z/TPF V1.1

2013 TPF Users Group

Title: z/TPF Debugger Education

Joshua Wisniewski
Ongoing TPF Education

AIM Enterprise Platform Software
IBM z/Transaction Processing Facility Enterprise Edition 1.1

Any reference to future plans are for planning purposes only. IBM reserves the right to change those plans at its discretion. Any reliance on such a disclosure is solely at your own risk. IBM makes no commitment to provide additional information in the future.

Agenda

- What's new in the realm of debugger education?
  - Education resources and links
- Debugger education articles
  - Problem diagnosis
  - Custom communication packages
  - Determining code path
  - Hints and Tips
  - Starting the debugger effectively
- Q & A

What's new in the realm of debugger education?

- A new set of practical education articles have been written.
  - They focus on how to use debugger features together to solve problems.
  - They also focus on the lesser known or hard to find features.
  - A sample of this content will be the main focus this presentation.
  - The list of the new articles is available on the next slide.
- A new set of appendices have been added to the z/TPF Application Modernization using Standard and Open Middleware Redbook
  - They focus on step by step examples of how to use debugger features. These appendices are applicable to anyone new to the TPF Toolkit or wanting to learn about a variety of features.

Education resources and Links

- The following resources focus on how to use debugger features together to solve problems and on lesser known features.
- http://www-01.ibm.com/software/htp/tpf/. See the Fast links section on the lower left side. Select Tools -> z/TPF Debugger and then view the contents of the education material table.
  - **developerworks.com article**
    - Debugging Entry Control Blocks created by custom communication packages on z/TPF
  - **Debugger education articles**
    - Determining code path
    - Starting the debugger effectively
    - Problem diagnosis
    - Hints and Tips

Education resources and Links

- These resources are a good source for seeing step by step usage:
- http://www-01.ibm.com/software/htp/tpf/. See the Fast links section on the lower left side. Select Tools -> z/TPF Debugger and then view the contents of the education material table.
  - **z/TPF Application Modernization using Standard and Open Middleware Redbook**
    - There are several appendices with step by step demonstrations of building and loading an application in the TPF Toolkit, Web Services features, Debugger, Code Coverage Tool, Performance Analyzer, Dump viewer, Trace Log and etc.
  - **Debugger Demo Movie**
    - This demo movie was created several years ago to highlight the function that was available at that time. Even though this movie is out of date, the education delivered in this format has been found to be very useful and the core function described continues to exist.

Education resources and Links

- These existing resources are a good source learn what functionality exists.
  - http://www-01.ibm.com/software/htp/tpf/. See the Fast links section on the lower left side.
    - **TPFUG presentations** – select TPFUG Presentations. A debugger and TPF Toolkit update is often provided at each TPFUG to announce new features, provide education and so on. These presentations are usually given in the TPF Toolkit Task Force or the Development Tools Subcommittee.
    - **The Debugger User's Guide** – select TPF Family Libraries -> Open Current Information Center -> z/TPF PUT -> Library -> Debugger User's Guide.
  - **TPF Toolkit help** that is found in the Help menu also provides information regarding the features that are available. Select the Help menu -> Help contents. Then select Debugging TPF Applications, Analyzing Code Coverage of TPF Applications, or Analyzing Performance of TPF Applications.

Problem diagnosis

- Topics
  - Dump viewer
  - Debugging stack corruption
  - Debugging heap corruption
  - Debugging infinite loops
  - Debugging memory leaks

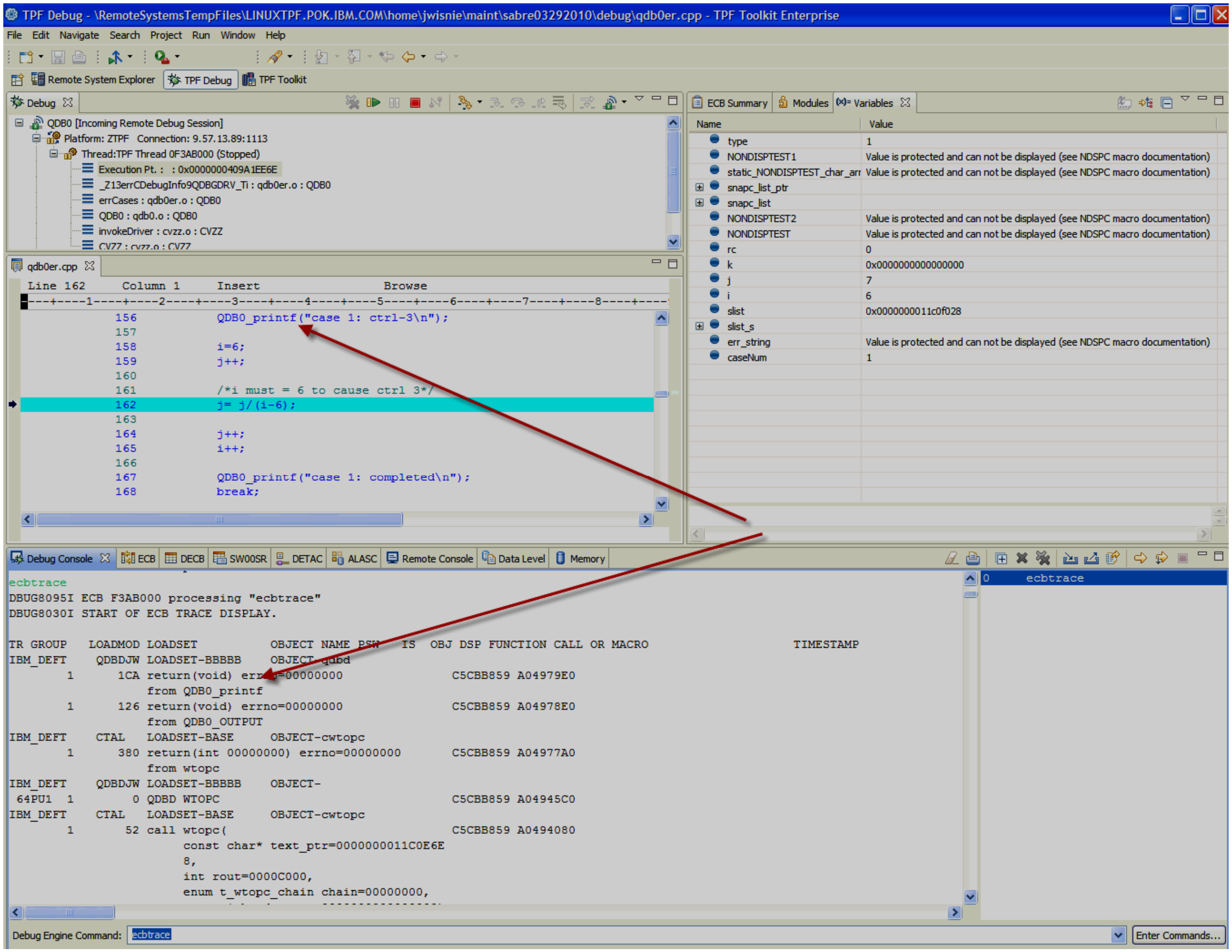
Dump viewer

- The dump viewer is a debugger like interface to view the contents of a dump.
  - The dump viewer is especially useful for C/C++ code with the ability to use the variables view to see all C/C++ variables at a glance. You can click through the stack frames and see C/C++ variables on previous stack frames.
  - The dump viewer provides the ability to apply XML maps in the memory views of given data areas to make it easier to read the data in the memory of the application.
  - Most debugger views will work as normal such as the SW00SR view, DETAC view, DECB view, TPF malloc view and so on, which could be difficult or impossible to view in a traditional z/TPF dump.
- Enter ZASER DUMPON DBUG to collect dump viewer dumps.
- The user exit UDDC_debuggerDumpCaptureUserExit in cdbaux.cpp allows you to capture additional data areas.
- These dumps are portable for viewing from z/TPF system to z/TPF system because the program attribute table (PAT) entries, database definition (DBDEFs), and so on are completely copied to the dump file.

Dump viewer

- The ECB trace can tell you what the ECB was doing recently. It will show you the macros and functions called as well as parameters passed in and values returned. The ECB trace is available while viewing dumps through the debug console command ECBTrace. A variety of other debug console commands are available.

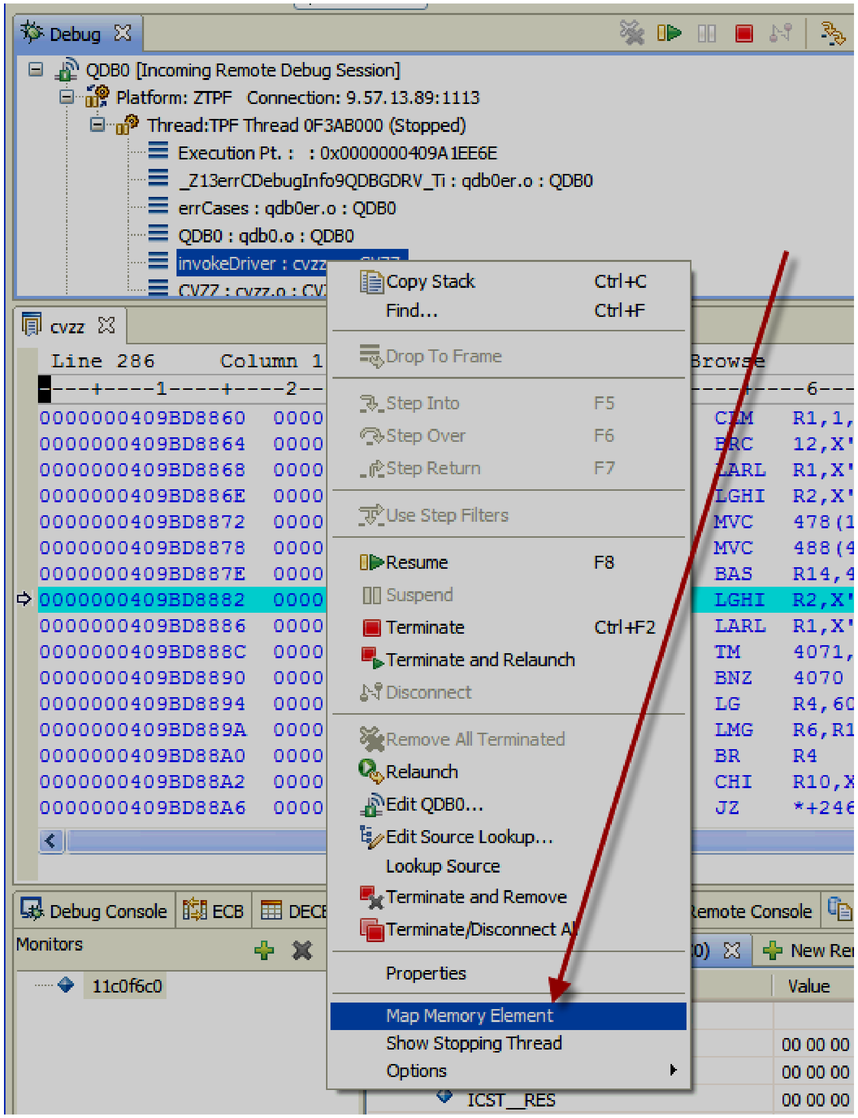
Dump viewer


TPF Debug - \RemoteSystemsTempFiles\LINUXTPF.POK.IBM.COM\home\jwisnie\maint\sabre03292010\debug\qdb0er.cpp - TPF Toolkit Enterprise
Debug Engine Command: ecbtrace

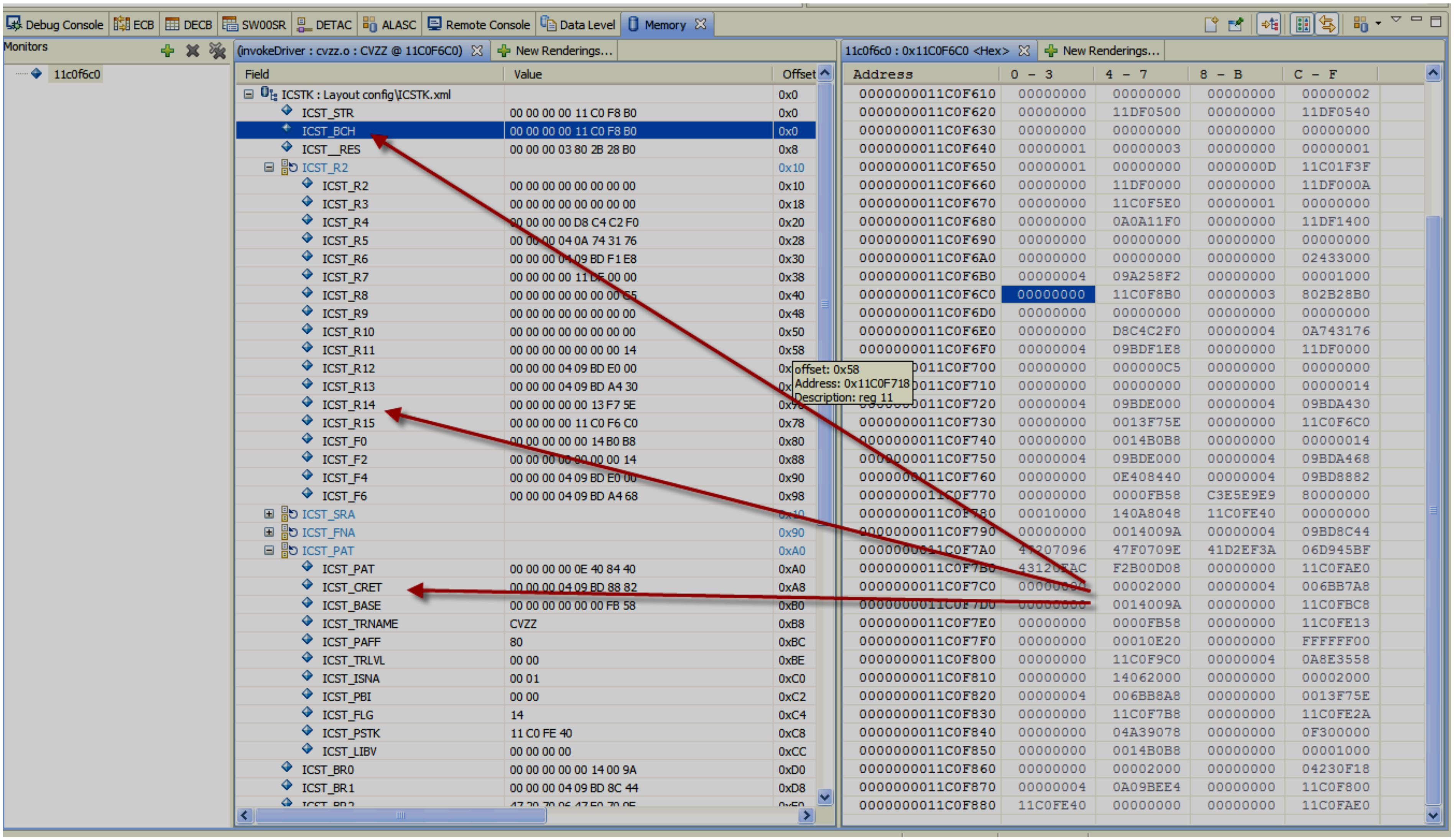
Debugging stack corruption

- **The following techniques apply to both the debugger and dump viewer.**
  - Click through the stack frames in the debug view and see what the local variable values are in the Variables view. You may notice that a character array containing a valid string appears to pour over into other variables in your stack. This can be an indication that your application is mishandling that string variable.
  - As you click through the stack frames, the properties view will show you details about that stack frame (size, address, etc).
  - You can also see the contents of the stack frame. Right click on a stack frame and choose map memory element to open an XML map of the stack frame in the memory. View picture on the next slide.

Debugging stack corruption



Debugging stack corruption

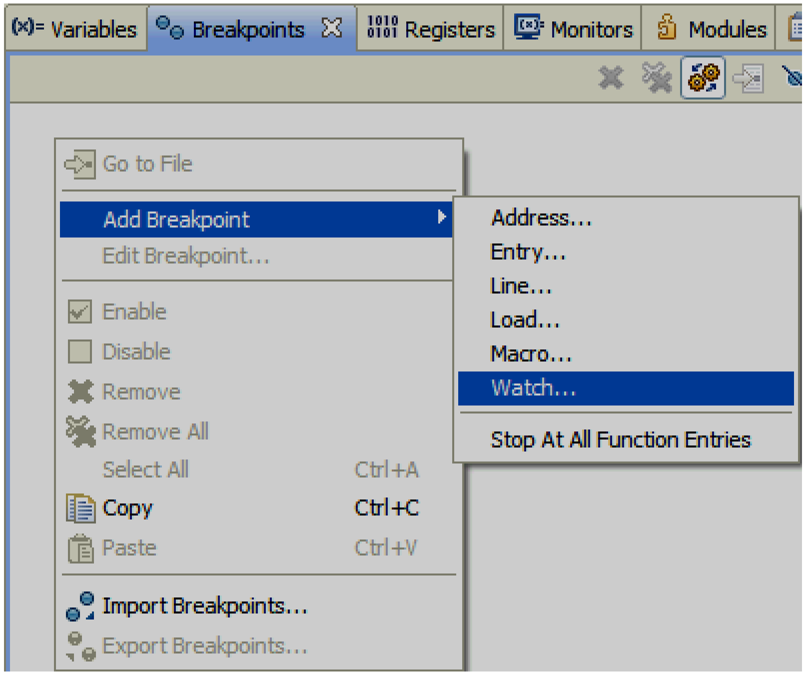


Debugging stack corruption

- **A couple things to take notice of in the stack frame:**
  - Register 14 (R14) is the typical return address register in the z/TPF system. However, if R14 points into CPS0, it is likely a C/C++ cross module call and the return address is found in CRET.
  - A bad back chain pointer (BCH) often indicates that the application is overwriting the stack by way of a memcpy, MVC, and so on.
  - This tip works frequently. Look at the stack contents rendered in EBCDIC or ASCII for a text string. Try doing a grep for that string in your application code. sprintf and similar functions are often the cause of stack corruption and this approach has been used to solve many of these types of dumps.
  - Another approach is to examine the contents of the entry control block (ECB) trace for function and macro parameters and return values that point into the stack address range as they may be the cause of the stack corruption.

