Demystifying Extended Tables

Stephen Davidson, Dartmouth Medical School

Abstract

The SAS/AF® product allows SAS® users to create powerful interactive tables called extended tables for information display or entry. However, creating an extended table can, at times, seem to be a very imposing and mysterious task. This tutorial will deal with the when, why, and how of extended tables. We'll see when it is appropriate to use one, why use one, and how to build one. We'll build both introductory and advanced tables along the way as examples. This paper assumes a basic knowledge of the SAS/AF Screen Control Language, and will be useful for people who have never used extended tables before or those would like to understand them better.

Introduction

Applications developers and SAS programmers have been using SAS/AF software to develop powerful applications for years. Version 5 of the SAS System® allowed us to develop full screen applications with the power of the SAS procedures behind it. Version 6 of the SAS System gave us SCL (Screen Control Language) which dramatically increased the development capabilities of SAS/AF. And with the birth of SCL, came extended tables. This powerful tool is often shunned as being too difficult or cumbersome to use, and is often pushed aside in favor of other tools. This paper will examine extended tables and how they can work for the applications developer. Let's start by examining some basic questions about extended tables.

What is an extended table?

An extended table is, at it’s simplest, an interactive tool to list SAS data sets. It can be much more than that though. It allows us the ability to enter data and update data sets using screens that are customized to the application begin developed, and, in advanced usage, can be completely data driven to allow the writing of one extended table that can be used in many different places or in many different cases. They have been unfairly labeled as being too difficult or complicated to use as they have a different program flow. We’ll talk more about that later.

When should an extended table be used?

Ask this question to a lot of programmers and they’ll say never! Actually, many programmers don’t hesitate to use extended tables, provided the situation is right. But when is it right? Basically, if you need more control of a list than SCL will give you from one of it’s standard commands such as DATALISTN or VARLIST, then an extended table may be right for the job. Also if there is any complex updating of a data set that needs to be done, but you need more control over the screen than SAS/FSP®’s FSVIEW or FSEDIT will give you then an extended table may be right. There are times when a SAS/FSP product or an extended table could be used interchangeably. At these times, programmers tend to go with whatever they are comfortable with. Given that SCL is available for use within the SAS/FSP products, there is a lot of overlap between the two choices, so neither is a bad choice.

An example of a program where an extended table is used quite often is one where the programmer needs to display a selection list from a data set from which multiple selections can be made. The DATALIST/DATALISTC statement only allows one selection to be made. Hence, the use of an extended table from which many selections can be made. There are ways to accomplish the problem using the DATALIST statements, but none as elegant or powerful as extended tables. In short, there are some cases that lend themselves quite well to extended table use, otherwise it’s a matter of the programmer's personal preference.

1. This paper was written based upon version 6.06 (6.07 Unix) of the SAS System and SAS/AF. Hence, some of the new features in versions 6.08 and after may invalidate some of the statements contained herein.
Why use an extended table?
Again, it's mainly a matter of personal preference. The main advantage of using extended tables is the amount of control that the programmer has to customize a screen or list to meet the needs of the user. Another advantage is the ability to create data-driven selection lists. The major disadvantage of extended tables is the time spent coding and testing the program as compared to the simple invocation of an SCL program statement such as DATAINSTN or SHOWLIST. Some of the deficiencies of these statements can be overcome by creative programming, but there are some things these statements weren't designed to do. This is where extended tables are so valuable.

Basics of Extended Tables

Setup of Program entry
Extended tables are more than just SCL program statements. The program has to be "informed" that an extended table is expected of it via system options field of the general attributes (GATTR) screen as seen in figure 1, and by the occurrence of "xxx" on the display screen to indicate that the list should repeat everything below the "xxx" as shown in figure 2. At this point we may begin to enter code for our extended table. Before we examine the actual code itself, let's look at the different statements and labels that make our simple program an extended table.

SETROW
The SETROW function is the activator for the extended table. Usually coded in the Init section of a program, when this function is called, the actual functioning of the extended table is performed. The statement syntax is:

CALL SETROW(num-rows, num-sel, sel-order, dynamic));

where num-rows is the number of rows in the extended table, num-sel is the number of selections that can be made, sel-order controls the position of the selected rows and dynamic is a flag indicating that a dynamic extended table is to be created.

