THE BUG STOPS HERE!

Dennis Heidner

Boeing Aerospace Company
1833 South 261st Place
Kent, WA 98032

The Bug Stops Here!

By Dennis Heidner
Boeing Aerospace Company

I. INTRODUCTION

The cost of software is rising, which is not a profound statement to make when you consider that we have become acustomed to the idea that software (and maintenance) will be 90% or more of the total cost of a computer system. Software is labor intensive, so as the cost of labor rises so does your software cost. But are you getting your money's worth? Software, just like hardware, has a life cycle: first there is the product conception, the investigation of the product and its market, then design, development, product test and finally delivery. But is that it? No! Most studies indicate that the largest cost of the software is AFTER the product is delivered, in what is known as the maintenance phase. (Ever wonder why the monthly maintenance costs for H-P software products are so high?)

Software maintenance generally falls into one of several different categories; they include such areas as adaptive maintenance, perfective maintenance, and simply fixing the outright program bugs. Adaptive maintenance is generally modifications made to the software product so that it remains functional; for instance, the IRS every year spends considerable time adapting their software to match the new tax laws passed by Congress. Perfective maintenance means that the software is being modified to enhance its usability or its position in the marketplace. Both of these types of maintenance generally provide a return on your time investment; however the third category, fixing bugs, simply brings the product up to what it should be, with no additional features. (Have you ever heard of a sales person bragging that they fixed 57 bugs in their product last year?)

Fortunately for most of us, less then 20% of our time is spent fixing program bugs, but would it not be nicer if we spent less than 5% of our time fixing bugs? [1] In many data-processing shops that translates into one additional head! The purpose of this paper is to present some ideas, which if incorporated into your software, will help reduce the amount of time spent tracking down nasty problems such as program aborts. The paper will cover three areas, spotting the bug, trapping the bug and finally, killing the bug!

Before we continue on, let me emphasize that the techniques I advocate in this paper are not substitutes for structured design, programming, code walk-throughs or testing! For those readers who would like to learn more about structured design, programming or testing, there is a list of references at the end of this paper. [2] [3] [4]

II. SPOTTING THE BUG

The best time to spot bugs in programs is before the product is out to the user (similiar to cleaning house before relatives visit)! This can be accomplished by establishing a rigorous test plan, which the software must pass before it's released. At the HP3000 International Conference in Anaheim, Dan Coates and Michael McCaffrey from H-P talked about the software quality assurance program that H-P has implemented. The quality assurance lab has developed over 800 stream jobs which contain more than 10,000 separate tests! [5]

Test procedures

Locating bugs is, of course, the goal of product test for several reasons; first the cost of fixing a bug once the product has been released is much higher, and second while in product test you are in a more controlled environment where you can generally locate and duplicate a bug more easily. Notice the general tone of this paragraph: we are looking for bugs, not trying to prove the program works. Let me digress another step and talk about the population of bugs. If you have a program that is one thousand lines long, and you are very optimistic, you might hope that the program is 99% free of bugs. What this means is that someplace in your program there may still be ten lines containing bugs. If you were out to prove the program was correct, the odds are that it will appear to you that it is, even though there are still a few bugs there! It is important to keep in mind Murphy's law of revelation, which is "The hidden flaw never remains hidden."

The test procedure, really, is a program written in the language of your application program. If your program is designed to control WIDGETS and use V/3000, then the native language of your test procedures is WIDGETS with the V/3000 enhancement. Most universities and colleges offer classes in programming in COBOL, PASCAL, FORTRAN, etc., but to my knowledge, there are no classes taught in programming in WIDGETS! This means that when you write your test procedure it will be a learning experience for your staff. Do not expect to have test procedures which cover all the possible cases. If you miss an important test case, this is really a bug in the test plan! It is not uncommon for the first test procedures to have as many or more bugs in them as the programs themselves!

V/3000 users have one additional problem on their hands: how to test the programs and screens in an automated manner. The only commercially available package of which I'm aware is called VTEST; written by Wick Hill Associates, it is marketed by TYMLABS [6].

It doesn't work!

We must recognize that even if we have a good test plan, there will be some bugs that are not caught. This brings up the next way that bugs are discovered: the user calls up and says, "It does not look right!". My initial response to such a general statment is quite negative; however it is our job to turn around the general reports and get the more detailed information we need. This is done by asking more specific questions. For instance, when the user reports that it does not work right, I will normally ask several questions such as: Who are you? What were you doing when it did not work right? What logon name had you used? Has this ever happened before? Is this problem preventing you from working?

Since we do not want to always be grilling our users when they believe they have spotted a bug, we must have a documented procedure for capturing as much information as possible. My first attempt at this was to beg the users to write down the information off the screen, along with the sequence of steps they were going through when the bug occurred. THIS FAILED HORRIBLY! What I found out was that most users have the same aversion for writing that I do, and when they do write, they are prone to transposing numbers. On many occasions I spent hours trying to locate a bug in the wrong procedure, because the stack marker which was written down was incorrect. The programs at our site are menu-driven, with a feature which allows the experienced user to enter in one step the commands to drop them several menus lower. In other words, if a user wanted selection #1 from the current menu, followed by choice #3 in the next level down, followed by #2 in the one below the second level, the user could enter in: 1,3,2. This is very handy for the users, but a problem for anybody trying to read the scenario that the user wrote down, which looked something like: 1,3,2,4,1,0,3,M,00007635,AC,ME !!!

