# BenchSystem Operating Language™, BSOP

**Tools for Scientists and Engineers**

## Features

|  |  |
|---|---|
| – **Powerful keywords to simplify communications with equipment:** | **gpib, lib, comm, console.** |
| – **Powerful keywords to parse and build text strings:** | **find, arg, copy, trim, format** |
| – **Data types all work together with automatic type-conversion:** | **int, float, string** |
| – **Arithmetic, Bit, Logical, and String operators:** | **+ - * / , & \| ~ ^ , < > ! , @** |
| – **Modern constructs to support reliability and documentation:** | **subroutine, alias, long name fields** |
| – **Advanced optional in-line programming.** |  |

## Overview

The BenchSystem Operating Language (BSOP) was designed to be a programming tool for non-professional programmers. BSOP combines the high-level IO concepts of BASIC with the look of C. The result is a small language that "codes" quickly and "documents" nicely.

## A Short Primer

BSOP is free-format. The structure is similar to the notes you might take when planning an experiment. Spaces, tabs, and blank lines are separators, and can be used freely to format a program for readability. Parentheses force the order-of-completion as they "bundle" a group of operations "in-line" into a single result.

Refer to the following code sample for our discussion of syntax:

```
/       **** SIMPLE TEST Example ****
/       Generate some output to the console terminal.
/       revision: date:     author:

int     x ;                                   / loop counter

start:  console       "Hello  ";              / print a greeting

        x = x + 1 ;
        if( x < 4 )   goto start ;            / limit the test loop

        exit ;                                / bye bye
```

At the top of the listing we "title" the experiment. Next we make declarations, and finally fall into the body of the program, where the actions take place. The "look" of this program is achieved by using tabs and spaces to keep labels, statements and comments in separate columns

Let's define some terms, starting at the top of the program listing

- We begin with **comment** lines that explain the purpose and operation of the program.
- Next we declare the **variables** (and **data types**) we will be using to hold numerical information during our experiment. We have used more comments to further explain the data. BSOP sets all variables to zero before the program "starts".
- Execution begins at the **start** label. **Label**s are reference points, generally for branching destinations.
- A **statement** ends with a semi-colon ';' and is an executable "chunk" of code. In this case we are sending a text string to the console terminal.
- **Keywords** are part of the BSOP language. The keywords in our example program are: **int, start, if, console, goto,** and **exit**. Labels and subroutine names, defined by the programmer, must be different from the keywords.
- **Operators** such as '+' or '<' or '=' are used to perform arithmetic, logical, or assignment actions.
- Numbers or strings found in the code body are called **constants**. Constants can be either numeric or text string, and are generally used with arithmetic and text string formation.
- x = x + 1; is a mathematical **expression**. The right-hand side uses the existing value of x throughout the calculation. The result of the calculation is then assigned to the left-hand side of the '=' operator.

## Comments

Comments are used to make notes or to further explain what a statement is doing; BSOP ignores them. Comments begin with a slash '/' and end at the end of the line. The slash must be the first non-space character on the line, or following a statement-ending semicolon. Comments are optional, but you'll be glad you took the time to write comments when you have to look at your code a few weeks later. Try to write them with the future in mind.

## Variables

Variables are storage bins that hold numbers (**int**eger, **float**ing point) or text (**string**) information. Integers range between +-2billion, floats between +-1e+-38, and strings hold up to 126 text characters. Multiple variables of the same type may be declared in a single statement, if separated by commas. The declaration is terminated by a semicolon ';'. Variables are 'looked at' in expressions, but are modified only by assignment.

```
example:        int x, y, z ;        / declare 3 integer variables

                x = x + y + 2;       / add the values of x, y, and 2 and store the result in x.
```

note: BSOP initializes all variables to zero when the program starts.

## Names

The programmer gives names to variables, labels, subroutines, and aliases. Names are made of alphanumeric characters and underscore '_'. *The first character must be alphabetic.* Names can be up to 126 characters long and are case sensitive, i.e. "var1" is different from "Var1".

**tip:** Meaningful names make the code easier to follow.

## Labels

Labels mark jump-destinations in the code. Label definitions start with the first non-space characters of a statement and are followed by a colon ':'.