GETROW
GETROW is a label indicating a block of code that is to be processed in presenting the data in the extended table. This is where data preparation occurs in displaying the table.

PUTROW
PUTROW is another label indicating a block of code that is to be processed for every row selected. This is one place where validating or error-checking, or updating can be done.

TOPROW/CURTOP
TOPROW is a function that scrolls a row to the top of the table. It has the following syntax:

CALL TOPROW(row);

where row is the number of the row to be scrolled to the top.

CURTOP is a statement that returns the number of the row current at the top of the table. It has the following syntax:

row = CURTOP();

where row is the number of the row at the top that is returned by the statement.

_N_ CURROW
_N_ CURROW_ is the automatic variable generated by the extended table and updated after every pass through the GETROW section. It is roughly equivalent to the _N_ automatic variable in SAS data step code. _CURROW_ increments for every row processed with _CURROW_ indicating the current row of the extended table that is currently being processed.

Now that we've seen the pieces that make a basic extended table,
let's look at how these pieces fit together.

Program Flow

The flow of an extended table is described in chapter 5 and figure 5.1 of the SAS Screen Control Language Reference and/or chapter 6 and figure 6.1 of the SAS Technical Report P-199: Using SAS Screen Control Language in Release 6.06. In description format, the flow of the extended table starts like all SAS/AF programs, in the INIT section. Usually at the end of the INIT section of code is the SETROW call which invokes the extended table. Prior to this invocation, data set opening and preparation are usually done. After execution of the SETROW, control is passed to the GETROW section. This section is processed once for every row of the table displayed. This means that if there are fifty observations in the data set to be processed and your extended table only has fifteen rows, the GETROW section is processed only fifteen times, once for every row to be displayed. It will then process the next fifteen as they are scrolled up onto the display. Now the program waits for input. If input comes from above the "***" (i.e. above or not from the extended table), the MAIN section of code is executed, just as in a normal or non-extended table program. However, if a field in the extended table is modified, control is passed to the PUTROW section which is executed once for every row of the extended table modified. After these executions, control is passed to the MAIN section which is processed just once. After the MAIN section is finished, control passes back to the GETROW section again and the cycle is repeated. If scrolling is done on the table, the GETROW sections are re-executed with the new set of rows displayed. These cycles is repeated until the user issues END or CANCEL. Upon an END command, the MAIN section is executed if any fields were modified, or, if not, it proceeds directly to the TERM section. If CANCEL is entered, control is passed directly to the TERM section for program cleanup and termination.

Figure 3 contains the sample code for the basic extended table example started in figures 1 and 2. Figure 4 contains the output generated by the extended table. For the examples, the SASUSER.CLASS data set was copied into the WORK library and used as a basis for all of the data.

If the SETROW command from figure 3 is examined, we can see that it has two parameters, the first indicates the number of rows in the table while the second tells us that this is a display-only table as no selections can be made.

We've seen the code used to generate a simple extended table, but now let's take our example a little further by adding the ability to update our CLASS data set.
Table Updates

Updating a data set via an extended table is really just a simple extension of our basic extended table example. Basically, we have to remember to open our data sets in extension of our basic extended table data.

`PUTROW` section II>

Updating a `FETCH` statement to move the data pointer to update, and an `UPDATE` statement to actually perform the update. We must be careful of the `FETCHOBJS` though, as a default `FETCHOBJS` will also perform a `SET` command which would overwrite our newly changed variable with their prior value to prevent this.

Our program looks the same as figure 4, with the exception that all fields are now accessible to tab and modify. After return is pressed and the `PUTROW` executed, the new values we have entered are updated to the data set.

We've seen how to update a data set with a fixed number of records in it. But what if you now need to add rows to a data set in addition to updating. Our current program can't handle this. We now have need for a dynamic extended table.

Dynamic Extended Tables

A dynamic extended table is one where the number of rows to be processed is not known. This is a fairly common occurrence when working on data sets that need to have records added or deleted, as well as data driven extended tables, where table content and makeup may differ from one task to the next.

A dynamic extended table is differentiated from a standard extended table in two ways. First, the `SETROW` function has a first parameter or number of rows of 0 (zero) and a last parameter or dynamic flag set to "Y". Second, the `GETROW` section needs some way to end the table after the maximum number of rows. This is accomplished with the `ENDTABLE` function. It has the syntax:

`CALL ENDTABLE();`

By using this in conjunction with the `FETCH`, `FETCHOBJS`, or whatever means used to load data into the table, the dynamic table is created. Once the loading method determines it's at then end of the file, the `ENDTABLE` function is called to mark the end of the extended table. These concepts and statements are shown in figure 6.

