

Part File Programming Handbook for the OpenSBP® Language

OpenSBP® Language Part Files offer virtually unlimited control of your tool and the execution of cutting instructions. This document describes working with Part Files and how to use the additional OpenSBP® 'Program Statements' to help control or program how a Part File runs.

ShopBot Tools, Inc.
3333 Industrial Drive
Durham, NC 27704
919.680.4800 or 888.680.4466

CONTENTS

Part File Programming Handbook for the OpenSBP® Language	1
OpenSBP® Part Files (.sbp).....	3
What's in a Part File?.....	3
Using the Editor.....	4
The Part File Layout.....	4
Introducing Part File Programming Statements.....	5
Nesting or Embedding Part Files	5
Math, User Variables (&), and System Variables(%)	7
Using Variables.....	7
Types of Program Statements	10
Control/Flow Statements.....	10
Conditional Branching Statements	10
Input/Output Statements	10
File Statements.....	11
Reference: Part File Programming Control Statements	12
Additional Part File Programming Considerations	28
Movement Blocks	28
Structuring Your Files.....	29
Adding Setup Lines to the Start of a File.....	29
Troubleshooting difficulties in Part Files	29
Why We Have a ShopBot Part File Format	31
ShopBot Log Files.....	32
Advanced Start-Up Information.....	33
Windows Command Line Options...Controlling the ShopBot software from outside programs (a)	33
Windows Registry Interface...Controlling the ShopBot software from outside programs (b).....	34
Virtual Tool System	35
System Variables [use as %(sys var#)]	36

OpenSBP® Part Files (.sbp)

You could control your ShopBot tool entirely through keyboard input using the OpenSBP®, 2-Letter Commands. However, the real potential of CNC and robotics tools is achieved by supplying your tool with a list of instructions to carry out complex cutting and machining movements. Such intricate cutting capability is why a robotics tool can do things you can't - and it can do them hundreds or thousands of times without a glitch.

What's in a Part File?

The list of instructions is provided through Part Files that you write in a Text Editor or generate in other software such as PartWorks, and bring to ShopBot. This list of instructions can be just a list of Commands you might enter from the keyboard using the two key-stroke shortcuts. On each line of the list there is the two-letter Command, followed in most cases by parameters giving the details of the action.

A Part File can be a short and simple list that looks like the following:

```
JZ, .5
J2, 10, 10
MZ, -.25
M2, 20, 10
M2, 20, 5
M2, 10, 5
M2, 10, 10
JZ, .5
```

Here we have instructed your tool to cut a rectangle. First (the JZ, .5), we pull the bit up to make sure it is not in the material when we make our positioning move (typically, you set up your Z axis with the convention that the top of your material is the Z location 0). Next, the tool will Jog to the XY location 10, 10 (we will assume that the ShopBot is configured for Absolute Distance moves). Then the bit is moved down to -.25 (we use a 'M'ove so that we will be moving at Move Speed now because we are cutting into material), and will drill into the material and be ready for our cut. There are then 4 cutting speed moves (M2's) that will define a 10 x 5 rectangle, with the starting point being in the upper left corner of the rectangle, and going around the rectangle in a clockwise direction (if you were using a 1/4in bit, this would create a cut-out that is actually 9.75 x 4.75 because the center of the bit would be following the specified path).

A OpenSBP® Part file can also, however, also contain Programming Commands that can only appear in Part files and can't be entered at the keyboard...that's what the rest of this manual is about.

Using the Editor

The files that ShopBot reads are in what is called 'text' format. That means you can look at them and edit them with any text editor or word-processor, however we recommend using the SbEdit text editor that comes with the ShopBot software. You can open the ShopBot Editor by typing [**FE**] or open a new blank part file by typing [**FN**]. You can also start SbEdit from the Windows 'Start' menu or by just double clicking on any ShopBot Part File. If you do use your own word-processor, make certain that after you have created or modified a Part File that it is re-saved in generic 'text' format and not a special format of your word-processor. The file name extension for a ShopBot Part file is '.sbp' . Other ShopBot related files have file extensions that start with '.sb_' and end with another letter for descriptive convenience. The file extensions '.ini', '.opt', and '.sbd' have special purposes and are discussed elsewhere, but these too are text files in ShopBot Part File format.

The Part File Layout

A Part File can consist of a single line or up to a million lines, but can have only one Command per line. You can generally use upper and lower case letters to your own preference. You can indent and skip lines and generally add white space to help structure your files to make them easy for you to read.

And, you can insert comments to document for you or others what you're up to in a particular file. To use a comment, just type an ' {apostrophe} and everything after it in the line will be ignored when the file is read by ShopBot. Details for the use of remarks and comments can be found under "REM" in the Reference section. Also note the very useful special case of a comment after a "PAUSE" that provides a quick message to the tool user.

To really appreciate the possibilities for file layout, take a look at the sample files included in your c:\SbParts folder that was installed with your software. These sample files illustrate various formatting and organizational techniques. The sample files are the files that start with the name 'S_' (for 'Sample').

Note that when using ShopBot Commands, if any parameter follows the Command, the two letters of the Command are **always** followed by either a comma or at least one space. See the following example (note use of comments):

```
M2, 3, 4.2    'ok ... using comma as Command separator
M2 3, 4.2    'ok ... using space as Command separator
M23, 4.2     'WRONG ... no separator
M2,, 4.2     'This commands moves Y to 4.2 because first
              comma is the separator (no X value)
M2 , 4.2     'This commands moves Y to 4.2 because space
              is the separator (no X value)
M2, 4.2     'This commands moves X to 4.2 because comma
              is the separator (thus no Y value)
M2 4.2      'This commands also moves X to 4.2 (no Y value)
```

During execution of a Part File, information about the files being opened and run are displayed in the main window. This display identifies which line in each file is currently being read or executed.

Introducing Part File Programming Statements

Beyond the execution ShopBot Commands there is an additional set of commands that can be used from within Part Files. We call these commands Program Statements. These Programming Statements constitute additional instructions that provide numerous capabilities and functions from within Part Files. They add flexibility to what you can do in files. These additional control statements, when used in ShopBot Part files, amount to a mini programming language and they are the primary topic of this **Handbook**.

The Program Statements that ShopBot uses are modeled after similar functions in the BASIC computer programming language. Some people already will be familiar with the uses of such statements and what you can do with them. Others will find the following discussion full of new stuff. In general, though, Program Statements just give you enhanced ways to give your tool a list of instructions. If you get involved in more serious programming of your ShopBot using the Part File Language, one thing to keep in mind about this kind of programming is that there are usually many ways to accomplish the same thing ... you'll probably be able to find one approach or another that works well for you.

Nesting or Embedding Part Files

One Part File can call up and execute another Part File. You simply use the [**FP**] Command in the first file in order to start the second file. When the second file finishes execution, action returns to the next line of the first file. This process of starting one file with another is referred to as nesting or embedding files. Typically, you might have a 'master' Part File that calls up various components that will be cut from the piece of material currently on your table. This master Part File can position the tool for cutting the components and thus manage the layout of the project. An example of placing a single part in multiple locations is described in the sample code here. You use the 'offset' function in the [**FP**] Command to cause the file to be cut from the location that the master file moves the tool to. Thus you can use nesting to place different parts of a project in their correct location.

```
'Start of master Part File ALLPARTS.SBP
M2, 10, 4           'move to location to start first part
  FP,  MYPART.SBP , , , , 2   'execute with 2D offset
M2, 20, 4           'move to start of second part
  FP,  MYPART.SBP , , , , 2   'execute with 2D offset
M2, 30, 4           '... and so on
  FP,  MYPART.SBP , , , , 2
M2, 40, 4
  FP,  MYPART.SBP , , , , 2
```

You can repeat nesting of files inside other files multiple times up to 8 levels deep. This means you can call up a part, and then within each part call up say a

repeated procedure such as a drilling and countersinking routine. This process of nesting is an example of what in programming would be called a sub-routine, sub-program, or procedure. Sub-routines are useful for organization and efficiency, and more generally provide a way to 'structure' your Part File work. You can have subroutines that you use with a variety of master Part Files. As an example from our shop, we have a drilling/countersinking routine that we use for many different purposes including making the table-tops for ShopBots. We have saved this counter-sinking routine, which is for just one hole, as a Part File. Then we call it as a sub-routine within any file in which we need to do drilling or countersinking.