## Statements

Statements are used to make decisions, compute numbers, communicate with equipment, and call routines. Code statements are terminated by semicolons ';' and may span more than one line; or a line may include more than one statement.

## Constants (numeric and text)

Numeric constants can be in decimal, hexadecimal, floating, or exponential format; i.e. -12, 0xF32, 12.345, or 1.2e-5. Hexadecimal representations begin with '0x' or '0X'. Numbers must begin with a sign (+-) or a digit (0-9).

Text string constants are enclosed in quotation marks and include any spaces, 'This is a sting constant. ".

## Operators and Operands

Operators are used for arithmetic calculations, logical comparisons and string concatenations. Operators work on operands; unary operators work on one operand, binary operators work on two operands. In the expression (5 * -2), five times negative two, 5 and 2 are operands, * is a binary operator, - is a unary operator.

Operands may be variables or constants of any type: int, float or string. For example, the expression:
.

```
var2 = var1 * -10 ;
```

will multiply var1 by negative 10 and save the result to var2. If var2 is int, and var1 is float, then BSOP will promote 10 to 10.0 during computation, and the result will be demoted to an int when it is assigned to var2. Demotion from float to int truncates the fractional part of the float, i.e. 9.56 is truncated to 9. See the section on ***Automatic Type Conversion Rules***.

Expressions are evaluated left to right, and operators have various priorities. The expression (3 + 4 * 8 ) evaluates to 35. Use parenthesis to force a different order, i.e. ((3 + 4) * 8) gives 56.

| operator | description | priority |
|---|---|---|
| + - ! ~ | unary. associates w/operand to the right | highest |
| * / % | binary. associates left to right | |
| + - | binary. associates left to right | |
| > < >= <= | binary. associates left to right | |
| != == | binary. associates left to right | |
| & | binary. associates left to right | |
| ^ | binary. associates left to right | |
| \| | binary. associates left to right | |
| && | binary. associates left to right | |
| \|\| | binary. associates left to right | |
| @ | binary. associates left to right | |
| = | binary. associates right to left | lowest |

Constant and variable operands used in an expression are <u>not</u> altered during operations, only their values are manipulated. Operators are divided into the following classes (there are **no spaces** between symbols in multi-symbol operators):

### Arithmetic

| | | | |
|---|---|---|---|
| + | Addition: | r = a + b | |
| - | Subtraction: | r = a – b | |
| | also for negation | r = a + -b | |
| * | Multiplication: | r = a * b | |
| / | Division: | r = a / b | |
| % | Modulo divide | r = a % b | the result is the remainder of a / b. |

### Binary (bit-level manipulations)

The symbol for Or '|' is shift-backslash '\' on some keyboards. The examples show only 4 of the 32 available bits.

| | | | |
|---|---|---|---|
| & | And: | r = a & b | 6 (0110) & 3 (0011) makes 2 (0010) |
| \| | Or: | r = a \| b | 6 (0110) \| 3 (0011) makes 7 (0111) |
| ^ | Exclusive Or: | r = a ^ b | 6 (0110) ^ 3 (0011) makes 5 (0101) |
| ~ | Inversion: | r = a & ~b | 6 (0110) & ~3 (1100) makes 4 (0100) |

### Arithmetic tests

| | | |
|---|---|---|
| > | Greater than: | if( a > b ) |
| < | Less than: | if( a < b ) |
| >= | Greater than or equal to: | if( a >= b ) |
| <= | Less than or equal to: | if( a <= b ) |
| != | is not equal to: | if( a != b ) |
| == | is equal to: | if( a == b ) |

## Logical tests

| | | | |
|---|---|---|---|
| && | AND: | if( (a < b) && (b<c) ) | |
| \|\| | OR: | if( (a < b) \|\| (b<c) ) | '\|' is shift-backslash '\' on some keyboards |
| ! | NOT: | if( ! (b<c) ) | |

## Assignment

| | | | |
|---|---|---|---|
| = | Assign right to left: | r = a & b | a and b are not affected by the operation |

## Text Strings

Text strings are made up of ascii characters, which are a set of one-byte values (0 to 255) used for information exchange between terminals, printers and other computer equipment. For example, the byte value of the ascii character 'A' is 65 (or 41 hex).

