User Guide

for

VISUAL TRACE
And
VISUAL PERFORMANCE ANALYSIS

Version 5.0

To receive further information about **Visual Data** products please contact:

Visual Data ApS
Attn: Lars Kruse Bøggild
Hvidtjørnen 36
DK-2791 Dragør / CPH
Denmark

Tel: +45 32 94 31 94

e-mail: support@visualdata.info
Internet: www.visualdata.info

Visual Data ApS
Attn: Gary J. Sowada
12181 Highway 27
Little Falls, MN USA 56345

Tel: +1 320.632.6200
Fax: +1 320.632.6240

e-mail: Sales@visualdata.info
Internet: www.visualdata.info

User Guide for Visual Trace and Visual Performance Analysis

Enhanced USAS Tracing

One of the primary ways for tracing USAS programs involves the use of the USAS TST: trace facilities. Although effective in tracing USAS programs, reading these low-level trace files with all references to calls shown as an X11 address and a BDI value can be difficult to interpret. **Visual Trace** will analyze these standard USAS trace files and display their contents by replacing all low-level address information with symbolic program names and source line numbers. To request this translation, enter:

TRCE: ***[qualifier*]filename. [trace-file-line-number][/format]***

If omitted, the ***qualifier*** will be the default qualifier for the TIP system, typically **TIP$**.

If the ***trace-file-line-number*** is not included in the input, the display will begin with the first line of the USAS trace file. Otherwise, the display will begin at the specified line number. Specifying a line number is especially useful when you know the lines you wish to view or when the output exceeds the capacity of the paging file.

All the output generated by **Visual Trace** has the ***trace-file-line-number*** in the left-most column enabling you to redisplay the trace file starting at a specific point.

The default TRACE output format

In the default trace format, **Visual Trace** analyzes and displays the USAS trace file by converting call lines containing low level address information into lines containing the symbolic program names and FTN line numbers of the call. Further, it will filter away all calls deemed to be of low interest like XBCA and XBCF calls origination from FTN stack allocation/deallocation. Internal calls made by ACBs are also removed to give a more concise overview of the transaction flow.

When **SNAPS** are produced that result in trace lines exceeding 80 characters, the lines are broken in two and the snapped data is shown as 2 lines of 4 octal words rather then 1 line by 8 words. This ensures the entire snap can be seen at any time and no left/right adjustment is needed.

Additional TRACE output formats

The ***format*** keywords alter the default output format in various ways making its analysis more effective. These keywords, which can be supplied anywhere in the input string, are **RAW**, **ALL** or **CALLS**.

By specifying the **RAW** output format, **Visual Trace** displays the USAS trace file in its raw format – the data as produced by USAS without any alteration or suppression of lines.

The **ALL** output format ensures that all lines in the trace file are shown – including lines otherwise filtered away in the default display.

The **CALLS** output format forces **Visual Trace** to suppress all lines from the trace file that are *not* a call line. This allows for a quick overview of the transaction flow, which is especially useful with trace files that contain record I/O traces and other large data **SNAPS**. You can use this format to follow the general program flow, and then more carefully analyze a portion of the trace by redisplaying it from a specified ***trace-file-line-number*** allowing for all lines to be shown.

You can combine the **ALL** and **CALLS** keywords to receive *all* call lines - even stack allocation/deallocation calls and internal ACB calls – but not receive lines containing **SNAPS** and record I/O traces.

Trace file usage

By default the **TRCE:** transaction will show you all traces captured to your trace file with this often including traces for more transactions. This can lead to a very long tracing process as every absolute present in the trace will need to be found and read before each line in the raw trace file can be converted from an absolute address line into a symbolic trace line. To avoid these long traces it is recommended that you qualify the data you are interested in and thereby reduces the trace display.

Qualifying your display requests will make it easier to find and view the area of interest and it will dramatically decrease the time spent producing the trace displays.

You can qualify your input by filtering on either a ***program*** name, a ***function-code*** or by specifying the **HVTIP** keyword when requesting an overview display.

Trace file overview

As trace files can normally be very long and contain traces from many different transactions it is usually a good idea to get an overview of what is in the trace file before requesting trace displays. The **HVTIP** keyword will limit the trace display to all HVTIP calls/returns and thus give you a quick and easy overview of what functions have been traced and what programs these transactions passed though during execution.

TRCE: *[qualifier*]filename.* HVTIP

Once the HVTIP display has been output, it is easy to request a new trace display with the appropriate filtering options set to enable a quick and accurate trace display showing all the data of interest.

Program filtering

Visual Trace allows you to filter your trace file symbolically by entering the ***program*** filter, thus suppressing all trace lines that do not belong to the specified program. To do this, enter:

TRCE: *[qualifier*]filename. program[/format]*

HVTIP library calls are still shown with the above request in order to give an overview of when the specified program was called. This program filter allows you to quickly view trace lines from the specific absolute. Especially on large trace files it is recommended that you use program filtering whenever possible as it will speed the trace process dramatically as only one absolute will need to be found and processed to produce the entire trace display.

Function code filtering

In the same way, **Visual Trace** allows you to filter your trace file on a ***function-code***, thus suppressing all trace lines that do not belong to the specified function. To do this, enter:

TRCE: *[qualifier*]filename. function-code[/*HVTIP*] [/format]*
TRCE: *[qualifier*]filename. function-code[/program] [/format]*

If your trace file contains more traces of the same specified ***function-code*** they will all be processed and be included in the trace display. Alternatively, you can combine the ***function-code*** filter with the **HVTIP** or ***program*** filter to further reduce the trace display.

Absolutes or Zooms read during Trace Analysis

In order for **Visual Trace** to perform the correct analysis on a trace file, it is important that **Visual Trace** is able to find all the actual absolutes/zooms that were used to create the trace file. Without the correct absolutes/zooms, the analysis can easily be flawed resulting in misleading and incorrect conversion of the low-level address information into symbolic names and source line numbers.

Therefore, it is always a good idea for you to verify that the absolutes/zooms used to analyze the trace file are the correct ones. At the end of each trace display, **Visual Trace** automatically informs you of which absolutes/zooms were used in the analysis by displaying the files they were retrieved from and the date/timestamp they were created.

The system and personal search paths

Visual Trace uses the same search path for absolutes as **The Visual Dump Analyzer**. This search path is made up of a *personal* search path and a *system* search path, with the personal search path being traversed before the system search path when looking for an element.

There is a unique *system* search path and *personal* search path defined for each computer system. On production systems, a personal search path is generally not required as all elements are stored within system files. On development systems, however, you will need to add your own files to your *personal* search path on each computer system so the correct absolute can be found.

To update your personal search path and display the system search path on a computer system, just enter:

RDMP:PATH

Your *personal* search path is saved in the file **SED$*nnnnn** where nnnnn is your VDU number (i.e., USAS PID number). Therefore, you will need to update your search path on all the PIDs you are using.

