

This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2014. <http://www.swfox.net>



# String Theory: Working with Serial Data in VFP

*Rick Borup  
Information Technology Associates, LLC  
701 Devonshire Dr, Suite 127  
Champaign, IL 61820  
Voice: (217) 359-0918  
Email: rborup@ita-software.com*

*We all know that VFP is great for database apps, but did you know it's also a superb tool for working with strings? In my own work I frequently encounter situations that require importing or exporting data in many different flat file formats. VFP's wide range of string functions and low-level file commands make it easy for developers to handle anything from lowly CSV files to hierarchical XML. Come to this session to learn some tricks and traps based on real-world experiences.*

## Introduction

With its built-in SQL language capabilities and database connectivity tools, Visual FoxPro is great at working with data in relational databases such as native VFP tables, Microsoft SQL Server, and others. However, sometimes it becomes necessary to import data from or export data to other systems where a direct exchange of relational database files is not an option. In those situations, flat files are often the lowest common denominator. If the two sides can agree on a mutually acceptable format, flat files can be used as the bridge between a VFP app on one side and a different system on the other side.

Records in flat files are simply strings in various formats. Thanks to its numerous built-in commands and functions, VFP excels at working with strings, making it easy to parse input from or generate output for virtually any flat file format. This makes VFP an idea tool for importing data from and exporting data to almost any other system.

## File formats

There are many different flat file formats commonly used to store data and to exchange data between diverse systems. In my own work I've had to deal with CSV, SDF, tab-delimited, pipe-delimited, tilde-delimited, asterisk-delimited, XML, and others. Each one is unique but with a little custom code VFP can handle them all.

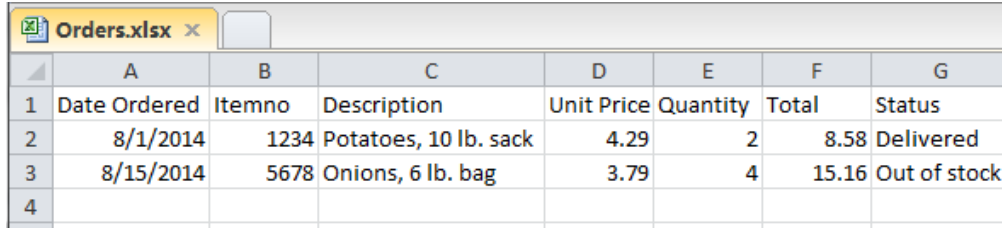
With the exception of XML, each line in a flat file represent one data record. XML is different because it's a hierarchical format rather than a flat file format, but I've included it here because XML data is entirely text-based and each line of an XML file can be treated as a string when writing custom code to parse or generate it.

## CSV

Comma-separated values (CSV) is one of the most common formats. In a standard CSV file, the first line is a header line containing the field names and each subsequent line is a data record whose fields are enclosed in quotes and separated by commas. CSV files are typically stored with a .csv file name extension.

It's been said that the nice thing about standards is that there are so many of them, and that's certainly true of CSV. Some implementations of CSV put quotes around every field while others omit the quotes except for fields that contain embedded commas. Other implementations omit the header line and contain only data rows. When working with CSV, it's incumbent on the developer to find out exactly what the particular CSV data looks like in order to program for it successfully.

One advantage of CSV files is that Excel can open them directly and can save spreadsheets in CSV format. Many people, especially those in non-technical positions, love to use Excel to store their data. While there are lots of issues with Excel data, which I'll discuss later, it's common for someone to say "I'll send you the data in CSV format" meaning they'll export it from Excel. **Figure 1** shows a simple Excel file from which a user might export data.



	A	B	C	D	E	F	G
1	Date Ordered	Itemno	Description	Unit Price	Quantity	Total	Status
2	8/1/2014	1234	Potatoes, 10 lb. sack	4.29	2	8.58	Delivered
3	8/15/2014	5678	Onions, 6 lb. bag	3.79	4	15.16	Out of stock
4							

**Figure 1.** A sample Excel file from which a user might export data.

**Listing 1** is the CSV file generated by Excel from the spreadsheet in Figure 1. Note that the first row contains the field names, and in all rows the fields are separated by commas. It appears that Excel puts quotes only around strings that have embedded commas – note that although both are strings, the description field values are enclosed in quotes while the status field values are not. Also note that the dates are written in mm/dd/yyyy format and the quantity field is written out as an integer with no explicit decimal point.

**Listing 1.** This CSV file was generated from a spreadsheet in Excel 2010.

```
Date Ordered,Itemno,Description,Unit Price,Quantity>Total,Status
8/1/2014,1234,"Potatoes, 10 lb. sack",4.29,2,8.58,Delivered
8/15/2014,5678,"Onions, 6 lb. bag",3.79,4,15.16,Out of stock
```

## Delimited

Another common flat file format is the delimited file, in which fields are separated with some unique character. While a delimited file can theoretically use any character as the field separator—including, for that matter, even a comma as in the CSV format—it's advisable to choose one that isn't likely to appear anywhere in the actual data. The tab character is a frequent choice, but I've also seen files where a pipe character, a tilde, or even an asterisk is used as the field delimiter. Again, VFP makes it easy to work with any of these. Delimited files are typically stored with a .txt file name extension regardless of the character used as the field delimiter.

Unlike CSV files, delimited files typically do not contain a header row and the fields are not normally surrounded by quotes even if they contain an embedded comma. However, tab delimited files exported from Excel do contain the header row and fields with embedded commas are enclosed in quotes. **Listing 2** is the same file shown in Listing 1, but saved from Excel in tab-delimited format rather than in CSV format. For display purposes, the tab characters in Listing 1 are represented by a double-arrow pointing right.

**Listing 2.** In a tab-delimited file, fields are separated with a tab character. This file was generated from an Excel spreadsheet so it has a header row and the description fields are surrounded by quotes.

```
Date Ordered»Itemno»  Description» Unit Price»  Quantity» Total» Status
8/1/2014» 1234»  "Potatoes, 10 lb. sack"»  4.29»  2»  8.58»  Delivered
8/15/2014»  5678»  "Onions, 6 lb. bag"»   3.79»  4» 15.16» Out of stock
```

If this tab-delimited file had been created programmatically instead of from Excel, it would typically not have the header row and the quote marks would be omitted, as shown in **Listing 3**.

**Listing 3.** A tab-delimited file generate programmatically would usually not have a header row and none of the fields would be surrounded by quotes.

```
8/1/2014» 1234» Potatoes, 10 lb. sack» 4.29» 2» 8.58» Delivered
8/15/2014» 5678» Onions, 6 lb. bag» 3.79» 4» 15.16» Out of stock
```

Pipe-delimited, tilde-delimited, asterisk-delimited, or files delimited with any other character are essentially the same as tab-delimited files except for the use of a different character as the field delimiter. VFP's APPEND FROM command natively recognizes BLANK and TAB as field delimiters, but a bit of special handling is required when other delimiters are used – more on that later.

**Listing 4.** The same file in pipe-delimited format.

```
8/1/2014|1234|Potatoes, 10 lb. sack|4.29|2|8.58
8/15/2014|5678|Onions, 6 lb. bag|3.79|4|15.16
```

In a delimited file, empty fields are completely empty. No space character, null, or any other character is used to represent the empty value. This means that when there is an empty field there are two adjacent field delimiters, unless it's the last field in the record. This can cause problems when importing data from a delimited file using VFP's GetWordCount( ) or GetWordNum( ) functions – more on that later, too.

## SDF

System data format (SDF) files have fixed-length records with fixed-length fields within each record. Rather than being separated by some delimiter character, the field boundaries in SDF files are determined by the byte position within the record. SDF files are commonly stored with either a .sdf or a .txt file name extension.

