The purpose of this brief overview is to provide the commercial applications programmer/analyst with both a general approach, and also a startup coded framework, for producing interactive stand-alone data assembly and retrieval systems to meet virtually any normal business need. While this may seem to some like a fairly ambitious undertaking in a presentation of this short length, there is a real point to be made here concerning the versatility which this SAS system allows.

To try to step around several tempting, but for our purposes unprofitable sidelines of inquiry, our discussion will exclude the low-probability fringes of system design which demand highly specialized methods of data extraction or rigorous statistical interpolation.

We will instead focus on the broad central band of applications programming which analysts such as we rely upon to respond to everyday business problems.

SAS has features enough to address the entire spectrum of electronic data interchange. Frequently, however, its remarkable statistical abilities are allowed to overshadow its equally comprehensive provisions for powerful, reliable, yet on balance, simple data entry and retention devices.

The modular system examined here attempts to appropriate some of the features and grace notes found in successful mainframe systems. For example, it will support almost any desired level of restricted access; it could examine a datum character by character, or substring it and compare string groupings to a full entry. It will even accustom itself to its human environment by allowing aliases and tolerating a fair degree of colloquialism or outright keypunch errors.

Yet the methodology used to build this system is neither revolutionary, nor is its code particularly abstruse, using, as it does, a mere fraction of SAS's capabilities.

On the contrary, this system was developed to answer a small-system business problem in a restricted timeframe. The examination and explanation of this code is aimed at the analyst like myself, down in the corporate tunnels at the cutting edge of the user/analyst interface.

I particularly recommend it to the self-taught programmer who more or less inherited the job merely by being on the spot with knowledge which no one else had. Expertise is where you find it, and many a manager drafts a motivated amateur to write local systems when the corporation's mainframe administrators are too slow, or simply uninterested.

Most of these small and forthright systems exhibit three general phases. We can group all the efforts to enter the system and add or change data as THE IN. We might usefully combine the emission of reports and direct screen summaries under the general heading of THE OUT. And the effort to make something of the data which has been assembled, which usually goes on in background driven by the source coded itself, we can generalize as THE CRUNCH.

If we split our basic system into these three segments, and consider each of them as a self-contained subsection of the entire development process, then it is possible not only to address contiguous and complimentary actions in one, but also to endow our resulting code with the advantage of deriving from a truly modular approach from the very start.

THE IN

Use a SAS macro to handle all of the data-input and variable definition chores. Use a second macro -- or a series of multiple macros -- to refine raw entries or to expand shorthand entries into full data strings.

A common usage for such a macro is to expand a datum. For example, expand a person's initials into a full name. Construct a table of name values against possible initials and pass the variable through the table as it is being read in by invoking that particular macro at the appropriate moment.

My own approach is to separate all the variables which are likely to require expansion, and then assign a separate logic sieve and a macro name to each one. I have found it particularly easy to keep track of the variables themselves, and also to maintain good control of the point at which they undergo expansion.

THE OUT

Not all systems require data to be returned piecemeal to the CRT screen. However, this sort of interactive, instant reportage was a must in the sample lodging and accommodations software. In this case, the operator receives data by telephone (from corporate travelers) and by mail (from hotels
and airlines requesting payment). The operator must be able to call up a particular traveler's current reservation, or at times an entire travel history for an individual or department. Moreover, invoices are sent for random time periods, and must be matched with data from the lodgings file without time-consuming hardcopy lookup.

Thus, the lodging and reservation system required a screen output, for which SAS macros are tailor made. MACRO COMEC, as an example, shows all reservations made against a particular charge account within some range of dates.

Note how the macro first dynamically allocates a flat output file to receive the report. This was intended to duplicate the live screen output for recordkeeping purposes. It then formats a query screen which asks the operator for a charge number and a range of dates. Now, follow the comments within the macro itself and watch the resulting inquiry against the database, and the formatting of the screen report.

