 b b b b b b b b		Set bit to one (1).
SJMP *code addr* (PC) ← (PC) + 2 (PC) ← (PC) + *relative offset*	2	1 0 0 0 0 0 0 0 o o o o o o o o		Jump to code address.
SUBB A,#*data* (A) ← (A) − (C) − *data*	1	1 0 0 1 0 1 0 0 d d d d d d d d	P OV AC C	Subtract immediate data from A.
SUBB A,@Rr (A) ← (A) − (C) − ((R*r*))	1	1 0 0 1 0 1 1 r	P OV AC C	Subtract contents of indirect address from A.
SUBB A,R*r* (A) ← (A) − (C) − (R*r*)	1	1 0 0 1 1 r r r	P OV AC C	Subtract register from A.
SUBB A,*data addr* (A) ← (A) − (C) − (*data address*)	1	1 0 0 1 0 1 0 1 m m m m m m m m	P OV AC C	Subtract contents of data address from A.
SWAP A	1	1 1 0 0 0 1 0 0		Exchange low order nibble with high order nibble in A.

Table B-1. Instruction Set Summary (Cont'd.)

Mnemonic Operation	Cycles	Binary Code	Flags P OV AC C	Function
XCH A,@R*r* *temp* ← ((R*r*)) ((R*r*)) ← (A) (A) ← *temp*	1	1 1 0 0 0 1 1 r	P	Move A to indirect address and vice versa.
XCH A,R*r* *temp* ← (R*r*) (R*r*) ← (A) (A) ← *temp*	1	1 1 0 0 1 r r r	P	Move A to register and vice versa.
XCH A,*data addr* *temp* ← (*data address*) (*data address*) ← (A) (A) ← *temp*	1	1 1 0 0 0 1 0 1 mmmmmmmm	P	Move A to data address and vice versa.
XCHD A,@R*r* *temp* ← ((R*r*)) 0–3 ((R*r*)) 0–3 ← (A) 0–3 (A) 0–3 ← *temp*	1	1 1 0 1 0 1 1 r	P	Move low order of A to low order nibble of indirect address and vice versa.
XRL A,#*data* (A) ← (A) XOR *data*	1	0 1 1 0 0 1 0 0 d d d d d d d d	P	Logical exclusive OR immediate data to A.
XRL A,@R*r* (A) ← (A) XOR ((R*r*))	1	0 1 1 0 0 1 1 r	P	Logical exclusive OR contents of indirect address to A.
XRL A,R*r* (A) ← (A) XOR (R*r*)	1	0 1 1 0 1 r r r	P	Logical exclusive OR register to A.
XRL A,*data addr* (A) ← (A) XOR (*data address*)	1	0 1 1 0 0 1 0 1 mmmmmmmm	P	Logical exclusive OR contents of data address to A.
XRL *data addr*,#*data* (*data address*) ← (*data address*) XOR *data*	2	0 1 1 0 0 0 1 1 mmmmmmmm d d d d d d d d		Logical exclusive OR immediate data to data address.
XRL *data addr*,A (*data address*) ← (*data address*) XOR A	1	0 1 1 0 0 0 1 0 mmmmmmmm		Logical exclusive OR A to data address.

Table B-2. Instruction Opcodes in Hexadecimal

Hex Code	Number of Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	*code addr*
02	3	LJMP	*code addr*
03	1	RR	A
04	1	INC	A
05	2	INC	*data addr*
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	*bit addr,code addr*
11	2	ACALL	*code addr*
12	3	LCALL	*code addr*
13	1	RRC	A
14	1	DEC	A
15	2	DEC	*data addr*
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	*bit addr,code addr*
21	2	AJMP	*code addr*
22	1	RET	
23	1	RL	A
24	2	ADD	A,*#data*
25	2	ADD	A,*data addr*
26	1	ADD	A,@R0
27	1	ADD	A,@R1
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7
30	3	JNB	*bit addr,code addr*
31	2	ACALL	*code addr*
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A,*#data*
35	2	ADDC	A,*data addr*
36	1	ADDC	A,@R0
37	1	ADDC	A,@R1
38	1	ADDC	A,R0
39	1	ADDC	A,R1
3A	1	ADDC	A,R2
3B	1	ADDC	A,R3