In an SDF file, each field needs to be wide enough to accommodate the largest possible value, meaning that SDF files frequently contain a lot of empty space. Unlike delimited files, all the field boundaries in an SDF file line up when the file is viewed in a text editor, but on the other hand there is no visible boundary between the fields unless there are leading or trailing spaces.

**Listing 5** shows the same file as Listing 4 but in SDF format. The date field is 10 characters wide, the item number is 4, the description is 25, the unit price is 7, the quantity is 3, the total price is 10, and the status is 15. The total length of each line, not counting the end-of-line (EOL) characters, is therefore 74 characters regardless of the length of the actual data in each field. String data, including the dates in this example, are left-justified while numeric values are right-justified.

**Listing 5.** In SDF format, all fields have a fixed size regardless of their content and all the rows in the file are the same length. The first line in this listing is not part of the file, it's just a ruler to help identify the field boundaries.

```
1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75
```

```
8/1/2014 1234Potatoes, 10 lb. sack      4.29  2      8.58Delivered
8/15/2014 5678Onions, 6 lb. bag        3.79  4     15.16Out of stock
```

One example of SDF files in widespread usage is the ACH format used by the banking industry's National Automated Clearing House Association for the exchange of debits and credits among financial institutions.

## XML

Extensible Markup Language (XML) is a hierarchical format rather than a flat file format, but there's no binary data so every line in an XML file can be treated as a string. While VFP has XML-specific functions such as `CURSORTOXML()` and `XMLTOCURSOR()`, the power of these functions is limited and may not work with complex XML files. When working with XML files whose structure is beyond the capabilities of the native XML functions, VFP's low-level file handling and string manipulation functions can be used to create and/or parse these files.

In the detail area of an XML file, each line represents one data field, so it takes multiple lines to represent one row of data. XML files also contain header data and may contain more than one type of data record, such as an order header followed by order detail.

**Listing 6** shows the order history file from in the previous examples the way it might appear in XML format. It's a simple file with only one type of data record called `OrderItem`.

**Listing 6.** The sample order history file might look like this in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<ns0:OrderHistories xmlns:ns0="http://some URL">
  <OrderHistory>
    <OrderItem>
      <DateOrdered>08/01/2014</DateOrdered>
      <Itemno>1234</Itemno>
      <Description>Potatoes, 10 lb. sack</Description>
      <UnitPrice>4.29</UnitPrice>
      <OrderQty>2</OrderQty>
      <ItemTotal>8.58</ItemTotal>
      <Status>Delivered</Status>
    </OrderItem>
    <OrderItem>
      <DateOrdered>08/15/2014</DateOrdered>
      <Itemno>5678</Itemno>
      <Description>Onions, 6 lb. bag</Description>
      <UnitPrice>3.79</UnitPrice>
      <OrderQty>4</OrderQty>
      <ItemTotal>15.16</ItemTotal>
```

```
<Status>Out of stock</Status>
</OrderItem>
</OrderHistory>
</ns0:OrderHistories>
```

## Importing Data

### VFP commands and functions for importing data

The VFP commands and functions for importing data can be grouped into three categories: basic commands, low-level file functions, and XML. The basic commands are APPEND FROM and IMPORT. The low-level file functions are FOPEN( ), FGETS( ), FREAD( ), FEOF( ), and FCLOSE( ). For XML there is XMLTOCURSOR( ). In the following section, each of these is introduced with its description and syntax quoted directly from the VFP Help file<sup>1</sup>, followed by a discussion and examples for some.

#### APPEND FROM Command

*Adds records to the end of the currently selected table from another file.*

```
APPEND FROM FileName | ?[FIELDS FieldList] [FOR lExpression]
  [[TYPE] [DELIMITED [WITH Delimiter | WITH BLANK | WITH TAB
    | WITH CHARACTER Delimiter] | DIF | FW2 | MOD | PDOX | RPD |
    SDF | SYLK | WK1 | WK3 | WKS | WR1 | WRK | CSV | XLS | XL5
    [SHEET cSheetName] | XL8 [SHEET cSheetName]]] [AS nCodePage]
```

The APPEND FROM command is probably the one most frequently used to import data from files in formats other than native VFP tables. As can be seen from its syntax, APPEND FROM is a versatile command that can handle many different file formats, including older but not modern versions of Excel.

When it comes to working with delimited files, the syntax of the APPEND FROM command and its description in the VFP Help file can be confusing. The portion of the syntax definition relating to delimited file is this:

```
DELIMITED [WITH Delimiter | WITH BLANK | WITH TAB | WITH CHARACTER Delimiter]
```

It takes a careful reading to understand how these clauses actually work, because the definition uses the word *delimiter* to refer to both field separators and field delimiters. A *field separator* is the character that separates one field from another, while *field delimiter* refers to the characters used to surround a field.

Consider this line from the tab-delimited file in Listing 3:

```
8/1/2014» 1234» Potatoes, 10 lb. sack» 4.29» 2» 8.58» Delivered
```

---

<sup>1</sup> Taken from the dv\_foxhelp\_vfp9sp2\_v1.07 edition of the Help file from VFPX, dated 6/12/2014.

In this example, the tab character is the field separator but the fields are not surrounded by any delimiters. The pipe-delimited version in Listing 4 is similar:

```
8/1/2014|1234|Potatoes, 10 lb. sack|4.29|2|8.58
```

Contrast these two examples with the first detail line from the CSV file in Listing 1, in which the field separator is a comma and quote marks are used as field delimiters around the description field, which contains an embedded comma.

```
8/1/2014,1234,"Potatoes, 10 lb. sack",4.29,2,8.58,Delivered
```

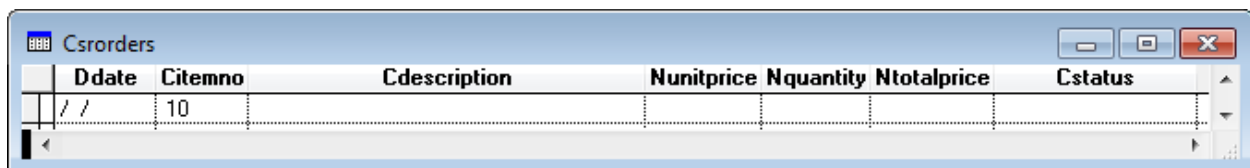
The TAB delimiter is built in to the syntax of VFP's APPEND FROM command, so a tab-delimited file can be imported with

```
APPEND FROM <filename> TYPE DELIMITED WITH TAB
```

The pipe character, on the other hand, is not built in to the syntax of the APPEND FROM command. Looking at the Help file, you might assume (as I did) that you could import a pipe-delimited file this way:

```
APPEND FROM <filename> TYPE DELIMITED WITH "|"
```

It turns out this doesn't work, and the results are completely unexpected:



Ddate	Citemno	Cdescription	Nunitprice	Nquantity	Ntotalprice	Cstatus
8/1/2014	10	Potatoes, 10 lb. sack				Delivered

The correct syntax for importing a pipe-delimited file using APPEND FROM requires using the WITH CHARACTER clause, like this:

```
APPEND FROM <filename> TYPE DELIMITED WITH CHARACTER "|"
```

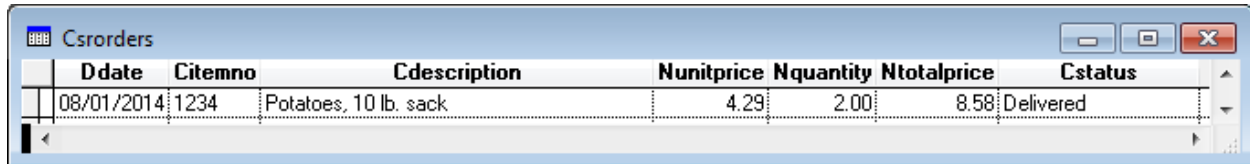
Understanding the difference between field separators and field delimiters makes it easier to understand one of the more obscure examples in the APPEND FROM topic in the VFP Help file, which describes being able to use both DELIMITED WITH and WITH CHARACTER clauses in the same statement. The example is this:

