Getting Started with Dynamic Data Exchange

Howard Schreier, U.S. Department of Commerce, Washington DC

Abstract

There are numerous ways to move information between SAS® and third-party applications, such as spreadsheets and data base packages, especially in the Microsoft Windows environment. Dynamic Data Exchange (DDE) is a versatile approach, because it provides a DATA step interface rather than data set conversion. This presentation will explain the basic techniques of using DDE to move data to and from SAS.

Introduction

SAS is, to say the least, a versatile software product. It can do lots of things. Nevertheless, most of us occasionally use other products; if we don't, our customers do. From this comes the frequent need to move data back and forth between SAS and various other proprietary file formats.

One method of accomplishing such data transfers is Dynamic Data Exchange (DDE), which is a feature of Microsoft Windows and IBM® OS/2® operating systems.

DDE is a client-server process. SAS can function only as a DDE client. The client is the application which initiates the communication, which can be a request for the server to receive data or a request for the server to send data.

Here is an example. Like all of the examples in this tutorial, it was run using Release 6.12/TS020 of the SAS System under Windows 95 and uses Microsoft Excel Version 5.0 as the target (server) application. However, it's important to remember that many other applications also support DDE.

Figure 1 presents an Excel spreadsheet (named "easy"), and the object of the example is to read these data into a SAS data set. Here is the SAS code which invokes DDE to do exactly that:

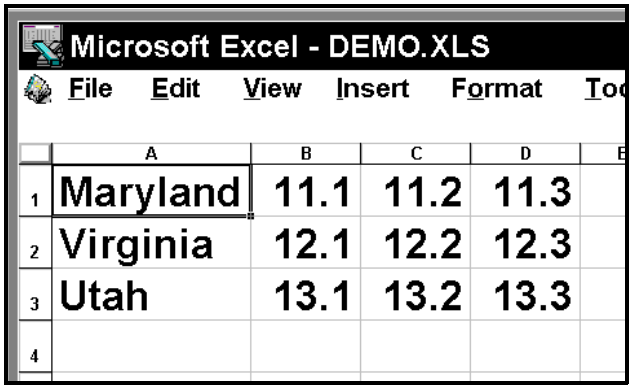

Figure 1

```
filename readata dde
 'Excel|[demo.xls]easy!R1C1:R3C4';
data demo_1;
infile readata;
input state $ val_b val_c val_d;
run;
```

To somebody who is familiar with SAS but unfamiliar with DDE, this is a very ordinary-looking DATA step following a very strange-looking FILENAME statement. Indeed, all of the magic is in the FILENAME statement.

Note the keyword "dde" and what is termed the DDE-triplet in quotation marks. It is called a triplet because it has up to three segments: (1) to the left of the vertical bar (|) is the name of the server application (in this case, Excel); (2) between the vertical bar and the exclamation point (!) is the so-called "topic", which in this situation is the name of the Excel worksheet, prefixed by the bracketed name of the container workbook; (3) following the exclamation point is the "item", in this case a range of cells.

The resulting SAS data set corresponds perfectly to the spreadsheet:

```
OBS   STATE      VAL_B   VAL_C   VAL_D

 1    Maryland    11.1    11.2    11.3
 2    Virginia    12.1    12.2    12.3
 3    Utah        13.1    13.2    13.3
```

Here's another example, working in the opposite direction. We'll write the same data back out to Excel, reversing the order of the variables for the sake of variety. The program:

```
filename otherway dde
 'Excel|[demo.xls]back2xl!R1C1:R3C4';
data _null_;
set demo_1;
file otherway;
put val_d val_c val_b state;
run;
```

Note that the target spreadsheet must be open. Unless otherwise indicated, all of the examples in this tutorial reference worksheets which already exist in open workbooks in the Excel workspace.

Figure 2 presents the resulting Excel spreadsheet. With that we have demonstrated simple data transfer in both directions between SAS and Excel.

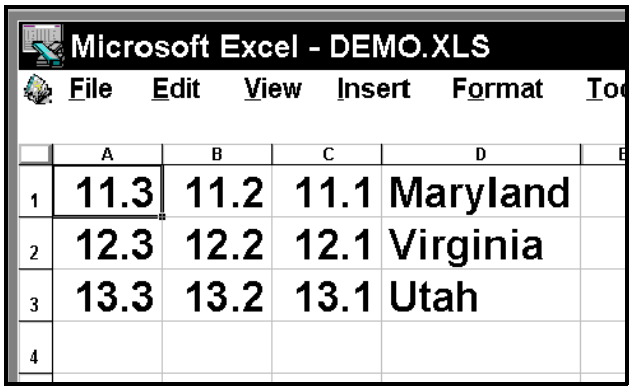

Figure 2

Alternatives

Before we delve into the nuts and bolts of DDE, let's put it into context by considering other options for moving data between SAS and various third-party proprietary formats. Possibilities include:

- Object Linking and Embedding (OLE)
- Open Data Base Connectivity (ODBC)
- pipes
- specific SAS data engines (either SAS/ACCESS® or third-party products)
- specific SAS conversion procedures
- file conversion utilities, such as DBMS/COPY.
- common-denominator strategies, using such intermediate file formats as comma-separated text or Hypertext Markup Language (HTML)
- "roll-your-own" methods based on reverse engineering; an example of this approach is Hockey's DBFTOSAS macro for reading dBase files

Deliberately not mentioned are the SAS Import Wizard and Export Wizard, because they do not actually represent an additional option. Rather, they provide ease of use for some of the methods listed.

It may seem like there is overkill here, with an excessive number of solutions to the problem. However, the various methods have different advantages and disadvantages. Kuligowski covers most of them at some length; meanwhile, here are things to consider:

1. *Are you working in a visual/object orientation or in a more traditional data orientation? If the latter, are you working with well-formed tables (that is, structures which correspond with the variable-by-observation organization of SAS data sets) or with more intricate or less regular structures which require a lower-level approach?* Figure 3 is an example of a data structure which already strongly resembles a SAS data set. While it certainly can be processed by SAS via DDE, it also lends itself to more high-level approaches, operating on the whole table as an entity. Figure 4 is a spreadsheet containing the same information, but in a more complex design. It calls for a more low-level approach, and that happens to be the real strength of DDE. This is why DDE is often a good choice for working with spreadsheets, which can be designed along the lines of Figure 3 but are very often structured more like Figure 4.

2. *Do you have the target application licensed and installed?* DDE requires the target product (eg, Excel) to be installed and running. This is because, unlike most of the other methods, which work with native files stored on disk, DDE communicates with the running program.

3. *Do you have available any additional software product (either from SAS Institute or from a third party) which is required?* DDE support is provided in base SAS software; no add-on product is required.

4. *Are you doing a one-time conversion of a small number of files? Or do you need a technique which lends itself to automation, either because it is a recurring operation or because the number of files is large?* DDE is fairly easy to automate.

5. *On what platform or platforms are you running SAS?* DDE is available in the OS/2 and Windows environments.

Keeping Tabs

Now we'll return to the mechanics of using DDE with SAS.

The introductory examples used very well-behaved data, which made it pretty simple to get DDE working. It required just the magic FILENAME statement.

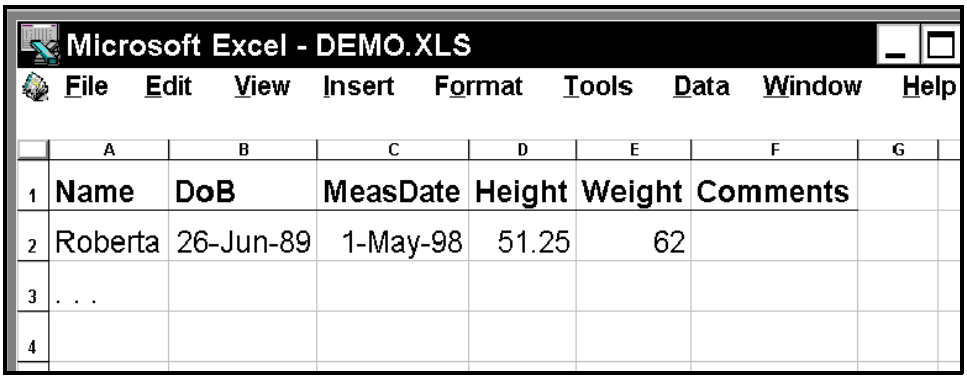

Figure 3

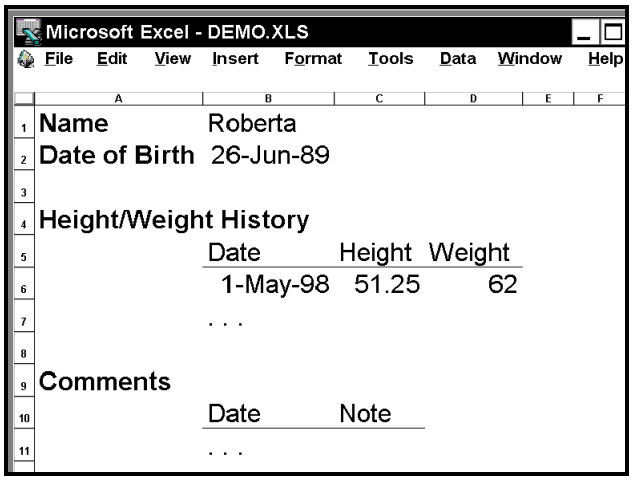

Figure 4

Let's turn to a what is perhaps a more realistic example. Look at the spreadsheet (named "noteasy") in Figure 5. Notice the missing value ... and the character value with a length exceeding eight ... and the embedded blank within a character value.