Math, User Variables (&), and System Variables(%)

When creating a PART File, you can use mathematical expressions, including variables to specify parameters just as you can when entering parameters at the main menu. Most mathematical functions are supported. When in doubt, try them out with the ShopBot Calculator first. Variables can also be used in Programming Statements.

Using Variables

When creating a part file, it is sometimes useful to indicate a value as a 'variable' rather than as a fixed number. Using a 'variable' just means that a word or symbol is used temporarily stand-in for a number. For example, let's say you are making a Part File to cut out a fancy grating for the front of a series of cabinets. Sometimes you will be cutting this grating in 3/8" material and sometimes in 1/2" material. There are dozens of Z plunges in the file and you know you don't want to have to rewrite all these plunge values for different depths of material. One way to deal with this problem is to use a variable for the Z depth for the cut. Variables in ShopBot are always designated with the use of '&' (ampersand) as the first character and are created like: `&Zdepth = -0.52` . Creating a variable like this at the start of a file means that we have assigned the value of -0.52 to every instance of the `&Zdepth` variable that follows in the file. So we would just write our Z plunges like: `MZ, &Zdepth` . Then if we want to change the plunge depth in a file, even though there may be numerous plunges, we only need to change the one variable definition at the top of the file.

You also can define new user variables with mathematical expressions.

```
&MyVariable = 5.0
```

or

```
&MyNewVariable = &MyVariable * 2.31-2
```

Remember that variable names must have an & as the first character. The case of the letters in the variable does not matter during ShopBot processing (internally all variables are converted to upper case as they are processed). This means you can put any of the lettering in a variable in upper or lower case, whatever helps it all make sense to you. Letters and underscores are fine, but don't use special characters in variable names that might be confused with math symbols or Windows folder designations (e.g. no `"\"` or `"*"` or `":"`).

Variables in programming are often defined in terms of specific 'types' based on what kind of numbers or character they are. For example, *integer* variables, *string* variables, *floating point* variables, etc... However ShopBot variables are a generic variable type that is handled behind the scenes as a string variable or floating point number depending on what is appropriate in

the situation (similar to 'variant' data type for those familiar with Visual Basic). This means you will not need to worry about variable typing.

When mathematical operations are carried out, if a variable can be made into a number it is handled as a single precision floating point (7-8 digits precision). If you use variables for counters and test for the end of the loop with an IF test, then test with an ">" or "<" rather than an "=" because the variable may not be stored as precisely the number you expect (i.e. a number that you expect to be "1" might actually be "1.000001").

Strings variables are not case sensitive. All characters in a string variable become capitals internally. Strings can be entered with or without quotation marks. For example:

```
&MyString = test string      'string variable without quotes
```

```
&MyString = "test string"    'or, string variable with quotes
```

Strings can be used for display, writing to files and some special functions. In most cases, defining a string in either manner will work fine. You can combine (or "concatenate") strings using the & character, so if:

```
&firstname = Bill
```

Then if you have a line that reads:

```
&newstring = "My name is " & &firstname
```

the value of &newstring would be "My name is Bill"

Variables that are used in Part files are "persistent" and "global", meaning that they retain their value until that value is changed or the ShopBot software is shut down and are available to any Part File that is run. This can be very handy for occasions when you might want to use the same value for all the files in a session like possibly &Zup for your safe Z-axis position. This also means, though, that if you use a variable that has been used in a previous file, you will need to be sure that you know it's value. The safest way to do this is to define all variables at the beginning of a file by giving them a value, a process referred to as "initializing." This also conveniently documents for you the list of variables that are used in any particular file.

System Variables

In addition to user variables, you can access certain ShopBot System Variables. These are variables maintained by the software and related to tool setup and operation. These values are accessed by a "%" (percent sign) followed by a number in parentheses (e.g., %(5)) and can be inserted where any value or parameter could normally be inserted in a file. Any system variable that involves a distance value (like the X-axis location) will be in the current user units

Here's an example of getting the current Y location (in the current user units; inches or mm):

```
&Y_now = %(2)
```

ShopBot System Variables are frequently expanded and updated. The current System Variable list is provided at the end of this manual and updates can be found in the "Program Files\ShopBot\Developer Tools" folder after installing the ShopBot software.

Types of Program Statements

All ShopBot Commands can be used in a Part File. In addition, a number of Program Statements are available for your use in enhancing the functionality of Part Files. These programming statements are modeled after statements in the BASIC programming language. We plan to continually add BASIC-like functions and control statements to ShopBot's capabilities. Here we provide a brief description of current programming statements by category. The subsequent section provides a detailed alphabetical reference to programming commands.

Control/Flow Statements

These are statements that direct the action during the execution of a Part File, moving it from one place to another and stopping and starting it.

```
GOTO                >sends action elsewhere
GOSUB ... RETURN    >sends action to subroutine and
returns it
PAUSE               >pauses a file
END                 >ends a file
EXIT SHOPBOT        >ends a file and exits ShopBot
```

Conditional Branching Statements

These are statements that perform a logical test or check for an Input Switch event and then appropriately re-direct the action in the Part File if something has happened.

```
IF ... THEN         >logical IF test
ON INPUT            >handles detection of an Input Switch
                    change
```

Input/Output Statements

Control what is displayed or entered on the File Display line at the bottom of the the screen during the running of a Part File.

```
INPUT              >get input from user/keyboard
PRINT              >display a message or variable on the
                    screen
' (or) REM         >comment in file that can optionally
                    be displayed

MSGBOX             >display a Windows style message box
WARNING OFF        >turn off warning light display

PLAY               >play a sound or audio recording to
                    alert the Operator (or just for fun)
```

File Statements

Statements used to manage input and output to files that are opened within and during the action of a Part File.

```
OPEN ... CLOSE >open or close a specified file
INPUT #_ >get information from the file
WRITE #_ >put information in the file
```

Reference: Part File Programming Control Statements

For consistency and clarity, here we use upper case letters for Programming Statements and ShopBot Commands. However, in the actual processing of a Part File case is ignored. So in the files that you write, you can use upper or lower case letters, or any combination of them. In the following section, programming statements are in upper case letters, variable or parameters are in italics, and variable or parameter choices are separated by '|', and optional functions are in braces ('{'}s).

CLOSE {#(*openfile*number)}

Closes a file that has previously been opened with OPEN statement. If used without the (*#openfile*number) then ALL open files will be closed

Be tidy and close files after you are finished using them. It is particularly important when writing to a file to make sure all data is saved in event of a subsequent crash. All user files are automatically closed when Part File finishes.