```
APPEND FROM mytxt.txt DELIMITED WITH _ WITH CHARACTER *
```

This describes a file where the field separator is an asterisk ( `*` ) and the field delimiter is the underscore ( `_` ) character. A line from such a file would look like this:

```
_8/1/2014_*_1234_*_Potatoes, 10 lb. sack_*_4.29_*_2_*_8.58_*_Delivered_
```

If this line is stored in a file called mytxt.txt and imported using the above APPEND FROM command, the results are as expected:



Ddate	Citemno	Cdescription	Nunitprice	Nquantity	Ntotalprice	Cstatus
08/01/2014	1234	Potatoes, 10 lb. sack	4.29	2.00	8.58	Delivered

When importing SDF data with the APPEND FROM command, all the columns in the target cursor have to be defined as character data type with a length corresponding to the exact width of their corresponding field in the input file. If different data types are required, as for numbers and dates, one way is to add extra fields of the desired data types to the end of the cursor's data definition and then to fill in the values of those fields after the APPEND FROM is finished.

**Listing 7** illustrates one way this can be done. I used the CTOD( ) function on the date field only for the sake of brevity; its use in real code is discouraged because of possible ambiguity in the date format.

Listing 7. The first seven columns in each row of the cursor are populated with string values from the APPEND FROM command. The last four columns are then populated by the REPLACE statement with the corresponding value of the correct data type for the date and numeric fields.

```
CREATE CURSOR csrOrders ;
( cDate c(10), ;
  cItemno c(4), ;
  cDescription c(25), ;
  cUnitPrice c(7), ;
  cQuantity c(3), ;
  cTotalPrice c(10),
  cStatus c(15), ;
  dDate D, ;
  nUnitPrice N(10,2), ;
  nQuantity N(3,0), ;
  nTotalPrice N(10,2) ;
)
APPEND FROM Orders_sdf.txt TYPE SDF
REPLACE ALL dDate WITH CTOD( cDate), ;
          nUnitPrice WITH VAL( cUnitPrice), ;
          nQuantity WITH VAL( cQuantity), ;
          nTotalPrice WITH VAL( cTotalPrice)
```

## IMPORT Command

*Imports data from an external file format to create a new Visual FoxPro table.*

```
IMPORT FROM FileName [DATABASE DatabaseName [NAME LongTableName]]
[TYPE] FW2 | MOD | PDOX | RPD | WK1 | WK3 | WKS | WR1 | WRK | XLS
| XL5 [SHEET cSheetName] | XL8 [SHEET cSheetName] [AS nCodePage]
```

Unlike the APPEND FROM command, the IMPORT command creates a new table and lacks several of the options available in the former. Because it's limited to only a few types of input files, including only older versions of Excel, I've never found a need to use the IMPORT command in actual practice.



### **FOPEN() Function**

*Opens a file for use with low-level file functions.*

```
FOPEN(cFileName [, nAttribute])
```

The FOPEN( ) function opens an existing file for use with other low-level file functions. By default, the file is opened in read-only mode, but the *nAttribute* parameter can be used to obtain different read/write privileges.

The important thing about the FOPEN( ) function is that it returns a file handle in the form of an integer. All subsequent low-level file functions performed on the file must reference that file handle, so that value needs to be stored in a memory variable or object property where it can be referenced as needed the entire time the file remains open. FOPEN( ) returns -1 if the file cannot be opened for any reason.

```
lnFileHandle = FOPEN( "Orders_tabDelimited.txt")
IF lnFileHandle = -1
    WAIT WINDOW "Failed to open file"
ELSE
    WAIT WINDOW "File is open with handle " + TRANSFORM( lnFileHandle)
ENDIF
```

### **FCLOSE() Function**

*Flushes and closes a file or communication port opened with a low-level file function.*

```
FCLOSE(nFileHandle)
```

The FCLOSE( ) functions closes a previously opened file. It requires the file handle as the parameter. FCLOSE( ) returns True if successful, otherwise it returns False.

### **FEOF() Function**

*Determines whether the file pointer is positioned at the end of a file.*

```
FEOF(nFileHandle)
```

The FEOF( ) function returns True when the end of the input file is reached; until then, it returns False. A typical program tests for FEOF( ) at the top of a loop that reads each line of a file until end of file is reached.

### **FGETS() Function**

*Returns a series of bytes from a file or a communication port opened with a low-level file function until it encounters a carriage return.*

```
FGETS(nFileHandle [, nBytes])
```

FGETS( ) reads bytes from a low-level file and returns them as a string. By default it returns 254 bytes, unless a carriage return is encountered first. If the records in the input file are

longer than 254 bytes, the *nBytes* parameter can be used to specify a larger value up to a maximum of 8192. Records longer than 8192 characters—and yes, I have worked with some—can be handled by a loop that brings in 8192 characters at a time and appends them to a string until end-of-line is reached.

FGETS( ) is useful when working with delimited files because it reads a line at a time regardless of the delimiters being used. It also works well with SDF files, because in both types of files each line is terminated with a carriage return, which terminates the string returned by FGETS( ).

A typical program places the FGETS( ) function inside a loop to read each line of the input file one by one until end of file is reached. **Listing 8** illustrates the use of low-level file functions to open a file, read and display each line, and then close the file.

**Listing 8.** FGETS( ) is typically used inside a loop to read each line of a file until EOF( ) returns True.

```
lnFileHandle = FOPEN( "Orders_tabDelimited.txt")
DO WHILE NOT EOF( lnFileHandle)
    lcString = FGETS( lnFileHandle)
    ?lcString
ENDDO
=FCLOSE( lnFileHandle)
```

If the length of the longest input line is unknown and might be greater than 254, include the *nBytes* parameter and specify a value greater than 254. FGETS( ) stops reading when it encounters a carriage return, so it's safe to use any large value up to the maximum of 8192.

### FREAD() Function

*Returns a specified number of bytes from a file opened with a low-level function.*

```
FREAD(nFileHandle, nBytes)
```

Unlike FGETS( ), the FREAD( ) function does not pay attention to carriage returns or any other type of end of line indicator. FREAD( ) simply returns a specified number of bytes up to a limit of 65,535. FREAD( ) is useful for working with SDF files because all lines in an SDF file are of the same length.

### XMLTOCURSOR() Function

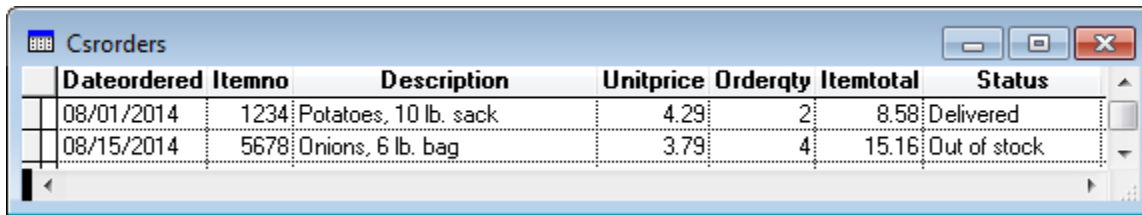
*Converts XML text into a Visual FoxPro cursor or table.*