We'll try the technique which worked before:

```
filename readata dde
 'Excel|[demo.xls]noteasy!R1C1:R3C4';
data demo_2;
infile readata;
input state $ val_b val_c val_d;
run;
```

The results reveal some difficulties. First look at excerpts from the SAS log:

```
NOTE: Invalid data for VAL_D in
      line 2 1-7.
RULE:
----+----1----+----2----+----3
2         Georgia 22.1 22.2 22.3 22
STATE=Californ VAL_B=21.1 VAL_C=21.3
VAL_D=. _ERROR_=1 _N_=1
NOTE: Invalid data for VAL_B in
      line 3 5-8.
3         New York 23.1 23.2 23.3 23
STATE=New VAL_B=. VAL_C=23.1 VAL_D=23.2
_ERROR_=1 _N_=2

NOTE: SAS went to a new line when INPUT
      statement reached past the end of
      a line.
```

Now here's the SAS data set which was created:

```
OBS   STATE      VAL_B   VAL_C   VAL_D

 1    Californ    21.1    21.3      .
 2    New           .     23.1    23.2
```

The only thing that's right is the variable VAL_B in the first observation! So we'll do a few things to make this data transfer work better. The techniques we're going to use are not specific to DDE; they are potentially useful any time a DATA step reads from an external file. Here's revised code:

```
filename readata dde
 'Excel|[demo.xls]noteasy!R1C1:R3C4';
data demo_3;
infile readata missover dsd dlm=' ';
input state:$15. val_b val_c val_d;
put _infile_;
run;
```

Three options have been added to the INFILE statement. MISSOVER prevents SAS from jumping to the next line of data when the INPUT statement exhausts the current line. DSD makes multiple consecutive delimiters significant (rather than treating them as one). DLM= is used here to keep the space character as the delimiter, since DSD implicitly changes the default delimiter to the comma.

Also, we've revised the INPUT statement to use a format along with the colon modifier for the character variable; this permits reading values longer than eight characters without having to specify the absolute column positions while still allowing the delimiter character to terminate the field.

One other change, the PUT statement, does not affect the SAS data set being built, but does aid in debugging. Note that _INFILE_ here is not a variable; rather it is a keyword specification. The statement causes the input buffer (that is, the current line from the file to which the INFILE statement points) to be dumped in the SAS log. It works when reading a "normal" file and it also works when the INFILE points to a DDE source.

Let's stop for a moment to put these techniques in context. In terms of record layout, there are basically two kinds of external files: fixed-format and free-format. With fixed format, fields appear in known column positions. With free format, if fields vary in length from record to record their column positions will vary, and the reading process depends on the detection of consistently used separators or delimiters. Our tune-up of the INFILE and INPUT

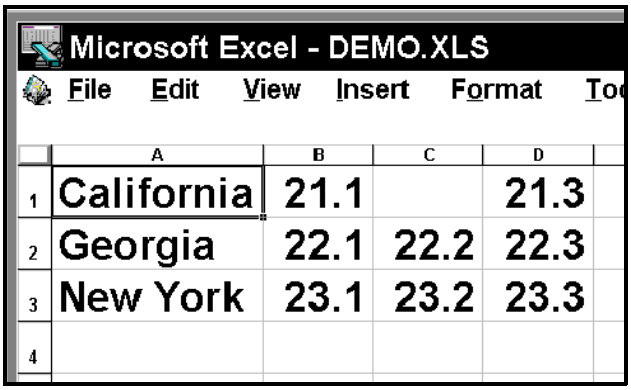

Figure 5

statements here exploits the SAS repertoire for free-format input.

When our revised program is run, the log includes the following:

```
California 21.1  21.3
Georgia 22.1 22.2 22.3
NOTE: Invalid data for VAL_B in
      line 3 5-8.
New York 23.1 23.2 23.3
RULE:
----+----1----+----2----+----3
3         New York 23.1 23.2 23.3 23
STATE=New VAL_B=. VAL_C=23.1 VAL_D=23.2
_ERROR_=1 _N_=3
```

The resulting SAS data set looks like this:

OBS	STATE	VAL_B	VAL_C	VAL_D
1	California	21.1	.	21.3
2	Georgia	22.1	22.2	22.3
3	New	.	23.1	23.2

It's better, but still not right.

