

# VBA Programming in Excel: Text Files for Input/Output – Error Handling

By Gilberto E. Urroz, November 2013

In this chapter we introduce the use of text files for input and output in VBA programs. The examples we have presented so far utilize input through *InputBoxes* or from the spreadsheet interface, and produce output using *MsgBoxes* or by writing data to the interface. In some instances, you may need to input, into your programs, data that are already available in a text file, e.g., numerical records from a transducer (velocities, currents, etc.), or from a data depository online, e.g., the FedStats website, <http://www.fedstats.gov/>, which provides access to a variety of statistics from the U.S. government. Alternatively, you may need to produce a printed report, in the form of a text file, showing the results from your program. The use of text files for input and output, therefore, is the main subject of this chapter.

## Text files

Data that is input from or output to a spreadsheet interface through a VBA program is permanently saved with the spreadsheet. Thus, we'll refer to those data as *permanent data*. Data that is defined or calculated within a VBA program or subprogram, but not placed in a spreadsheet interface, is said to be *volatile data*. In other words, such data is lost once the program execution ends. Besides saving permanent data in a spreadsheet interface, data can be saved, permanently, into a text file.

A text file is a file containing ASCII characters representing text and numbers. Files typed using a text editor, such as Window's own *Notepad*, are text files. Their names typically end in *.txt*, although, other suffixes such as *.dat* (for *data*), can be used.

### The *Notepad* text editor in Windows

You can find the *Notepad* text editor under the Accessories folder in Windows 8, or under the *Applications* folder in earlier versions of Windows. If using *Notepad* to create or read a file, it is recommended that you use the following settings:

(1) *Format > Word Wrap* (unchecked)

(2) *File > Format > Font*: select **Font**: Courier New, **Font style**: Regular, and **Size**: 10

The reason for selecting *Courier New* as the font is because *Courier New* is a fixed-width font, so that formatting of files is straightforward. In other words, alignment of columns is easier to recognize when reading, or to do while typing a file. Unchecking *Word Wrap* allows the user to see the file rows and columns as originally intended by the file's author. If the window size for *Notepad* is too small it may not accommodate the entire file for showing, but you can always scroll up-and-down or sideways to see the entire file contents.

The figure below shows the *Notepad* interface showing a text file named *MyInputFile.txt*. This file contains two columns of numerical data representing points ( $x,y$ ) that could be used for input to a program aimed at performing, for example, a linear regression.

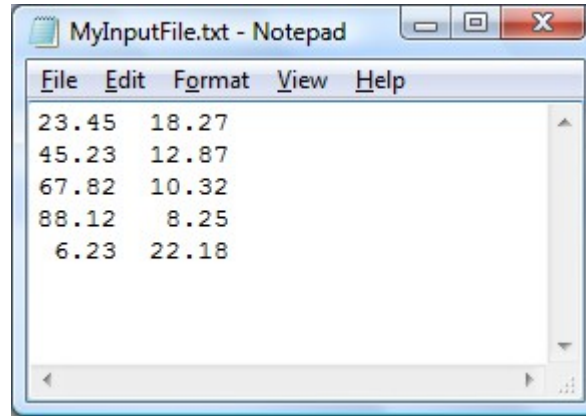


Figure 1. *Notepad* interface showing a two-column text file.

### Tab-separated and comma-separated files

In entering the data for Figure 1 is used a *tab* character to separate the columns. The tab stops in *Notepad* are set to move to the next position that is a multiple of 8, i.e., if your cursor is at the left margin of a *Notepad* window, pressing the [Tab] key will move the cursor to position 8, next to position 16, 24, etc. A data file, such as that shown in Figure 1, is referred to as a *tab-separated* file, since the columns of data are created by separating them with the “tab” character. When you press the [Tab] key, an invisible “tab” character is entered into the file, which will move the cursor to the next tab position as described earlier.

An alternative to using “tab” characters to separate columns in a text file is to separate the entries using commas. The file of Figure 1, typed using commas, is shown in the following figure.

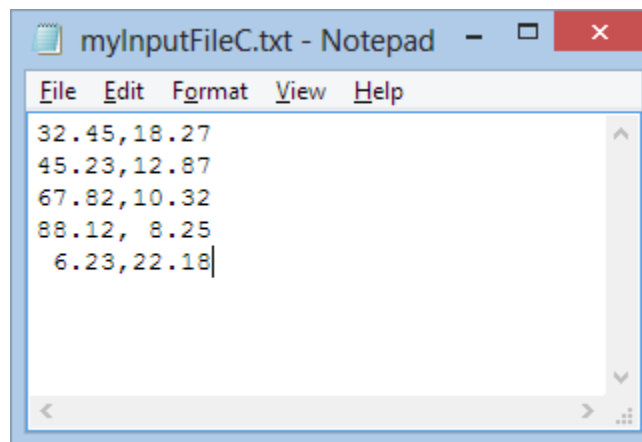


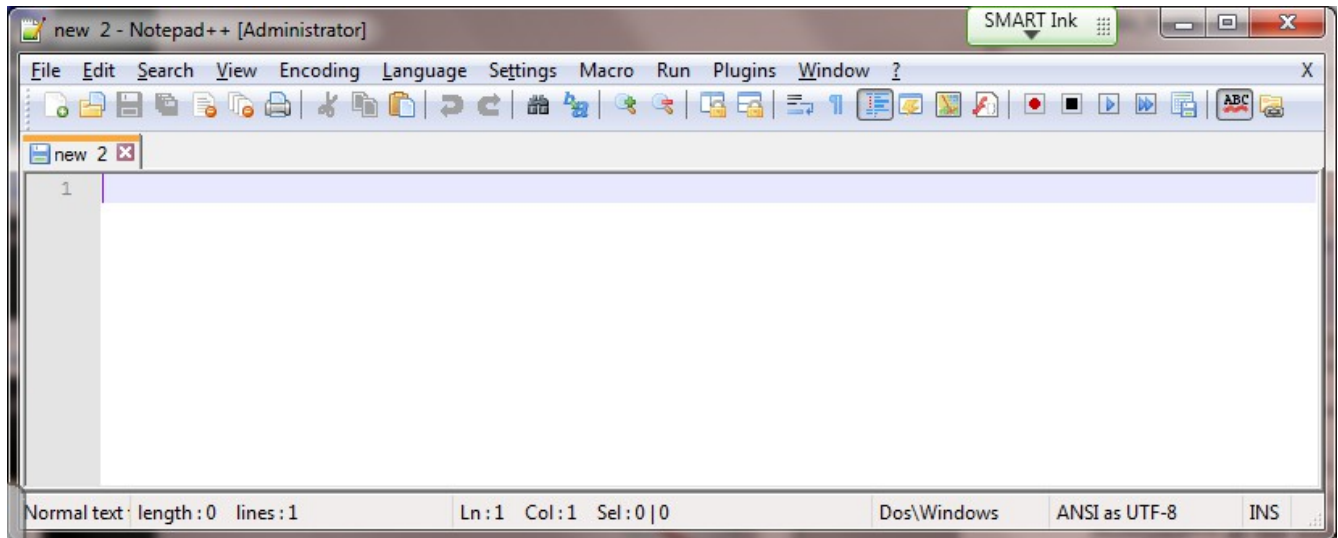
Figure 2. A comma-separated text file shown in the *Notepad* interface.

### A better text editor: *Notepad++*

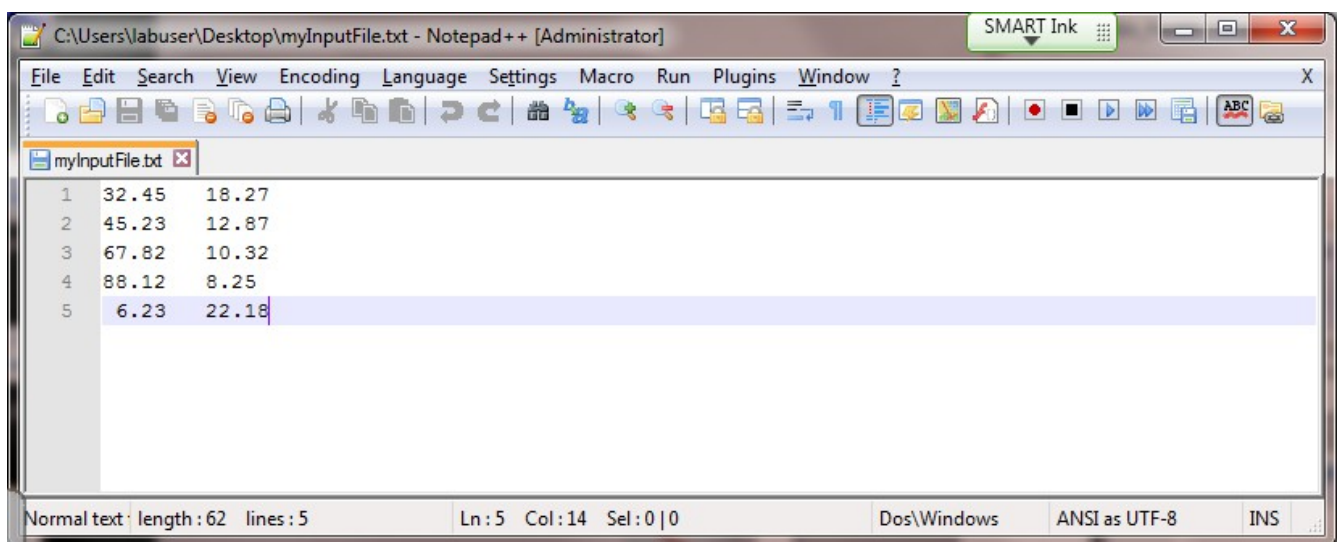
*Notepad++* is a free text editor with many more features than *Window's Notepad*. You can find *Notepad++* at the following website: <http://notepad-plus-plus.org/>

Click on the *DOWNLOAD* tab and then click on the *Download* link. Save the file and install it by double-clicking on it.