```
XMLTOCURSOR(eExpression | cXMLFile [, cCursorName [, nFlags ]])
```

The XMLTOCURSOR( ) function reads data from an XML file and converts it to a VFP cursor or table. There are several variations of this function, determined by the value of the *nFlags* parameter. A value of 512 means that the first parameter is the name of an XML file, so the code to read a file named OrderHistory.xml into a cursor called csrOrders looks like this:

```
XMLTOCURSOR( "OrderHistory.xml", "csrOrders", 512)
```

The cursor created when this code is run using the XML file in Listing 6 as the input file is shown in **Figure 2** (see footnote<sup>2</sup>). Note that the cursor's column names are derived from the XML node names.



Dateordered	Itemno	Description	Unitprice	Orderqty	Itemtotal	Status
08/01/2014	1234	Potatoes, 10 lb. sack	4.29	2	8.58	Delivered
08/15/2014	5678	Onions, 6 lb. bag	3.79	4	15.16	Out of stock

**Figure 2.** This cursor was created from the XMLTOCURSOR( ) function.

Although powerful, the XMLTOCURSOR( ) function has limitations. For instance, in the example above, XMLTOCURSOR( ) did not recognize that the first column is a date and created a character field instead.

When XMLTOCURSOR isn't suitable or you want finer grained control, the XMLDOM can be used to read parse the incoming XML file. **Listing 9** illustrates how to use the XMLDOM to read the XML file from Listing 6, parse its fields into individual memory variables, and insert a row into a cursor for each order record. The code first creates the target cursor and can therefore determine the data type of each column.

Listing 9. The XMLDOM can be used to read an XML file and parse out the individual rows and fields.

```
CREATE CURSOR csrOrders ;
(
  dDate D, cItemno C(4), cDescription C(25), nUnitPrice N(7,2), ;
  nQuantity N(3,0), nTotalPrice N(10,2), cStatus C(15) ;
)
loXML = CREATEOBJECT("Microsoft.XMLDOM")
loXML.Async = .F.  && Load entire file before proceeding
loXML.Load( "OrderHistory.xml")
loRoot = loXML.DocumentElement
FOR EACH oElement IN loRoot.ChildNodes
  FOR EACH oOrder IN oElement.ChildNodes
    FOR EACH oItemField IN oOrder.ChildNodes
      lcNodeName = UPPER( oItemField.NodeName)
      DO CASE
        CASE lcNodeName == "DATEORDERED"
          ldDate = CTOD( oItemField.Text)
        CASE lcNodeName == "ITEMNO"
          lcItemno = oItemField.Text
        CASE lcNodeName == "DESCRIPTION"
          lcDescription = oItemField.Text
        CASE lcNodeName == "UNITPRICE"
          lnUnitPrice = VAL( oItemField.Text)
        CASE lcNodeName == "ORDERQTY"
          lnQuantity = VAL( oItemField.Text)
```

---

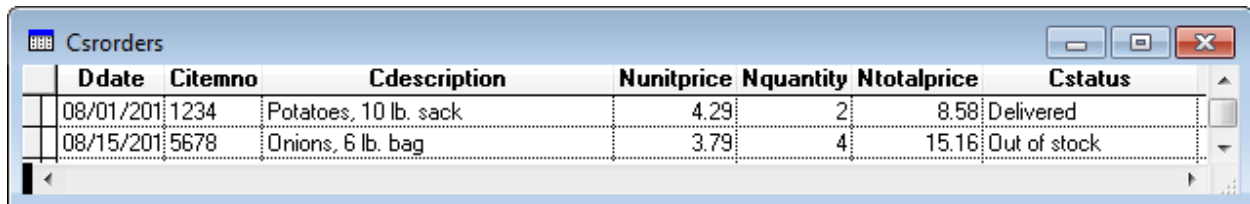
<sup>2</sup> The namespace attribute in Listing 6 refers to a non-existent URL, so that reference was removed to make the XMLTOCURSOR( ) function work.

```

        CASE lcNodeName == "ITEMTOTAL"
            lnItemTotal = VAL( oItemField.Text)
        CASE lcNodeName == "STATUS"
            lcStatus = oItemField.Text
        ENDCASE
    ENDFOR
    INSERT INTO csrOrders VALUES ;
    (
        ldDate, ;
        lcItemno, ;
        lcDescription, ;
        lnUnitPrice, ;
        lnQuantity, ;
        lnItemTotal, ;
        lcStatus ;
    )
ENDFOR
ENDFOR

```

The contents of the cursor come out as expected, as illustrated in **Figure 3**.



Ddate	Citemno	Cdescription	Nunitprice	Nquantity	Ntotalprice	Cstatus
08/01/2011	1234	Potatoes, 10 lb. sack	4.29	2	8.58	Delivered
08/15/2011	5678	Onions, 6 lb. bag	3.79	4	15.16	Out of stock

**Figure 3.** The contents of the cursor after reading and parsing an XML file using the XMLDOM.

When working with XML files with complex schemas or where for any other reason neither XMLTOCURSOR nor the XMLDOM are workable solutions, low-level file functions and string parsing functions may be the answer. Because XML elements are enclosed in delimiters, the STREXTRACT() function is a good choice. **Listing 10** shows how this could be done. Written in procedural style with all those nested loops, this is admittedly an inelegant piece of code but it serves to illustrate the technique. The resulting cursor is identical to that shown in Figure 3.

**Listing 10.** The STREXTRACT() function is a good choice for parsing XML files.

```

CREATE CURSOR csrOrders ;
(
    dDate D, cItemno C(4), cDescription C(25), nUnitPrice N(7,2), ;
    nQuantity N(3,0), nTotalPrice N(10,2), cStatus C(15) ;
)
lnFileHandle = FOPEN( "OrderHistory.xml")
WAIT WINDOW "lnFileHandle = " + TRANSFORM( lnFileHandle)
IF lnFileHandle = -1
    RETURN
ENDIF
DO WHILE NOT FEOF( lnFileHandle)
    lcString = ALLTRIM( FGETS( lnFileHandle))
    DO CASE
        CASE INLIST( LEFT( lcString, 5), "<?xml", "<ns0:")
            * ignore

```

```

CASE lcString = "<OrderHistory>"
DO WHILE NOT FEOF( lnFileHandle)
    lcString = ALLTRIM( FGETS( lnFileHandle))
    IF lcString = "<OrderItem>"
        LOOP
    ENDIF
DO WHILE NOT FEOF( lnFileHandle)
    IF lcString = "</OrderItem>"
        INSERT INTO csrOrders VALUES ;
            (   ldDate, ;
                lcItemno, ;
                lcDescription, ;
                lnUnitPrice, ;
                lnQuantity, ;
                lnItemTotal, ;
                lcStatus ;
            )
        EXIT
    ELSE
        lcLeftDelim = STREXTRACT( lcString, "<", ">", 1, 4)
        lcRightDelim = STRTRAN( lcLeftDelim, "<", "</")
        lcNodeName = UPPER( STREXTRACT( lcLeftDelim, "<", ">", 1))
        DO CASE
            CASE lcNodeName == "DATEORDERED"
                ldDate = CTOD( STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1))
            CASE lcNodeName == "ITEMNO"
                lcItemno = STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1)
            CASE lcNodeName == "DESCRIPTION"
                lcDescription = STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1)
            CASE lcNodeName == "UNITPRICE"
                lnUnitPrice = VAL( STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1))
            CASE lcNodeName == "ORDERQTY"
                lnQuantity = VAL( STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1))
            CASE lcNodeName == "ITEMTOTAL"
                lnItemTotal = VAL( STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1))
            CASE lcNodeName == "STATUS"
                lcStatus = STREXTRACT( lcString, lcLeftDelim,
lcRightDelim, 1)
        ENDCASE
        lcString = ALLTRIM( FGETS( lnFileHandle))
    ENDIF
ENDDO
ENDDO
ENDCASE
ENDDO
FCLOSE( lnFileHandle)

```

## VFP commands and functions for parsing string data

### SUBSTR() Function

*Returns a character string from the given character expression.*

SUBSTR(cExpression, nStartPosition [, nCharactersReturned])

For importing data from flat files, the SUBSTR( ) function is most useful when working with SDF data where the fields have fixed lengths and fixed positions within each record. **Listing 11** creates a VFP cursor from the sample order history file in SDF format.

Listing 11. The SUBSTR( ) function is ideal for working with SDF data.

