

The DATA Step Debugger

Paul Hamilton
Fred Hutchinson Cancer Research Center
HIVNET/SC

Abstract

The DATA step debugger is a newly released, frequently neglected component of the SAS® Software System. Descended from the Screen Control Language debugger, it provides rich functionality for finding logical or data-driven errors in the DATA step.

Traditionally, SAS programmers and applications developers have relied on the use of **PUT** statements embedded in the aberrant DATA step in order to examine the inner logic and sequencing of the code. Once the **PUT** statement was coded to print out the relevant data values at the relevant time, the problem (although perhaps not the solution) became readily apparent. However, this often required repeated tries, considerable time pouring through the SASLOG, and a substantial amount of effort. Using **PUT** was further complicated if the DATA step involved **DO** loops of one sort or another.

This tutorial will examine the tools provided by the DATA step debugger which aid in program correction and validation. We will also examine the role of the debugger for both the novice and accomplished SAS user.

Introduction

The SAS System has always offered tremendous power and flexibility: myriad procedures make common tasks simple, while the DATA step provides all the capability of a procedural language. This power can come at the cost of some complexity for the programmer learning the DATA step. Invariably, a DATA step will err—that is, the results will be unexpected or undesirable. There are generally four reasons why this can happen:

1. A syntax error exists in your DATA step. The DATA step debugger will not help in this case. The debugger will not load if the DATA step cannot be compiled.
2. There is an error in the data coming into the DATA step. The solution in this case is to correct the data.
3. Your DATA step contains a logic error. This may be due to misunderstanding of a statement's function or purpose, an incorrect algorithm, or simply getting lost in the complexity of the problem.
4. An interaction of case two and three. These are some of the most subtle, hard to detect, and annoying errors. They usually come about because the programmer has an incorrect or incomplete knowledge of the processes underlying the data input stream, and the DATA step does not correctly handle the unexpected data condition.

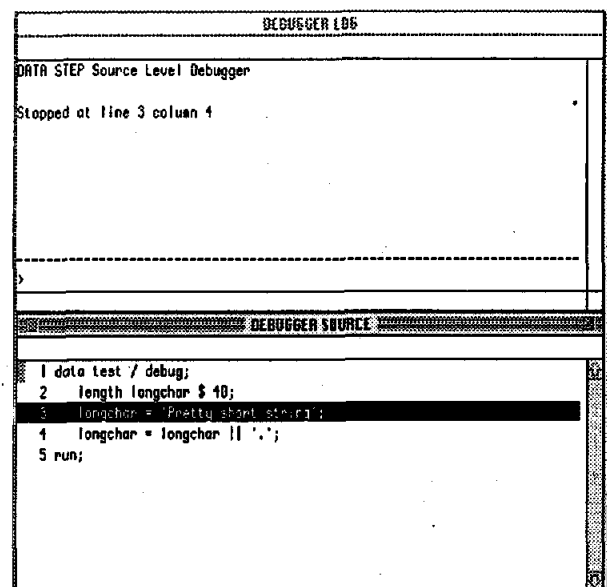
In the distant past, programmers solved these problems by staring at code and data printouts for hours, and thinking very hard. This was a tortuous and not terribly productive process. Traditionally, SAS programmers insert **PUT** statements inside the aberrant DATA step, writing out intermediate values from the Program Data Vector (PDV) to the SASLOG. At some point, useful information illuminating the error would appear, and the nature of the problem would become self-evident. However, this often took many repeated invocations of the DATA step, with the **PUT** statements being placed according to logic, intuition, or sometimes desperation. The DATA step debugger makes this technique archaic.

The Basics

The DATA step debugger is only available in the context of an interactive SAS session. Starting the debugger is extremely simple. Recall the DATA step into the program window, and change the DATA statement to include the keyword **DEBUG**, after a slash:

data test;	* before debugger;
data test / debug;	* Invoke debugger;

SUBMIT the DATA step for execution. After the DATA step compiles, the debugger **SOURCE** and **LOG** windows appear.



