Using A Text File Device Driver As A String Parser

by Jon Q Jacobs

Extracting information from text strings, into other strings or numeric variables, for example, is something we have all probably spent many programming hours on. Listing 1 shows a typical example of the kind of thing I'm talking about. Although Delphi and its ancestors have simple and effective string manipulation abilities, sometimes I yearned for something a little easier, or at least something that looks less messy.

I looked wistfully at the fact that using `read` or `readln` to pull a number from a text file did not require the explicit removal of spaces. To be sure, in order to process a number from string form into numeric variables, the system must do some kind of space-stripping or space-skipping behind the scenes. The fact remains that Borland already wrote that code and included it in the compiler package. I wanted to get access to it! Also noteworthy was the ease with which `readln` broke up a stream of variable length data in a text file if it was delimited by carriage returns. Again, I know there is code behind the scenes that performs such manipulations (it doesn't happen by magic!).

How pleased I was when I ran across the concept of the text file device driver (TFDD). With the TFDD, Borland very thoughtfully provided hooks into their text file handling system. As I read about it, I came to realize that it was, as the name implied, oriented to 'devices'. So I dismissed the usefulness of the concept except for such things as processing a stream of data to and from a temperature controller through an intelligent serial card. Use it for a 'device' in other words. At last I awoke to the fact that I could interpret 'device' more generously. Imagine an ordinary Pascal-style string as a device. I will now endeavor to explain my string device driver (you will find more about TFDDs in general in Brian Long's article last month).

Assign A Relationship

Delphi uses an `AssignFile` procedure to make an association between a filename (the device) and a variable of type `Text`. There is a procedure called `AssignCrt` which associates the screen with a `Text` variable. This is one example of a TFDD that maps something besides a disk file to a `Text` variable. Usually an assign procedure takes two parameters: the `Text` variable and something that identifies the device. In the case of `AssignFile` the second parameter is the name of the disk file to be used. For its inner workings the `AssignFile` procedure uses MS-DOS system calls to complete the association (make no mistake, Windows 95 contains a lot of good old MS-DOS code). `AssignCrt`, on the other hand, does not require a second parameter, because the screen on a given computer is (usually anyway) unique and already known.

Roll Your Own

For a TFDD we write our own assign procedure. I've called mine `AssignSt` (Listing 2). It is mapping a Pascal string to the `Text` variable. The procedure will have a few housekeeping chores to accomplish, but the power of the TFDD is already in place.

The `TTextRec` is a record type declared in `SysUtils` that we can use to typecast the more common `TextFile` type to give visibility to its fields. To take advantage of its 16-byte `UserData` field I declared a type called `usr` for typecasting that

➤ *Listing 1*

```
{comma delimited data}
p := pos(',',StringRecord);
FirstPart := copy(StringRecord,1,pred(p));
system.delete(StringRecord,1,p);
p := pos(',',StringRecord);
SecondPart := copy(StringRecord,1,pred(p));
system.delete(StringRecord,1,p);
{number coming up}
p := pos(',',StringRecord);
temp := copy(StringRecord,1,pred(p));
system.delete(StringRecord,1,p);
{number may be left justified}
while temp[length(temp)]=' ' do dec(temp[0]);
{number may be right justified}
while (temp[1]=' ') and (temp<>'') do system.delete(temp,1,1);
val(temp,IntegerOne,err);
if err<>0 then HandleConversionError;
p := pos(',',StringRecord);
ThirdPart := copy(StringRecord,1,pred(p));
system.delete(StringRecord,1,p);
{And so on...}
```

➤ *Listing 2*

```
procedure AssignSt(var t:TextFile; var s:string);
begin
  with TTextRec(t), usr(UserData) do begin
    Mode := fmClosed;
    BufSize := SizeOf(buffer);
    BufPtr := @buffer;
    OpenFunc := @OpenStr;
    Name[0] := #0;
    ps := @s;
    Handle := 0;
  end; {with}
end; {AssignSt}
```

field. I also declared PString so I would not have to use another unit just for that tiny declaration:

```
PString = ^string;
usr = record
  ps : PString;
  ud : array[5..16] of byte;
end;
```

A Text (or TextFile) variable has its own 128-byte buffer. The TFDD allows for the possibility that we may need a buffer of a different size, so also included in the record structure is a pointer field called BufPtr and a field in which we can store the size of the buffer, BufSize. Now that we have briefly considered the data structure involved, let us see what AssignSt actually does. First it sets the file mode as closed, which is a good idea, since we can assign in one step and open later, perhaps never. I decided to use the buffer already provided, which is named, appropriately, buffer. Therefore I just assigned its size to BufSize and put its address in BufPtr. AssignSt does not need a name for the string device, so it just puts a null character at the start of the Name field. AssignSt makes the important association between the Text variable and the 'device' (string) by the simple expedient of putting its address in the string pointer variable that will be available to some other procedures. Finally AssignSt sets Handle to zero. Since this TFDD will not be accessing a physical file it does not need to use the file handle for its original purpose. Instead, it becomes useful as an index into the string.