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

Hex Code	Number of Bytes	Mnemonic	Operands
3C	1	ADDC	A,R4
3D	1	ADDC	A,R5
3E	1	ADDC	A,R7
3F	1	ADDC	A,R7
40	2	JC	*code addr*
41	2	AJMP	*code addr*
42	2	ORL	*data addr*,A
43	3	ORL	*data addr,#data*
44	2	ORL	A,*#data*
45	2	ORL	A,*data addr*
46	1	ORL	A,@R0
47	1	ORL	A,@R1
48	1	ORL	A,R0
49	1	ORL	A,R1
4A	1	ORL	A,R2
4B	1	ORL	A,R3
4C	1	ORL	A,R4
4D	1	ORL	A,R5
4E	1	ORL	A,R6
4F	1	ORL	A,R7
50	2	JNC	*code addr*
51	2	ACALL	*code addr*
52	2	ANL	*data addr*,A
53	3	ANL	*data addr,#data*
54	2	ANL	A,*#data*
55	2	ANL	A,*data addr*
56	1	ANL	A,@R0
57	1	ANL	A,@R1
58	1	ANL	A,R0
59	1	ANL	A,R1
5A	1	ANL	A,R2
5B	1	ANL	A,R3
5C	1	ANL	A,R4
5D	1	ANL	A,R5
5E	1	ANL	A,R6
5F	1	ANL	A,R7
60	2	JZ	*code addr*
61	2	AJMP	*code addr*
62	2	XRL	*data addr*,A
63	3	XRL	*data addr,#data*
64	2	XRL	A,*#data*
65	2	XRL	A,*data addr*
66	1	XRL	A,@R0
67	1	XRL	A,@R1
68	1	XRL	A,R0
69	1	XRL	A,R1
6A	1	XRL	A,R2
6B	1	XRL	A,R3
6C	1	XRL	A,R4
6D	1	XRL	A,R5
6E	1	XRL	A,R6
6F	1	XRL	A,R7
70	2	JNZ	*code addr*
71	2	ACALL	*code addr*
72	2	ORL	C,*bit addr*
73	1	JMP	@A + DPTR
74	2	MOV	A,*#data*
75	3	MOV	*data addr,#data*
76	2	MOV	@R0,*#data*
77	2	MOV	@R1,*#data*

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

Hex Code	Number of Bytes	Mnemonic	Operands
78	2	MOV	R0,*#data*
79	2	MOV	R1,*#data*
7A	2	MOV	R2,*#data*
7B	2	MOV	R3,*#data*
7C	2	MOV	R4,*#data*
7D	2	MOV	R5,*#data*
7E	2	MOV	R6,*#data*
7F	2	MOV	R7,*#data*
80	2	SJMP	*code addr*
81	2	AJMP	*code addr*
82	2	ANL	C,*bit addr*
83	1	MOVC	A,@A + PC
84	1	DIV	AB
85	3	MOV	*data addr,data addr*
86	2	MOV	*data addr*,@R0
87	2	MOV	*data addr*,@R1
88	2	MOV	*data addr*,R0
89	2	MOV	*data addr*,R1
8A	2	MOV	*data addr*,R2
8B	2	MOV	*data addr*,R3
8C	2	MOV	*data addr*,R4
8D	2	MOV	*data addr*,R5
8E	2	MOV	*data addr*,R6
8F	2	MOV	*data addr*,R7
90	3	MOV	DPTR,*#data*
91	2	ACALL	*code addr*
92	2	MOV	*bit addr*,C
93	1	MOVC	A,@A + DPTR
94	2	SUBB	A,*#data*
95	2	SUBB	A,*data addr*
96	1	SUBB	A,@R0
97	1	SUBB	A,@R1
98	1	SUBB	A,R0
99	1	SUBB	A,R1
9A	1	SUBB	A,R2
9B	1	SUBB	A,R3
9C	1	SUBB	A,R4
9D	1	SUBB	A,R5
9E	1	SUBB	A,R6
9F	1	SUBB	A,R7
A0	2	ORL	C,/*bit addr*
A1	2	AJMP	*code addr*
A2	2	MOV	C,*bit addr*
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@R0,*data addr*
A7	2	MOV	@R1,*data addr*
A8	2	MOV	R0,*data addr*
A9	2	MOV	R1,*data addr*
AA	2	MOV	R2,*data addr*
AB	2	MOV	R3,*data addr*
AC	2	MOV	R4,*data addr*
AD	2	MOV	R5,*data addr*
AE	2	MOV	R6,*data addr*
AF	2	MOV	R7,*data addr*
B0	2	ANL	C,/*bit addr*
B1	2	ACALL	*code addr*
B2	2	CPL	*bit addr*
B3	1	CPL	C