We've also added one more field to the program, but this time its

```c
/* Program : update.program */
/* Purpose : Extension of basic example of extended/ +
table to display updating the CLASS */
/* Date : 1/25/93 */
/* Code by : Steve Davidson */
/* Dartmouth Medical School */
/* History : 1/25/93 - Initial Code */

.Init:
rc=pmenu('ok');
dsid::open('work.class','U'); /* open class dataset */
call set(dsid); /* links screen to dsn vars */
nobs::set(dsid,'nobs'); /* get num rows for table */
call setrow(nobs,0); /* create ext table */
return;

FETCH:
if fetchobs(dsid,_currow_) ge 0 then
  _msg_="ERROR : An error has occurred";
return;

PUTROW:
rc=fetchobs(dsid,_currow_,'noset'); /* get rec*/
rc=update(dsid); /* update data set record*/
return;

Main:
return;

Term:
rcclose(dsid);
return;

Figure 5
```

```c
/* Program : dynamic.program */
/* Purpose : Extension of update example of extended/ +
table to display a dynamic ext table. */
/* Date : 1/25/93 */
/* Code by : Steve Davidson */
/* Dartmouth Medical School */
/* History : 1/25/93 - Initial Code */

.Init:
rc=pmenu('ok');
dsid::open('work.class','U'); /* open class dataset */
call set(dsid); /* links screen to dsn vars */
call setrow(0,0,0,'Y'); /* create ext table */
return;

FETCH:
if rc ne 0 then do:
  _msg_="ERROR : An error has occurred";
  end;
  return;

PUTROW:
rc=fetchobs(dsid,_currow_,'noset'); /* get rec*/
rcclose(dsid); /* update data set record*/
return;

Main:
if modified(add) then
  rc=append(dsid,'noset'); /* add a record*/
return;

Term:
rcclose(dsid);
return;

Figure 6
```
us to add a record to our class data set. The screen layout is shown in figure 7 and an example of the output of the extended table is shown in figure 8. When the Add Record button is selected, the MAIN section of code executes and a blank record is appended to the data set. Control is then returned to the extended table and we can fill in our blank row just as if we were modifying any of the other rows.

We've now covered the basic topics regarding extended tables by exploring basic and dynamic extended tables, and updating data sets via extended tables. Let's take a look at a few other things tools that can make an extended table more user friendly.

Other Topics

Multiple Column Extended Tables

It's possible to "trick" the extended table into thinking you've got more than one column of table data. The basic idea here is that sometimes it's preferable to have wider table with more information on it, than it is to have a narrow column that needs to be paged through. For example, let's look at figure 8 again. Imagine that instead of 19 rows displayed, we had 300 rows to display. In this case, it's not much fun to sit and scroll through page after page after screen after screen of names looking for one individual. One solution to this is to widen the screen and double the amount of information on it. Figure 9 shows the display screen used for this example. We've repeated the columns to make a left side and a right side of names. Figure 10 shows the code used. For this example, two temporary SQL views were created to illustrate one way to perform multi-column tables. One was created for Males, the other for Females. The table being created is dynamic, and as we read in a record from each data view, we assign the record to its appropriate side. We use the Mdone and Fdone flags to indicate the end of a data set. Only when both data sets are done do we issue the ENDTABLE function. The results of this table are shown in figure 11.

Sorting an Extended Table

Another common request for extended tables is the ability to sort by any of the columns. This can be accomplished by the use of choice groups in our extended tables. If we make the column headings choice groups, we then have the ability to click on a column and have our data sorted by it. This can be demonstrated by expanding upon our last example. Figure 12 shows the display
program: multi-col.program