Grand Opening

Next we need to open the device. The TFDD must have an open function that will be used by reset, rewrite and append. I called the open function OpenStr and AssignSt puts its address in the OpenFunc pointer field. This open function and several others are the hooks into the text file system. All the hooks are called by means of pointer variables, so they have to be far calls. Further, the TFDD hooks are responsible for much of the error mechanism, so they are each functions returning type integer. In each case the function result will wind up in the InOutRes variable. That variable has the value that is returned and cleared by calls to ioResult. All of the string device driver hooks return 0: no errors are generated, except for numeric conversion errors that are handled out of our view.

While OpenStr is a little heftier than AssignSt, it remains fairly simple also. See Listing 3. We identify the close function by putting its address in the CloseFunc pointer field. The bulk of the work is done in the case statement. Most of that work consists of making the proper assignments to the InOutFunc and FlushFunc fields. The value of the Mode field, which AssignSt initially made fmClosed, depends on the way the device is opened.

In And Out

Text files can be open for either input or output, but not both at the same time. Access is sequential, not random. Opening a text file is performed by the reset, rewrite and append procedures. Each one sets the value of Mode and calls the procedure indicated by the OpenFunc field. In our string device driver, OpenFunc points to OpenStr.

If Mode has the value of fmInput then the open function was called by the reset procedure. In this case OpenStr makes InOutFunc point to InStr, which is strictly an input function.

If Mode has the value of fmOutput then the open function was called by the rewrite procedure. OpenStr makes InOutFunc point to OutStr, an output function. Notice that FlushFunc also points to the same function: in the output mode we want to ensure that data is forced out, not just hanging in a buffer until we close.

If Mode has fmInOut for its value, it is not appropriate for text files at all, since they are either input or output, but not both. Append calls the open function with Mode set this way merely to indicate that it was append and not just rewrite. The open function changes Mode to fmOutput and puts the Handle index past the end of the string. Of course it also sets the InOutFunc and FlushFunc to the address of the OutStr function.

➤ *Listing 3*

```
function OpenStr(var t:textFile):integer; far;
begin
  with TTextRec(t),usr(UserData) do begin
    CloseFunc := @CloseStr;
    case Mode of
      fmInOut : begin
        Mode := fmOutput;
        InOutFunc := @OutStr;
        FlushFunc := @OutStr;
        Handle := length(ps^);
      end;
      fmInput : begin
        InOutFunc := @InStr;
        FlushFunc := @FlushStr;
      end;
      fmOutput : begin
        InOutFunc := @OutStr;
        FlushFunc := @OutStr;
        ps^ := '';
      end;
    end; {case}
  end; {with}
  Result := 0; {for ioResult}
end; {OpenStr}
```

➤ *Listing 4*

```
function InStr(var t:textFile):integer; far;
begin
  Result := 0; {for ioResult}
  with TTextRec(t),usr(UserData) do begin
    if (BufPos<BufEnd) and (Handle<>0) then exit;
    BufPos := 0;
    BufEnd := length(ps^)-Handle;
    if BufEnd>BufSize then BufEnd := BufSize;
    move(ps^[succ(Handle)],BufPtr^,BufEnd);
    inc(Handle,BufEnd);
  end;
end; {InStr}
```

So far the routines we have examined have been purely bookkeeping measures. It is time to see some action. The InStr function gets data in from the device: see Listing 4.

Again, the function returns 0 to indicate no error. The text file system takes care of moving data out of the buffer into the appropriate variable, incrementing BufPos as it scans down the buffer. For example, a readln(t,x) where x is an integer variable will take data out of the buffer until the appropriate numeric digits have all been collected. It will eat up any leading spaces on the way. It will know that it has finished gathering digits for the number when it reaches a space or a carriage return. Since we chose readln for our example, it will continue scanning down the buffer until the end of the 'device' or it finds a carriage return. It will also eat a line feed if present immediately after the carriage return. The text file system handles the data conversion and error reporting without requiring any code from the programmer. (Exactly what I was after!) So, what is the job of InStr? Its job is to keep the buffer supplied. The text file system will call InStr as often as needed to keep the buffer provided with data until the variables in the read or readln statement have been satisfied, or until there is no more data available.

