Debug Your Code

an overview of debugging methods

-
Alyce

Debug Your Code | Avoid Debugging | Data Validation | On Error Goto | Code Format | Code Construction | Compiler Reporting | Procedure View | Use the Helpfile | Debug in the MainWindow | Lite Debug | Breakpoints | Debugger Map | Debugger Help

Avoid Debugging

It's nearly impossible to write bug-free code. We all need to find and fix errors, at least occasionally. We can avoid some errors by writing code that validates data, handles errors, is well-structured and is easy to understand. If we still encounter bugs, there are numerous tools and methods to help us debug.

Data Validation

Always validate data before continuing program execution. Invalid data can create problems, whether it was input by a user or generated by the program. Here is a small example. Since it is impossible to divide by zero, the following code halts with an error message.

```
a = 0
b = 45
print b/a
```

It is easy to find the error in this small program, but it might be difficult to isolate in a large program. We can prevent problems by validating the numeric data before attempting to perform a division operation.

```
a = 0
b = 45
if a <> 0 then
    print b/a
else
    print "Cannot divide by zero."
end if
```

Validating string data is equally important and it is just as easy to do. Since Liberty BASIC is not able to

open a file for input if it doesn't exist, we need to check the return from a filedialog command. If the user cancels the dialog without selecting a file, the program will halt with an error when we try to open the file. The following program will halt with an error if the user cancels the filedialog.

```
filedialog "Open","*.*",filename$
open filename$ for input as #f
lenFile=lof(#f)
close #f
```

Before attempting to open the file, validate the filename$ returned by the filedialog.

```
filedialog "Open","*.*",filename$
if filename$="" then
    print "No file chosen."
else
    open filename$ for input as #f
    print "Length of file is ";lof(#f)
    close #f
end if
```

On Error Goto

We can use the "ON ERROR" statement to cause program execution to jump to an designated error handler routine when it encounters an error. When the program raises an error, the error number is placed in the special variable "Err" and the error description is in the special variable "Err$". Since these are variable names, they are case sensitive. Some errors do not have a number value associated with them, in which case Err is set to 0.

Some error numbers are listed below. The list is not comprehensive.

3 RETURN without GOSUB
4 Read past end of data
8 Branch label not found
9 Subscript out of range
11 Division by zero
53 OS Error: The system cannot find the file specified.
58 OS Error: Cannot create a file when that file already exists.
55 Error opening file
52 Bad file handle
62 Input past end of file

Program execution halts at the error handler routine. You must handle the errors there and tell the program what to do next. After you've handled the error, you can cause execution to continue where it was when the error was raised with the RESUME statement.

Each program will have different errors and methods for handling them. Here is a very small example. It uses the code that raises a "division by zero" error. This time, an error handler is put in place. When the error is raised, the error handler checks the value of Err to see if the value is 11. If it is, the variable "a" is assigned a value. The RESUME statement then allows execution to resume and the answer "9" is printed in the mainwindow. If Err has a different value, the program simply stops and waits for more user input.

```
on error goto [oops]
a = 0
b = 45
print b/a

wait

[oops]
if Err=11 then
    a=5
    resume
else
    wait
end if
wait
```

You might want to have a series of conditional statements as above, or a SELECT CASE block to evaluate and handle the errors likely to be generated by your program.

Code Format

Code that is easy to read is easier to debug. You may like to write compact code, but it is more difficult to find errors in a dense block of text. White space is your friend!

Indent code for loops, conditional statements, and the ends of routines. Look at the following code. It repeats the same lines. The first time there is no indentation; the second time has indented code. The second block of code is much easier to read.

```
if Err=11 then
```

```
a=5
resume
else
wait
end if

if Err=11 then
    a=5
    resume
else
    wait
end if
```

Here is the same example, but it is missing the "end if" statement. You can see that it is much easier to spot this omission when the code is indented.

```
if Err=11 then
a=5
resume
else
wait


if Err=11 then
    a=5
    resume
else
    wait
```

It's best to avoid chaining commands. Liberty BASIC allows you to place multiple statements on a single line if they are separated by colons. This practice can make it more difficult to isolate errors. The following example illustrates this.

```
a=12:b=4:print b/a:a=0:print b/a
```

Liberty BASIC allows you to add comments to code. They are ignored by the compiler, so they do not cause your program to be larger when tokenized. You can begin a line with the letters REM or with an apostrophe character. This tells Liberty BASIC to ignore the line. You can also add a comment to the end of a line with the apostrophe character.

You may know what's going on in a routine when you write the code, but as the program grows and time passes, it will become difficult to remember everything.

```
REM This program by John Q. Coder
' Please give credit if you use this code.

[branchOne]
a = 1    'a is number of hours worked
b = 3.50 'b is wage
c = 3    'c is employee type
d = a*b  'd is week's paycheck
```

Comments such as those above make it easier to understand code and to debug it.

Use descriptive names for variables and routines to make it even easier to follow the code and to find errors. It might require a little more typing, but the effort is worth it. It might not be necessary to add comments if descriptive names are used.

```
REM This program by John Q. Coder
' Please give credit if you use this code.

[calculateWages]
hours =  1      ' number of hours worked
wage =   3.50   ' wage
employee = 3    ' employee type
paycheck = a*b  ' week's paycheck
```

It is easier to debug code that is constructed with few GOTO statements. Coders can be passionate in denigrating the GOTO, while others staunchly defend it. Regardless of your feelings about the use of GOTOs, program flow is more difficult to follow if the code jumps around erratically. This makes debugging more difficult.