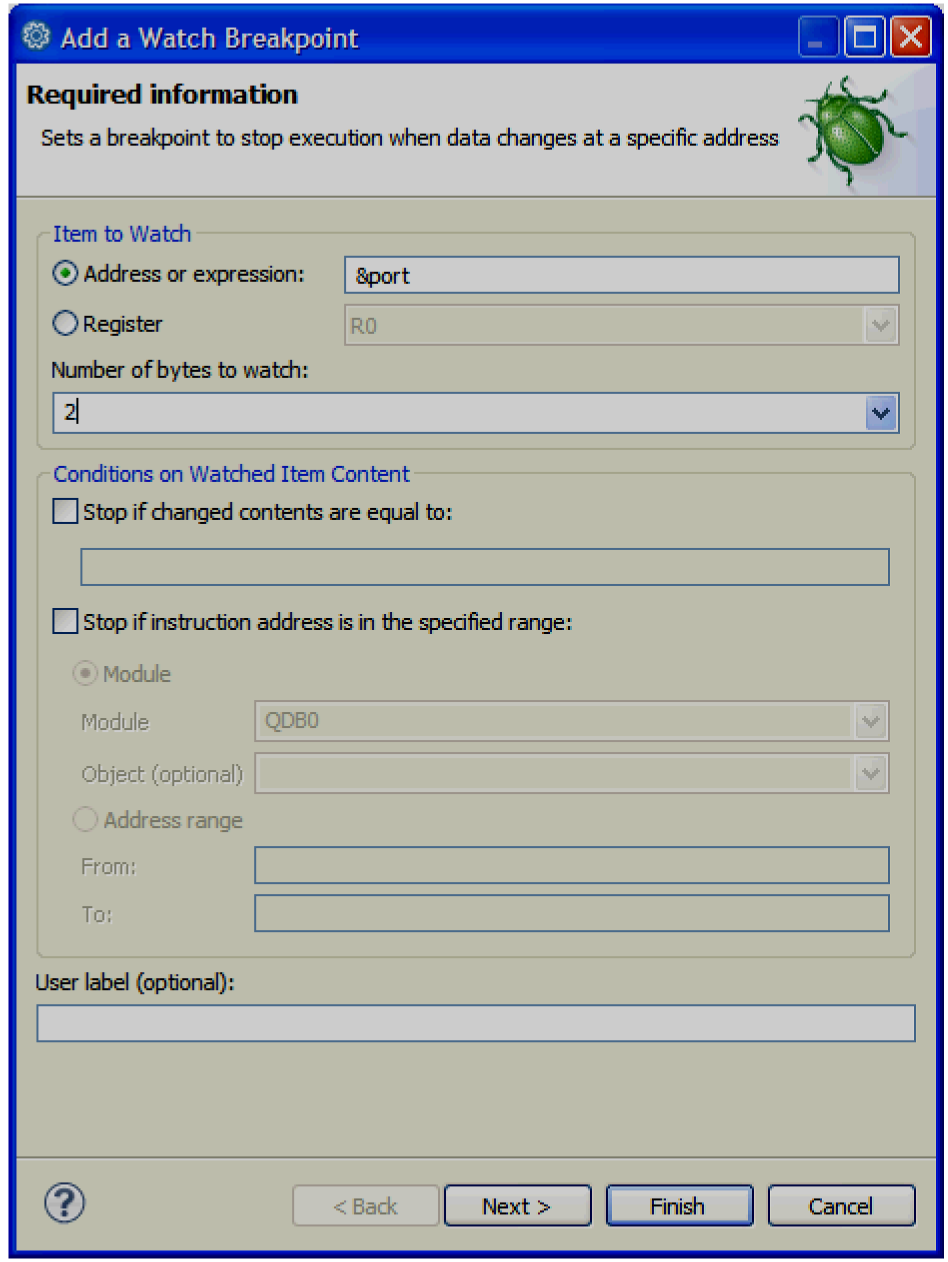
Debugging stack corruption

- If you are using the debugger and know that a particular stack address will become corrupted (such as the back chain pointer or a variable such as i), you can use the watch breakpoint support to stop the debugger when the change occurs.

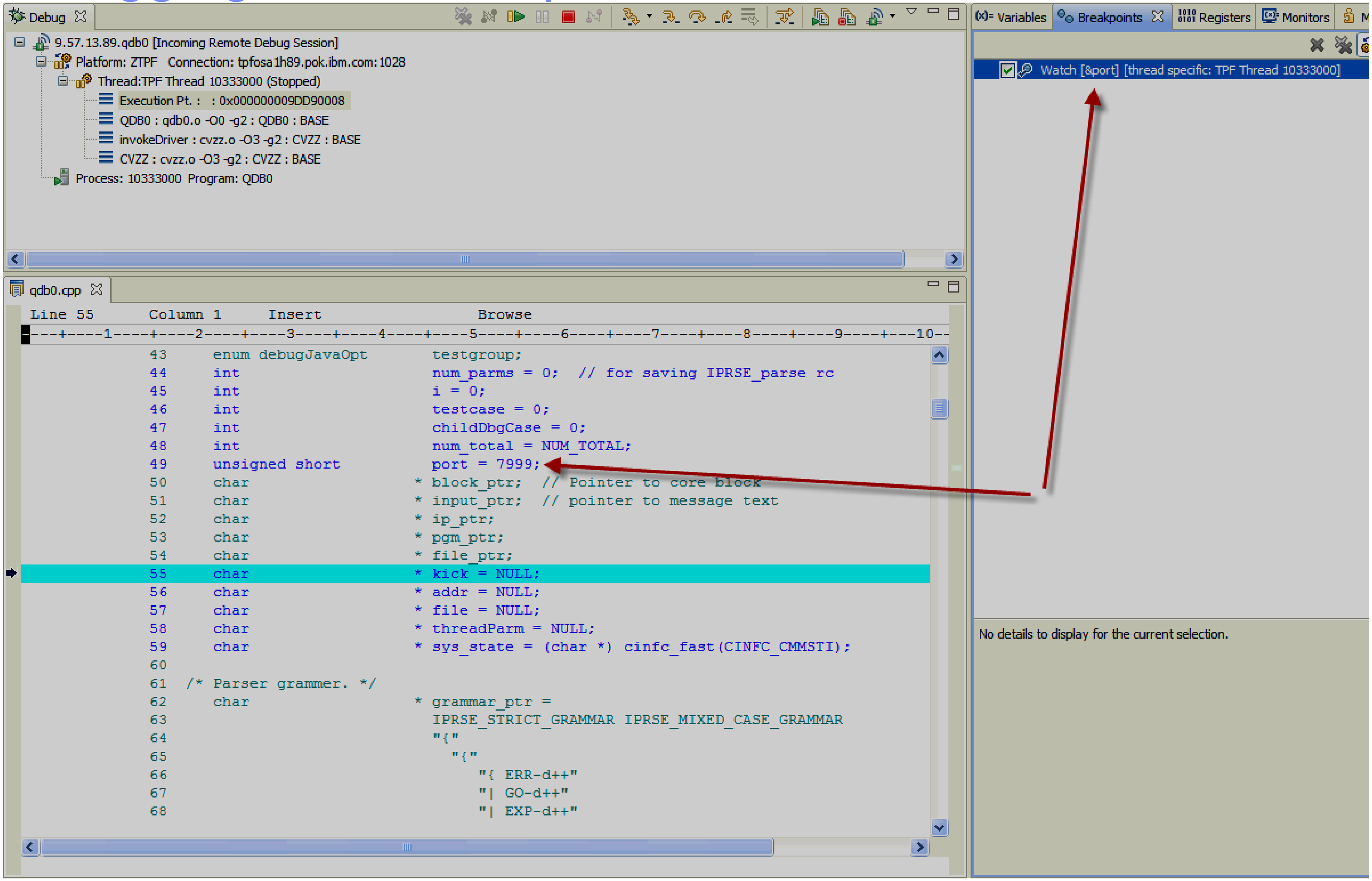


Debugging stack corruption

- Enter an address as 0x1234, a pointer expression or & of the variable such as &port.
- The debugger will stop at the source line/instruction after the source line/instruction that modified the storage.



Debugging stack corruption



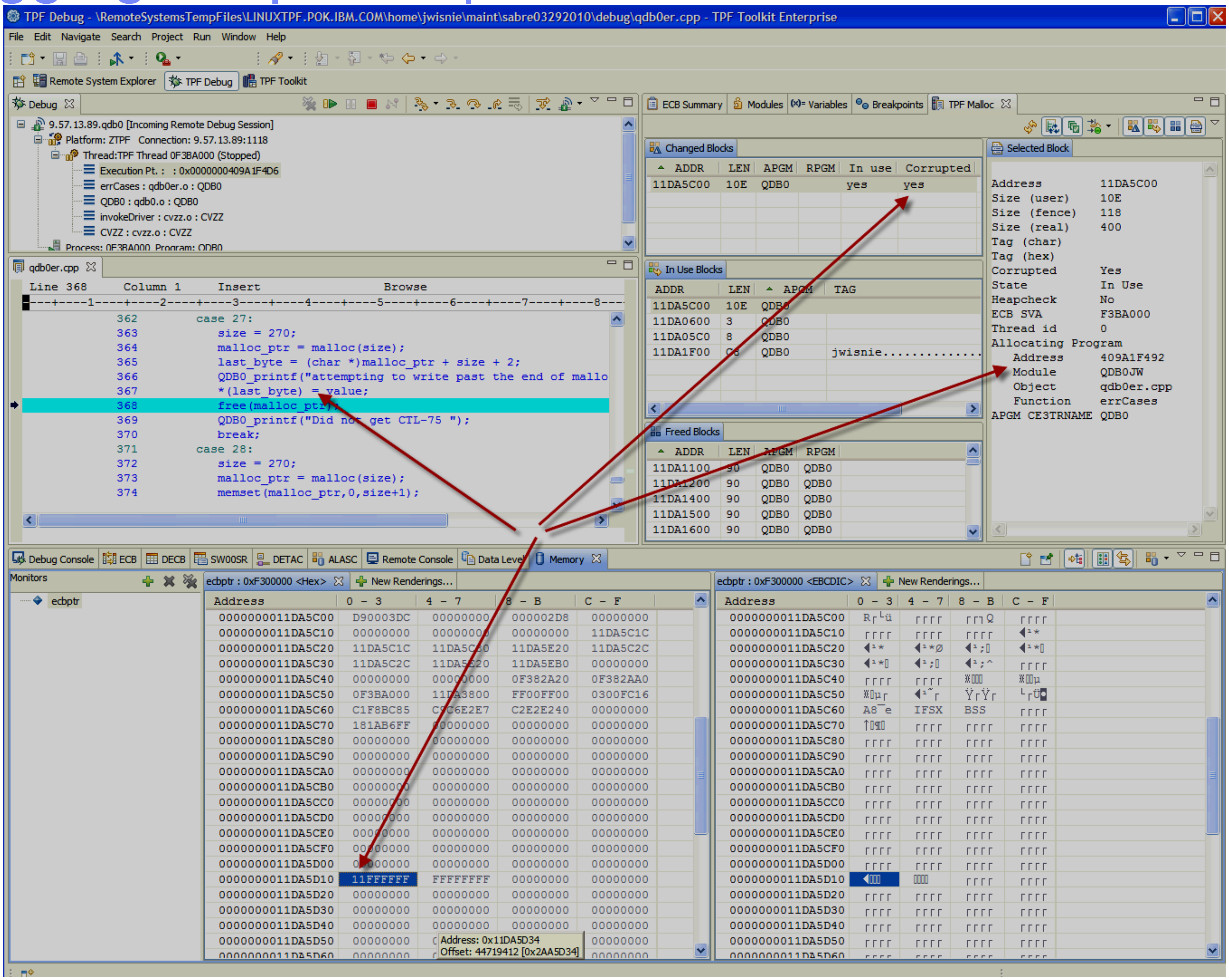
Debugging heap corruption

- A CTL-75 is a dump indicating that heap (malloc) corruption has occurred.
- In z/TPF this dump is typically issued when the 0xFFFFFFFF FFFFFFFF fence field immediately following an allocated malloc block is corrupted.
- However, this detection for the CTL-75 dump occurs when the malloc block is freed.
- CTL-75 dumps occur in the control program and as a result, you can not run the debugger to the dumping location or use register by system error for these dumps.

Debugging heap corruption

- The TPF Malloc view can be used to locate corruption of malloc blocks.
- If the corruption column is shown in the Malloc view, the corruption detection will be performed. Malloc entries that are corrupted will appear in the changed pane at all times, as shown on the next slide.
- One thing to note, using corruption detection in the TPF Malloc view may impact debugger performance.

Debugging heap corruption

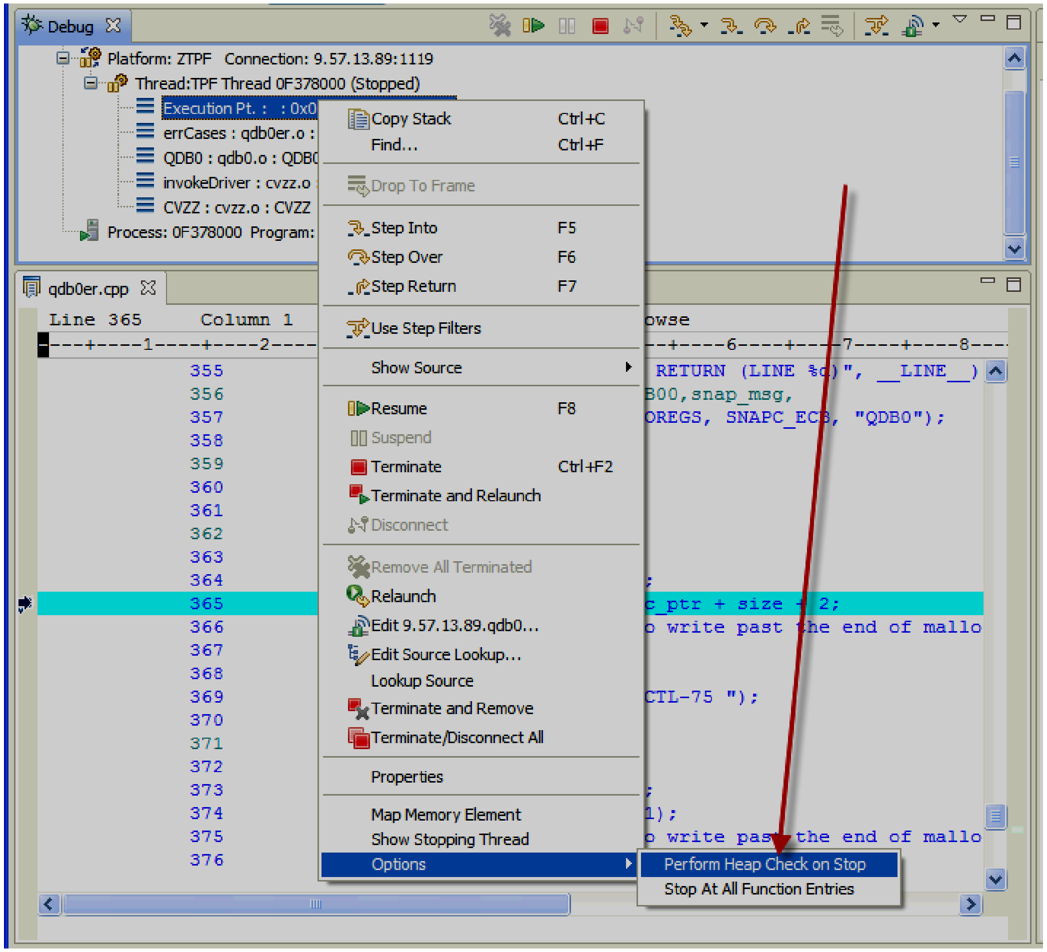


Debugging heap corruption

- While the TPF malloc view is a great way to learn about your malloc blocks and effectively shows you what corruption has occurred, it cannot indicate when that corruption occurred.
- The **perform heap check on stop** feature tells the debugger to detect any heap corruption whenever the execution of the application is stopped.
- When heap corruption is detected, a pop up window is displayed indicating that corruption has been detected.
- However, the user must step or run the application such that the application is periodically stopping.