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

Hex Code	Number of Bytes	Mnemonic	Operands
B4	3	CJNE	A,*#data,code addr*
B5	3	CJNE	A,*data addr,code addr*
B6	3	CJNE	@R0,*#data,code addr*
B7	3	CJNE	@R1,*#data,code addr*
B8	3	CJNE	R0,*#data,code addr*
B9	3	CJNE	R1,*#data,code addr*
BA	3	CJNE	R2,*#data,code addr*
BB	3	CJNE	R3,*#data,code addr*
BC	3	CJNE	R4,*#data,code addr*
BD	3	CJNE	R5,*#data,code addr*
BE	3	CJNE	R6,*#data,code addr*
BF	3	CJNE	R7,*#data,code addr*
C0	2	PUSH	*data addr*
C1	2	AJMP	*code addr*
C2	2	CLR	*bit addr*
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A,*data addr*
C6	1	XCH	A,@R0
C7	1	XCH	A,@R1
C8	1	XCH	A,R0
C9	1	XCH	A,R1
CA	1	XCH	A,R2
CB	1	XCH	A,R3
CC	1	XCH	A,R4
CD	1	XCH	A,R5
CE	1	XCH	A,R6
CF	1	XCH	A,R7
D0	2	POP	*data addr*
D1	2	ACALL	*code addr*
D2	2	SETB	*bit addr*
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	*data addr,code addr*
D6	1	XCHD	A,@R0
D7	1	XCHD	A,@R1
D8	2	DJNZ	R0,*code addr*
D9	2	DJNZ	R1,*code addr*
DA	2	DJNZ	R2,*code addr*
DB	2	DJNZ	R3,*code addr*
DC	2	DJNZ	R4,*code addr*
DD	2	DJNZ	R5,*code addr*
DE	2	DJNZ	R6,*code addr*
DF	2	DJNZ	R7,*code addr*
E0	1	MOVX	A,@DPTR
E1	2	AJMP	*code addr*
E2	1	MOVX	A,@R0
E3	1	MOVX	A,@R1
E4	1	CLR	A
E5	2	MOV	A,*data addr*
E6	1	MOV	A,@R0
E7	1	MOV	A,@R1
E8	1	MOV	A,R0
E9	1	MOV	A,R1
EA	1	MOV	A,R2
EB	1	MOV	A,R3
EC	1	MOV	A,R4
ED	1	MOV	A,R5
EE	1	MOV	A,R6
EF	1	MOV	A,R7

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

Hex Code	Number of Bytes	Mnemonic	Operands
F0	1	MOVX	@DPTR,A
F1	2	ACALL	*code addr*
F2	1	MOVX	@R0,A
F3	1	MOVX	@R1,A
F4	1	CPL	A
F5	2	MOV	*data addr*,A
F6	1	MOV	@R0,A
F7	1	MOV	@R1,A
F8	1	MOV	R0,A
F9	1	MOV	R1,A
FA	1	MOV	R2,A
FB	1	MOV	R3,A
FC	1	MOV	R4,A
FD	1	MOV	R5,A
FE	1	MOV	R6,A
FF	1	MOV	R7,A