Your DATA step code appears in the bottom (SOURCE) window; DATA step debugger commands are issued in the debugger LOG window shown above.*

Using the Debugger

The debugger is designed to step through your DATA step code at a pace determined by the programmer. To prepare for using the debugger, it is helpful to format your code in a specific style. Place only one statement on each physical line in the Program Editor, and place the IF... statement part separate from the ELSE ... part. If you format your code differently, the debugger will function but will be more difficult to use. If the DATA step hits an end of data condition, the debugger will stop just as the DATA step itself does. You can quit the debugger at any time by issuing the QUIT command.

Consider the following DATA step:

```
data test;
  length longchar $ 40;
  longchar = 'Pretty short string';
  longchar = longchar || '.';
run;
```

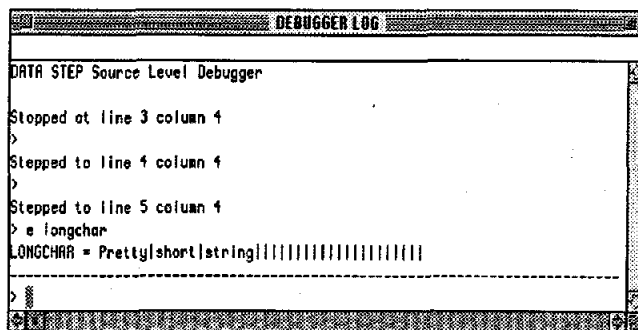
The SAS novice might wonder why the resulting string *longchar* does not end with a period. The code produces no error messages or warnings, just unintended results. Modify the **DATA** statement to include the / **DEBUG** option and **SUBMIT** the new step. With the LOG window active, simply press the ENTER key repeatedly. This is set by default to the **STEP** command, which executes the current statement (shown in reverse video) and move the pointer to the next executable statement. After the initial assignment of *longchar* in line 3, issue the **EXAMINE** debugger command to display the contents of the PDV: **E LONGCHAR**. The value is just as expected (Pretty short string). Pressing the ENTER key again executes the next statement, which you might believe should add a "." to the string. If we **EXAMINE** *longchar* again, however, we find the period was not added; the value remains unchanged.

After thinking about the code a bit, the programmer might begin to wonder what is being stored in *longchar* aside from the visible text. One way to discover this is to insert a new statement after line 3, utilizing the **TRANSLATE** function to change all occurrences of the blank character to a vertical bar "|".

* Further references to the LOG window are meant to indicate the debugger LOG, not the usual Display Manager LOG window.

```
data test / debug;
  length longchar $ 40;
  longchar = 'Pretty short string';
  longchar = translate(longchar, '|', ' ');
  longchar = longchar || '.';
run;
```

Run the debugging DATA step again, and **EXAMINE** *longchar* after line 4.



Perhaps things are now becoming clearer. This program is attempting to add a new character "." on the end of a 40 byte variable that is already filled with text and blanks. This understanding leads to the correct solution, which is to **TRIM** the trailing blanks from *longchar* before attempting to append the period.

```
data test;
solution;
  length longchar $ 40;
  longchar = 'Pretty short string';
  longchar = trim(longchar) || '.';
run;
```

Unexpected DATA Step Behavior

Sometimes the DATA step compiler does something you might not expect. Perhaps you have multiple years of sales figures, and want to perform some numeric calculation that involves different constants for various ranges of years. One approach is to use the **ARRAY** statement and a **DO;...END;** loop to process all the sales variables. Once inside the **DO;** loop, use **CALL VNAME** to determine which year is being processed. You might reason that since **CALL VNAME** is designed to return the name of the variable into the second parameter (here called *varname*), that the compiler would automatically pre-define *varname* for you as character 8. However, when you run this DATA step, it appears that any variable or code depending on *varname* is not working correctly.

```
data test / debug;
  array sales (*) y89 - y97;
  do i = 1 to dim(sales);
    call vname(sales{i}, varname);
  end;
run;
```

Trying this step in the DATA step debugger, you observe the following debugger LOG after issuing the command **E I** VARNAME:

```

DEBUGGER LOG
> ex i varname
| = |
VARNAME = .

