## IF...THEN

| BS1 | BS2 | BS2e | BS2sx | BS2p | BS2pe | BS2px |
|---|---|---|---|---|---|---|

NOTE: PBASIC 1.0 and PBASIC 2.0 only support IF *Condition(s)* THEN *Address*. PBASIC 2.5 supports all syntax variations.

1 All 2

**IF *Condition(s)* THEN *Address***

All 2

**IF *Condition(s)* THEN *Statement(s)* { ELSE *Statement(s)* }**

All 2

**IF *Condition(s)* THEN**
  ***Statement(s)***
**{ ELSEIF *Condition(s)* THEN**
    ***Statement(s)...* }**
 **{ ELSE**
    ***Statement(s)* }**
**ENDIF**

### Function

Evaluate *Condition(s)* and, if true, go to the *Address* or execute the *Statement(s)* following THEN, otherwise process the ELSEIF/ELSE block(s), if provided. ELSEIF is optional and works just like IF, but is only evaluated if the *Condition(s)* in the preceding IF is false. The ELSE block is optional and is executed if all *Condition(s)* in all preceding IF/ELSEIFs are false. The program will continue at the next line of code (single-line syntax) or the line that follows ENDIF (multi-line syntax) unless *Address* or *Statement(s)* are executed that cause the program to jump.

- ***Condition*** is a statement, such as "x = 7" that can be evaluated as true or false. *Condition* can be a very simple or very complex relationship, as described below.
- ***Address*** is a label that specifies where to go in the event that *Condition(s)* is true.
- ***Statement*** is any valid PBASIC instruction. Multiple statements may be placed on the same line (though not recommended) by separating each statement with a colon ( : ).

## Quick Facts

| | BS1 | All BS2 Models |
|---|---|---|
| **Comparison Operators** | =, <>, >, <, >=, <= | =, <>, >, <, >=, <= |
| **Conditional Logic Operators** | AND, OR | NOT, AND, OR, XOR |
| **Format of *Condition*** | *Variable Comparison Value*; where *Value* is a variable or constant | *Value1 Comparison Value2*; where *Value1* and *Value2* can by any of variable, constant or expression |
| **Parentheses** | Not Allowed | Allowed |
| **Max nested IF…THENs** | n/a | 16 |
| **Max ELSEIFs per IF** | n/a | 16 |
| **Max ELSEs per IF** | n/a | 1 |
| **Related Command** | None | SELECT…CASE |

**Table 5.38:** IF...THEN Quick Facts.

## Explanation

IF...THEN is PBASIC's decision maker that allows one block of code or another to run based on the value (True or False) of a condition. The condition that IF...THEN tests is written as a mixture of comparison and logic operators. The available comparison operators are:

| Comparison Operator Symbol | Definition |
|---|---|
| = | Equal |
| <> | Not Equal |
| > | Greater Than |
| < | Less Than |
| >= | Greater Than or Equal To |
| <= | Less Than or Equal To |

**Table 5.39:** IF...THEN Comparison Operators.

Comparisons are always written in the form: Value1 Comparison Value2. The values to be compared can be any combination of variables (any size), constants, or expressions.

NOTE: On the BS1, expressions are not allowed as arguments. Also, the Value1 (to the left of comparison) must be a variable. 1

1 All 2

The following example is an IF…THEN command with a simple condition:

A SIMPLE FORM OF IF…THEN

```
IF value < 4000 THEN Main
```

This code will compare the value of *value* to the number 4000. If *value* is less than 4000, the condition is true and the program will jump (implied

GOTO) to the label called *Main*. This is the simplest form of IF…THEN and is the only form supported by PBASIC 1.0 and PBASIC 2.0.

ALL ABOUT *CONDITION(S)*.

The *Condition(s)* argument is very powerful and flexible. In the next few pages we'll demonstrate this flexibility in great detail and afterwards we'll discuss the additional, optional arguments allowed by PBASIC 2.5.