In the InStr function, Handle = 0 signals that the first buffer loading has not happened yet (remember what AssignSt did with Handle?). BufPos >= BufEnd signals that it is time to refill the buffer with BufSize bytes. If neither condition is true, InStr just exits. Otherwise InStr transfers another BufSize bytes of data from the string into the buffer, or fewer if there are not enough bytes left. BufEnd is set according to the amount of data actually transferred into the buffer. BufPos is reset to the beginning of the buffer. Finally, InStr sets the value of Handle to indicate how far down the string it has travelled, just as the text file system adjusts BufPos to indicate how far down the buffer it has scanned.

OutStr (Listing 5) is the other key function. While InStr transferred data out of the string into the buffer, OutStr transfers data out of the buffer into the string. The text file system handles moving data from the parameters in the write or writeln statement, with data conversion as needed, into the buffer, calling on the services of OutStr as often as necessary to keep the buffer from overflowing. In the output scenario, the roles of BufPos and BufEnd seem reversed. At the end of each use of OutStr, BufEnd is reset. Again, Handle keeps track of the position within the string, though it's not really needed in outputs, since all the characters are added to the end of the string which is handled automatically by the length byte.

Finishing Up

The flush function is called when an output sequence is done. If the transfer of data from the parameters of write or writeln has ended with a partially filled buffer, the flush function can send the remaining data out to the device, if that is desired. For our string device driver, that is exactly what we want, so FlushFunc was set (in OpenStr) to the address of OutStr. This final call to OutStr copies any data remaining in the buffer to the string. When the text file system is performing input, rarely does the flush function have a useful purpose, but the hook is provided, just in case. For our string device driver, FlushFunc is not needed for input, so OpenStr pointed it to a do-nothing function called FlushStr: its only job is to assure the text file system that no error has happened:

```
function FlushStr(
  var t:textFile):integer; far;
begin
  Result := 0; {for ioResult}
end; {FlushStr}
```

It is fitting that the close function closes the discussion of the various functions the string device driver supplies. In this simple driver, the close function just sets Mode to fmClosed, resets Handle and comforts the text file system with a "no error" return value (Listing 6).

Now all that remains is to mention the delim procedure and put the string device driver unit to the test. The purpose of delim is to convert a given delimiter character in a string into carriage returns, thus enabling the TFDD to break up the string into parts. Setting the undo parameter to True allows delim to reverse that process.

Perhaps it would have been cleaner to convert the delimiters to carriage returns in the buffer within InStr, but it seemed to be a little more efficient to perform the operation only once on the string, rather than several times on the buffer. I wrote the delim procedure in assembly for still more efficiency. It performs a simple operation and each significant step is described in the comments. See the code in Listing 7.

➤ *Listing 5*

```
function OutStr(var t:textFile):integer; far; var i : integer;
begin
  with TTextRec(t),usr(UserData) do begin
    for i := BufEnd to BufPos-1 do ps^ := ps^+BufPtr^[i];
    Handle := length(ps^);
    BufEnd := BufPos;
  end; {with}
  Result := 0; {for ioResult}
end; {OutStr}
```

➤ *Listing 6*

```
function CloseStr(var t:textFile):integer; far;
begin
  with TTextRec(t) do begin
    Mode := fmClosed;
    Handle := 0;
  end;
  Result := 0; {for ioResult}
end; {CloseStr}
```

This Is Only A Test

The program in Listing 8 exercises the string device driver unit. Clicking the `Test 1` button fires the code in the procedure `test1Buttonclick`, giving the output in Figure 1. We use a comma as the delimiter, open the string, break it into variously sized parts using `Read` and `ReadLn` returning `byte` and `word` variables. It also shows opening for output. Notice that `closeFile` is not needed before changing modes, because all three open procedures close the file first if it is open. In fact, the test program never calls `CloseFile`, because the device driver uses no DOS file handles which need cleaning up. The `rewrite` procedure makes `s` a null string. Each `write` then builds onto it, doing number to string conversions as needed. This is only for demonstration: ordinary string operations work just fine in place of these outputs. Even if you want to write to your strings, you probably will not want to use `WriteLn`, because that will add a line feed character as well as a carriage return.

