ABSTRACT

The SAS® Screen Control Language Source-Level Debugger is a MUST for developing SAS/AF® applications. This presentation will cover using system return codes, system messages, and the debugger to debug code. The debugger is a powerful utility that allows stepping through an executing SCL program, examining or monitoring values of SCL variables, checking program flow and logic, bypassing groups of statements, displaying arguments passed to a called program, and changing values of SCL variables during execution. Tips on how to use the debugger when testing a series of nested programs will also be discussed.

INTRODUCTION

In the early days of programming, programmers placed numerous "PUT" statements in the source code of misbehaved programs in order to trace the program flow and examine the value of variables to find the error(s). It usually took many iterations of placing "PUTS" in the source code, recompiling, and testing the program to find the cause of the misbehavior, or better known as the "bug". Then the bug had to be fixed, the PUT statements removed, and the program recompiled. This is the 90's! You can throw out those "PUT" statements. With the SCL debugger, you compile only ONCE with the Debug option on, and then interactively, you trace the program flow, examine the values of variables, set up breakpoints and variables to be watched, alter the value of variables, and more. There is no compiling over and over and no removing a multitude of "PUT" statements from the code once the bug has been found. However, it is recommended that once the program has been fixed to recompile it with the Debug option off because there is additional code and overhead when the debugger option is on. The program code size will be smaller when it is compiled without the debugger.

The SCL debugger is a source level utility which works in a windowing environment with both SAS/AF programs and SAS/FSP® software. This paper will address only SAS/AF and the SCL debugger.
THE DEBUGGING SESSION

To execute program(s) with the debugger executing at the same time is a two step process. First, the debugging environment has to be prepared and then the debugger has to be started.

PREPARING THE DEBUGGING ENVIRONMENT

Three methods may be used to create the debugging environment.

METHOD 1 Prepare Individual Programs for Debugging.

Select a program from the catalog directory window. On the command line of any of the program windows such as the display window, the source window, the attribute window, type "Debug on" and press Enter. Next type "compile" on the command line. Notice the code size. It is larger due to the overhead of the debugger. Repeat the above steps for each program that you want the debugger active when the program is executing.

Method 2 Preparing all the Programs in a Catalog for Debugging.

Do this when invoking PROC BUILD. Use the compile statement with the debug option. For example,

PROC BUILD c=userapp.cat;
    compile debug;
run;

Method 3 For Programs that have Already Been Compiled with the Debug Option On.

In this case, it is not necessary to compile the programs again when invoking PROC BUILD to have the SCL debugger active. This is accomplished by omitting the Compile statement and specifying the "Debug" option on the PROC BUILD statement rather than the Compile statement. For example,

PROC BUILD c=userapp.cat debug;
run;

Caution:
If you have finished a debugging session, exited from PROC BUILD, returned to the SAS® Display Manager System, and then you decide to do some more debugging and reenter PROC BUILD, you should not be surprised if the debugger is not activated when you execute the same programs. You know that you have compiled them with the Debug option on and successfully ran them with the SCL debugger active. Now these same programs execute but the SCL debugger is not active. You do not need to compile these programs again with the Debug option on. The problem is that whenever beginning a debugging session the debugging environment has to be prepared using one of the three methods explained above, preferably method 3 in this case. Anytime you want to do a debugging session, you must prepare the debugging environment either by invoking PROC BUILD with the
debug option on as in Method 3 or by invoking PROC BUILD without the debug option on and from
the command line, type "debug on" and press Enter. Now, you are ready to test the programs and
the SCL debugger will be active with the programs that were previously compiled with the "debug
on" option.

ACTIVATING THE DEBUGGING SESSION

There are two methods to start a debugging session depending on whether the program(s) you want
to debug has submit blocks of SAS statements that you want to process.

Method 1  No Submit Blocks to be Processed.

Issue the "testaf" command either on the command line from one of the program windows or from the
catalog directory window, type "t" in the field preceding the program name.

Method 2  Submit Blocks are to be Processed during the Debugging Session.

Exit from PROC BUILD. In the display manager program window, on the command line, type the
following command to start the application with the SCL debugger in effect:

   AF c=userapp.cat.entryname.program debug=yes

CHARACTERISTICS OF THE DEBUGGING SESSION

During the entire debugging session, a split window will appear on the screen. The upper window is
called the SCL debugger Source window. It displays the program source code, and the line of
source code where execution is suspended is highlighted. The lower window is the SCL debugger
MESSAGE window. The bottom line in this window is the SCL debugger command line. It is
preceded by the "DEBUG>" prompt. Usually, you enter SCL debugger commands from here.
However, notice that both the SOURCE and MESSAGE windows have a command line at the top of
each of these windows. From the command line at the top of either window, display manager
commands and either SAS/AF or SAS/FSP global commands, depending on which product you are
executing, can be entered. Also, SCL debugger commands can be entered at the command line at
the top of both windows but must be preceded by the word "SCL".