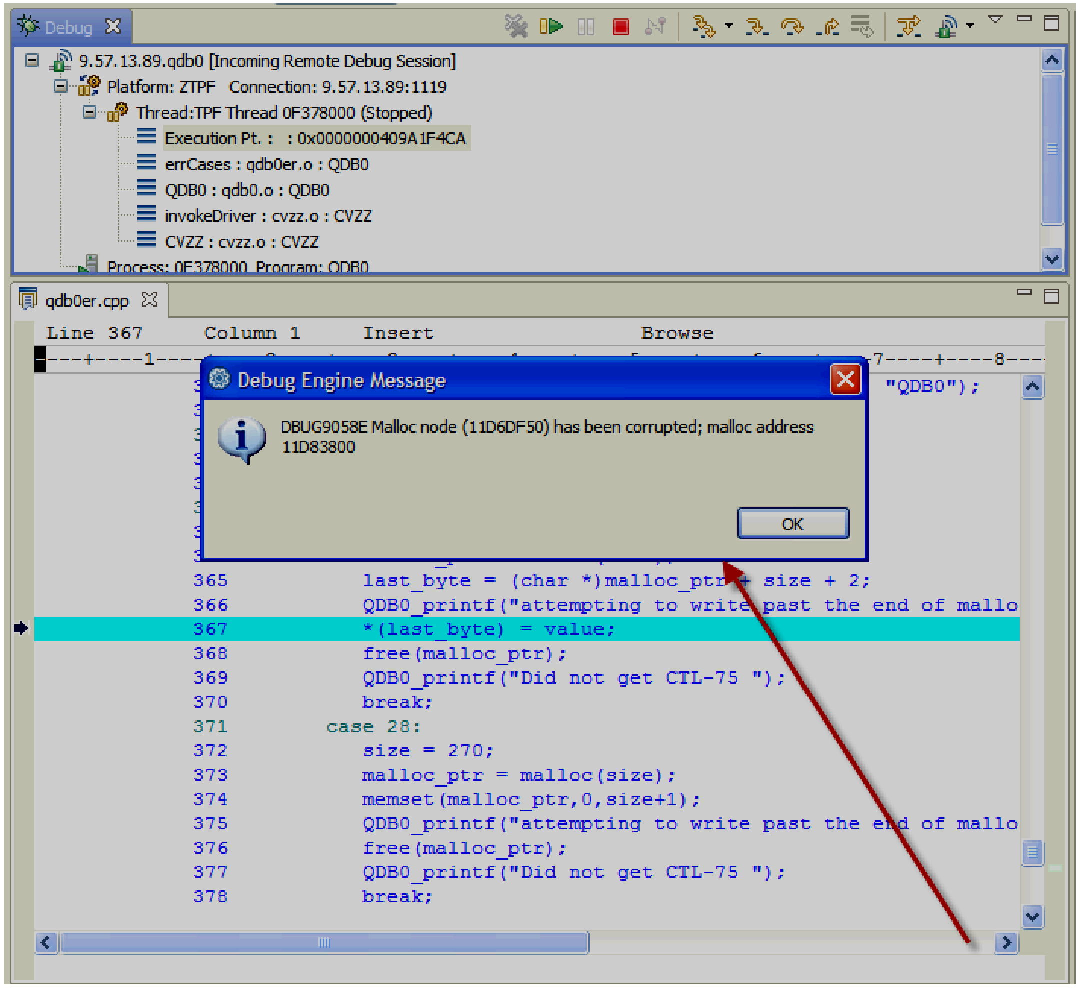
Debugging heap corruption

- To turn on the **perform heap check on stop** feature, right click in the stack frame and choose **perform heap check on stop**.



Debugging heap corruption

- When corruption is detected, as in this case where a step into each line occurred, a pop up appears like this:



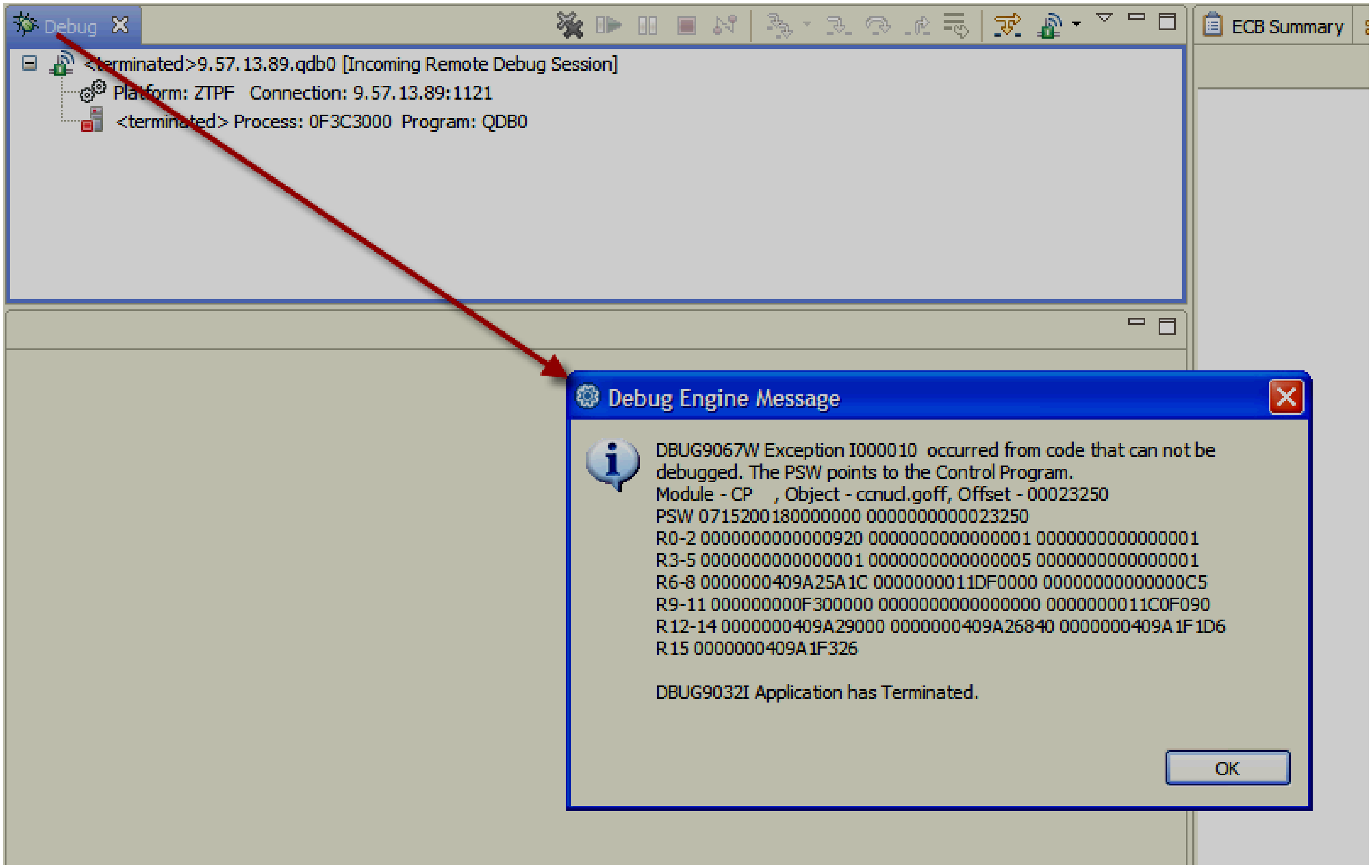
Debugging heap corruption

- CTL-75 dumps occur in the control program and as a result, you cannot run the debugger to the dumping location or use register by system error for these dumps.
- However, if your heap corruption is writing past the fence (a typical case) you can use the heapcheck system feature in conjunction with the debugger to quickly locate the problem code.
- Heapcheck mode causes every malloc to use at least one 4 K frame, the malloc area with the fence is located at the end of the 4 K frame, and the next 4 K frame is invalidated.
- When the application writes past the fence in corrupting the malloc buffer, the application will start to write over the invalid frame and an OPR-4 will occur. The application must write beyond the fence because overwriting the fence is not enough to cause the OPR-4.
- As a result, you can debug the application, clear the breakpoints, and run to the OPR-4. Or you can register the OPR-4 in the system error registration.

Debugging infinite loops

- CTL-10 dumps occur in the control program and as a result the debugger cannot stop the application at the location of the error.
- The debugger attempts to do infinite loop detection.
- However, the application must periodically stop in order for the debugger to perform its detection. This is because the debugger attempts to allow the application to run as fast as possible to provide the optimal debugging experience. As a result, the infinite loop detection cannot occur without you setting breakpoints or stepping of some sort.
- The debugger attempts to make you aware of dumps that occur when the application dumps.

Debugging infinite loops



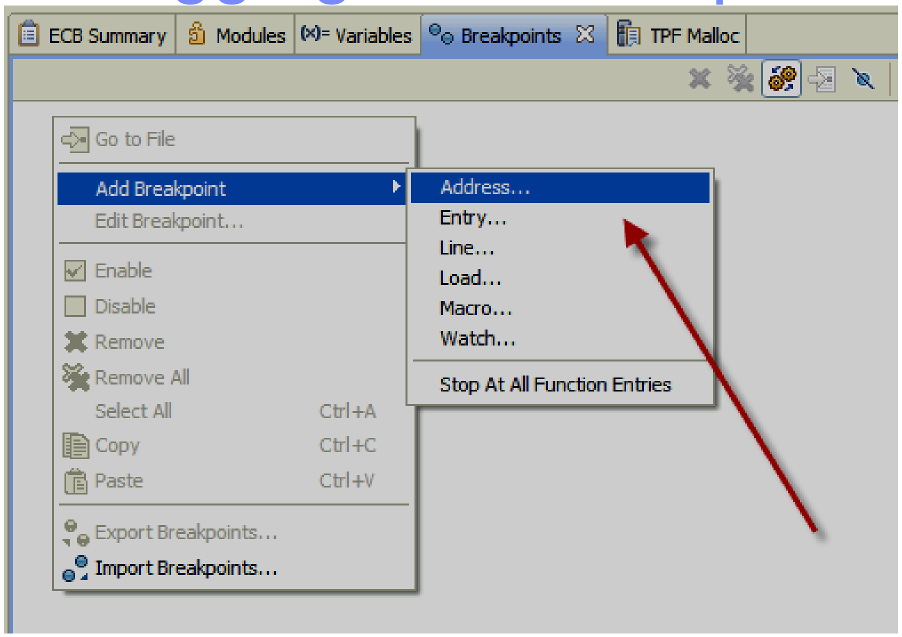
Debugging infinite loops

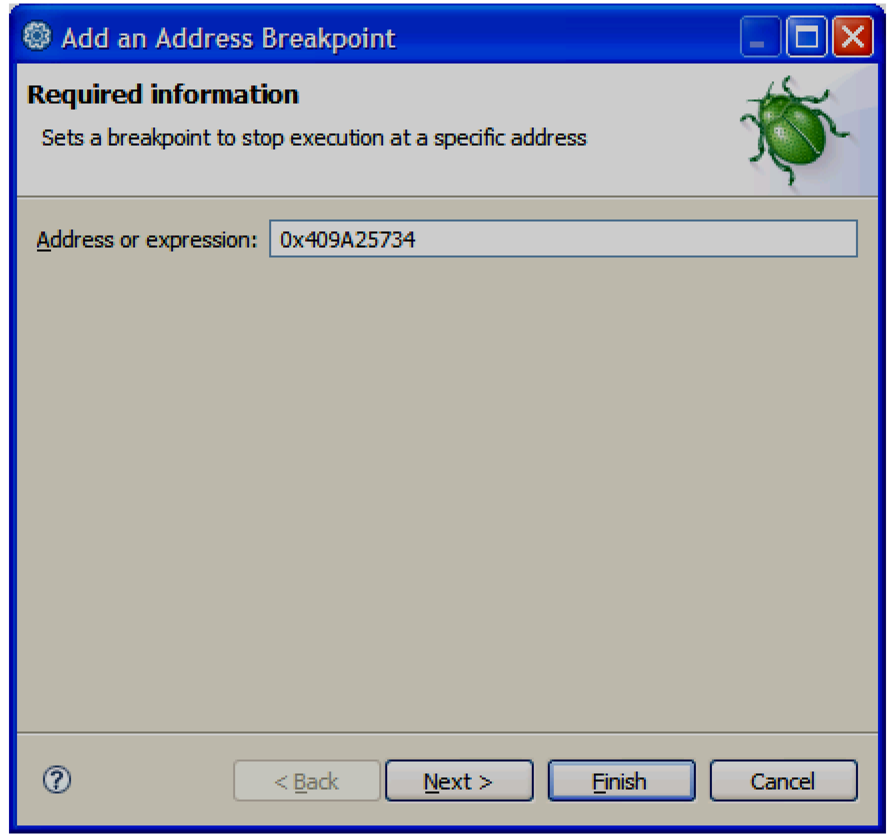
- Use ZDMAP to determine as low of an address as possible and as high of an address as possible. Doing a ZDMAP a-XXXX where XXXX is the address in R15 may be a good way to narrow in on a module to create breakpoints around. Notice that the value in R15 in the figure above falls into the range of QDB0 in the figure below.

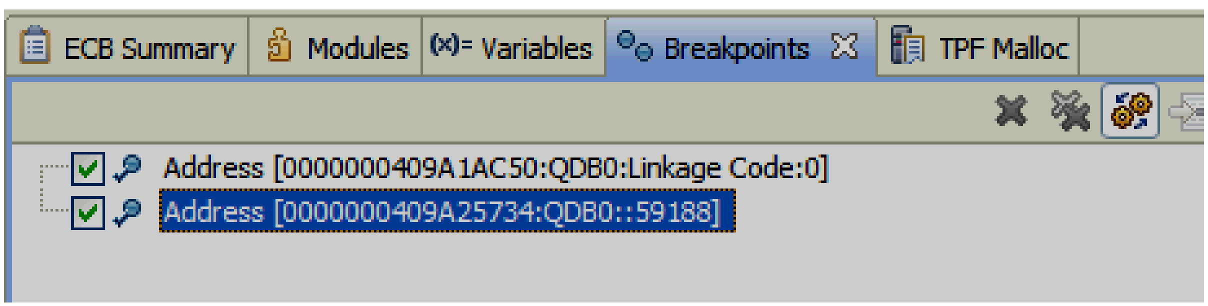
```
AAES0008I 00 ==> zdmap qdb0
CSMP0097I 11.54.28 CPU-B SS-BSS   SSU-HPN   IS-01
DMAP0003I 11.54.28 LINK MAP DATA DISPLAY
                   QDB0JW ACTIVE IN LOADSET BBBBB IN SUBSYSTEM BSS
                   PROGRAM ADDRESS                  000000409A1AC50
_
                   PROGRAM SIZE                             0000AAE4
```

- This gives us an address range of: 409A1AC50 to 409A1AC50 + AAE4 (409A25734). Now start the debugger on your application and use these two addresses to create address breakpoints.

Debugging infinite loops



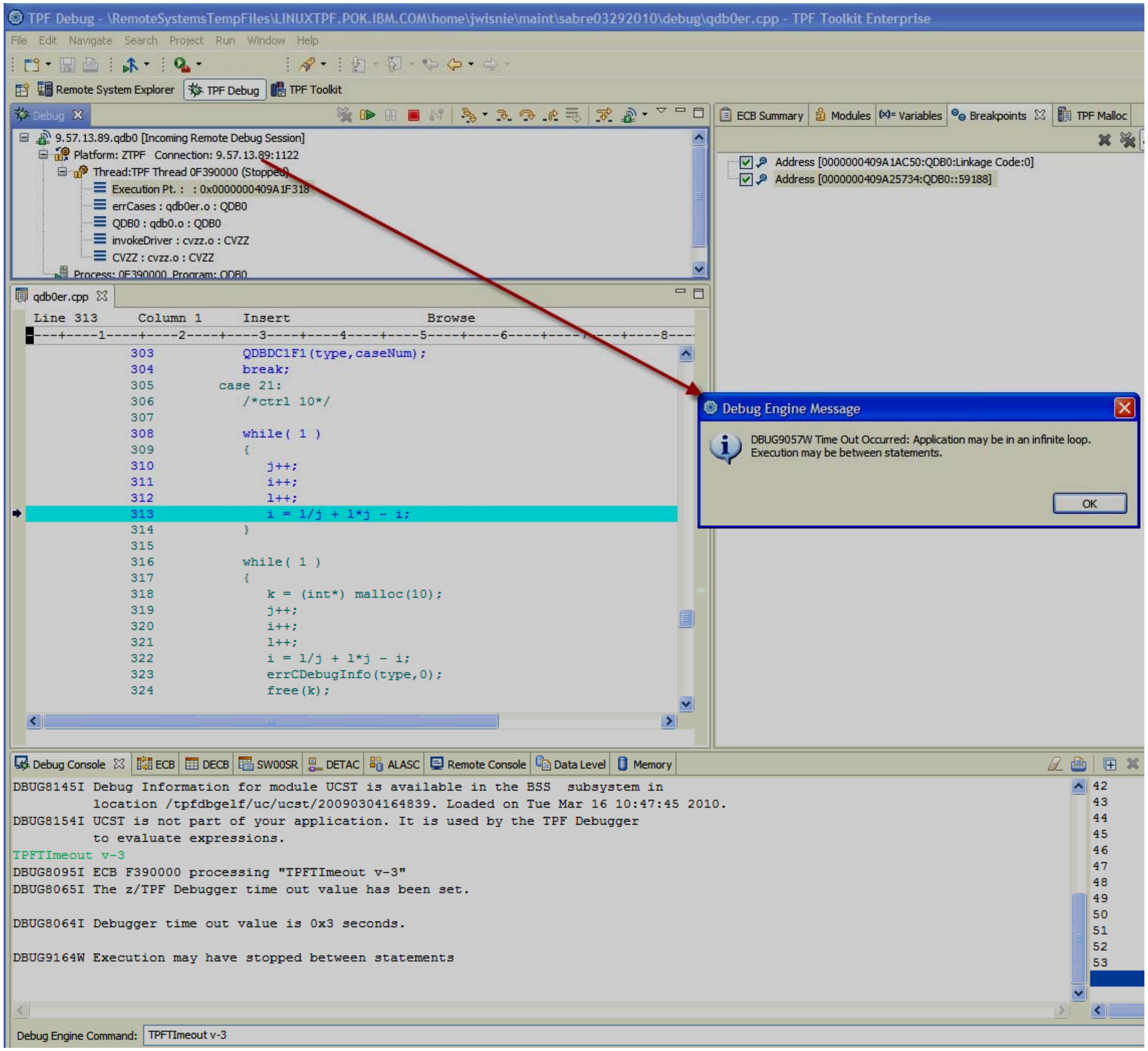




Debugging infinite loops

- Infinite loop detection is controlled by a time out that you can set. The default setting is 30 seconds. You can use the TPFTimeout debug console command to shorten the time you will need to wait.
- Next push the resume button and wait the specified number of seconds. A pop up will appear indicating that a possible CTL-10 has been found as shown on the next slide.
- The debugger will show you the current stopping location for you to investigate. You can continue to debug as normal or press the resume button to run to the next possible infinite loop detection point.