```

Why is the debugger reporting *varname* as a missing numeric? To ensure you understand how *varname* is defined, issue the command **DESCRIBE VARNAME**:

```

DEBUGGER LOG
> desc varname
Name = VARNAME, Type = NUM, Length = 8

```

Now the problem is readily apparent. Despite what your assumptions might be about what the DATA step compiler should do, it has in fact allowed *varname* to default to a numeric variable. This program needs a character variable to hold a character value, so the solution becomes self-evident:

```

data test;
solution;
  length varname $ 8; * define var
yourself!;
  array sales (*) y89 - y97;
  do i = 1 to dim(sales);
    call vname(sales{i}, varname);
  end;
run;

```

Dealing With Incorrect Data

When unexpected results appear in a calculated value, the DATA step debugger simplifies tracking down the problem. Consider the following merge program:

```

data dept;
merge
  pnwsug.sdates /* contains start date
*/
  pnwsug.edates; /* contains end date
*/
by dept;

```

```

flow = enddate - strtdate;
format
  flow 2.
  strtdate enddate mmdyy8.;
run;

```

The results from **PROC PRINT** look normal, except that the output also contains an unexpected result:

DEPT	STRTDATE	ENDDATE	FLOW
A1A	12/27/59	01/01/60	5
B2B	10/24/97	11/02/97	9
C3C	10/30/97	10/31/97	**
D4D	05/03/52	05/18/52	15
E5E	09/01/56	09/06/56	5
F6F	05/14/49	05/31/49	17

Why does *flow* have ** printed for department C3C? It certainly appears that moving from 10/30/97 to 10/31/97 should result in a reasonable 1 day flow time. Note that only one observation has unexpected and inexplicable results. When we invoke the DATA step debugger, it would be tedious to **STEP** through and **EXAMINE** every observation, when only one department seems erroneous. There are excellent tools available in the debugger for just this type of purpose.

The initial debugger command will be **BREAK 7 WHEN DEPT='C3C'**, which translates into "Set up a breakpoint on line 7, but only break when the variable *dept* takes on the value C3C. Establishing a breakpoint in your code gives the debugger a place at which to pause and permit further keyboard commands. In this case, it will execute the DATA step normally until the third observation, then pause just before executing line 7. Note the "!" on line 7, denoting a breakpoint placement.

Rather than pressing the **ENTER** key many times, issue the **GO** command instead. The DATA step will execute normally, with no interaction with windows, until the condition set on the **BREAK** command is reached. Then, execution pauses and command returns to the debugger LOG window. Press the **ENTER** key one more time to have *flow* be recalculated for observation 3.

```

DEBUGGER SOURCE
2 data dept / debug;
3 merge
4   pnwsug.sdates
5   pnwsug.edates;
6 by dept;
! 7 flow = enddate - strtdate;
8 format flow 2. strtdate enddate mmdyy8.;
9 run;

```

Based on the previous example, perhaps it would be best to ensure that the variables in the Program Data Vector are defined as expected.

```

DEBUGGER LOG
> desc strtdate enddate flow
Name = STRTDATE, Type = NUM, Length = 8
Format = MNDYYV8.
Name = ENDDATE, Type = NUM, Length = 8
Format = MNDYYV8.
Name = FLOW, Type = NUM, Length = 8
Format = 2.

```

Nothing unusual stands out from the results of the **DESCRIBE** command. Perhaps looking at the values will shed some light:

```

DEBUGGER LOG
> e _all_
DEPT = C3C
STRTDATE = 10/30/97
ENDDATE = 10/31/97
FIRST.DEPT = 1
LAST.DEPT = 1
FLOW = **
ERROR_ = 0
JL = 3

```

At this point, nothing appears erroneous. The match-merge worked perfectly, combining the correct observations from the two data sets. The values of the dates appear reasonable, yet *flow* is still overflowing two digits.

Finally, **EXAMINE** the two date variables, but use greater precision on the format used in the command. The **EXAMINE** command allows use of built-in or user-defined formats to display values. When re-displayed using 10 rather than 8 digits, the problem is obvious.

```

DEBUGGER LOG
> e strtdate enddate mndyy10. flow best.
STRTDATE = 10/30/97
ENDDATE = 10/31/9797
FLOW = 2848891

```

An incorrect date has been entered in the *EndDate* variable, 8000 years in the future! Clearly this is a simple matter of correcting invalid data; the DATA step performed as expected and desired.

