3.48 - STOP

The STOP statement stops the program from executing and places the interpreter in the command mode. If the RUN MODE jumper is ON then the program will break and re-run as in the ERROR statement.

3.49 - STRING [expression],[expression]

The STRING statement is used to allocate memory for strings. The first expression is used to set the total number of bytes used and the second expression sets the number of bytes per string. The number of bytes needed is equal to: (string length+1)*(# strings)+1 Each string needs 1 additional byte for the terminating (CR) plus 1 additional byte overall.

To store 10 strings at 20 bytes each you are required to allocate: ((20+1)*10)+1=211
`The statement would read: STRING 211,20`

It is recommended that the programmer allocate STRING memory at the beginning of the program. Use STRING 0,0 to de-allocate memory if the program was changed.

3.50 - TIME, TIME%, TIME#

The TIME statement causes the time value to be sent to any serial port followed by a CRLF (carriage return line feed). If the CRLF is not desired then a semicolon (;) will suppress the CRLF. The time is sent in the format HOUR:MINUTE:SECOND and if the programmer wants these values to be assigned to variables then the clock memory locations can be read and the variable assignment made.

EXAMPLE:

```
10 TIME
40 TIME#;
50 PRINT"The time is "; : TIME; : PRINT" HR:MIN:SEC"
60 H1=XBY(0A92H) : M1=XBY(0A91H) : S1=XBY(0A90H)
70 PRINT H1,":",M1,":",S1
80 TS=S1+(M1*60)+(H1*3600)
90 PRINT"The total seconds = ",S1
```

3.51 - WARMBOOT [line number]

The statement WARMBOOT performs a test on four volatile internal memory locations to determine if the controller is recovering from an initial power up state or a reset state invoked by either the command RESET, watchdog timer, or a manual reset (push button). By using 4 memory locations there is a 2E32 (4294967296) chance that these locations would have the same values as the predetermined ones. This would be considered a very small probability.The WARMBOOT [line number] works just like the GOTO but also contains the warm or cold boot test. That is, if the warm boot condition is true (warm boot has occurred) then the program jumps execution to the line number in the warm boot statement. If a cold boot condition occurred then the program execution would continue at the next line following the WARMBOOT statement.This allows the programmer the ability to jump around an initialization section of the program.If the programmer wants to test the function of his/her initialization routines then the following can be performed to fake a cold boot. The warm or cold boot is determined by the contents of memory location 02FFH. If a 055H is in that location then a warm boot will occur and if the contents are 0AAH then a cold boot will occur. Because the test is performed in the controllers reset routines, the only time that location will change is after a reset. Therefore by changing the contents of memory location 02FFH before the WARMBOOT executes the programmer can alter the BOOT state.

EXAMPLES:

```
10  WARMBOOT 100
20  OTU 000: OTU 100: CLEAR I: CLEAR B
30  REM  more initialization code here
100 REM  here if a warm boot
```

```
Change the boot state for program testing
1   XBY(2FFH)=0AAH : REM make it always cold boot
10  WARMBOOT 100
20  OTU 000: OTU 100: CLEAR I: CLEAR B
30  REM  more initialization code here
100 REM  here if a warm boot
```

3.52 - XCT[integer],[(@)integer]

The XCT statement is useful in working with counters. It does an on the fly examination of the counters accumulated value and if it matches the value in the XCT statement the XCT reads true and only true for that one value. The counter number is the first integer and the second is the examined value.

EXAMPLES:

```
10 CTU 000,201 : CTR 000
20 XCT 000,050 : GOSUB somewhere at 50th loop
30 XCT 000,100 : GOSUB at 100th loop
40 XCT 000,150 : XIH 016 : PRINT"150th and 16 ON"
50 XCT 000,200 : CTU 001,100:CTR 001: PRINT"20,000thloop!"
999 GOTO 10
```

3.53 - XDAY [(@)integer]

The XDAY statement examines the day of the week (1-7) in the real time clock and if it matches the integer then the statement reads true as other control statements. The XDAY can be used to select different subroutines for real time controlling or when used with one shots execute an event one every day. The integer value is a single digit when used directly or 3 digits when indexed.
Sunday is day 1 and saturday is day 7.