There must be a better way! The good news is that there are two programs in the contributed library [7], PSCREEN and SCOPY which will copy the information from a screen to a file or the lineprinter. The bad news is that these programs only work with H-P terminals and will operate improperly if the terminal was in block mode. Where possible I set up a logon UDC so when a program aborts, the screen is automatically copied.

Although screen copy routines are a great improvement over relying on handwritten information, they provide only external information to the debugger. When the a bug occurs, what appears on the screen is almost always an incomplete picture. It would be extremely useful if, in addition to the screen copy, information about the files open, and the values of the program variables could also be saved. After spending a number of hours reading the MPE intrinsic and DEBUG manuals looking for a solution, I found it! The solution is the intrinsic called STACKDUMP. This intrinsic will copy and format the program stack markers and the data area of the stack (anybody who has had a program abort has seen these pesty markers). The person maintaining the program can then use the screen copy, the stack dump, a copy of the program PMAP, a programmer's calculator and a complete listing of the program to locate the bug accurately. Here is an example of a STACKDUMP output:


```
***      STACK  DISPLAY      ***

         S=000070    DL=177644    Z=002266
 Q=000074 P=000010 LCST= 000 STAT=U,1,1,L,0,1,CCG  X=000000

 Q=000062 P=000002 LCST= 001 STAT=U,1,1,L,0,0,CCG  X=000000
 Q=000056 P=000004 LCST= 002 STAT=U,1,1,L,0,0,CCG  X=000000
 Q=000050  P=000033 LCST= 003 STAT=U,1,1,L,0,0,CCG  X=000000

..DB..            OCTAL                      ASCII
 00000     000000 000144 000000 177777     .. .d .. ..
 00004     000000 000000 000000 000000     .. .. .. ..
 00010     000000 000000 000000 140032     .. .. .. ..
 00014     000004 000020 040000 000000     .. .. @. ..
 00020     000066 000000 000020 000000     .6 .. .. ..
 00024     000007 172623 031540 000040     .. .. 3` .
 00030     073473 010010 120004 051501     w; .. .. SA
 00034     046520 046105 020123 052101     MP LE  S TA
 00040     041513 042125 046520 020040     CK DU MP
 00044     020040 000000 000034 060304        .. .. `.
 00050     000034 040140 000000 000000     .. @` .. ..
 00054     000005 060303 000006 000000     .. `. .. ..
 00060     000003 060302 000004 177776     .. `. .. ..
 00064     000000 000106 000000 000000     .. .F .. ..
** AREA OUT OF BOUNDS **
```

Once the individuals who will maintain the code have taught themselves to how to read program variable maps and program PMAPs, this method of locating bugs is very effective. However it is generally very difficult to teach! This was illustrated to me when I began to explain to another individual in the company how the program collects all this nice information for debugging. The reponse was "How does it work over the phone?" Yes, over the phone! The team that would maintain the software was located some distance from the actual computer hardware. Thus all of our neat stack dumps and screen copies were generally useless!

After a little more careful thought, I realized that generally we do not wish to see the whole stack dump, just selected portions, so why not develope a little program which would read the stack dump from the file, and display only what you asked for? This was the birth of a program called ADPAN [8] (Application DumP ANalyzer).

Due to problems with the STACKDUMP intrinsic, I wrote my own stack dump facility which I call SNAPSHOT. When SNAPSHOT is called it creates a dump file, then copies an exact image of the data stack to the file, along with information on the MPE files which were open and in use at the time. This snapshot of the process is then later analyzed by running ADPAN.

ADPAN has seven different screens of information which can be displayed; they are: CODE, DUMP, FILES, FILE nn, FLUT, INFO, and TRACE.

The TRACE screen is probably the most important of the screens. This screen displays the procedure names, segment names, p-relative address, Q address and the status for each of the markers in the SNAPSHOT. This allows the user of ADPAN to locate the cause of a program error quickly without needing to refer to a PMAP or have a programmers calculator handy. The TRACE screen looks like:

```
ADPAN 7/83 - Rev 1.1 (C) The Boeing Co, Seattle WA
DUMP: D1921810.PUB.GOODIDEA          PROGRAM: ADEMO.PUB.GOODIDEA

  Q    L   SEGMENT NAME     PROCEDURE NAME  P'REL        STATUS

 00174     ERRORHANDLER     SNAPSHOT        00123 UM,XIN,TRAPS,L,CCG
 00122     ERROR'HANDLER    OVERFLOW        00004 UM,XIN,TRAPS,L,CCG
 00114 ? SL %0173           P'REL = %011026       UM,XIN,TRAPS,L,CCL
 00057     HELP'HELP        OOPS            00005 UM,XIN,TRAPS,L,CCL
 00050     ADPAN'DEMO       PROCEDUREB      00006 UM,XIN,TRAPS,L,CCL
 00044     NEXT'BEST'THING  PROCEDUREA      00002 UM,XIN,TRAPS,L,CCE
 00040     ADPAN'DEMO       SUPERPROGRAM    00035 UM,XIN,TRAPS,L,CCE
 00033 S $MORGUE            TERMINATE'            PM,XIN,L,CCG
```

In this and other examples of screens from ADPAN, the entire line of interest (normally highlighted on HP terminals) is shown underlined.