```
CREATE CURSOR csrOrders ;
  ( dDate D, cItemno c(4), cDescription c(25), nUnitPrice n(7,2), ;
    nQuantity n(3,0), nTotalPrice n(10,2), cStatus c(15))
lnFileHandle = FOPEN( "Orders_SDF.txt")
DO WHILE NOT EOF( lnFileHandle)
  lcString = FGETS( lnFileHandle)
  ldDate = CTOD( SUBSTR( lcString, 1, 10))
  lcItemNo = SUBSTR( lcString, 11, 4)
  lcDescription = SUBSTR( lcString, 15, 25)
  lnUnitPrice = VAL( SUBSTR( lcString, 40, 7))
  lnQuantity = VAL( SUBSTR( lcString, 47, 3))
  lnTotalPrice = VAL( SUBSTR( lcString, 50, 10))
  lcStatus = SUBSTR( lcString, 60, 15)
  INSERT INTO csrOrders ;
    ( dDate, ;
      cItemno, ;
      cDescription, ;
      nUnitPrice, ;
      nQuantity, ;
      nTotalPrice, ;
      cStatus ;
    ) ;
  VALUES ;
  ( ldDate, ;
    lcItemno, ;
    lcDescription, ;
    lnUnitPrice, ;
    lnQuantity, ;
    lnTotalPrice, ;
    lcStatus ;
  )
ENDDO && WHILE NOT EOF()
=FCLOSE( lnFileHandle)
```

### STREXTRACT( ) Function

*Retrieves a string between two delimiters.*

STREXTRACT(cSearchExpression, cBeginDelim [, cEndDelim [, nOccurrence  
[, nFlag]])

The STREXTRACT( ) function is useful when working with data whose fields are delimited in some way. There is more than one way to parse individual fields out of a delimited string, but STREXTRACT( ) is particularly well suited to working with XML files where the delimiters are different for every field, as shown in Listing 10.

### **STRTRAN( ) Function**

*Searches a character expression for a second character expression and replaces each occurrence with a third character expression*

```
STRTRAN(cSearched, cExpressionSought [, cReplacement  
[, nStartOccurrence [, nNumberOfOccurrences [, nFlags]]]])
```

The STRTRAN( ) function is probably most often used when exporting data, but it can be useful when importing data as well. Either way, its purpose is to replace one character expression with another one, for example to replace double quotes with single quotes.

```
lcString = ["Fox Rocks!"]      && String contains double quotes  
?STRTRAN( lcString, ["", ['])  && Displays 'Fox Rocks!'
```

The replacement expression can be the empty string, so STRTRAN( ) can also be used to remove characters from a string, for example to remove embedded apostrophes from fields in situations where they might be interpreted as field delimiters.

```
lcString = [Mrs. O'Leary's cow] && String has embedded apostrophes  
?STRTRAN( lcString, [''], [])  && Displays Mrs. OLearys cow
```

In Listing 10, STRTRAN( ) was used as a convenient way to create the right XML delimiter from the corresponding left delimiter by replacing the < character with </.

```
lcRightDelim = STRTRAN( lcLeftDelim, "<", "</")  && <foo> becomes </foo>
```

### **GETWORDCOUNT( ) Function**

*Counts the words in a string.*

```
GetWordCount(cString[, cDelimiters])
```

When I first began working with imported string data I explored the VFP Help file looking for commands and functions to help parse the individual fields out of each incoming record. The GETWORDCOUNT( ) function and its companion GETWORDNUM( ) seemed like ideal choices, and to some extent they were.

GETWORDCOUNT( ) returns a numeric value which is the number of words, or fields, in the string based on the specified delimiter. That value can be used to verify that an imported data record contains the expected number of fields and also as the terminating value in a FOR ... ENDFOR loop when parsing out the individual fields.

The problem with this function arises when there are empty fields in the string. If the empty field is an empty string, as is generally the case, the line has two consecutive

delimiters and GETWORDCOUNT( ) fails to include the missing field. For example, if each line in a delimited data file has four fields but in there is an empty field in one row, GETWORDCOUNT( ) returns 3 instead of 4 for that row. I discussed this in some detail in my 2009 paper on *Quibbles, Quirks, and Quickies*, which is available for download from my website [1].

### GETWORDNUM( ) Function

*Returns a specific word from a string.*

```
GetWordNum(cString, nIndex[, cDelimiters])
```

The GETWORDNUM( ) function is similar to GETWORDCOUNT( ) except it returns the word, or field, located in the specified ordinal position within a string. Unfortunately, it has the same problem with empty fields that GETWORDCOUNT( ) does, so if you request the third field from a string where the third field is empty, GETWORDNUM( ) returns the fourth field instead of an empty string.

**Listing 12** demonstrates the problem these two functions have with empty fields.

**Listing 12.** GETWORDCOUNT( ) and GETWORDNUM( ) return unexpected results if a field is empty.

```
lcString = "ALFKI,Anders,Maria,567.89"  && string has four fields
?GETWORDCOUNT( lcString, ",")         && returns 4
?GETWORDNUM( lcString, 3, ",")          && returns "Maria"

lcString = "ALFKI,Anders,,567.89"       && field 3 is now empty
?GETWORDCOUNT( lcString, ",")         && returns 3
?GETWORDNUM( lcString, 3, ",")          && returns 567.89
```

There are a couple of workarounds for this problem, which I discuss in the previously cited paper [1], but my preference these days is to use ALINES( ) instead.

### ALINES( ) Function

*Copies each line in a character expression or memo field to a corresponding row in an array.*

```
ALINES(ArrayName, cExpression [, nFlags] [, cParseChar [, cParseChar2 [, ...]]])
```

Given a data record in the form of a string where the fields are separated by some delimiter, ALINES( ) parses the string into an array where each row of the array is one field from the string. The string is specified by the *cExpression* parameter and the delimiter is specified by *cParseChar*.

By default, ALINES( ) includes empty fields in the array so it does not suffer from the same problems as GETWORDCOUNT( ) and GETWORDNUM( ), but the *nFlags* parameter can be used to modify this and other behaviors. ALINES( ) returns the number of rows in the array, which corresponds to the number of fields in the string. **Listing 13** illustrates the use of ALINES( ) and shows how it handles empty fields correctly.



**Listing 13.** The ALINES() function handles empty fields correctly.