The HELP debugger command is one of the most important commands. It will display a one page
list of topics ranging from an Overview to the individual debugger commands. Until you are familiar
with this product, it is useful to start each session by viewing the Overview before beginning to
debug. The Overview is an one page concise description of all the debugger commands. Then if you
need more detailed information on individual commands, you can select them individually from the
Help Main Menu or by typing "help command name" at any time during the debugging session on
the MESSAGE window command line.

The MESSAGE window is the active window; however, you can make the SOURCE window the
active window by typing the SWAP debugger command. This is useful if one window is covering up
too much of the other or if you want the SOURCE window in the foreground while examining the
code in order to decide where to set breakpoints and which variables are to be watched. From this
point forward, any references to command line will refer to the command line in the MESSAGE

1490
window. Any references to the command line in the SOURCE window will state SOURCE command line.

To stop the debugging session, type the QUIT debugger command on the command line.

A command must fit on one line. Multiple commands separated by a semicolon may be entered on the command line as long as they do not exceed the length of the command line.

DEBUGGING COMMANDS

The commands and their various options are divided into the following levels of expertise: Beginners, Intermediate, and Advanced. These levels are only recommendations. You should use any or all commands as soon as you want or need to. The reason for dividing them up into levels is to keep the amount of new information to be learned to a minimum initially. Abbreviations exist for the commands but will not be used in this paper.

BEGINNERS COMMANDS

The STEP command executes one statement and then suspends execution. If you wanted to, you could go through the entire program one statement at a time, but that would be extremely tedious. However, you will want to issue the STEP command many times during the debugging session. In order to minimize the amount of typing, the STEP command can be assigned to the Enter key. You do this by typing "enter step" on the command line. Now, whenever you want to issue the STEP command to the SCL debugger, you merely press the Enter key but there must not be a command on the command line. If you type "step 10" on the command line, the SCL debugger will execute the next ten program statements and then suspend execution, waiting for you to specify the next action.

The GO command executes all the program statements until the next statement where a previous debugger command has specified execution to suspend. If there are no such prior debugger commands in effect, the GO command will cause the SCL debugger to continue execution of the program until it is finished without suspending execution again. If you type "go main", the SCL debugger executes all statements until it reaches the beginning of the MAIN section of the program and then suspends execution to wait for you to specify the next action. If you type "go 100", the SCL debugger executes all statements until it reaches statement 100, provided statement 100 exists.

The EXAMINE command causes the SCL debugger to display the value of a variable(s). If you type "examine salary", the SCL debugger will display the value of the variable salary. If you type "examine _all_", the SCL debugger will display the value of all the variables in the program. This command is very useful when execution is suspended at various points in the program.

The SET command causes the SCL debugger to change the value of a variable to a value that you specify. If you type "set salary=100000" and then issue the Step or Go command, the variable salary will be set to 100,000 and that is the value that will be used in any further program calculations. This command is very useful if you think you have discovered what the bug is and want to fix it on the run to verify you get the program results you would expect if the error had not been in the program.
INTERMEDIATE COMMANDS

If you think you know approximately where in the program the bug is located, you may want the SCL debugger to execute all the statements until this point and suspend execution. Then you can set up other statements where you want execution to suspend, and variables whose values you want displayed so that you can determine what the problem is. This approach takes you right to the statement you want to stop at without tediously issuing Step commands to get there.

You need to know the difference between SCL executable statements and non-executable statements. There are just a few non-executable statements. They are as follows: Array, Entry, Length, Replace, statements within Submit blocks, and comments. All other SCL statements are executable.

The BREAK command stops execution at a particular program statement, which is referred to as a breakpoint.
If you type "break 1119", the SCL debugger will stop execution at source code statement 1119, provided that it exists and that it is an executable statement.
If you type "break main", the SCL debugger will stop execution at the beginning of the Main section of the program.
Caution: Breakpoints can only be set on lines with executable code. You should read the message stating the breakpoint was successfully set.

The DELETE command will remove a statement(s) from the list of breakpoints.
If you type "delete break 1119", the SCL debugger will no longer suspend execution at this statement.
If you type "delete break _all_", the SCL debugger will remove all breakpoints.