The CODE screen displays the decompiled code around the PCAL instruction currently being examined by ADPAN. Since not all terminals are capable of scrolling, ADPAN breaks the code down into three regions, and simulates the scrolling programically. Here is a code screen:


```
000004 031003  2.  PCAL 3
000005 004000  ..  DEL ,NOP
000006 031004  2.  PCAL 4
000007 031400  3.  EXIT 0
000010 176031  ..  LRA  P+31  ,I,X (PB+000041)

000011 035002  :.  ADDS 2                        SUPERPROGRAM    <==PROC
000012 004000  ..  DEL ,NOP
000013 021004  ".  LDI  4
000014 033406  7.  LLBL 6
000015 031007  2.  PCAL F'ARITRAP
000016 000707  ..  DZRO,DZRO
000017 021002  ".  LDI  2
000020 172003  ..  LRA  P+3   ,I   (PB+000023)
000021 031011  2.  PCAL FMTINIT'
000022 140005  ..  BR   P+5        (PB+000027)
000023 000014  ..  NOP ,DIVL
000024 044105  HE  LOAD P+105 ,X   (PB+000131)
000025 046114  LL  LOAD P+114 ,I,X (PB+000141)
000026 047400  O.  LOAD Q+ 0  ,I,X
000027 040403  A.  LOAD P+3        (PB+000024)
000030 034403  9.  LDPN 3          (PB+000033)
000031 021005  ".  LDI  5
```

The DUMP screen displays either an area around the current stack marker or a specific region in memory. The user has a choice of OCTAL, HEX, DECIMAL, CHARacter and NOCHARacter formats. The DUMP screen is the default screen. (Any other screen can be requested from the DUMP screen.) For example:

```
ADPAN 7/83 - Rev 1.1 (C) The Boeing Co, Seattle WA, JUL 14 1983
DUMP: D1921810.PUB.GOODIDEA          PROGRAM: ADEMO.PUB.GOODIDEA
Q%000057 P=%000006 X=%000000 STAT=%060703 S=%000071 DL=%177740

ADDR                              DATA
 000036 000047 061305 000005 000000 000003 061304 000004 .'b......
 000045 000000 000007 060705 000004 076400 000000 000004 ....a...}
 000054 000000 000006 060703 000007 000001 010550 111401 ....a....
 000063 000065 000152 111401 000065 140001 000012 135635 .5.j...5.
 000072 000000 001000 000000 000000 000005 177766 000001 .........
 000101 000002 141001 000002 000000 177747 000016 000173 .........
 000110 000052 000004 011027 062573 000035 000001 000115 .*....e{.
 000117 000004 000005 062302 000006 177777 000011 110223 ....d....

>D Q-1;A    'aC'
>D Q-1    %060703
>D Q-1;L    %060703   TRUE
>D Q-1;H    61C3
>D Q-1;I    25027
>D Q-1;D    1640169479
```

Several important items should be noted. The first is that ADPAN will locate and highlight the current stack marker. In our example above this was done by underscoring. Next is that the DUMP screen actually has three separate windows: the header, the data area and the command window. ADPAN uses cursor addressing (if possible) to implement wraparound scrolling within the command window.

The FILES screen allows the user to identify the MPE files that the program had open at the time of the SNAPSHOT. The information displayed includes file number, file name, file options, access options, record size, current record pointer, the number of logical records processed, and the file limit.

F#	FILENAME	FOPT%	AOPT%	RECSIZE	RECPT
3	FTN06	000614	001401	-81	167
4	FTN05	000244	001400	-80	167
5	D1921809.PUB.GOODIDEA	000000	000001	128	3

The FILE nn screen allows a user to ZOOM in on a specific file and look at virtually all attributes for the file. In this example we will zoom in on file number five.

```
FILE NAME IS D1921809.PUB.GOODIDEA
FOPTIONS: STD,FEQ,CCTL,F,*FORMAL*,BINARY,NEW
AOPTIONS: WAITIO,BUF,DEF,NOLOCK,SREC,WRITE
RECORD SIZE:   128     BLOCK SIZE:   128   (WORDS)
RECPTR:          3     RECLIMIT:       400
LOGCOUNT:          3   PHYSCOUNT:          1
EOF AT:          3
FILE CODE:     0       # OF USER LABELS:  0
FILE SYSTEM ERROR:     0
```

If the program being examined was written in FORTRAN, the user of ADPAN can request that the FORTRAN LOGICAL UNIT TABLE be displayed; this is the FLUT screen.