```
lcString = "ALFKI,Anders,Maria,567.89"  && four fields
lnCount = ALINES( laLines, lcString, 2, ",")
?lnCount      && returns 4
?laLines[1]   && "ALFKI"
?laLines[2]   && "Anders"
?laLines[3]   && "Maria"
?laLines[4]   && 567.89

lcString = "ALFKI,Anders,,567.89"  && four fields but the third is empty
lnCount = ALINES( laLines, lcString, 2, ",")
?lnCount  && still returns 4
?laLines[1] && "ALFKI"
?laLines[2] && "Anders"
?laLines[3] &&
?laLines[4] && 567.89
```

### TEXT ... ENDTEXT Command

*Sends lines of text specified by TextLines to the current output device or memory variable.*

```
TEXT [TO VarName [ADDITIVE] [TEXTMERGE] [NOSHOW]
      [FLAGS nValue] [PRETEXT eExpression]]
      TextLines
ENDTEXT
```

Coupled with TEXTMERGE and its ability to send its output to a memory variable, the TEXT ... ENDTEXT command is a powerful tool for creating data strings in many formats. While commonly used to programmatically build SQL statements for the SQLEXEC() function, it's also a great way to create a flat file from data in a VFP table or cursor. **Listing 14** is an example of creating a tab-delimited file from a cursor.

**Listing 14.** The TEXT ... ENDTEXT command makes it easy to create flat files from VFP data.

```
* Create a VFP cursor to use as input
CREATE CURSOR csrOrders ;
  ( dDate D, cItemno c(4), cDescription c(25), nUnitPrice n(7,2), ;
    nQuantity n(3,0), nTotalPrice n(10,2), cStatus c(15))
INSERT INTO csrOrders VALUES ;
  ({^2014-08-01}, "1234", "Potatoes, 10 lb. sack", 4.29, 2, 8.58, "Delivered")
INSERT INTO csrOrders VALUES ;
  ({^2014-08-15}, "5678", "Onions, 6 lb. bag", 3.79, 4, 15.16, "Out of stock")

* Create a tab delimited file from the cursor
lcDelimiter = CHR(9)  && tab
lnFileHandle = FCREATE( "orders.txt")
SCAN
  TEXT TO lcString TEXTMERGE NOSHOW PRETEXT 15
    <<DTOC( dDate) + lcDelimiter>>
    <<cItemno + lcDelimiter>>
    <<ALLTRIM( cDescription) + lcDelimiter>>
    <<TRANSFORM( nUnitPrice) + lcDelimiter>>
```

```
<<TRANSFORM( nQuantity) + lcDelimiter>>  
<<TRANSFORM( nTotalPrice) + lcDelimiter>>  
<<ALLTRIM( cStatus)>>  
ENDTEXT  
FPUTS( lnFileHandle, lcString)  
ENDSCAN  
FCLOSE( lnFileHandle)
```

The file created by this code is identical to the file in Listing 3, with one minor exception: each field other than the first is preceded by a space character following the tab separator. It appears the space is being added by function because each field is written on a separate line, even though PRETEXT 15 has been specified to eliminate spaces, tabs, carriage returns, and line feeds between each line.

One way to get rid of the space that precedes each field is to combine all seven lines between TEXT and ENDTEXT into a single line, but that approach doesn't scale well to a file with many more fields. Another way to eliminate the leading space is to STRTRAN( ) each line after ENDTEXT but before the FPUTS( ) statement, like this:

```
lcString = STRTRAN( lcString, CHR(9) + CHR(32), CHR(9))
```

### TEXTMERGE() Function

*Provides evaluation of a character expression.*

```
TEXTMERGE(cExpression [, lRecursive [, cLeftDelim [, cRightDelim]]])
```

The TEXTMERGE( ) function is similar to the EVALUATE( ) function in that it evaluates a string. The difference is that TEXTMERGE( ) uses delimiters. Listing 15 illustrates the difference between these two functions using the default double left and right angle bracket delimiters.

**Listing 15.** The TEXTMERGE( ) function is similar to the EVALUATE( ) function.

```
lcFoo = "foo"  
?EVALUATE( "lcFoo")    && displays foo  
  
lcString = "<<lcFoo>>"  
?EVALUATE( "lcString") && displays <<lcFoo>>  
?TEXTMERGE( lcString)  && displays foo
```

### Line breaks

The end of each line in a text file is defined by a line break character or characters, also known as end-of-line or EOL. Line breaks are typically denoted by a carriage return and/or a line feed character. In the ASCII character set, a carriage return (CR) has a hexadecimal value of 0x0D and a line feed (LF) is hex 0x0A. In VFP, these values correspond to CHR(13) and CHR(10) respectively.

Different operating systems use different EOL characters. Windows uses both CR and LF, in that order, while Unix uses only LF and Mac uses only CR. **Figure 4** shows the difference when the same file is stored in each of these three different formats.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	0123456789ABCDE
00000000	54	68	69	73	20	69	73	20	6C	69	6E	65	20	31	0D	This is line 1.
0000000F	54	68	69	73	20	69	73	20	6C	69	6E	65	20	32	0D	This is line 2.
0000001E																

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	0123456789ABCDE
00000000	54	68	69	73	20	69	73	20	6C	69	6E	65	20	31	0A	This is line 1.
0000000F	54	68	69	73	20	69	73	20	6C	69	6E	65	20	32	0A	This is line 2.
0000001E																

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	54	68	69	73	20	69	73	20	6C	69	6E	65	20	31	0D	0A	This is line 1..
00000010	54	68	69	73	20	69	73	20	6C	69	6E	65	20	32	0D	0A	This is line 2..
00000020																	

**Figure 4.** End of line is denoted with a CR in the Mac operating system (top), with LF in Unix (middle), and with both CR and LF in DOS/Windows (bottom).

Fortunately, VFP is versatile: the APPEND FROM <file name> TYPE DELIMITED command works on all three types of files, so you don't need to be concerned with this when importing data. However, when exporting data you may need to be sure to use the appropriate EOL character(s) if the target machine is running a different OS.

## Dates and numbers

A string is a string, but dates and numbers can be expressed in a variety of ways regardless of the file format.

### Dates

In the United States, the most common way of writing a date is mm/dd/yyyy, as in 09/17/2014. In some files dates are actually stored in this format, for example in CSV and tab-delimited file created by Excel. Other popular choices for storing dates in a file are non-delimited strings in the format mmddyyyy or yyyyymmdd. Sometimes dates are written as mm-dd-yyyy, using dashes instead of slashes.

When working with flat files that contain date fields, be sure to find out which format is being used. VFP can import dates directly if they're stored in certain formats; otherwise, some manipulation may be required. **Figure 5** shows the result of importing dates in various formats in a date field in a cursor.

	Source	Format	Result	OK
	08/15/2014	mm/dd/yyyy	08/15/2014	Yes
	08-15-2014	mm-dd-yyyy	08/15/2014	Yes
	20140815	yyyymmdd	08/15/2014	Yes
	08152014	mmddyyyy	/ /	No

**Figure 5.** VFP recognizes many but not all date formats as actual dates.

## Numbers

Like dates, numbers can be stored in many ways in a flat file. Sometimes they may contain an explicit decimal point while other times the decimal point may be inferred. In some files numeric values may be stored with leading zeros but in other files right-justified with leading spaces or even left-justified with trailing spaces. I've even seen files where numeric values greater than 999 were stored with explicit commas. VFP can import some of these formats directly into numeric data type fields, while other formats require special handling.

Consider a delimited file with five lines, each expressing the same numeric value in one of five different ways as shown in **Listing 16**. You may argue that the fourth value of "0123456789" is not the same as the others because the others have an explicit decimal point, but it's not uncommon to find numeric values in flat files stored this way. This is common in SDF file in which numeric fields have an implied decimal point that is defined in the format specification but there is no explicit decimal point in the actual file.

**Listing 16.** This file has the same numeric value stored in five different ways.