This may be verified during the debugging step by using the **SET** and **CALC** commands. An enormous advantage of the DATA step debugger is the ability to change the value of variables in the PDV during pauses in execution. This is accomplished by using the **SET** command; restarting and recompiling the DATA step is not required. Once the invalid date value has been corrected, program validation can conclude by recalculating and examining the flow between the two dates. The **CALC** command can function as a handy utility function while in the debugger. However, in this debugging session it can be used to recalculate the variable *flow* now that the data upon which it depends have been corrected. This is much more convenient than quitting and restarting the debugger.

```

set enddate = '31Oct1997'd; calc flow =
enddate - strtdate; e flow.

```

Note that by using semicolons, all these commands can be entered at one time, just as in the Display Manager.

```

DEBUGGER LOG
> e strtdate mndyy10. enddate mndyy10.
STRTDATE = 10/30/1997
ENDDATE = 10/31/9797

```

Dealing with Loops

DO;...END; loops are a very powerful feature of the DATA step, greatly reducing the amount of code required for many projects. However, complex looping structures may be perplexing without the aid of the DATA step debugger. Consider the following simple financial calculation, using the simplest type of **Do** loop:

```

data loopy;
  retain
    intrate 0.10
    term 5;
  do year = 1 to term;
    funds + 500;
    interest = funds * intrate;
    cumint + interest;
    funds + interest;
  end;
run;

```

PROC PRINT reveals what may be a surprising result:

Interest Rate	Year	Interest	Funds
0.1	6	\$857.81	\$3,357.81

A few moments with a calculator will validate the interest and total fund amounts, but why does *year* have a value of 6 when the **Do** loop clearly runs from 1 to 5?

Issuing the debugger commands **TRACE ON**; **WATCH YEAR** results in the following trace log:

```
DATA STEP Source Level Debugger
Stopped at line 5 column 4
> trace on
> watch year
Watchpoint set for variable YEAR
> [the ENTER key pressed many times... ]
YEAR = 1
Old value = .
Value changed at line 5 column 4
Stepped to line 6 column 7
Stepped to line 7 column 7
Stepped to line 8 column 7
Stepped to line 9 column 7
Stepped to line 10 column 4
(lines skipped for brevity)
YEAR = 5
Old value = 4
Value changed at line 10 column 4
Stepped to line 6 column 7
Stepped to line 7 column 7
Stepped to line 8 column 7
Stepped to line 9 column 7
Stepped to line 10 column 4
YEAR = 6
Old value = 5
Value changed at line 10 column 4
Stepped to line 11 column 1
```

The first interesting thing to note is where the debugger starts executing the code. The first statements is *Stopped at line 5 column 4*. The **RETAIN** statement on lines 2-4 is not an executable statement; during the compile phase it declares two numeric variables in the PDV, sets their values as indicated, and sets a *retain* flag to prevent them from being automatically reset to missing values (.) during the execution phase.

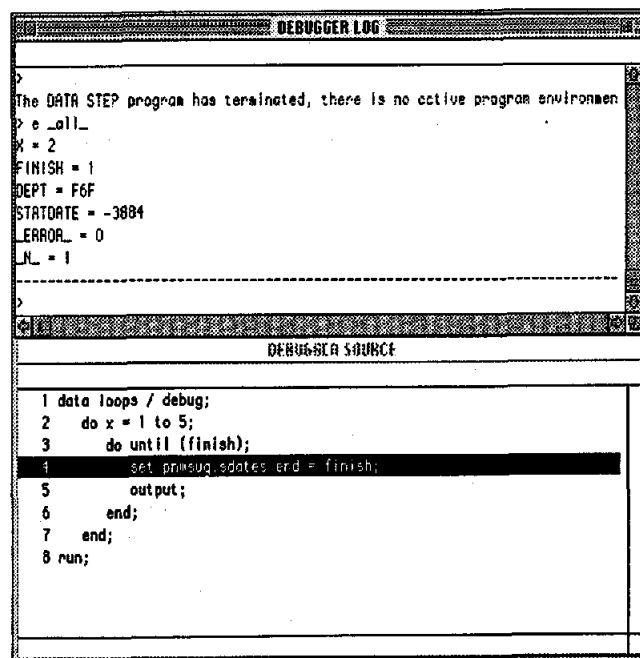
As the DATA step debugger steps through the program, it makes what may be an unexpected sixth pass at the **Do** loop. Watching this behavior may serve to remind the novice SAS programmer that the natural flow of a **Do** loop is to increment the loop counter (*year*) until it exceeds the loop stopping value specified on the **Do** statement. Thus the value of 6 appearing in the output from this step is a correct and expected phenomena.