UNIT	F#	FILENAME	FOPT%	AOPT%	RECSIZE	RECPT
6	3	FTN06	000614	001401	-81	167
5	4	FTN05	000244	001400	-80	167

The INFO screen lets the user review the general PREP capabilities of the program. In addition the INFO screen displays information on the way the program was segmented, data stack utilization information, and any run-time INFO strings or parms.

```
ADPAN 7/83 - Rev 1.1 (C) The Boeing Co, Seattle WA, JUL 14 1983
DUMP: D1921810.PUB.GOODIDEA          PROGRAM: ADEMO.PUB.GOODIDEA
Q=%000057 P=%000005 X=%000000 STAT=%060703 S=%000071
PROGRAM CAPABILITIES=BA,IA                 SNAPSHOT ID: 1

    STACK INFORMATION                      CODE SEGMENT INFO

   DL-DB:    92  7.0%                         5 SEGMENT(S)
   DB-QI:    21  1.6%                       SMALLEST:     8
    QI-Q:    26  2.0%                        LARGEST:   488
     Q-S:    78  5.9%                        AVERAGE:   118
     S-Z:  1096 83.5%                    TOTAL WORDS:   592

  MAXDATA:   ??
 MAX Z-DL:  1313

 RUN TIME PARM VALUE:      0
 INFO STRING: ** NO INFO STRING **
```

As you can see, ADPAN provides much more information about the process than the STACKDUMP intrinsic. A common (and very good) practice at a number of HP sites I have visited is to assign an error number to each important step in their programs. Then if there is a problem encountered in that step the program prints out the step number and stops. This is a very simple (but effective) form of defensive programming. Examples of more sophisicated error handling include most of AGADER's functions and the MPE operating system itself. (System failures are MPE's way of preventing further damage by continuing with corrupted system tables.) This process can be enhanced by calling SNAPSHOT, passing it the error number from the program. In this way we can capture the complete environment prior to aborting the program, thus guaranteeing that we always have enough information to properly diagnose the problem.

Databases and bugs

If your application is dependent on a database, then you have a different set of problems. The cause for the wrong information on the screen may be wrong information in the database. One common mistake made by application designers is to assume that once the data has been correctly entered into the database, it will always remain semantically correct. What I mean by semantically correct is that if the weight of a pallet may be between 0 and 30,000 pounds, then a value of -200 is semantically wrong! Another problem can occur when a value from one dataset is used to chain (or point) into another set, but the second entry is missing.

Generally when a program runs into such cases (if not anticipated) the results are very unpredictable.

There are three techniques which can be used to locate bugs in our databases before they appear later as bugs in the programs. The first is to write a custom program which checks for and reports semantic errors in the database. For example, database checking programs should verify that items which are defined as dates in the programs contain VALID dates in the database. Fields which contain monetary values or other numeric quantities should be checked to make sure that their range is LEGAL and REASONABLE. Fields which are names of products, companies or individuals should be checked for garbage characters in the fields. Fields which contain phone numbers, addresses or postal mail codes should be verified. Finally if the applications chain from one dataset into another, the test program should do the same. As you might have already guessed, the error check program is a major system in itself. At our site, I run this highly tuned program once a month; its work takes more then six hours!

The second method to locate errors in the database involves active checking for semantic errors by all the application programs. The way this works is that after the user enters in the account number or part number, the program validates all the information related to that number BEFORE the information is displayed. This method assures that before the user is aware that a problem exists, the program has a chance to detect and correct it. This is the method that I use on our main application for the computer.

The final method uses a checksum or hash total for each entry in the database. The application programs, as a next-to-last step before updating the database, generate a checksum for the entity in question. This checksum value becomes an integral part of the item. When the reporting programs read the entry at a later date, they only need to recalculate the checksum value and compare to make sure that they are the same. This technique is most useful for detecting changes made in the database by unauthorized programs or QUERY. Unfortunately if the error was made before the checksum was generated the first time then it will not be detected later. An example of the use of a checksum to detect unauthorized changes is in the file labels on the HP3000.

When I first started writing programs which accessed IMAGE databases, I would generally check the status of the IMAGE intrinsics, then call DBEXPLAIN. After the first time a user wanted to know what all the clutter about dataset so-and-so was, I made an effort to remove the calls and replace them instead with a routine which opens up an error log file, calls DBCALL [9] to get a readable explanation of the problem, then calls DBERROR to obtain the intrinsic name, database name and dataset name. A final call is made to DBSTATUS [10], then all the available information is written to the error log file. For example:


```
==>ZEP  .ZESTY  ,DATA   LDEV:43 #S81  TUE, MAY  1, 1984 8:01P
Rev 2.00-84114 PROGRAM: TESTPROG P=%014.002514  Q=%015263
(PROG-ERR 2.29) Internal application or data base error
DBGET   mode 5  on SPECIFICATION of PAZAZZ opened mode  1
END OF CHAIN
DBSTATUS:  15 .....  %00452  1/ 405 %010076 %015032 5 %004601
 SET: SPECIFICATION:   ITEM-NAME: MODELCODE;
 CHAR. EQUIV OF ITEM: 0003FIDDLE
 DEC.  EQUIV OF ITEM:  12336  12339  17993  17476  19525   8224
```

Remember I said that I generally checked the status of IMAGE calls? Not long after our application was up and running a number of strange errors occured; apparently somebody had used QUERY to delete several entries that the programs always expected to be there. Since the program did not check the status of the previous IMAGE call, it did not detect the problem. The end result was a bug which migrated throughout the database and took several days to track down! Always check the status to make sure it is acceptable!