Note: be sure to read about the OPEN statement further down this list

END

Terminates the execution of an individual Part File. It is sometimes useful in a file to place an explicit END statement to stop processing and make certain that processing can not move into a section of the Part file such as a section that should only be entered with a GOTO or GOSUB. The use of an END statement can also add clarity to understanding the flow of a program. However, since Part Files are sequential and execute from the first line to the last (unless redirected by a GOSUB or GOTO statement), an END statement is not required in a Part File as the action of the file will terminate on execution of the last statement. Note that an END statement in a nested Part File that is being used as a sub-routine will end the execution of that File and return control to the Command Prompt or the higher level Part File that started or called it.

ENDALL

Terminates execution of all Part Files. In the case of nested files, ENDALL will all files above and below the file in which the ENDALL statement occurs.

EXIT SHOPBOT

This statement will produce the termination of the Part File and the immediate exit from the ShopBot software. This statement is useful for the situation where the ShopBot software has been started by another program and has started a file in ShopBot by passing the Part File name when SB.EXE was called (see Developer's section below for details). When the ShopBot task is completed and the EXIT SHOPBOT Program Command encountered, the computer will leave the ShopBot Program and return to action in the calling program.

GOSUB *label*

This statement is used to invoke a subroutine and causes execution of the file to shift to the first line under the designated *label* where *label*: is the name of a block of instruction and is positioned as the first line. Note that where the *label* is actually used as the name of the block, it is followed by a colon, but it is not followed by a colon in the GOSUB statement. The block of code is terminated by the RETURN instruction, which causes execution to return to the line immediately below the GOSUB statement.

GOTO *label*

Causes execution of the file to shift to the first line under the designated *label* where *label*: is the name for a position elsewhere in the file. Note that where the *label* is actually used as the name of the new position it is followed by a colon, but not in the GOSUB statement.

IF statement to evaluate THEN ShopBot command | variable assignment | GOTO label

This control statement does a logical test of a statement, and, if the statement is "true," the statement executes a ShopBot command, assigns a value to a variable, or branches to a label--whichever action is specified after THEN. Statements to evaluate are in the form of "IF &Count = 20 THEN" Moreover, the logical comparisons [= , > , < , AND, OR, NOT] all can be used in the statement along with user variables, system variables, or computed values. You can test your statements using the ShopBot Calculator [UU] or F10 (-1 = "true"; 0 = "false"). You can then have various "action" choices for the true case, including branching to another section of the file. If you have any programming experience, this statement will be familiar to you. If not, try playing around with it to get a feel for how it allows you to control the flow of program execution. Also have a look how IF's are used in various sample files (the ones whose filenames begin with S_ in the ShopBot Folder).

Note: An IF statement may contain only one logical test. For example, you can not test IF &varA => &varB THEN ... rather, you must check

for one logical condition at a time. For example, IF &varA > &varB THEN ...

See the section above on string variables for an explanation of how explicit quotes must be used in IF tests of strings.

INPUT "Optional Text Message" &variablename { ,&variablename, &variablename, etc... up to 10 variables}

Gets input from a user and assigns it to a variable. The "Optional Text Message" is any text in quotes (without commas) and can be used to let the operator know what sort of information is being requested. &variablename is a ShopBot user variable that the file will store that input in, beginning with an ampersand("&"). The INPUT statement calls up a message box for user input that provides OK and Cancel buttons. OK places the value that the user typed in the text box into the variable, and Cancel removes the box and ends the program. If the variable exists, its value is overwritten; if it does not exist, it is created. The user must separate input values with commas if more than one value is being input. If the text box is empty and only one variable is expected, it treats that variable as an empty string and continues. If more than one variable is expected, one or more missing variables is treated as an error, and an empty box is handled as a Cancel. In both cases the Program is terminated.

There are two special cases for INPUT:

1. Including a variable in the Text Message of INPUT. If it is desirable, a ShopBot User Variable can be used to display dynamic information within the Text Message display. This is accomplished by putting a **semicolon** immediately after the end of the Text Message. The INPUT statement will have the form:

```
INPUT "You Entered - " & &variable & " Is this Correct? Y or N"; &test1
```

In this case the returned variable will be the response to the Y or N question. The &variable needs to be defined prior to the INPUT call to work correctly.

2. Including a default value in the INPUT Box. A default value can be placed in the input box by defining it before the INPUT call and using colon in front of the default variable name, which will be the first variable in the list, occurring just before the of variables to be gotten from the user. This INPUT statement takes the form:

```
INPUT "Displaying default value - ": &default, &newvar
```

In both cases, as with the standard INPUT, there can be up to 10 user variables in the list of those being collected. The two special cases can be combined, using the semicolon call of the dynamic variable processing first.

INPUT #(openfilename), &variablename { ,&variablename, &variablename, etc }

Statement for reading data from a file that has been opened from within thShopBot Part file with the OPEN for INPUT statement and having been assigned an *openfilename*. Variables are assigned as they are assigned with the INPUT Command. When all variables have been read, an open file should be closed with the CLOSE statement.

label:

A label provides a marker in the file that serves as both a title of a section and, more often, an entry point for a GOTO or GOSUB instruction. Labels can be any word or single group of characters, but labels must exist on a line all by themselves and have a colon (":") as their last character. See GOTO and GOSUB above.

MSGBOX (*body text*, *button type*, *title text*)

The MSGBOX command creates a Windows style message box that can be customized, and places the value of the button that was clicked into a user variable named *&msganswer*.

The body text is the text that appears in the main part of the message box and can be used to ask the User questions or give them information without them having to type in an answer the way they would with the INPUT statement. You can also combine strings with variables, either system or user, to make a more descriptive message, using the rules for combining strings in the "Strings" section above .

The only restriction for the body text is that I can not contain a comma.

The second message box parameter is the Button type and there are lots of options to customize the look and action of the messagebox. The defaults include...

Style parameter	Display
0 (or "OKOnly")	OK Button only .
1 (or "OKCancel")	OK and Cancel buttons
2 (or "AbortRetryIgnore")	Abort, Retry, and Ignore
3 (or "YesNoCancel")	Yes, No, and Cancel buttons
4 (or "YesNo")	Yes and No buttons
5 (or "RetryCancel")	Retry and Cancel buttons.
16 (or "Critical")	Critical Message icon with OK button only.
32 (or "Question")	Warning Question icon with OK button only.
48 (or "Exclamation")	Warning Message icon with OK button only.
64 (or "Information")	Information Message icon with OK button only.
256	Second button is default
512	Third button is default
768	Fourth button is default.

These numbers (or the text options for the first 10) can be typed in as the second parameter in the command just the way they are to get the default message boxes. If you look at the table above you'll see that the parameters separate nicely into three groups. The first 6 define the button choices that the operator will see, the second 4 define the icon that will appear in the box to show the level of urgency, and the last 3 define which button has focus. Adding the values of items from each group will let you create custom boxes.

For instance if you type "276" as the second parameter in the command (4+16+256), it will create a message box with Yes and No buttons...a value of 4, a "critical" icon...a value of 16, and the second button (No) selected by default...a value of 256.. Any incorrect value will default to 0, a plain OK Only box.

The third parameter is the text that appears in the title block at the top of the message box. Keep this short and sweet...there's not much room...and don't include any ";" in the text.

So if you put this command in your file...

```
MSGBOX ( Do you want to open the keypad?, YesNo, Start Keypad?)
```

...then when the file got to that line this box would appear...

When a button is clicked by the User, the value of that button is saved



If the command was..

```
MSGBOX (You can not cut plywood that thick with that size
        bit!,16,Thickness problem!)
```

...the message box would look like this..



When a button is clicked in the message box , the “value” of that button is saved as a string in a variable named “&msganswer” so that you can use it in an IF test to act on the operator’s answer. The possible values for that variable are...