Combining TRACE with the Enhance Paging function (RPAG)

By combining the TRCE: function for analyzing and displaying the contents of a USAS trace file and the RPAG: function for traversing within the paging output and searching for particular text, you can efficiently view the trace file information without having to open a demand session.

Use the RDMP:SETUP to turn the RPAG Enhanced Paging ON or OFF.

Even when RPAG is turned OFF, you can activate it for the current paging file by using RPAG:TRCE and then use it on this specific output.

Visual Breakpoint Traps

A trace is commonly used to determine the path taken by a transaction during its execution. Normal Usas tracing, however, only provides a trace that includes calls to ACB and FLSS routines. The volume of code that exists between ACB calls is not traced and therefore its form of execution is unknown. This problem is solved by **Visual Breakpoint Traps** (**TRAPS**)**.** Because **TRAPS** doesn't rely on any special events such as ACB calls for intercepting the execution, any piece of code can be analyzed in extensive detail.

Besides intercepting when a specific piece of code is executed, **Visual Breakpoint Traps** also allows you to get control when a specific variable or address is referenced or updated. This is a very useful way of finding code that alters or destroys specific data.

Among its many features, **TRAPS** can inform you whether you have executed a given line number or subroutine, it can snap data in various ways, it can force an abort at the "right" spot, and it can cause an abort when a particular variable acquires a specified value.

In order to use **Visual Breakpoint Traps**, just enter the following and a mask will be displayed for you to enter your **TRAPS** and **SNAPS** request:

TST:TRAP

Use the **TRAP** portion of this mask to define where you want to intercept your program and cause a Breakpoint to occur. Setting **Breakpoint Traps** is very simple process even when the trap itself is very complex in nature.

TRAPS are divided into two different types: I-Bank **TRAPS** and D-Bank **TRAPS**. I-Bank **TRAPS** are traps that occur whenever a specific I-Bank instruction is executed and D-Bank **TRAPS** are traps that that occur when a given D-Bank address is referenced or updated.

The **TRAP** and **SNAP** portions of the input mask can both be defined or either part can individually be specified within a request. When **SNAPS** are used alone, it will produce snaps on all calls to and returns from ACB and FLSS routines. When the **SNAPS** are used in combination with **TRAPS**, it will also produce snaps every time a specified **TRAP** occurs.

Setting line number Traps

In order to set a **TRAP** on a specific line number, you only need to specify the program name and line number. This is the simplest form of a **TRAP** that can be set as it identifies a specific location (instruction) in the program and ensures the **TRAP** occurs every time this instruction is executed. The location of the **TRAP** will remain the same during the execution of the program, which may result in the trap occurring many times during the execution of the transaction.

Setting a **TRAP** on a line number in an ACB or FLSS routine is done in the same way as normal HVTIP programs. You only need to enter the ACB or FLSS name (e.g., SYSWRT1 or SYSFLS3) along with the name of the relocatable routine and FTN line number. See below example which sets a **TRAP** on line 561 in the relocatable ACBROU within the SYSWRT1 absolute.

```
TST:TRAP/

      VISUAL BREAKPOINT TRAPS AND SNAPS

TRAP PROGRAM NAME                      ]SYSWRT1 [
 IN RELOCATABLE NAME                   ]ACBROU  [
 AT LINE NO OR ADR OR INTERNAL SUBR.   ]     561[
 OR WHEN IT IS CALLED AND WHEN IT RETURNS    ]N[
 OR WHEN IT CALLS OTHER HVTIP PROGRAMS       ]N[
 OR WHEN BELOW SNAP IS REFERENCED            ]N[
 OR WHEN BELOW SNAP IS UPDATED               ]N[

SNAP VARIABLE/DBA/PDB/ADDRESS/VA  ]            [
 IN INTERNAL SUBR./DBA OFFSET     ]            [
 NUMBER OF WORDS - TRAP ON 1ST WORD ONLY ]     [
 SNAP ON BREAK POINT HITS ONLY               ]N[

WHEN THE TRAP HITS
 ABORT ON HIT NUMBER                    ]     [
 ABORT WHEN SNAP VALUE EQUALS     ]            [
                                             ] [
```

When a line number **TRAP** is set, it is normally placed on the first assembler instruction generated by the FTN line number. If the FTN line does not actually cause any assembler instructions to be generated (e.g., Comment lines, ENDIF and CONTINUE statements), the **TRAP** will be set on the first executable instruction following the FTN line number.

Note that if the Z-option (optimization) is used with the FTN compiler, it can merge assembler code for multiple FTN lines in order to generate more efficient code. This can cause a line number **TRAP** to be set on instructions away from where it would appear to be set if compared to the source code. This, however, is usually not a problem, but if you want to be in complete control of the exact location of your **TRAP**, use **RABS:** to see the assembler code and set the **TRAP** by using an absolute address rather than a line number. (*See Setting Address Traps*)

When setting **TRAPS** on IF-Statements, it can sometimes be difficult to predict if the instruction is actually executing and therefore should often be avoided.

Setting HVTIP program Traps

You can specify a **TRAP** for an HVTIP program so the hit occurs every time the program is called or returned from. See below example, which sets a **TRAP** on a call to and return from MYPROG.

```
TST:TRAP/

      VISUAL BREAKPOINT TRAPS AND SNAPS

TRAP PROGRAM NAME                      ]MYPROG  [
 IN RELOCATABLE NAME                   ]        [
 AT LINE NO OR ADR OR INTERNAL SUBR.   ] [
 OR WHEN IT IS CALLED AND WHEN IT RETURNS    ]Y[
 OR WHEN IT CALLS OTHER HVTIP PROGRAMS       ]N[
 OR WHEN BELOW SNAP IS REFERENCED            ]N[
 OR WHEN BELOW SNAP IS UPDATED               ]N[

SNAP VARIABLE/DBA/PDB/ADDRESS/VA  ]            [
 IN INTERNAL SUBR./DBA OFFSET     ]            [
 NUMBER OF WORDS - TRAP ON 1ST WORD ONLY ]     [
 SNAP ON BREAK POINT HITS ONLY               ]N[

WHEN THE TRAP HITS
 ABORT ON HIT NUMBER                    ]     [
 ABORT WHEN SNAP VALUE EQUALS     ]            [
                                           ] [
```

This **TRAP** will occur when MYPROG is called and when MYPROG returns. Because this **TRAP** occurs in two unique locations, this type of **TRAP** is considered complex. (*See Defining Complex Traps*)

Another type of HVTIP **TRAP** is **WHEN IT CALLS OTHER HVTIP PROGRAMS**. By selecting this type of **TRAP**, you will get a hit every time MYPROG calls forward to another HVTIP program and you will also get a hit every time the called program returns to MYPROG.