Who did it?

If we have detected an error in the database, how do we locate the cause of the problem? Hewlett-Packard has provided database users with the ability to log transactions made to an IMAGE database to either a disc file or a magnetic tape. This record can then be replayed at a later date either to recover after a system failure, or in the case of bugs, to audit the database. There are currently two programs available which can be used to audit the log, DBAUDIT and LOGLIST [11] [12] [13] [14].

III. TRAPPING THE BUG

Some times we do not have sufficient warning to set an error number and abort; for example a BOUNDS VIOLATION will generally abort the program and print out the VERY UNFRIENDLY STACK MARKER in the middle of your V/3000 form. In most cases using a screen copy routine or having the users write the information down is ineffective since the stack marker is spread throughout the form. We really want the computer to transfer to our error routines when an abnormal condition occurs. There is a facility to do this; it is called USER TRAPS.

Choosing the right trap

User traps are probably one of the least understood features of the HP3000 computer and its operating system. This is unfortunate when you consider the power they provide to detect and correct program errors. Traps are provided for the following items: [15]

Type of error encountered	Trap intrinsic
Enable hardware arithmetic traps	(ARITRAP)
Floating point divide by zero	(XARITRAP)
Integer divide by zero	(XARITRAP)
Floating point underflow	(XARITRAP)
Floating point overflow	(XARITRAP)
Integer overflow	(XARITRAP)
Extended precision overflow	(XARITRAP)
Extended precision underflow	(XARITRAP)
Decimal overflow	(XARITRAP)
Invalid ASCII digit	(XARITRAP)
Invalid decimal digit	(XARITRAP)
Invalid source word count	(XARITRAP)
Invalid decimal operand length	(XARITRAP)
Decimal divide by zero	(XARITRAP)
Bad stack marker	(XCODETRAP)
Bounds Violation	(XCODETRAP)
CST Violation	(XCODETRAP)
STT Violation	(XCODETRAP)
Illegal address	(XCODETRAP)
Non-responding module	(XCODETRAP)
Privileged Mode intruction	(XCODETRAP)
Unimplemented instruction	(XCODETRAP)
Compiler library errors (55 total)	(XLIBTRAP)
Invalid substring designator	(XLIBTRAP)
Formatter errors (FORTRAN)	(XLIBTRAP)
MPE intrinsic errors	(XSYSTRAP)

Setting the traps

Except in FORTRAN programs the user traps must be enabled by calling the respective MPE intrinsic. When enabling the trap, the plabel for the desired error-handling routine is checked to make sure that it is valid, according to the following rules:

1. If the call to enable the trap was made from a non-privileged program, group SL or public SL, the trap handling routine must also be non-privileged.

2. If the call to enable the trap was made from a privileged program, group SL or public SL, then the trap handling routine may be privileged or non-privileged, in either the program, group SL or public SL.

3. If the call to enable the trap was made from an MPE SL segment, then the error handling routine must reside in any non-MPE SL segment.

Arithmetic errors

For example, the user may enable a trap routine for arithmetic errors by calling XARITRAP as shown below.

```
              IV   IV      I        I
XARITRAP(mask,plabel,oldmask,oldplabel)

mask     -  Bit mask indicating which types of
            arithmetic errors are to be trapped
            (refer to the HP intrinsic manual [16]).
            mask = 0 disables the traps.

plabel   -  External type label of the application's trap
            procedure.  plabel = 0 disables the traps.

oldmask  -  The previous bit mask for the arithmetic
            traps.

oldplabel-  The previous external type label of the
            application's error procedure (0 if not
            previously enabled).
```

Example of an SPL routine to enable all arithmetic traps:

```
PROCEDURE    ARMTRAPS;
BEGIN
  INTRINSIC XARITRAP;
  INTEGER OLDMASK,OLDPLABEL;
  XARITRAP(%37777,@ARITH'ERROR,OLDMASK,OLDPLABEL);
END;
```

EXAMPLE of an SPL routine to handle traps caused by arithmetic errors:

```
PROCEDURE ARITH'ERROR;
BEGIN
  ARRAY BUFF(0:40);
  BYTE ARRAY STRING(*)=BUFF;
  INTRINSIC PRINT,QUIT;
  SNAPSHOT(0);
 MOVE STRING := ("Arithmetic error! SNAPSHOT was taken!");
  PRINT (BUFF,-38,0);
  QUIT(0);
<< WISHFUL THINKING. WE CAN NEVER RETURN THROUGH THE END! >>
  END;
```

Users of FORTRAN have the ability to enable traps selectively by using the "ON error condition CALL subroutine" statement [17]. unique procedures. The trap mechanism in FORTRAN very flexible; it does not come free, though. In order to separate integer overflows from divide by zero, the FORTRAN run-time library plays a few games. Using the ON statement results in a named COMMON called TRAPCOM' being established on your behalf. When an integer overflow occurs, the computer transfers control not directly to your routine, but to a library routine. This library routine then determines the type of hardware trap that was invoked and accesses TRAPCOM' to obtain the plabel for your routine. Once the library has a valid plabel, it transfers control to your error handling routine by placing the plabel on the top of the stack and performing a PCAL 0.