- OK
- Cancel
- Abort
- Retry
- Ignore
- Yes
- No

...depending on the buttons that were available in the message box.

ON INPUT(*switch# , 0 | 1 | 2 | 3*) *ShopBot command | variable assignment | GOTO label*

This control statement establishes an event handling process defining what is to be done if action occurs for any Input Switch. Place in code before location where input detection is desired. Define by Input Switch number and switch condition whose occurrence is to be detected; where 1 = ON and 0 = OFF. Then provide simple instruction such as GOTO (a label) or setting a variable. To explicitly clear the interrupt handling condition and instruction, use the same defining condition for the switch, but leave the instructions blank.

After a switch interrupt has occurred, the interrupt is automatically cleared. This prevents any unexpected reactivation (e.g. bounce) after the interrupt.

To use the ON INPUT interrupt again, it needs to be reset with the same ON INPUT command.

Any change at any of the switches causes the ShopBot software to look for instructions for handling that particular change or event. (The state for all 8 Input and Output Switches is displayed in "lights" below the location readouts in the main red Location panel.) For example, if Input Switch #2 had just been activated, and if you had set "ON INPUT(2,1) VA, 0.0" as the desired response to this event (for example, the closure of a limit switch), then your X axis location would be reset to 0 when the switch was triggered. In other words, ON INPUT() sets up the instructions for what to do "in the event of " a Switch closure (or opening). *Switch #* is the number of the input switch that the instructions apply to. *1 or 0* indicates whether the instructions are for the event of coming on (1) or the event of going off (0).

The instruction identifying the desired response can consist of a ShopBot command, the assigning of a value to a variable, or branching to a label -- any one, but only one, of these actions. (The actions are similar to those possible after an IF ... THEN statement.) The ON INPUT() statement typically should be placed near the beginning of your file or before a procedure segment. ON INPUT() itself does not cause execution of the instruction to occur; it only establishes what to do if an Input Switch event occurs. That is, technically it sets up 'event handling'. Note, too, that the action is not triggered unless a change in condition actually happens. The event handling that you specify in an ON INPUT() statement is active only for the duration of the file execution. When control is returned to the main menu, the specified ON INPUT() action is no longer active (although activation of the switches still will be indicated on the panel if the switches are enabled).

If a move is being executed at the time of the occurrence of the ON INPUT interrupt, motion will stop instantly (see the zzero.sbp routine for a good example). If however, you would like a gradual, ramped stop, then use the following switch conditions numbers, 3 = detect Input Switch coming ON and executed a ramped stop before continuing with instruction, 2 = detect Input Switch going OFF and execute a ramped stop before continuing with instruction.

OPEN "path&filename" FOR {INPUT/OUTPUT/APPEND} AS #{number}

ShopBot Files are sequential access text files and use standard BASIC syntax to open a user file for various purposes. If your ShopBot file is going to write to the file you should open it FOR OUTPUT, if you are going to be reading data from it you should open it FOR INPUT, and if you want to add information to an existing file without deleting its contents you should open it FOR APPEND. If you open a file FOR OUTPUT and the file doesn't already exist then the ShopBot will create it, but if the file does exist the data in it will be overwritten. If you need to preserve the data in the file then you may want to either use another name or add your data to the end using the FOR APPEND option.

The file number is the number you are assigning the file you are writing to and is used by the WRITE and PRINT statements to identify which file to work with. Up to 9 open files permitted (number = 1 to 9).

The path&filename can be a variable or specified file name. For path&filename, the path can be relative as in...

```
OPEN "myfile.sbp" FOR OUTPUT as #1
```

...which will look for "myfile.sbp" in the current part file folder, or ...

```
OPEN "myfolder\myfile.sbp" FOR OUTPUT as #1
```

...which will look in your current part file folder for a subfolder named "myfolder" and then look there for "myfile.sbp". You can also use the full path to the file like...

```
OPEN "C:\myfolder\myfile.sbp" FOR OUTPUT as #1
```

It's important to remember that any file that you've opened with the OPEN statement should be closed when you're finished using the CLOSE statement

PAUSE {seconds; or, UNTIL Input switch#, state}

Creates a pause in the execution of a Part File. The PAUSE Programming Instruction is one function that you will find useful even if you don't get deeply involved with the programming language features of ShopBot. Using this Programming Instruction allows you to put a brief pause or a stop in the execution of a Part File.

PAUSE #. The number after the PAUSE is the duration of the pause in Seconds and should be accurate to about 1/100th of a second. For example, programming a **PAUSE 3** will cause a 3 second pause in the action and then the file execution will continue. If you put a comment line with an apostrophe just before the pause, that line will be displayed during the time the PAUSE is in effect.

PAUSE. A PAUSE statement on a line without a number after it will cause the execution of the file to stop indefinitely and a message box will appear with a choice of two keys ... "OK" to continue, or "Cancel" to end the Part File. If the previous line is NOT a comment then the box will display "Continue?".



If the full line preceding the pause is a comment (') then this comment line will be displayed in the box. For example, if you want to make sure that a vacuum holddown pump is turned on before the file starts cutting you might add the following to your file...

```
' Is the Vacuum Pump on? .... (Hit OK or ENTER when Ready to  
Continue)  
PAUSE
```

When your program gets to this place in the file, a message will appear with your message to check the vacuum pump, and your ShopBot would stop until the OK button is clicked or the ENTER key is hit.



If you just want to notify the operator about something in the file, though, without the operator having to act on it, you might :

```
'Finished the first Cut-Out, Moving on to the Second  
PAUSE 5
```

Here the tool would stop for a 5 seconds while displaying the message, then automatically continue to the next task.

PAUSE UNTIL works just like PAUSE (without a time) except that an Input Switch can be used as the equivalent of the OK input button. For example, programming **PAUSE UNTIL 5, 1** will PAUSE until the OK button is entered or the #5 Input Switch is closed. Any message in a preceding comment line will be displayed as with the simple **PAUSE** instruction. Use a 0 to detect an Input Switch opening, or 1 to detect a Switch closing.

By default, when motion is initiated after an Input Switch PAUSE, there will be a warning and short wait, just like at the beginning of a file to provide a safety interval. This can be eliminated by turning the wait off with **[SW]** before the call to function, then restoring it. There is a sample file in the SbParts folder (sample_for_switch.sbp) that illustrates.

PLAY *path&wavfilename*

Plays a .wav sound file. The statement will start the execution of the sound and immediately proceed to the next instruction...if you do not want to proceed until after the sound finishes, add an appropriate length PAUSE after

the PLAY statement. In the dev folder under ShopBot there is a helpful program called "wavelength.exe" that will help select and time your sound files.

```
PLAY path&wavfilename  
PAUSE time(sec)
```

For instance, if you wanted to play the Windows "Tada" sound somewhere in your file and wanted the file to continue running while the sound was playing you would add...

```
PLAY C:\WINDOWS\Media\tada.wav
```

...to the file. If you wanted the file to stop executing while the sound played you would add..

```
PLAY C:\WINDOWS\Media\tada.wav  
PAUSE 2
```

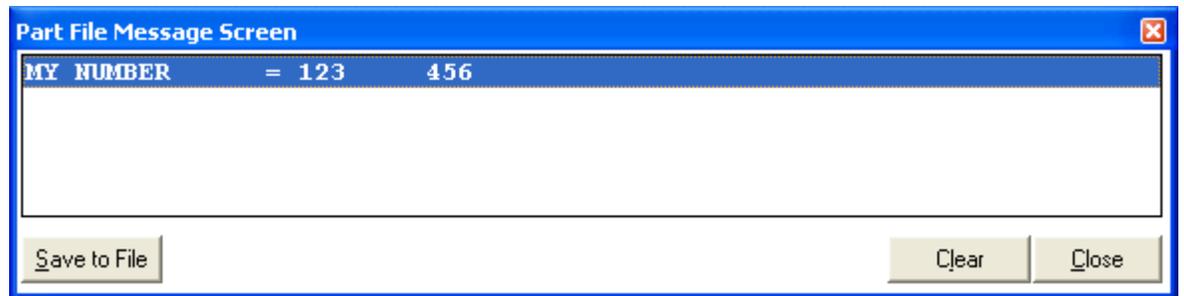
PRINT {variable or quote delimited text, variable or quote delimited text, etc...}

The PRINT Statement will display a variable or text and forces the PartFile Message Window to Display. If the first separator is a comma, this will be the default item separator and will cause each item in the line to be separated by a tab (5 spaces). A semicolon as the last character will suppress the line feed, but if a comma is used first a semicolon in any other position will raise an error.