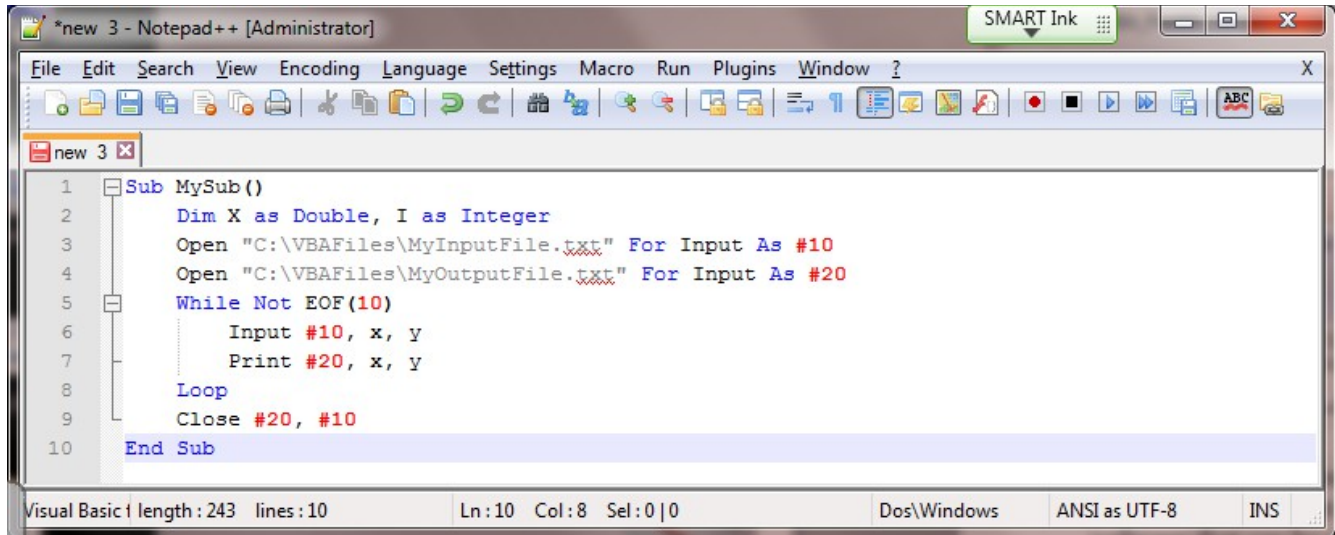
The *Notepad++* interface is shown next:



- For HELP on the use of *Notepad++* click on the question mark [?] button in the toolbar, and select *Help Contents* (or press *Shift+F1*)
- Select the menu option *Languages > N > Normal Text* to see text files. For example, file *MyInputFile.txt* open in *Notepad++* is shown next:



You can use the menu option *Language > V > VB* if you want to type *Visual Basic* code in *Notepad++*. Notice that this is not *Visual Basic for Applications (VBA)*, but *Visual Basic (VB)*, the *Microsoft* software that can be used to create self-standing applications in the *Windows* operating system – the current version is *Visual Basic 2012*. However, the *VB* code produced using *Notepad++* is close enough to *VBA* code that you can develop your code in *Notepad++* and then cut and paste your code to the *VBA IDE*. Here is an example of *VB* code in *Notepad++*:



```

1 Sub MySub()
2     Dim X as Double, I as Integer
3     Open "C:\VBAFiles\MyInputFile.txt" For Input As #10
4     Open "C:\VBAFiles\MyOutputFile.txt" For Input As #20
5     While Not EOF(10)
6         Input #10, x, y
7         Print #20, x, y
8     Loop
9     Close #20, #10
10 End Sub

```

Visual Basic | length: 243 | lines: 10 | Ln:10 Col:8 Sel:0|0 | Dos\Windows | ANSI as UTF-8 | INS

Remember to select *Languages > N > Normal Text* to work with text files if you use *Notepad++*.

**Example 1 – Download a comma-separated textfile online, open, and save it in Excel.** Let's download a comma-separated file from the web site *FedStats*, which is a depository of data for the U.S. Federal Government. Follow these instructions:

- Open the web site: [www.fedstats.gov](http://www.fedstats.gov)
- Click on the link *Statistical Reference Shelf*.
- Find the link *Energy Information Administration's Monthly Energy Review*, and click on it.
- Find the Topic: *1.1 Primary Energy Production by Source*. Towards the right you will see links labeled: PDF, XLS (Excel file), CSV (comma-separated values), and GRAPH (in PDF format).
- Lets' download both the XLS and the CSV files, by clicking on the icon and saving the corresponding file.

The name of the CSV file I downloaded is *MER\_t01\_02.csv* (but it could be a different name at a later date). Opening this file in *Notepad*, shows the first few lines of the file as follows. Notice the commas separating individual elements.

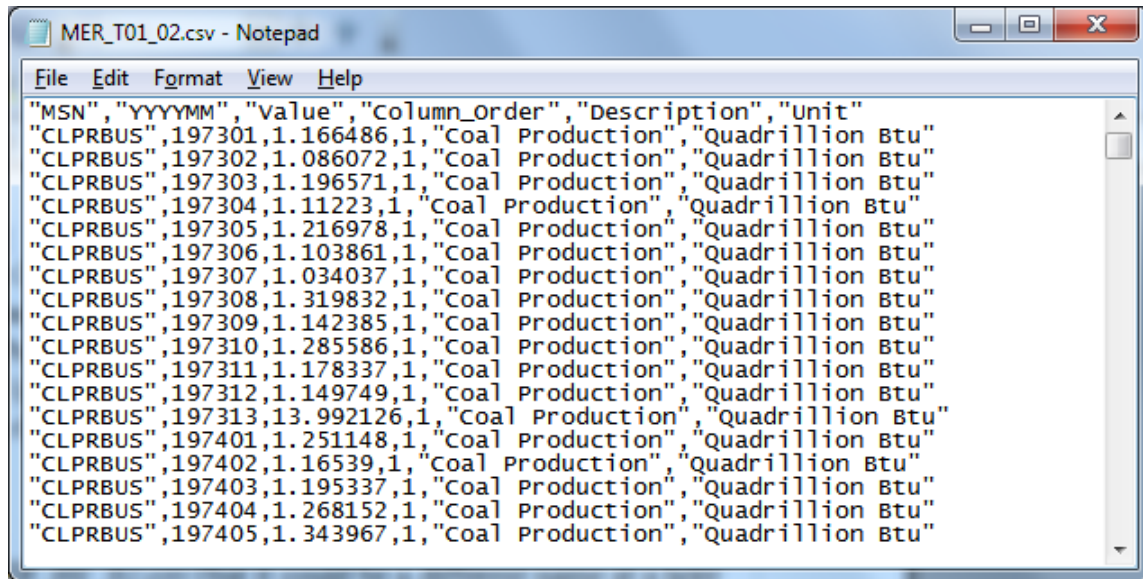


Figure 3. Example of a comma-separated file downloaded from the *FedStats* web site open in *Notepad*.

Comma-separated files (their names typically end in *.csv*) are easy ways to store tabular data online in the form of text files. Comma-separated files can be easily open in the Excel spreadsheet as follows:

- Open a new Excel spreadsheet
- Select the menu option *File > Open*
- Make sure that the option *All files (\*.\*)* is selected in the *Open* form
- Navigate towards the location where the file *MER\_T01\_02.csv* was stored and click on it
- Press the [Open] button
- In the resulting form, check off the option [ ] *Comma*, and press the [ OK ] button.