| | | | |
|---|---|---|---|
| "" | text constant definition: | "Enter >" | This string is 7 characters, including the space, and could serve to prompt the operator at the console terminal. |
| @ | Concatenate strings: | "ta" @ "bx" | the result is "tabx" |
| | if the value of nvar is 45: | "T = " @ nvar | the result is "T = 45"; the number is converted to text |
| | if the value of svar is HI: | "P = " @ svar | the result is "P = HI" |

### Special Characters (Escape Sequences)

Escape sequences make it possible to put non-printable ascii characters or any byte values into a text string.

| | | | |
|---|---|---|---|
| \n | ascii "new line" | value = 10 | "Volts? \n" |
| \r | ascii "carriage return" | = 13 | |
| \t | ascii "horizontal tab" | = 9 | |
| \b | ascii "back space" | = 8 | |
| \" | ascii "quote" | = 34 | use to put quotes within a string constant. |
| \\ | ascii "back slash" | = 92 | use to put a back slash within a string constant. |
| \xnn | nn are two hex digits | = 00 to ff | "\x07" Ascii BELL, will sound at most consoles |
| \Xnn | nn are two hex digits | = 00 to ff | |

## Automatic Type Conversion Rules

BSOP simplifies the code by interpreting variables per the context of the statement. It is a good idea to know the rules. Variables are never physically altered during a computation unless by assignment, i.e. they are on the left side of '='. Promotions/demotions only affect the "value" represented by the variable as it's used by the computation.

### Strings

Strings are interpreted to be the 'numerical value' represented, in numerical operations, i.e. "6.34" / 2 yields 3.17. Note that non-numerical characters terminate the numeric conversion and therefore, alphabetic strings have a value of zero.

See the '@' operator to concatenate a variable to a text string.

### Float to int demotion

When a float value is demoted to an int, as would be the case in **intvar = floatvar + 3.2 ;** the decimal value is dropped. Be careful since the float value for say 1.0 might be stored as 0.99999, and conversion would be 0. To round the final result add 0.5 to positive values, and add –0.5 to negative values before the demotion assignment. This is left to the programmer since rounding isn't always wanted.

### Int to float promotion

Integers and string values are promoted to floats in a computation as soon as a float enters into the calculation or comparison.

## Keywords

Keywords are special names that are recognized by BSOP. Many require arguments, which follow the keyword in a specific order. The keyword, with arguments, is referred to as a keyword phrase. Be sure not to duplicate keywords when naming your variables, labels, etc.

In the following definitions:

- s1,s2 are string constants or variables.
- n1,n2 are numerical constants or variables.
- sr,nr indicate that a keyword returns a string or numeric value, which may be assigned to a variable by using '=', used as an argument itself, or just ignored; as determined by the context of the statement.

### *Branching*

Branching controls the flow of the code execution. Code flow, like a flow chart, often initializes hardware and processes, then controls events based on how the system is performing.

| | | | |
|---|---|---|---|
| **goto** | moves the code execution | goto test_loop ; | / execution branches to 'test_loop" |
| **if** | if (true condition) do this | if( a < b ) goto test3 ; | / skip the branch if a>=b |
| **else** | optionally follows an 'if statement" | else goto test4 ; | / do this if the preceding 'if' fails |
| **exit** | stop and leave the program | exit ; | / leave the program, we're done. |
| **step** | Pause program flow if in debug mode | step ; | / pause, wait for console input |

The keyword **if** is followed by a logical or arithmetic test result, and executes the remainder of the statement if the result is true; otherwise it skips over it. If the subsequent statement begins with **else**, then that statement executes only if the preceding 'if' did not execute.

The keyword **goto** is optional; just writing the destination, i.e. if( n > 9 ) test3; will branch just the same.

