3.21 - IIC (cont'd) - IMAGE TABLE MAPPING

The following is a map of IIC hardware states to image table bits. The user program must test the appropriate state bits to control IIC block data transfers. It is up to the user to clear all bits! The IIC state only sets bits ON.

Bus error	state = 00	bit = 502
MT/MR Master start (receive or transmit)	state = 08	bit = 488
MT/MR Master repeated start	state = 10	bit = 489
MT Slave address + Write transmitted ACK received	state = 18	bit = 490
MT Slave address + Write transmitted NO ACK received	state = 20	bit = 491
MT Data transmitted and ACK received	state = 28	bit = 492
MT Data transmitted and NO ACK received	state = 30	bit = 493
MT/MR Arbitration lost in R/W or data	state = 38	bit = 494
MR Slave address + Read transmitted ACK received	state = 40	bit = 495
MR Slave address + Read transmitted NO ACK received	state = 48	bit = 496
MR Data received and ACK transmitted	state = 50	bit = 497
MR Data received and NO ACK transmitted	state = 58	bit = 498
SR Own address + Write received ACK transmitted	state = 60	bit = 472
SR Arbitration lost in Slave address	state = 68	bit = 473
SR General call (not applicable)	state = 70	bit = 474
SR Arbitration lost in general call	state = 78	bit = 475
SR Data received ACK transmitted	state = 80	bit = 476
SR Data receive NOACK transmitted	state = 88	bit = 477
SR General call (not applicable)	state = 90	bit = 478
SR General call (not applicable)	state = 98	bit = 479
SR Stop or Start while still addressed	state = A0	bit = 480
ST Slave address + Read received and ACK returned	state = A8	bit = 481
ST Arbitration lost in slave address	state = B0	bit = 482
ST Data transmitted and ACK received	state = B8	bit = 483
ST Data transmitted and NOACK received	state = C0	bit = 484
ST Last data byte transmitted and ACK received	state = C8	bit = 485

3.22 - INPUT, INPUT% and INPUT#

The INPUT statement has been extended and changed on the BAC552ES controller. The INPUT statement can now only be executed in the COMINT subroutine. The INPUT statement allows data input to either strings or variables in the program. The INPUT requires a CR as a terminator and a comma as a separator. The interrupt character must be a CR when the INPUT statement is used. One single INPUT can handle both strings and variables as long as they are properly separated by commas. Should the INPUT find illegal data such as a string which exceeds the defined length the BASIC will just ignore the extra. If INPUT finds no data at all then it will REM or waste the whole line. This prevents crashing the program and transferring control to the ERROR handler.

EXAMPLE

```
10 COMINT %,1000: CIC %,1: OTU 300
 20 STRING (100,10)
 30 REM loop
 40 XIH 300: PRINT "just received a CR, no data": OTU 300
 .
 100 GOTO 30
 .
 1000 CIC %,0: OTL 300
 1010 INPUT% $(0): OTU 300
 1020 PRINT% "The received string was ",$(0)
 1030 CIC %,1
 1040 RETI
```

3.23 - JMP [(@)integer]

The statement JMP is used to extend a BASIC line. The JMP statement integer can range from 0 to 255. Once a specified JMP is executed, only a LBL statement with the same number will be true for 1 execution. The JMP is very useful in extending the number of control statements in one single line, improving the ability to do OR condition testing and very complex condition testing.

EXAMPLE:

```
 10 XIH 000 : XIH 001 : XIH 003 : XIL 200 : JMP 002
 20 LBL 002 : XIL 015 : PRINT"All conditions met!"
 REM use JMPs 005 to perform
OR
 30 XIH 000 : XIL 009 : XIL 200 : JMP 005
 40 XIL 001 : XIL 008 : XIH 201 : JMP 005
 50 LBL 005 : XIH 012 : ?"line 30 OR 40 AND 12 ON !!"
```

3.24 - LBL [(@)integer]

The JMP sets a special control bit and the LBL does the testing. It is important to note that after the LBL performs the test on the specific bit and that bit is ON or high the LBL clears or turns that bit OFF. This is a very important feature in that if the conditions that where true to set the JMP go false then execution after LBL would only occur that initial time! The special bit remains unaffected by the LBL if it is low or OFF. Multiple lines can set the JMP as seen in the above example but only one LBL should exist. The JMP and LBL can be both indexed by a memory location using the @ sign. Indexing the LBL provides a means of clearing multiple JMPs and indexing the JMP offers a way to one shot multiple events.

