



User Manual for
Basic 52 v1.1
C4x Extensions v1.13

Manual Revision 1.01

Table of Contents

TABLE OF CONTENTS	2
CHAPTER 1 > INTRODUCTION	7
1.1 INTRODUCTION TO C4x BASIC-52 WITH MODBUS.....	7
1.2 GETTING STARTED.....	7
1.3 WHAT HAPPENS AT POWER ON.....	8
1.4 DEFINITION OF TERMS:	9
<i>COMMANDS:</i>	9
<i>STATEMENTS:</i>	9
<i>FORMAT STATEMENTS:</i>	9
<i>FLOATING POINT DATA FORMAT:</i>	9
<i>INTEGER DATA FORMAT:</i>	10
<i>CONSTANTS:</i>	10
<i>OPERATORS:</i>	10
<i>VARIABLES:</i>	10
<i>SCALAR and DIMENSIONED VARIABLES</i>	11
<i>EXPRESSIONS:</i>	11
<i>RELATIONAL EXPRESSIONS:</i>	11
<i>SPECIAL FUNCTION OPERATORS:</i>	11
<i>SYSTEM CONTROL VALUES:</i>	11
1.5 STACK STRUCTURE.....	12
<i>ARGUMENT STACK</i>	12
<i>CONTROL STACK</i>	12
<i>INTERNAL STACK</i>	12
1.6 LINE EDITOR:.....	12
CHAPTER 2 > BASIC-52 GENERAL COMMANDS.....	13
<i>CONT COMMAND</i>	13
<i>LIST COMMAND</i>	14
<i>LIST# COMMAND</i>	15
<i>LIST@ COMMAND</i>	15
<i>NEW COMMAND</i>	15
<i>NULL COMMAND</i>	16
<i>RUN COMMAND</i>	16
CHAPTER 3 > BASIC-52 FILE COMMANDS	17
FLASH MEMORY PROGRAM STORAGE/RETRIEVAL.....	17
<i>ERASE COMMAND</i>	17
<i>PROG COMMAND</i>	18
<i>RAM, ROM, RROM COMMANDS</i>	19
<i>XFER COMMAND</i>	19
CHAPTER 4 > BASIC-52 STATEMENTS	20
<i>BAUD STATEMENT</i>	20
<i>CALL STATEMENT</i>	21
<i>CLEAR STATEMENT</i>	21
<i>CLEARI, CLEARS STATEMENTS</i>	22
<i>CLOCK1, CLOCK0 STATEMENTS</i>	23
<i>DATA-READ-RESTORE STATEMENTS</i>	24
<i>DIM STATEMENT</i>	25
<i>DO-UNTIL STATEMENTS</i>	26

<i>DO-WHILE STATEMENTS</i>	27
<i>END STATEMENT</i>	28
<i>FOR-TO-STEP-NEXT STATEMENTS</i>	29
<i>GOSUB-RETURN STATEMENTS</i>	30
<i>GOTO STATEMENT</i>	31
<i>ON-GOTO, ON-GOSUB-RETURN STATEMENTS</i>	31
<i>IDLE STATEMENT</i>	32
<i>IF-THEN-ELSE STATEMENTS</i>	33
<i>INPUT STATEMENT</i>	35
<i>LET STATEMENT</i>	37
<i>ONERR STATEMENT</i>	38
<i>ONEX1 STATEMENT</i>	38
<i>ONTIME STATEMENT</i>	39
<i>PRINT, TAB, SPC, CR STATEMENTS</i>	41
<i>PRINT USING () STATEMENT</i>	42
<i>PRINT USING (#.#) STATEMENT</i>	43
<i>PRINT# STATEMENT</i>	44
<i>PH0, PH1, PH0#, PH1# STATEMENTS</i>	44
<i>PRINT@, PH0.@, PH1.@ STATEMENTS</i>	45
<i>PUSH STATEMENT</i>	46
<i>POP STATEMENT</i>	47
<i>PWM STATEMENT</i>	48
<i>RBANK STATEMENT</i>	49
<i>REM STATEMENT</i>	50
<i>RETI STATEMENT</i>	51
<i>RRROM STATEMENT</i>	51
<i>ST@, LD@ STATEMENTS</i>	52
<i>STOP STATEMENT</i>	53
<i>STRING STATEMENT</i>	54
<i>UI1, UI0 STATEMENTS</i>	55
<i>UO1, UO0 STATEMENTS</i>	56
CHAPTER 5 > ARITHMETIC AND LOGIC OPERATORS AND EXPRESSIONS	57
5.1 DUAL OPERAND OPERATORS.....	57
+ ADDITION OPERATOR.....	57
** EXPONENTIATION OPERATOR	57
* MULTIPLICATION OPERATOR	57
- SUBTRACTION OPERATOR.....	57
.AND. LOGICAL AND OPERATOR.....	58
.OR. LOGICAL OR OPERATOR.....	58
.XOR. LOGICAL EXCLUSIVE OR OPERATOR.....	58
COMMENTS ON LOGICAL OPERATORS .AND., .OR., and .XOR.....	58
5.2 UNARY OPERATORS-GENERAL PURPOSE.....	59
ABS([expr]).....	59
NOT([expr]).....	59
INT([expr]).....	59
SGN([expr])	59
SQR([expr]).....	59
RND	60
PI.....	60
5.3 UNARY OPERATORS-LOG FUNCTIONS.....	61
LOG([expr]).....	61
EXP([expr]).....	61
5.4 UNARY OPERATORS-TRIG FUNCTIONS.....	62
SIN([expr])	62

<i>COS([expr])</i>	62
<i>TAN([expr])</i>	62
<i>ATN([expr])</i>	62
<i>COMMENTS ON TRIG FUNCTIONS</i>	63
5.5 UNDERSTANDING PRECEDENCE OF OPFRATORS	64
5.6 HOW RELATIONAL EXPRESSIONS WORK.....	65
CHAPTER 6 > STRING OPERATORS DESCRIPTION	66
6.1 WHAT ARE STRINGS?.....	66
6.2 THE ASC OPERATOR	67
6.3 THE CHR OPERATOR	69
CHAPTER 7 > SPECIAL OPERATORS DESCRIPTION.....	70
7.1 SPECIAL OPERATORS FOR BASIC-52 MEMORY	70
<i>CBY([expr])</i>	70
<i>DBY([expr])</i>	70
<i>XBY([expr])</i>	71
7.2 SPECIAL OPERATORS FOR BASIC-52 INTERNAL OPERATIONS	72
<i>GET</i>	72
<i>TIME</i>	73
<i>XTAL</i>	73
7.3 SPECIAL OPERATORS FOR THE 8052 SFRS	74
<i>IE</i>	74
<i>IP</i>	74
<i>PORT1</i>	74
<i>PCON</i>	74
<i>RCAP2</i>	75
<i>T2CON</i>	75
<i>TCON</i>	75
<i>TMOD</i>	75
<i>TIMER0</i>	76
<i>TIMER1</i>	76
<i>TIMER2</i>	76
7.4 EXAMPLES OF MANIPULATING SPECIAL OPERATOR VALUES	77
7.5 SYSTEM CONTROL VALUES.....	78
<i>FREE</i>	78
<i>LEN</i>	78
<i>MTOP</i>	78
CHAPTER 8 > ERROR MESSAGES, CONTROL-C, DMA AND AUTO-RUN	79
8.1 ERROR MESSAGES.....	79
8.1.1 BAD SYNTAX.....	79
8.1.2 BAD ARGUMENT.....	79
8.1.3 ARITH. UNDERFLOW	79
8.1.4 ARITH. OVERFLOW.....	79
8.1.5 DIVIDE BY ZERO.....	80
8.1.6 NO DATA.....	80
8.1.7 CANT CONTINUE.....	80
8.1.8 PROGRAMMING.....	80
8.1.9 A-STACK	80
8.1.10 C-STACK.....	80
8.1.11 I-STACK	80
8.1.12 ARRAY SIZE	81
8.1.13 MEMORY ALLOCATION	81