Before you press the [OK] button, you may notice that the different entries in the file, the ones separated by commas, are distributed in columns of the spreadsheet shown at the bottom of the form. This suggests the format in which the data will be entered in the spreadsheet. Once you press the [OK] button, the worksheet will look as follows:

	A	B	C	D	E	F
1	MSN	YYYYMM	Value	Column_Order	Description	Unit
2	CLPRBUS	197301	1.166486	1	Coal Production	Quadrillion Btu
3	CLPRBUS	197302	1.086072	1	Coal Production	Quadrillion Btu
4	CLPRBUS	197303	1.196571	1	Coal Production	Quadrillion Btu
5	CLPRBUS	197304	1.11223	1	Coal Production	Quadrillion Btu
6	CLPRBUS	197305	1.216978	1	Coal Production	Quadrillion Btu
7	CLPRBUS	197306	1.103861	1	Coal Production	Quadrillion Btu
8	CLPRBUS	197307	1.034037	1	Coal Production	Quadrillion Btu
9	CLPRBUS	197308	1.319832	1	Coal Production	Quadrillion Btu
10	CLPRBUS	197309	1.142385	1	Coal Production	Quadrillion Btu

Figure 4. The comma-separated file of Figure 3 open in an Excel interface.

The figure above shows the first few rows of the file. The top row represents the following column headings:

- **MSN**: energy type, e.g., *CLPRBUS* (Coal), *NGPRBUS* (Natural Gas Dry), *PAPRBUS* (Crude Oil), *NLPRBUS* (Natural Gas Liquid), *FFPRBUS* (Total Fossil Fuels), *NUETBUS* (Nuclear), *HVTCBUS* (Hydroelectric), *GETCBUS* (Geothermal), *SOCTCBUS* (Solar), *WYTCBUS* (Wind), *BMPRBUS* (Biomass), *REPRBUS* (Total Renewable Energy), *TEPRBUS* (Total Primary Energy).
- **YYYYMM**: date, year (YYYY, 4 digits) and month (MM, 2 digits) when the data is reported.
- **Value**: amount of energy production in units given in the **Unit** column (described below).
- **Column\_Order**: values 1 through 13 corresponding to the 13 different energy types reported under the **MSN** column (see above).
- **Description**: specific description of the energy source corresponding to the 13 different energy types listed under the **MSN** column or their corresponding numbers listed under **Column\_Order**.
- **Unit**: you can check that all the data is given in units of *Quatrillion Btu*. One quadrillion is one million trillion units, or one million million billion units, or one million million million million units. Since  $1 \text{ million units} = 10^6 \text{ units}$ ,  $1 \text{ quadrillion units} = 10^6 \cdot 10^6 \cdot 10^6 \cdot 10^6 = 10^{24} \text{ units}$ . One *Btu* (or *BTU* = *British Thermal Unit*) is equal to 1055.06 joules. Since all units are the same, we consider only the value shown in this column for processing data.

Having opened the *csv* file into an Excel interface, we can now save the file under the *ods* (open document spreadsheet) format by using *File > Save As ...*, keeping the same name, and selecting the option *ODF Spreadsheet (.ods)* (\*.ods) in the *Save as type*: menu. After pressing the [Save] button, the file name will be changed, in this case, to *MER\_T01\_02.ods*. At this point we can process the data in various ways, utilizing VBA code.

### Example 1 - Processing of data in downloaded file using VBA code

Each entry in the interface shown in Figure 4, past the first line which represents column headings only, consists of 6 data items described above. The columns **MSN**, **Column\_Order**, and **Description** are equivalent as indicated in the following list:

Table 1. Different types of energy production from Figure 4.

<b>MSN</b>	<b>Column_order</b>	<b>Description</b>
CLPRBUS	1	Coal
NGPRBUS	2	Natural Gas Dry
PAPRBUS	3	Crude Oil
NLPRBUS	4	Natural Gas Liquid
FFPRBUS	5	Total Fossil Fuels (i.e., 1 through 5)
NUETBUS	6	Nuclear
HVTCBUS	7	Hydroelectric
GETCBUS	8	Geothermal
SOCTCBUS	9	Solar
WYTBUS	10	Wind
BMPRBUS	11	Biomass
REPRBUS	12	Total Renewable Energy (6 through 11)
TEPRBUS	13	Total Primary Energy (5 and 12)



Thus, we need only to identify the **Column\_order** in each row to know which category of energy production each list belong to. All the **Unit** column specifications represent the same units (Quatrillion BTUs), thus, we need not worry about that particular column. The columns of interest, therefore, are **Column\_order** (4<sup>th</sup> column), **YYYYMM** (2<sup>nd</sup> column), and **Value** (3<sup>rd</sup> column).

The data shown in Figure 4 should be reordered, in a new worksheet (call it “DataSummary”), so that we produce two columns corresponding to a date and the value (in Quatrillion BTU), for each of the 13 types of energy production listed in Table 1, above. Once we have those 26 columns in the worksheet “DataSummary” (2 columns for each of the 13 categories in Table 1), we can proceed to produce plots of the different energy categories.

To evaluate the dates, we can read the values of the column **YYYYMM** as integer values, and then separate them as follows. Suppose that we call a particular **YYYYMM** value with the variable name *YearAndMonth*. An inspection of the data in the worksheet of Figure 4 reveals that the earliest year available is 1973. Thus, we can split a particular value of *YearAndMonth* into a *Year* value and a *Month* value as follows:

$$Year = \text{INT}(YearAndMonth / 100)$$

$$Month = YearAndMonth - Year * 100$$

Where the VBA math function INT produces the integer part of the corresponding argument. For example, if *YearAndMonth* = 198506, then *Year* = INT(*YearAndMonth*/100) = INT(198506/100) = INT(1985.06) = 1985, and *Month* = *YearAndMonth* – *Year* \* 100 = 198506 – 1985 \* 100 = 198506 – 198500 = 06, or simply, 6. Thus, extracting the *Year* and *Month* out of the entry *YearAndMonth* = 198506, produces *Year* = 1985 and *Month* = 6.

We also want to convert the *Year* and *Month* values to a *Time* value representing the number of months since January 1973, by using:

$$Time = (Year - 1973) * 12 + Month$$

Thus, if *Year* = 1985 and *Month* = 6, *Time* = (*Year* – 1973) \* 12 + *Month* = (1985 – 1973) \* 12 + 6 = 12\*12 + 6 = 144 + 6 = 150. This means that June of 1985 is the 150-th month since January of 1973. This way, using January 1973 as our zero *Time*, we can keep track of the number of months corresponding to a particular value of *YearAndMonth* (the 2<sup>nd</sup> column in Figure 4).

The algorithm to process the more than 6000 rows in the worksheet shown in Figure 4 will require us to:

- Read the **Column\_order** values (4<sup>th</sup> column in Figure 4),
- Read the corresponding value of *YearAndMonth* (or **YYYYMM** in the 2<sup>nd</sup> column in Figure 4), convert it into *Year* and *Month* data, and then into *Time* data (in months, with zero value at January 1973),
- Read the **Value** from the 3<sup>rd</sup> column of Figure 4,

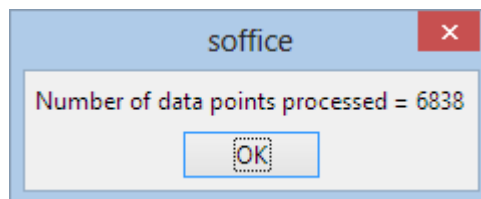
- Write the *Time* value and then the corresponding **Value** (i.e., energy production value in Quadrillion BTUs) into columns in the worksheet “DataSummary”, that we would have added to our workbook
- When the value of **Column\_order** changes from its current value, we will start writing data in the next column to the right from the current output columns, and continue until the data is exhausted.
- You may notice that for each year there is one **YYYYMM** corresponding to a 13<sup>th</sup> month. In reality, this 13<sup>th</sup> month data point represents the total yearly production. That data value will be skipped in the data processing.
- Also, certain energy production types, such as solar energy, had no production reported in the 1970s. Instead, a very large value (10000000) is placed in the data set to represent a zero value. Thus, when reading a value that large, the actual value should be converted to zero.

The code that takes care of processing the data according to the requirements shown above is presented in next page. Part of the resulting table of values extracted from “Sheet1” into “DataSummary” is shown below.