**Code Sample**

```
int     x ;

start:  x = console 'Enter a value from –10 to +10 > " ;      / prompt and then wait for input
        if( x < -10 || x > 10 )    goto quit;                  / test for valid input
        else if ( x < 5 )          console 'Value is less than 5. ";     / compare valid input to limit
        else                       console 'Value is greater than 4. ";
quit:   exit;
```

### *Device-specific Functions*

See the *data sheet* of the *Controller* running the BSOP interpreter, for detailed definitions of the specific keywords that support the its capabilities. Some examples are briefly noted below:

| | |
|---|---|
| **clrgpib** | Force IFC, DCL and initialize the GPIB port. |
| **gpib n1 s1** | Send and/or receive over GPIB port. |
| **comm s1** | Send and/or receive over the AUX port . |
| **commchr** | Receive one character over the AUX port. |
| **defcomm n1 s1** | Define baud and protocol. |
| **lib** | Send and/or receive over the LIB port |
| **console** | Send and/or receive over the CONSOLE port |
| **conschr** | Receive one character over the CONSOLE port |
| **clockms** | return the system clock time in mS since power up. |
| **waitms n1** | delay program 0-65000 ms |

## Text String Parsing

| | | | |
|---|---|---|---|
| **copy n1 n2 s1** | copy chars of s1 per n1and n2 | sr = copy 2 3 “awxyz”; | / returns “wxy” |
| | n1 (>0) char position from start | | |
| | n1 (<0) char position from end | sr = copy -3 2 “awxyz”; | / returns “xy” |
| | n2 (>0) is number of chars to copy | | |
| | n2 (<0) is char position from end | sr = copy 1 –2 “awxyz”; | / returns “awxy” |
| **find s1 s2** | find string s1in s2 | nr = find “cd” “abcde”; | / returns 3 |
| | Returns char position from start. | nr = find “dc” “ abcde”; | / returns 0 |
| | If s1 is null “”, return the string length | nr = find “” “ abcde”; | / returns 5 |
| **trim s1** | remove leading/trailing spaces and tabs | sr = trim “   xy  z  “; | / returns “xy  z” |
| **arg n1 s1** | copy comma-separated arg # n of s1 | sr = arg 2 “ab,bc,cd”; | / returns “bc” |

## Text String Number Formatting

The format statement defines the default format to be used when building strings with float-type numbers. Floating point format is “f4” which means float with 4 significant digits. Other formats are ‘e’ and ‘E’ which are both exponential. The range of ‘f’ digits is from 1 to 20, the range for ‘e’ or ‘E’ is from 1 to 7 and the exponent between +-55. The initial default format is “e6”.

| | | | |
|---|---|---|---|
| **format s1** | floating point | format “f4”; | / float, 4 significant digits 12.3456 |
| | print with lower case exponent | format “e2”; | / exp, 2 significant digits 1.23e+01 |
| | print with upper case exponent | format “E3”; | / exp, 3 significant digits 1.234E+01 |

**Code Sample**

```
alias   NL      console “\r\n”;                 / go to the next line on the console
float   x ;

start:  x = console “Enter a number > “; NL;     / prompt operator and wait for a value.  Interpret as floating point.
        format “f2”;    console x ; NL;
        format “e2”;    console x ; NL;
        format “E4”;    console x ; NL;
        if( x < 1e30 )  goto start;             / set an escape point
        exit;
```

## Math

| | | | |
|---|---|---|---|
| **abs n1** | return the absolute value | nvar = abs n1 ; | / magnitude of n1 |
| **sqrt n1** | return the square root | nvar = sqrt n1 ; | / n1< 0 is illegal, returns n1. |

## Declarations

Declarations begin as the first non-space characters of a statement. Declarations can be made anywhere in the program. It is often a good idea to group variables and aliases, then place the groups near the block of code that is most associated with the their definition.
.
Storage space and data-types for variables must be declared in the program. Good comments by the data and descriptive names help one understand the program.

Alias names can be thought of as inserting the declared text wherever the name is used, for example, the address path of a device or the value of a constant. They improve readability and reliability in that changing the text is done in only one place, and it is the same wherever the alias is used.

Subroutines are implicitly declared by the syntax of a "block" of statements, contained within braces '{}', immediately following the subroutine name. The subroutine block of statements "runs" whenever its name is encountered during code execution.