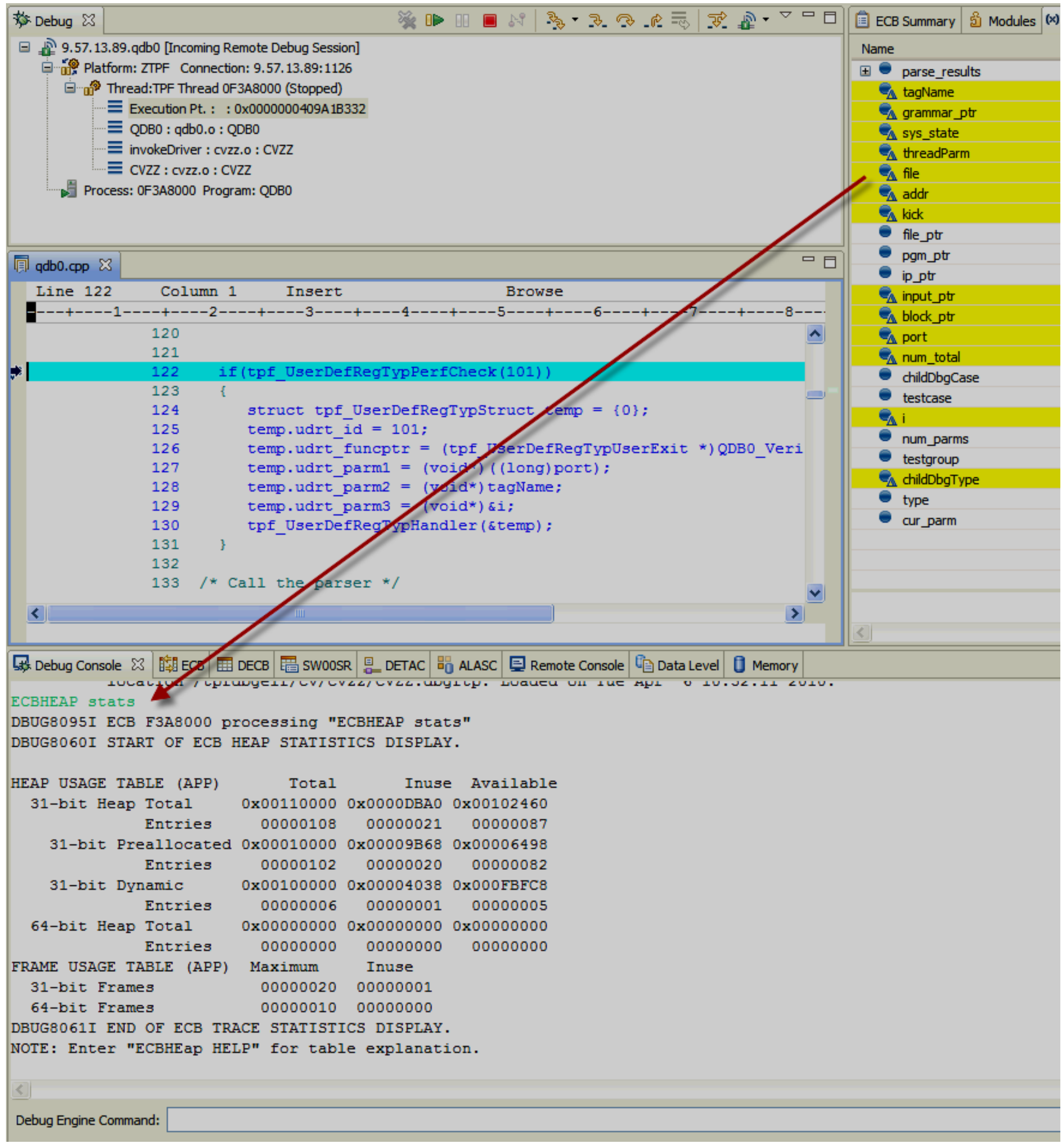
Debugging infinite loops



Debugging memory leaks

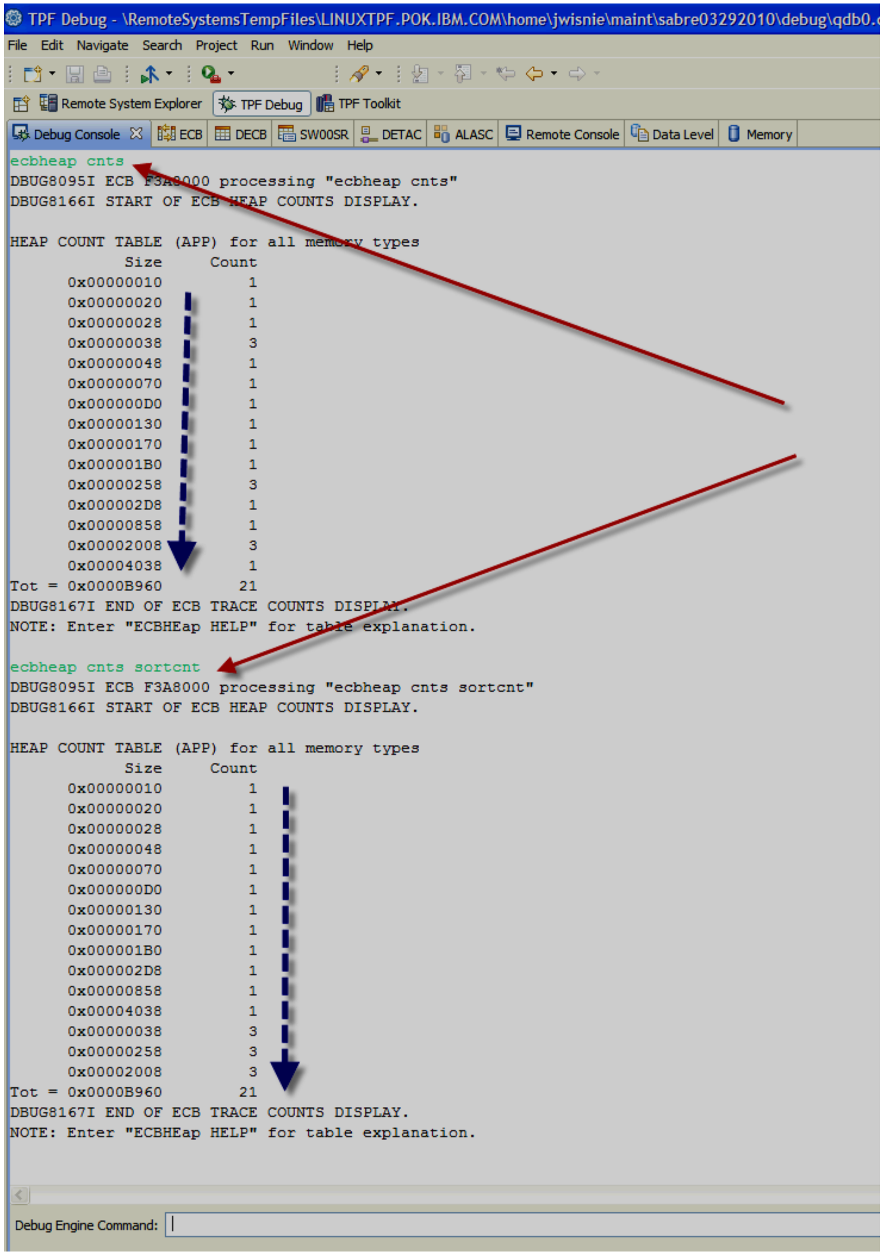
- The z/TPF debugger provides a few features to help identify memory leaks in the application. However they do require that you do some investigating because the debugger cannot determine when a malloc block is no longer used.
- The ECBHEAP debug console command allows you to gather information regarding the use of heap by the application.
- The ECBHEAP STATS debug console command shows how much memory is in use and what types of memory is in use. In the slide that follows, notice that no 64 bit memory is in use.

Debugging memory leaks



Debugging memory leaks

- The ECBHEAP CNTS [sortcnt] debug console command provides the counts of all malloc entries based on size. It can sort based on size or based on the number of malloc entries of a given size.



Debugging memory leaks

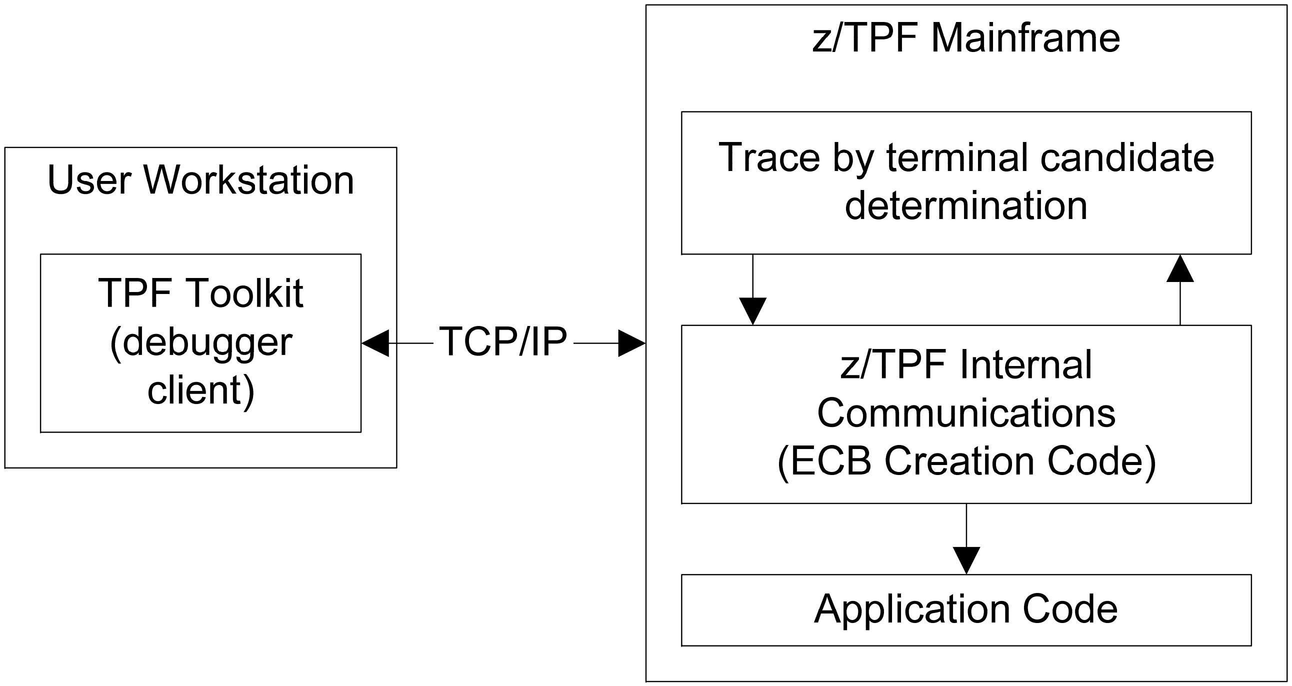
- One way to use this feature is to step over a function, perform some action, and so on and then look at the ECBHEap counts to see what has changed. Make note of what memory sizes are not getting freed. Use the TPF malloc view to choose a given size entry and use the selected block pane to know what code is allocating malloc of that size.
- Another thing to look at is which part of the application is using the largest blocks of memory. Use the malloc view to examine the malloc blocks further (for example sort the malloc view data by size and Look at largest blocks)

Custom communication packages

- Topics
  - Using tpf_flag_for_debug
  - Using CDBX_DebuggerTBTRegistrationTerminalUserExit
  - Using tpf_flag_for_debug and CDBX_DebuggerTBTRegistrationTerminalUserExit together
  - User defined registration: The ultimate solution

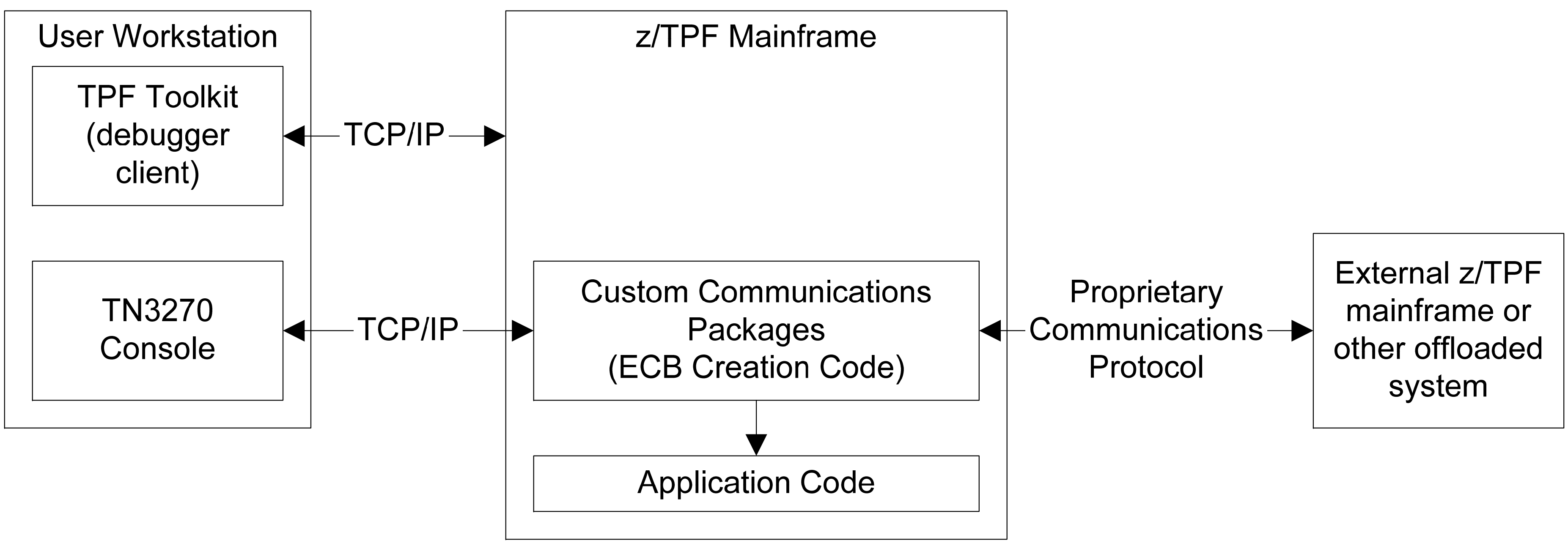
Custom communication packages: Intro

- When a user registers by LNIATA, IP address, or LU, TPF marks ECBs as candidates for trace by terminal debugging.
- If those candidate ECBs enter the registered program, function or etc, a debugger session is started.