Purpose: Ext table the display CLASS data set

which males in one col and females in other col as an exam of a multi-column
other col as an exam of a multi-column

ext table:

purpose: Ext table the display CLASS data set

program: multi-col.program

init:
return;

date: 1/25/93

code by: Steve Davidson

dartmouth Medical School

go history: 1/25/93 - initial code/

length name $8 sex $1 age height weight $;
/define dem vars/

init:
rc=pmenu('ok');
submit continue sql;
/*create temp views of M & F*/
create view work.classf as
select *
from work.class
where sex='M';
create view work.classm as
select *
from work.class
where sex='F';
endsubinit; /*create our temporary views of M & F*/
dm=open('work.classm','u'); /*open class view for M*/
dsf=open('work.classf','u'); /*open class view for F*/
call set(dsm); /*link dsn to program vars*/
call set(dsf);
call setrow(0,0,' ','Y'); /*create ext table*/
return;

getrow:
if _currow_=1 then do;
/*reset end of table flags*/
mdone='N';
dfopen='N';
end;
/*reset end of table flags*/
rc=fetchobs(dsm,_currow_); /*get a record*/
if rc ne 0 then do; /*oops .. error*/
if rc=-1 then mdone='Y'; /*eof male recs .. set flag*/
end;
/*valid record .. move vars to screen*/

else d0; /*valid record .. move vars to screen*/
name=name; sex=sex; age=age; height=height; weight=weight;
end; /*valid record .. move vars to screen*/
rc=fetchobs(dsf,_currow_); /*get a record*/
if rc ne 0 then do; /*oops .. error*/
if rc=-1 then dfopen='Y'; /*eof female recs .. set flag*/
else_ms9_='ERROR: An error has occurred:
end;
/*valid record .. move vars to screen*/
if mdone='Y' and dfopen='Y' then
endtable(); /*no more rows*/
return;

putrow;
return;

main:
return;

term:
rc=close(dm);
rc=close(ds);
rc=delete('work.classf','view'); /*erase temp views*/
rc=delete('work.classm','view');
return;

Figure 10

screen for a sorting table. Notice that each of the column heading
is now a screen variable. All of the columns on the left have been
defined as choice group male while all on the right are choice

group female. Figure 13 shows the code for this example. We can

program: sorted.program

Purpose: Extension of multi-col program

that shows use of choice groups and sorting in an ext table.

date: 1/25/93

code by: Steve Davidson

dartmouth Medical School

go history: 1/25/93 - initial code/

length name $8 sex $1 age height weight $;
/define dem vars/

init:
rc=pmenu('ok');
submit continue sql;
/*create temp views of M & F*/
create view work.classf as
create view work.classm as
where sex='M';
where sex='F';
create table work.classm as
create table work.classf as
select * from work.class
select * from work.class
where sex='M';
where sex='F';
call setrow(0,0,0,0,' ','Y'); /*create ext table*/
return;

getrow:
if _currow_=1 then do;
/*reset end of table flags*/
mdone='N';
dfopen='N';
end;
/*reset end of table flags*/
rc=fetchobs(dm,_currow_); /*get a record*/
if rc ne 0 then do; /*oops .. error*/
if rc=-1 then mdone='Y'; /*no more male records*/
else_ms9_='ERROR: An error has occurred:
end;
/*oops .. error*/
else d0; /*valid record .. move vars to screen*/
name=name; sex=sex; age=age; height=height; weight=weight;
end; /*valid record .. move vars to screen*/
rc=fetchobs(ds,_currow_); /*get a record*/
if rc ne 0 then do; /*oops .. error*/
if rc=-1 then dfopen='Y'; /*no more female records*/
else_ms9_='ERROR: An error has occurred:
end;
/*oops .. error*/

Figure 10 (cont)
sex2=sex;
age2=age;
height2=height;
weight2=weight;
end; /* valid record... move vars to screen */
rc=fetchobs(dsf,_currow_);
get a record
if rc ne 0 then do; /* oops... error */
if rc=-1 then fdone='Y'; /* no more F records */
else _msg_='ERROR : An error has occurred';
end;
/* oops... error */
else do;
valid record... move vars to screen
name2=name;
sex2=sex;
age2=age;
height2=height;
weight2=weight;
end;
valid record... move vars to screen
if mdone='Y' and fdone='Y' then
call endtable(); /* no more rows */
Return;
Putrow:
Return;
Main:
i, t
isactive('male') then do; /* M ch grp was picked*/
select (i,sactive('male')); /* find sort var*/
when (1) rc=sort(dsm,'name');
when (2) rc=sort(dsm,'sex');
when (3) rc=sort(dsm,'age');
when (4) rc=sort(dsm,'height');
when (5) rc=sort(dsm,'weight');
othersize;
end; /* find sort var */
end; /* M ch grp was picked*/
if isactive('female') then do; /* F ch grp picked*/
select (isactive('female')); /* find sort var*/
when (1) rc=sort(dsf,'name');
when (2) rc=sort(dsf,'sex');
when (3) rc=sort(dsf,'age');
when (4) rc=sort(dsf,'height');
when (5) rc=sort(dsf,'weight');
othersize;
end; /* find var to sort by */
end; /* female choice group was picked*/
Return;
Term:
rc=close(dsm);
rc=close(dsf);
rc=delete('work.classm','view'); /* del temp views */
rc=delete('work.classf','view');
Return;