EXAMPLE:

```
10  FOR B=0 TO 4 : XBY(800H)=B : LBL @000
20  NEXT B : REM make sure no JMP bits are set ON
30  FOR C=0 TO 2
40  XBY(809H)=C
50  JMP @009 : REM sets the JMP bit ON
60  LBL 000 : PRINT"Here when JMP 0 is set"
70  LBL 001 : OTL 009
80  LBL 002 : OTU 008 : OTU 010
90  NEXT C
```

3.25 - LD@ [expression]

The LD@ statement loads the top of the stack with the floating point number pointed by the expression from memory. The LD@ is also redirected by the HIMEM statement. The @[expression] is different from the @[integer]!

EXAMPLE:

```
10 PUSH A : ST@ 1805H : REM save A at 1805H
20 LD@ 1805H : POP B : REM load from 1805H to B
```

3.26 - LIO or LIO!

The statement LIO reads the local (on main board) inputs and updates the bit image table. The (!) option does an immediate output update only, no input bits are updated. At the time of execution the bit image table is read and the current status of the output bits are transferred to the output relays and LEDs. The first 16 bits of the table are mapped to the outputs and the next 16 bits are mapped to the inputs. To turn ON the first relay 0 the first bit is set and the LIO is executed. To examine the first input LIO is executed and then bit 16 will be the result of the input status. This might at first seem very awkward but actually the LIO has to be only executed once! The BASIC control program is continually looping. Placing the LIO statement just before the GOTO, causes two things happen. First the outputs are written to based on what ever logic occurred throughout the program. Second the inputs are read and their status is now available for the program logic to determine what to do. The input status is held until the next LIO statement. The controller's connection to the real world is only actual at the time of the LIO execution. Any higher bits which are used for internal relays change upon execution of there statements. The BASIC control statements are very fast but a very intensive program can take some time to loop back, which affects the IO response time. By using counters to branch to subroutines which do not require high speed response the IO update time can be kept up. Also by jumping around program areas until they are required can save a lot of time. Any bit can be operated on by any bit manipulation statement where allowed. For example a XIH can be used examine a output bit and setting a bit ON which maps to the inputs can fake that input until the LIO updates the location. Although setting a input bit cannot invoke an interrupt if enabled, interrupts are read in the back ground and operate separately from the LIO. There are no restrictions on how many LIOs exist in a program. If the program breaks from the main loop at any time such as error trapping the LIO! might be required to turn some outputs OFF or if an interrupt requires immediate response then the LIO! would have to be included in the interrupt routine. The OTU and OTL statements also have an immediate(!) option, please see OTU and OTL for more information.

EXAMPLE:

```
5  REM simple program                                          10
XIH 016 : XIH 017 : XIL 017 : OTU 000                        20
XIH 020 : OTL 001                                          30 XIL
020 : OTU 001                                             40 LIO :
GOTO 10


5  REM Immediate output response to an interrupt
10 ONINT 0,1000 : DIC 0,1                                     20
XIH 030 : OTU 009                                           30
XIL 030 : OTL 009                                         99 LIO
: GOTO 20                                                    .
.
REM execute an immediate LIO as soon as 008 is cleared
1000 OTU 008 : LIO! : RETI : REM get fast response to OTU 008
```

```
10 REM use of counters to help execution time
20 REM *MAIN LOOP START*
30 CTU 000,201 : CTR 000
40 REM a very large floating point routine is broken up
50 REM into 4 smaller blocks executed every 50 main loops
60 XCT 000,050 : GOSUB 1st BLOCK
70 XCT 000,100 : GOSUB 2nd BLOCK
80 XCT 000,150 : GOSUB 3rd BLOCK
90 XCT 000,200 : GOSUB 4th BLOCK
100 XIL 020: GOTO 200 : REM skip it if 20 is OFF
.
200 REM jumped around some stuf
990 CTU 001,100 : FLFP 013 : CTR 001
995 REM lpi led will show how long our prg takes
996 REM for 100 loops. ON for 100 then OFF for 100 loops
999 LIO : GOTO 20
```