These HVTIP **TRAPS** are very useful for determining which programs have updated certain data by causing hits on the boundaries between HVTIP programs where you can **SNAP** data or cause aborts at these locations.

Setting subroutine Traps

Setting subroutine **TRAPS** is similar to setting HVTIP **TRAPS** except that you request the **TRAP** to occur on a specified internal FTN/UFTN subroutine instead of the main HVTIP program. The example below sets a **TRAP** on the calls and returns from the internal subroutine SENDIT located within the PXCRTO absolute.

```
TST:TRAP/

      VISUAL BREAKPOINT TRAPS AND SNAPS

TRAP PROGRAM NAME                      ]PXCRTO  [
 IN RELOCATABLE NAME                   ]        [
 AT LINE NO OR ADR OR INTERNAL SUBR.   ]SENDIT  [
 OR WHEN IT IS CALLED AND WHEN IT RETURNS    ]Y[
 OR WHEN IT CALLS OTHER HVTIP PROGRAMS       ]N[
 OR WHEN BELOW SNAP IS REFERENCED            ]N[
 OR WHEN BELOW SNAP IS UPDATED               ]N[

SNAP VARIABLE/DBA/PDB/ADDRESS/VA  ]            [
 IN INTERNAL SUBR./DBA OFFSET     ]            [
 NUMBER OF WORDS - TRAP ON 1ST WORD ONLY ]     [
 SNAP ON BREAK POINT HITS ONLY               ]N[

WHEN THE TRAP HITS
 ABORT ON HIT NUMBER                    ]     [
 ABORT WHEN SNAP VALUE EQUALS     ]            [
                                           ] [
```

Subroutines **TRAPS** are considered to be complex as they will occur on both calls to and returns from the internal subroutine. (*See Defining Complex Traps*)

Setting Address Traps

All types of **TRAPS** are eventually set using an absolute address. For symbolic **TRAPS**, **Visual Trace** calculates the address to set the **TRAP** on. You can, however, set the **TRAPS** on a specific address by entering a program name in the **TRAP** portion of the mask and the absolute address in **AT LINE NO OR ADR OR INTERNAL SUBR**. By entering the absolute address with a leading zero indicating octal, you have informed **Visual Trace** that the value is an address instead of a line number.

You can use **RABS:** to find the address you want to set the **TRAP**. If the address is in an ACB or FLSS, you don't need to be concerned about the BDI as **Visual Trace** is aware of these values.

Setting Variable or D-bank Traps

You can set a Variable or D-Bank **TRAP** by defining information in the **SNAP** portion of the mask and then selecting either:

WHEN BELOW SNAP IS REFERENCED

or

WHEN BELOW SNAP IS UPDATED.

This will cause a hit every time the **SNAP** data is referenced or updated. Combining D-Bank **TRAPS** along with **SNAPS** results in a variable being snapped whenever the **TRAP** occurs.

The **SNAP** data may be defined to be several words, an entire table, or a DBA; but the **TRAP** will only be effective on the first word of the snapped data.

If you don't want the data to be snapped or displayed on every call to an ACB or FLSS routine, you can select the **SNAP ON BREAK POINT HITS ONLY** and the data will only be displayed when a **TRAP** occurs. (*See Setting Snaps*)

Note that if the Z-option (optimization) is used on the FTN compiler, it may generate more efficient assembler code by using a register instead of your defined variable. In this case, if you set the **TRAP** on the variable, the trap will never occur as the variable is not actually being used. By using **RABS:**, you can analyze the generated assembler code about the line numbers in question and verify if compiler has actually used the variable.

Setting Snaps

The **SNAP** portion of the mask is used to define the data you want snapped (displayed) when a **TRAP** occurs. Besides snapping the data when the **TRAP** occurs, the snapped data will also be displayed BEFORE and AFTER every call to an ACB or FLSS routine.

SNAPS that are displayed before and after an ACB or FLSS routine provides additional information for analysing when a particular piece of data has changed. Be aware that ACB or FLSS calls can be nested in that one ACB routine may call other ACB routines. When this happens, you will see both the BEFORE and AFTER **SNAP** on the initial ACB routine call, but for technical reasons only BEFORE **SNAPS** will be produced for the nested ACB calls. Snaps for XFSxx and XDFxx I/O routines will always be shown both before and after the calls.

You can set **SNAPS** on Variables, DBAs, PDBs, Basic Mode Addresses and Extended Mode VAs. Due to the nature of defines, however, you cannot set a **SNAP** directly on a define using its symbolic name. Instead, you must calculate its address location and then set the **SNAP** on the DBA.

Setting Variable Snaps

The below example sets a **SNAP** on the local variable MYVAR in the internal subroutine MYSUB, which is located in the relocatable MYREL within the MYPROG program.

```
TST:TRAP/

      VISUAL BREAKPOINT TRAPS AND SNAPS

TRAP PROGRAM NAME                      ]MYPROG  [
 IN RELOCATABLE NAME                   ]MYREL   [
 AT LINE NO OR ADR OR INTERNAL SUBR.   ]        [
 OR WHEN IT IS CALLED AND WHEN IT RETURNS    ]N[
 OR WHEN IT CALLS OTHER HVTIP PROGRAMS       ]N[
 OR WHEN BELOW SNAP IS REFERENCED            ]N[
 OR WHEN BELOW SNAP IS UPDATED               ]N[

SNAP VARIABLE/DBA/PDB/ADDRESS/VA  ]MYVAR       [
 IN INTERNAL SUBR./DBA OFFSET     ]MYSUB       [
 NUMBER OF WORDS - TRAP ON 1ST WORD ONLY ]    1[
 SNAP ON BREAK POINT HITS ONLY               ]N[

WHEN THE TRAP HITS
 ABORT ON HIT NUMBER                    ]     [
 ABORT WHEN SNAP VALUE EQUALS     ]           [
                                          ] [
```

Note that even though the **TRAP** area of the mask is used to identify the program and relocatable where the **SNAP** variable is found, this will not cause a trap to be set as a location has not been defined within the **TRAP** portion of the mask.

Variables can be snapped by entering their symbolic name such as LINE, WORK or WORK(5). Also, if the variable LINE is an 80 character variable, you can snap all of it by setting the number of words to 20.

If the size of the **SNAP** data is greater than one, then more **SNAP** lines are produced. The first line will give the variable name, its absolute address, and the value of the first word. The following lines will show four octal words each until all the requested **SNAP** data has been displayed.

Again note that if the Z-option (optimization) is used on the FTN compiler, more efficient code may have been generated resulting in a register being used instead of your actual variable. This can easily cause confusion, so when in doubt use **RABS:** to see the generated assembler code about the line numbers in question and verify if the compiler has actually used the variable.