The WATCH command causes the SCL debugger to suspend program execution whenever certain variables that you have specified change in value. The old and new values of these variables are displayed automatically when the program suspends execution. You do not have to enter the Examine command. The Watch command is very useful when you think that a variables or variables are not being calculated correctly or set to the correct values. If you type "watch salary, payrate, hours", whenever any of these three variables change in value, the program stops executing and the old and new values of the variable(s) that have changed are displayed.

The DELETE command will remove a variable(s) from the list of watched variables.
If you type "delete watch salary", the salary variable will no longer be watched, but payrate and hours will continue to be watched.
If you type "delete watch _all_", the SCL debugger will remove all watched variables.

The PARM command causes the SCL debugger to display what values are being passed to a function or another program with the Call Display statement before passing execution to the other program. If you type "parm" and the program statement is "Call Display('Salary',hrrate,yrsal)", then the SCL debugger will display the values of hrrate and yrsal.
If a program statement has nested function calls, you need to type a Parm command for each function. For example, if the program statement is x=round(sqrt(y),.01);, then you need to type "parm;parm" on the command line.

The ARGS command causes the SCL debugger to display what values were passed to a called
program from a calling program if the called program is using an Entry statement to receive the
values and execution has passed to the called program. If the program statement
"Call display('Salary', hrrate, yrsal)" has been executed and the SCL debugger has suspended
execution at the beginning of the "Salary" program and the "Salary" program has the following
statement "Entry hrrate 4.2 yrsal 8.2", typing "args" on the command line will cause the SCL
debugger to display what values were passed in for variables hrrate and yrsal to program Salary.

The CALC command acts just like a calculator and will allow you to calculate a value that you would
like to set a variable to. For example, to set Salary to 100,000 * 1.5, you type on the command line
"calc 100000*1.5" and the SCL debugger will display the result. It is useful to use CALC and then
the SET debugger command to calculate a new value for a variable and then to set the variable to
this value.

The DESCRIBE command causes the SCL debugger to display a variable’s attributes, such as name,
type, and length. If you need to know a variable's length, type, etc., type "describe variable name "
on the command line. For example, you would type "describe salary" on the command line, and the
SCL debugger will display salary's name, type, and length.

The STEP OVER command causes the SCL debugger to stop execution at the next statement
immediately following a Call Display, Call FSEDIT, or CALL FSview statement. The SCL debugger
must currently be positioned waiting at the Call Display, Call FSEDIT, or Call Fsview statement for
your next command. Issuing the Step Over command three statements prior to the Call Display or
Call FSEDIT statement will not get the above result. You only need to use the Step Over command if
the program that is being called was compiled with the Debug on option and you do not want
execution with the SCL debugger active passed to it. The called program will not be executed at
all using STEP OVER.

The JUMP command causes the SCL debugger to skip backwards or forwards to any executable
statement that you specify. The skipped statements are not executed. One use for the Jump
command is to leave a DO loop after having observed enough iterations.

ADVANCED COMMANDS

The TRACE command will let you follow a program’s execution without stopping execution like a
breakpoint does. If you type "trace 20 80 120" on the command line, the SCL debugger will display
a message for each statement in the Message window, but the program continues executing without
requiring a command from you. Each of the statements referred to in the trace command are called
tracepoints.

The LIST command causes the SCL debugger to display all the breakpoints, watched variables, and
tracepoints in effect. For example, if you type, "list _all_" on the command line, the SCL debugger
will display the currently executing program’s breakpoints, watched variables, and tracepoints.

The DELETE command can remove breakpoint(s), watched variable(s), and tracepoint(s).
If you type, "delete trace _all_", the SCL debugger will remove all the tracepoints.
If you type, "delete trace main", the SCL debugger will remove all tracepoints in the Main section of
the program.
If your application consists of a series of nested programs or entries, SCL creates an execution stack which records which programs have already been called and what program is currently executing. The execution stack is maintained using the last-in, first-out algorithm. When a program is called, it is added to the stack. When it terminates, it is removed from the stack. For example, if program A starts the application, it is put on the stack. Program A has a Call Display statement calling program B. Program B is placed on the stack, and the order of the stack is program B, then program A. Program B has a Call Display statement calling program C. Program C is placed on the stack, and the stack order is programs C, B, and A, with C being the last one placed at the top of the stack. Program C terminates first, and the stack order is programs B and A, with program B being last on the stack. Program B terminates, and the stack order is program A. Program A terminates. The application is finished, and the stack is eliminated. Understanding how the stack works will allow you to switch from program to program and experiment on the run with the application execution path and vary it from the normal execution path without changing the programs' code and recompiling them.

The TRACEBACK command will cause the SCL debugger to display the execution stack for the application. The list of programs that have been called would be displayed with the currently executing program listed first, and the first program to execute listed last. It also tells which line in the calling program called the called program. This information is useful if you want to return to a calling program, do some processing that would change the values for the parameters passed to the called program, and then return to the called program with these new values. You will discover other more sophisticated reasons for altering the execution path. The next command will allow you to alter the execution path.