```
1234567.89
00001234567.89
1,234,567.89
0123456789
$1234567.89
```

When imported into a numeric field in a VFP table using APPEND FROM, the result varies depending on the way the number is stored in the flat file. **Figure 6** shows the results in a VFP table for the values in Listing 16.

Nfield1
1234567.89
1234567.89
1.00
123456789
0.00

**Figure 6.** The value imported into a VFP table using APPEND FROM varies depending on how the number is stored in the flat file.

In the first two rows the value comes out as expected, but the third, fourth, and fifth rows have problems. The third row is wrong because VFP stopped converting to a numeric value as soon as the first comma in "1,234,567,89" was reached, resulting a value of "1.00". The

fourth row is wrong because there is not explicit decimal point, meaning the developer needs to run code after importing and divide by 100 to get the correct value. The fifth row is wrong because the leading dollar sign is not a numeric character.

### Monetary values

Monetary values are numeric values with the added complication that in some flat files the currency symbol may be included, as shown in Figure 6. If a monetary value is stored with a leading dollar sign, for example as \$1234567.89, that value ends up as 0.00 when imported into a numeric field using APPEND FROM.

The VAL( ) function, however, does recognize a currency symbol and returns a proper value with a data type of currency as long as there are no embedded commas in the string.

```
VAL( "$1234567.89")          && 1234567.8900
VARTYPE( VAL( "$1234567.89")) && Y (currency)
VAL( "$1,234,567.89")        && 1.00
```

## Exporting Data

### VFP commands and functions for exporting data

Like those for importing data, the VFP commands and functions for exporting data can be also be grouped into basic commands, low-level file functions, and XML.

#### COPY TO Command

*Creates a new file from the contents of the currently selected table.*

```
COPY TO FileName [DATABASE DatabaseName [NAME LongTableName]]
    [FIELDS FieldList | FIELDS LIKE Skeleton | FIELDS EXCEPT Skeleton]
    [Scope] [FOR lExpression1] [WHILE lExpression2]
    [ [WITH] CDX ] | [ [WITH] PRODUCTION ] [NOOPTIMIZE]
    [ [TYPE] [ FOXPLUS | FOX2X | DIF | MOD | SDF | SYLK | WK1 | WKS | WR1
    | WRK | CSV | XLS | XL5 | DELIMITED [ WITH Delimiter | WITH BLANK
    | WITH TAB | WITH CHARACTER Delimiter ] ] ] [AS nCodePage]
```

The COPY TO command is the counterpart to APPEND FROM. Both have an extensive list of built-in file types, including CSV, both have the DELIMITED clause that natively recognizes BLANK and TAB, and both can use other delimiters as well.

When using the COPY TO function to export VFP data to CSV, the column names in the table or cursor become the field names in the header row of the CSV file, as shown in **Listing 17**.

**Listing 17.** This CSV file was created by the COPY TO command.

```
ddate,citemno,cdescripti,nunitprice,nquantity,ntotalpric,cstatus
08/01/2014,"1234","Potatoes, 10 lb. sack",4.29,2,8.58,"Delivered"
08/15/2014,"5678","Onions, 6 lb. bag",3.79,4,15.16,"Out of stock"
```

If the column names in the VFP table are cryptic or abbreviated, or even if they just have a leading character to denote the data type, the names in the header row of the CSV file may not be meaningful to the end user. In that case, a query can be run to create a cursor with more appropriate column names, or the CSV file could be built with low-level file functions for greater control over the output.

A tab-delimited file created by COPY TO comes out as expected, but, as with the APPEND FROM command, care must be taken when using other delimiters. A pipe-delimited file created by the code

```
COPY TO orderHistory_pipe.txt TYPE DELIMITED WITH "|"
```

doesn't come out quite as expected. The COPY TO command interprets the pipe character as a field delimiter instead of a field separator and inserts a comma as the field separator, with the following result:

```
08/01/2014,|1234|,|Potatoes, 10 lb. sack|,4.29,2,8.58,|Delivered|
08/15/2014,|5678|,|Onions, 6 lb. bag|,3.79,4,15.16,|Out of stock|
```

Adding the CHARACTER attribute

```
COPY TO orderHistory_pipe.txt TYPE DELIMITED WITH CHARACTER "|"
```

yields a file closer to the desired result, but there are still extraneous quotes around the string fields.

```
08/01/2014|"1234"|"Potatoes, 10 lb. sack"|4.29|2|8.58|"Delivered"
08/15/2014|"5678"|"Onions, 6 lb. bag"|3.79|4|15.16|"Out of stock"
```

To get a file without the quotes, use the syntax

```
COPY TO orderHistory_pipe.txt TYPE DELIMITED WITH "" WITH CHARACTER "|"
```

The resulting file is now exactly what's desired:

```
08/01/2014|1234|Potatoes, 10 lb. sack|4.29|2|8.58|Delivered
08/15/2014|5678|Onions, 6 lb. bag|3.79|4|15.16|Out of stock
```

### EXPORT Command

*Copies data from a Visual FoxPro table to a file in a different format.*

```
EXPORT TO FileName [TYPE]
    DIF | MOD | SYLK | WK1 | WKS | WR1 | WRK | XLS | XL5
    [FIELDS FieldList] [Scope] [FOR lExpression1] [WHILE lExpression2]
    [NOOPTIMIZE] [AS nCodePage]
```

In the same way the IMPORT command is kind of a limited version of APPEND FROM, the EXPORT command is a limited version of COPY TO. The EXPORT command can create files in one of several formats, including older versions of Excel, but it cannot create the types of

delimited files commonly used today. For that reason, I've never found a use for it in a real application.

### **FCREATE() Function**

*Creates and opens a low-level file.*

```
FCREATE(cFileName [, nFileAttribute])
```

The FCREATE( ) function creates a low-level file and opens it for use. Like FOPEN( ), it returns a numeric file handle, or returns -1 if the file cannot be created. The *nFileAttribute* parameter can be used to set file attributes such as read-only, hidden, and system. The file handle returned by FCREATE( ) needs to be captured and stored in a memory variable or property so it can be used by other low-level file functions that need to access the file.

```
lnFileHandle = FCREATE( "myFile.txt")
```

### **FPUTS() Function**

*Writes a character string, carriage return, and line feed to a file opened with a low-level file function.*

```
FPUTS(nFileHandle, cExpression [, nCharactersWritten])
```

The FPUTS( ) function is the output equivalent of FGETS( ). It writes a string to the file referenced by *nFileHandle* and automatically terminates the line with a carriage return and line feed. This makes it ideal for use in a loop when generating a flat file, as illustrated in **Listing 18** where it's used to generate a pipe-delimited file.

Listing 18. The FPUTS( ) function writes a string to a low-level file and automatically adds a carriage return and line feed.

```
USE OrderHistory
lnFileHandle = FCREATE( "orderHistory.txt")
lcDelimiter = "|"
SCAN
    lcString = DTOC( dDate) + lcDelimiter
    lcString = lcString + OrderHistory.cItemno + lcDelimiter
    lcString = lcString + OrderHistory.cDescription + lcDelimiter
    lcString = lcString + TRANSFORM(OrderHistory.nUnitPrice) + lcDelimiter
    lcString = lcString + TRANSFORM(OrderHistory.nQuantity) + lcDelimiter
    lcString = lcString + TRANSFORM(OrderHistory.nTotalPrice) + lcDelimiter
    lcString = lcString + OrderHistory.cStatus
    FPUTS( lnFileHandle, lcString)
ENDSCAN
FCLOSE( lnFileHandle)
```

By default, the length of the string fields in the output file – the item number, description, and status fields in this example – is the full width of the corresponding column in the VFP table. Trailing spaces are significant in SDF files but not in delimited files, so it's customary to trim the string fields before writing them to the file.

### **FWRITE() Function**

*Writes a character string to a file opened with a low-level file function.*