8.2	DISABLING CONTROL-C.....	82
8.3	IMPLEMENTING "FAKE DMA".....	83
8.4	AUTO-RUN OPTION.....	83
CHAPTER 9 > ANOMALIES OR "BUGS"		84
CHAPTER 10 > INTERRUPTS, RESOURCE ALLOCATION		85
10.1	BASIC-52 INTERRUPTS.....	85
10.2	8052 HARDWARE RESOURCE ALLOCATION.....	86
CHAPTER 11 > SYSTEM CONFIGURATION		87
11.1	MEMORY/HARDWARE CONFIGURATION.....	87
	<i>RAM ONLY MODE</i>	87
	<i>RAM/FLASH MODE</i>	87
	<i>MEMORY/HARDWARE CONFIGURATION</i>	88
	<i>FLASH PROGRAMMING CONFIGURATION/TIMING</i>	88
	<i>SERIAL PORT IMPLEMENTATION</i>	88
CHAPTER 12 > BASIC-52 RESET OPTIONS.....		89
CHAPTER 13 > ADDING COMMANDS TO BASIC-52.....		90
13.1	COMMAND/STATEMENT KEYWORDS AND TOKENS.....	93
CHAPTER 14 > USER CODE MEMORY MAP AND VECTORS		94
APPENDIX A		96
A.1	BASIC-52 MEMORY USAGE.....	96
	<i>INTERNAL MEMORY ALLOCATION:</i>	96
	<i>EXTERNAL MEMORY ALLOCATION</i>	98
A.2	USING THE PWM STATEMENT.....	99
A.3	BAUD RATES AND CRYSTALS.....	102
A.4	QUICK REFERENCE.....	103
	<i>COMMANDS:</i>	103
	<i>STATEMENTS:</i>	104
	<i>OPERATORS-DUAL OPERAND:</i>	107
	<i>OPERATORS-SINGLE OPERAND:</i>	107
	<i>OPERATORS-SPECIAL FUNCTION:</i>	108
	<i>STORED CONSTANT:</i>	108
A.5	INSTRUCTION SET SUMMARY.....	109
A.6	FLOATING POINT FORMAT.....	110
A.7	VARIABLE AND STRING MEMORY ALLOCATION.....	111
	<i>LOCATION (H-L) NAME DESCRIPTION</i>	111
	<i>LOCATION VALUE DESCRIPTION</i>	112
A.8	FORMAT OF A BASIC-52 PROGRAM.....	113
	<i>LINE FORMAT</i>	113
	<i>EPROM (FLASH) FILE FORMAT</i>	114
A.9	ANSWERS TO A FEW QUESTIONS.....	115
APPENDIX B		116
B.1	C4x BASIC-52 EXTENSIONS.....	116
	<i>Unsupported Basic-52 Features</i>	116
	<i>Quick Start Information</i>	116
B.2	APPLICATION PROGRAM STORAGE.....	117
	<i>ERASE <slot num></i>	117

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

<i>NEW</i>	117
<i>PROG</i>	117
<i>ROM</i> <slot num> / <i>XFER</i>	117
<i>RROM</i> <slot num>	117
B.3 KEYPAD	118
<i>Keypad Example</i> :.....	118
B.4 LIQUID CRYSTAL DISPLAY (LCD)	119
<i>LC Display Backlight control</i>	119
<i>LCD Print Formatting</i>	119
B.5 MODBUS REGISTERS	120
<i>REGREAD</i> [regnumber]- <i>The Modbus Register Read Command</i>	120
<i>REGWRITE</i> [regnumber],[value]- <i>The Modbus Register Write Command</i>	120
B.6 MODBUS ERRORS	121
<i>ERRNO</i> - <i>The Modbus Error Number Comand</i>	121
<i>Error Codes returned by ERRNO</i> :	121

CHAPTER 1 > Introduction

1.1 INTRODUCTION TO C4x BASIC-52 with MODBUS

Welcome to the C4x BASIC-52 and Modbus Operating System. This Operating System (OS) functions as a BASIC interpreter with Modbus functionality occupying 16K of FLASH Memory in an INTEL compatible 8052 microcontroller. BASIC-52 provides most of the features of "standard" Basic Interpreters, plus many additional features, including the Modbus RTU protocol and functions for Automation environments.

The design goal of C4x BASIC-52 was to provide a software program that would make it easy for a hardware/software designer to interact with the C4x 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 or other embedded operating systems. It is a powerful software tool that can significantly reduce the design time and increase the reliability of many projects. C4x BASIC-52 with Modbus is ideal for so called embedded control systems, where remote controllers are attached to systems to control, monitor and manage equipment and data.

C4x BASIC-52 offers many unique hardware and software features, including the ability to reliably store and execute the user program out of FLASH Memory, the ability to process interrupts within the constructs of a BASIC program, plus an accurate Timer and Real Time Clock. In addition, the arithmetic routines and I/O routines contained in C4x BASIC-52 can be accessed with Modbus and 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" or 'How to Use Modbus' manual. Many excellent texts on these subjects are available.

The descriptions of many of the statements in this manual involve rather detailed discussions that relate to interfacing BASIC-52 and Modbus programs for the C4x hardware. If the user is not interested in using Programming languages with the C4x, these discussions may be ignored since the device can be used without programs. If you are only interested in programming the C4x device in BASIC, you can treat all statements the same way they would be in any standard BASIC interpreter. The Modbus system is also industry standard and operates independent of the BASIC-52 system as well as in conjunction with BASIC-52 if desired.

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 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 the Interpreter and C4x device work, connect the device to a suitable Terminal Program set for 19,200bps, N, 8, 1 via RS232 cable, apply power, and start typing your Basic Program!

After power is applied to the C4x device, the I/O Processor initializes the hardware and goes into a 'Ready >' loop awaiting your instructions.

! >> See Appendix B for the C4x Extensions and modifications to Basic-52 before attempting to program the C4x device.

1.3 WHAT HAPPENS AT POWER ON

After Power On or Reset,:

- 1) BASIC-52 initializes all operating memory and sets up for operation on Com 0
- 2) Modbus Operating System Initializes all setup parameters and sets up for operation on Com 1
- 3) Modem processor initializes and sets up for operation on Com 2
- 4) BASIC-52 checks Register 4807 for Auto-Run of Program in Slot 1

At this point, the PWR and CHG LEDs should be flashing to indicate the I/O Operating systems are running and all other LEDs should have cycled through tests and are all Off indicating the Modem initialized OK.

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 using the Timer.

The Modbus OS should now be scanning every 10 milliseconds for Modbus activity on Com 1 and via BASIC-52.

BASIC-52 initializes the 8052 Special Function Registers (SFRs), TMOD, TCON, and T2CON with the following values:

```
TCON-244 (0F4H)
TMOD-17 (11H)
T2CON-52 (34H)
```

After Reset the Com 0 console device should display the following:

```
Control Design C4X vx.xx CC
READY
>
```

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

```
>PRINT XTAL, TMOD, TCON, T2CON
```

BASIC should respond :

```
11059200 17 244 52
```

If it does, everything is working properly. If it does not make sure that the Com 0 serial port is connected to the computer serial port and is running a Terminal program set for 19200bps,N,8,1, and the 12v Power is connected and working.

[See the C4x User Manual for further setup information.](#)

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 C4x BASIC-52. You might want to use this section to gain a quick understanding, of the C4x BASIC-52 language. Those of you who are familiar with the BASIC language will notice that most of the STATEMENTS and COMMANDS used in C4x BASIC-52 are "standard," so getting started should not be a problem. The COMMANDS that will differ significantly from 'standard' BASICs are related to Modbus.

The LCD, Keypad, Flash Memory, Real Time Clock and Modbus Registers and Functions are the non-standard Extensions added to BASIC-52. These Extensions allow use of the C4x controller functions via Radio links and locally as a standard Modbus device in addition to allowing complete programmability for local, on-site management.

The user can conveniently write and upload programs to the BASIC system via Com 0. The Modbus access is via Com 1 and the built-in Radio Telemetry Link in the C4x device.

1.4 DEFINITION OF TERMS:

COMMANDS:

BASIC-52 operates in two modes, the COMMAND or Direct mode and the RUN or Interpreter mode. BASIC-52 Commands can only be entered when the processor is in the COMMAND or Direct mode. 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 BASIC-52: ASSIGNMENTS, INPUT/OUTPUT, and CONTROL. The DESCRIPTION OF STATEMENTS 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 79 characters.
- 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.

FLOATING POINT DATA FORMAT:

The range of numbers that can be represented in BASIC-52 is:
+-1E-127 to +-.99999999E+127.

There are eight digits of significance in 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, 0A6EH, 1.23456E+ 3

INTEGER DATA FORMAT:

In BASIC-52, integers are numbers that range 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 as 0A000H). When an operator, such as .AND, requires an integer, BASIC-52 will truncate the fraction portion of number so it will fit the integer format. All line numbers used by 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. The brackets are NOT entered when typing the actual number or variable.