Setting DBA or PDB Snaps

You can set a **SNAP** on a DBA or PDB by entering its symbolic name. If the symbolic name entered is used to define both a DBA and a PDB, you must qualify the name by entering, for example, FM$DBA or FM$PDB.

The example below sets a **SNAP** on the 25th word in the FM DBA and snaps a total of 100 words. When setting **SNAPS** on DBAs and PDBs, you must enter an offset into the DBA or PDB. The offset is 1-relative so if you want to set a **SNAP** on the first word, you need to enter 1.

```
TST:TRAP/

      VISUAL BREAKPOINT TRAPS AND SNAPS

TRAP PROGRAM NAME                      ]        [
 IN RELOCATABLE NAME                   ]        [
 AT LINE NO OR ADR OR INTERNAL SUBR.   ]        [
 OR WHEN IT IS CALLED AND WHEN IT RETURNS    ]N[
 OR WHEN IT CALLS OTHER HVTIP PROGRAMS       ]N[
 OR WHEN BELOW SNAP IS REFERENCED            ]N[
 OR WHEN BELOW SNAP IS UPDATED               ]N[

SNAP VARIABLE/DBA/PDB/ADDRESS/VA  ]FM          [
 IN INTERNAL SUBR./DBA OFFSET     ]         25[
 NUMBER OF WORDS - TRAP ON 1ST WORD ONLY ]  100[
 SNAP ON BREAK POINT HITS ONLY               ]N[

WHEN THE TRAP HITS
 ABORT ON HIT NUMBER                   ]     [
 ABORT WHEN SNAP VALUE EQUALS     ]           [
                                           ] [
```

Using the **TRAP** area of the mask is optional. If it is not used, the **SNAPS** will start as soon as the DBA or PDB gets allocated and will stop if the DBA or PDB gets released. If the DBA or PDB is later re-allocated, the snapping will again start even if the re-allocation caused the DBA or PDB to be allocated at a new location.

Snapping DBAs or PDBs can result in very large trace files. You can limit the number of **SNAPS** by entering a program name in the **TRAP** portion of the mask. If this is done, the DBA or PDB will only be snapped while in the specified program. You can further limit the **SNAPS** by specifying a program name, relocatable or internal subroutine.

Setting Address or VA Snaps

To set a **SNAP** for a Basic Mode Address, just enter the absolute address in the **SNAP** part of the mask. The address must be entered in octal with a leading zero (0) being recognized as an address rather then a line number. This address must be in the D-bank or in the Main PDB. You cannot set address **SNAPS** for the I-Bank or for PDBs besides the Main PDB. The snapping will start as soon as the Address becomes a valid Basic Mode address.

A **SNAP** for an Extended Mode VA is done in the same way. You just enter the VA (bdi+adr) in the **SNAP** part of the mask. The snapping will start as soon as the VA becomes a valid VA in the Extended Mode environment.

You can limit the number of **SNAPS** by entering program information in the **TRAP** part of the mask, just as for **SNAPS** set on DBAs or PDBs.

Forcing Aborts at the right spot at the right time

Having a dump from an abort taken at the right spot and at the right time is often the best way to solve problems. This, however, can often be very difficult to accomplish. **Visual Trace** allows you to run a trace, look at it and then decide when and where you would like the abort to occur.

Once you know where you want the abort to occur (the right spot), set a **TRAP** for the location using the **TRAP** part of the mask, and then run the transaction again. The trace will now include a hit every time the program passed the location you set the **TRAP** for. All of these hits are numbered sequentially in the trace file and then with analysis you can decide which of these hits would be the best time to take an abort.

Next use the **WHEN** part of the mask to force an abort at the right spot and at the right time by entering the hit number for the hit where you want the abort to occur in **`ABORT ON HIT NUMBER`**. Run the transaction again and this time **Visual Trace** will force a Contingency 014 abort – Breakpoint – when it gets to the right spot at the right time.

Forcing Aborts when a memory location gets destroyed

Some of the must complicated problems to solve are when data gets destroyed by a flawed process that has corrupted memory that doesn't belong to itself. This kind of memory destruction is normally very difficult to find as they are likely to cause aborts very far from where the problem actually is. The memory destroyed in this way can be related to common blocks, variables and defines in DBAs.

Visual Breakpoint Traps makes this kind of problem easy to solve. To do this, first set a Variable or DBA **SNAP** for the memory location that is being destroyed. Then select **`WHEN BELOW SNAP IS UPDATED`** and next run the transaction again. Even though you have not yet requested **Visual Breakpoint Traps** to abort the transaction, the trace file will show a hit every time the memory location was updated. Often times this is enough to solve the problem as the trace will show exactly where the memory location was updated, what it was updated to and what code or process performed the update.

Some very complex problems, however, will be affected by the fact that traces are turned ON and thus stop destroying the data you are monitoring as it – due to the trace itself – may now be located at a completely different memory location. As a consequence, the trace will NOT show anything wrong and inform only on legal updates to the snapped memory location.
(*See Controlling the AP DBA*)

This situation can occur when the process destroying the data does not own it but destroys data that belongs to another process. This can happen because turning traces ON for a transaction caused the memory used for DBAs to be changed. (*See Memory Allocation and Tracing*)

Visual Breakpoint Traps will aid you with this situation also as it enables you to run a trace without producing a trace file, thereby not altering the memory layout for the transaction. Since you do not get a trace file to look at, you will have to force the abort by using the **WHEN** part of the mask. To do this, keep both the **SNAP** and **TRAP** of the destroyed memory location active and enter the "bad" value of the destroyed memory location in the **ABORT WHEN SNAP VALUE EQUALS**. This informs **Visual Breakpoint Traps** to cause a Contingency 014 abort – Breakpoint – when the "bad" value is stored in the memory location.

Before running the transaction, you must inform **Visual Breakpoint Traps** to not produce a trace file. You do this by selecting **DROP AP DBA TO PRESERVE MEMORY LAYOUT** in the **TST:SNAP** mask.
(*See Visual Advanced Snaps*)

Now run the transaction again, but ensure you have turned OFF all other TST trace flags except **TST:TRAP** and **TST:SNAP**. This time you should get an abort informing you of the specific code that updated the memory location to contain the "bad" value.

Dynamic PRTADP calls

To set a Dynamic PRTADP call, just set a **TRAP** on a line number and enter the **SNAP** data you want displayed once the trap occurs. This logic is simple and effective and allows you to dynamically produce **SNAPS** that work just like PRTADP calls without having to change and recompile your program.

Defining Complex Traps

When setting **TRAPS**, **Visual Breakpoint Traps** reads the absolute to calculate the actual address the **TRAP** needs to be set on when the mask is updated. The address is then saved as part of the normal test and training logic for the CLR.