Before we do anything else, look at the buffers dumped in the SAS log by the PUT statement. They look like ordinary lines of data which might have been keyed into an editor and saved in a plain text file. That's why everyday INFILE and INPUT techniques, not specific to DDE, are appropriate. Another implication is that the DDE strategy for data transfer is similar to the "common-denominator" strategy using text files (in other words, exporting plain text from one application and reading it into the other application). DDE just does it on the fly and avoids creation of an intermediate file on disk. At this point, it should be clear that the DDE approach to importing data operates at a low level; there may be a lot of inherent structure in the source (data typing and other field attributes), but DDE does not see it. Rather, the SAS DATA step must be crafted to detect and/or rebuild the structure on the SAS side.

Getting back to our example, there still is a problem with the embedded blank in "New York". The format with the colon modifier causes it to be mistaken for the end of the field.

So far, we have been refining the DATA step code. Now we go back to the FILENAME statement. Here's the code for our third try at this spreadsheet:

```
filename readata dde
 'Excel|[demo.xls]noteasy!R1C1:R3C4'
 notab;
data demo_4;
infile readata missover dsd dlm='09'x;
input state:$15. val_b val_c val_d;
put _infile_;
run;
```

The NOTAB option has been added to the FILENAME statement. At the same time the value of the DLM= option of the INFILE statement is changed from a blank to a hexadecimal 09, which denotes a tab character. This may not immediately make sense, but the resulting SAS data set is at last a correct reflection of the Excel spreadsheet:

OBS	STATE	VAL_B	VAL_C	VAL_D
1	California	21.1	.	21.3
2	Georgia	22.1	22.2	22.3
3	New York	23.1	23.2	23.3

Also, it's worthwhile looking at the dumped input buffers in the SAS log. Up until now, we've just seen text transcribed from the log window. In this case, we will look at an actual screen shot from the SAS Display Manager System (Figure 6). The little rectangles represent tab characters.

Let's stop for a moment and go over what is happening. Excel, as a DDE server, passes tab characters to SAS as field separators; that much is a given. The default behavior of SAS as a DDE client is to replace these tabs with blanks. This is fine as long as these are the **only** blanks (in other words, if there are no blanks -- leading, trailing, or embedded -- within the spreadsheet cells). That was the situation in our initial example. If there **are** blanks as part of the data, it is impossible to distinguish these from the blanks which are generated during DDE processing to replace the tab separators.

The NOTAB option inhibits the substitution of blanks for the tabs which Excel sends. Instead, the tabs are passed through to the DATA step's input buffer. That's why it was necessary to change the DLM= option in the INFILE statement. Once these changes were made, the INPUT statement knew to recognize tabs as field separators and to consider blanks as part of the data, and the import was done correctly.

It's a bit confusing, because using the NOTAB

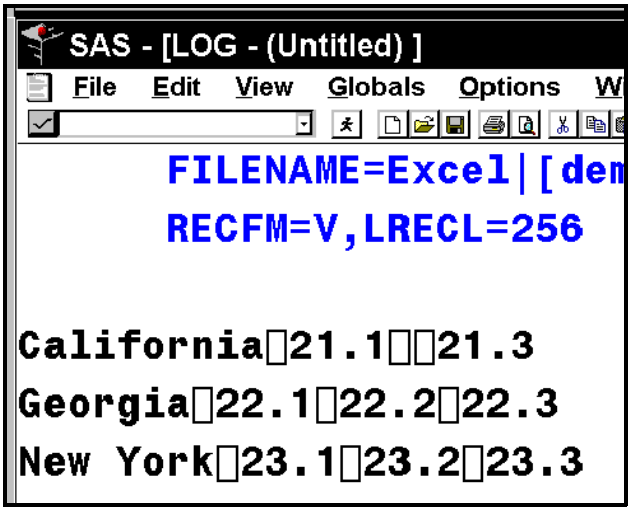

Figure 6

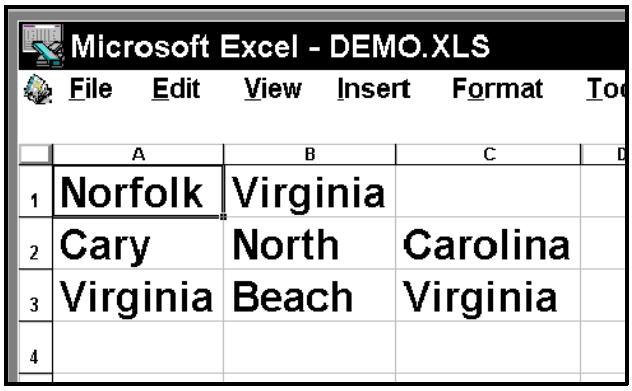

Figure 7

option means that you **do** get tab characters in your data stream, and have to process them. NOTAB really means that there is no automatic substitution of blanks for tabs (perhaps it should have been dubbed the "YESTAB" option).