CONSTANTS:

A constant is a real number that ranges from +1E-127 to+.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.

VARIABLES:

In BASIC-52 a variable can 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)).

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 are as follow:

```
FRED, VOLTAGE1, I_11, ARRAY(ELE_1)
```

When using expanded variable names in BASIC-52 it is important to note that :

- 1) It takes longer for BASIC-52 to process these expanded variable names.
- 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.
- 3) Variables with the same first, last characters and length are considered the same in BASIC-52.

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. 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.

SCALAR and DIMENSIONED VARIABLES

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 DIM. This document will refer to VARIABLES as: [var].

Relative to a Dimensioned variable, it takes 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.

Scalar variables also require that the first and last characters differ when the length of the variables match. Variables that start and end with the same character and are the same length match in BASIC-52. BASIC-52 only reads the first, last and length as the variable. The remainder of the variable name is for the users convenience.

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. $12*EXP(A)/100$, $H(I)+55$, 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] .

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 (SFRs) on the 8052 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 OPERATORS are XTAL and TIME. Details of the SPECIAL OPERATORS are covered in the section- SPECIAL 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.

1.5 STACK STRUCTURE

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 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 BASIC-52 to ASSEMBLY language routines.

ARGUMENT STACK

The Argument Stack occupies locations 301 (12DH) through 510 (1FEH) in external ram memory. This stack stores all constants that BASIC-52 is currently using. Operations such as ADD, SUBSTRACT, 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 stack. Each floating point number placed on the Argument Stack requires 6 BYTES of storage.

CONTROL STACK

The Control Stack occupies locations 96 (60H) through 254 (0FEH) 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 (0FEH) and "grows down."

INTERNAL STACK

The stack pointer on the 8052 (Special Function Register (SFR), SP) is initialized to 77 (4DH). The 8052's stack pointer "grows up" as values are placed on the stack. In BASIC-52 the user has the option of placing the 8052's Stack Pointer anywhere (above location 77) in internal memory.

1.6 LINE EDITOR:

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 BACKSPACE 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.

Control-Q (X-ON) and Control S (X-OFF) are used for the serial port. The user is cautioned not to accidentally type a Control-S when entering information because the 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 (ENTER) is indicated by the symbol (cr). The carriage return is the RETURN or ENTER key on most keyboards.

CHAPTER 2 > BASIC-52 GENERAL COMMANDS

CONT COMMAND

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
```

LIST COMMAND

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 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 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
```

LIST# COMMAND

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 BASIC-52 device.

LIST@ COMMAND

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 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 accidentally 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 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.

NEW COMMAND

COMMAND: NEW(cr)

ACTION TAKEN:

When NEW(cr) is entered, 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.

NULL COMMAND

COMMAND: NULL [integer](cr)

ACTION TAKEN:

The NULL[integer] (cr) command determines how many NULL characters (00H) 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 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.

RUN COMMAND

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 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), 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
>30 NEXT
>RUN

1
2
3

READY
>
```

CHAPTER 3 > BASIC-52 FILE COMMANDS

FLASH MEMORY PROGRAM STORAGE/RETRIEVAL

One of the unique and powerful features of C4x BASIC-52 is that it has the ability to SAVE and Execute programs in Flash Memory. C4x BASIC-52 generates all of the timing signals needed to program the internal Flash memory of the Operating System. Saving programs in Flash is a reliable and simple alternative to Eproms, especially in control and/or noisy environments.

BASIC-52 can save multiple programs or one program in Flash. In fact, it can save as many programs or as large a program as the size of the Flash memory permits. The programs are stored sequentially in Flash and any program can be retrieved and executed (ROM, XFER and RUN or RROM). The sequential storing (PROG) and erasing (ERASE) of programs is referred to as the Program FILE or SLOT commands.

ERASE COMMAND

COMMAND : ERASE [integer]

ACTION TAKEN :

The ERASE command allows the exact File or Slot to be erased where a program File is stored. If the user desires to save programs in an exact File or Slot, the PROG command will work in conjunction with the ERASE command to allow program storage where a previous file has been ERASEd before using PROG. ERASE without an [integer] will default to File or ROM Slot 1.

EXAMPLE :

To select ROM File or Slot 4 for a PROG command to use the same File, assuming ROM Slots 1 thru 3 have no Program saved in those locations, the user must write a simple program and save it to the first three File Slots before being able to use Slot 4.

```
> 10 PRINT 'File Slot 1'  
> PROG  
1
```

```
> 10 PRINT 'File Slot 2'  
> PROG  
2
```

```
> 10 PRINT 'File Slot 3'  
> PROG  
3
```

The next slot available is 4. Clear RAM by typing 'NEW' then enter or upload the program to be saved to Flash. When PROG is executed again :

```
> PROG  
4
```

```
READY  
>
```

To save the next program to File slot 1 :

```
> ERASE 1
```

Enter the program to save then :

```
> PROG  
1
```

PROG COMMAND

COMMAND: PROG(cr)

ACTION TAKEN:

The PROG COMMAND programs the resident Flash with the current selected program. The current selected program may reside in either RAM or FLASH.

Since no file slot can be specified with PROG, the next available file or Slot is used to save the program. See the ERASE command to select the file slot PROG can be forced to use.

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

EXAMPLE:

Type in the following sample program :

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

Type in the PROG command

```
>PROG
```

```
6          REM BASIC-52 displays the file slot the program was saved in.
```

Select Flash slot (file) 6

```
>ROM 6
```

```
READY
```

```
>LIST
```

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

In this example, the program just placed in Flash is the 6th program stored.

VARIATIONS: None.

RAM, ROM, RROM COMMANDS

COMMANDS: RAM, ROM [integer], RROM [integer]

ACTION TAKEN:

These commands tell the C4x 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 FLASH. The RAM address is assumed to be 512 (200H) and the FLASH address begins at 32784 (8010H).

RAM

When RAM(cr) is entered C4x 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, RROM

When ROM [integer] (cr) is entered BASIC-52 selects the current program in Flash memory. If no integer is typed after the ROM command (i.e. ROM (cr)) C4x BASIC-52 defaults to ROM 1. Since the programs are stored sequentially in Flash, the integer following the ROM command selects which program the user wants to RUN or LIST (if transferred (XFER) to Ram).

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 Flash) the message ERROR: PROM MODE will be displayed.

When RROM [integer] (cr) is entered in Direct or Run modes, BASIC-52 selects the current program in Flash memory, Transfers the program to RAM (XFER) and Runs the program (RUN) all at once. If no [integer] is specified, BASIC-52 defaults to ROM slot 1.

BASIC-52 does not transfer the program from Flash 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 Flash 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 Flash.

The SYSTEM CONTROL VALUES -MTOP and FREE always refer to RAM not Flash.

VARIATIONS: None.

XFER COMMAND

COMMAND: XFER(cr)

ACTION TAKEN:

The XFER (transfer) command transfers the current selected program in Flash to RAM and then selects the RAM mode. If XFER is typed while C4x 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 are 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.

CHAPTER 4 > BASIC-52 STATEMENTS

BAUD STATEMENT

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 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). 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 baudrate. 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.

CALL STATEMENT

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 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 24104H 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.

CLEAR STATEMENT

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 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.

CLEARI, CLEARS 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 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 (0FEH) 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.

CLOCK1, CLOCK0 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 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.

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, 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 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 BASIC-52 statement that can disable the REAL TIME CLOCK. CLEAR and CLEARI will NOT disable the REAL TIME CLOCK.

VARIATIONS: None.

DATA-READ-RESTORE 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 RRINT A,B
>40 NEXT I
>50 RESTORE
>60 READ A,B
>770 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.

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.

DIM STATEMENT

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 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 redimension 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)
```

DO-UNTIL STATEMENTS

STATEMENTS: DO-UNTIL [rel expr]

MODE: RUN

TYPE: CONTROL

The DO-UNTIL [rel expr] instruction provides a means of "loop control" within an 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

DO-WHILE STATEMENTS

STATEMENTS: DO-WHILE [rel expr]

MODE: RUN

TYPE: CONTROL

The DO-WHILE [rel expr] instruction provides a means of "loop control" within an 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
```

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
2
3
4
DONE
>
1 1 1
1 2 2
1 3 3
2 1 2
2 2 4
2 3 6
3 2 6
3 3 9
READY
>
```

VARIATIONS: None

END STATEMENT

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 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