Code Construction

When you write code, build one routine at a time. Test it thoroughly. When a bug is encountered, it is probably in the newest section of code.

If you attempt to merge two large blocks of code, perhaps from two separate programs, it's easy to introduce errors.

If you attempt to make many changes throughout the code before testing any of them, it will be very difficult to isolate bugs.

Compiler Reporting

Liberty BASIC has an incredibly useful tool called Compiler Reporting. This is an optional tool in the Liberty BASIC editor, so be sure to check the box in the preferences dialog in the Setup Menu.

If compiler reporting is active, a reporting pane will be displayed at the bottom of the editor any time Liberty BASIC finds similar variable names at compile time.

Be sure that the "Enable Compiler Reporting" box is checked, then run this two-line program.

```
var=27
Var=33
```

The reporting pane will be displayed and it will contain this message:

- similar variables: Var, var

Procedure View

The "Jump-To" button (Alt-G) opens a scrollable list of procedures on the left side of the editor. Type a letter to jump automatically through procedure names that begin with that letter, or scroll with the scrollbar. This listing helps you keep track of the code structure. If you use meaningful procedure names it will be very easy to find sections of code.

Use the Helpfile

Read the pertinent topics in the helpfile to make sure the syntax is correct. On the forums we sometimes see questions like, "I can't make the !copy command work. What's wrong?" When we ask for code to illustrate the problem we find that the control is a textbox. The "!copy" command works in texteditors, but not in textboxes. If the questioner had read the helpfile topic on textboxes, he would not have found that command listed.

All commands understood by a control are listed in the control's topic in the helpfile. Textboxes and texteditors understand some of the same commands, but not all of them. The same is true for listboxes and

comboboxes, as well as for buttons and bmpbuttons, etc.

Debug in the MainWindow

You can use the mainwindow while you are developing your program. Remove any NOMAINWIN statements so that the mainwindow will be displayed. Print messages in the mainwindow, such as the values of variables or the section of code being run. Here is an example that prints values to the mainwindow.

```
pi=3.1416
print "Calculate the circumference of a circle."
print "Radius?"
input radius
diam=2*radius
circum=pi*diam
print pi
print radius
print diam
print circum
```

The demo above prints a column of numbers into the mainwindow. Although this can be helpful, it can also be confusing. Take just a little more time to give yourself meaningful messages, as in the improved version below.

```
pi=3.1416
print "Calculate the circumference of a circle."
print "Radius?"
input radius
diam=2*radius
circum=pi*diam
print "Pi is ";pi
print "Radius is ";radius
print "Diameter is ";diam
print "Circumference is ";circum
```

Lite Debug

Look under the RUN menu in the Liberty BASIC editor to find "Lite Debug." This is by far the easiest way to find an error. If you test with "Lite Debug" instead of with a simple "Run" and the program

encounters an error, the debugger will automatically be displayed with the errant line highlighted. You can't get much easier than that!

Breakpoints

If you know the section of code that causes the problem, add a breakpoint, then run the program normally. It will execute code until it encounters the breakpoint. At that point the debugger will open.

There are two ways to add a breakpoint.

The TRACE command adds a breakpoint.

```
TRACE n
```

Here is the description of TRACE from the helpfile that describes the three possible values for n.
There are three trace levels: 0, 1, and 2. Here are the effects of these levels:

0 = full speed no trace or RUN
1 = animated trace or ANIMATE, logs variables and highlights current line
2 = single step mode or STEP, requires programmer to click STEP button to continue to next line of code to execute, logs variables

You can add a breakpoint in the editor by double-clicking in the area to the left of the editor pane. Double-click a second time to remove the breakpoint. When you run the program with the debugger, execution will stop at the breakpoint.

Debugger Map

The debugger window has several sections. They are described below.

variable pane This is at the top of the debugger window. It lists variables and their current values. It can scroll automatically or manually. It includes the option to show values of the default variables. Use this to check the variables in your code as well as their values.

code pane This texteditor displays program code. It scrolls to show the current code and highlights the currently executing line.

execute pane This handy tool allows you to execute code against a running program. You can type in a command, then press execute button to activate. One example of use is to change the value of a variable to

see how it affects the program.

left bar The bar at the left of the texteditor is for breakpoints. Double click to add a breakpoint dynamically, or to remove one.

buttons

- *Resume* runs your program at full speed in the debugger. While in this mode, you won't see variables change or program source code highlighted.
- *Stop* will cause your program to stop, and it will highlight the line where it stopped, and it will show the current variable contents
- *Step Into* will execute the next line of code. If the next line is inside a subroutine or function it will follow execution into the subroutine or function.
- *Step Over* will execute the next line of code. It will not step into subroutines or functions, but skips over them.
- *Step Out* will run until the current subroutine or function exits, and then stops to show the next line of code and variables.
- *Animate* runs your program, showing each line as it executes, and also updating variables as it runs.

Debugger Help

Click the question mark button to open a tutorial that explains the features of the debugger.