The NOTAB option is also relevant when SAS is sending data **to** Excel. The default behavior is for SAS to replace each blank in a line of output with a tab before it passes the line to Excel.

Here is a SAS data set which will illustrate the issue:

```
OBS    CITY              STATE

 1     Norfolk           Virginia
 2     Cary              North Carolina
 3     Virginia Beach    Virginia
```

The following code will export it to Excel:

```
filename otherway dde
 'Excel|[demo.xls]cities!R1:R3';
data _null_;
set cities;
file otherway;
put city state;
run;
```

Remember that a simple list style PUT statement like the one in this example inserts single blanks between fields.

Figure 7 shows the result in Excel. The first row of the spreadsheet is fine, because neither the city nor the state has a blank within its name. The other lines are not properly recorded; city or state names with embedded blanks are split across cells.

The solution begins with adding the NOTAB option to the FILENAME statement. Here's the revised program:

```
filename otherway dde
 'Excel|[demo.xls]cities2!R1:R3' notab;
data _null_;
set cities;
file otherway;
put city state;
run;
```

Figure 8 presents the output. The NOTAB option indeed inhibits the substitution of tabs for blanks. In fact, because no tabs whatsoever were passed to Excel, there is no separation and each line (city and state) ends up in Column A of the spreadsheet. This is the situation in which the term "NOTAB" does makes sense in describing the behavior.

But we **do** want the cities and states separated in Excel, so tab characters have to be supplied, and once again, NOTAB means dealing with tabs. There is no DLM= option available on the FILE statement, but a hexadecimal constant can simply be included in the PUT statement. Here's the final version of this little program:

```
filename otherway dde
 'Excel|[demo.xls]cities3!R1:R3' notab;
data _null_;
set cities;
file otherway;
put city '09'x state;
run;
```

Look at the results, in Figure 9. Now we have the city names in Column A and the state names in COLUMN B. Once again, the solution was to arrange things so that tab characters are separators and blanks are data.

To sum up the discussion of the NOTAB option: there are cases in which it is not needed, but as a rule it is needed and it is generally good practice to code it and make the necessary adaptations in the DATA step.

Other Pitfalls

Understanding the NOTAB option and its implications is easily the most important thing in getting up to speed with SAS and DDE, but there are some other pitfalls to consider.

Data typing is an issue. Numeric values are passed as formatted strings. This can lead to loss of precision. Date and time values require particular attention.

It is sometimes necessary to code the LRECL=

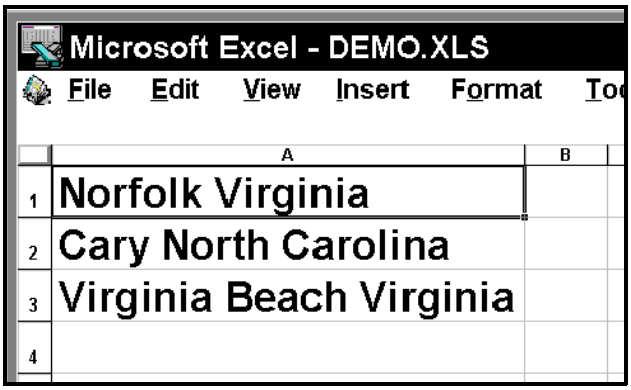

Figure 8

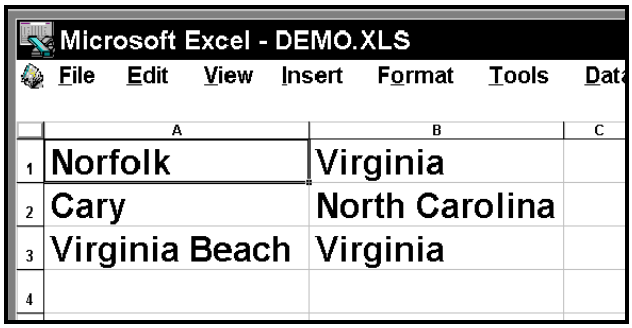

Figure 9

option on the FILE or INFILE statement to override the default value and avoid truncating the records which are transferred via DDE.