EXAMPLE:

```
10 XDAY 7 : ROS 000 : OST 006 : PRINT"SATURDAY"
20 XDAY 1 : ROS 001 : OST 000 : PRINT"SUNDAY"
30 XDAY 2 : ROS 002 : OST 001 : PRINT"MONDAY"
40 XDAY 3 : ROS 003 : OST 002 : PRINT"TUESDAY"
50 XDAY 4 : ROS 004 : OST 003 : PRINT"WEDNESDAY"
60 XDAY 5 : ROS 005 : OST 004 : PRINT"THURSDAY"
70 XDAY 6 : ROS 006 : OST 005 : PRINT"FRIDAY"
80 XDAY 7 : GOSUB 1000 : REM special routine
90 XDAY 1 : GOSUB 2000 : REM special routine
```

3.54 -XHMS [(@)integer],[(@)integer],[(@)integer]

The XHMS statement combines the examination of hours, minutes and seconds of the real time clock in one statement. If all match the specified integers then the statement reads true. The XHMS can be used to control events relative to real time with 1 second resolution. The integer values are all two digit and the ranges apply as in a 24 hour clock. A very important feature of the XHMS is that it behaves as a one shot for the time it executes. That is, after it is true once it will not execute any more even if the program loop passes it many times during the same time. Therefore if many other statements require execution, then a gosub to a separate routine should be used.

EXAMPLE:

```
10 XHMS 23,00,00 : PRINT "11 HRS 0 MIN 0 SEC"
20 XHMS 23,00,00 : PRINT "TRY IT AGAIN ?"
30 REM 20 will not execute because of the one shot effect
40 XHMS 10,00,00 : GOSUB 2000
50 XHMS 22,00,30 : PLW A,16,02
60 XHMS 12,55,00 : IF ADC3<T2 THEN OTL 002
```

3.55 - XHR [(@)integer]

The XHR statement examines the hour of the day (00-23) in the real time clock and if it matches the integer then the statement reads true as other control statements. The XHR can be used to select different subroutines for real time controlling or when used with one shots execute an event hourly. The integer value is two digits when used directly or 3 digits when indexed.
The real time clock is in 24 hour format.

EXAMPLE:

```
10 XHR 23 : GOSUB 1000
20 XHR 22 : ROS 000 : REM reset oneshot on the hour before
30 XHR 23 : OST 000 : ?"Here at 11 O'clock only once!"
```

3.56 - XIH [integer]

The XIH statement examines a specified bit in the image table and if that bit is ON or high the program execution continues past the XIH. If the bit is OFF or low then the program execution passes to the next line number after the XIH line. This method of bit testing yields very high speed operation. Multiple XIH or XIL statements can perform efficient AND logic testing.

EXAMPLE:

```
10 XIH 016 : PRINT"Bit 16 is ON"
20 XIH 017 : XIH 018 : XIH 019 : XIL 020 : OTL 005
30 REM  if 17 18 19 is ON and 20 is OFF then 5 ON
```

Remember that program execution continues down the line as long as conditions tested are true. If insufficient line space does not allow all the required conditions to fit on 1 line then the JMP and LBL statements are required to effectively lengthen the line length.

3.57 - XIL [integer]

The **XIL** statement examines a specified bit in the image table and if that bit is OFF or low the program execution continues past the **XIL**. If the bit is **ON** or high then the program execution passes to the next line number after the **XIL** line.
Multiple **XIH** or **XIL** statements can perform efficient **AND** logic testing as seen in the **XIH** statement description.

EXAMPLE:

```
10 XIL 020 : XIL 021 : PRINT" 20 and 21 are OFF"
20 XIL 025 : GOSUB 1000
30 XIL 032 : XIH 100 : GOTO 100
40 XIL 001 : IF ADC7>800 THEN OTL 001
50 REM if 1 is OFF and if ADC7>800 set 1 back ON
60 XIH 009 : OTU 009 : REM if 9 goes ON turn it OFF
```

The **XIH** or **XIL** statements can be used to skip around program blocks. If part of an application is dependent on qualifying input conditions then that part can be skipped until those conditions are met. This is useful in very large programs to keep the execution speed up.