Here's a complete example of IF...THEN in action:

NOTE: For BS1's, change line 1 to **SYMBOL value = W0** and line 4 to **DEBUG #value, CR**

```
value   VAR     Word

Main:
  PULSIN 0, 1, value
  DEBUG DEC value, cr
  IF value < 4000 THEN Main
  DEBUG "Pulse value was greater than 3999!"
```

Here, the BASIC Stamp will look for and measure a pulse on I/O pin 0, then compare the result, *value*, against 4000. If *value* is less than (<) 4000, it will jump back to Main. Each time through the loop, it displays the measured value and once it is greater than or equal to 4000, it displays, "Value was greater than 3999!"

All 2

On all BS2 models, the values can be expressions as well. This leads to very flexible and sophisticated comparisons. The IF…THEN statement below is functionally the same as the one in the program above:

```
IF value < (45 * 100 – (25 * 20)) THEN Val_Low
```

All 2

Here the BASIC Stamp evaluates the expression: 45 * 100 = 4500, 25 * 20 = 500, and 4500 – 500 = 4000. Then the BASIC Stamp performs the comparison: is *value* < 4000? Another example that is functionally the same:

```
IF (value / 100) < 40 THEN Val_Low
```

WATCH OUT FOR UNSIGNED MATH COMPARISON ERRORS

All 2

It's important to realize that all comparisons are performed using unsigned, 16-bit math. This can lead to strange results if you mix signed and unsigned numbers in IF...THEN conditions. Watch what happens here when we include a signed number (–99):

```
 Test:
  IF -99 < 100 THEN Is_Less
  DEBUG "Greater than or equal to 100"
  END

Is_Less:
  DEBUG "Less than 100"
  END
```

Although –99 is obviously less than 100, the program will say it is greater. The problem is that –99 is internally represented as the two's complement value 65437, which (using unsigned math) is greater than 100. This phenomena will occur whether or not the negative value is a constant, variable or expression.

LOGICAL OPERATORS (NOT, AND, OR AND XOR).

IF...THEN supports the conditional logic operators NOT, AND, OR, and XOR to allow for more sophisticated conditions, such as multi-part conditions. See Table 5.38 for a list of the operators and Table 5.40 for their effects.

NOTE: The NOT and XOR operators are not available on the BS1.

The NOT operator inverts the outcome of a condition, changing false to true, and true to false. The following IF...THENs are equivalent:

```
IF x <> 100 THEN Not_Equal
IF NOT x = 100 THEN Not_Equal
```

The operators AND, OR, and XOR can be used to join the results of two conditions to produce a single true/false result. AND and OR work the same as they do in everyday speech. Run the example below once with AND (as shown) and again, substituting OR for AND:

1

NOTE: For BS1's, change lines1 and 2 to:
**SYMBOL value1 = B2**
**SYMBOL value2 = B3**

```
value1          VAR     Byte
value2          VAR     Byte

Setup:
  value1 = 5
  value2 = 9

Main:
  IF value1 = 5 AND value2 = 10 THEN Is_True
  DEBUG "Statement is False"
  END

Is_True:
  DEBUG "Statement is True"
  END
```

The condition "value1 = 5 AND value2 = 10" is not true. Although value1 is 5, value2 is not 10. The AND operator works just as it does in English; both conditions must be true for the statement to be true. The OR operator also works in a familiar way; if one or the other or both conditions are true, then the statement is true. The XOR operator (short for exclusive-OR) may not be familiar, but it does have an English counterpart: If one condition or the other (but not both) is true, then the statement is true.

1
NOTE: On the BS1, parentheses are not allowed within arguments.

Table 5.40 below summarizes the effects of the conditional logic operators. On all BS2 models you can alter the order in which comparisons and logical operations are performed by using parentheses. Operations are normally evaluated left-to-right. Putting parentheses around an operation forces PBASIC 2.0 and PBASIC 2.5 to evaluate it before operations that are not in parentheses.