Figure 13 (cont)

see that the data set that is to be sorted is determined by the choice group selected. Then the data set is sorted on the variable that was selected. The data underlying the extended table is being sorted, not the table itself. Each time it repeats the GETROW cycle, it uses the newly sorted data and returns the data to the table in their new order. Figure 14 shows the results of this example. We see the Males sorted by Height and the Females sorted by Weight.

Searching an Extended Table

Another common request for extended tables is the ability to search through a table to find a value. This task cannot be accomplished easily using SCL statements though. The most efficient way to do this is to use the SAS window commands KEYFIELD and FIND. KEYFIELD defines a field on the screen to search and FIND searches that field for a match. For an example of this, we return to our most basic extended table. First, to illustrate the effect more clearly, a temporary data set is created that has nine times the number of rows of the original class data set. Figure 15 shows the screen display for this example. Here we can see the SEARCH and SRCNAME screen variables. This is where we'll enter our name to search for, then click on SEARCH when it's entered. Upon clicking on SEARCH, the KEYFIELD and FIND window commands will be executed and the search performed. The SCL code for this example can be seen in figure 16 and the results in figure 17.
Program: search.program
Purpose: Extension of basic ext table to show how to search an extended table using the find command
Date: 1/25/93
Code by: Steve Davidson
Dartmouth Medical School
History: 1/25/93 - Initial Code

Init:
r0-omen('OK'); submit continue; /*create temp dsn to use*/
data work.classbig(drop=z onamel);
set work.class;
output:
oname=name;
do %,,1 to 9;
name=compress(oname)|put(z,l.l);
age .. age+z;
height=height+z;
weight=weight+z;
output;
end;
run;
endsubmit; /*create temp dsn to use*/
dsid=open('work.classbig'); /* open class dsn R/O */
call set(dsid); /* links screen to dsn vars */
nobs=attrn(dsid,'nobs'); /* get num rows for table*/
call setrow(nobs,Oj; /* create ext table */
Return;
Getrow:
rc=fetchobs(dsid,_currow_); /* get a record */
if rc ne 0 then do: /* oops .. error */
  if rc=-l then
    msg_='ERROR: End of data set reached';
  else
    msg_='ERROR: An error has occured';
  end; /* oops .. error */
Return;
Putrow:
Return;
Main:
if modified(search) then /*if search selected, then find value*/
call execute('keyfield name, find *'1 I srcname I I .j;
Return;
Term:
rc=close(dsid) ; Return,

Summary
This paper has presented usages and examples of SAS/AF extended tables. We've started from the basics of extended table usage and examined different uses for these tables. Common requests from users of extended tables have been examined and examples given of their usage. We've shown that extended tables are one of the most powerful tools afforded us by SAS/AF, but it is not always appropriate to use anywhere. If desired, SCL list statements such as DATALISTS, SHOWLIST, or VARLIST are much more efficiently used because of their speed in implementation. However these tools don't give the level of control that extended tables do. Extended tables are an effective and efficient tool that will allow you to get the most out of your SAS/AF software.

References

Acknowledgments
The author would like to thank Don Henderson, David Septoff, Greg Smith, and the rest of the staff at SAS Consulting Services, Inc. in Rockville, MD for their help, patience, and friendship in developing the SAS skills necessary to create this and other papers and programs.

Suggested Papers
Building Data-Driven Object-Oriented Applications: Selected Techniques by SAS Consulting Services Staff, SAS Consulting Services, Inc.

Contacting the Author
Stephen Davidson
218 Strasenbury, HB 7251
Dartmouth College
Hanover, NH 03756
Phone: (603) 650-1971 / Fax: (603) 650-1153
Internet: stave.davidson@dartmouth.edu

SAS, SAS/AF, and SAS/FSP are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.