FOR-TO-STEP-NEXT STATEMENTS

STATEMENTS: FOR-TO-{STEP}-NEXT

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

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

EXAMPLE:

```
10 FOR A=3 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 BASIC-52. Unlike some BASICS, the index MAY NOT be omitted from the NEXT STATEMENT in BASIC-52 (i.e. the NEXT statement MUST always be followed by the appropriate variable).

EXAMPLES:

>10 FOR I=1 TO 4	>10 FOR I=0 TO 8 STEP 2
>20 PRINT I	>20 PRINT I
>30 NEXT I	>30 NEXT I
>RUN	>RUN
1	0
2	2
3	4
4	6
	8
READY	
>	READY
	>

In BASIC-52 it is possible to 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 the NEXT statement can 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 BASIC-52. The variable associated with the NEXT is always assumed to be the variable used in the last FOR statement.

GOSUB-RETURN STATEMENTS

STATEMENTS: GOSUB[In num]-RETURN

MODE: RUN

TYPE: CONTROL

GOSUB

The GOSUB [In num] STATEMENT will cause BASIC-52 to transfer control of the program directly to the line number ([In 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
6
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
>
```

NOTE-The Control Stack permits a graceful exit from incompleting control loops, given the following **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
```

This would permit the programmer to exit the subroutine even though the FOR-NEXT loop might not be allowed to complete if X did equal 1.

GOTO STATEMENT

STATEMENT: GOTO [In num]

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

The GOTO [In num] STATEMENT will cause BASIC to transfer control directly to the line number ([In 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, 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.

ON-GOTO, ON-GOSUB-RETURN STATEMENTS

**STATEMENTS: ON [expr] GOTO[In num], [In num], . . . [In num]
ON [expr] GOSUB[In num], [In num], . . . [In num]
RETURN**

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 BASIC-52 program.

IDLE STATEMENT

STATEMENT: IDLE

MODE: RUN

TYPE: CONTROL

The IDLE statement forces the BASIC-52 device into a "wait until interrupt mode." Execution of statements is halted until either an ONTIME [expr], [In num] or an ONEX1 [In 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 BASIC-52 device will enter a "wait forever mode" and for all practical purposes the system will have crashed.

When an ONTIME [expr], [In num] or an ONEX1 [In num] is received while in the IDLE mode, the 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 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 BASIC-52 device. When the 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 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.

IF-THEN-ELSE 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:

```
[In 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 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.

COMMENTS ON IF-THEN-ELSE

EXAMPLE 1:

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

EXAMPLE 2:

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

BASIC-52 executes the remainder of line 10 if and only if the test $A=B$ proves to be true. This means in EXAMPLE 1 IF A did equal B, Basic-52 would then set $C=A$, then set $A=A/2$, then execute line 100. IF A did not equal B, Basic-52 would then PRINT A and ignore the statements $C=A$: $A=A/2$: GOTO 100. This same logical interpretation holds true for the ELSE statement as well.

INPUT STATEMENT

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 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
```

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, BASIC-52 will prompt the user with a question mark.

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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
>
```

LET STATEMENT

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)

ONERR STATEMENT

STATEMENT: ONERR[In num]

MODE: RUN

TYPE: CONTROL

The ONERR[In 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 BASIC-52 interpreter will pass control to the line number following the ONERR[In num] statement. The programmer can handle the error condition in any manner suitable to the particular application. Typically, the ONERR[In 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[In 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.

ONEX1 STATEMENT

STATEMENT: ONEX1 [In num]

MODE: RUN

TYPE: CONTROL

The ONEX1 [In num] statement lets the user handle interrupts on the I/O Processor's INT1 pin with a BASIC program. The line number following the ONEX1 statement tells the 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 I/O Processor 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 8052's Interrupt Enable Register (IE). Before an interrupt can be processed, the BASIC-52 interpreter must complete execution of the statement it is currently processing. Because of this, interrupt latency can vary from microseconds to tens of milliseconds. The ONTIME [expr], [In num] interrupt has priority over the ONEX1 interrupt. So, the ONTIME interrupt can interrupt the ONEX1 interrupt routine.

ONTIME STATEMENT

STATEMENT: ONTIME [expr], [In num]

MODE: RUN

TYPE: CONTROL

Since BASIC-52 processes a line in the millisecond time frame and the timer/counters on the 8052 operate in the microsecond time frame, there is an inherent incompatibility between the timer/counters on the 8052 and BASIC-52. To help solve this situation the ONTIME [expr], [In 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 ([In num]) following the expression ([expr]) in the ONTIME statement.

Since the ONTIME statement uses the BASIC-52 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 - " ".

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.

ONTIME 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 interrupts.

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 CLEAR1 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 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.

PRINT, TAB, SPC, CR STATEMENTS

STATEMENT: PRINT or P. or ?

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

The PRINT statement directs 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. ? 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 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, 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 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.

CR

The CR function is interesting and unique to 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.

PRINT USING () STATEMENT

USING(special characters)

The USING function is used to tell BASIC-52 what format to display the values that are printed. 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 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. BASIC-52 will not output any trailing zeros, so the number of digits will vary depending upon the number. 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
>
```

PRINT USING (##) STATEMENT

USING(##) - This will force 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 BASIC-52. The maximum number of "#" characters is 8. If 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 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, BASIC-52 is placed in the USING(0) format.

PRINT# STATEMENT

STATEMENT: PRINT# or P.#, ?#

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#.

PH0, PH1, PH0#, PH1# STATEMENTS

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

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

The PH0. and PH 1. 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), 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 04H	>PH1. 2*2 0004H	>PRINT 99H 153	>PH0. 100 64H
>PH0. 1000 3E8H	>PH1. 1000 03E8H	>P. 3E8H 1000	>PH0. PI 03H

PRINT@, PH0.@, PH1.@ STATEMENTS

STATEMENT: PRINT@, PH0.@, PH1.@

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.

NOTE : The user defined output driver for the C4x devices is the Liquid Crystal Display.

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 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 accidentally 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 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.

PUSH STATEMENT

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 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 VARIABLES

```
>10 A=10
>20 B=20
>30 PRINT A,C
>40 PUSH A,C
>50 POP A,B
>60 PRINT A,B
>RUN
```

```
10 20
20 10
```

```
READY
>
```

SUBROUTINE PASSING

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

```
-1 -.5
```

```
READY
>
```

POP STATEMENT

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 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 BASIC-52.

PWM STATEMENT

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 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 "audible" 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.

RBANK STATEMENT

STATEMENT: RBANK [integer]

MODE: COMMAND and/or RUN

TYPE: CONTROL

RBANK is used to set and retrieve the current external Ram BANK. The C46 contains 128K bytes of external Static Random Access Memory (SRAM). The external RAM can be accessed in 8 banks of 16K bytes each. Banks are selected by the Integer 0 to 7.

The banked memory is located at address 8000H (or 32768 Decimal) and extends to BFFFH (or 49151 Decimal).

IMPORTANT NOTE: Banks 0 and 1 are reserved for BASIC-52 and should not normally be used by applications. Portions of Banks 2 and 3 are used for Modbus and other Operating System usage. Applications are free to read and write to banks 4 through 7. This gives applications a total of 64K of free RAM for data storage.

EXAMPLE:

To read the current RAM bank in to variable X (the value returned will be from 0 to 7):

```
10 RBANK
20 POP X
```

To set the active RAM bank to bank 2:

```
10 RBANK 2
```

To read and write data to the RAM bank:

```
REM The RAM bank base address is 32768
10 B = 32768
REM Write a 10 to the first location in the selected RAM bank
20 XBY(B)=10
REM Read the last location in the selected RAM bank to variable X
REM Each RAM bank is 16K bytes (16384 bytes)
30 X=XBY(B+16383)
```

REM STATEMENT

STATEMENT: REM

MODE: COMMAND and/or RUN

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 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 BASIC-52 device. This helps to conserve memory.

RETI STATEMENT

STATEMENT: RETI

MODE: RUN

TYPE: CONTROL

The RETI statement is used to exit from interrupts that are handled by an 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.

RROM STATEMENT

STATEMENT: RROM [integer]

MODE: COMMAND AND/OR RUN

TYPE: CONTROL

RROM stands for RUN ROM. What it does is select a program in the Flash Memory files (Slots), then execute the program. The integer after the RROM statement selects what program in the Flash 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 Flash 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 Flash file and the user enters RROM 8, or no Flash is in the system), no error will be generated and 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 BASIC-52 evoked interrupts are cleared.

ST@, LD@ STATEMENTS

STATEMENTS: ST@ [expr] and LD@ [expr]

MODE: COMMAND and/or RUN

TYPE: INPUT/OUTPUT

ST@

The ST@ [expr] statement lets the user specify where 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, 0F01H, and 0F000H.

STOP STATEMENT

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!!!

STRING STATEMENT

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. 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 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, 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.

UI1, UI0 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 BASIC-52. After UI1 is executed BASIC will call external program memory location 4033H when a console input is requested.

NOTE : The user defined input driver for the C4x devices is the Keypad.

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 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 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. 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 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 (IEH) 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 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.

UO1, UO0 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 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. 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 BASIC-52.

UO0

UO0 STATEMENT assigns the console output routine back to the software drivers resident on the 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 BASIC-52 output drivers will be used. The assembly language programmer can use this information to change the output device selection in assembly language.

CHAPTER 5 > Arithmetic and Logic Operators and Expressions

5.1 DUAL OPERAND OPERATORS

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
```

.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, 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 BASIC-52 eliminates ALL spaces when it processes a user line and inserts spaces before and after STATEMENTS when it LISTS a user program. 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 UNARY OPERATORS-GENERAL PURPOSE

ABS([expr])

Returns the ABSOLUTE VALUE of the expression.

EXAMPLES:

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

INT([expr])

Returns the integer portion of the expression.

EXAMPLES:

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

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
```

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
```

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 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.

EXAMPLE:

```
PRINT RND  
.30278477
```

PI

PI is not really an operator, it is a stored constant. In 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 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 desirable, 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.

5.3 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
```

5.4 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
```

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:

$$\text{REDUCED ARGUMENT} = (\text{user arg}/\pi - \text{INT}(\text{user arg}/\pi)) * \pi$$

The REDUCED ARGUMENT, from the above equation, will be between 0 and π . The REDUCED ARGUMENT is then tested to see if it is greater than $\pi/2$. If it is, then it is subtracted from π 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 $(\text{user arg}/\pi - \text{INT}(\text{user arg}/\pi))$ expression. As a general rule, try to keep the arguments for the TRIG functions as small as possible!

5.5 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 is 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 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.

5.6 HOW RELATIONAL EXPRESSIONS WORK

Relational expressions involve the operators =, <>, >, >=, <, and <=. These operators are typically used to "test" a condition. In 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 > String Operators Description

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.

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 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 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 BASIC-52, two operators manipulate STRINGS. These operators are ASC() and CHR(). Admittedly, the string operators contained in 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 BASIC-52 it is possible to manipulate strings in almost any way imaginable. This in itself is a commendable feat since BASIC-52 was designed primarily to be a sophisticated BASIC language oriented controller, not a string manipulator. The string operators available in 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

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 ."

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.

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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

EXAMPLE:

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

```
AECDEFGHIJKL  
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 $(1)"BASIC-52"  
>20 FOR I=1 TO 12 : PRINT CHR$(1),I), : NEXT I  
>30 PRINT : FOR I=12 TO 1 STEP -1  
>40 PRINT CHR$(1),I), : NEXT I  
>RUN  
  
MCS 3ASIC-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 Description

7.1 SPECIAL OPERATORS for BASIC-52 MEMORY

SPECIAL OPERATORS are called SPECIAL OPERATORS because they directly manipulate the I/O hardware and the memory addresses on the C4x device. All SPECIAL 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 BASIC-52. The SPECIAL FUNCTION OPERATORS in 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.

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.

7.2 SPECIAL OPERATORS for BASIC-52 INTERNAL OPERATIONS

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 selected by UI0 (C4x Com0) or UI1 (C4x Keypad).

Actually, GET 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.

TIME

The TIME operator is used to retrieve and/or assign a value to the BASIC-52 Timer. After reset, TIME is equal to 0. The CLOCK1 statement enables the Timer. When the Timer is enabled, the SPECIAL OPERATOR TIME will increment once every 5 milliseconds. The TIME operator uses TIMER0 and the interrupts associated with TIMER0 on the 8052. The Integer 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
```

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,[In 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.

XTAL

The XTAL operator tells BASIC-52 what frequency the system is operating at. The XTAL operator is used by 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.

7.3 SPECIAL OPERATORS for the 8052 SFRs

The following operators directly manipulate the 8052's Special Function Registers (SFRs). Specific details of the operation of these registers is in the 8052 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.0BFH`. The only statements in BASIC-52 that write to the IE register are the `CLOCK0`, `CLOCK1`, `ONEX1`, `CLEAR`, and `CLEARI` statements.

EXAMPLES:

```
IE=81H and A=IE.AND.0BFH
```

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`. 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.

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/COUNTER1` 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 BASIC-52 device. The following can be used to determine what BAUD rate the BASIC-52 device is operating at:

$$\text{BAUD} = \text{XTAL} / (32 * (65536 - \text{RCAP2}))$$

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. 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!

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. 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. 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. 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 BASIC-52, either timer/counter can be placed in any mode required by the specific application.

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. 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. 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. 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.

7.4 EXAMPLES OF MANIPULATING SPECIAL OPERATOR VALUES

Using the logical operators available in BASIC-52, it is possible to write to or read from any byte of the special function registers that 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 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 BASIC-52 device. Certain crystals may not be able to use TIMER1 as the baud rate generator, especially at high (above 2400) baud rates.

7.5 SYSTEM CONTROL VALUES

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

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, BASIC-52 does not require any "dummy" arguments for the SYSTEM CONTROL VALUES.

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.

MTOP

After reset, BASIC-52 sizes the external memory and assigns the last valid memory address to the SYSTEM CONTROL VALUE, MTOP. 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
```

CHAPTER 8 > Error Messages, Control-C, DMA and AUTO-RUN

8.1 ERROR MESSAGES

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:

8.1.1 BAD SYNTAX

A BAD SYNTAX error means that either an invalid 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 BASIC-52 a BAD SYNTAX ERROR is also generated if the programmer attempts to use a reserved keyword as part of a variable.

8.1.2 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.

8.1.3 ARITH. UNDERFLOW

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

8.1.4 ARITH. OVERFLOW

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

8.1.5 DIVIDE BY ZERO

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

8.1.6 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.

8.1.7 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.

8.1.8 PROGRAMMING

If an error occurs while the 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.

8.1.9 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.

8.1.10 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 BASIC-52 can handle because 9 times 17 equals 153. If the user attempts to use more control stack than is available in BASIC-52 a C- STACK error will be generated. In addition, GSTACK errors will occur if a RETURN is executed before a GOSUB, WHILE or UNTIL before a DO, or a NEXT before a FOR.

8.1.11 I-STACK

An ISTACK (INTERNAL STACK) error occurs when BASIC-52 does not have enough stack space to evaluate an expression. Normally, ISTACK 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.

8.1.12 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
```

8.1.13 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 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 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 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 BASIC-52 device does not contain an hardware mechanism that supports Direct Memory Access (DMA). However, the DMA function is supported in software by BASIC-52. During DMA operation 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 (/INT0) acts as a DMA input pin. /INT0 is pin 12 on the 8052AH. Whenever /INT0 is pulled low (to a logical zero state), the BASIC-52 device will enter the DMA mode and no accesses will be made to external memory. To acknowledge that BASIC-52 has entered the DMA mode, BASIC-52 outputs a zero on pin 7 (P1.6). In essence, PORT 1.6 is the /DMA ACK pin of the 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 BASIC-52 system. After the user pulls the /INT0 pin high, BASIC-52 will output a one on P1.6 and normal program execution will continue. During this "fake DMA" cycle, the BASIC-52 program does nothing except wait for the /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 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 BASIC-52 system.

8.4 AUTO-RUN OPTION

Version 1.1 of 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 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), 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.

The C4x Operating System, using Modbus, allows the user to write a value to Register 4x4807 via the C4x Basic-52 extension REGWRITE command to set the Basic-52 for Auto-Run on Power-Up.

Examples :

<code>REGWRITE 4807,13312</code>	<code>REM disables auto-run on power-up or reset</code>
<code>REGWRITE 4807,13313</code>	<code>REM enables auto-run on power-up or reset</code>

CHAPTER 9 > ANOMALIES or "BUGS"

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, 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 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              >L1ST
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 BASIC-52 eliminates all spaces when a line is processed. So ASC() will be stored as ASC() and this becomes an error.