BAC552 CONTROLLER - OUTPUT AND INPUT BIT MAPPING

OUTPUT	BIT #	INPUT	BIT #	INTERRUPT #
relay 0	000	00	016	0
relay 1	001	01	017	1
relay 2	002	02	018	2
relay 3	003	03	019	3
relay 4	004	04	020	4
relay 5	005	05	021	5
relay 6	006	06	022	6
relay 7	007	07	023	7
relay 8	008	08	024	
relay 9	009	09	025	
ERR LED	010	10	026	
RUN LED	011	11	027	
LED 012	012	12	028	
LED 013	013	13	029	
LED 014	014	14	030	
LED 015	015	15	031	

3.27 - MOVE [source address],[destination address],[length]

The MOVE statement will copy a block of memory up to 255 bytes in length. The MOVE is much faster than using the typical FOR NEXT loop to perform the same task. The programmer can use the HIMEM switch to redirect the destination of the MOVE only!

The MOVE can be very useful for data logging applications. The clock data can be very efficiently moved to a data storage block of memory, either FLASH or SRAM.

EXAMPLE:

```
5       REM log clock every time 016 goes high
10     PT=7: PUSH PT: HIMEM 1: ST@ (5): HIMEM 0
30      FOR T=1 TO 50: NEXT T
40      XIH 016: OST 000: GOSUB 100
50      XIL 016: ROS 000
90      FLFP 012: LIO : GOTO 30
.
100     HIMEM 1
REM     Recover the PT pointer every time
110     LD@ (5): POP PT: IF PT>60000 THEN 130
REM     Clock is 7 bytes stating at 0A90H to 0A96H
120     MOVE 0A90H,PT,7:PT=(PT+7)+6: PUSH PT: ST@ (5)
125     PRINT PT;
130     HIMEM 0: RETURN
```

3.28 - ON [expression] GOTO [ln num],[ln num],...,[ln num]

The expression after the ON statement is used to look up where the program execution is to be transferred to. If the value of the expression is less than zero then a bad argument error would result and if the value is greater than the number of lines in the line list then a bad syntax error would result. The values of the expression must be valid integers and to eliminate the program from crashing during execution the programmer should always check the range of the expression first.

EXAMPLE:

```
10 A=INT(ADC0/50)
20 IF A>5 THEN A=5 : REM limit range of A
30 ON A GOTO 100,110,120,130,140,150
```

3.29 - ON [expression] GOSUB [ln num],[ln num],...,[ln num]

The ON - GOSUB functions similar to the ON - GOTO but subroutine calls are made rather than jumps. After returning from the subroutine the program execution is returned to the line after the ON - GOSUB.

EXAMPLE:

```
10 FOR A=0 TO 3
20 ON A GOSUB 100,110,120,130
30 NEXT A
.
100 PRINT "ZERO" : RETURN
110 PRINT "ONE" : RETURN
120 PRINT "TWO" : RETURN
130 PRINT "THREE" : RETURN
>RUN

ZERO
ONE
TWO
THREE
```

3.30 - ONERR [error number],[line number]

The ONERR statement provides the programmer with selective error trapping in the BASIC program. Once a particular error trap is enabled by the execution of an error statement, the interpreter will, upon encountering an error look up the line number associated with the error and goto that line. If the error trapping was not enabled for that type of error then the handler would break to the command prompt as long as the RUN MODE jumper is OFF. With the jumper ON (RUN MODE enabled) the handler would display the error and then restart the program. This operation is more than undesirable and it is recommended that for an embedded application with RUN MODE enabled that all error traps be set even if they all goto the same line.