3.58 - XMN [(@)integer]

The XHR statement examines the minute of the hour (00-59) in the real time clock and if it matches the integer then the statement reads true as other control statements. The **XMN** can be used to select different subroutines for real time controlling or when used with one shots execute an event every minute. The integer value is two digits when used directly or 3 digits when indexed.

3.59 - XSEC [(@)integer]

The XHR statement examines the seconds (00-59) in the real time clock and if it matches the integer then the statement reads true as other control statements. The XSEC can be used to select different subroutines for real time controlling or when used with one shots execute an event at various seconds. The integer value is two digits when used directly or 3 digits when indexed.

EXAMPLE:

```
10 XSEC 29 : ROS 000
20 XSEC 30 : OST 000 : PRINT "HERE ONCE EVERY 30 SECONDS"
30 XMN 59 : GOTO 2000 : REM do this for 1 min @ every 59 mins
```

3.60 - XSB(E) "character string" or string variable

XSB is very powerful statement for testing the contents of the serial port receive buffers. The XSB statement will be true if the string or string variable (example XSB "NO CARRIER" or XSB $(0)) matches the string in the buffer. The string in the XSB statement does not have to be complete. For example the statement XSB "NO CAR" will read true just as the more complete statement XSB "NO CARRIER". The XSB only matches the characters up to the last quote (") for a non-exact match or an exact match when using the 'E' switch. Only keyboard characters can be tested between the quotes. If the user must test for control characters then the actual memory locations of the buffer would have to be tested. The buffer is located in external memory addressed 0B00H to 0BFFFH. The buffer is a ring buffer, therefore it wraps around if the received characters exceed 255. The CIC 1 statement zeros the buffer after execution. Therefore it should only be executed after all tests on the buffer contents are preformed.

EXAMPLE:

```
10   COMINT %,1000: CIC %,1
.
.
1000 REM * Aux serial interrupt subroutine *
1010 CIC %,0: REM turn off serial interrupts
1020 XSB% "ON1": OTL 001: REM ON after user types ON1 +CR
1030 XSB% "OFF1": OTU 001: REM turn it off
1035 XSBE%"STOP": STOP: REM Stop program
1040 CIC %,1: REM interrupts back on
1050 RETI
```

NOTE: REM statements are included here to describe the program action. REM statements should be omitted from the users program to keep execution times as fast as possible!

Arithmetic Operators and Expressions

ARITHMETIC and LOGICAL OPERATORS AND EXPRESSIONS
DUAL OPERAND OPERATORS (Operations on two expressions)

4.1 - (+) ADDITION
EXAMPLE:
```
10 A=XBY(800H)+10
```

4.2 - (-) SUBTRACTION
EXAMPLE:
```
10 A=ADC7-100
```

4.3 - (*) MULTIPLICATION
EXAMPLE:
```
10 A=B*C
```

4.4 - (/) DIVISION
EXAMPLE:
```
10 A=XBY(800H)/25
```

4.5 - (**) EXPONENTIATION (maximum power = 255)
EXAMPLE:
```
10 A=10**3
```

4.6 - (.AND.) LOGIC (expressions <= 65535)
EXAMPLE:
```
10A=PORT0.AND.PORT1
```

4.7 - (.OR.) LOGIC (expressions <= 65535)
EXAMPLE:
```
10A=1.OR.4
```

4.8 - (.XOR.) LOGIC (expressions <= 65535)
EXAMPLE:

```
10A=15.XOR.XBY(8FFH)
```

ARITHMETIC and LOGICAL OPERATORS AND EXPRESSIONS
UNARY OPERATORS

4.16 - EXP([expression])
The EXP operator raises 'e' (2.7182) to the power of the expression.
EXAMPLE:
```
10 A=EXP(3)
```

4.17 - SIN([expression])
The SIN operator returns the SINE of the expression expressed in radians.
EXAMPLE:
```
10 A=SIN(PI/4)
```