**Table 5.40:** Conditional Logic Operators Truth Tables.

1
NOTE: The NOT and XOR operators are not available on the BS1.

| Truth Table for Logical Operator: NOT | |
|---|---|
| **Condition A** | **NOT A** |
| False | True |
| True | False |

| Truth Table for Logical Operator: AND | | |
|---|---|---|
| **Condition A** | **Condition B** | **A AND B** |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

| Truth Table for Logical Operator: OR | | |
|---|---|---|
| **Condition A** | **Condition B** | **A OR B** |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

| Truth Table for Logical Operator: XOR | | |
|---|---|---|
| **Condition A** | **Condition B** | **A XOR B** |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | False |

INTERNAL REPRESENTATION OF BOOLEAN VALUES (TRUE VS. FALSE).

Internally, the BASIC Stamp defines "false" as 0 and "true" as any value other than 0. Consider the following instructions:

All 2

```
flag    VAR     Bit

Setup:
  flag = 1

Test:
  IF flag THEN Is_True
  DEBUG  "False"
  END

Is_True:
  DEBUG "True"
  END
```

Since *flag* is 1, IF...THEN would evaluate it as true and print the message "True" on the screen. Suppose you changed the IF...THEN command to read "IF NOT flag THEN Is_True." That would also evaluate as true. Whoa! Isn't NOT 1 the same thing as 0? No, at least not in the 16-bit world of the BASIC Stamp.

Internally, the BASIC Stamp sees a bit variable containing 1 as the 16-bit number %0000000000000001. So it sees the NOT of that as %1111111111111110. Since any non-zero number is regarded as true, NOT 1 is true. Strange but true.

AVOIDING ERRORS WITH BOOLEAN RESULTS.

The easiest way to avoid the kinds of problems this might cause is to always use a conditional operator with IF...THEN. Change the example above to read IF flag = 1 THEN is_True. The result of the comparison will follow IF...THEN rules. Also, the logical operators will work as they should; IF NOT Flag = 1 THEN is_True will correctly evaluate to false when *flag* contains 1.

This also means that you should only use the "named" conditional logic operators NOT, AND, OR, and XOR with IF...THEN. The conditional logic operators format their results correctly for IF...THEN instructions. The other logical operators, represented by symbols ~, &, |, and ^ do not; they are binary logic operators.

All 2

The remainder of this discussion only applies to the extended IF…THEN syntax supported by PBASIC 2.5.

IF…THEN WITH A SINGLE *STATEMENT*

In addition to supporting everything discussed above, PBASIC 2.5 provides enhancements to the IF…THEN command that allow for more powerful, structured programming. In prior examples we've only used the first syntax form of this command: IF *Condition(s)* THEN *Address*. That form, while handy in some situations, can be quite limiting in others. For example, it is common to need to perform a single instruction based on a condition. Take a look at the following code:

```
' {$PBASIC 2.5}

x  VAR  Byte

FOR x = 1 TO 20                           ' count to 20
  DEBUG CR, DEC x                         ' display num
  IF (x // 2) = 0 THEN DEBUG " EVEN"      ' even num?
NEXT
```

This example prints the numbers 1 through 20 on the screen but every even number is also marked with the text " EVEN." The IF…THEN command checks to see if *x* is even or odd and, if it is even (i.e.: x // 2 = 0), then it executes the statement to the right of THEN: DEBUG "  EVEN." If it was odd, it simply continued at the following line, NEXT.