INDEX

R

S

T

U

V

X

DOMESTIC SALES OFFICES

ALABAMA

Intel Corp.
5015 Bradford Drive
Suite 2
Huntsville 35805
Tel: (205) 830-4010

ARIZONA

Intel Corp.
11225 N. 28th Drive
Suite 214D
Phoenix 85029
Tel: (602) 869-4980

Intel Corp.
1161 N. El Dorado Place
Suite 301
Tucson 85715
Tel: (602) 299-6815

CALIFORNIA

Intel Corp.
21515 Vanowen Street
Suite 116
Canoga Park 91303
Tel: (818) 704-8500

Intel Corp.
2250 E. Imperial Highway
Suite 218
El Segundo 90245
Tel: (213) 640-6040

Intel Corp.
1510 Arden Way, Suite 101
Sacramento 95815
Tel: (916) 920-8096

Intel Corp.
4350 Executive Drive
Suite 105
San Diego 92121
Tel: (619) 452-5880

Intel Corp.*
2000 East 4th Street
Suite 100
Santa Ana 92705
Tel: (714) 835-9642
TWX: 910-595-1114

Intel Corp.*
San Tomas 4
2700 San Tomas Expressway
Santa Clara, CA 95051
Tel: (408) 986-8086
TWX: 910-338-0255

COLORADO

Intel Corp.
3300 Mitchell Lane, Suite 210
Boulder 80301
Tel: (303) 442-8088

Intel Corp.
4445 Northpark Drive
Suite 100
Colorado Springs 80907
Tel: (303) 594-6622

Intel Corp.*
650 S. Cherry Street
Suite 915
Denver 80222
Tel: (303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corp.
26 Mill Plain Road
Danbury 06810
Tel. (203) 748-3130
TWX: 710-456-1199

EMC Corp.
222 Summer Street
Stamford 06901
Tel: (203) 327-2934

FLORIDA

Intel Corp.
242 N. Westmonte Drive
Suite 105
Altamonte Springs 32714
Tel: (305) 869-5588

Intel Corp.
6363 N.W. 6th Way, Suite 100
Ft. Lauderdale 33309
Tel: (305) 771-0600
TWX: 510-956-9407

FLORIDA (Cont'd)

Intel Corp.
11300 4th Street North
Suite 170
St. Petersburg 33702
Tel: (813) 577-2413

GEORGIA

Intel Corp.
3280 Pointe Parkway
Suite 200
Norcross 30092
Tel: (404) 449-0541

ILLINOIS

Intel Corp.*
300 N. Martingale Road, Suite 400
Schaumburg 60172
Tel: (312) 310-8031

INDIANA

Intel Corp.
8777 Purdue Road
Suite 125
Indianapolis 46268
Tel: (317) 875-0623

IOWA

Intel Corp.
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52402
Tel: (319) 393-5510

KANSAS

Intel Corp.
8400 W. 110th Street
Suite 170
Overland Park 66210
Tel: (913) 345-2727

MARYLAND

Intel Corp.
7321 Parkway Drive South
Suite C
Hanover 21076
Tel: (301) 796-7500
TWX: 710-862-1944

Intel Corp.
7833 Walker Drive
Greenbelt 20770
Tel: (301) 441-1020

MASSACHUSETTS

Intel Corp.*
Westford Corp. Center
3 Carlisle Road
Westford 01886
Tel: (617) 692-3222
TWX: 710-343-6333

MICHIGAN

Intel Corp.
7071 Orchard Lake Road
Suite 100
West Bloomfield 48033
Tel: (313) 851-8096

MINNESOTA

Intel Corp.
3500 W. 80th Street
Suite 360
Bloomington 55431
Tel: (612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corp.
4203 Earth City Expressway
Suite 131
Earth City 63045
Tel: (314) 291-1990

NEW JERSEY

Intel Corp.*
Parkway 109 Office Center
328 Newman Springs Road
Red Bank 07701
Tel: (201) 747-2233