If the first separator is a semicolon, this will be the default item separator but will produce no separation in the displayed item unless the separation was added as spaces as part of the print string. A semicolon as the last character will suppress the line feed. A comma later on in the parameters will be treated as an item to display. For example:

```
&firstvar = "My number"  
PRINT &firstvar, " = 123,456 "
```

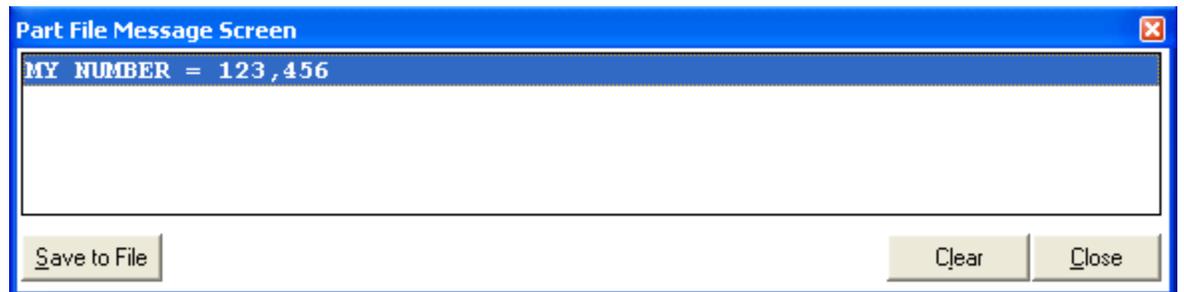
with a comma after "&firstvar" to separate the parameters will be split at the commas and look like this...



With a semicolon separating the parameters like this...

```
&firstvar = "My number"
PRINT &firstvar; " = 123,456 "
```

it will look like this...



If a variable is used in a PRINT statement that hasn't been given a value then the variable name will appear in it's place in the message screen.

The location and size of the message screen (measured in twips) can be specified using the VD command, either from the keyboard or within a part file. It can also be dragged to a new location and resized on the screen with the mouse and the new location and size will be remembered. If you don't want the Message Window to display you can turn it off by setting the "Show Comments" setting to 0 in the VD command..

REM or more easily just, '

Indicates a Remark or as we refer to it here, a Comment line. You can actually use either the letters REM or an apostrophe ('), but using the (') is preferred. The text in a comment is not treated as a command and is ignored during the running/cutting/execution of the file, even if the comment text contains one of the Shopbot Commands

It is very helpful when creating Part Files to make notes for yourself in the file so that when you look at it at some later time, what you've done will make sense to you. This is a process that programmers call 'documenting'. To make a note to yourself in a file, all you need to do is type in an apostrophe (

') and everything that follows it on that line will be ignored by the ShopBot software when the file is executed. You can put in the apostrophe and make a note to yourself about what a particular line, or portion of the file, is supposed to be doing. You can put the apostrophe at the beginning of the line and use the whole line for comments, or you can put it after a Command and make a short note to yourself. If you want to make several lines of comments, put an apostrophe at the beginning of each line. A full-line comment is displayed at the bottom of the screen when a file is executing, so this is an easy way to remind yourself of things while the file is operating (see the 'PAUSE' instruction to learn how to make this display remain visible for as long as you would like).

```
'This is an example of documenting a file
'You can use an apostrophe (or a REM) at start of a line
REM You can use an REM (or an apostrophe) at start of a line
'This file is to create a rectangle (written by me, 1/1/2000)
' ... none of the above effects the cutting of your file
' ... the line below is left blank for legibility

JZ, .5                                'this line pulls the bit up
J2, 10, 10                            'you can add a comment to a line
CR, 10, 5, , , , -.25, , , 1
    'You can indent lines or comments and add blank lines

'The End
```

RETURN

Causes Part File execution to return to the line that is immediately below the GOSUB Statement that initiated action in a sub-routine. If you have placed subroutines at the bottom of a Part File, you will want to put an END statement before them to make sure that execution of the Part File does not inadvertently flow into the sub-routine.

SHELL, "path and program and rest of command line (as single string variable or in quotes)", {display option}, {async or sync (use words ASYNC or SYNC)}

This Programming Statement that starts an external program. First parameter gives the file to start (including path), followed as part of the same string with any command line parameters/options. These parameters are those specified by the program being called. They can be compiled into the string as ShopBot variables using normal string concatenation.

The display options are: 0- hidden and focus hidden (don't use this unless you understand what you are doing; it's a good way to lose control of the computer); 1- normal focus and size (def); 2- minimized with focus; 3- maximized with focus; 4- normal size, focus stays on ShopBot; 6- minimized, focus stays on ShopBot.

ASYNC (def) or SYNC mode:

In normal ASYNC mode, the ShopBot part file will continue to execute after the outside file has been called

In SYNC mode, ShopBot Part File will stop execution while called program executes and control will not be returned to ShopBot until the outside program is closed.

Examples:

```
SHELL, "sedit.exe c:\sbparts\sample_shopbot_logo.sbp"
```

```
SHELL, "C:\windows\notepad.exe", 1, SYNC
```

```
&filepath = "c:\sbparts\sample_shopbot_logo.sbp"
```

```
&run_this = "sedit.exe" & &filepath
```

```
SHELL, &run_this, 1, ASYNC
```

(Calling a Virtual Tool with variable:)

```
&speed = 11500
```

```
TR, &speed
```

WARNING OFF

Turns off the flashing warning display when it would be distracting, as it might be during execution of sections of a file that involve no movement. Any movement command turns the warning display back on.

WRITE #(openfilename)

Statement for writing data to a text file that has been opened from within the ShopBot Part file using the OPEN FOR OUTPUT statement and having been assigned an *openfilename*. An open file should be closed with the CLOSE statement.

There are 2 options for the WRITE# statement...separating the items with commas or with semicolons. See PRINT statement for examples of how this works

WRITE #(openfilename), (data list, items separated by comma) ...

This is the standard way to use the Write statement. Separate each data item with a comma and they will be evaluated and separated by commas in the output file.

WRITE #(openfilename); (data list; items separated by semicolons)

... This is an alternative Write function in which all spacing characters are suppressed except those explicitly supplied in data list. This version of the Statement gives the most explicit control of punctuation in the output file

Notes: In both versions a semicolon as the last character will suppress the line feed in the file being written to so that subsequent data will be added to the same line.