This general scheme may be used for any sort of on-line inquiry against a SAS database. Indeed, MACROS INVOICE and LOG are merely variations on this same subroutine, as a glance at their code structure will confirm.

THE CRUNCH

In some cases, data can be processed, and then output, exactly as entered without internal manipulation. However, as we shall see, if SAS/FSP is being used as an entry vehicle, it might profit the programmer/analyst to include a logic sieve just downstream of the FSP screen exit. It is this logic sieve which forms the CRUNCH in the lodging system.

MACRO UPDATE

The FSP screen into which the operator enters data is actually only a transaction dataset which, after the operator exits from the FSP screen, is run against the master dataset to produce an update. Data is never added to the master set unless it passes successfully through the logic sieve.

Each observation from the transaction FSP screen passes through the sieve. If it falls all the way through without tripping any of the error messages by failing to satisfy the error condition within a particular loop, the observation is added to an update dataset. This dataset, if it contains anything, is always updated to the master dataset.

Up to now, we have seen a standard use of the UPDATE command. But, if an observation from the transaction set trips one of the error loops on its way down through the sieve, an error flag is set which, at the end of the sieve, routes it to a holding WORK file. The contents of this file then become the new transaction dataset.

When the FSP screen is next visited by the operator, he/she will see any errors left over from the last session. If left unfixed, they will continue to loop through the logic sieve and back to the transaction set indefinitely, until they are either corrected or deleted.

An accompanying feature is the on-screen error log, which exhibits each observation found to contain one or more errors. Macro variables are used to assign an error message value, and create a new observation found to a holding file to be printed on the screen. Thus, if one observation generates seven errors (perhaps seven different variables in that observation were incorrectly entered), then the holding file ends up with seven different observations.

If they are sorted and their key variables are indexed using the FIRST or LAST features, a file can be written to the CRT screen which contains the observation's key, a list of variable values which the logic sieve found in error, and the accompanying error message.

As soon as the operator exits from a data entry session, he/she is presented with an on-line list of errors, and specific messages telling how to correct them. The operator can elect to fix them right away, or to come back later, whereupon he/she will find them waiting in the FSP transaction set.

The rest of the process of writing our system is merely to identify the actions we wish to take, and then to split them into as many modules as we find convenient.

Our code can be divided along a large number of lines within these general sections.

The place where we make the split into discrete macro calls depends largely upon our own convenience. That is to say, we should section our code so that it will be easy to maintain and enhance. In this respect, SAS's extraordinary permissiveness in allowing many different ways to achieve a particular result will simplify the splitting.

The first step for an analyst/programmer who approaches his/her first macro system is to try to re-orient old habits of thought, particularly if she/he has written linear system in the older SAS versions. Instead of a code which runs top to bottom, the first executable line in your new system will be the last line in the dataset. Instead of a code which runs through each data step automatically as it is taken up in sequence by the SAS processor, your macro modules will run only when you call them from the CRT screen or
within the program by the source code itself.

The system herein described provides a sturdy framework within which to hang virtually any linear working code. A chief attraction aside from its simplicity is its portability and ease of maintenance. But its main virtue, and indeed the attribute of SAS macro programming which should endear it to any analyst, is its universality. You can quickly adapt it to accept literally any applications program, written for any client, to do any sort of standard business work.

GENERAL SYSTEM OPERATIONS

Here's how it works. In TSO, a CLIST allocates the SAS storage dataset and any other required operating system files. It then starts the SAS processor with a standard SAS OPTIONS (SYSIN=) statement.

The allocations may just as easily be done using SAS's TSO prefix within the SAS code itself. In fact, the output macros do make dynamic allocations of expendable output files which transit the output lattice as simple hardcopy vehicles, and are, at the last, deleted to save space. However, I prefer to use the CLIST because it provides a handy reference of allocations without executing a LISTALC.