	A	B	C	D	E	F	G	H	I	J	K
1	Time (month)	Coal	Natural Gas	Crude Oil	Natural Gas	Total Fossil	Nuclear	Hydroelectric	Geothermal	Solar	Wind
2	1	1.166486	1.908249	1.649833	0.210876	4.935444	0.068103	0.272703	0.001491	0	0
3	2	1.086072	1.922543	1.525783	0.197822	4.73222	0.064634	0.242199	0.001363	0	0
4	3	1.196571	1.86843	1.667094	0.217711	4.949805	0.072494	0.26881	0.001412	0	0
5	4	1.11223	1.777561	1.616791	0.212524	4.719106	0.06407	0.253185	0.001649	0	0
6	5	1.216978	1.859241	1.665377	0.21831	4.959906	0.062111	0.26077	0.001537	0	0
7	6	1.103861	1.802065	1.603224	0.209742	4.718893	0.073968	0.249859	0.001763	0	0
8	7	1.034037	1.841884	1.65724	0.218022	4.751184	0.07589	0.23567	0.001869	0	0
9	8	1.319832	1.857199	1.648505	0.219387	5.044923	0.084883	0.222077	0.001762	0	0
10	9	1.142385	1.803086	1.577362	0.211491	4.734325	0.085724	0.179733	0.001492	0	0
11	10	1.285586	1.836779	1.658452	0.220444	5.001261	0.081971	0.191723	0.00176	0	0
12	11	1.178337	1.825548	1.594008	0.215484	4.813377	0.088773	0.210285	0.001889	0	0
13	12	1.149749	1.886808	1.629568	0.216966	4.88309	0.087557	0.274435	0.002435	0	0
14	13	1.251148	1.895424	1.60631	0.211267	4.96415	0.084855	0.304506	0.002154	0	0
15	14	1.16539	1.728512	1.484696	0.1941	4.572698	0.090577	0.27995	0.001901	0	0

Figure 5. Processed data in the “DataSummary” worksheet.

When the program finished the processing, the following *MsgBox* is shown.





```

Option VBASupport 1
Option Explicit
Option Base 1
'-----
Sub DataProcessing()
'Definition of variables
    Dim dataCount As Integer, dataType As Integer, countType(13) As Integer
    Dim YearAndMonth As Long, Year As Integer, Month As Integer
    Dim dataValue As Double, k As Integer, Time As Integer
    Dim typeString(13) As Variant
    typeString = Array("Coal", "Natural Gas Dry", "Crude Oil", _
        "Natural Gas Liquid", "Total Fossil Fuels", _
        "Nuclear", "Hydroelectric", "Geothermal", _
        "Solar", "Wind", "Biomass", "Total Renewable", _
        "Total Primary")
'Initialize counter for each type of energy production and write headings for
table
    Worksheets("DataSummary").Range("A1").Value = "Time"
    For k = 1 To 13
        countType(k) = 0
        Worksheets("DataSummary").Range("A1").Cells(1,k+1).Value = typeString(k)
    Next k
'This variable keeps track of the total number of data points processed
    dataCount = 1
'The following loop process the data as required
    Do While Worksheets("Sheet1").Range("A2").Cells(dataCount,4).Value <> ""
        dataType = Worksheets("Sheet1").Range("A2").Cells(dataCount,4).Value
'input dataType
        YearAndMonth = Worksheets("Sheet1").Range("A2").Cells(dataCount,2).Value
'input YYYYMM
        Year = INT(YearAndMonth/100) : Month = YearAndMonth - Year * 100
'separate YYYY and MM
        If Month <> 13 Then 'For months other than 13 process data
            countType(dataType) = countType(dataType) + 1 'Increment each energy
type counter
            Time = (Year - 1973) * 12 + Month 'Calculate time
            dataValue = Worksheets("Sheet1").Range("A2").Cells(dataCount,3).Value
'Write value to table
            If dataValue > 1.E+5 Then 'If a large value exists, make it zero
                dataValue = 0.0
            End If
            If dataType = 1 Then 'Write Time only once
                Worksheets("DataSummary").Range("A2").Cells(countType(dataType),1).Value = Time
            End If
            'Write energy production value to table
            Worksheets("DataSummary").Range("A2").Cells(countType(dataType),dataType+1).Value
= dataValue
            End If
            dataCount = dataCount + 1 'Increment the data counter by one
        Loop
        'Adjust and report total data count
        dataCount = dataCount - 1
        MsgBox("Number of data points processed = " & Ctr(dataCount))
    End Sub

```

## VBA commands for using text files for input and output

Suppose that you don't want to convert the comma-separated file from Example 1 into a worksheet before processing the data. Instead you want to read the data to be processed directly from the original text file. Furthermore, suppose that you want to write the results of your data processing to a text file instead of writing them to a second worksheet. VBA allows you to *open* and *read from* input text files, as well as *open* and *write* to output text files. You can also *append* data to an existing text file. Once you have finished using your input and/or output files, you can *close* them to get them ready to be used by other VBA programs, if needed.

### Location of input and/or output text files

You can store a text file to be used for input or output in any location in your computer's hard disk. To find out the full specification of a particular file's location, follow the procedure detailed next. Suppose that, somewhere in my computer's hard disk, I stored the file I downloaded earlier from *FedStats*. After navigating through my hierarchy of folders, I found the file in the folder shown in the following figure, and clicked on the file to selected. Then, I clicked on the down arrow highlighted in Figure 6, and typed *Cntl+C*, to copy the field to the left of the down arrow. If you open a *Notepad* or *Notepad* + blank file, and type *Cntl+V*, for pasting, you will see the location specification for the folder as:

```
C:\Users\Gilberto\Documents\MyVBATextFiles
```

Thus, the full specification for the file is:

```
C:\Users\Gilberto\Documents\MyVBATextFiles\MER_T01_02.csv
```

This type of specification is needed in VBA to be able to access specific text files for input or output.

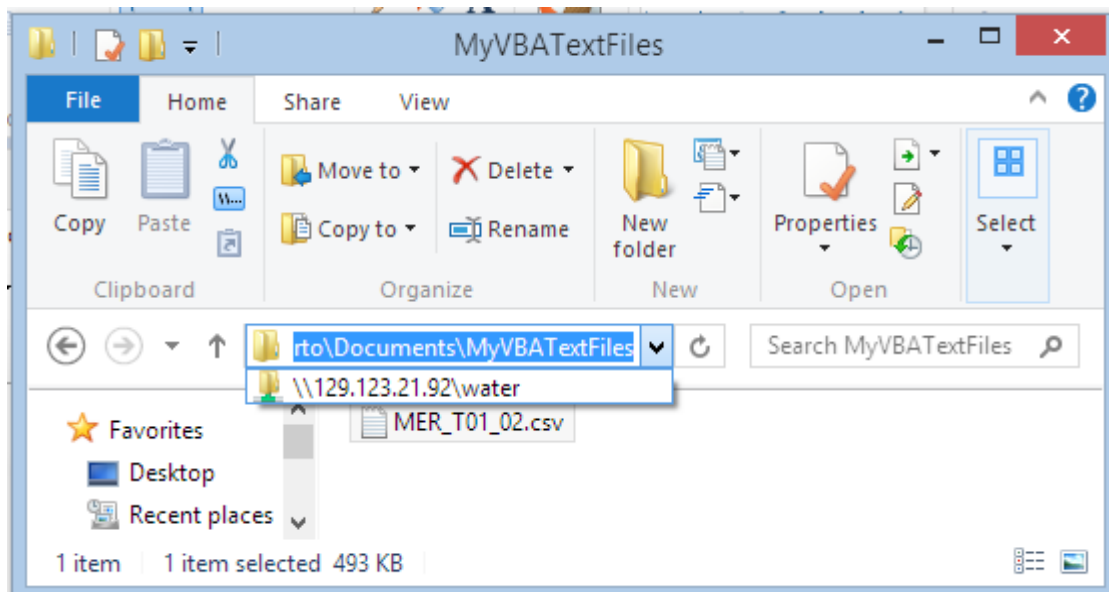


Figure 6. Determining the location specification of a particular file in a folder.

To facilitate typing the name of text files in VBA/Excel I recommend creating a folder in your C:\ drive called VBAFiles (or any suitable, easy-to-remember, name). This will be the folder where you will place all your input and output files for *Excel/VBA* programming. This way, if you were to create a file called, for example, *MyInputFile.txt* (see Figure 1), and place it within the *VBAFiles* folder in your C:\ drive, you can refer to it by using the name:

```
C:\VBAFiles\MyInputFile.txt
```

and avoid long names as the one listed earlier. Thus, if you were to copy the file downloaded earlier from *FedStats* into this folder, the reference to this file would be simply:

```
C:\VBAFiles\MER_T01_02.csv
```

Next, we discuss the VBA commands needed to manipulate text files for input and output.

### The *Open* statement

To access a file within a *VBA* program you need to open it using the *Open* statement. The general form of this statement, as provided in the *Excel/VBA Help* facility (<http://msdn.microsoft.com/en-us/library/aa266177%28v=vs.60%29.aspx>), is:

```
Open pathname For mode [Access access] [lock] As [#] filenumber [Len=reclength]
```

The **Open** statement has the following parts:

Part	Description
<i>pathname</i>	Required. String expression specifying a file name.
<i>mode</i>	Required. Keyword specifying file mode as: Append, Binary, Input, Output, or Random.
<i>access</i>	Optional. Keyword specifying the operations permitted on the open file: <b>Read</b> , <b>Write</b> , or <b>Read Write</b> .
<i>lock</i>	Optional. Keyword specifying the operations permitted on the open file by other processes: <b>Shared</b> , <b>Lock Read</b> , <b>Lock Write</b> , and <b>Lock Read Write</b> .
<i>filenumber</i>	Required. A valid file number (1 to 511). Use the <i>FreeFile</i> function to obtain the next available file number.
<i>reclength</i>	Optional. Number ≤ 32,767 (bytes). For Random access files, this is the record length. For sequential files, it's the number of characters buffered.