So to generate **&myVar = 23.5** in the output file you would use:

```
&someVar = 23.5
Write #1; "&myVar = "; &someVar
```

Or if you wanted to generate: M2, 23.5, 23.5 you would use

```
&someVar = 23.5
Write #1; "M2, "; &someVar; ", "; &someVar
```

Thus, the Write #?; version of the Statement (with a semicolon";" after the file number) gives the most explicit control of punctuation in the output file. Alternatively, the last statement could have been produced more simply with:

```
&someVar = 23.5
Write #1, "M2", &someVar, &someVar
```

This means that you can not have a comma inside quotes if you're using a Write #?, Command (with a comma "," after the file number) but you can accomplish this using the more explicit controls of Write #?; (with a semicolon ";" after the file number) ...

(The variations in the WRITE and PRINT statement may seem confusing because the difference between the ";" and "," versions are not immediately obvious...they look like the same command unless you know to look for them).

ADDITIONAL REGISTRY COMMANDS. The following ShopBot Programming Instructions are provided to interact with the registry:

SetUsrVal, *textvalue*

Sets user-determined variable. Can be any text (or number) value with the variable being saved to the registry as a string

- This value is not erased until done so by the user! (That is, it persists even when ShopBot is closed and the computer turned off)
- The saved value can be text or a number
- When GetUsrVal is used to read from the value back, the registry string is placed in the designated ShopBot variable as a variant (string or number)
- The current value of uValue can be inspected in ShopBot using [UD]

SetUsrValClrd, *textvalue*

As for SetUsrVal, sets a user-determined variable. But this value is automatically cleared at the end/beginning of each file. Value can be any text (or number) value with the variable being saved to the registry as a string

- The saved value can be text or a number
- When GetUsrValClrd is used to read from the value back, the registry string is placed in the designated ShopBot variable as a variant (string or number)
- The automatic clearing provides a "new file" flag that can be set to "used" or other appropriate indication that some function or process has been already been done once in a file. Value can only be inspected while file is running.

SetSpindleStatus, *0-1*

Sets ShopBot flag for Spindle Start-Up Dialog. If set to 0, then indicates spindle is considered *not active* and Dialog should display. If 1, then spindle is considered *active* and spindle will restart without dialog.

GetUsrVal, *&any_ShopBotVariable_name*

Retrieves previously set UsrVal to a ShopBot Variable. See above.

GetUsrValClrd, *&any_ShopBotVariable_name*

Retrieves previously set UsrValClrd to a ShopBot Variable. See above.

GetUsrPath, *&any_ShopBotVariable_name*

Retrieves path to User Data Folder (defined by Windows).

GetAppPath, &any_ShopBotVariable_name

Retrieves application path for ShopBot program.

GetPartFileName, &any_ShopBotVariable_name

Retrieves most recent Part File read. This will bring up name of Part File from last session if no others have yet been run.

GetSpindleStatus, &any_ShopBotVariable_name

Retrieve the current SpindleStatus. 0 indicates the spindle is considered *not active*. Next start will evoke the spindle-start dialog. 1 indicates spindle is considered *active*. See above.

There is a more extensive set of Registry Variables available to developers who are using their own software to directly read and save to the registry or command line system. See section below on "Windows Registry Interface".

Additional Part File Programming Considerations

Movement Blocks

When a Part File is read by your ShopBot Software, the program's first priority is to process all the tool movement commands in such a way that they can be most efficiently and smoothly executed. To make execution smooth, as the software reads through the file it accumulates tool movement commands and stores them into memory in lists in an efficient pre-processed form. We call this stored list of moves a 'movement stack'. At the time the movement stack is being stored, the moves are also analyzed to determine where acceleration and deceleration ramps should be placed, and how other features such as 'tabbing' should be applied. Establishing the movement stack is always the first priority of the software as files are read.

In most cases, arranging your file into movement stacks is completely handled by ShopBot, so you won't need to think about it. However, in certain situations it can be helpful to take control of how movement blocks are handled so we'll explain this process here.

As a Part File is being read, when the first tool move is encountered (e.g. an M2) a movement block is created. The software will then continue to read lines accumulating movement instructions. No movement will be executed until the whole stack has been read.

The end of the stack is determined by:

1. Reaching an Command that is not a Move, Cut, or SwitchON Command, or a special case Insert.
2. Or, by reaching the end of stack memory.

After the stack is read, all the moves in it will execute. If the block is terminated by a non-Move Command, then that next command will be executed. Following it, when a new Move Command is encountered in the file, the next movement block will be acquired and executed (e.g. a J2, is a Jog and not a Move; the Jog ends the present stack and the stack is executed, then the Jog is executed as a single stack, point-to-point move).

If reaching the limit of available memory punctuated the reading of the Part File, then at the end of execution of the stack, the bit will be pulled to the 'Safe-Z' height (if this feature is active). After the pull-up, an additional stack will be read and then the bit will be re-inserted for cutting that stack of moves.

There are a couple of situation where you may wish to interrupt the automatic handling of movement blocks. The [**SC, 2**] Command can be used to explicitly put a stack-end and stack-restart [SC,2] at a specific location in a Part File.

The reason you may want to consider controlling stacks is because it may allow you to **prevent an interruption in cutting to load more of the file** from occurring at

an awkward location. By placing SC,2's at locations in a file before memory is used up by the current block, the block will be executed and a new one started. You can thus force the blocking to occur at the location in the file where it will not have an adverse effect on the cutting.

Structuring Your Files

Your efficiency in Part File creation, the speed of your de-bugging, and your ability to reuse your Programming work will be enhanced by adding structural elements to your programming.

First, layout your file with indents, labels and comments to make it as easy to read as possible and to break it into logical chunks. Second, use Sub-routines to carry out repetitive processes and try to write these subs in a generic way so that they can be used in other work. Third, break projects into multiple Part Files that are called or controlled by a master file. This will provide you with ready to use sub-Part Files for your other projects.

Adding Setup Lines to the Start of a File

After you have created a Part File that does some type of cutting or machining for you, you will want to consider putting a few setup lines at the beginning of the file, particularly if this is a file you will use regularly. Because all ShopBot Commands can be used in a Part File, you can put all the instructions that you might want to give to configure your tool at the start of the file. For example, to make sure that your tool is optimally configured for cutting the project, you may want to place Commands that set the speeds, cutter size, or other features of ShopBot operation that you will always use with this file:

```
'Setup lines for this file
  VS, 1.25, .45, , 2.75, .80   'to set speeds
  VC, .25                     'always a .25 in cutter for this file
  SA                          'distances will be absolute
```

The "Header Writer" Virtual Tool in the ShopBot software makes it easy to add header and footer information to your part files.

Troubleshooting difficulties in Part Files

When you are creating a Part File, you will no doubt have to debug it to remove little errors that creep in a typos or, math or logic problems. Debugging can sometimes be more difficult than the actual creation of the file itself ... and it is certainly less fun. We have tried to provide helpful error messages, but it is sometimes difficult for the software to detect why a Part File program fails. Here are some suggestions to help make debugging a part file easier:

1. Break Part File projects into smaller Files so that you can test and correct them individually in a more manageable size.
2. Use the PRINT and PAUSE Statements to display values of variables at various places in your file to check to make sure that things are working as you think they should.
3. Step through Files using the [**FG**] Command so that you can identify trouble places.
4. Inspect the values of variables with [**UL**] to make sure they are what they should be.
5. You should get similar results with MOVE and PREVIEW Modes. So it's probably best to work out all complicated files in PREVIEW, before you run the tool.

Why We Have a ShopBot Part File Format ...