Server-Side Activity

We've now concluded our discussion of the basic mechanics of DDE in transferring data between SAS and Excel. However, there are other aspects of DDE, involving activity on the Excel (server) side of things. Up until now, we've assumed that all is static there -- no data or format changes going on, no disk input or output operations, printing, etc.

In the examples so far which involved passing data from Excel to SAS, we've assumed that the Excel cells are not altered while the DDE link to SAS is open. What if they in fact **are** changing? The HOTLINK option on the FILENAME statement can handle this circumstance. Every time a change occurs anywhere within the range of cells identified in the triplet, the entire data stream (reflecting all of the cells in the range) is refreshed for SAS.

While HOTLINK extends the capability of DDE, it is, to say the least, tricky to use. For one thing, there are not going to be any changes on the Excel side unless you have devised some sort of synchronized activity. Second, HOTLINK disables the "normal" DATA step behavior of looping through input data then stopping automatically. Instead, you have to define and/or detect a condition signifying the end of the process and code a STOP statement to terminate the DATA step. Using HOTLINK is really beyond the scope of this tutorial, but see below for an example.

There can also be synchronization concerns when data are flowing in the opposite direction, from SAS to the target application. Ordinarily, when SAS sends data to Excel (or any other DDE server application), the data stream is stored during execution of the DATA step and delivered when SAS does its housekeeping at the termination of the step. However, by incorporating the string '!DDE_FLUSH' in the output, you can force SAS to deliver the output accumulated up to that point. Note that this is **not** an option on a statement, detected at compile time. Rather, it is a value to be embedded in the output at execution time. In other words, you would code a statement like:

```
put '!DDE_FLUSH';
```

with quotation marks around the magic word.

In simple DDE applications such as our earlier examples, the default behavior is appropriate and there is no need for this feature. However, if you are trying to synchronize the data transfer with Excel operations (such as SAVE or PRINT), this buffer-flushing technique becomes critically important and useful.

But how can you do that (trigger activity on the Excel side)? You could code references to the SLEEP function in your DATA step, so that SAS would suspend activity long enough for you to hot-key over to Excel and work its menus. But that's going to be precarious. You could use the HOTLINK option to make your DATA step loop indefinitely. But that's pretty advanced.

Another method is based on the support which Excel (like some other applications) provides for a special DDE "system" topic, which lets the client application (for our purposes, this means SAS) send commands which are then carried out as if they had been initiated by keyboard or mouse activity or by Excel macros.

Specifically, this is done by sending to Excel references to its macro functions (using the Excel Version 4 macro style, which predates the adoption of Visual Basic for Applications, or VBA). The function references, wrapped in brackets, are transmitted and processed immediately; that is, they are **not** buffered like data passed via "ordinary" DDE.

Here is a rather trivial example, which shades alternate rows of a worksheet:

```
filename commands dde 'excel|system';
data _null_;
file commands;
put
 '[WORKBOOK.SELECT'
 '("cities3","cities3")]';
do row = 1 to 15 by 2;
   put '[SELECT("R' row +(-1)
             'C1:R' row +(-1) 'C2")]';
   put '[PATTERNS(4,48,4,TRUE)]';
   end;
run;
```

The PUT statement preceding the DO loop makes a particular worksheet ("cities3") active. The first PUT statement within the loop selects a different range of cells with each iteration of the loop by embedding the loop index into the Excel macro function reference. The last PUT statement sets a particular color and pattern for the shading.
Figure 10 shows the spreadsheet after this little program has been run.

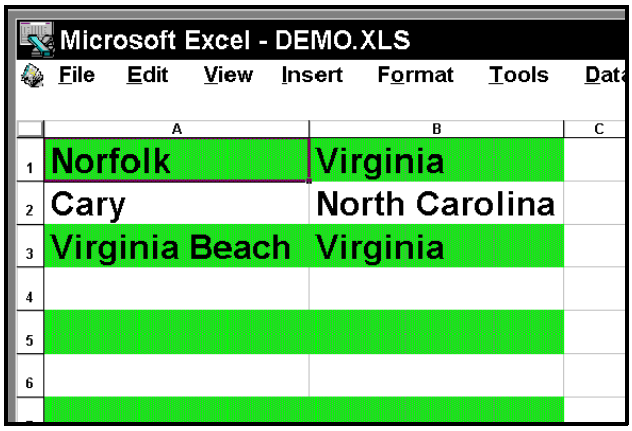

Figure 10

Let's return to the question of synchronization. Consider a DATA step which writes to two DDE links, the "system" topic for Excel commands and an ordinary spreadsheet (cell range) topic for data. Recall that the data transfer will by default be deferred to the end of the step, but not the commands. Thus, if you send Excel 50 rows of data then send a SAVE command, you will save an empty spreadsheet because the SAVE happens right away while the data are still "in the pipeline" -- unless you remember to pass the string '!DDE_FLUSH' after the 50 rows of data. DDE applications of this sort are beyond the scope of this tutorial, although an example does appear below.