### *Remarks:*

- You must open a file before any I/O<sup>1</sup> operation can be performed on it. **Open** allocates a buffer for I/O to the file and determines the mode of access to use with the buffer.
- If the file specified by *pathname* doesn't exist, it is created when a file is opened for **Append**, **Binary**, **Output**, or **Random** modes.
- If the file is already opened by another process and the specified type of access is not allowed, the **Open** operation fails and an error occurs.
- The **Len** clause is ignored if mode is **Binary**.
- **Important:** In **Binary**, **Input**, and **Random** modes, you can open a file using a different file number without first closing the file. In **Append** and **Output** modes, you must close a file before opening it with a different file number

### *File modes*

---

1 I/O stands for “Input/Output”

Notice that you can use the *Open* statement to access *binary* files. Unlike text files, binary files cannot be edited by a text editor. In this chapter we use only text files, therefore, examples using binary files will not be addressed herein.

Selecting a file for *Input* typically means that you will not be writing data into the file, whereas a file selected for *Output* will be used specifically for writing data to it. The *Append* mode is used to add data to the end of an existing file. Typically data records are accessed one after the other, thus files with *Input*, *Output*, and *Append* mode are referred to as *sequential* files.

A *Random* access file is one where you can read and write at any location as long as you specify that input or output location. When using random access files you need to specify the record length (*reclength*, above).

As an example of function *Open*, if you were to get data from file *C:\VBAFile\MyInputFile.txt* for input, you can open the file with the command:

```
Open "C:\VBAFiles\MyInputFile.txt" For Input As # 10
```

#### The Close statement

The *Close* statement is used to close open files. Several options for the *Close* statement are shown below:

- Close [#] *filenumber* - to close a single file
- Close [#] *filenumber1*, [#] *filenumber2*, ... - to close a list of files
- Close - to close all open files

For example, the file opened above as file number 10 could be closed by using:

```
Close # 10
```

#### The Input statement

The *Input* statement is used to input data from a file, identified as *filenumber*, while assigning the values read from a record in the file to variables in a list *varlist*. The general form of the statement is:

```
Input # filenumber, varlist
```

Each time that an *Input* statement is executed, a new record (i.e., line) in the input file is read. For example, for the file *MyInputFile.txt*, listed above with 5 rows of two data items each, we could use the following statements:

```
Input # 10, x1, y1
Input # 10, x2, y2
Input # 10, x3, y3
Input # 10, x4, y4
Input # 10, x5, y5
```

### A program using *Open*, *Input*, and *Close*

The figure below shows a program that opens file *MyInputFile.txt*, inputs data from it, and writes the data to the *Excel* interface.

```
Option Explicit
Public Sub OpenInputCloseExample()
' =====
' Example of Open, Input, and Close statements
' =====
' Declaration of variables
Dim x1 As Double, y1 As Double
Dim x2 As Double, y2 As Double
Dim x3 As Double, y3 As Double
Dim x4 As Double, y4 As Double
Dim x5 As Double, y5 As Double
' Opening input file
Open "C:\VBAFiles\MyInputFile.txt" For Input As #10
' Input data
Input #10, x1, y1
Input #10, x2, y2
Input #10, x3, y3
Input #10, x4, y4
Input #10, x5, y5
' Close input file
Close #10
' Write data
Range("A1") = x1: Range("A2") = y1
Range("B1") = x2: Range("B2") = y2
Range("C1") = x3: Range("C2") = y3
Range("D1") = x4: Range("D2") = y4
Range("E1") = x5: Range("E2") = y5
End Sub
```

Figure 7. Example of a program using *Open*, *Input*, and *Close*.

After executing the program, the interface will look as follows:

	A	B	C	D	E
1	23.45	45.23	67.82	88.12	6.23
2	18.27	12.87	10.32	8.25	22.18
3					

Figure 8. Interface after writing the data from the program in Figure 7.  
The input data are stored in file *MyInputFile.txt* as shown in Figure 2.

### The EOF function

The *EOF* (End Of File) function is used to detect the end of a file so as to stop inputting data from it. The function *EOF(filename)* gets the value of TRUE if the end of file *filename* is found, or FALSE before the end of that file is found. The EOF file can be used as a condition in a *Do While ... Loop* statement which iterates through the input file, entering lines of data until the *end-of-file* is found.

As an example, the figure below shows a program performing a similar function as that of Figure 7, but using a *Do While ... Loop* statement and the *EOF* function:

```
Option Explicit
Public Sub OpenInputCloseExample()
    '=====
    ' Example of Open, Input, and Close statements
    '=====
    ' Declaration of variables
    Dim x As Double, y As Double
    Dim k As Integer
    ' Opening input file
    Open "C:\VBAFiles\MyInputFile.txt" For Input As #10
    ' Input data using EOF and Input, and write data to Excel
    k = 0
    Do While Not EOF(10)      ' Loop until end of file.
        Input #10, x, y      ' Read data x,y
        k = k + 1
        Cells(k, 1) = x
        Cells(k, 2) = y
    Loop
    ' Close input file
    Close #10
End Sub
'=====
```

Figure 9. Program using the *EOF* function and a *Do While ... Loop* statement. Data is input from a file and output to a worksheet interface.

### The Print statement

The *Print* statement can be used to print a number of values in a variable list (*varlist*) into a file referred to by a *filename*. The general form of the *Print* statement is:

**Print** [#] filename, varlist

The following program opens files for input and output, reads data out of the input file using a loop and function *EOF*, prints results to the output file, and closes both the input and output files.



```

Option Explicit
Public Sub OpenInputClosePrintExample()
    '=====
    ' Example of Open, Input, Print, and Close statements
    '=====
    ' Declaration of variables
    Dim x As Double, y As Double
    Dim k As Integer
    ' Opening input file
    Open "C:\VBAFiles\MyInputFile.txt" For Input As #10
    Open "C:\VBAFiles\MyOutputFile.txt" For Output As #20
    ' Input data using EOF and Input, and write data to Excel
    k = 0
    Do While Not EOF(10)      ' Loop until end of file
        Input #10, x, y      ' Read data x,y
        Print #20, x, y      ' Print data to file
    Loop
    ' Close input and output file
    Close #10, #20
End Sub
'=====

```

Figure 10. Program using the *EOF* function and a *Do While ... Loop* statement. Text files are used for both input and output.

After the program is run, file *MyOutputFile.txt* is created within the folder [C:\VBAFiles](#):

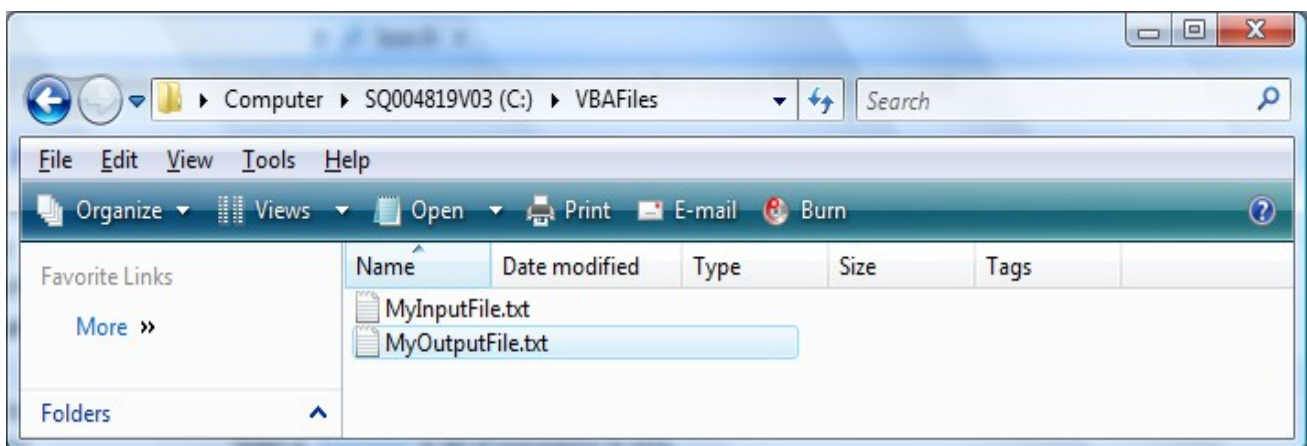
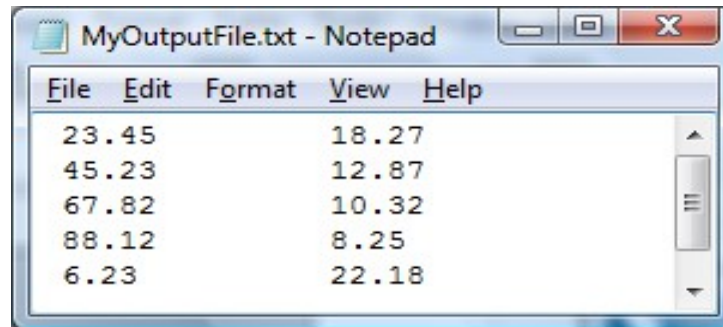


Figure 11. Folder *C:\VBAFiles* showing the input and output files used by the program of Figure 10.