To ensure the **TRAP** occurs when it is expected to, it is vital that the **TRAP** address is calculated correctly. This will only be the case if the absolute used to calculate the **TRAP** address is the exact same absolute as the one executing. **Visual Trace** will initially search for the absolute in the Library Load File used to load the program into the HVTIP library/bank. If the HVTIP library/bank was loaded from a temporary file like TPF$, the normal search path will be used. (*See The system and personal search paths*)

For simple **TRAPS**, using a wrong absolute to calculate the **TRAP** address may result in the **TRAP** hits being a bit off. When working with complex **TRAPS**, however, the slightest miscalculation in the **TRAP** address will likely prevent the **TRAP** logic from working correctly.

Once the **TRAP** address is calculated, you will be informed of the absolute that was used to calculate the **TRAP** address. It is a good idea to ensure that this is the correct absolute.

You can use the below TST: function to see what trace flags (including **TRAPS**, **SNAPS** and **PERF**) are set for your device.

TST:DISPLAY or **TST:D**

This display will also inform you of what absolute was used to calculate the **TRAP** address.

Memory Allocation and Tracing

Traces are typically used to inform of the path taken by a transaction throughout its execution. In doing this, the standard Usas trace logic will use the AP DBA to hold the trace lines before they are written to the actual trace file. As the trace DBA is always the first DBA to be allocated, it will be located right after the CB control area at the top of the dynamic D-bank (CBAREA).

This will naturally cause every DBA allocated after the AP to be located at a different memory location compared to when traces are OFF. This is normally not a problem, but could result in X525 aborts if the AWA runs out of memory or it can make tracing difficult if you are analyzing a problem related to specific memory locations.

If you are tracing to solve a memory-related problem and the memory shifts location when traces are ON so that the error no longer occurs, you have two options that are both controlled by the **TST:SNAP** mask. (*See Visual Advanced Snaps*)

First, try retaining the original memory layout as much as possible by requesting **Visual Trace** to move the AP DBA down to the bottom of the AWA rather than having it at the top. This ensures that all other DBAs will be allocated at the same address location as when traces were OFF.

Second, you can request **Visual Breakpoint Traps** to prevent the AP DBA from being allocated. In this case, no trace will be written so you will have to work with **TST:TRAP** and forced aborts to solve the problem.

Visual Breakpoint Traps also needs a little memory to enable the **TRAPS** and **SNAPS**. This memory is not allocated in a DBA and thus will not influence DBA allocation by causing x525 aborts or memory shifting.

The **Visual Breakpoint Traps** work area is about 100 words and is located at the bottom of the FSTACK$ area. It is preceded with a "**VDA$**" token in the first word. **Visual Breakpoint Traps** protects this work area from being destroyed by normal stack allocation. However, it cannot protect a program transfer from taking the entire FSTACK$ area and thereby destroying the **Visual Breakpoint Traps** work area. This is an unlikely situation, but if the work area is compromised, the result of the trace will be unpredictable or it can even cause **Visual Breakpoint Traps** or the transaction itself to error. Though not ideal, it is the only area where **Visual Breakpoint Traps** can allocate a work area without affecting normal memory usage.

Visual Advanced Snaps

Advanced **SNAPS** allows you to snap special information. **Visual Advanced Snaps** can work alone or in combination with **Visual Breakpoint Traps**.

By entering the input below, the **Visual Advanced Snap** mask will be displayed.

TST:SNAP

The below mask allows you to enter the advanced snap information.

```
TST:SNAP/

      VISUAL ADVANCED SNAP INFORMATION

SNAP PFEDIT LINES                              ]N[
SNAP FREE CHAIN INFORMATION WHEN DESTROYED     ]N[
MOVE AP DBA DOWN TO REDUCE MEMORY SHIFTING     ]N[
DROP AP DBA TO PRESERVE MEMORY LAYOUT          ]N[

                                               ] [
```

In this mask, you can also request PFEDIT output lines to be routed to the trace file. This provides a useful way of seeing where you are in the processing when reading the trace file. Note that for practical reasons, the width of each PFEDIT line is limited to 71 characters.

You can also request **Visual Advanced Snaps** to monitor the AWA Free Chain and report when it detects the chain to be destroyed. Once the AWA Free Chain is destroyed, **Visual Advanced Snaps** will place a line informing of this fact in the trace file. This will only be reported once in order to avoid filling the trace file with this identical warning statement.

Controlling the AP DBA

The AP DBA is used by Usas Sys 11r2 to hold the trace lines before they are written to the trace file. When traces are ON, Usas Sys 11r2 allocates the AP DBA as the first DBA in the AWA. This memory shifting means that all other DBAs allocated in the AWA will be at different address locations than when traces are OFF. (*See Memory Allocation and Tracing*)

Normally this is not a problem, but some complex problems tend to go away when traces are ON and re-appear when traces are OFF. Naturally this makes finding this kind of problems very difficult. This nasty behavior is most likely due to memory allocation and is therefore affected by the fact that the AP DBA is causing all other DBAs to be shifted down in the AWA. By setting **`MOVE AP DBA DOWN TO REDUCE MEMORY SHIFTING`**, **Visual Advanced Snaps** will force the AP DBA to be located at the very bottom of the AWA rather than at the top. The memory locations of all other DBAs are thereby preserved and hopefully the problem will then be traceable and keep occurring when traces are ON. Note that the AP DBA is still present in the AWA and thus when moved to the bottom of the AWA is will start affecting the stack locations for HVTIP programs.

If forcing the AP DBA to the bottom of the AWA does not make the problem traceable the only other solution is to complete drop the AP DBA. If you select "**`DROP AP DBA TO PRESERVE MEMORY LAYOUT`**" **Visual Advanced Snaps** will allow **Visual Breakpoint Traps** to be effective while preventing the AP DBA from being allocated. Further, you must ensure that any TST: trace flag that will place information in the trace file wild is turned OFF as they will otherwise re-allocate the AP DBA. When the AP DBA is not allocated at all the memory usage is completely the same when traces are ON and OFF thus the problem will be traceable.

However, as the AP DBA is dropped no trace file information will be capture and produced by Usas Sys 11r2 and thus the trace will be empty. You will have to rely on **Visual Breakpoint Traps** to detect where and when the data gets destroyed. (*See Forcing Aborts when a memory location gets destroyed*)

Visual Performance Traces

Performance Traces allow you to capture performance information for a later **Performance Analysis** of your transaction. Once a **Performance Trace** has captured its data, it can be use in multiple **Performance Analysis.** Each **Performance Analysis** will report on how much time (CPU) was used by the transaction and where the time was spent within the transaction. (*See Creating The Performance Reports*)

To turn ON **Performance Traces** to capture performance data, enter the following:

TST:PERF

TST:CTR ICR TAC are prerequisites for the **Performance Traces** and must be turned ON along with **TST:PERF** to enable the capture of all the information needed. You can set all of these flags using **TST:VDA.**

Once you have turned your **Performance Trace** ON, run the transaction and obtain the trace file be entering:

TST:PRINT NOPR

The trace file will now contain all the performance information needed to enable a **Performance Analysis**.