4.18 - COS([expression])
The COS operator returns the COSINE of the expression expressed in radians.
EXAMPLE:
```
10 A=COS(PI/4)
```

4.19 - TAN([expression])
The TAN operator returns the TANGENT of the expression expressed in radians.
EXAMPLE:
```
10 A=TAN(PI/4)
```

4.20 - ATN([expression])
The ATN operator returns the ARCTANGENT of the expression expressed in radians.
EXAMPLE:
```
10 A=ATN(PI)
```

4.21 - PI
PI is not a operator, but a stored constant. (PI=3.1415926)

4.22 - RND
RND returns a pseudo-random number in the range of 0 to 1 from a 16 bit binary seed.
EXAMPLE:
```
        10 PRINT
RND
```

Character String Operators

ASC

CHR

5.2 - CHR([expression])

The CHR operator returns an ASCII character equal to the value (0-255) in the expression. he CHR also functions like a BASIC MID$ function. That is it can be used to pick out characters from a string. The syntax for character picking is the same as the ASC uses for pointing to a character.

EXAMPLE:

```
10 PRINT CHR(122) : REM print a 'z'
20 $(1)="BAC552"
30 FOR C=4 TO 6
40 PRINT CHR($(1),C);
50 NEXT C : PRINT
>RUN

552
```

to print a string PRINT $(1)

Memory Access Operators

XBD -read or write a 16 bit Integer from external memory

DBY-read or write a 8 bit Integer from internal data memory

XBY-read or write a 8 bit Integer from external memory

6.1 - CBY(expression)

The CBY operator returns the value of the 552's external code memory locations. Since the external code memory is ROM the CBY is a read only operator. No value can be assigned to CBY.

EXAMPLE:

```
 10 A=CBY(5000H)
```

6.2 - DBY(expression)

The DBY operator accesses the 552's internal RAM memory locations. Since the internal RAM is only 256 bytes the value in the expression can only range from 0-255. The internal RAM holds all system control values and the 552's stack so altering locations will crash the BASIC. See the internal memory map for useful memory locations.

EXAMPLE:

```
10 PRINT DBY(04CH) :REM print timer low byte
20 PRINT DBY(04BH) :REM print timer high byte
```

6.3 - XBY(expression)

The XBY operator accesses the 552's external RAM memory locations. The 552's external RAM is 64K bytes so the expression can only range from 0-65535. The external RAM has a reserved area for control stacks, argument stacks, control statements values and bit image tables. Altering some of these locations will also crash the BASIC. Locations are listed in the memory mapping table.

EXAMPLE:

```
10 PRINT XBY(0A90H) : REM read clock seconds
20 XBY(1800H)=156 : REM save 156 at location 900H
```

Special Function Operators

ADC(0-7)- read analog input values

DAC(0-1)-write analog ouput values

PORT(0-4)-read or write 8 bit words to or from ports

INKEY

PWROFF

MTOP

LEN

FREE

7.1 - ADC0 ADC1 ADC2 ADC3 ADC4 ADC5 ADC6 ADC7

The ADC operators contain no expression. ADC1 is an separate operator just like ADC7, the numerical value represents the analog input channel the value is returned from. These operators actually function like a variable in BASIC.
NOTE: The analog data is read at the time of the ADC execution on the BAC552ES controllers. The older BAC552(X) controllers had the analog values read in the background at a fixed rate.

The results of any analog to digital conversion can be read in three internal data memory locations with the DBY(). This is very important for data logging applications because the result can be saved in 1 SRAM location for 8 bit resolution and in 2 bytes for 10 bit resolution. Immediately after the execution of ADC(0-7), DBY(04FH) holds the 8 bit result, DBY(050H) holds the lower 8 bits of the 10 bit result and DBY(051H) holds the remaining upper 2 bits of the 10 bit result. Remember that the 8 bit result is actually bit 9 to 2 of the ten bit result with bits 1 and 0 dropped. DBY(04FH) will not be the same as DBY(050H).
NOTE: The ADC operator must be assigned to a variable or a SYNTAX error will occur! Therefore the programmer must let some dummy variable equal ADC even if they are only interested in recover data using the DBY() operators! See example below.