Perhaps circumstances dictate that you collect multiple copies of the observations in a data set. This could be accomplished in a number of ways (**PROC APPEND** being

the most obvious), but perhaps some additional processing must occur which prescribes the use of the DATA step. The initial approach is to nest two **Do** loops together: an inner loop to read all observations from the data set, surrounded by a **Do ... To...** loop, which should replicate the data set the required number of times.

```
data loops;
  do x = 1 to 5;
    do until (finish);
      set pnwsug.sdates end = finish;
      output;
    end;
  end;
run;
```

After executing this DATA step, one might be surprised to discover that there are only as many observations as the data set being read; the code does not make multiple copies as desired. Invoking the debugger, one could press the **ENTER** key many times, until the step runs out of data. In this instance, simply issuing the **Go** command is much quicker. As shown in the logs below, the DATA step ran out of data while trying to read from the data set. Typing the command **EXAMINE _ALL_** shows that the value of the outer loop is set to 2, and the last observation read into the PDV is the last observation from the data set being read. Careful reading of the documentation of the **Set** statement reveals the problem: "Since reading an end-of-file indicator ends a DATA step automatically..."¹



¹ SAS Language, Version 6, page 485.

On reflection, perhaps 2 **Do...To...** loops would serve this problem best. The inner (read data set) loop could run from 1 to *noobs*, *noobs* being retrieved from the data set descriptor during the compile phase as shown in the previous example.

Interaction with Macros

There are 3 basic ways the macro facility can interact with the use of the DATA step debugger:

1. The DATA step you need to debug is being produced by a macro,
2. The DATA step you need to debug is producing a macro or macro call (e.g. **CALL EXECUTE**),
3. You can write macro statements to facilitate debugging a DATA step that has nothing to do with macros.

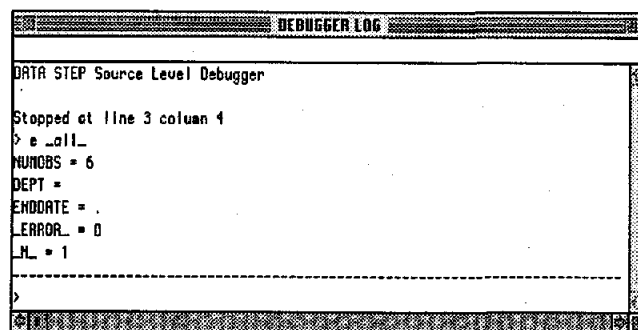
The SAS documentation (page 78) defining the DATA step debugger clearly outlines the 6 steps used to solve the first case. Techniques discussed in this paper will facilitate finding and correcting problems occurring in the second case. In the third case, defining macros can reduce repetitive typing in the LOG window; however such techniques are beyond the scope of this paper.

An Odd Technique

One of the favorite tricks employed by many macro programmers is a ingenious but obtuse method to determine if a SAS data set has any observations before executing a section of code. Without respect to any larger questions of the macro language, consider the following code segment:

```
data _null_ / debug;
  if 0 then set phwsug.edates noobs =
numobs;
  call symput('numobs', put(numobs,best.));
  stop;
run;
```

When first encountering this code, it would appear to be worthless. Line 3 creates a macro variable called *numobs* (*SYMBOL PUT*) and the use of **NOBS** option on the **SET** statement is clearly documented. However, it would appear that the code guarantees that the **SET** statement never executes, by placing a logical condition (*if 0*) which should always fail. This intuition is well placed; the statement in fact never does execute. The reason why this technique does work is that the value of **NOBS** (*numobs* in this code) is assigned during the compile phase of the DATA step, not during execution. The **EXAMINE** command below, typed before a single statement has been executed, clearly shows this to be true.

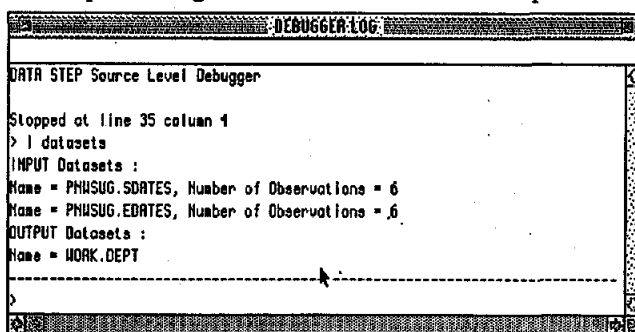


The reason this technique is so popular among experienced programmers is exactly because of the reason it's so confusing. Because very few statements execute, the DATA step accomplishes its task extremely efficiently.