```
FWRITE(nFileHandle, cExpression [, nCharactersWritten])
```

Unlike FPUTS( ), the FWRITE( ) function does not automatically include a carriage return and line feed. This makes FWRITE( ) suitable for creating files targeted for Mac or Unix machines, where it may be necessary to use a different end-of-line character than is used on Windows. The *nCharactersWritten* parameter can be used to control how many bytes of the string specified in *cExpression* are written to the file.

### **FFLUSH() Function**

*Flushes to disk a file opened with a low-level function.*

```
FFLUSH(nFileHandle [, lForce])
```

FFLUSH( ) flushes the output buffer for a low-level file function to disk and clears the memory space used by the buffer. FCLOSE( ) also flushes the buffer, so unless working with huge strings and limited memory it's generally not necessary to explicitly use FFLUSH( ).

### **CURSORTOXML() Function**

*Converts a Visual FoxPro cursor to XML.*

```
CURSORTOXML(nWorkArea | cTableAlias, cOutput [, nOutputFormat  
[, nFlags [, nRecords [, cSchemaName [, cSchemaLocation [, cNameSpace ]]]]])
```

The CURSORTOXML( ) function converts a VFP table or cursor to XML format. Parameters determine the XML output format, whether it's written to a memory variable or to a file, and several other options.

The most common output format in my experience is the default element-centric format, which is specified with a value of 1 for the *nOutputFormat* parameter. A value of 512 in the *nFlags* parameter means write the output to the file whose name is specified by *cOutput*. An example is shown in **Listing 19**.

**Listing 19.** The CURSORTOXML( ) function creates an XML file from a VFP table or cursor.

```
CURSORTOXML( "orderHistory", "orderHistory.xml", 1, 512)
```

**Listing 20** is the XML generated by this code. Note that the top-level node is named *VFPData* and the field names are the same as the column names in the input table.

**Listing 20.** The top-level node in XML generated by the CURSORTOXML( ) function is named VFPData.

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>  
<VFPData>  
  <orderhistory>  
    <ddate>2014-08-01</ddate>
```



```
<itemno>1234</itemno>
<descripti>Potatoes, 10 lb. sack</descripti>
<nunitprice>4.29</nunitprice>
<nquantity>2</nquantity>
<ntotalpric>8.58</ntotalpric>
<cstatus>Delivered</cstatus>
</orderhistory>
<orderhistory>
  <ddate>2014-08-15</ddate>
  <itemno>5678</itemno>
  <descripti>Onions, 6 lb. bag</descripti>
  <nunitprice>3.79</nunitprice>
  <nquantity>4</nquantity>
  <ntotalpric>15.16</ntotalpric>
  <cstatus>Out of stock</cstatus>
</orderhistory>
</VFPData>
```

The CURSORTOXML( ) function is useful as far as it goes, but I've run into situations where more complex XML needs to be created. In every case, I've been able to write custom code with FCREATE( ) and FPUTS( ) to meet the requirement. The code can be lengthy when the XML schema is complex and the hierarchy is deeply nested, but it's not difficult, just tedious.

One additional thing to mention when generating XML is that in order to qualify as legal XML, certain characters such as ampersands, apostrophes, and quote marks need to be encoded. This is commonly referred to as entity encoding. **Listing 21** is a little method that uses STRTRAN( ) to accomplish this. In my apps, it's part of a utility class with other related support functions.

**Listing 21.** Certain characters need to be entity encoded to generate proper XML.

```
*-----
*   Utility :: EntityEncode
*-----
*   Function: Replace invalid XML characters with their entity-encoded
*             equivalents.
*   Pass: tcString - the string to encode
*   Return: Character
*   Comments:
*-----
FUNCTION EntityEncode( tcString as String) as String
IF VARTYPE( tcString) <> "C"
  RETURN ""
ENDIF
LOCAL lcString
lcString = tcString
lcString = STRTRAN( lcString, [&], [&amp;])
lcString = STRTRAN( lcString, [''], [&apos;])
lcString = STRTRAN( lcString, ["], [&quot;])
lcString = STRTRAN( lcString, [>], [&gt;])
lcString = STRTRAN( lcString, [<], [&lt;])
```

```
RETURN lcString  
ENDFUNC && EntityEncode
```

### Why Excel is not data

W. C. Fields, known to have a fondness for alcoholic beverages, is alleged to have said “I don’t drink water because fish [muck] in it.” Whether that’s true or not, similar advice applies to developers: don’t rely on Excel data because users [muck] with it.

In my experience, users – especially non-technical users – love Excel. This stands to reason because Excel is a powerful product that’s relatively easy to learn and use by almost anyone. When the source for data to be used in a VFP app is an Excel spreadsheet, it’s not difficult to design an interface to read that data either in actual workbook format (.xls or .xlsx) or in CSV or tab-delimited format in a file exported from Excel.

Regardless of the incoming file format, once an interface has been designed and programmed the app depends on the data always being in exactly that format. The primary problem with Excel data is that users who have full control over their spreadsheets love to make changes for their own convenience. When done correctly these changes don’t break the spreadsheet so everything’s fine from the user’s point of view, but even a minor change can wreak havoc with the programming interface.

Users are very creative. Some of the things I’ve found they like to do with their Excel spreadsheets include:

- adding or removing header rows
- moving columns around
- inserting new columns
- adding subtotal lines between detail lines, and
- using *Alt+Enter* to create a line break in a cell containing text

All of these changes can potentially break a programmatic data interface that relies on there being either no or at least a fixed number of header rows, data fields always being in the same columns, records always having the same number of fields, the detail portion containing only detail records, and not prematurely terminating a record with an embedded carriage return character.

The developer has no real control over the user’s behavior in this regard, so the best thing he or she can do is to program as many checks and validations as possible to prevent bad data from getting into the system. The first things to verify are the number of fields in each incoming data record and some kind of validation of the type and content of the critical fields. These validations will catch many of the errors, but some may still sneak through. When that happens, the problems are often detected only after the fact when someone notices the data is wonky or when calculations produce weird results.

The point of this discussion is simply to raise awareness that when an app imports data from an Excel file maintained by someone else, the developer cannot rely on that data always being in the agreed-upon format.

### Summary

Visual FoxPro has a rich set of built-in commands and functions for working with strings, making it an ideal tool for importing and exporting data in almost any flat file format. Basic commands like APPEND FROM and COPY TO are sufficient in many situations, but when these fall short VFP's low-level file functions give the developer fine-grained control over the individual fields and lines. When working with XML, low-level file functions can also be employed if the built-in CURSORTOXML( ) and XMLTOCURSOR( ) functions aren't enough. As in many other areas, VFP proves itself to be a powerful and versatile tool.

### Biography

Rick Borup is owner and president of Information Technology Associates, LLC, a professional software development, computer services, and information systems consulting firm he founded in 1993. Rick earned BS and MBA degrees from the University of Illinois and spent several years developing software applications for mainframe computers before turning to PC database development tools in the late 1980s. He began working with FoxPro in 1991, and has worked full time in FoxPro and Visual FoxPro since 1993. He is a co-author of the books *Deploying Visual FoxPro Solutions* and *Visual FoxPro Best Practices For The Next Ten Years*. He has written articles for FoxTalk and FoxPro Advisor, and is a frequent speaker at Visual FoxPro conferences and user groups. Rick is a Microsoft Certified Solution Developer (MCSD) and a Microsoft Certified Professional (MCP) in Visual FoxPro.

*Copyright © 2014 Rick Borup. Windows® is a registered trademark of Microsoft Corporation in the United States and other countries. All other trademarks are the property of their respective owners*

### Bibliography

[1] Rick Borup, *Quibbles, Quirks, and Quickies*, Southwest Fox 2009, <http://bit.ly/igbHlx>