#4) The Variable E, when it is the last character in a Variable name and it is followed by a space, will be discarded by BASIC-52. The E will also be discarded if used in a variable name EP, EPR, EPRO if that Variable name is followed by a space.

This is an original program downloaded to BASIC-52 :

```
100 PRINT C,D,E : PRINT C,D,E
110 PRINT E : PRINT E : PRINT EP : PRINT EP
120 PRINT EPR : PRINT EPR : EPRO=55 : EPRO = 55
```

After BASIC-52 processes the lines, note the errors :

```
100 PRINT C,D, : PRINT C,D,E
110 PRINT E : PRINT : PRINT EP : PRINT P
120 PRINT EPR : PRINT PR : EPRO=55 : PRO=55
```

#5) Do NOT use Variable names which have embedded instructions in them. For Example, JUICE=5 will generate a BAD SYNTAX error because it contains the instruction UI. Also, SNOT=777 contains the NOT instruction.

#6) Variables each require 8 bytes of external memory- 2 for the name and 6 for the value. Variables with identical first and last characters and identical length (such as ABCD9 and AFGH9) are considered to be the same Variable by BASIC-52. Variables longer than 8 characters in length and having the same first and last characters will also produce unpredictable results.

CHAPTER 10 > Interrupts, Resource Allocation

10.1 BASIC-52 INTERRUPTS

Interrupts can be handled by BASIC-52 in two distinct ways. The first, which has already been discussed, allows statements in an 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 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 BASIC-52 device are "mirrored" to external PROGRAM MEMORY LOCATIONS 4003H through 402BH inclusive. The only 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	TIMER0 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 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	UNDEFINED	TIMER2	SERIAL PORT	TIMER1	EXT1	TIMER0	EXT 0

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 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 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 BASIC-52 device can be accessed is when the CALL statement in BASIC-52 is used to transfer control to the users assembly language program. The reason for this is that the BASIC-52 interpreter must be in a "known" state before the user can call the routines available in the 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.

10.2 8052 HARDWARE RESOURCE ALLOCATION

Specific statements in 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 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.

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 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 11 > System Configuration

11.1 MEMORY/HARDWARE CONFIGURATION

BASIC-52 always requires at least 1K bytes of external memory. After reset, BASIC-52 sizes the external memory. If less than 1K bytes of external memory are available, 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 BASIC-52 device.

BASIC-52 sizes consecutive 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. 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 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 /RD (pin 17) and /WR (pin 16) pins on the 8052AH are used to enable DATA memory and /PSEN (pin 29) pin is used to enable PROGRAM memory. Depending on the hardware configuration, BASIC-52 operates in two distinct "memory" modes.

RAM ONLY MODE

In this mode of operation, Read/write memory is connected to the 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 (/CS) signal for the RAM devices. The /RD pin on the 8052AH is used to generate the OUTPUT ENABLE (/OE) strobe and the /WR pin generates the WRITE ENABLE (/WE or /WR) strobe. /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 BASIC-52 device. An example of this configuration is shown in Figure 1 (p.137). Since /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.

RAM/FLASH MODE

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

- 1) The /RD and the /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 (/CS) for the RAM devices and /RD and /WR are used to enable the /OE and /WE or (/WR) pins respectively.
- 2) The /PSEN pin on the BASIC-52 device is used to enable EPROM memory that is addressed from 2000H to 7FFFH. Addresses are used to decode the chip select (/CS) for the EPROM devices and /PSEN is used to enable the /OE pin.
- 3) For addresses between 8000H and 0FFFFH both the /RD and the /PSEN pin on the 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 /RD and the /PSEN pins to enable addresses in this address space, /RD and /PSEN must be logically "ANDed" together. This can be accomplished with a simple TTL gate such as a 74LS08. The /WR pin on the BASIC-52 device is used to write to RAM memory in this same address space. The /PSEN and /RD signals do not have to be ANDed beyond address 7FFFH to enable 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 BASIC-52 programs that are located in this address space.

MEMORY/HARDWARE CONFIGURATION

This scheme of memory addressing actually permits 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 /RD and /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 BASIC-52 is used, BASIC assumes that the EPROM to be programmed is addressed starting at location 8000H. 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, 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 (p.138, 139 and 140).

FLASH PROGRAMMING CONFIGURATION/TIMING

IMPORTANT NOTE

C4x BASIC-52 operates the read/write functions of the 8051 Internal Flash. No other routines are needed.

SERIAL PORT IMPLEMENTATION

The serial port I/O signals on the 8052 are TTL compatible signals. They are typically not compatible with most terminals. The serial port is initialized by 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.

CHAPTER 12 > BASIC-52 Reset Options

After RESET, BASIC-52 initializes the 8051 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 /PSEN), 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 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 BASIC-52 software package will clear the internal memory of the 8052 and proceed with the RESET routine.

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 simply by initializing the necessary Special Function Registers and then jumping back into either BASIC-52's COMMAND mode or RUN MODE. For a warm start-up or RESET (warm means that the 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 BASIC-52 supplies.

(RCAP2H and RCAP2L can be loaded with the proper baud rate values.)

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 accomplish the STACK POINTER initialization.

After the above are initialized by the user supplied RESET routine, the user may enter 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 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 implementing this RESET detection mechanism will not be discussed here as the possible hardware options vary depending upon the design.

CHAPTER 13 > Adding Commands to BASIC-52

BASIC-52 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 BASIC-52 device that the expansion options are available. This is done by putting the character 5AH in CODE memory location 2002H. When BASIC-52 enters the command mode it will examine CODE memory location 2002H. If a 5AH is in this location, 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 BASIC-52 that the expansion option is available. The following simple code will accomplish all that is stated above:

```
ORG 2002H
DB 5AH

ORG 2048H
SETB 45
RET
```

STEP 2

With BIT 45 SET, 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.

STEP 3

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

- 1) THE USER TOKENS ARE THE NUMBERS 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 BASIC-52 "de-tokenizes" a line during a LIST.

STEP 4

Step 3 tokenizes the user keyword, this means that 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 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
                               ;
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
                               ;
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
                               ;

```

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

```

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

The user may also use the command/statement extension option to re-define the syntax of 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 BASIC-52 tokens and keywords are provided in the back of this chapter. BASIC-52 will always list out the program using the user defined syntax, 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
D8 'DISPLAY'
D8 00H
;
; REST OF USER_TABLE
;
DB 0FFH ; END OF TABLE
```

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.

13.1 Command/Statement Keywords and Tokens

TOKEN	KEYWORD	TOKEN	KEYWORD	TOKEN	KEYWORD
80H	LET	080H	ABS	0ECH	<=
81H	CLEAR	081H	INT	0EDH	<>
82H	PUSH	0B2H	SGN	0EEH	<
83H	GOTO	083H	NOT	0EFH	>
84H	PWM	084H	COS	0FOH	RUN
85H	PH0.	085H	TAN	0FIH	LIST
86H	UI	0B6H	SIN	0F2H	NULL
87H	UO	087H	SOR	0F3H	NEW
88H	POP	088H	CBY	0F4H	CONT
89H	PRINT	089H	EXP	0F3H	PROG
89H	P.	08AH	ATN	0F6H	XFER
89H	? (V1.1 ONLY)	088H	LOG	0F7H	RAM
8AH	CALL	08CH	DBY	0F8H	ROM
88H	DIM	08DH	XYB	0F9H	FPROG
8CH	STRING	08EH	PI	0FAH-0FFH	NOT USED
8DH	BAUD	08FH	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	0C8H	T2CON		
9AH	INPUT	0CCH	TCON		
98H	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.		
0A8H	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 14 > User Code Memory Map and Vectors

External CODE memory locations that 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
2001 H	ON RESET, BASIC-52 LOOKS FOR 0AAH IN THIS LOCATION, IF PRESENT, CALLS LOCATION 2090H
2002H	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	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	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	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	BASIC-52 CALLS THIS LOCATION WHEN THE USER EVOKES THE ASSEMBLY LANGUAGE RESET OPTION
4003H	EXTERNAL INTERRUPT 0
400BH	TIMER0 INTERRUPT
4013H	EXTERNAL INTERRUPT 1
401BH	TIMER1 INTERRUPT
4023H	SERIAL PORT INTERRUPT
402BH	TIMER2 INTERRUPT
4030H	USER CONSOLE OUTPUT
4033H	USER CONSOLE INPUT
4036H	USER CONSOLE STATUS
403CH	USER PRINT@ OR LIST@ VECTOR
4100H-41FFH	USER CALLS FROM 0 TO 7FH

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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	TIMER1 INTERRUPT TRAP IF BIT 1AH OF INTERNAL RAM IS SET. PSW IS PUSHED ONTO THE STACK.

These vectors are chosen so that addresses 2000H and 4000H can be overlaid and create no conflicts. The Overlaid 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 through 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 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 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.

APPENDIX A

A.1 BASIC-52 MEMORY USAGE

The following list specifies which locations in Internal and External memory are used by BASIC-52 :

INTERNAL MEMORY ALLOCATION:

LOCATION(S) IN HEX 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 TIMER1 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

LOCATION(S) IN HEX 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

LOCATION(S) IN HEX 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	TIMER1 RELOAD LOCATION-HIGH BYTE
41H	TIMER1 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	TIMER0 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

All Specifications subject to change without notice.

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

EXTERNAL MEMORY ALLOCATION

LOCATION(S) IN HEX 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 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 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.

A.2 USING THE PWM STATEMENT

The PWM statement can be used to generate 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		IDEAL OCTAVE	ACTUAL FREQUENCY	HEX FREQUENCY	RELOAD	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	2BBOH	
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

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

NOTE	IDEAL OCTAVE	ACTUAL FREQUENCY	HEX FREQUENCY	RELOAD	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

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

NOTE	IDEAL OCTAVE	ACTUAL FREQUENCY	HEX FREQUENCY	RELOAD	RELOAD
C	7	2092.992	2094.545	220	00DCH
C#	7	2217.472	2215.385	208	00DOH
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

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