Conclusions and Recommendations

We will go over some more examples, but first we can state some generalizations.

Spreadsheets are very popular, and are the most comfortable environment for many people's confrontations with data. So, DDE, which lets SAS read and write spreadsheets directly, is a useful addition to the SAS kit of tools.

DDE has its difficulties, however. We've discussed or at least mentioned most of them.

In addition, the documentation is fragmented, because you have to piece together information from the SAS side with instructions or examples from the server application side, which as a rule will not even mention SAS. Sometimes this documentation is sketchy, and scattered between conventional manuals and online help. Some important SAS documentation appears only in Tech Support papers. The situation is far from completely discouraging, though. The "SAS companion" item which is on the Help menu in recent releases delivers much of the key information about DDE. And of course the Internet can be a rich source of information and support.

System commands are helpful, but mostly for essential housekeeping incidental to data transfer. In theory, you can control complex behavior of the DDE server (Excel), but it's difficult to get information back to SAS on the outcomes of commands sent to Excel, and suppressing user prompts is also a problem. You are probably better off using native (Excel) macro facilities for anything at all intricate, but that is beyond the scope of a beginning tutorial.

Use of System commands and (for input) the HOTLINK option permit SAS to sweep through multiple files. This is powerful, but keep in mind that there is an alternative: use of the SAS macro facility to repeat SAS steps which carry out simpler DDE actions. Again, we are getting away from "beginning" subject matter.

More Examples

This example sweeps through multiple spreadsheets and transfers data to SAS. For input, we will use the two spreadsheets used in earlier examples (Figure 1 and Figure 5).

Here is the code:

```
filename commands dde 'excel|system';
filename pasted dde
 "Excel|[demo.xls]forpaste!R1C1:R3C4"
  hotlink notab;
data demo_5;
infile pasted missover dsd dlm='09'x;
do pp = 'easy   ', 'noteasy'; drop pp;
   file commands;
   put '[WORKBOOK.SELECT("' pp +(-1)
    '","' pp +(-1) '")]';
   put '[SELECT("R1C1:R3C4","R1C1")]';
   put '[COPY()]';
   put '[WORKBOOK.SELECT'
       '("forpaste","forpaste")]';
   put '[SELECT("R1C1")]';
   put '[PASTE()]';
   do line = 1 to 3; drop line;
      input state:$15. val_b
            val_c val_d;
      output;
      end;
   end;
stop;
run;
```

The second FILENAME statement points to a worksheet ("forpaste") which will be used to hold data temporarily. The outer DO loop iterates through the worksheet names ("easy" and "noteasy"), and the first reference to Excel's WORKBOOK.SELECT function embeds these names so that the worksheets are selected in turn. The next four PUT statements are pretty straightforward; they select a three-by-four range of cells, copy it to the clipboard, change the worksheet selection to the work area which has been set up for DDE, and select the upper left cell in the DDE range. The final PUT statement is straightforward too; it causes Excel to paste the contents of the clipboard. However, the use of the HOTLINK option adds a wrinkle; each

time the paste operation is performed, the DDE range is changed and a fresh copy of the range is fed to SAS. Thus, SAS will see data from each spreadsheet addressed by the outer DO loop. The inner DO loop simply reads the data using the technique which we developed earlier. The STOP statement is necessary to keep the DATA step from running indefinitely.

Recall that the cell range set up for DDE here contains three rows; the following line from the SAS log confirms that the HOTLINK option causes it to be read twice:

```
NOTE: 6 records were read from the
      infile PASTED.
```

Here is the result, which correctly concatenates the spreadsheets into a single SAS data set.

OBS	STATE	VAL_B	VAL_C	VAL_D
1	Maryland	11.1	11.2	11.3
2	Virginia	12.1	12.2	12.3
3	Utah	13.1	13.2	13.3
4	California	21.1	.	21.3
5	Georgia	22.1	22.2	22.3
6	New York	23.1	23.2	23.3

Now for our final example. Instead of sweeping through multiple spreadsheets as **input**, we will develop a method to **generate** multiple spreadsheet files.