Suppose you also wanted to mark the odd numbers. You could take advantage of the optional ELSE clause, as in:

```
' {$PBASIC 2.5}

x  VAR  Byte

FOR x = 1 TO 20                           ' count to 20
  DEBUG CR, DEC x
  IF (x // 2) = 0 THEN DEBUG " EVEN" ELSE DEBUG " ODD"
NEXT
```

This example prints the numbers 1 through 20 with " EVEN" or " ODD" to the right of each number. For each number (each time through the loop) IF…THEN asks the question, "Is the number even?" and if it is it executes DEBUG " EVEN" (the instruction after THEN) or, if it is not even it executes DEBUG " ODD" (the instruction after ELSE). It's important to note that this form of IF…THEN always executes code as a result of *Condition(s)*; it either does "this" (THEN) or "that" (ELSE).

SINGLE-LINE VS. MULTI-LINE IF…THENS

The IF…THEN in the previous example is called a "single-line" syntax. It is most useful where you only have one instruction to run as the result of a *Condition*. Sometimes this form may be a little hard to read, like in our above example. For these cases, it would be better to use the "multi-line" syntax of IF…THEN. The multi-line format allows the same flexibility as the single-line format, but is easier to read in most cases and requires an ENDIF statement at the end. For example, our IF…THEN line above could be re-written as:

```
IF (x // 2) = 0 THEN
  DEBUG " EVEN"                        ' even number
ELSE
  DEBUG " ODD"                         ' odd number
ENDIF
```

This example runs exactly the same way, is much easier to read and also leaves extra room to add some helpful comments on the right. We also indented the *Statement(s)* for clarity and suggest you do the same.

Did you notice that multi-line syntax requires ENDIF to mark the end of the IF…THEN…ELSE construct? That is because the *Statement(s)* argument can be multiple instructions on multiple lines, so without ENDIF there is no way to know just where the IF…THEN…ELSE ends.

NESTED IF…THENS

Occasionally, it may be necessary to have compound IF statements. One way to achieve this is through nested IF…THEN…END constructs:

```
' {$PBASIC 2.5}

value           VAR     Word

DO
  PULSIN 0, 1, value                          ' measure pulse input
  DEBUG DEC value, CR
  IF (value > 4000) THEN                      ' evaluate duration
    DEBUG "Value was greater than 4000"
  ELSE
    IF (value = 4000) THEN
      DEBUG "Value was equal to 4000"
    ELSE
      DEBUG "Value was less than 4000"
    ENDIF
  ENDIF
  DEBUG CR, CR
  PAUSE 1000
LOOP
```

Here, the BASIC Stamp will look for and measure a pulse on I/O pin 0, then compare the result, *value*, against 4000. Based on this comparison, a message regarding the pulse width value will be printed.

If *value* is greater than 4000, "Value was greater than 4000" is printed to the screen. Look what happens if *value* is not greater than 4000… the code in the ELSE block is run, which is another IF…THEN…ELSE statement. This "nested" IF…THEN statement checks if *value* is equal to 4000 and if it is, it prints "Value was equal to 4000" or, if it was not equal, the last ELSE code block executes, printing "Value was less than 4000." Up to sixteen (16) IF…THENs can be nested like this.

USING THE ELSEIF CLAUSE

The nesting option is great for many situations, but, like single-line syntax, may be a little hard to read, especially if there are multiple nested statements or there is more than one instruction in each of the *Statement(s)* arguments. Additionally, every multi-line IF…THEN construct must end with ENDIF, resulting in two ENDIFs right near each other in our example; one for the innermost IF…THEN and one for the outermost IF…THEN. For this reason, IF…THEN supports an optional ELSEIF clause. The ELSEIF clause takes the place of a nested IF…THEN and removes the need for multiple ENDIFs. Our IF…THEN construction from the example above could be rewritten to:

```
' {$PBASIC 2.5}

IF (value > 4000) THEN                      ' evaluate duration
  DEBUG "Value was greater than 4000"
ELSEIF (value = 4000) THEN
  DEBUG "Value was equal to 4000"
ELSE
  DEBUG "Value was less than 4000"
ENDIF
```

This IF…THEN construct does the same thing as in the previous example:

1) if value is greater than 4000:
   it displays "Value was greater than 4000"
2) else, if value is equal to 4000 (the ELSEIF part):
   it displays "Value was equal to 4000"
3) and finally (ELSE) if none of the above were true:
   it displays "Value was less than 4000"

Note that an IF…THEN construct can have as many as sixteen (16) ELSEIF clauses, but a maximum of only one (1) ELSE clause.