A user may enable traps for integer overflows and integer divide by zero by using the following FORTRAN stetements:

```
ON INTEGER OVERFLOW CALL OVERFLOW ROUTINE
ON INTEGER DIV 0    CALL DIVIDEO  ROUTINE
```

HP sites that are heavy users of COBOL have a completely different story on their hands. COBOL deliberately calls a routine called C'TRAP to enable SELECTED traps. This was done because when a field is MOVEd in a COBOL program, the COBOL library handles any type conversion that is necessary. The traps that C'TRAP enables are:

- Integer divide by 0
- Integer overflow
- Decimal overflow
- Decimal divide by 0
- Invalid Decimal digit
- Invalid ASCII digit

One annoying feature of COBOL programs is that when an invalid ASCII character is detected while moving a character field to a numeric field, the COBOL run-time library attempts to "fixup" the mistake (this was done to be compatible with users who read data generated on punched cards, using overpunching). You may change the traps that are enabled so the program will not attempt a fixup but will instead abort, by using the following SPL routine:

```
PROCEDURE  ABORTBADASCII;
   BEGIN
         INTRINSIC XARITRAP;
         INTEGER NEWPLABEL,OLDMASK,OLDPLABEL;
         XARITRAP(0,0,OLDMASK,OLDPLABEL);
         NEWPLABEL := OLDPLABEL;
         XARITRAP(%22422,NEWPLABEL,OLDMASK,OLDPLABEL);
   END;
END.
```

Bounds violations

Bounds violations, bad stack markers and invalid instructions may be trapped by the UNDOCUMENTED user-callable procedure XCODETRAP. This routine, which has been around for a number of years, is used by DEBUG and, believe it or not, COBOL! The calling sequence for this intrinsic is:


```
          I          IV
XCODETRAP(newplabel,oldplabel)
```

newplabel - External type plabel of the application's trap procedure. plabel = 0 will disable the trap.

oldplabel - Previous external type plabel of the application's trap procedure. If the trap was disabled, 0 is returned.

NOTE: XCODETRAP is not in the intrinsic SPLINTR file, therefore do not try to declare it as an intrinsic or your programs will not compile.

FORTRAN users may enable this routine by using the following code:

```
EXTERNAL BOUNDS ROUTINE
CALL XCODETRAP(BOUNDS ROUTINE,IOLDPLABEL)
```

Currently users of other languages such as COBOL must use an SPL routine to enable the trap, such as the following:

```
<< Since we can not declare XCODETRAP as an intrinsic
   we must declare it here so the SPL compiler knows
   that it exists.                                   >>
PROCEDURE XCODETRAP(NEWLABEL,OLDLABEL);
VALUE NEWLABEL;
INTEGER NEWLABEL,OLDLABEL;
OPTION EXTERNAL;

PROCEDURE    ARMTRAP;
BEGIN
  INTEGER OLDMASK,OLDPLABEL;
  XCODETRAP(@BOUNDSVIOLATION,OLDPLABEL);
END;
```

Example of the bounds violation trap routine:

```
PROCEDURE BOUNDSVIOLATION;
BEGIN
  ARRAY BUFF(0:40);
  BYTE ARRAY STRING(*)=BUFF;
  INTRINSIC PRINT,QUIT;
  SNAPSHOT(0);
MOVE STRING := ("BOUNDS VIOLATION!  SNAPSHOT was taken!");
  PRINT (BUFF,-40,0);
  QUIT(0);
<< WISHFUL THINKING. WE CAN NEVER RETURN THROUGH THE END! >>
  END;
```

Run-time library errors

With the exception of SPL, all of the languages on the HP3000 use run-time libraries. If an error is detected while in the library the user has the option to request transfer to a trap handling routine, rather than to abort the program. The calling sequence for this routine is:

```
             IV        I
XLIBTRAP(newplabel,oldplabel)

newplabel  -  External type plabel of the application's trap
              procedure.  plabel = 0 will disable
              the trap.

oldplabel  -  Previous external type plabel
              that was in effect.  If the trap was
              disabled, 0 is returned.
```

FORTRAN users may enable this trap by using the statements:

```
ON INTERNAL ERROR CALL LIBRARY ROUTINE
ON FORMAT ERROR   CALL LIBRARY ROUTINE
```

Currently users of other languages such as COBOL must use an SPL routine, such as the following, to enable the trap.

```
PROCEDURE    ARMLIBTRAP;
BEGIN
  INTRINSIC XLIBTRAP;
  INTEGER OLDMASK,OLDPLABEL;
  XLIBTRAP(@LIBRARYROUTINE,OLDPLABEL);
END;
```

An example of a library trap routine:

```
PROCEDURE LIBRARYROUTINE;
BEGIN
  ARRAY BUFF(0:40);
  BYTE ARRAY STRING(*)=BUFF;
  INTRINSIC PRINT,QUIT;
  SNAPSHOT(0);
  MOVE STRING := ("LIBRARY error! SNAPSHOT was taken!");
  PRINT (BUFF,-36,0);
  QUIT(0);
<< WISHFUL THINKING. WE CAN NEVER RETURN THROUGH THE END! >>
  END;
```