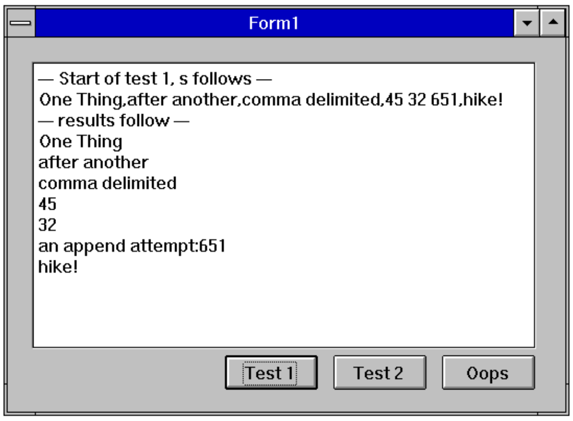
The next few lines demonstrate the effect of the `append` procedure, which is to leave the existing string intact and build on the end. The procedure `test2ButtonClick` is fired when the `Test 2` button is clicked. Figure 2 shows the results. The delimiter, which `delim` converts, is a backslash instead of a comma. The most noteworthy difference over the first test is:

```
readln(t,r);
add(FloatToStr(r));
```

The type for `r` is `extended`, a floating point type. You could just as easily use `single`, `double` or even `real`, but in the adjustments of precision you wind up with some trash digits after a bunch of zeros *[See this month's Clinic for an explanation. Editor]*. Use `FloatToStrF` to have better control of the precision and format of the string result.

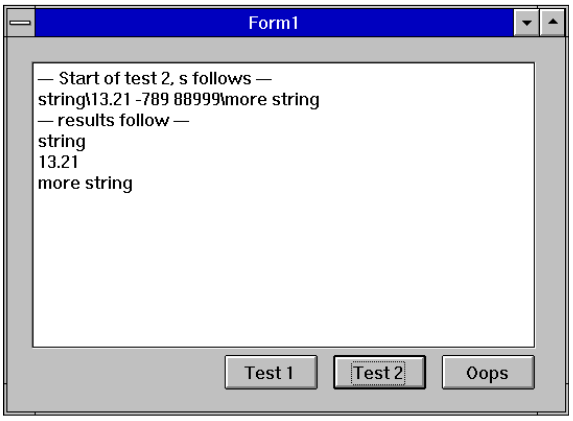
Notice in the output that `-789` and `88999` were skipped because `readln(t,r)` just scanned to the next carriage return once the variable was satisfied. The string TFDD works just like a text file on disk!

➤ *Figure 1*



```
— Start of test 1, s follows —
One Thing,after another,comma delimited,45 32 651,hike!
— results follow —
One Thing
after another
comma delimited
45
32
an append attempt:651
hike!
```

➤ *Figure 2*



```
— Start of test 2, s follows —
string\13.21 -789 88999\more string
— results follow —
string
13.21
more string
```

```
procedure delim(var s:string; c:char; undo:boolean); assembler;
asm
  {point to the string with es:di and put its length into cx}
  les di,s
  mov cl,es:di.0
  {bail out if null string}
  or cl,cl
  jz @@2
  xor ch,ch
  {search direction forward}
  cld
  {make sure length byte not tested}
  inc di
  {set up the comparison}
  mov al,c
  mov ah,13
  mov bl,undo
  or bl,bl
  {leave normal if undo false, c will become #13}
  jz @@1
  {restore if undo, #13 will become c}
  xchg al,ah
  @@1:
  {search until match found}
  repnz scasb
  {if no match here then done, got here by reading end}
  jnz @@2
  {substitute the found char}
  mov es:di.-1,ah
  {check for found and at end at the same time}
  or cl,cl
  jz @@2
  {look for next match}
  jmp @@1
  @@2:
end;
```