The processor begins at the first line of the SYSIN file, and falls through each of the macros without execution, to the last line of code. This line calls a macro from within the SYSIN file itself, which clears the operator's screen and generates the primary menu (MACRO MAINSCR).

The menu offers the operator a choice of responses, each of which are themselves additional macro calls. If the operator enters anything else, the processor simply refreshes the screen.

Auto-refreshment will recur throughout the system wherever it may logically take the place of an error message. I have found this preferable to messages for systems with a high operator turnover, since it saves code, and since the most lucid of error messages is still interpretive, while a refreshed screen is not.

NOTE: Auto refreshing is achieved by interrogating a command interpreter which resides in a separate macro. At sign-on, the first executable line of code calls a macro (MACRO MAINSCR) which clears the operator's screen, generates the main menu screen, and waits for an interactive response from the keyboard.

The operator then keys in a choice from the screen. The macro passes this response to the interpreter (MACRO MAINMAC), which compares it to its table of valid commands. When found in the table, the interpreter calls the appropriate macro. If the command is not found, control returns to the menu generator (MACRO MAINSCR), which simply refreshes the screen. This continues until the driver sees a valid command.

The enclosed framework includes a macro call on the primary menu which calls a secondary report screen. The reports are segregated to a secondary screen merely to avoid cluttering the primary menu. Otherwise, they could easily be contained therein. Each report request, when entered, is passed to the same interpreter (MAINMAC again).

Each report macro is, in reality, a self-contained mini system of its own. It requests one or more specific data ranges from the operator, assembles data from storage within the ranges, and outputs a PROC PRINT to the screen. If no data is found, it so indicates on the screen, and reprints the ranges supplied. This, and so the operator to recognize a keypunch error, or to supply other ranges. Whenever the processor encounters errors, whether from screen entries or in resident data, it tries to answer the three questions:

What values am I actually reading?  
What do I find that seems wrong?  
How can it be fixed?

If the error messages answer these three questions clearly, and if the error itself is left in unresolved stasis until the operator acts to fix it, most errors can be corrected almost as they are made.

Let us now examine those macros which are to drive the basic system. Although we must limit our discussion for reasons of space, I have also a written copy of the system-specific lodging and accommodations code for your interest, and so that you may view the system as an integral whole. Please refer to the macro calls in the attached source code as we proceed.

MACRO MAINSCR

This is not only formats the main menu, but it is also the first executable statement in our SYSIN dataset. The code phrasing is a straightforward series of macro puts and accompanying variables. At the end, the system pauses and waits for a value to be passed to it interactively from the operator.

Upon receiving the value (NREPLY), it calls the internal interpreter (MACRO MAINMAC) and passes it the new value. This series of executions reiterates each time the system operator responds.

MACRO MAINMAC

It functions as interpreter of interactive commands and as the main driver of
the system, from which the system macros are actually called. This body of code extends to us the two concepts of concentration of function, and of a system's responsibility to adapt to the human condition.

MAINMAC examines the value passed to it by the operator, and compares it against a sheaf of valid responses. It resembles a standard lookup table in operation, but its translational features carry its function beyond a simple interrogation.

In the first four lines, the interpreter will find that the values QUIT quit Q and q are equally acceptable to trigger a logoff. Similarly, the values INVOICE or invoice will invoke an on-line routine which matches a specific invoice (see MACRO INVOICE later) to selected stored data...but so will INV or inv.

By extending this alias function one step further, the system makes an effort to decrease the pedantry of many systems which demand replies in all caps, or in the singular only. For example, the secondary report screen can be called with the obvious command REPORT (or report), but the system will also tolerate its plural REPORTS and its noncaps version, in an attempt to meet normal human memory halfway; it will even accept the attenuated R or r alias.

These macro calls are non-consequential in that the operator is never penalized for making a wrong entry or choice beyond a simple screen refreshment; nevertheless, some care should be taken to build explicit meaning into each one-word macro invocation to avoid confusion for the beginner, and to decrease the turnover learning curve.