Note that when traces are turned on, the CPU usage by a transaction will always be much higher than normal as the tracing itself takes time. You can request **Visual Performance Trace** to reduce the CPU times displayed by reducing all CPU values by an estimated trace overhead. This will cause CPU values to be closer to normal, but they should be considered estimates. (*See Setting up Advanced Performance Traces*).

Further note that **TRAPS** and **Performance Traces** are mutually exclusive and once one is turned ON the other will automatically be turned OFF.

Setting up Advanced Performance Traces

Setting up a **Performance Trace** ensures the performance information needed for a **Performance Analysis** is captured in the trace file. Analyzing the performance of a transaction should always start with a **Normal Performance Analysis** based on basic performance information. The **Normal Performance Analysis** reports what HVTIP programs and ACB or FLSS routines are called allowing you to determine the areas that may need further analysis by using **Advanced Performance Traces**.

Once the normal **Performance Analysis Report** has revealed a specific program or routine to be using more CPU than expected, you can use the **Advance Performance Traces** to capture a new performance trace requesting an internal routine to be meticulously monitored. When monitored in this way, the **Advanced Performance Trace** will capture performance information at the internal subroutine level for this specific internal subroutine enabling the later **Performance Analysis** to further breakdown the analysis and inform on the CPU usage of the internal subroutine separately.

While analyzing performance at internal subroutine level, one internal subroutine can be analyzed at a time. This means that you may have to run the transaction multiple times while tracing a new internal subroutine each time in order to have a complete analysis.

To analyze an internal subroutine, request that **Visual Performance Trace** monitor a specified internal subroutine and capture its performance data so it can be analyzed separately when included in the **Performance Reports.**

To display the mask for entering the performance information parameters, enter:

TST:PERF/program

The below example displays how the **SHOW** part of the mask is updated to request the internal subroutine MYSUB to be analyzed individually. MYSUB is an internal subroutine in the MYREL source program, which is mapped into the MYPROG absolute.

```
TST:PERF/

       VISUAL PERFORMANCE INFORMATION

THE VDA PERFORMANCE ANALYSIS WILL BREAK DOWN THE
QUANTUM TICS USED BY A TRANSACTION AND REPORT ON
THE CPU USAGE FOR EACH HVTIP PROGRAM.

FURTHER EACH BASIC MODE PROGRAM IS BROKEN DOWN
INTO THE MAIN PROGRAM AND MAPPED IN SUBROUTINES.

USING THIS MASK YOU CAN REQUEST AN INTERNAL FTN
SUBR. TO BE SEPARATELY SHOWN IN THE PERF REPORT.

SHOW THIS INTERNAL SUBR. SEPARATELY          ]Y[
THE INTERNAL SUBR. IS IN PROGRAM      ]MYPROG  [
 AND IN RELOCATABLE/OBJECT MODULE     ]MYREL   [
 AND THE INTERNAL SUBROUTINE NAME IS  ]MYSUB   [
SHOW PERF WITHOUT ESTIMATED TRACE OVERHEAD   ]N[

                                             ] [
```

You need to be sure the **SHOW THIS INTERNAL SUBR. SEPARATELY** is selected. Every time a new internal subroutine is updated in this mask, you must rerun the transaction to create a new trace file containing information for the specified internal subroutine before you can create new **Performance Reports** analyzing the CPU use for this internal subroutine. (*See Creating the Performance Report*)

When traces are turned ON, the CPU usage for a transaction will always be much higher than normal as the tracing itself takes time. You can request **Visual Performance Trace** to lower the CPU times by reducing all CPU values by an estimated trace overhead. This will cause CPU values to be closer to normal, but they still need to be considered as estimates.

Normally there isn't a need to reduce the CPU usage calculations by the estimated trace overhead as all values are shown as percentages anyway. Thus, the relative usage of CPU by a transaction will not be dramatically affected. Further, working on the CPU values that include the trace overhead only means the calculations being used are somewhat larger numbers, but they are usually very helpful in determining areas that are performing excessive processing.

To reduce the CPU values by the estimated overhead, select **SHOW PERF WITHOUT ESTIMATED TRACE OVERHEAD** and run the transaction again to create a new trace file that will be ready for **Performance Analysis**.

Creating the Performance Reports

Before you can create a **Performance Report**, you must first produce a trace file containing performance information. Once you have this file, you can create as many **Performance Reports** as you like with each one being based on the same information for accurate comparison purposes.

To create a **Performance Report**, enter.

TRCE: ***[qualifier*]filename.*** **PERF/*report***

This creates a complete **Performance Report** for each transaction in the trace file. You can limit the **Performance Reports** to be produced for a specific function code by using the below input.

TRCE: ***[qualifier*]filename.*** *function-code* /**PERF/*report***

When a program name is specified on the input, only the specified program is analyzed and the **Performance Report** will be limited to holding information for the specified program.

TRCE: ***[qualifier*]filename.*** *program* /**PERF/*report***

You can also combine these inputs to select a specific program in a specific function.

TRCE: ***[qualifier*]filename.*** *function-code*/*program* /**PERF/*report***

Report is used to inform which of the three (3) **Performance Reports** are to be created. When omitted, the **Performance Report** will be the default or **Normal Performance Report**. When present, it can be one of the two keywords - **SUBS** or **CALLS** - each representing a report type.

The **Performance Report** will report how much CPU time was used by the entire transaction. It will also break up the transaction in different ways to give you various views of the transactions CPU usage. Further, it will report on how often ACB and FLSS routines were called along with how much CPU time these calls totaled for each program. Finally, it will report how much CPU was used by the main parts of the transaction programs.

Once a **Performance Analysis Report** has been created, you should take a note of routines that process as an unexpectedly large percentage of the entire transaction. This can indicate that the program or routine is using more CPU than it should and analysis may show that this code could be more efficient.

Due to the nature of Mixed Mode, it should be noted that Extended Mode programs that call Basic Mode ACB routines can create extra overhead when the call-parameter data needs to be moved in memory. The CPU time for this overhead is measured and counted on the Extended Mode program and NOT on the Basic Mode ACB routine.

The Normal Performance Report

This report lists the CPU usage that in incurred by each HVTIP program along with ACB and FLSS routines within a transaction. It provides a quick overview of the overall performance of the transaction and helps you decide whether further **Performance Analysis** is needed.