➤ *Listing 7*

An Error Test

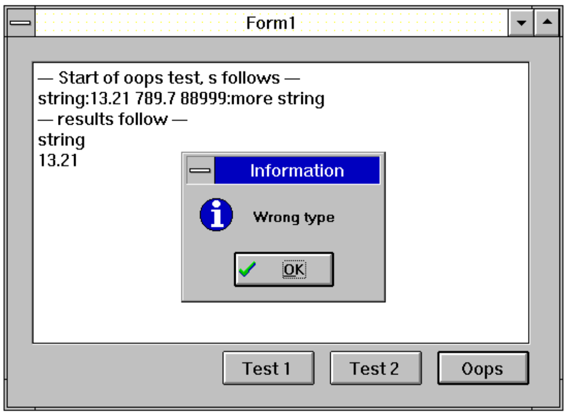
The `oopsButtonClick` fires when the `Oops` button is clicked and will produce a deliberate error (Figure 3). The problem is an attempt to read `float` type data into an `integer` type variable. The message box appears and the last two lines do not appear in the memo until you close the message box. The `w` variable never gets shown. You can experiment with several ways to handle such errors: you could leave out the custom exception handling and let the default exception handler take it. Its message box will say *Invalid* *numeric input*. The last two lines will not appear at all, because the default handler will bail out of the block where the exception happened. Another approach would be to turn off built-in I/O error checking with the {$I-} compiler directive, then you would directly program error checking with the ioResult function, which would return a value of 106.

Variations

You can now move on to experiment! I made delim work on the device string itself for the sake of efficiency. A more self-contained (and more satisfying) approach would be to make delim work on the buffer and call it from within InStr. Replace the undo parameter with a size parameter of type byte and pass BufSize (assuming buffers smaller than 256 characters). Use mov cl,size to limit the search and remove the inc di. Add a third parameter to AssignSt: the delimiting character. You could keep it in the 5th position in UserData. You could allow several delimiters, passed as a string to AssignSt. They could be stored in adjacent places in 'UserData with a null at the end. In InStr you'd call delim for each possible delimiter until the null character was reached. You may think of many more ideas.

Jon Jacobs is a software engineer at Mastercomp, Inc., an industrial automation company in Dallas. Email him at mstrcomp@gte.net or visit http://home1.gte.net/mstrcomp/index.htm

➤ Figure 3



```
— Start of oops test, s follows —
string:13.21 789.7 88999:more string
— results follow —
string
13.21
```

Wrong type

➤ Listing 8

```
unit Test1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    test1Button: TButton;
    test2Button: TButton;
    oopsButton: TButton;
    procedure FormCreate(Sender: TObject);
    procedure test1ButtonClick(Sender: TObject);
    procedure test2ButtonClick(Sender: TObject);
    procedure oopsButtonClick(Sender: TObject);
   private
    t : TextFile;
    s : string;
   public
  end;
var
  Form1: TForm1;
implementation
uses
  sdd;
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Clear;
  AssignSt(t,s);
end;
procedure TForm1.test1ButtonClick(Sender: TObject);
var
  s1,s2,s3,s4 : string[23];
  n1,n2 : byte;
  n3 : word;
begin
  s :=
   'One Thing,after another,comma delimited,45 32 651,hike!';
  with memo1,lines do begin
    add('-- Start of test 1, s follows --');
    add(s);
    add('-- results follow --');
    delim(s,',',false);
    reset(t);
    readln(t,s1);
    readln(t,s2);
    readln(t,s3);
    read(t,n1);
    readln(t,n2,n3);
    readln(t,s4);
    add(s1);
    add(s2);
    add(s3);
    rewrite(t);
    write(t,n1);
    add(s);
    rewrite(t);
    write(t,n2);
    add(s);
    s := 'an append attempt:';
    append(t);
    write(t,n3);
    add(s);
    add(s4);
  end;
end;
procedure TForm1.test2ButtonClick(Sender: TObject);
var
  s1,s2 : string[23];
  r : extended;
begin
  s := 'string\13.21 -789 88999\more string';
  with memo1,lines do begin
    add('-- Start of test 2, s follows --');
    add(s);
    add('-- results follow --');
    delim(s,'\',false);
    reset(t);
    readln(t,s1); add(s1);
    readln(t,r);  add(FloatToStr(r));
    readln(t,s2); add(s2);
  end;
end;
procedure TForm1.oopsButtonClick(Sender: TObject);
var
  s1,s2 : string[23];
  r : extended;
  w : word;
  l : longint;
  io : integer;
begin
  s := 'string:13.21 789.7 88999:more string';
  with memo1,lines do begin
    add('-- Start of oops test, s follows --');
    add(s);
    add('-- results follow --');
    delim(s,':',false);
    reset(t);
    readln(t,s1); add(s1);
    read(t,r);    add(FloatToStr(r));
    try
      read(t,w);  add(IntToStr(w));
    except
      MessageDlg('Wrong type',mtInformation,[mbOk],0);
    end;
    readln(t,l);  add(IntToStr(l));
    readln(t,s2); add(s2);
  end;
end;
end.
```