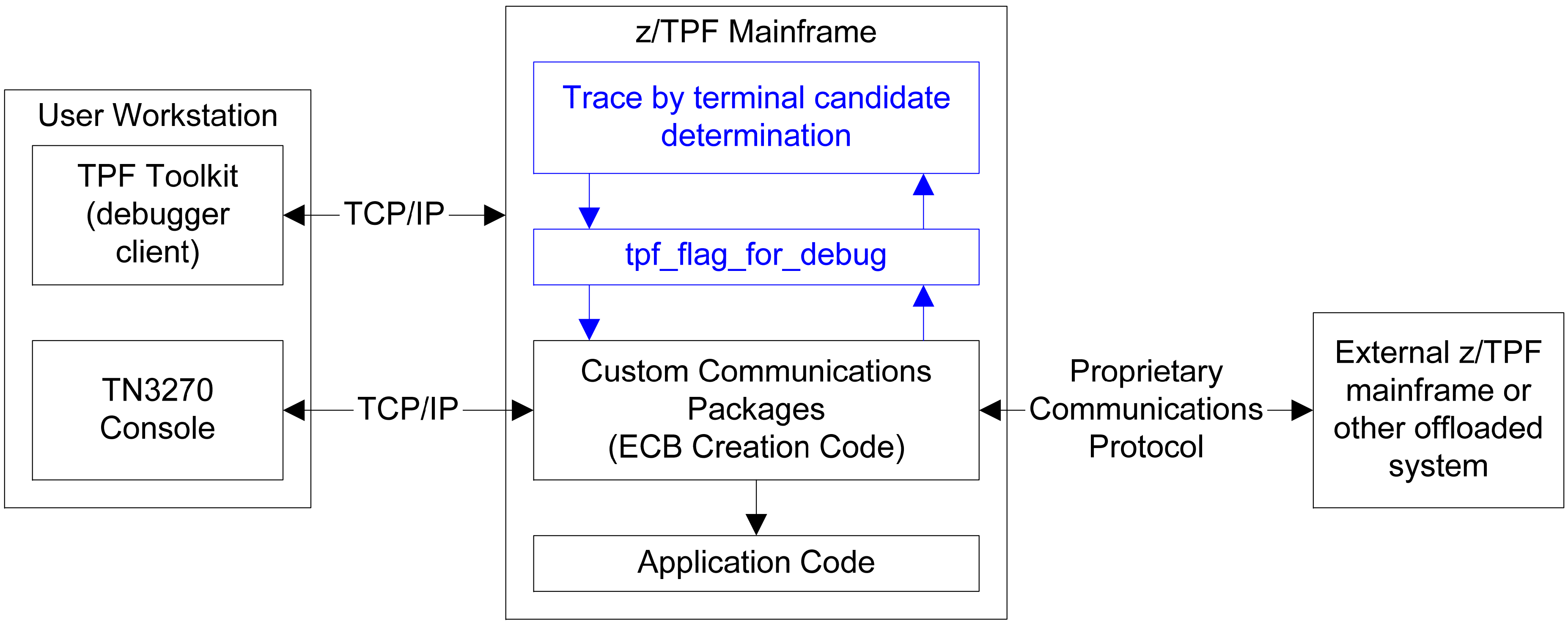
Custom communication packages: Intro

- If you implemented a custom communication package (TN3270, inter-processor communications, etc), it is possible that the ECBs in your system will not be marked as candidates for trace by terminal debugging.



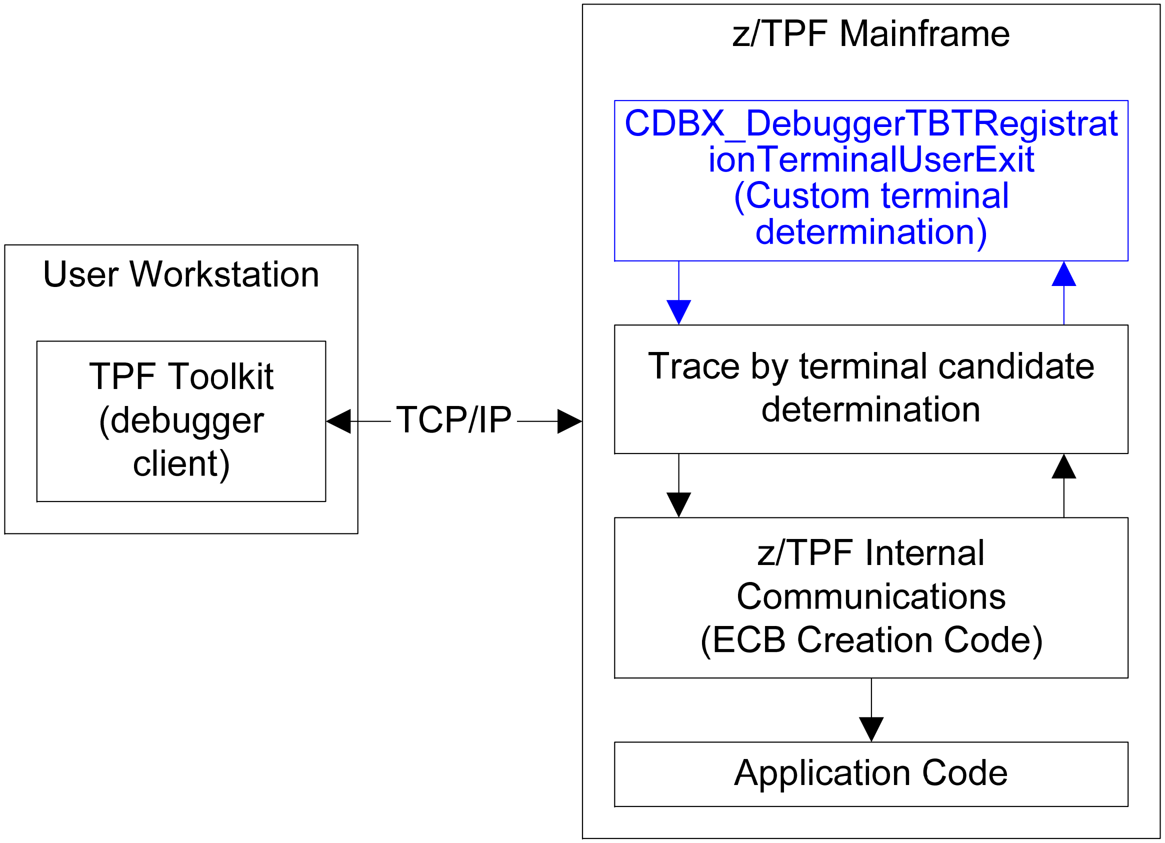
Using tpf_flag_for_debug

- tpf_flag_for_debug is a system service that allows your custom communication package to mark ECBs as candidates for trace by terminal debugging.



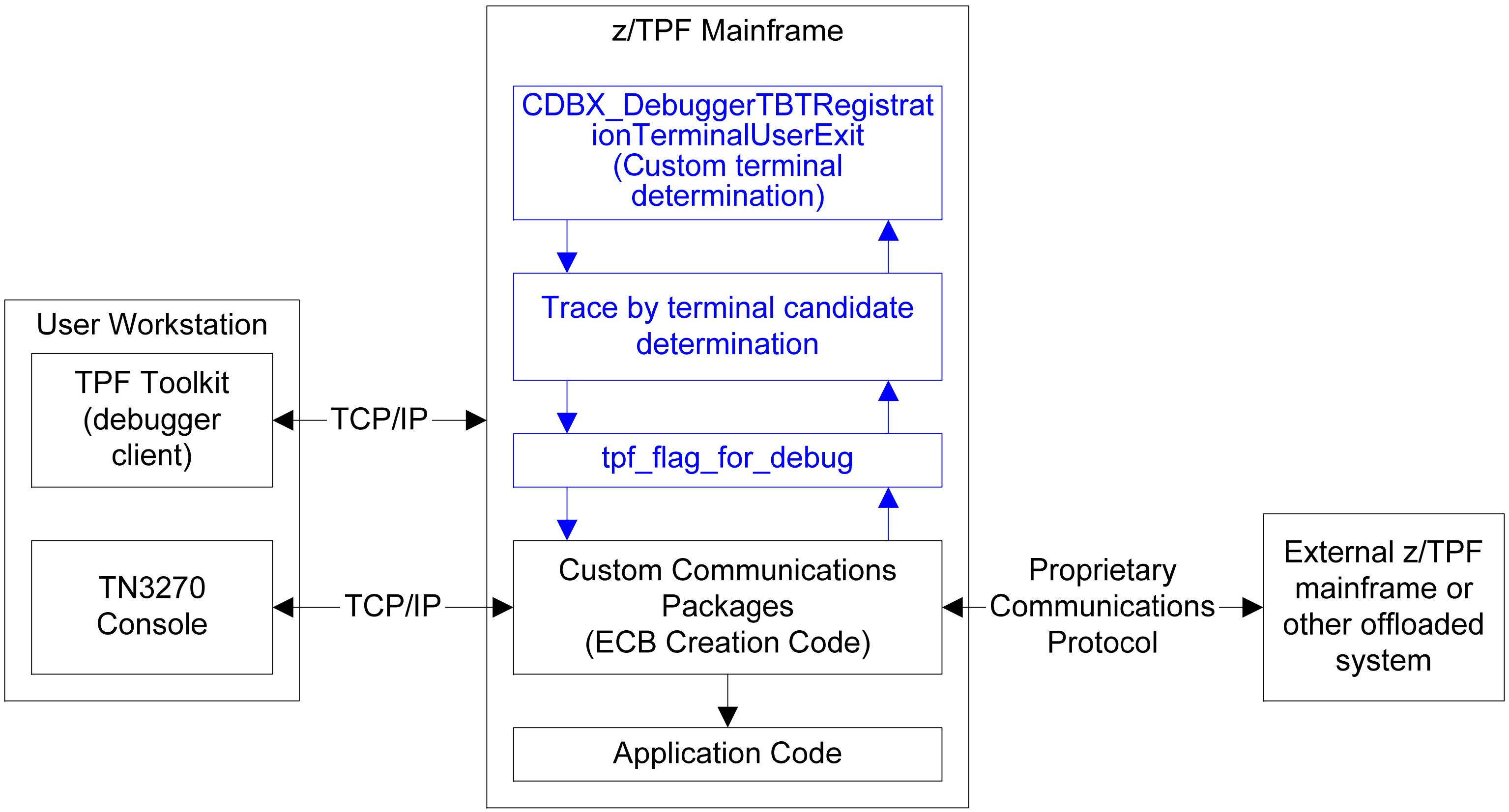
Using CDBX_DebuggerTBTRegistrationTerminalUserExit

- This user exit is in the routine that marks ECBs as candidates for trace by terminal debugging. It allows you to inspect the ECB and provide a custom terminal to the debugger.
- For example, if you have implemented TN3270 support, this user exit could return an LNIATA for an ECB created by your package such that the debugger user can register for trace by terminal by LNIATA.



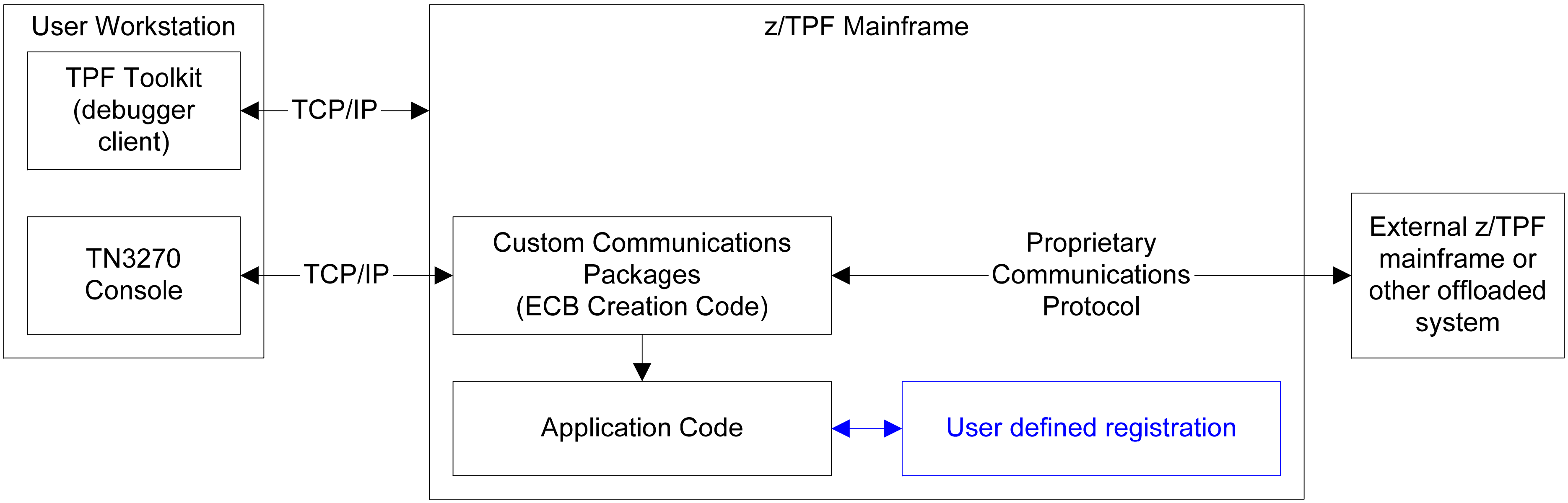
Using tpf_flag_for_debug and CDBX_DebuggerTBTRegistrationTerminalUserExit together

- Using these two features together allows your custom communications package to call to mark the ECB as a candidate for debugging and allows you to specify the terminal to use.



User defined registration: The ultimate solution

- This feature allows you to start the debugger virtually anywhere in your application under the conditions you define.
- For example, the user could register: their ID, a transaction type, a transaction identifier, and etc to debug only the ECB they need to debug.
- See the developerworks article, the redbook or the debugger user's guide for an example implementation.



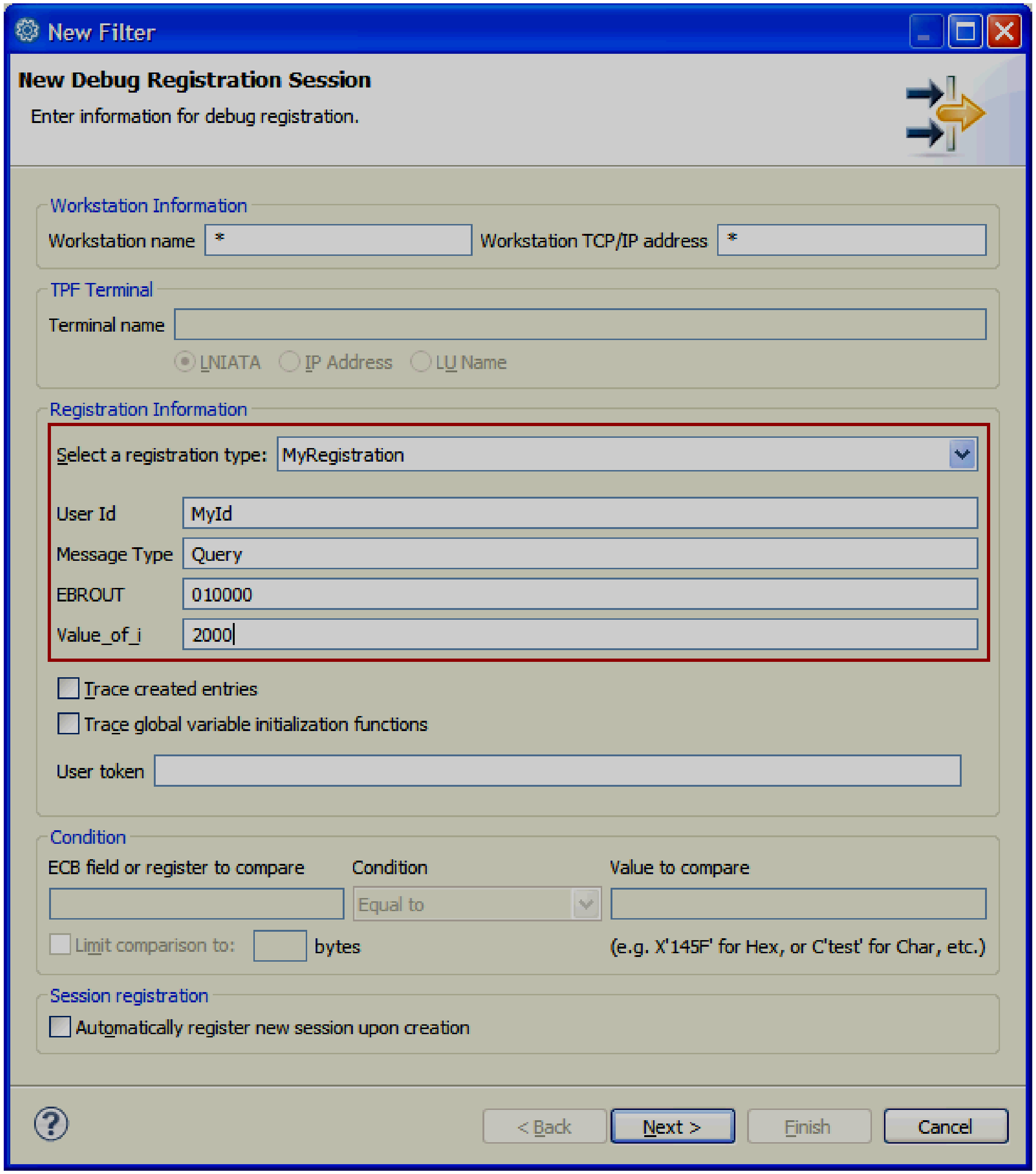
User defined registration: The ultimate solution

- **Define conditions to test**
  - Modify the file <TPF Toolkit install dir>\Config\TPFSHARE\Debug Registration\customDebugRegTypes.xml to
    - define the names of the conditions (parameters) to be tested
    - define the name of the registration type
    - define the registration type id
  - Restart the TPF Toolkit

```
<customRegistration>
        <id>101</id>
        <name>MyRegistration</name>
        <parameter>User Id</parameter>
        <parameter>Message Type</parameter>
        <parameter>EBROUT</parameter>
        <parameter>Value_of_i</parameter>
</customRegistration>
```

User defined registration: The ultimate solution

- The names of the conditions will be shown to the user with a text box for the user to provide the comparison value.