Below is an example of a RDMP: transaction:

```
TRCE: TIP$*T$VUELKB(5). PERF
............................ PERFORMANCE ANALYSIS ..............................
ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS       ICP %     FUNC %
ICP               0       000000101174     0.00009383      82.80       1.39
XTFLWR            1       000000010663     0.00001272      11.23       0.18
PRGZOM            1       000000004547     0.00000676       5.96       0.10
TOTAL CALLS       2       000000015432     0.00001948      17.19       0.28
-------------------------------------------------------------------------------
TOTAL ICP                 000000116626     0.00011332      99.99       1.68

ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS    SEDDMP %     FUNC %
SEDDMP            1       000001672752     0.00137342      58.61      20.43
XBCA(STACK)      26       000000307221     0.00028660      12.23       4.26
PFDM03           11       000000127045     0.00012522       5.34       1.86
XBCF(STACK)      25       000000301132     0.00027782      11.85       4.13
PFNBAS           12       000000135333     0.00013436       5.73       1.99
XBCA              4       000000037052     0.00004470       1.90       0.66
XBCF              4       000000035712     0.00004299       1.83       0.63
PFDTSD            1       000000010121     0.00001173       0.50       0.17
PVOCTA            3       000000030045     0.00003462       1.47       0.51
PFDM04            1       000000010034     0.00001158       0.49       0.17
TOTAL CALLS      87       000001242161     0.00096966      41.34      14.42
-------------------------------------------------------------------------------
TOTAL SEDDMP              000003135133     0.00234308      99.95      34.85

ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS    SEDINP %     FUNC %
SEDINP            1       000001152275     0.00088933      56.04      13.23
XBCA(STACK)      16       000000176502     0.00018211      11.47       2.70
XBCA              4       000000036703     0.00004441       2.79       0.66
XBCF(STACK)      16       000000174434     0.00017913      11.28       2.66
PFNBAS            1       000000007566     0.00001111       0.70       0.16
PFDM03           20       000000227273     0.00021769      13.71       3.23
PFMA13            1       000000011132     0.00001319       0.83       0.19
PFMA04            3       000000032354     0.00003805       2.39       0.56
PFMA01            1       000000010127     0.00001175       0.74       0.17
TOTAL CALLS      62       000000744757     0.00069748      43.91      10.37
-------------------------------------------------------------------------------
TOTAL SEDINP              000002117254     0.00158682      99.95      23.60

ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS    SEDOUT %     FUNC %
SEDOUT            8       000000252722     0.00024580      36.01       3.65
XBCA(STACK)     16(8)     000000076475     0.00009005      13.19       1.33
PFEDIT            8       000000262546     0.00025700      37.65       3.82
XBCA              1(1)    000000000000     0.00000000       0.00       0.00
XBCF(STACK)     16(8)     000000076217     0.00008957      13.12       1.33
TOTAL CALLS      41       000000457462     0.00043663      63.96       6.49
-------------------------------------------------------------------------------
TOTAL SEDOUT              000000732404     0.00068243      99.97      10.15
```

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDFND %	FUNC %
SEDFND	1	000000061561	0.00007150	76.20	1.06
XBCA(STACK)	1	000000007403	0.00001079	11.50	0.16
PRTADP	1(1)	000000000000	0.00000000	0.00	0.00
XBCF	1(1)	000000000000	0.00000000	0.00	0.00
XBCF(STACK)	1	000000010013	0.00001153	12.29	0.17
TOTAL CALLS	4	000000017416	0.00002233	23.79	0.33
TOTAL SEDFND		000000101177	0.00009383	99.99	1.39

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDSHW %	FUNC %
SEDSHW	4	000000106327	0.00010127	53.36	1.50
XBCA(STACK)	4	000000036533	0.00004412	23.24	0.65
XBCF(STACK)	4	000000036667	0.00004437	23.38	0.66
TOTAL CALLS	8	000000075422	0.00008850	46.62	1.31
TOTAL SEDSHW		000000203751	0.00018977	99.98	2.82

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDVAL %	FUNC %
SEDVAL	1	000000211441	0.00019784	74.49	2.94
XBCA(STACK)	2	000000017452	0.00002241	8.43	0.33
PVASCB	2	000000017244	0.00002203	8.29	0.32
XBCF(STACK)	2	000000020146	0.00002329	8.77	0.34
TOTAL CALLS	6	000000057064	0.00006774	25.49	1.00
TOTAL SEDVAL		000000270525	0.00026558	99.98	3.95

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDFNC %	FUNC %
SEDFNC	1	000000136322	0.00013578	66.27	2.02
XBCA(STACK)	3	000000027566	0.00003412	16.65	0.50
XBCF(STACK)	3	000000030242	0.00003497	17.06	0.52
TOTAL CALLS	6	000000060030	0.00006910	33.71	1.02
TOTAL SEDFNC		000000216352	0.00020488	99.98	3.04

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDCLS %	FUNC %
SEDCLS	1	000000021315	0.00002502	24.16	0.37
XBCA(STACK)	2(1)	000000007405	0.00001080	10.42	0.16
PFEDIT	1	000000047754	0.00005747	55.48	0.85
XFSLK	1(1)	000000000000	0.00000000	0.00	0.00
XFSWU	1	000000007115	0.00001028	9.92	0.15
PROTOC	1(1)	000000000000	0.00000000	0.00	0.00
XBCF(STACK)	1(1)	000000000000	0.00000000	0.00	0.00
TOTAL CALLS	7	000000066476	0.00007855	75.82	1.16
TOTAL SEDCLS		000000110013	0.00010358	99.98	1.54

ROUTINE	CALLS(NEST)	QUANTUM TICS	CPU.SUPS	SEDPAG %	FUNC %
SEDPAG	1	000000315347	0.00029548	29.90	4.39
XBCF(STACK)	13(10)	000000026767	0.00003305	3.34	0.49
XBCA(STACK)	12(10)	000000016712	0.00002142	2.16	0.31
PRTADP	4	000000035622	0.00004283	4.33	0.63
XBCA	1	000000010133	0.00001176	1.19	0.17
XFSRL	1	000000007560	0.00001110	1.12	0.16
PFDM04	9	000000104310	0.00009835	9.95	1.46
PFEDIT	10	000000502321	0.00046368	46.93	6.89
PROTOC	10(10)	000000000000	0.00000000	0.00	0.00
XDFWR	1	000000007063	0.00001021	1.03	0.15
TOTAL CALLS	61	000000741352	0.00069243	70.05	10.30
TOTAL SEDPAG		000001256721	0.00098791	99.95	14.69

```
ROUTINE       CALLS(NEST)  QUANTUM TICS      CPU.SUPS    PXCRTO %     FUNC %
PXCRTO            1        000000027533     0.00003405      22.62       0.50
XBCA(STACK)       3(2)     000000010114     0.00001171       7.78       0.17
XTRANO            1        000000100640     0.00009321      61.92       1.38
MCBOUT            2(2)     000000000000     0.00000000       0.00       0.00
XTFLWF            1(1)     000000000000     0.00000000       0.00       0.00
XBCF(STACK)       3(2)     000000010015     0.00001154       7.66       0.17
TOTAL CALLS      10        000000120771     0.00011647      77.36       1.73
------------------------------------------------------------------------------
TOTAL PXCRTO                000000150524     0.00015052      99.98       2.23

TOTAL RDMP                  000011101570     0.00672177                 99.94
```