Grouping the command interpreter functions into one module not only provides an instant system overview, but greatly simplifies future maintenance. To remove a function from the system, simply comment out, or delete, its MAINMAC lattices. To add a new function, add it here. At the same time, build in a cushion of aliases to soften the system's inevitable abrasion against its human exosphere.

COMBINING WITH SAS/FSP

I have not specified SAS/FSP as a prerequisite for using this macro framework, even though it offers by far the most efficient of all the alternatives for data entry.

Since the TSO environment provides the programmer/analyst direct access to flat files and PDS members; it is quite possible to construct a perfectly acceptable data entry tool by using TSO's resident language itself. Or, the analyst can create a macro using only the code available in MACRO MAINSCR, which will accept data from the screen and pass it to a WORK or SAS data step.

Aside from gaining the advantage of further practice in the macro language, I feel that creating a macro for this sort of work might be more trouble than it's worth. SAS/FSP will do the job far more quickly, with less effort on the programmer's part, and it presents additional retrieval features such as the well-known command line specifiers which would be prohibitively costly to duplicate in linear, or even in macro SAS code.

One of SAS/FSP's features which we may wish to augment with our own code is its data-evaluation ability. For example, its background processor can evaluate for data which exceeds a maximum and minimum value, but it cannot evaluate for more than one value within that range. It can accept the value 234 as being >100 and <=300, but it cannot determine that only the values 234, 289, and 216 are valid.

Therefore, we must supplement the FSP processor with our own logic sieve, which you will find in our discussion of MACRO UPDATE.

SUMMARY

The above discussion includes the highlights of writing in the SAS MACRO environment, but certainly falls far short of describing the enormous possibilities of macros. For simple, direct systems built upon the IN-CRUNCH-OUT format, macros, in my opinion, provide the absolute irreducible minimum of efficiency versus versatility.

When development time is short—as it almost always is when a local manager or client is looking over the analyst's shoulder and wondering politely when the operators can get started—then use the above described framework to take care of the interactive housekeeping chores. This frees you to concentrate on the details of linear structure within.

Or, use it to rough out a quick interactive system to get data in and basic reports out, and then spend time refining it later on.

In either case, or in situations which I cannot even begin to guess, if you feel my advice would be profitable, please call or drop a note. I would be pleased to share knowledge and experiences with anyone, professional or amateur, who is interested in business system design.
1 The attention paid to this feature will pay particular dividends in systems used by a high turnover of operators. Moreover, since systems like these usually feature little human engineering, your efforts are at least likely to produce astonishment, even if no outright thanks.

2 The final statement in each table assigns the original entered value to the expansion variable if no valid value has been encountered so far. This pops errors or omissions right out in front of the operator and saves writing validation subroutines and accompanying error messages.

3 The SAS UPDATE command will change the master values for any observation which exists in both datasets, and will add a new observation to the master set for those which exist in transaction but not in master.

4 Attached is a sample system which contains all codes except the report outputs. They are of little interest, being merely an assembly of standard formatting statements.

5 We can avoid complexity by not addressing the security implications of this environment. However, I should point out that it would be very simple, at sign-in, to interrogate a file for an employee number or password, and then produce a menu screen carrying only certain of the available macro commands, or to call a supervisor subroutine to monitor and stamp entries, and so on to almost any level of complexity desired.

6 Passing interactive commands through an interpreter not only allows aliases, but also frees macro calls from the constraints of SAS naming conventions. Screen entries can thus be tailored to fit local habits of jargon, or even colloquial speech patterns, which will humanize and simplify command recognition and system operation.

7 This implies that any macro call can be made by the operator from any screen in the system, which, of course, it can. Thus, the experienced operator can bypass the menus entirely after launching the first call from the primary menu.

8 In SAS Version 4 a macro statement group must appear upstream of its call.