Intel Corp.
75 Livingston Avenue
First Floor
Roseland 07068
Tel: (201) 740-0111

NEW MEXICO

Intel Corp.
8500 Menual Boulevard N.E.
Suite B 295
Albuquerque 87112
Tel: (505) 292-8086

NEW YORK

Intel Corp.*
300 Vanderbilt Motor Parkway
Hauppauge 11788
Tel: (516) 231-3300
TWX: 510-227-6236

Intel Corp.
Suite 2B Hollowbrook Park
15 Myers Corners Road
Wappinger Falls 12590
Tel: (914) 297-6161
TWX: 510-248-0060

Intel Corp.
850 Cross Keys Office Park
Fairport 14450
Tel: (716) 425-2750
TWX: 510-253-7391

NORTH CAROLINA

Intel Corp.
5700 Executive Center Drive
Suite 213
Charlotte 28212
Tel: (704) 568-8966

Intel Corp.
2700 Wycliff Road
Suite 102
Raleigh 27607
Tel: (919) 781-8022

OHIO

Intel Corp.*
3401 Park Center Drive
Suite 220
Dayton 45414
Tel: (513) 890-5350
TWX: 810-450-2528

Intel Corp.*
25700 Science Park Drive
Beachwood 44122
Tel: (216) 464-2736
TWX: 810-427-9298

OKLAHOMA

Intel Corp.
6801 N. Broadway
Suite 115
Oklahoma City 73116
Tel: (405) 848-8086

OREGON

Intel Corp.
15254 N.W. Greenbrier Parkway, Bldg. B
Beaverton 97006
Tel: (503) 645-8051
TWX: 910-467-8741

PENNSYLVANIA

Intel Corp.
1513 Cedar Cliff Drive
Camphill 17011
Tel: (717) 737-5035

Intel Corp.*
455 Pennsylvania Avenue
Fort Washington 19034
Tel: (215) 641-1000
TWX: 510-661-2077

Intel Corp.*
400 Penn Center Boulevard
Suite 610
Pittsburgh 15235
Tel: (412) 823-4970

PUERTO RICO

Intel Microprocessor Corp.
South Industrial Park
P.O. Box 910
Las Piedras 00671
Tel: (809) 733-3030

TEXAS

Intel Corp.
313 E. Anderson Lane
Suite 314
Austin 78752
Tel: (512) 454-3628

Intel Corp.*
12300 Ford Road
Suite 380
Dallas 75234
Tel: (214) 241-8087
TWX: 910-860-5617

Intel Corp.*
7322 S.W. Freeway
Suite 1490
Houston 77074
Tel: (713) 988-8086
TWX: 910-881-2490

Industrial Digital Systems Corp.
5925 Sovereign
Suite 101
Houston 77036
Tel: (713) 988-9421

UTAH

Intel Corp.
5201 Green Street
Suite 290
Murray 84123
Tel: (801) 263-8051

VIRGINIA

Intel Corp.
1603 Santa Rosa Road
Suite 109
Richmond 23288
Tel: (804) 282-5668

WASHINGTON

Intel Corp.
155-108 Avenue N.E.
Suite 386
Bellevue 98004
Tel: (206) 453-8086
TWX: 910-443-3002

Intel Corp.
408 N. Mullan Road
Suite 102
Spokane 99206
Tel. (509) 928-8086

WISCONSIN

Intel Corp.
450 N. Sunnyslope Road
Suite 130
Chancellory Park 1
Brookfield 53005
Tel: (414) 784-8087

CANADA

BRITISH COLUMBIA

Intel Semiconductor of Canada, Ltd.
301-2245 W. Broadway
Vancouver V6K 2E4
Tel: (604) 738-6522

ONTARIO

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Tel: (613) 829-9714
TELEX: 053-4115

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Tel: (416) 675-2105
TELEX: 06983574

QUEBEC

Intel Semiconductor of Canada, Ltd.
620 St. Jean Boulevard
Pointe Claire H9R 3K3
Tel: (514) 694-9130
TWX: 514-694-9134

*Field Application Location