We will start with the data set created in the very first example:

OBS	STATE	VAL_B	VAL_C	VAL_D
1	Maryland	11.1	11.2	11.3
2	Virginia	12.1	12.2	12.3
3	Utah	13.1	13.2	13.3

As output, we will generate a separate spreadsheet file for each state, with the state name followed by the three numeric values running down the first column of the first worksheet. Here is the program:

```
filename commands dde 'excel|system';
filename multout dde
 'Excel|[shell.xls]Sheet1!C1';
data _null_;
file commands;
put '[NEW(5)]';
put '[SAVE.AS("c:\sesug\shell.xls",'
    '1,"",FALSE,"",FALSE)]';
run;
data _null_;
set demo_1 end=done;
file multout;
put state / val_b / val_c / val_d;
put '!DDE_FLUSH';
file commands;
put '[SAVE.AS("c:\sesug\' state +(-1)
    '.xls",1,"",FALSE,"",FALSE)]';
if done then put "[FILE.CLOSE()]";
run;
```

There are two DATA steps here. The first one simply creates and saves a new Excel workbook (shell.xls) which will be used as a vehicle. The second DATA step begins with very ordinary-looking SET, FILE, and PUT statements. The second PUT statement adds the magic string '!DDE_FLUSH', and since it is unconditional, this causes the DDE buffer to be flushed at this point on each pass through the DATA step. The remaining PUT statements are directed to the DDE "system" topic. The first of these saves the workbook, embedding the variable STATE into the file name. The other one closes the active workbook at the conclusion of processing.

Here is one interesting line from the log.

```
NOTE: 15 records were written to the
      file MULTOUT.
```

Now there are three states and four variables for each state (including the state name). That accounts for twelve records. The other three are records containing the string '!DDE_FLUSH', which are just ordinary data when they are generated by the DATA step, and only have special significance to DDE.

Figure 11 shows the data in one of the Excel workbook files. Similar files are created for the other states represented in the input data.

The '!DDE_FLUSH' is critical. Without it, the entire stream of data would be held and passed to Excel after completion of the last pass through the DATA step, after the Excel files were saved; we would get files containing no data.

This example also illustrates how very tricky some of these more advanced DDE techniques are. Consider that when the first observation is processed, the SAVE.AS operation gives the active workbook in the Excel environment the name of the first state, so there is no longer a workbook named "shell.xls". Nevertheless, the DDE link which was set up for "shell.xls" continues to operate as intended. In fact, if the program is changed so that shell.xls is opened anew for each pass through the DATA step, it will **not** work correctly. Such little mysteries tend

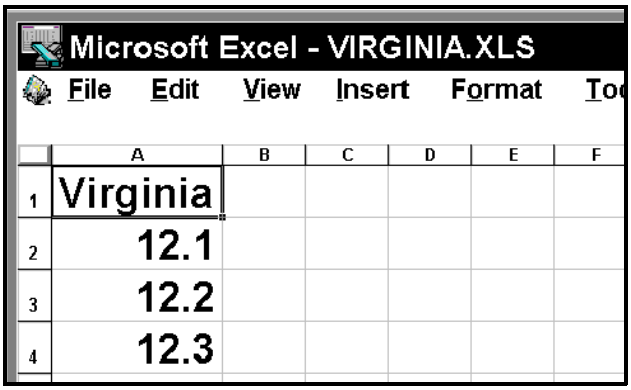

Figure 11

to be hard to unravel because the feedback is usually limited. Typically there are uninformative generic error indicators, or no indicator at all except for the absence of expected data.

References

Bodt, Mark, "Talking to PC Applications Using Dynamic Data Exchange", *Observations®: The Technical Journal for SAS Software Users*, Volume 5, Number 3

Hockey, Richard, "dbftosas reincarnated", in Newsgroup comp.soft-sys.sas, also in SAS-L Archives (gopher://jse.stat.ncsu.edu:70/11/othergroups/sasl), 6 July 1993

Kuligowski, Andrew, "You Can RUN; But Your Data Cannot Hide: Advanced Methods to Introduce External Data into the SAS® System", *SAS Users Group International Conference Proceedings*, 1998

SAS Institute Inc., *SAS® Companion for the Microsoft Windows Environment*, Cary NC: SAS Institute Inc., 1996. 302 pp.

SAS Institute Inc., *SAS® Language: Reference, Version 6, First Edition*, Cary NC: SAS Institute Inc., 1990. 1042 pp.

SAS Institute Inc., SAS® Technical Note TS-325, *The SAS System and DDE*, Cary NC: SAS Institute Inc., 1995. 18 pp.

SAS Institute Inc., SAS® Technical Report P-222, *Changes and Enhancements to Base SAS® Software, Release 6.07*, Cary NC: SAS Institute Inc., 1991. 344 pp.

Author Contact Information

Howard Schreier
Stop H-2815
U.S. Dept. of Commerce
Washington DC 20230

(202) 482-4180
Howard_Schreier@ita.doc.gov