```

A.3 BAUD RATES AND CRYSTALS

The 16 bit auto-reload timer/counter (TIMER2) that is used to generate baud rates for the 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 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.

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,"/."
```

A.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	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

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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
CLEAR S	CLEAR Stacks	CLEAR S
CLEAR I	CLEAR Interrupts	CLEAR I
CLOCK1	enable BASIC-52 Timer	CLOCK1
CLOCK0	disable BASIC-52 Timer	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
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	ONERR or GOTO line number	ONERR 1000

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

STATEMENT	FUNCTION	EXAMPLE(S)
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
PRINT@	PRINT to user defined driver (C46 LCD)	PRINT@ 5*5
PH0.@	PH0. to user defined driver (C46 LCD)	PH0. @ XBY(5EH)
PH1.@	PH1. to user defined driver (C46 LCD)	PH1.@ A
PGM	Program an EPROM (Not Used)	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

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

STATEMENT	FUNCTION	EXAMPLE(S)
ST@	store top of stack at user specified location	ST@ 1000H ST@ A
LD@	load top of stack from user specified location	LD@ 1000H LD@ A
IDLE	wait for interrupt	IDLE
RROM	run a program in Flash	RROM 3
RBANK	select a 16K block of external memory	RBANK 4

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

OPERATORS-DUAL OPERAND:

OPERATOR	FUNCTION	EXAMPLE(S)
+	ADDITION	1+1
/	DIVISION	10/2
**	EXPONENTATION	2**4
*	MULTIPLICATION	4*4
-	SUBSTRACTION	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()	ONE'S 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)

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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
----	----------------	----

A.5 INSTRUCTION SET SUMMARY

<u>COMMANDS</u>	<u>STATEMENTS</u>	<u>OPERATORS</u>
RUN	BAUD	ADD (+)
CONT	CALL	DIVIDE (/)
LIST	CLEAR	EXPONENTIATION (**)
LIST#	CLEAR(S&I)	MULTIPLY (*)
LIST@	CLOCK(1&0)	SUBTRACT (-)
NEW	DATA-READ-RESTORE	LOGICAL AND (.AND.)
NULL	DIM	LOGICAL OR (.OR.)
RAM	DO-UNTIL	LOGICAL XOR (.XOR.)
ROM	DO-WHILE	LOGICAL NOT (.OR.)
XFER		ABS ()
PROG		INT ()
RROM	END	SGN ()
ERASE	FOR-TO-STEP	SQR ()
	IDLE	EXP ()
MODBUS	IF-THEN-ELSE	LOG ()
<u>COMMANDS :</u>	INPUT	SIN ()
	LET	COS ()
REGREAD	ON-GOTO	TAN ()
REGWRITE	ON-GOSUB	ATN ()
ERRNO		=, >, >=, <, <=, <>
		ASC ()
	ONERR	CHR ()
	ONEX1	CBY ()
	ONTIME	DBY ()
	PRINT	XBY ()
	PRINT#	GET
	PRINT@	IE
	PH0.	IP
	PH0.#	PORT1
	PH0.@	PCON
	PH1.	RCAP2
	PH1.#	T2CON
	PH1.@	TCON
	PGM	TMOD
	PUSH	TIME
	POP	TIMER0
	PWM	TIMER1
	RBANK	TIMER2
	REM	XTAL
	RETI	
	RND	
	RROM	MTOP
	STOP	LEN
	STRING	FREE
	ST@, LD@	PI
	UI1, UI0	
	UO1, UO0	

A.6 FLOATING POINT FORMAT

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

A.7 VARIABLE and STRING MEMORY ALLOCATION

This section is intended to answer the question: where does 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

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 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 implicitly or explicitly 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 $6 * 11=66$.

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:

CONTROL DESIGN, INC
Basic52 Manual with C4x Extensions

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 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, BASIC-52 checks both the DIMUSE and VARUSE pointers to make sure that VARUSE>DIMUSE. If the relationship is not true, a MEMORY ALLOCATION ERROR is generated.

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.

A.8 FORMAT OF A BASIC-52 PROGRAM

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

LINE FORMAT

Each line of 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.

EPROM (FLASH) 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 BASIC-52 to determine if a valid program is present. If the user types ROM 6, 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, 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 BASIC-52. These locations contain initialization information when the PROGX options are used. Version 1.0 of 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 PROG1 WAS USED, CONTAINED A 32H IF PROG2 WAS USED

8001H BAUD RATE (RCAP2H)

8002H BAUD RATE (RCAP2L)

Version 1.1 of 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 PROG3, 4, 5, 6 options.

IMPORTANT NOTE:

The PROG X options simply store ASCII character following the PROG command in location 8000H. That is why PROG1 stores a 31H in location 8000H, PROG2 a 32H, PROG3 (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.

A.9 ANSWERS TO A FEW QUESTIONS

QUESTION

Why can't BASIC-52 access the 8052's SFR 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 BASIC-52's console routines to crash.

QUESTION

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

ANSWER

BASIC-52 is actually capable of accepting characters at 38,400 baud. The problem is that after 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, 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 BASIC-52 is tokenizing the line, the characters will be lost. What the user must do in the download routine is wait until BASIC-52 responds with the prompt character (>) after a carriage return is sent to the BASIC-52 device. The prompt (>) informs the user that BASIC-52 is ready to receive characters from the console device.

QUESTION

I am writing in assembly language and I notice that the 8052 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          ; SWAPA<>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!

QUESTION

After RESET or Power-Up, 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.

APPENDIX B

B.1 C4x BASIC-52 EXTENSIONS

The Control Design Basic Interpreter provides Basic 52 programmability for the C46 Controller. It is exact in syntax and function to the Intel Basic-52 Interpreter. Control Design has added Extensions that support access to peripherals such as the LCD, Keypad, Real-Time Clock, Digital and Analog I/O Ports. The Control Design Basic Interpreter also provides Modbus Integer and Floating Point Registers support as well as a seamless interface between Basic and Modbus.

Unsupported Basic-52 Features

The following Basic-52 features are not supported in the Control Design version of the Interpreter:

BAUD, LIST#, NULL, PGM, PH0.#, Any FPROG, PH1.#, PRINT#, PROGn, ROM, UO0, UO1, XTAL

While the use of these commands may not cause the interpreter to issue an error message, using these commands may cause software or hardware malfunction. We recommend against using these features in this version of the Basic interpreter:

IE, IP, PCON, PORT1, PWM, RCAP2, T2CON, TCON, TMOD

Quick Start Information

To use the C4X module and get up and running quickly, you will need the following items:

1. C4x Intel Basic-52 manual with C4x Extensions
2. C46 Technical manual v6.xx (Modbus Register Map)
3. RS-232 Straight thru serial cable with Female D9 on PC end and Male D9 on C4X end.
4. CDI Software toolbox
5. Terminal program (such as Tera-Term Pro)
6. Word processor (such as Notepad, Wordpad or any editor that will save Text files)

Step1- load the CDI Toolbox and Terminal programs onto your PC.

Step 2- start the Terminal program and set it for 19200,n,8,1 for use with Basic on the C4X Com0 port.

Step 3- start the CDI Toolbox and set it for 9600,n,8,1 for use with the Modem on the C4X Com2 port.

NOTE: the CDI toolbox may conflict with Terminal program if they are both on the same PC com port.

Step 4- connect to the PC and Com0 on the C4X with the serial cable (see #3 above).

Step 5- apply power, 12 to 24vdc, to the C4X. <100ma will supply the C4X without radio transmit.

Step 6- the Basic sign-on should appear on the Terminal program screen and the PWR LED should be flashing.

If you experience problems or have questions, call toll free > [888-422-1442](tel:888-422-1442) for support.

B.2 Application Program Storage

ERASE <slot num>

Programs may be removed from Flash memory by issuing the **ERASE** command. The syntax of this command is **ERASE** <slot num> where <slot num> is a default number in the range of 1 to 8.

There are no error conditions or return values from this command.

NOTE: If the Basic Application storage slot size has been changed, the <slot num> value will vary accordingly.

NEW

The new command clears Basics operating memory in RAM. Use this before loading your code to ensure that only the code you want is in memory after loading. The Basic interpreter will sort line numbers as they are entered, so any inleft overlt and not replaced by your new code will remain and possibly be executed.

PROG

Programs can be stored using the **PROG** command. Once a program has been loaded and is operating correctly, stop it and issue the **PROG** command to store it to the first available program slot in Flash. **PROG** returns the slot number where the program was stored. The default settings for the Flash program memory is eight (8) slots of 4096 (4K) bytes each. This setting can be changed by writing to Modbus Register 4x4810.

NOTE: PROG does not allow the user to select the slot in which the program will be stored. Basic simply searches for the next available empty slot and puts the program in it. It then prints to the console the slot number it used.

To force programs to store in a specific slot, the user must put a program, even a one line program will do, in every available slot, thereby filling all slots. Then issue the ERASE <slot num> command for the specific slot desired. When the PROG command is issued, Basic will be forced to use the only available slot.

ROM <slot num> / XFER

To load a program that is stored in Flash issue the **ROM** command with a slot number as a parameter. This will set an internal pointer to the correct position in Flash for the **XFER** command to copy the code to RAM.

RROM <slot num>

This command will do the equivalent of the **NEW** command, the **ROM** command, the **XFER** command, and the **RUN** command all at once. Use this command inside your Basic programs to chain to another program. Example to call another program from one running in Slot 1 (or any slot):