For those who are interested in such things, we would like to convey that the reason for having a specific Part File format is to make the process of producing, understanding, and maintaining your CNC files as easy as possible. There is an industrial standard for CNC files. This standard is referred to as 'G-code'. Though we don't promote it, the most recent versions of ShopBot software will run G-code as well as ShopBot Part file code. However, G-code is a language that was created in the days when cutting files were stored on punched tape and its format was optimized for brevity. These origins have left g-code unnecessarily arcane and difficult to understand when just inspecting a file. Moreover, it is a very loosely implemented standard with virtually every manufacturer using a different version and implementing it in idiosyncratic fashions.

We felt that a Command language with mnemonically meaningful commands which were the same Commands used for direct control of the tool offered the best option for new robotics tool users. These types of Commands would be easier to learn, easier to understand, and easier to use in everyday work, whether at the keyboard or creating Part Files.

In addition, we believed that if additional functionality were provided as 'programming' instructions in the format of the BASIC language, it would make them ready for use by anyone familiar with programming. This convention would also provide a clear set of prescriptions for the format in which functions should be added to the ShopBot control language. G-code has very limited programming capability and that which it has is awkward to use at best.

If you are already an experienced CNC programmer and prefer G-code or have software that generates G-code just go ahead and run it on ShopBot. Our converter [FC] also makes it possible to rapidly make a translation from a G-code file to a ShopBot formatted Part File so that you can have the file in this more useful format. You will probably find that when you inspect a Part File, it will make perfect sense to you because it functionally resembles a G-code file -- being primarily just a list of coordinates through which the tool is moved.

Most popular CAM programs and design programs that generate toolpaths now offer the option of outputting the more usable ShopBot Part File format. In addition, at least one CNC Control system can already use the more easy to work with ShopBot format. We expect that other CNC vendors will increasingly also adopt the .SBP language.

The ShopBot Part File language, officially known as OpenSBP®, is not a proprietary language. ShopBot has published it as an open standard and set up a system for expanding and developing the standard. We expect to see it increasingly integrated into user-oriented CNC systems.

ShopBot Log Files

By default, ShopBot software writes two different types of 'log' files. 1) A **Part File log** for each Part File that is run, which tracks the utilization of that specific file. 2) An **SbSystem log** that records the use of the ShopBot Control Software, the tool, and the files that are run.

Part File Logs. If writing of Part File logs is enabled [VD], each time a Part File is run whether in Move Mode or Preview, a record is made of that running and saved in the a file with the same name as the Part File but with a ".log" extension. Each time the file is run, the file is appended with the new information, which is contained in two text lines. Several information items about the file are recorded as well as whether the file was terminated before completion and what line the termination occurred.

SbSystem Log. If writing of the SbSystem log is enabled [VD], then when ShopBot is started, closed, or files run, entries are appended into the log. The log is stored by default as C:\Program Files\ShopBot\ShopBot 3\Bin\SbSys.log. This log file could be used to create a system for monitoring use of the ShopBot. We have several ideas for additional functionality in the system and dummy output values have been stubbed in, but are not yet implemented.

There are 3 types of lines in the SbSys file. A) Start-up or mode switch line; B) Part File line; C) Closing line.

- A) The Start-up line has either "NFP" or "StartFile" as the first data entry, where NFP indicates that the software was started normally and not by a command line call that passed a file [see this system in next section]. A command line call startup will be indicated by "StartFile".
- The second data item indicates whether the start occurred into CUT or PREVIEW mode
 - 5 data items follow that indicate the parameters passed if this was a command line start, otherwise these parameters are empty
 - Next items are TIME and DATE
 - Empty data item
 - Control Box Version if available
 - Software Version
 - Serial#, Operator, sys1,..., sys4 (FUTURE IMPLEMENTATION PLANNED)
 - ~##### (an internal check number used by ShopBot)
- B) The Part File line describes any Part File that is run by the software. It begins with full path and name of the Part File.
- CUT or PREVIEW Mode
 - Elapsed time
 - Next is TIME and DATE
 - Empty data item
 - Control Box Version if available
 - Software Version
 - Serial#, Operator, sys1,..., sys4 (FUTURE IMPLEMENTATION PLANNED)
 - ~##### (an internal check number used by ShopBot)

- C) The Closing line is records when the Control system is exited. The first data entry indicates the type of Close (NORMAL for a user close; or Type of close as set-up by a command line startup [see below])
- "Total Time" -- just a printed term
 - Elapsed time
 - Next is TIME and DATE
 - Empty data item
 - Control Box Version if available
 - Software Version
 - Serial#, Operator, sys1,..., sys4 (FUTURE IMPLEMENTATION PLANNED)
 - ~##### (an internal check number used by ShopBot)

Advanced Start-Up Information

From the beginning of ShopBot, we have sought to make our product open for use, improvement, and modification by others and to make it highly adaptable to a range of applications. We are working to support the use of our systems by outside developers in several ways: One priority is to support those who are interested in creating non-parametric and parametric design systems and cutting projects. For these developers, the functionality of the Part File language and built-in cutting routines is the reason for this manual. We will continue to expand the flexibility of our Part File language and the ease with which it can be adapted to creative projects.

A second priority has been to make the ShopBot Control Software open for use by outside programs. Along those lines, we have created two simple but powerful systems for allowing outside programs to run and manage the ShopBot software.

Windows Command Line Options...Controlling the ShopBot software from outside programs (a)

A ShopBot can be quite effectively started and run from other software (e.g. drawing programs, CAM programs, custom interfaces). As such, the actual operation of the ShopBot system can be almost transparent to the operator. Many companies have created custom interfaces to ShopBot that allow an operator to interact with a proprietary system that then runs the ShopBot when tool motion or action is needed.

The system works using the option for a "Command Line" startup of the ShopBot software (Sb3.exe). A number of startup parameters help optimize the use of ShopBot software, while the specifics of the required action are passed in a file(s).

Command line format:

SB3.EXE path&filename, Port, OpenMethod, timing, CloseMethod, DisplayMethod, Offset

Parameters:

- Path&Filename (can be sent with or without " "s)

- Port - (1 – 16) optional
- OpenMethod -
 - 1 (default) = MOVE/CUT; with FP Fill-In Opened set to this file
 - 2 = MOVE/PREVIEW; with FP Fill-In Opened set to this file
 - 3 = (currently same as #2)
 - 4 = MOVE/CUT; no stop for Fill-In
 - 5 = MOVE/PREVIEW; no stop for Fill-In
 - 6 = Only make file current Part File
- Timing (not used in Windows software)
- CloseMethod
 - 0 (default) = No close after running passed file
 - 1 = Close after file, but with confirm question
 - 2 = Immediate close and exit from ShopBot at end of file
- DisplayMethod
 - 0 (default) = whatever is current in shopbot.ini
 - 1 = Force console on
 - 2 = Force console off
 - 9 = Force console to always be off (OEM version, no use of SB Command Keys, must read file)
- Offset (Only applies if MOVE/CUT or PREVIEW with no stop for Fill-In)
 - 0 = no offset (default)
 - 1 = Offset 3D
 - 2 = Offset 2D

More information and examples can be found in the SBStarter folder in the Developer Tools folder in the ShopBot install.

Windows Registry Interface...Controlling the ShopBot software from outside programs (b)

For Developers we have provided a Windows Registry-based system for communicating between an outside program and ShopBot.