The contents of the file *C:\VBAFiles\MyOutputFile.txt* are shown next:



23.45	18.27
45.23	12.87
67.82	10.32
88.12	8.25
6.23	22.18

Figure 12. Contents of the output file *MyOutputFile.txt* created and filled with data values using the program of Figure 10.

### Adding labels to output

The example of Figure 10, shown above, shows a simple variable list for input, as well as for output. If you want to add, for example, identifying labels to the output, you can combine variable names with label strings, separated by semi-colons, as illustrated in the following example.

```
Option Explicit
Public Sub PrintExample02 ()
    '=====
    ' Example - printing strings and variables using semi-colons
    '=====
    ' Declaration of variables
    Dim x As Double, y As Double
    Dim k As Integer
    ' Opening input and output files
    Open "C:\VBAFiles\MyInputFile.txt" For Input As #10
    Open "C:\VBAFiles\MyOutputFile02.txt" For Output As #20
    ' Input data using EOF and Input, and print data to file
    Do While Not EOF(10)      ' Loop until end of file
        Input #10, x, y      ' Read data x,y
        Print #20, "This is x = "; x; "This is y = "; y
    Loop
    ' Close input and output file
    Close #10, #20
End Sub
'=====
```

Figure 13. Example of a program using labels for output.

The *Print* statement shown in the program of Figure 13, above, produces the following output file:

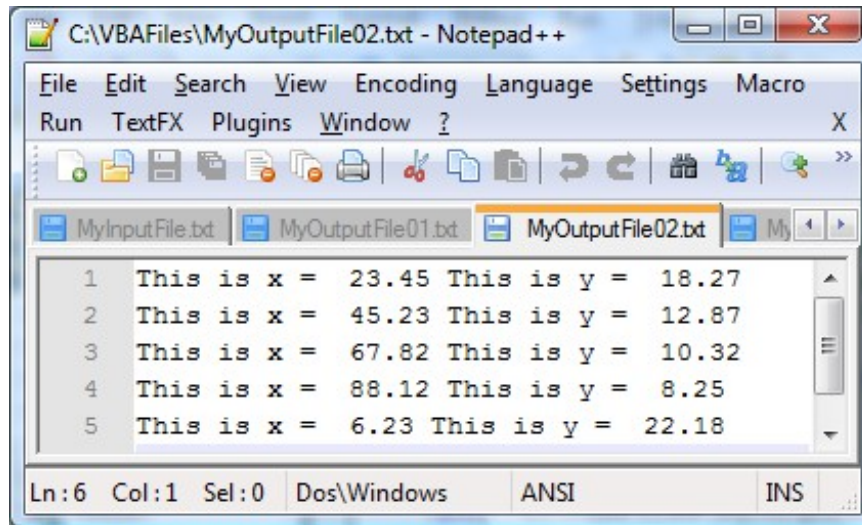


Figure 14. Contents of the output file *MyOutputFile02.txt* produced by the program of Figure 13.

The use of semi-colons to separate printing fields minimizes the use of spaces between fields. In contrast, using commas in the *Print* command, i.e.,

```
Print #20, "This is x = ", x, "This is y = ", y
```

produces additional spacing between fields:

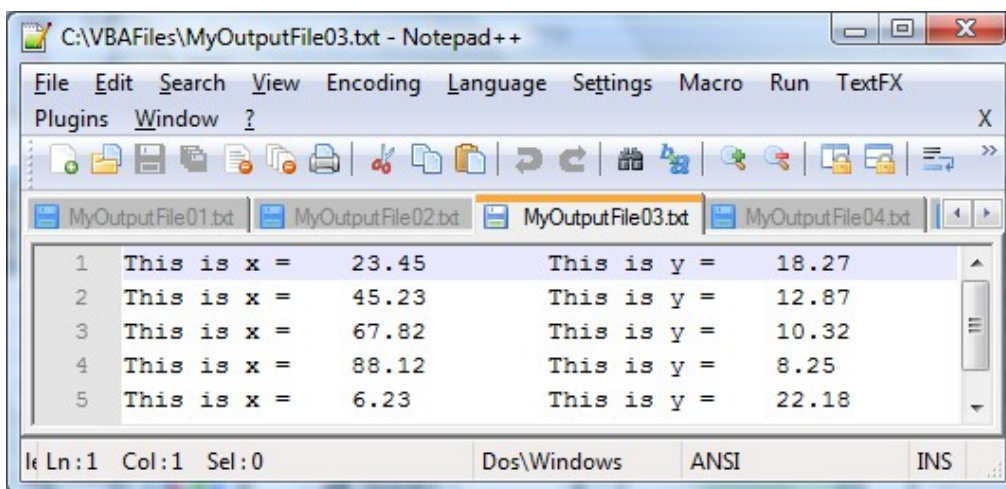


Figure 15. Contents of the output file *MyOutputFile02.txt* produced by the program of Figure 13 after changing semi-colons (;) to commas (,) in the *Print* statement.

The strings shown in the output lines of the program of Figure 13 associated with the function *Print*, e.g., "This is x = " or "This is y = " are string constants that are printed in the output file as text. Numbers, contained in these examples in variables *x* and *y*, are also printed as text. Thus, combinations of strings and numbers, for output into files using *Print* statements, are ideal for producing text reports since the strings and numbers merge into pure text in output.

### Function *Write*

Function *Write* is an alternative to function *Print* for outputting. However, unlike *Print*, string constants are written within double-quotes in output lines, and individual data items, in output, are separated with commas. For example, the output files produced by the statements:

```
Write #20, "This is x = ", x, "This is y = ", y
```

or

```
Write #20, "This is x = "; x; "This is y = "; y
```

are exactly the same, and are shown here:

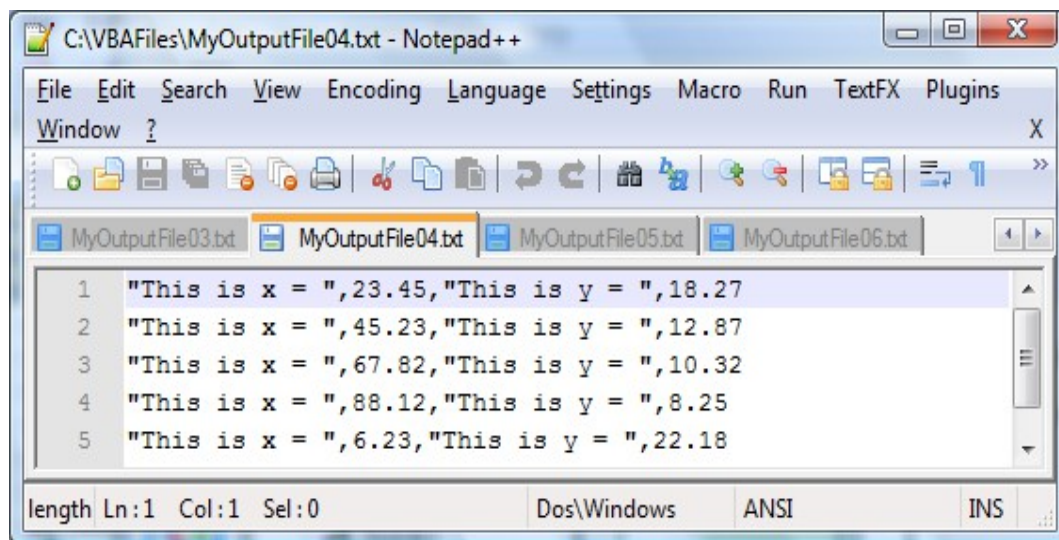


Figure 16. Output file *MyOutputFile04.txt* produced using *Write* statements.

As you can see from the results of Figure 16, string and numbers are output as data lists in each line of the output file. Thus, the use of function *Write* for output to a file is ideal for writing data files that can be used for input by other programs.

In summary, *Print* statements are recommended to produce report files, while *Write* statements are recommended to produce data storage files that can be used as input files for other programs.