INTERPRETER ERROR CODES
0 - BAD SYNTAX
1 - DIVIDE BY ZERO
2 - BAD ARGUMENT
3 - MEMORY ALLOCATION
4 - ARRAY SIZE
5 - ARITH. UNDERFLOW
6 - INVALID LINE NUMBER
7 - ARITH. OVERFLOW
8 - NO DATA
9 - CANNOT CONTINUE
10 - INTERNAL STACK
11 - EXTRA IGNORED
12 - ARGUMENT STACK
13 - CONTROL STACK
14 - PROGRAM IN SRAM MEMORY IS PROTECTED
15 - PROGRAM CHECKSUM FAILED
16 - EXPANSION I/O COMMUNICATION ERROR
17 - POWER LINE TRANSMISSION BUFFER FULL
18 - EEPROM COMMUNICATION ERROR
19 - RENUM FUNCTION ERRORS
20 - LINE FILE ERRORS
21 - FLASH MEMORY ERRORS

EXAMPLE:

```
10 ONERR 0,1000 : ONERR 1,1000 : ONERR 2,1000
20 ONERR 3,1000 : ONERR 4,1000 : ONERR 5,1000
.
1000 RESET
```

3.31 - ONINT [interrupt number],[line number]

The ONINT statement sets up the line number to GOSUB when a OFF to ON transition is detected at the specified input. The ONINT is a BASIC interrupter interrupt and not a processor hardware interrupt, therefore some latency can be experienced because the interpreter has to complete the current statement.This does not mean that the interrupt is missed, just a small delay may be experienced. The interrupt control switch (DIC statement) enables or disables the interrupt. The interrupt subroutine must be terminated with a RETI (return from interrupt). Eight interrupts are supported and are mapped to the discrete inputs as follows:

DISCRETE INTERRUPTS

ONINT 0 - 016 input	ONINT 4 - 020 input
ONINT 1 - 017 input	ONINT 5 - 021 input
ONINT 2 - 018 input	ONINT 6 - 022 input
ONINT 3 - 019 input	ONINT 7 - 023 input

DIC [interrupt #],[1/0]

The DIC statement provides a means of controlling the enabling and disabling of a particular interrupt. The ON/OFF switch uses a '0' to represent OFF and a '1' to represent ON. The DIC can be used at any time to control a interrupt.

RETI

The RETI statement clears the internal flags that generated the interrupt and returns program execution to where it was interrupted from.

EXAMPLES:

```
10 ONINT 0,1000
20 DIC 0,1 : REM set it on
.
.

1000 REM count 20 transitions from first input (016)
1010 CTU 100,020 : CTR 100 : OTL 005 : DIC 0,0
.
rem  after 20 transitions turn 005 ON and disable
rem  interrupt
.
1020 RETI
```

3.32 - ONTIME [time value],[line number]

The ONINT statement sets up the line number to GOSUB when the 1 second time base timer reaches the specified time value. The ONTIME is a BASIC interrupter interrupt and not a processor hardware interrupt and some latency can be experienced. This does not mean that the interrupt is missed, just a small delay may be experienced. The ONTIME interrupt has priority over the discrete interrupts which are just done after if required. The timer has a range of 1 to 65535 seconds and the interrupt is always disabled after the timer has timed out, therefore for continual operation the TIC has to be re- executed. When disabled, the time count is cleared to zero, if the programmer wants to read the time then two internal memory locations can be accessed to obtain the values. The addresses of these values are in the below example.

TIC [1/0]

The TIC statement provides a means of controlling the enabling and disabling of the timer interrupt. The ON/OFF switch uses a '0' to represent OFF and a '1' to represent ON. The TIC can be used at any time to control the interrupt.