Other Debugging Commands

Other useful techniques in the debugger include:

- When pausing inside a **DO; END;** DATA step loop is desired, try something like **BREAK 41 AFTER 10**, meaning "set a breakpoint at line 41, pausing after executing that line 10 times". The number specified **AFTER** should be an even divisor of your loop endpoint. For the statement **DO I = 1 TO 32 BY 4;** breaking **AFTER 4** or **AFTER 8** would be desirable.
- It is possible to build **DO; END;** and **IF ...THEN** logic into your debugging statements as well.
- The **GO** command accepts a line number as a parameter. The DATA step will execute normally, without interaction with windows, until the specified line number is reached.
- If it becomes apparent that a particular variable is causing the difficulty, use the **WATCH** command. By entering **WATCH DEPT**, the debugger will suspend execution every time the value of *dept* changes, print the new and the old values. This is much more convenient than issuing an entire series of **STEP...EXAMINE** commands.
- The debugger will list all: break points, watched variables, external files, and open data sets in your DATA step. Entering the command **LIST DATASETS** produces:



Useful for Novices

The DATA step debugger is obviously useful for correcting DATA steps that are not producing desired results. Another, less obvious use is as a training device. By invoking the DATA step debugger on code that functions properly, the novice can observe and learn important features of the DATA step:

- Learn the flow of control (the DATA step is a big loop!).
 - ♦ What happens when the **RUN;** statement is encountered?
 - ♦ What happens when using multiple **SET** statements and you read the last observation from one data set?
 - ♦ Where does execution flow after encountering a **RETURN;** statement?
 - ♦ What happens during **DO; ... END;** loops?
- Distinguish between those statements that are executable, and those that only take effect during the compile phase
 - ♦ **RETAIN**
 - ♦ **KEEP/DROP**
 - ♦ **FORMAT, INFORMAT**
 - ♦ **LABEL).**
- When using the command **EXAMINE _ALL_**, the entire PDV is printed, including temporary values that only exist during execution of the step (e.g. **_ERROR_**, **_N_**, and **FIRST.var** and **LAST.var**), and are never printed in subsequent steps like **PROC PRINT** or **PROC REPORT**.

Useful for Everyone

Even people who have used the DATA step for decades might find the debugger to be useful. Aside from being useful in understanding very intricate code with complex logic, several other circumstances arise where the debugger can prove invaluable:

- New language elements. The SAS System is not static; new statements and functionality are being added continuously. DATA steps written decades ago probably function perfectly well today, but a host of new features may be used to make old programs execute more efficiently, or simply be more explicit in meaning. When these new language elements are introduced, programmers of all levels of experience must take time to learn and understand them.
- Legacy code. Often the most troubling DATA step code to maintain is legacy code, produced by former colleagues who failed to make the code, or

documentation (if any!) clear and understandable. The debugger can provide a wonderful tool to help comprehend what is actually occurring. This is critical before proceeding to modify or extend the code's functionality.

Conclusion

The DATA step debugger is an extremely powerful tool, beneficial for virtually any project solved using SAS software, and any experience level of SAS programmer. Using the debugger, the DATA step remains as powerful as ever, but that power becomes much more accessible. The debugger is useful for learning the intricate workings of the DATA step, both as novice SAS programmers learn the basics, and for experienced people as new language elements are introduced to the DATA step.

Acknowledgments

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

References

- SAS Institute, Inc.(1990), *SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.
- SAS Institute Inc.(1995), *SAS Software Changes and Enhancements: Release 6.11*, Cary NC: SAS Institute Inc..

Author Contact Information

Paul Hamilton
Fred Hutchinson Cancer Research Center
HIVNET/Statistical Center
P.O. Box 19024 MP-900
Seattle, WA 98109
voice: (206) 667-7883
FAX: (206) 667-4812
e-mail: hamilton@hivnet.fhcrc.org