There are three demo programs below. The first two demonstrate the PBASIC 1.0 (BS1) and PBASIC 2.0 (all BS2 models) form of the IF…THEN command. The last example demonstrates the PBASIC 2.5 (all BS2 models) form of IF…THEN.

## Demo Program (IF-THEN.bs1)

1

```
' IF-THEN.bs1
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number.  The program counts how many numbers it
' prints, and quits when this number reaches 10.

' {$STAMP BS1}
' {$PBASIC 1.0}

SYMBOL  sample          = W0            ' Random number to be tested
SYMBOL  samps           = B2            ' Number of samples taken
SYMBOL  temp            = B3            ' Temporary workspace


Setup:
  sample = 11500

Mult3:
  RANDOM sample                         ' Put a random number into sample
  temp = sample // 3
  IF temp <> 0 THEN Mult3               ' Not multiple of 3? -- try again
    DEBUG #sample, "divides by 3", CR   ' show sample divisible by 3
    samps = samps + 1                   ' Count multiples of 3
    IF samps = 10 THEN Done             ' Quit with 10 samples
  GOTO Mult3                            ' keep checking

Done:
  DEBUG CR, "All done."
  END
```

## Demo Program (IF-THEN.bs2)

All 2

NOTE: This example program can be used with all BS2 models by changing the $STAMP directive accordingly.

```
' IF-THEN.bs2
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number.  The program counts how many numbers it
' prints, and quits when this number reaches 10.
' {$STAMP BS2}
' {$PBASIC 2.0}


sample          VAR     Word            ' Random number to be tested
samps           VAR     Nib             ' Number of samples taken
temp            VAR     Nib             ' Temporary workspace


Setup:
  sample = 11500

Mult3:
  RANDOM sample                         ' Put a random number into sample
  temp = sample // 3
  IF temp <> 0 THEN Mult3               ' Not multiple of 3? -- try again
    DEBUG DEC5 sample, " divides by 3", CR
    samps = samps + 1                   ' Count multiples of 3
    IF samps = 10 THEN Done             ' Quit with 10 samples
  GOTO Mult3                            ' keep checking

Done:
  DEBUG CR, "All done."
  END
```

All 2

## Demo Program (IF-THEN-ELSE.bs2)

NOTE: This example program can be used with all BS2 models by changing the $STAMP directive accordingly.

```
' IF-THEN-ELSE.bs2
' The program below generates a series of 16-bit random numbers and tests
' each to determine whether they're evenly divisible by 3. If a number is
' evenly divisible by 3, then it is printed, otherwise, the program
' generates another random number. The program counts how many numbers it
' prints, and quits when this number reaches 10.

' {$STAMP BS2}
' {$PBASIC 2.5}                         ' version 2.5 required

sample          VAR     Word            ' Random number to be tested
hits            VAR     Nib             ' Number of hits
misses          VAR     Word            ' Number of misses


Setup:
  sample = 11500

Main:
  DO
    RANDOM sample                       ' Put a random number into sample
    IF ((sample // 3) = 0) THEN         ' divisible by 3?
      DEBUG DEC5 sample,                ' - yes, print value and message
            " is divisible by 3", CR
    hits = hits + 1                   ' count hits (divisible by 3)
  ELSE
    misses = misses + 1               ' count misses
  ENDIF
LOOP UNTIL (hits = 10)                ' quit after 10 hits

DEBUG CR,
      "All done.", CR, CR,            ' display results
      "Hits:    ", DEC hits, CR,
      "Misses:  ", DEC misses, CR,
      "Samples: ", DEC (hits + misses)
END
```