The ENVIRONMENT command allows you to refer to a program that is not the one currently executing, that is, not the last one on the stack. You can refer to a prior program in the stack that either called the current program or that called a program that called a program on the stack, etc., that called the current program. For example, if program A called program B, and program B called program C, and program C is the current program in the execution stack, if you want to return to program A, you type "environment A\50". You will be returned to program A statement 50.

Caution: Even though you have returned to program A for debugging purposes, the execution stack still considers program C the current program. You do need to return to program C in order for it to terminate normally, close datasets, etc. To do this, type "environment run".

Some of the debugger commands previously explained can be used on programs in the application stack, not just the current program on the stack.

If you want to display the value of a variable in a program that is not the current program, type "examine entry-name\variable name". For example, if program C is the program currently executing, and it was called by program B, and program B was called by program A, you would type "examine A\salary" to display the value of variable salary in program A.

If you want to set a breakpoint in a program that is not currently executing, type "break entry-name\label or line number\". For example, to set a breakpoint in program A at statement 100 when program C is currently executing, you would type "break A\100". To set a breakpoint at the Main section of program A, you would type "break A\main".
If you want to delete breakpoints, tracepoints, or watched variables in a program that is not the currently executing program, you would type "delete <break[trace[watch] entry-name\<label or line number>".

For example, to delete a watched variable Salary in program A when program C is the currently executing program, you would type "delete watch Asalary".

If you want to find out the attributes of a variable in a program that is not currently executing but is in the application execution stack, you would type "describe entry-name\variable name". For example, to display the attributes of the salary variable from program A, you would type "describe A\salary".

If you want execution to continue without stopping until you are at a statement in another program in the execution stack, you would type "go entry-name\statement number". For example, if you want to go to statement 200 in program C and you are currently in program A, you would type "go C\200".

If you want to display a list of breakpoints, watched variables, or tracepoints for a program that is in the execution stack but not the currently executing program, you would type "list entry-name\_all\_". For example, to display program A's breakpoints, watched variables, and tracepoints, you would type "list A\_all\_". To display program B's watched variables, you would type "list B\watch".

If you want to set a variable in a program that is in the execution stack but not the currently executing program, you would type "set entry-name\variable expression". For example, to set the salary variable in program A to 500000, you would type "set A\salary=500000".

If you want to set up a watched variable in a program that is in the execution stack but not the currently executing program, you would type "watch entry-name\variable". For example, to make the salary variable in program A a watched variable, you would type "watch A\salary".

There are more options like the WHEN and AFTER that can be used with many of the debugger commands to specify the conditions when to execute the debugger command; otherwise, the debugger command will not be executed. The DO option can be used to execute a series of debugger commands once execution has suspended. You might want to reset all the values of an array after the nth iteration of a program DO loop.

**SAS RETURN CODES AND SYSTEM MESSAGES**

For debugging purposes, the SCL SYSRC function is not that useful because it returns numerous numeric codes that would have to be looked up. What is very useful in debugging is the SCL SYSMSG function because it returns an error message describing the most recent error or warning. Caution: Always capture the result of SYSMSG immediately after it is called because it is reset to blank after it is called. For example, in your program, you might want to know if a fetch statement resulted in any data. After the program statement with the fetch SCL function, you would have a statement like "fetchmsg=sysmsg()". Now, when you are debugging the program, you can use the "examine" debugger command on the variable Fetchmsg and display a meaningful message in most cases.
TIPS ON USING THE SCL DEBUGGER WITH NESTED PROGRAMS

During a debugging session working with nested programs, it may appear that the SCL debugger stops running and the program just finishes execution without allowing you to stop execution at various places, examine variables, or do any of the debugger functions. One possible cause for the debugger to deactivate is that it was given an ambiguous command or told to go to a non-executable statement. For example, the third program on the stack is currently executing, and you issue a "go init" command. Which INIT in which program should it go to? In the currently executing program, telling the debugger to "go init" would be ambiguous because the init section only executes once.

CONCLUSION

The SCL debugger is a very powerful tool which can increase productivity. Like any tool, there is a learning curve; however, to shorten the learning time, use the on-line HELP command every time you are in a debugging session. Before you start execution, type "help" on the command line and read the Overview. It will refresh your memory of all the commands before you start debugging, and it will help you to learn the SCL debugger faster. You should remember when you are finished debugging and you have fixed the problem to turn the debugger off and recompile the program. Happy Debugging!

REFERENCES


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