User defined registration: The ultimate solution

- **Define where to test the conditions:** The next thing that you need to do is to modify your application to call the test program with the conditions in your application to be tested against the comparison values registered by the user. C/C++ and Assembler interfaces are provided.
  - The first line of this block of code uses the performance-sensitive macro tpf_UserDefRegTypPerfCheck to see whether a given user-defined registration type is actively registered on the system. Because the user-defined registration code is contained within a block that is encapsulated by the performance-sensitive macro, this code can be left in your production-level code for test points that can be used in the future.
  - Now define an instance of the tpf_UserDefRegTypStruct structure and populate it with the registration type ID, a resolving function (in this example, we'll just use the user exit provided), and the comparison values to be passed as parameters.
  - Lastly, you call tpf_UserDefRegTypHandler.

User defined registration: The ultimate solution

- Build and load your application.

```

num_parms = IPRSE_parse ( input_ptr,grammar_ptr,&parse_results,
                          IPRSE_ALLOC | IPRSE_PRINT , "DBUG");

if(tpf_UserDefRegTypPerfCheck(101))
{
   struct tpf_UserDefRegTypStruct temp = {0};
   temp.udrt_id = 101;
   temp.udrt_funcptr = (tpf_UserDefRegTypUserExit *)cdbxud_user_exit;
   temp.udrt_parm2 = (void*)reqType;
   temp.udrt_parm4 = (void*)&i;
   tpf_UserDefRegTypHandler(&temp);
}

/* display help manual if parser error                    */
if (num_parms < 1)
```

User defined registration: The ultimate solution

- **Define how to test the conditions**
  - Implement the code that performs the test of the conditions, for example in the user exit code cdbxud.c. It can be defined in assembler, in other code locations and etc.
  - The contents of the UDRT_ptr (the state of the executing ECB) are compared to the contents of tbu_entry (the comparison values registered by the user as stored in the debugger registration entries).
  - Notice that you can compare the registered variable against the values in the ECB, system or etc.
  - The parameters are passed as void pointers so that your code must know how to interpret the comparison values, such as using functions like atoi, sscanf, and etc.
  - Set rc to true to tell the debugger to start.

User defined registration: The ultimate solution

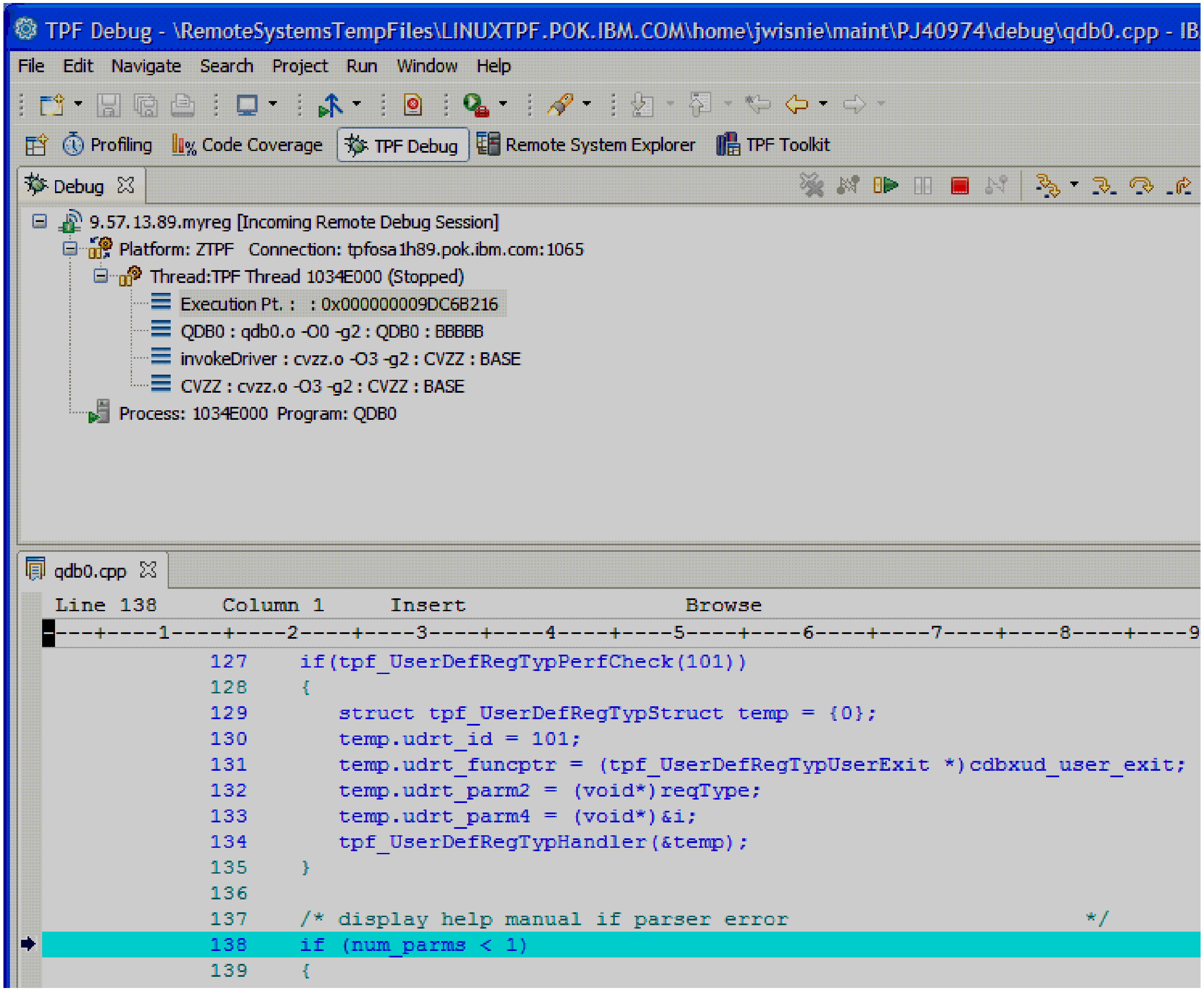
- Build and load your code that tests the conditions.

```
unsigned int cdbxud_user_exit(struct tpf_UserDefRegTypStruct* UDRT_ptr,
                              struct itbpentry* tbu_entry)
{
   unsigned rc = FALSE;   /* set default return to false    */

   if((UDRT_ptr == NULL) || (tbu_entry ==NULL))
     {
     return rc;
     }

   switch(UDRT_ptr->udrt_id)
     {
     case 101:
       {
          if(0 != strncmp((char*)&ecbptr()->ebw000,
                          (char*)tbu_entry->itbp_udrt_parmValue[0],8))
             break;
          if(0 != strncmp((char*)UDRT_ptr->udrt_parm2,
                          (char*)tbu_entry->itbp_udrt_parmValue[1],8))
             break;
          unsigned int lniata = 0;
          if(1 != sscanf((char*)tbu_entry->itbp_udrt_parmValue[2],"%x",&lniata))
             break;
          if(ecbptr()->ebrout != lniata)
             break;
          if(*((int *)UDRT_ptr->udrt_parm4) !=
             atoi((char*)tbu_entry->itbp_udrt_parmValue[3]))
             break;
          rc = TRUE;
          break;
       }
```

User defined registration: The ultimate solution

- **Using user-defined registration**
  - Register your user-defined debugger registration entry as you would for other registration types and then run your application.
  - When the debugger is notified by your condition-testing code (cdbxud.c) that a debugger session should be started, the debugger will stop the application at the next line of code following the code snippet in your application that passed in the state of the application.

User defined registration: The ultimate solution



User defined registration: The ultimate solution

- Example Uses
  - Thousands of ECBs might be started per second in a given program (CRETC, network traffic, etc), and you might need to debug only one specific ECB (for example, the one ECB out of a thousand with 0 on data level 1).
  - Perhaps your system has a proprietary communication package that requires the user to register multiple pieces of information.
  - Maybe you need to debug a particular location in code where a set of conditions occur, such as a single entry point transaction application where a query is performed on a particular account number.

User defined registration: The ultimate solution

- **APAR PJ36059**
- **PUT6**
- **TPF Toolkit Level v3.4.3**

Determining code path

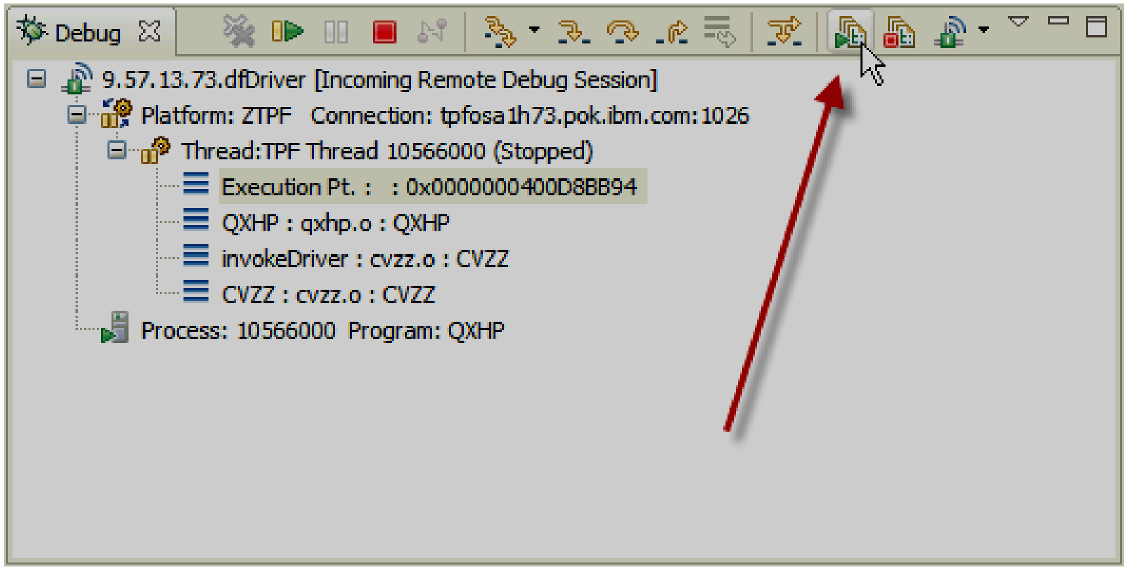
- Topics
  - Using trace log and the code coverage tool together
  - Debugger: stop on all functions and high level breakpoints
  - Debugger: ECB Summary view, animated step into, execute shortcuts
  - Debugger: optimized debugging vs non-optimized debugging

Using trace log and the code coverage tool together

- A trace log is an integrated macro and function trace that provides you parameter values, return values, macro call details and the path through the application code at a high level.
- The code coverage tool allows you to see what source lines, macros and instructions your application has executed. The code coverage tool gives you lower level detail allowing you to infer code path.
- Using trace log and the code coverage tool together can help you better understand the code path of your application.

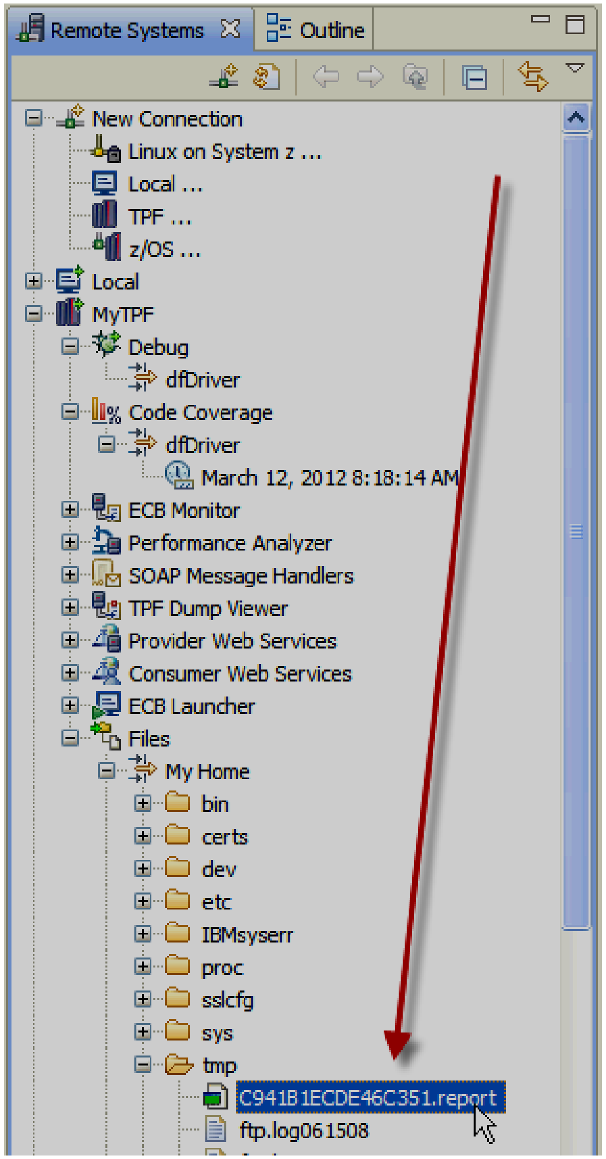
Using trace log and the code coverage tool together

1. Register and start code coverage for your application.
2. Register the debugger for the entry point of your application.
3. If necessary, change the number of trace log sessions allowed on your system with ZASER TRLOG-X
4. Start your application, the debugger starts.
5. Turn on trace log

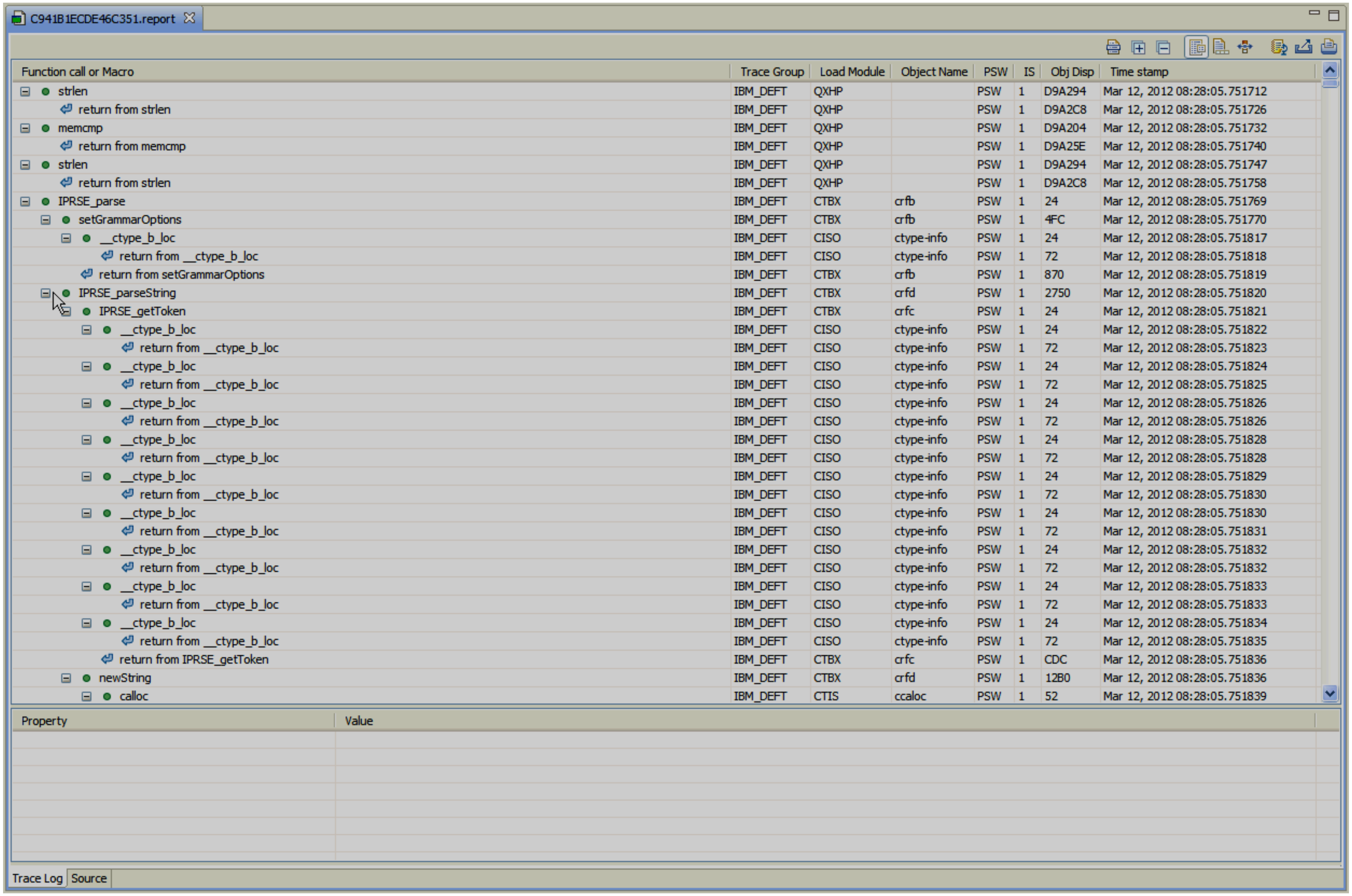


Using trace log and the code coverage tool together

6. Click the resume button to run your application to completion.
7. Double click the report file created in the Files subsystem (GUI FTP interface to the file system on TPF)
8. The report file opens in the editor window showing you the trace log contents. The default view shows you functions and macros called in an indented fashion to show the call stack. See next slide.