Nested ACB calls

ACB routine calls are often nested whereby one ACB routine calls another ACB routine before returning. To avoid counting the CPU time as part of both of the nested ACB routines, the CPU time for the nested call is set to zero. The number of nested calls made to a routine is shown within parentheses in the display.

```
ROUTINE       CALLS(NEST)  QUANTUM TICS      CPU.SUPS    MYPROG %     FUNC %
PXCRTO            1        000000027533     0.00003405      22.62       0.50
XBCA(STACK)       3(2)     000000010114     0.00001171       7.78       0.17
```

In this example, XBCA(STACK) was called 3 times with 2 of them being nested calls. This indicates that the total CPU time spent in XBCA (0.00001171) was the result of 1 direct call.

The SUBS Performance Report

The SUBS Performance Report provides more detail by breaking down all of the Basic Mode HVTIP programs into their main and mapped-in subroutines. This will assist you in determining if **Advanced Performance Traces** should be used to further breakdown the analysis on an internal subroutine level.

In the below report, the **Performance Analysis** has been limited to the HVTIP main program SEDINP with **Visual Performance** being requested to display all mapped-in subroutines by using the **SUBS** option.

```
TRCE: TIP$*T$VUELKB(5). SEDINP/PERF/SUBS
.......................... PERFORMANCE ANALYSIS ..............................
ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS    SEDINP %     FUNC %
SEDINP             1      000000741322     0.00069236      43.63      10.30
XBCA(STACK)        1      000000010063     0.00001164       0.73       0.17
PFNBAS             1      000000007566     0.00001111       0.70       0.16
PFDM03            20      000000227273     0.00021769      13.71       3.23
PFMA13             1      000000011132     0.00001319       0.83       0.19
PFMA04             3      000000032354     0.00003805       2.39       0.56
PFMA01             1      000000010127     0.00001175       0.74       0.17
XBCF(STACK)        1      000000007351     0.00001072       0.67       0.15
TOTAL CALLS       28      000000332352     0.00031418      19.77       4.67
-------------------------------------------------------------------------------
SEDINP(SEDXBC)     4      000000061067     0.00007062       4.45       1.05
XBCA(STACK)        4      000000037520     0.00004552       2.86       0.67
XBCA               4      000000036703     0.00004441       2.79       0.66
XBCF(STACK)        4      000000036526     0.00004410       2.77       0.65
TOTAL CALLS       12      000000135151     0.00013404       8.42       1.99
TOTAL SEDINP(SEDXBC)      000000216240     0.00020467      12.87       3.04
-------------------------------------------------------------------------------
SEDINP(SEDFD)     11      000000127664     0.00012634       7.96       1.87
XBCA(STACK)       11      000000126677     0.00012494       7.87       1.85
XBCF(STACK)       11      000000126335     0.00012430       7.83       1.84
TOTAL CALLS       22      000000255234     0.00024924      15.70       3.70
TOTAL SEDINP(SEDFD)       000000405120     0.00037559      23.66       5.58
-------------------------------------------------------------------------------
TOTAL SEDINP              000002117254     0.00158682      99.93      23.60
```

When an **Advanced Performance Trace** is analyzed, any separately monitored internal subroutine will be shown in the SUBS report as if they were a mapped-in subroutine. This includes having separate counters and CPU times for all of its ACB and FLSS calls.

Due to the nature of Extended Mode programs, they cannot be automatically broken down into the main program and linked in Object Modes in the same way as Basic Mode programs can be analyzed. However, you can select an entire Object Mode or any internal subroutine within an Object Mode to be separately analyzed in the **Performance Reports**. (*See Setting up Advanced performance Traces*)

The CALLS Performance Report

The **CALLS Report** is used to report on the total number of ACB and FLSS calls that were made along with their associated CPU usage. It should be used when you want to analyze the performance of an ACB or FLSS routine rather than of a specific transaction.

```
TRCE: TIP$*T$VUELKB(5). PERF/CALLS
............................ PERFORMANCE ANALYSIS ..............................
ROUTINE      CALLS(NEST)  QUANTUM TICS      CPU.SUPS     HVTIP %
HVTIP                     000004550077     0.00346336      51.52
XTFLWR            1       000000010663     0.00001272       0.18
PRGZOM            1       000000004547     0.00000676       0.10
XBCA(STACK)     85(21)    000000760451     0.00071418      10.62
XBCA            10(1)     000000106110     0.00010087       1.50
XBCF(STACK)     84(21)    000000752323     0.00070531      10.49
PFNBAS            13      000000145121     0.00014548       2.16
PFDM03            31      000000356340     0.00034292       5.10
PFMA13             1      000000011132     0.00001319       0.19
PFEDIT            19      000001035043     0.00077816      11.57
PFMA04             3      000000032354     0.00003805       0.56
PFMA01             1      000000010127     0.00001175       0.17
PRTADP             5(1)   000000035622     0.00004283       0.63
XBCF               5(1)   000000035712     0.00004299       0.63
PFDTSD             1      000000010121     0.00001173       0.17
PVASCB             2      000000017244     0.00002203       0.32
PVOCTA             3      000000030045     0.00003462       0.51
PFDM04            10      000000114344     0.00010994       1.63
XFSLK              1(1)   000000000000     0.00000000       0.00
XFSWU              1      000000007115     0.00001028       0.15
PROTOC            11(11)  000000000000     0.00000000       0.00
XFSRL              1      000000007560     0.00001110       0.16
XTRANO             1      000000100640     0.00009321       1.38
MCBOUT             2(2)   000000000000     0.00000000       0.00
XTFLWF             1(1)   000000000000     0.00000000       0.00
XDFWR              1      000000007063     0.00001021       0.15
TOTAL CALLS      294      000004331471     0.00325840      48.37
-------------------------------------------------------------------
TOTAL RDMP                000011101570     0.00672177      99.89
```

Because all of the **Performance Analysis Reports** are produced from the same trace file, their data completely relates to one another for comparison purposes. The various reports can assist by providing statistics in many ways so that a transaction can be properly analyzed and areas for improving the code can be identified.

Help Requests

You can request online help to assist you in formatting your requests for viewing the Usas trace file by entering the following:

TRCE: *[qualifier*]filename.* HELP

This will display general input formats for the TRCE: functionality.

Table of Contents