RETI

The RETI statement clears the internal flags that generated the interrupt and returns program execution to where it was interrupted from. It is the same for all interrupt types.

EXAMPLE:

```
10 ONTIME 60,1000
20 TIC 1 : REM set it on
.
.
1000 REM here after 60 seconds
1010 PRINT" 60 SECOND INTERRUPT"
1020 TIC 1 : RETI


10 ONTIME 1000,1000
20 TIC 1
30 ONINT 0,900
40 DIC 0,1
50 REM on the first transition, read timer and print value
.
900 DIC 0,0 : TH=DBY(048H) : TL=DBY(049H) : TIC 0
910 T=(256*TH)+TL
920 PRINT"TIME FROM START TO TRANSITION WAS ",T
930 DIC 0,1 : TIC 1 : RETI
.
1000 PRINT"TIMER REACHED 1000 SECS AND NO TRANSITION"
1010 TIC 1 : RETI
```

3.33 - OST [(@)integer]

The OST statement reads true for only one time after it has been reset by the ROS statement. This provides a means of executing one or multiple statements (on the same line) only once. Remember that the BASIC program is always looping and the one shot allows execution only once. The integer range is 000-255.

ROS [(@)integer]

The ROS statement resets the specified one shot statement (000-255). The ROS is required because once a OST (one shot) has executed nothing else will allow the one shot to go again.

EXAMPLE:

```
10 XIL 016 : ROS 000
20 XIH 016 : OST 000: ?"Only here when 16 goes OFF to ON"
30 XIH 016 : ROS 001
40 XIL 016 : OST 001: ?"Only here when 16 goes ON to OFF"
.
REM lines 10-40 can be condensed
10 XIL 016: ROS 000: OST 001: ?" HIGH TO LOW"
20 XIH 016: ROS 001: OST 000: ?" LOW TO HIGH"
REM note the OSTs have to be after the ROSs

.
10 CLEAR 0
REM used to clear all oneshots
```

3.34 - OTC

The OTC statement clears all bits set ON by an OTE statement. The result is that any OTEs are only valid ON for each time the OTC executes. The OTC should be placed after the last LIO or EXIO statement and before the GOTO back to the start of the main loop.

3.35 - OTE [integer]

The OTE statement sets an output bit in the image table as the OTL statement but, when used in conjunction with the OTC statement the affect is quite different. Along with setting a bit in the image table the OTE also sets a special control bit. When the OTC statement executes (usually after a LIO) all control bits are checked and if any have been set by an OTE then that corresponding output bit is cleared. The net result is that the conditions which caused the OTE to execute must be true every scan in order to have the OTE stay ON. An example below will help to clarify this operation.

EXAMPLE:

```
10 REM show how a single input turns a output OFF and ON with
30 XIH 016: OTL 001
40 XIL 016: OTU 001
50 LIO: GOTO 30

10 REM show how the OTE makes the whole thing simpler
20 XIH 016: OTE 001
30 LIO: OTC: GOTO 20
```

3.36 - OTU [integer] or OTU! [integer]

The OTU statement un-latches or clears a bit in the controllers input/output image table. This image table is 64 bytes by 8 bits which yields 511 bits total. Therefore the range of the integer value in the statement is 000-511. All control statements use a high speed integer routine which requires that all leading zeros be included. The OTU! immediately turns OFF the specified local bits 0-15 without having to execute an LIO.

EXAMPLE:

```
10 OTU 000 : REM turn OFF bit0
```

3.37 - OTL [integer] or OTL! [integer]

The OTL is the opposite of the OTU statement. OTL latches or sets a bit ON in the image table. The OTL! immediately turns ON the specified local bit 0-15 without executing a LIO.

EXAMPLE:

```
10 OTU 001 : REM set bit 1 ON
```

3.38 - POP [variable]

The POP statement is used to pop the argument stack to the variables following the POP statement. If POP was executed and there where no variables on the stack then an argument stack error would result.