### *Variables*

| | | | |
|---|---|---|---|
| **int v** | range**:** -2,147,483,648 to +2,147,483,647 | int v1,v2 ; | / declare 2 integer variables |
| **float v** | range**:** +/-1.175e-38 to 3.40e+38 | float v3 ; | |
| **string v** | range**:** 0 to 126 text characters | string v4 ; | |

### *Alias*

| | | |
|---|---|---|
| **alias Name alias_text** | alias Vmeter gpib 6 ; | / use Vmeter "V?"; to take a reading |
| | | / in place of gpib 6 "V?"; |
| | alias PI 3.1415926 ; | / use PI in calculations |

### *Subroutines*

```
subroutine_name
{    statement ;
          …
     statement ;
}
```

## Example BSOP Code

The following code samples were cut from a program used to calibrate a multi channel system controller. The top of the listing holds the declarations for the code body that follows them. Not all of the DMM aliases are used in these routines; not all of the routines are shown. This sample illustrates an interactive user input mode and prompting. Aliases are a matter of style and are optional.

```
/------------------------------------------
/           MSBC INPUT CALIBRATION PROGRAM
/           Using BTC1 controller.
/
/           rev: 1     9feb02   Initial.
/------------------------------------------

alias     N                   "\r\n"     ;                             / looks better in statements, subjective
alias     NL                  console N;

string    cmd;                                                         / the operator's 'typed in' command sting


start:    console "--- MSBC INPUT CALIBRATION (c)ExacTest Corp'  02--\r\n\n" ;
          clrgpib;
          init_dmm;                                                    / initialize the attached test equipment

loop:     cmd = console "Enter: '  cal'  , '  save'  , '  status'  , or '  quit'  : " ; NL;   / put a new line after the input is 'entered'

          if        (find "quit" cmd)          quit;                   / jump per what was typed in
          else if   (find "save" cmd)          save_cal_msbc;          / call subroutine per what was typed in
          else if   (find "cal"  cmd)          cal_msbc;
          else if   (find "sta"  cmd)          status_msbc;
          else                                 console "Error: Invalid entry.\r\n";       / no matches
          NL; loop;                                                                       / go back for more
quit:
          console "bye bye..\r\n";
          clrgpib;                                                     / return equipment to a manual state
          init_dmm;
          exit;


/************** Digital Multi-Meter ROUTINES **************
/ The following statements are used to configure the Keithley Instruments model 2001 dmm for
/ the various  tests and procedures used by this calibration program.
/ Refer to the dmm manual for device commands and syntax.

alias     dmm                 gpib 16 ;                                / port and address of dmm
alias     setV200             "conf:volt:dc;:volt:dc:rang 200";        / dmm-specific configuration strings
alias     setI200mA           "conf:curr:dc;:curr:dc:rang .2";
alias     read_dmm_mA         dmm "read?" * 1000 ;                     / read dmm and normalize to mA

init_dmm
{
          dmm ":system:preset \n";                                     / preset defaults
          dmm ":format:elem read \n";                                  / send only the reading
          dmm  setV200;                                                / select a configuration
}
```

## Design and Debug

Begin a project by writing down what you want the system to do. Define test criteria, equivalent circuits, and user interface messages. Consider timing, data, and errors. Define the process in terms of tasks. Resolve any unknowns. Use interactive commands and small test programs to verify operation of the attached equipment.

Use a modular approach to build and debug the system. Use low-level modules to communicate with the attached devices or on-board functions. Rough in the highest-level modules right from the start, so the whole system is kept in view. Fill in the modules as needed to develop each function of the system, testing as you go. Thoroughly test each module over its entire range of stimuli and limits.

Debugging is a process used to verify that a program is working as intended. The BTC1 provides single-stepping and descriptive error messages as debug tools. Single stepping allows the programmer to follow the code execution statement-by-statement. See the console programming command section.

You may also put temporary statements into your program that print out variable values and markers to indicate execution. These can later be removed or just "commented out" by placing a slash '/' in front of them. You could use an alias to control printing and even range testing:

```
int debug;       debug = 1;                                   / debug control , set to 0,1,2

…code …

if( debug > 0)          console var1 ;                        / conditional print a value

… code …

if(debug == 2 )         var2 = console "enter test  var >";   / prompt to assign a value
```