STATUS FLAGS indicating ShopBot Status as binary coded info in single decimal byte:

ORDER(&decimal value) [starting with lsd] FileRunning (1), PreviewMode(2), KeyPadOpen(4), PauseinFile(8), StopHit(16), StackRunning(32)

READ REGISTRY "ShopBot", "UserData", "Status", *Single byte as Text*
 -All bits cleared to "0" when ShopBot software starts

INPUT SWITCH FLAGS indicating Input Switch Condition [*input switch 1 through 8; binary coded in single decimal byte*]

READ REGISTRY "ShopBot", "UserData", "InputSwitches", *Single byte as Text*

OUTPUT SWITCH FLAGS indicating Output Switch Condition [*output switch 1 through 8; binary coded in single decimal byte*]

READ REGISTRY "ShopBot", "UserData", "OutPutSwitches", *Single byte as Text*

AND using the Registry, **a Standard ShopBot Command can be passed** to ShopBot from another program. The Command will be read and executed as soon as ShopBot finishes executing any running PartFiles, and any Commands already sent via a Command Line Pass (Method a, above).

The Command is PASSED FROM THE REGISTRY by setting:

- "ShopBot", "UserData", "uCommand", *command*
- Where *command* is a two letter ShopBot Command followed by normally formatted ShopBot parameter settings.
 - The command is erased when it is read by ShopBot, and is always initially cleared when the software starts.

Two additional pieces of information are also available from the registry:

- "ShopBot", "AppData", "uAppPath", *fullpath*
- Provides the full path to the folder containing the current ShopBot.ini and Problem.log files.
- "ShopBot", "UserData", "uUsrPath", *fullpath*
- Provides the full path to the folder containing the current S3.exe file.

Virtual Tool System

ShopBot Software provides a system for installing Add-In Tools. These are Tools that have been developed as standalone programs that use the Command Line option system to control ShopBot. But in the case, the programs are additionally made available as added Virtual Tools that will show up, and can be started by, the drop-down menus in ShopBot. They can also be called from within Part Files by the using the designated two letter commands. Optionally, a command line string can be passed to the Tool by adding it as a parameter to the call from the Part File.

The "c:\Program Files\ShopBot\Virtual Tools\VirtualTools.ini" file provides examples of how to set up a new add-in Virtual Tool that will be displayed and used in the ShopBot software. Once an add-in is installed and working properly, it can be called from a Part File like this:

[*two letter command*], *parameter(s)* (for the program)

Example (assuming the shortcut letter for the Tool is "C"):

TC, 1500, small, 2

ShopBot variables can be substituted for literal parameters, as:

TC, &some_value

System Variables [use as %(sys var#)]

SYS VAR #	VAR	units
1	Location X	userUnits
2	Location Y	userUnits
3	Location Z	userUnits
4	Location a	userUnits
5	Location b	userUnits
6	Base Coord X	userUnits
7	Base Coord Y	userUnits
8	Base CoordZ	userUnits
9	Base Coord a	userUnits
10	Base Coord b	userUnits
11	Min Table Limit X	userUnits
12	Max Table Limit X	userUnits
13	Min Table Limit Y	userUnits
14	Max Table Limit Y	userUnits
15	Min Table Limit Z	userUnits
16	Max Table Limit Z	userUnits
17	Min Table Limit a	userUnits
18	Max Table Limit a	userUnits
19	Min Table Limit b	userUnits
20	Max Table Limit b	userUnits
21	DISTANCE	0-1
22	MODE	0-1
23	CUTTER Dia	userUnits
24	SW Delay	sec
25	UNIT	0-1
26		
27	Number of Axes	3-5
28	Safe_Z	userUnits
29	Safe_A	userUnits
30	Number of Axes to Ck Lim	1-5
31	WUx	single
32	WUy	single
33	WUz	single
34	WUa	single
35	WUb	single
36	Driver Channel 1	XYZA or B
37	Driver Channel 2	XYZA or B
38	Driver Channel 3	XYZA or B
39	Driver Channel 4	XYZA or B
40	Driver Channel 5	XYZA or B
41	Tabbing Dist UP	single
42	Tabbing Dist Next	single
43	Tabbing Size	single
44	Tabbing Lead Size	single

45		
46	Inp #3 Mode	0 - 1
47	Inp #4 Mode	0 - 1
48	Software Version	string
49	Firmware Version	integer
50		
51	Inp #1	0-1
52	Inp #2	0-1
53	Inp #3	0-1
54	Inp #4	0-1
55	Inp #5	0-1
56	Inp #6	0-1
57	Inp #7	0-1
58	Inp #8	0-1
59		
60	Current File Name	string
61	X Offset Current File	single
62	Y Offset Current File	single
63	Z Offset Current File	single
64		
65		
66	X Proportion Current File	single
67	Y Proportion Current File	single
68	Z Proportion Current File	single
69		
70		
71	MoveSpeed X	userUnits
72	MoveSpeed Y	userUnits
73	MoveSpeed Z	userUnits
74	MoveSpeed a	userUnits
75	MoveSpeed b	userUnits
76	JogSpeed X	userUnits
77	JogSpeed Y	userUnits
78	JogSpeed Z	userUnits
79	JogSpeed a	userUnits
80	JogSpeed b	userUnits
81	MoveRamp X	userUnits
82	MoveRamp Y	userUnits
83	MoveRamp Z	userUnits
84	MoveRamp a	userUnits
85	MoveRamp b	userUnits
86	JogRamp X	userUnits
87	JogRamp Y	userUnits
88	JogRamp Z	userUnits
89	JogRamp a	userUnits
90	JogRamp b	userUnits
91	Mode Input #1	0-4
92	Mode Input #2	0-4
93	Mode Input #3	0-4
94	Mode Input #4	0-4

95	Mode Input #5	0-4
96	Mode Input #6	0-4
97	Mode Input #7	0-4
98	Mode Input #8	0-4
99	Torch Height	0-1
100	Full Output Byte	byte
101	status Out #1	0-1
102	status Out #2	0-1
103	status Out #3	0-1
104	status Out #4	0-1
105	status Out #5	0-1
106	status Out #6	0-1
107	status Out #7	0-1
108	status Out #8	0-1
109	Mode # 4	-1 to -4
110		
111	Mode Output #1	0-5
112	Mode Output #2	0-5
113	Mode Output #3	0-5
114	Mode Output #4	0-5
115	Mode Output #5	0-5
116	Mode Output #6	0-5
117	Mode Output #7	0-5
118	Mode Output #8	0-5
119	Mode Output #9	0-5
120	Mode Output #10	0-5
121	Mode Output #11	0-5
122	Mode Output #12	0-5
123	IndexerON	0-1
124	Analog Input #1	0-9
125	Analog Input #2	0-9
126	Inp #9	0-1
127	Inp #10	0-1
128	Inp #11	0-1
129	Inp #12	0-1
130	Elapsed Time	long
131	Mode Input #9	0-4
132	Mode Input #10	0-4
133	Mode Input #11	0-4
134	Mode Input #12	0-4
135	Upper Output Byte	byte
136	status Out #9	0-1
137	status Out #10	0-1
138	status Out #11	0-1
139	status Out #12	0-1
140		
141	Z/A Trigger ON Thresh	dist
142	Z/A Trigger OFF Thresh	dist
143		
144	Location c	userUnits

145	Base Coord c	userUnits
146	Date	string
147	Time	string
148		
149		
150	Step InterVal Divider	
151	Multiplier X	0-10
152	Multiplier Y	0-10
153	Multiplier Z	0-10
154	Multiplier A	0-10
155	Multiplier B	0-10
156	Multiplier C	0-10
157		
	Current uController	
158	Voltage	0-9
	Current uController	
159	Temp	Centegrade
160		
161	RPM Spindle	string
162		
163		
164		
165		
166		
167		
168		
169		
170	xxxx current max xxxx	