MPE intrinsic errors

Almost any abnormal condition which occurs within the MPE intrinsics can be detected by using system traps (XSYSTRAP). The calling sequence for this intrinsic is:


```
             IV        I
XSYSTRAP(newplabel,oldplabel)

newplabel  -  External type plabel of the application's trap
              procedure.  plabel = 0 will disable
              the trap.

oldplabel  -  Previous external type plabel
              that was in effect.  If the trap was
              disabled, 0 is returned.
```

FORTRAN users may enable this trap by using the statement:

```
ON SYSTEM ERROR CALL SYSTEM ROUTINE
```

Currently users of other languages such as COBOL must use an SPL routine to enable the trap. An example of an SPL enabling routine is:

```
PROCEDURE    ARMSYSTRAP;
BEGIN
  INTRINSIC XSYSTRAP;
  INTEGER OLDMASK,OLDPLABEL;
  XSYSTRAP(@SYSTEMROUTINE,OLDPLABEL);
END;
```

An example of system trap routine:

```
PROCEDURE SYSTEMROUTINE;
BEGIN
  ARRAY BUFF(0:40);
  BYTE ARRAY STRING(*)=BUFF;
  INTRINSIC PRINT,QUIT;
  SNAPSHOT(0);
  MOVE STRING := ("SYSTEM error! SNAPSHOT was taken!");
  PRINT (BUFF,-36,0);
  QUIT(0);
<< WISHFUL THINKING. WE CAN NEVER RETURN THROUGH THE END! >>
  END;
```

A bug! Catch it!

When an error occurs, the hardware transfers control to the correct trap, if it was enabled, otherwise the computer enters standard H-P abort routines. The user-written error handling routine may be in the program, the group SL, or the public SL. User traps are usable from all languages currently available for the HP3000; however there are some special considerations for COBOL and RPG programs [18].

The error handling routines can be written so that they either attempt to correct the problem (COBOL does this with Invalid ASCII digits) or abort the program. Regardless of which is done, be sure that as much information as possible about the cause of the error is written to a separate error log, so that the bug can be easily corrected.

IV. KILLING THE BUG

Once the process information has been saved or printed, we can abort the program (if desired) in a manner I call STRUCTURED PROGRAM FAILURES. This means that we abort the program in a clearly defined and orderly manner. For instance our abort routine switches the terminal back to character mode, prints a standard abort message on the user's terminal, displays the procedure name in which the bug was detected, then prints an abort message on the operator console (so special program recovery steps can be taken if necessary). A message is sent to any user who is logged on to the programming account, the JCW CIERROR is set to 976 (program abort), JCW is set to FATAL, and finally the program calls QUIT to abort the whole process tree (if any).