Using trace log and the code coverage tool together



C941B1ECDE46C351.report

Function call or Macro	Trace Group	Load Module	Object Name	PSW	IS	Obj Disp	Time stamp
strlen	IBM_DEFT	QXHP		PSW	1	D9A294	Mar 12, 2012 08:28:05.751712
return from strlen	IBM_DEFT	QXHP		PSW	1	D9A2C8	Mar 12, 2012 08:28:05.751726
memcmp	IBM_DEFT	QXHP		PSW	1	D9A204	Mar 12, 2012 08:28:05.751732
return from memcmp	IBM_DEFT	QXHP		PSW	1	D9A25E	Mar 12, 2012 08:28:05.751740
strlen	IBM_DEFT	QXHP		PSW	1	D9A294	Mar 12, 2012 08:28:05.751747
return from strlen	IBM_DEFT	QXHP		PSW	1	D9A2C8	Mar 12, 2012 08:28:05.751758
IPRSE_parse	IBM_DEFT	CTBX	crfb	PSW	1	24	Mar 12, 2012 08:28:05.751769
setGrammarOptions	IBM_DEFT	CTBX	crfb	PSW	1	4FC	Mar 12, 2012 08:28:05.751770
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751817
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751818
return from setGrammarOptions	IBM_DEFT	CTBX	crfb	PSW	1	870	Mar 12, 2012 08:28:05.751819
IPRSE_parseString	IBM_DEFT	CTBX	crfd	PSW	1	2750	Mar 12, 2012 08:28:05.751820
IPRSE_getToken	IBM_DEFT	CTBX	crfc	PSW	1	24	Mar 12, 2012 08:28:05.751821
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751822
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751823
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751824
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751825
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751826
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751826
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751828
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751828
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751829
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751830
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751830
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751831
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751832
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751832
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751833
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751833
__ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	24	Mar 12, 2012 08:28:05.751834
return from __ctype_b_loc	IBM_DEFT	CISO	ctype-info	PSW	1	72	Mar 12, 2012 08:28:05.751835
return from IPRSE_getToken	IBM_DEFT	CTBX	crfc	PSW	1	CDC	Mar 12, 2012 08:28:05.751836
newString	IBM_DEFT	CTBX	crfd	PSW	1	12B0	Mar 12, 2012 08:28:05.751836
calloc	IBM_DEFT	CTIS	ccaloc	PSW	1	52	Mar 12, 2012 08:28:05.751839

Property	Value

Trace Log | Source

Using trace log and the code coverage tool together

9. Examine the function/macro parameters and return values, search or filter the results in an LPEX editor view, press the analyze button to see statistics about what macros were used, memory allocation, segments entered and etc, and generally understand the overall path of execution of your application.
10. Stop and save the code coverage session and run source analysis.
11. Use the code coverage view to navigate to modules, objects, functions and source files of interest. Examine the execution statistics.
12. Examine the source lines or instructions executed. See next slide.

Using trace log and the code coverage tool together

C941B1ECDE46C351.report | qxhp.cpp

```
               }
            free(hotel_parms);
         break;
      case FLIGHT_DB:
         struct qxhf_parms *flight_parms;            // Create Parameter Struct
         flight_parms=(struct qxhf_parms*) malloc(4069);
         if (flight_parms == NULL) { printf("malloc failure\n"); }
         if (num_parms > 0)
          {
            init_flight_parms(flight_parms);
            do
            {
              pr_ptr = pr_ptr->IPRSE_next;           // point to the first "variable/value" paramete
              process_flight_parms(flight_parms, pr_ptr->IPRSE_parameter,
                                   pr_ptr->IPRSE_value);
            } while (--num_parms);
          }
             switch (cur_func) {
                case INIT_FUNC:
                   qxhf_init(sub_func, DFED_PROMPT, "QXFI");                      //D16691f_2
                   break;
                case BUILD_FUNC:
                   qxhf_build(sub_func, DFED_PROMPT);
                   break;
                case IBUILD_FUNC:                                                 //D18438
                   qxhf_ibuild();                                                 //D18438
                   break;                                                         //D18438
                case ADD_FUNC:
                   switch (cur_sub_func) {           // Sub-Function Calls:
```

Source | Properties

Using trace log and the code coverage tool together

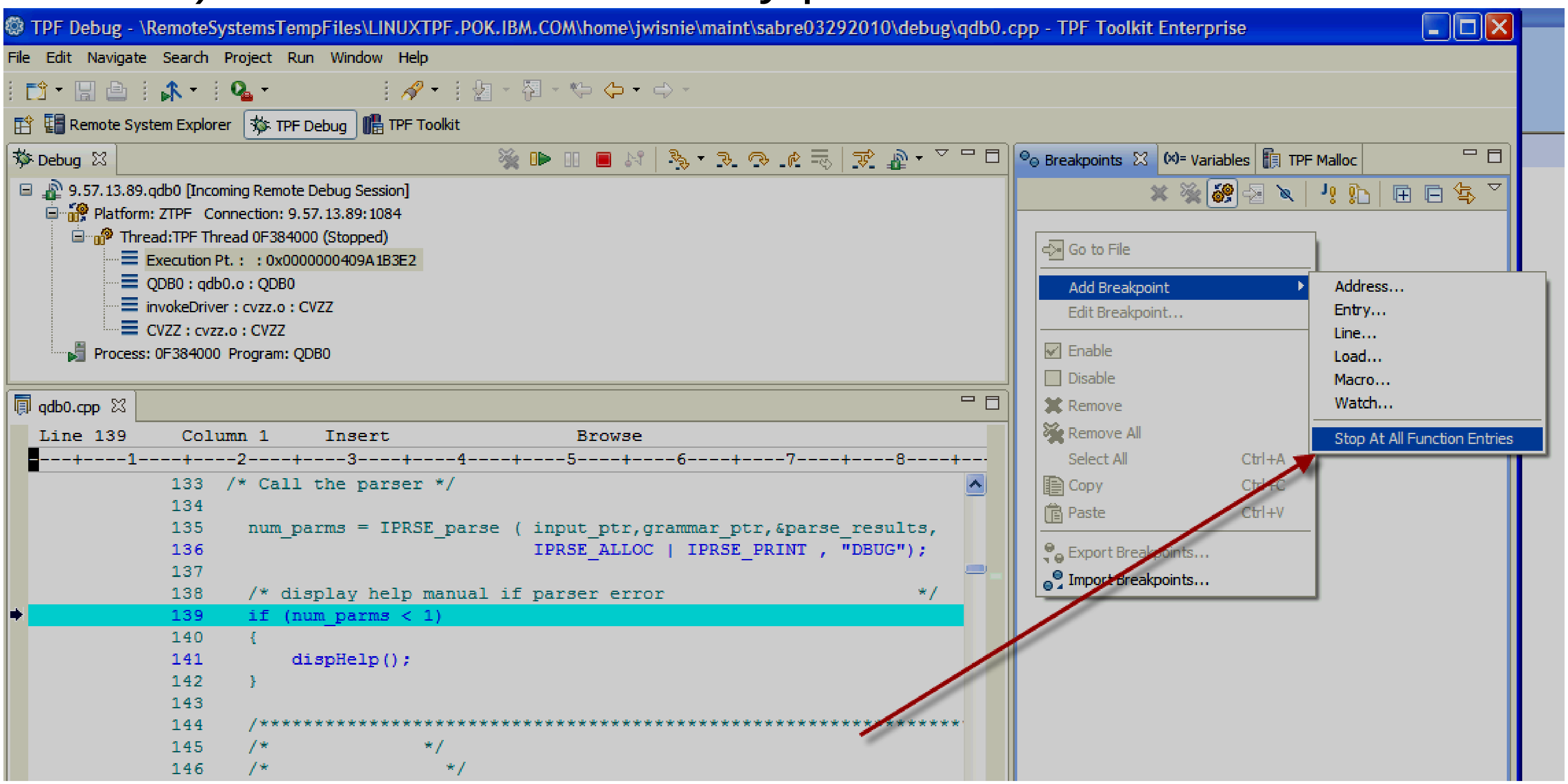
- **Use this methodology to...**
  - learn about the code path of an application you don't know.
  - learn why your code fix did not work properly. For example, was your new code even executed?
  - determine the best place to start a debugger session.
  - understand deviations between two slightly different situations. For example, run both scenarios separately as described above, use the code coverage comparison tool to identify where the paths deviate, and then use trace log to see parameters and return values to understand why the deviation occurred.

Debugger: stop on all functions and high level breakpoints

- You can use debugger features such as stop on all functions and high level breakpoints to understand the execution path of your code.
  1. Register the debugger for the entry point of your application
  2. Set up stop on all functions and/or other high level breakpoints.
  3. Use the resume button to run from location to location to understand the path of execution of your application.

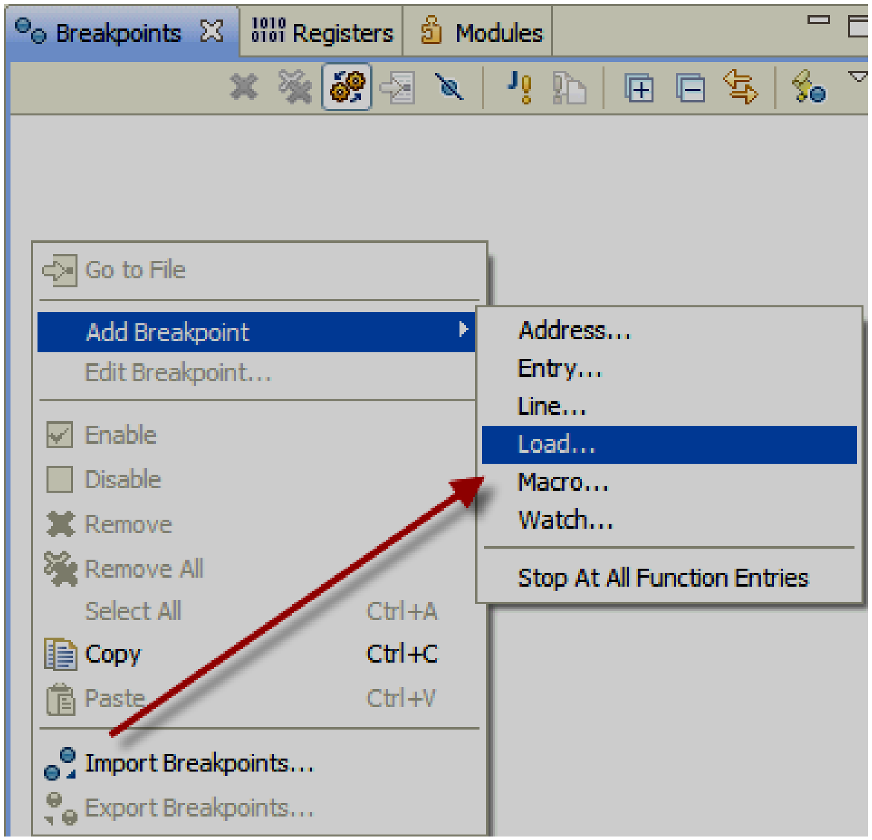
Debugger: stop on all functions, high level breakpoints, etc

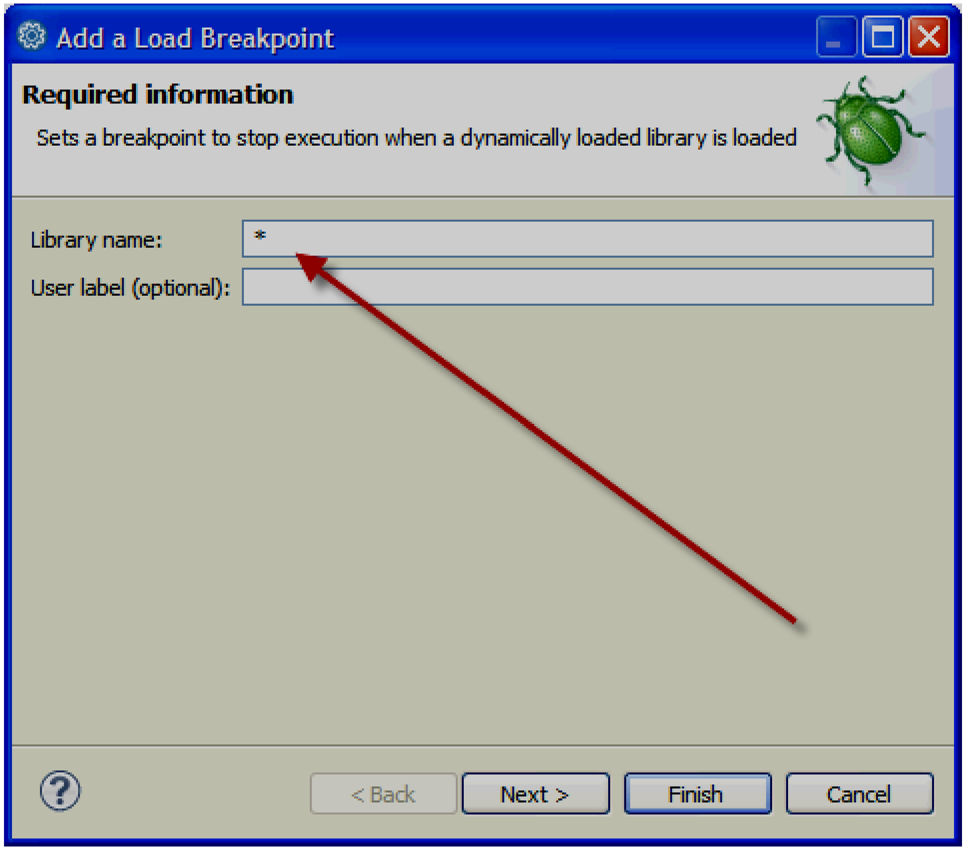
- **Stop on all function entries** behaves as if you set a breakpoint at every C/C++ function entry point (including TMSPC and PRLGC) and BAL external entry points.



Debugger: stop on all functions, high level breakpoints, etc

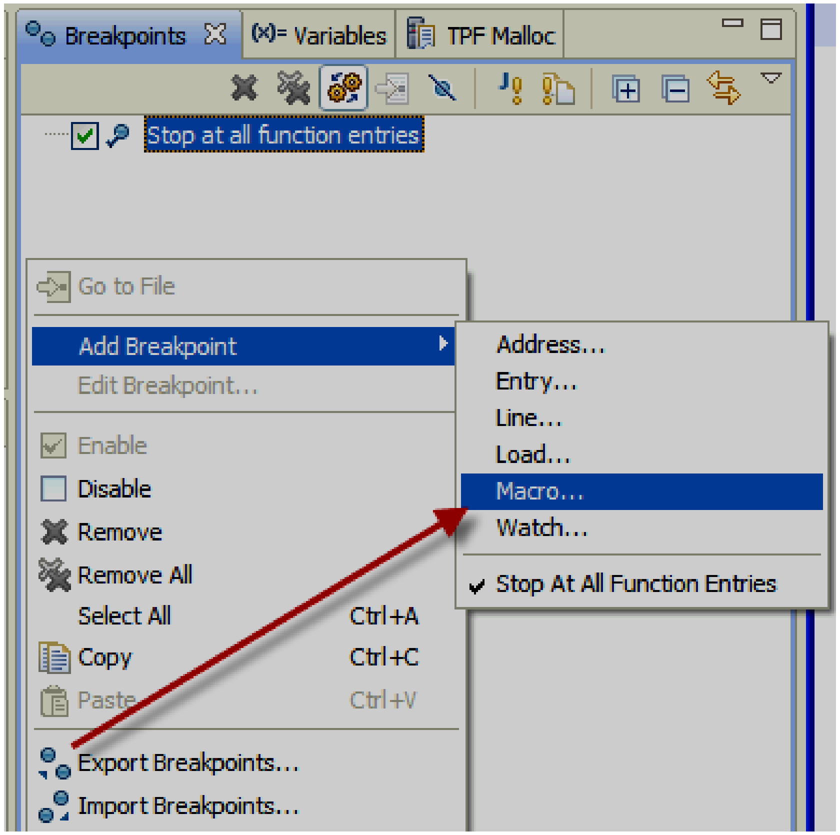
- Load breakpoints stop the execution of your ECB at the entry point of a module the first time it is called. Such as specifying * for the module.