```
1000 RROM 2
```

The above line will clear Basics operating ram of the current program, reset all variables to 0, point to the program in Slot 2, transfer it into operating Ram and Run it from the first line.

To save and transfer variables to the called program, simply write them to Modbus registers or store them to scratch pad memory using XBY and ST@, LD@ commands.

B.3 Keypad

The Keypad supports 2 Rows by 5 Keys per Row.

The Basic Application may read from the Keypad as needed. The 10 keys are scanned in the background and the last key pressed is stored for the application to retrieve. To access the Keypad the application will use the UI1, GET and UI0 commands.

For example:

```
110 UI1 REM switch to the keypad
120 K=GET REM check for a keypress
130 UI0 REM switch back to the basic console (com0)
```

If a key is available, its value will be in the variable K. If no key is available then the variable K will be set to zero.

To wait for a key press, do the following:

Keypad Example:

```
100 DO          REM set up a loop
110 UI1        REM switch to the Keypad
120 K=GET      REM check for a keypress
130 UI0        REM switch back to the basic console (Com0)
140 WHILE (K=0) REM loop until a key is pressed
```

The UI1 command switches the normal Basic serial port console (Com0) to the Keypad. The UI0 command returns control back the normal Basic serial port console.

NOTE: Only the Basic Application should switch the console to the keypad to retrieve a key. If the application switches the console to the keypad indefinitely, there will be no way to gain control over the application via the normal Basic console.

DO NOT type UI1 from command mode. This will cause loss of control of Basic via Com0. Always run the command sequence above from the Basic Application.

Below are the values returned for each key on the keypad:

Key	ASCII	DECIMAL
F1	48	0
F2	49	1
F3	50	2
F4	51	3
F5	52	4
F6	53	5
F7	54	6
F8	55	7
(up arrow)	56	8
(down arrow)	57	9

Please note that the UI1 command does not exclusively read the keypad. If a character is available from the normal console it will be returned if no character is waiting at the keypad.

B.4 Liquid Crystal Display (LCD)

The LCD supports 4 Rows by 20 Characters per row.

The LCD is controlled with the Basic PRINT@ command. This command functions similar to the PRINT command except characters are sent to the C46 LCD instead of the Basic console.

The Basic commands PH0.@ and PH1.@ are also supported for displaying hexadecimal values.

If the data to be displayed exceeds the 20 Character line length the data will continue to be displayed on the next line. If the 4 th line is exceeded, the data will wrap back to the 1 st line, overwriting what was previously being displayed on that line.

LC Display Backlight control

For units that have backlit LCD modules, backlighting can be controlled by printing the value CHR(1) to turn the backlight ON and CHR(0) to turn the backlight OFF.

```
PRINT@ CHR(1), : REM turn backlight on
```

```
PRINT@ CHR(0), : REM turn backlight off
```

LCD Print Formatting

(See also: Basic-52 Programming manual for the PRINT Statement)

The **CR**, modifier forces a carriage return without a line feed. The CR, modifier can be used to repeatedly update a single line on the display.

The **SPC**({expr}) function is used with PRINT@ to move the cursor {expr} columns.

The **TAB**({expr}) function specifies the absolute column number where the next PRINT@ value begins.

The **USING**({format expr}) function allows the output to be formatted. This function is very useful for displaying floating-point values.

An enhanced display function allows the entire display to be cleared. To clear the display:

```
PRINT@ CHR(12),.
```

B.5 MODBUS REGISTERS

REGREAD [regnumber]- The Modbus Register Read Command

The REGREAD command is used to return the value (Integer or Floating Point) of a MODBUS register. The application must provide the Register Number as a parameter to the call. After executing the REGREAD command the application must POP the result from the BASIC-52 Stack.

For example to read the Integer value from Register 4001:

```
100 REGREAD 4001    REM read modbus register 4001
110 POP A           REM get the value
120 PRINT@ A        REM print value to lcd
130 PRINT A         REM print value to basic console (com0)
```

After executing REGREAD, the variable A will have the integer value (0-65535) that was stored in MODBUS register 4001.

If an error occurs during the REGREAD command, the ERRNO command will return an error code. An ERRNO code of 0 means no error occurred. An ERRNO code of 1 means that an illegal MODBUS register was specified. (See ERRNO in next section.)

REGWRITE [regnumber],[value]- The Modbus Register Write Command

The REGWRITE command is used to set the value of a MODBUS register. The application must provide the Register Number and the Value to write as parameters to the command. There is no value to POP after REGWRITE.

For example, to write the floating point value 10.50 to register 7001:

```
100 REGWRITE 7001,10.50
```

If an error occurs during the REGWRITE command, the ERRNO command will return an error code. An ERRNO code of 0 means no error occurred. An ERRNO code of 1 means that an illegal MODBUS register was specified. (See ERRNO in next section.)

B.6 MODBUS ERRORS

ERRNO- The Modbus Error Number Comand

The ERRNO command is provided so that a well-written application may check for unexpected errors when using the C4x extensions. After executing ERRNO the application must perform a POP that will return an integer value with the error status of the last requested command.

The following commands set the ERRNO value:

REGREAD
REGWRITE

Use of ERRNO is recommended but not required. An ERRNO code of 0 always means no error occurred.

EXAMPLE: ERRNO will be 0 in the first REGREAD and 2 in the second REGREAD.

```
REM first regread
110 REGREAD 4001 REM register 4001 is a standard holding register
120 POP A
130 ERRNO
140 POP E
150 PRINT 'Error Number:',E
```

```
REM second regread
160 REGREAD 3000 REM register 3000 does not exist
170 POP A
180 ERRNO
190 POP E
200 PRINT 'Error Number:',E
```

NOTE: If ERRNO is non-zero then the values POPed for REGREAD are undefined.

Error Codes returned by ERRNO:

0 = No Error
1 = Illegal Function Code
2 = Illegal Data Address
3 = Illegal Data Value.