### Simple output formatting with the *Print* statement

The functions *Tab* and *Spc* functions can be used to produce some formatting of the output line produced by the *Print* statement. The specification *Spc*(*n*) adds *n* spaces to the output line, while the specification *Tab*(*n*) starts the next output field at position *n*. The following program produces formatted output to a file using *Tab* and *Spc* to create the heading of a table. Make sure all your print specs are separated by semi-colons.

```
Public Sub PrintExample06()  
    ' Example - Writing headings in output file  
    ' Declaration of variables  
    Dim x As Double, y As Double, k As Integer  
    Open "C:\VBAFiles\MyInputFile.txt" For Input As #10          ' Open input file  
    Open "C:\VBAFiles\MyOutputFile06.txt" For Output As #20  
    ' Write headings to the output file using functions Spc() and Tab()  
    Print #20, "Example of printing with file headings"  
    Print #20, "Here we use Spc: "; Spc(10); " 10 spaces included in this line"  
    Print #20, "Here we use Tab: "; Tab(20); " move to tab position 20 here"  
    Print #20, "Using Spc"; Spc(5); "again"; Spc(5); "and again"; Spc(5); "and again"  
    Print #20, "Using Tab"; Tab(10); "again"; Tab(20); "and again"; Tab(30); "and again"  
    Print #20, "Next we print data read from an input file using semi-colons:"  
    Print #20, "-----"  
    k = 0 ' Input data using EOF and Input, and write data to output file  
    Do While Not EOF(10) ' Loop until end of file  
        k = k + 1  
        Input #10, x, y ' Read data x,y  
        Print #20, "Line No. "; k; " The value of x is "; x; " while that of y is "; y  
    Loop  
    ' Close input file (#10) and reopen it again file #30  
    Close #10  
    Open "C:\VBAFiles\MyInputFile.txt" For Input As #30  
    ' Add more headings to the output file  
    Print #20, '<----- adds a blank line to output file'  
    Print #20, "Repeating printing input data using commas:"  
    ' Input data using EOF and Input, and write data to Excel  
    k = 0  
    Do While Not EOF(30) ' Loop until end of file  
        k = k + 1  
        Input #30, x, y ' Read data x,y  
        Print #20, "Line No. ", k, " The value of x is ", x, " while that of y is ", y  
    Loop  
    ' Print last headings to the output file  
    Print #20, "-----"  
    Print #20, "This is the end of printing for this file"  
    ' Close input file #30 and output file  
    Close #30, #20  
End Sub
```

Figure 17. Program illustrating the use of *Open*, *Input*, *Print* – including some uses of the *Spc* and *Tab* specifications for formatting – and *Close* statements.

The result is the following file:



```

MyOutputFile06.txt - Notepad
File Edit Format View Help
Example of printing with file headings
Here we use Spc:          10 spaces included in this line
Here we use Tab:   move to tab position 20 here
Using Spc   again   and again   and again
Using Tab again   and again and again
Next we print data read from an input file using semi-colons:
-----
Line No.  1  The value of x is  23.45  while that of y is  18.27
Line No.  2  The value of x is  45.23  while that of y is  12.87
Line No.  3  The value of x is  67.82  while that of y is  10.32
Line No.  4  The value of x is  88.12  while that of y is   8.25
Line No.  5  The value of x is   6.23  while that of y is  22.18

Repeating printing input data using commas:
Line No.      1          The value of x is          23.45          while that of y is          18.27
Line No.      2          The value of x is          45.23          while that of y is          12.87
Line No.      3          The value of x is          67.82          while that of y is          10.32
Line No.      4          The value of x is          88.12          while that of y is           8.25
Line No.      5          The value of x is           6.23          while that of y is          22.18
-----
This is the end of printing for this file

```

Figure 18. Text file (report style) produced by the program of Figure 17.

### Uses of input and output files in VBA

Many data collection instruments or devices can generate large text files that can be use for input data into VBA programs for, say, statistical analysis. For example, to analyze the temperature data in Salt Lake City, you can download file *UTSALTLK.txt*, available at:

<http://academic.udayton.edu/kissock/http/Weather/g sod95-current/UTSALTLK.txt>

Downloading this text file, and opening in, say, *Notepad++*, you'll find that the data contains four columns corresponding to a month (1 to 12), day (1 to 28, or 1 to 29, or 1 to 30, or 1 to 31), year (from 1995 on), and a temperature value in Fahrenheit degrees. The following figure shows the contents of the text file:

```

C:\Users\Gilberto\Desktop\UTSALTLK.txt - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins
Window ?
UTSALTLK.txt
1 1 1 1995 20.1
2 1 2 1995 21.9
3 1 3 1995 21.8
4 1 4 1995 24.3
5 1 5 1995 29.6
6 1 6 1995 32.6
Ln: 1 Col: 1 Sel: 0|0 Dos\Windows ANSI as UTF-8 INS

```

Figure 19. First few lines of file *UTSALTLK.txt* containing temperature data for Salt Lake City, UT.



Data from this file can be used to produce statistical analysis and data summaries, for example:

- Mean value for the entire period that the data was collected (1995 to date)
- Standard deviation for the entire period that the data was collected
- Mean value per month for each year
- Mean value per month for all years
- Mean value per year
- etc.

VBA programs can be written to open this input file, read the day, month, year, and temperature data, and then summarize the data according to some criteria (e.g., those listed above). Results for the different data summaries can then be written to other output files.

If the number of data points were small, say, 28 to 31 data values corresponding to a single month of temperature data, then you could simply type those data values in a worksheet and read them from there for analysis. However, the file *UTSLATLK.txt* contains more than 6000 data points, in which case, reading out of the original text file would be more convenient.

### **Error Handling**

VBA provides a couple of programming statements or structures aimed at handling selected run-time errors. In Chapter 2 of the class notes it was indicated that errors are typically of three types: (1) *syntax errors*: violations of VBA syntax – these are immediately detected by the editor, and can be corrected right away; (2) *run-time errors*: problems with data or calculations may produce these type of errors (e.g., *division by zero*, *overflow*, *underflow*, etc.); and, (3) *logical errors*: these are errors in the algorithm, or omissions, or mistyping of commands, that do not trigger syntax or run-time errors, but that still produce the wrong results. The latter type of errors are more difficult to find, and typically require some debugging. The *error-handling* structures provided by VBA are to be used to catch predictable run-time errors so that they can be handled in such a way that the program will not crash.

There are two structures to consider:

- *On Error Resume Next* is used to handle run time errors by resuming execution in the next statement following that triggered the error. This approach works fine if skipping a statement that triggers an error would not directly affect the results from other statements.
- *On Error GoTo ErrorHandler* is used to handle run time errors by skipping to an *ErrorHandler* label containing a block of statements to execute if an error is detected. The block of statements typically ends with a *Resume* statement that would send the control to the line following that where the error was detected. This approach would be used if contingency is provided for handling a particular error before the program is allowed to resume. As an alternative to the *Resume* statement, an *Exit Sub* statement could be used that would terminate the program.

The *On Error* statements (whether *On Error Resume Next* or *On Error GoTo ErrorHandler*) are typically located at the top of a *Sub* or *Function*. On the other hand, the *ErrorHandler* label and block of statements ending in *Resume* (or *Exit Sub*), are typically located towards the end of the code, right after the *End Sub* statement.

### Example of *On Error Resume Next*

Figure 20, below, shows a simple example where we calculate the square root of a real number. If the user enters a negative number as argument for the square root (in cell B3), then Excel produces an error as shown in Figure 20. The program stops and an Excel error message is shown indicating that an *Invalid procedure call* was made. The arrow in the code window points to the line  $y = \text{Sqr}(x)$  as the culprit. The user, then, has to press the [OK] button in the warning window to end the procedure.

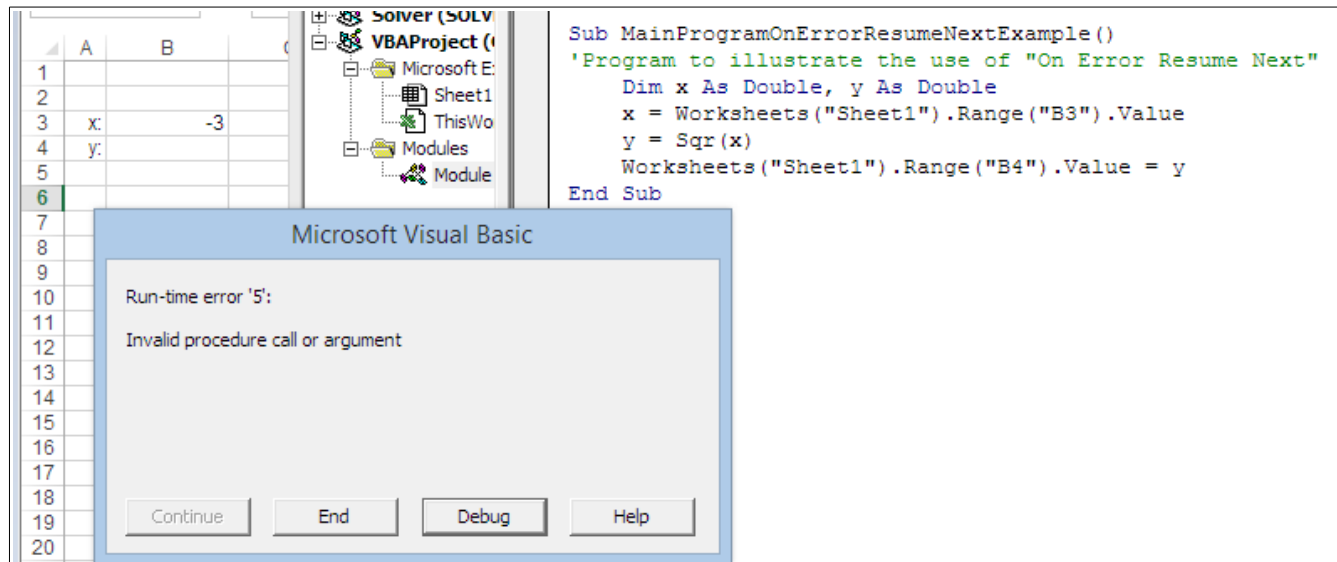


Figure 20. Example of a run-time error generated by a negative argument to the *Sqr* (square root) function.