Here is an example of a FORTRAN abort procedure, which illustrates the above:

```
$CONTROL MAP,LOCATION,LABEL,STAT
      C
      C  F SUDDEN DEATH:  The purpose of this routine is to provide
      C                  a means of a structured program failure
      C                  similiar to HP's SUDDEN DEATH intrinsic.
      C
      C                  This routine DOES NOT halt the machine or
      C                  cause SF's, it does abort the process
      C                  tree!
      C
      C                  There are two passed variables for this
      C                  routine, IERR and PROCEDURE.
      C                  The  IERR contains the programmer-
      C                  assigned step number, which is included
      C                  in the SNAPSHOT and printed out when the
      C                  program aborts.
      C
      C                  The value of PROCEDURE is a character
      C                  string which is printed on the user's
      C                  screen, and the operator console.
      C                  A corresponding JCW name is checked and
      C                 decremented.  If the resulting JCW is
      C                 greater then zero, this routine will
      C                 return to the calling process.
      C
      C                  In addition, this procedure checks for a
      C                  JCW called DEBUG; if it exists, and > 0,
      C                 the the procedure calls, the H-P
      C                 program debugger.
      C                    written by Dennis Heidner
      C
            SUBROUTINE  F SUDDEN DEATH(IERR,PROCEDURE)
            CHARACTER PROCEDURE*16,COMIMAGE*80,JCWNAME*16
            INTEGER IERR,JCWVALUE
            LOGICAL LTEXT(40),MUST STOP,LJCWVALUE
            EQUIVALENCE (LTEXT(1),COMIMAGE),(JCWVALUE,LJCWVALUE)
            SYSTEM INTRINSIC COMMAND,PRINT,PUTJCW,FINDJCW,DEBUG
            SYSTEM INTRINSIC STACKDUMP,QUITPROG
      C
      C     Take a picture of the data stack....
      C
            CALL SNAPSHOT(IERR)
      C
            DO 100 LENGTH OF STRING=1,16
            IF(PROCEDURE[LENGTH OF STRING:1].EQ.";") GOTO 200
            IF(PROCEDURE[LENGTH OF STRING:1].EQ." ") GOTO 200
      100   CONTINUE
            LENGTH OF STRING = 16
      C
      C     CHECK THE JCW, WHICH CORRESPONDS TO THE PROCEDURE NAME.
      C
```

```
200   IF(LENGTH OF STRING .GT. 1) GOTO 300
      PROCEDURE = "NULL"
      LENGTH OF STRING = 5
C
300   JCWNAME = PROCEDURE[1:LENGTH OF STRING - 1]
C
C     DOES THE JCW EXIST?
C
      MUST STOP = .TRUE.
      CALL FINDJCW( JCWNAME, LJCWVALUE, ISTATUS)
      IF(ISTATUS.NE.0) GOTO 500
C
C     DECREMENT THE JCW VALUE
C
      JCW VALUE = JCW VALUE - 1
      CALL PUTJCW ( JCWNAME, LJCWVALUE, ISTATUS)
      IF( JCW VALUE .GT. 0) MUST STOP = .FALSE.
C
C     DISPLAY THE ABORT MESSAGE
C
500   COMIMAGE="Program error in procedure: "
      COMIMAGE[30:LENGTH OF STRING] =
     & PROCEDURE[1:LENGTH OF STRING]
      CALL PRINT(LTEXT,-50,%0)
C
C     NOTIFY THE SYSTEM OPERATOR....
C
      COMIMAGE="TELLOP  Program aborting in procedure: "
      COMIMAGE[40:LENGTH OF STRING] =
     & PROCEDURE[1:LENGTH OF STRING]
      COMIMAGE[40+LENGTH OF STRING+1:1]=%15C
      CALL COMMAND( COMIMAGE, ICOMERR,IPARM)
C
C     DO WE DROP INTO DEBUG FIRST?
C
      JCWNAME="DEBUG"
      CALL FINDJCW(JCWNAME, LJCWVALUE, ISTATUS)
      IF(( ISTATUS.NE.0) .OR. (JCWVALUE .LE. 0)) GOTO 1000
      CALL DEBUG
C
C     SET THE JCW'S   CIERROR TO 976   AND   JCW  TO FATAL
C
1000  JCWNAME="CIERROR"
      CALL PUTJCW(JCWNAME,%1720L,ISTATUS)
C
      JCWNAME="JCW"
      CALL PUTJCW(JCWNAME,%100001L,ISTATUS)
C
C     SAY YOUR PRAYERS.....
C
      IF ( MUST STOP )  CALL QUITPROG(IERR)
      RETURN
      END
```


After the bug has been detected or reported, make sure that you use sound software maintenance practices and keep a log of the bugs, the work-arounds, and the fixes. This will enable you to provide better estimates of your future software mainenance costs, estimate number of bugs remaining, provide an indispensible diary for others who might later maintain the software and perhaps most important, provide an experience base so that future software products can be clean and free of similiar bugs.

V. EPITAPH

Although it is impossible to eliminate all bugs from software, it is possible to design the software so that it is easy to maintain and self-diagnosing. This paper has covered several techniques, which if incorporated will help reduce the cost of software maintenance.

VI. REFERENCES

[1] Martin, James and McClure, Carma, "Software Maintenance: The Problem and Its Solutions" (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983). p. 4.

[2] Martin, James and McClure, Carma, "Software Maintenance: The Problem and Its Solutions" (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983).

[3] Glass, Robert L. and Noiseux, Ronald A. "Software Maintenace Guidebook" (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979).

[4] Myers, Glenford J., "Software Reliability: Principles and Practices" (New York, NY: John Wiley & Sons, Inc., 197?)

[5] Coats, Dan and McCaffrey,Michael, "Customer Satisfaction through Quality Software", Anaheim PROCEEDINGS, HPIUG 1984, p. 7.

[6] VTEST available from: TYMLABS
211 East 7th Street
Austin, Texas 78701
(512) 478-0611

[7] Contributed Library Tape, Available from:
HP3000 International Users Group
(INTEREX)
2570 El Camino Real West
4th Floor
Mountain View, CA 94040

[8] ADPAN, 1984 Anaheim Swap Tape, Available from INTEREX.

[9] Green, Robert M. "The IMAGE/3000 Handbook", (Seattle, WA: WORDWARE, 1984). p. 283.

[10] ibid, p. 283

[11] Green, Robert M., "Auditing with IMAGE Transaction Logging", San Antonio PROCEEDINGS, HPIUG, 1982

[12] Heidner, Dennis L., "Transaction Logging and Its Uses" San Antonio PROCEEDINGS, HPIUG, 1982

[13] Green, Robert M. and Heidner, Dennis L., "Transaction Logging Tips", Montreal PROCEEDINGS, HPIUG 1983

[14] DBAUDIT, Available from: Robelle Consulting Ltd.
8648 Armstrong Road, RR#6
Langley, B.C. V3A 4P9
Canada (604)-856-3838

LOGLIST, Available from: INTEREX (HPIUG)

[15] Hewlett=Packard, "Intrinsics Reference Manual", Part number 30000-90010

[16] ibid. p. 2-199.

[17] Hewlett-Packard, "FORTRAN Reference Manual", Part number: 30000-90040, p. 4-21 thru p. 4-26.

[18] Hewlett-Packard, "COBOL II Reference Manual", Part number: 32233-90001