EXAMPLE:

```
 10 PRINT ADC0
20 T=INT(ADC0*.00488)                                             30
IF ADC0<512 THEN OTL 001
 .
 REM Must assign a dummy variable to Abcs
 10 DV=ADC7: PRINT"ADC7 = ",DBY(04FH), DBY(050H), DBY(051H)
```

7.2 - DAC0 DAC1

The DAC operators are read or write. These are just like the ADC in that the numerical value (0 or 1) refers to a separate operator. The 552 produces a pulse width modulated output which is equal to the 8 bit value assigned to the operator. If the operator is read then the last value assigned is returned. The PWM signal is RC filtered to produce a output voltage proportional the 8 bit value. DAC0=128 will produce a half scale output of approximately 2.5 volts on channel 0.

EXAMPLE:

```
 10 DAC0=0
20 FOR X=0 TO 255                                           30
DAC0=X                                                   40
NEXT X
 10 PRINT DAC1 : REM print last value of DAC1
```

7.3 - PORT0 PORT1

The operators PORT0 and PORT1 return a byte wide value (8 bits) from the input ports. PORT0 is the first 8 bits (016-023) and PORT1 is the second (024-031) group. These operators are read only. They provide an easy way to read parallel data devices connected to the input ports.

EXAMPLE:

```
10 PRINT PORT0
20 A=PORT0.AND.00FH : REM read lower 4 bits from port0
30 IF PORT1<>0 THEN 1000 : REM any inputs on ?
```

7.4 - INKEY

The INKEY operator holds the value of the last character before the CR which was generated the console communication interrupt. INKEY will return a full 8 bit value but that value is cleared to 0 once INKEY is executed once. Therefore if multiple tests of INKEY are to be performed then INKEY should be assigned to another variable.

EXAMPLE:

```
1000 REM communication interrupt
1010 A=INKEY
1020 IF A=3 THEN STOP
1030 IF A=130 THEN OTL 000
1040 IF A=131 THEN OTU 000
1050 A=0 : RETI
```

7.5 - PWROFF

The PWROFF is a special operator which returns a floating point value equal to the number of seconds from the last time the controller was running to the next time it was powered up.

This value is valid for a maximin of a 6 day period of no operation. If the controller was stopped on friday at 13:00 and restarted 7 days later at 13:20 then PWROFF would equal 20 seconds. If the restart was the same day but at 12:40 then PWROFF would equal -20 seconds. As long as day OFF <> day ON then the time will be correct for any period of 6 days or less. The PWROFF was not intended for long period calculation, rather for short interruption periods. For very long periods BASIC could be used to modify the PWROFF values by reading the day of the month rather than day of the week as PWROFF does.

EXAMPLE:

```
10 REM program start up
20 IF PWROFF>200 THEN GOSUB 1000
30 IF SGN(PWROFF) = -1 THEN GOSUB 2000
40 IF PWROFF>XBY(1800H) THEN GOSUB 3000
50 REM 900H holds a interrupt loaded time value
```

7.6 - MTOP

This operator is used to retrieve the top of memory value. After a reset condition the value of MTOP is 32768 with the MSIZE jumper OFF. With the MSIZE jumper ON the MSIZE is 65535 bytes. If the BAC552ES is used for data acquisition applications or the application requires more memory space for special storage, use the smaller memory size (jumper OFF).

EXAMPLE:

```
>PRINT MTOP
65535
```

7.7 - LEN

The **LEN** operator returns the length of the user program in RAM memory. If no program is in memory then LEN would equal 1.

EXAMPLE:

```
>PRINT LEN                                                  235

 *after NEW*
>PRINT LEN
 1
```

7.8 - FREE

The FREE operator returns the number of bytes free to the user program. After a NEW, FREE would equal MTOP-(actual prgm start). With a program in memory FREE would equal the number of bytes left.

EXAMPLE:

```
>PRINT FREE

28978
```