By adding an *On Error Resume Next* to the top of the program (right after the *Dim* statement), the program skips the error message and gives to variable *y* its default value of zero. This is illustrated in Figure 21, below.

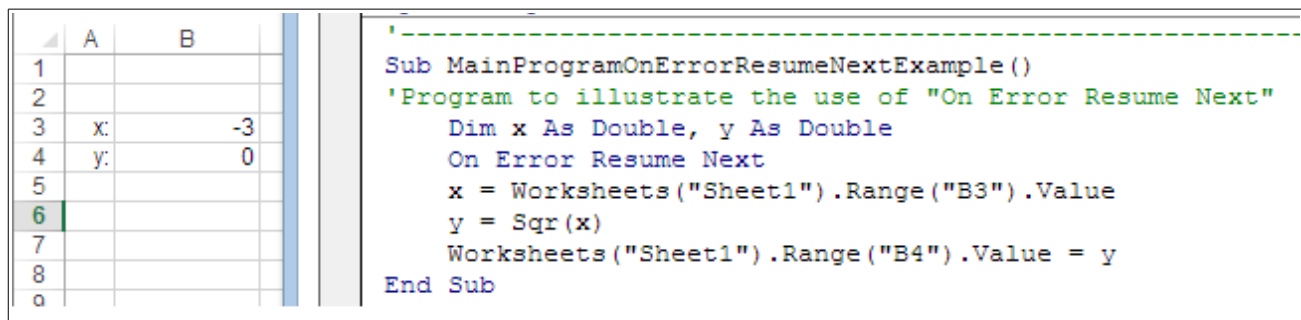


Figure 21. Use of an *On Error Resume Next* to handle the run-time error shown in Figure 20.

### Example of *On Error Resume GoTo ErrorHandler*

The use of *On Error Resume Next* to resolve the run-time error of Figure 20 (as shown in Figure 21) may not be the best way to handle this situation. A better alternative would be to switch the minus sign in cell B3 in Figure 20 or 21 to a plus sign, and recalculate the square root. Figure 22 shows a way to handle this run-time error through an *On Error Resume GoTo ErrorHandler* statement, and an *Error Handler* code segment.

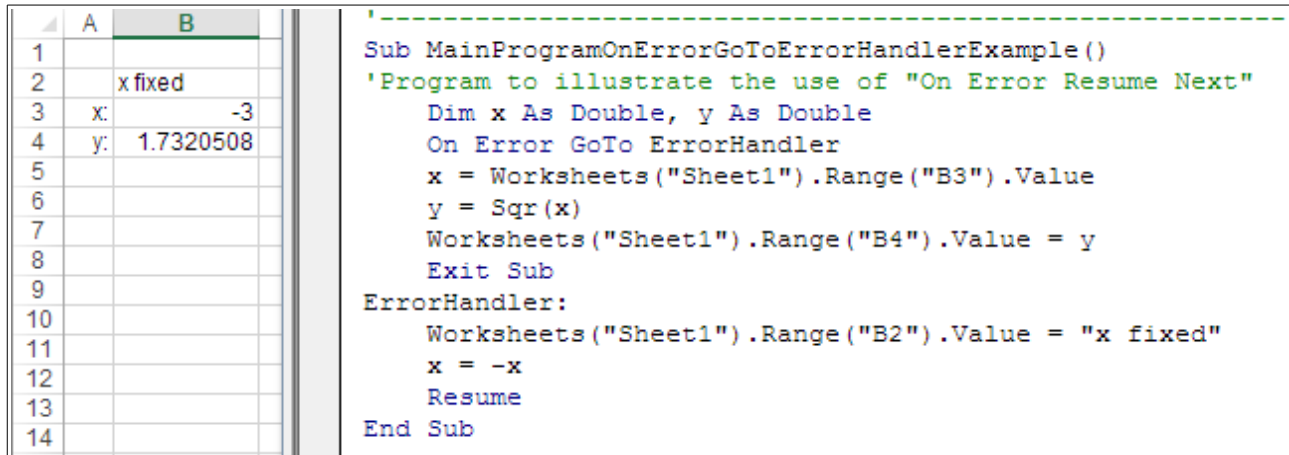


Figure 22. Use of an *On Error GoTo ErrorHandler* to handle the run-time error shown in Figure 20.

The code of Figure 22 uses a label, called *ErrorHandler* to define a segment of code (located between the label *ErrorHandler* and *End Sub*) to handle the error. The actions taken in case an error is detected, as shown in Figure 22, is to write the message “x fixed” in cell B2 of the interface and then change the sign of x ( $x = -x$ ). The *ErrorHandler* code segment ends with a *Resume* statement that would send the program control back to the statement where the error was detected, i.e., back to the statement  $y = \text{Sqr}(x)$ , which now calculates the square root of a positive value. The program control then proceeds to execute the line `Worksheets("Sheet1").Range("B4").Value = y` and skips the remaining code.

Notice that the label for the error-handling code doesn't necessarily has to be *ErrorHandler*. It could be any label name, but the name *ErrorHandler* is descriptive of the purpose of the code, and that is why it was selected for this program.

Figure 23 shows the general operation of the *On Error GoTo ErrorHandler* scheme for error handling:

- (a) If no error is detected, VBA executes the main code and ends execution at the *Exit Sub* statement.
- (b) If an error is detected, control is sent to the *ErrorHandler* label
- (c) If the error is properly resolved, a *Resume* statement sends the control to the line where the error was detected to continue with the main code
- (d) If the error is not properly resolved, the sub ends

The purpose of using *On Error* error-handling statements is to be able to produce code that will not crash if a run-time error is detected. Therefore, if the programmer suspects that a run-time error may occur, he or she can provide a remedy for such a situation using one of the two *On Error* statements described above.

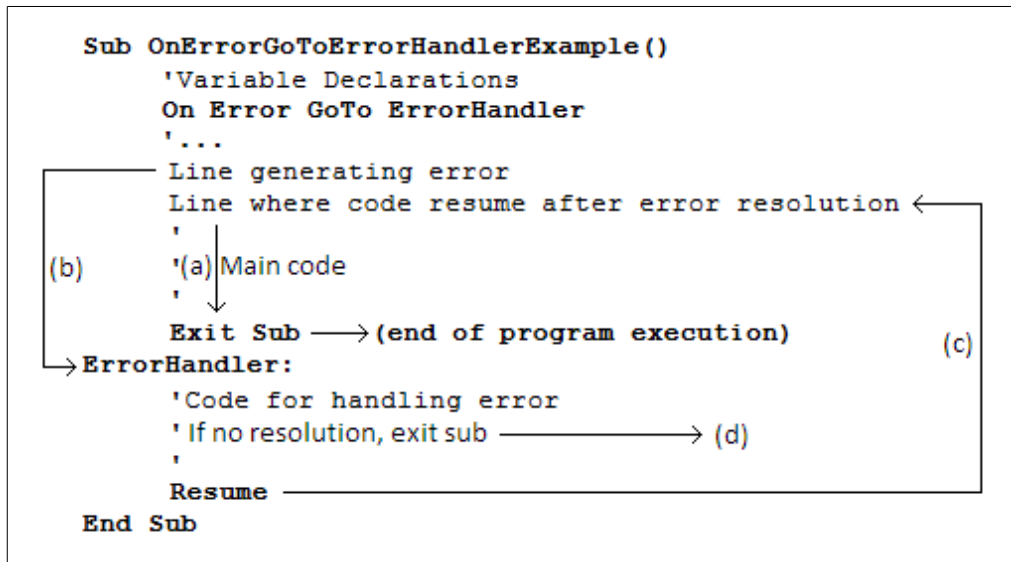


Figure 23. Structure of the *On Error GoTo ErrorHandler* VBA statement for the handling of run-time errors.

### In Summary

In this chapter we introduce the basic use of text files for input and output in VBA, as well as a couple of error-handling structures to deal with predictable run-time errors.