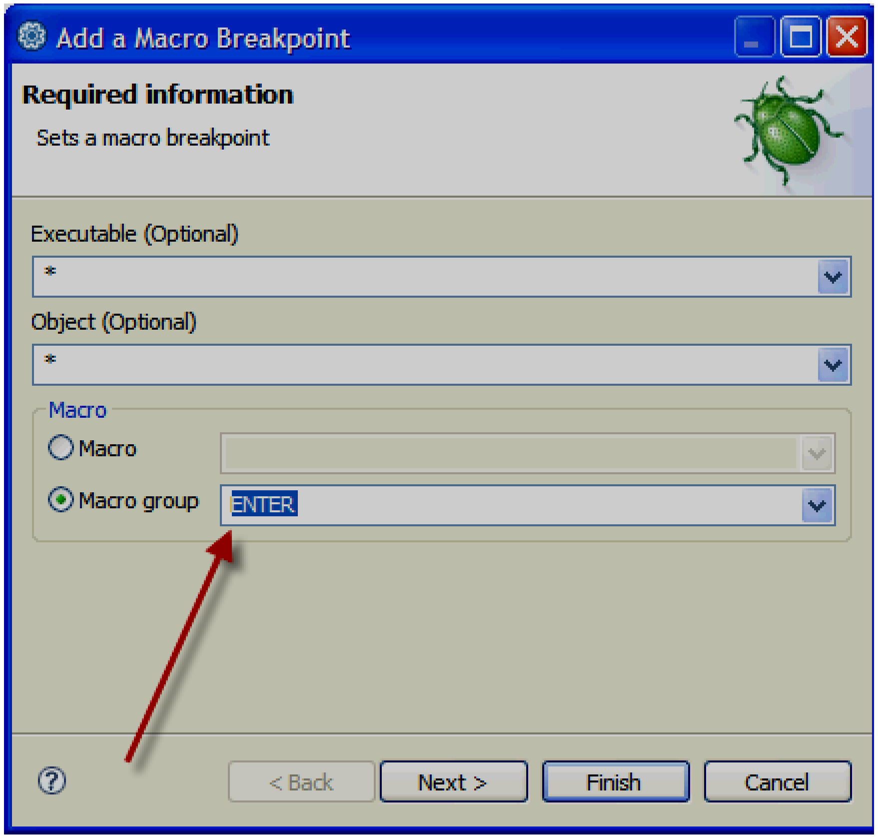




Debugger: stop on all functions, high level breakpoints, etc

- Macro and Macro group breakpoints stop on the execution of macros. Macro groups such as ENTER (all ENTxC and BACKC), DFALL (all TPFDF) and ALLSVC may be particularly useful in this capacity.





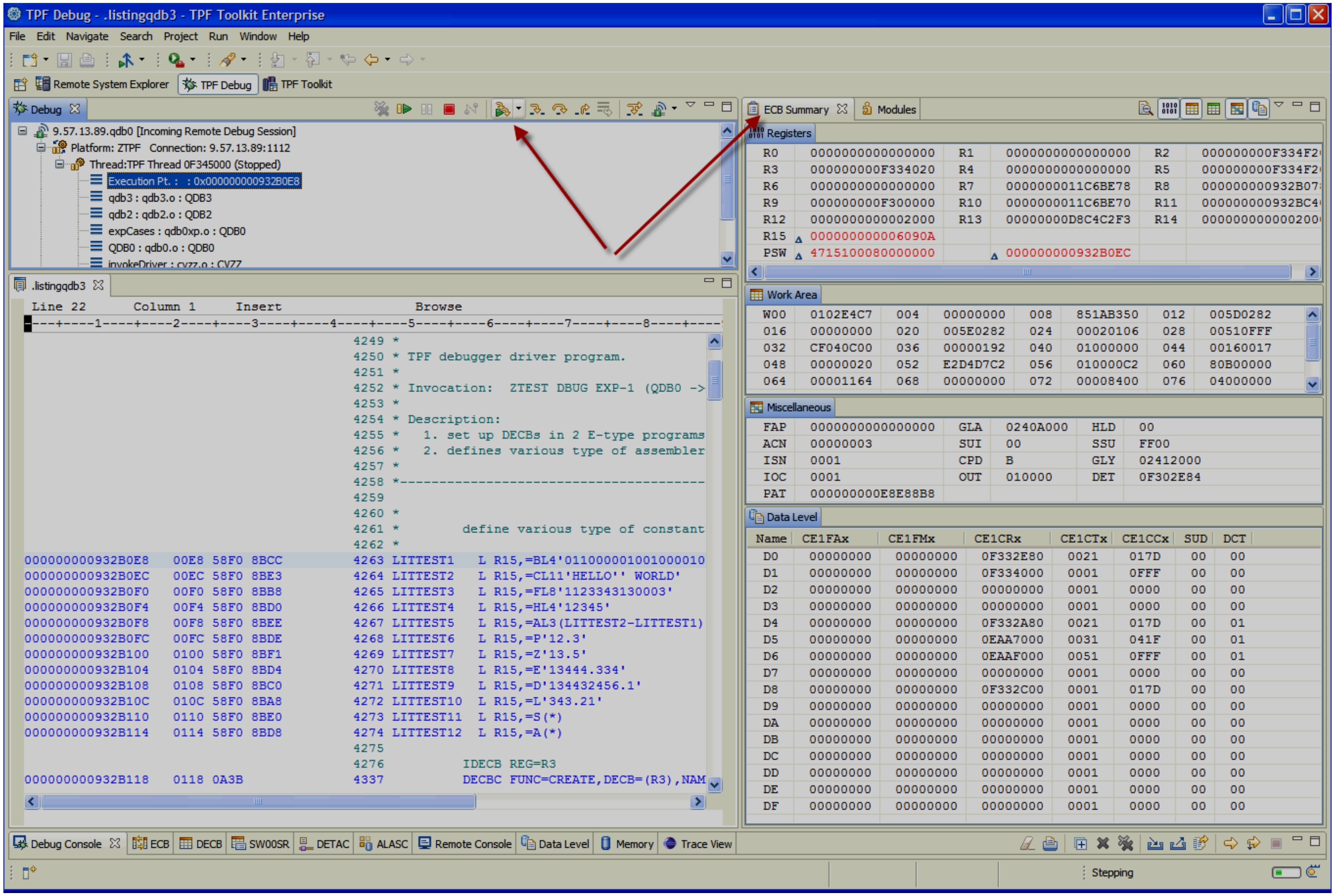
Debugger: ECB Summary view, animated step into, execute shortcuts

- Suppose the code path of your application is less important to the debugging of a problem than the current state of the ECB, such as in debugging a recursive program.

1. Minimize the editor view.
2. Arrange other views to view the state of the application at a glance such as the ECB Summary view, SW00SR view, DECB view, variables view or etc. Ensure the debug view is visible.
3. Use stop on all functions and high level breakpoints previously discussed and watch the state of the application change. Or use execute shortcut keys to execute the application manually. Or use animated step into to walk through your application step by step automatically. Or use step debug to debug a small set of applications.

Debugger: ECB Summary view, animated step into, execute shortcuts

- ECB Summary and the animated step button for automatic stepping.



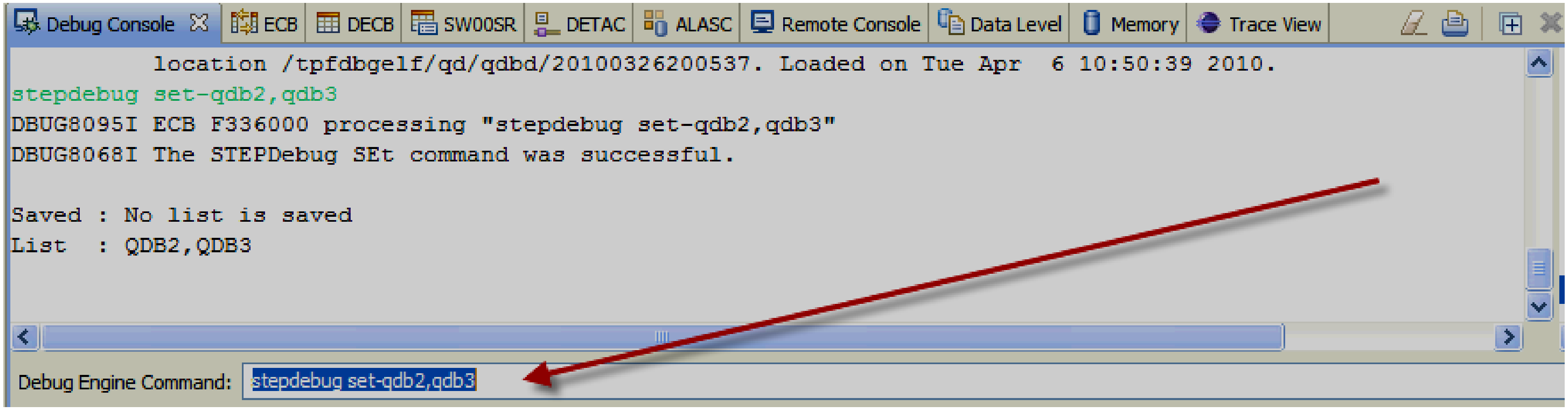
Debugger: ECB Summary view, animated step into, execute shortcuts

- TPF Toolkit provides short cut keys to issue execute actions without clicking buttons which can help you to focus on the state of your application:
  - F5 – Step into
  - F6 – Step over
  - F7 – Step return
  - F8 – Resume

Debugger: ECB Summary view, animated step into, execute shortcuts

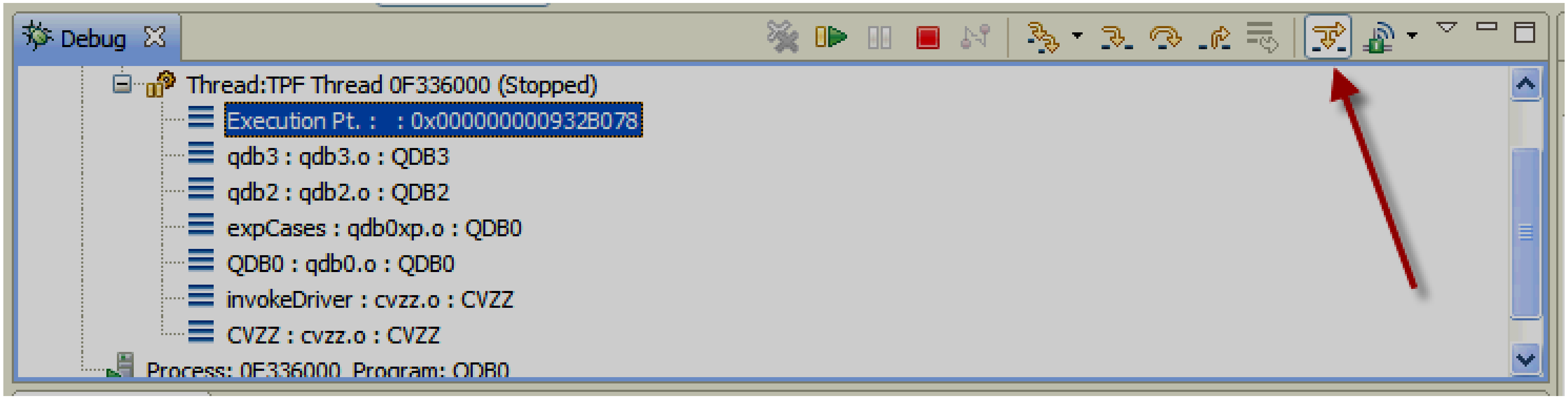
- **step debug** is a feature that allows you to limit your debugging to a list of specified modules.

  1. In the debug console, use the step debug set command to set up the list of programs to limit the application stopping

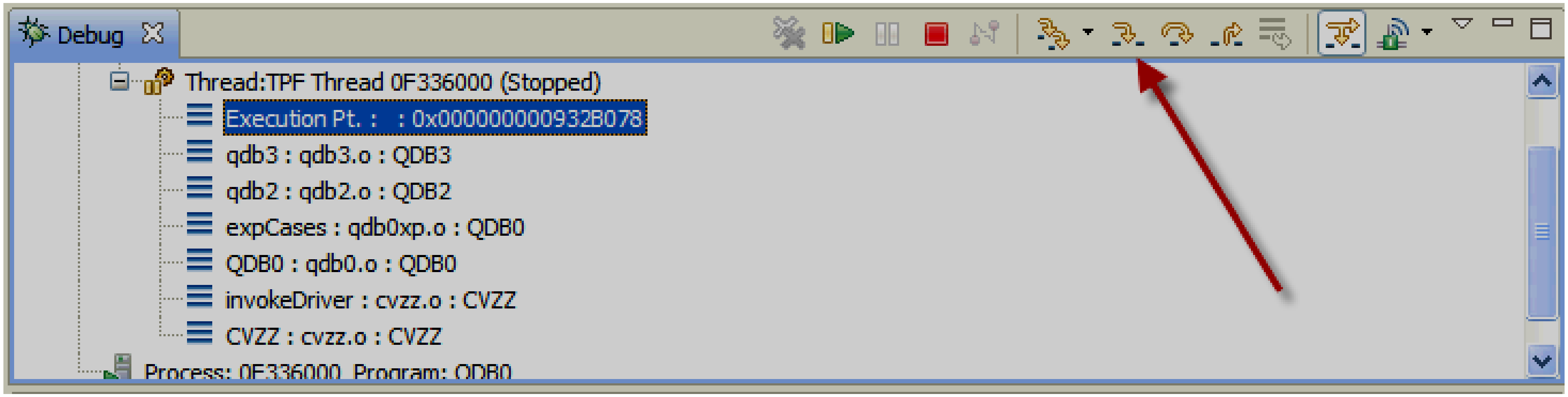


Debugger: ECB Summary view, animated step into, execute shortcuts

2. Toggle on the step debug (step filters) button on



3. Now use the step into button. It will only stop in the modules listed in the step debug list or stop at any breakpoints that you've set.



Debugger: ECB Summary view, animated step into, execute shortcuts

- IMPORTANT NOTE: make sure you toggle off the step debug (step filters) button when you are finished. The setting of the step debug (step filters) button setting is saved. A pop up box warns you the first time you press the step into button and it is behaving as step debug. Do not ignore this warning! Many users have thought the debugger was broken when they simply forgot to turn this step debug feature off.

Debugger: optimized debugging vs non-optimized debugging

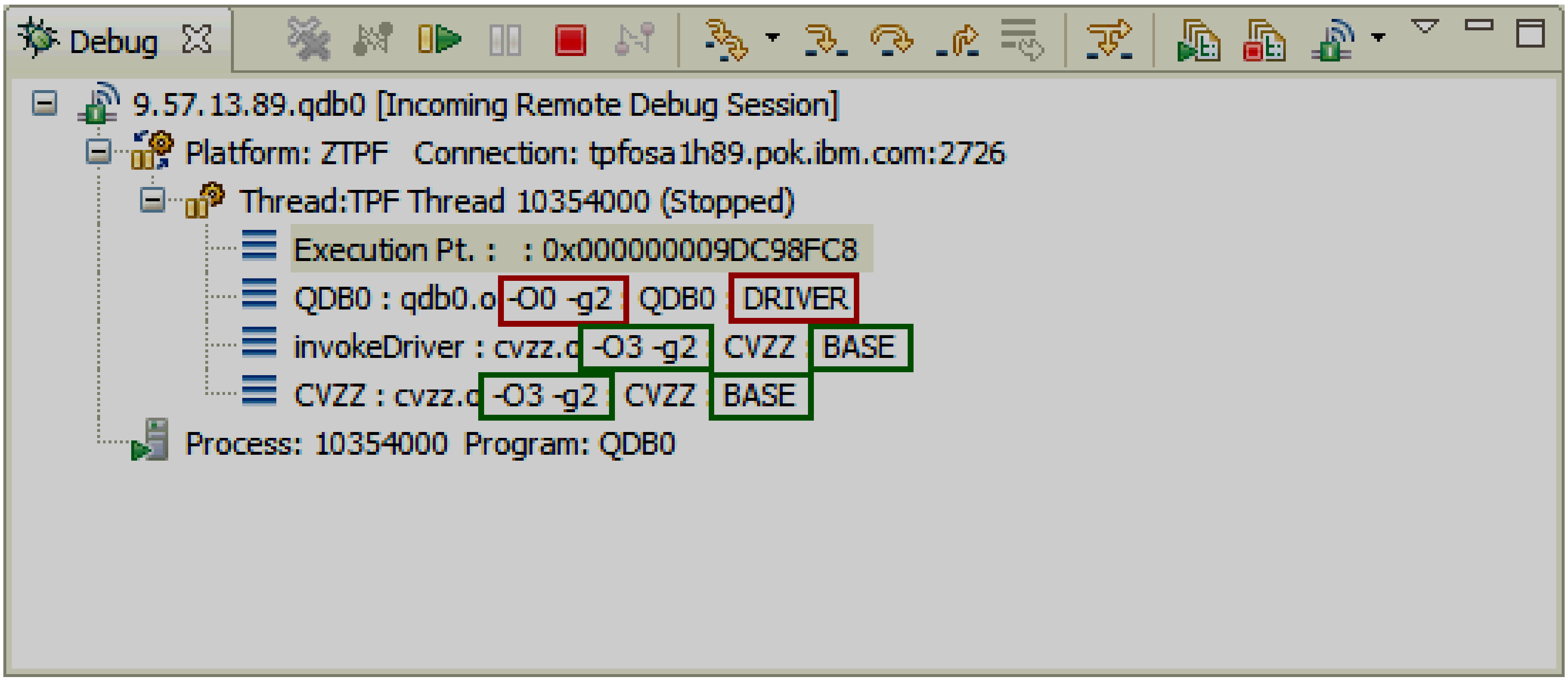
- The features used to determine code path can be used to debug optimized code with or without debug information loaded.
- Once you have a high level view of your application, you can begin to use other debugger features to narrow in on the source of your problem.
- As you narrow in on the area of the problem, rebuild those segments –O0 and load the code with debug information to have an ideal debugging experience with all available variables and linear code execution when stepping.
- Assembler code does not need to be rebuilt, just load debug information. You can also use the Remote Debug Information feature to have the debugger automatically load the debug information for you.

Hints and Tips

- Topics
  - What code am I debugging?
  - Debugger performance

What code am I debugging?

- The stack view can be used to see the loadset for each module on the stack.
- The stack view also shows the compiler options for each object.



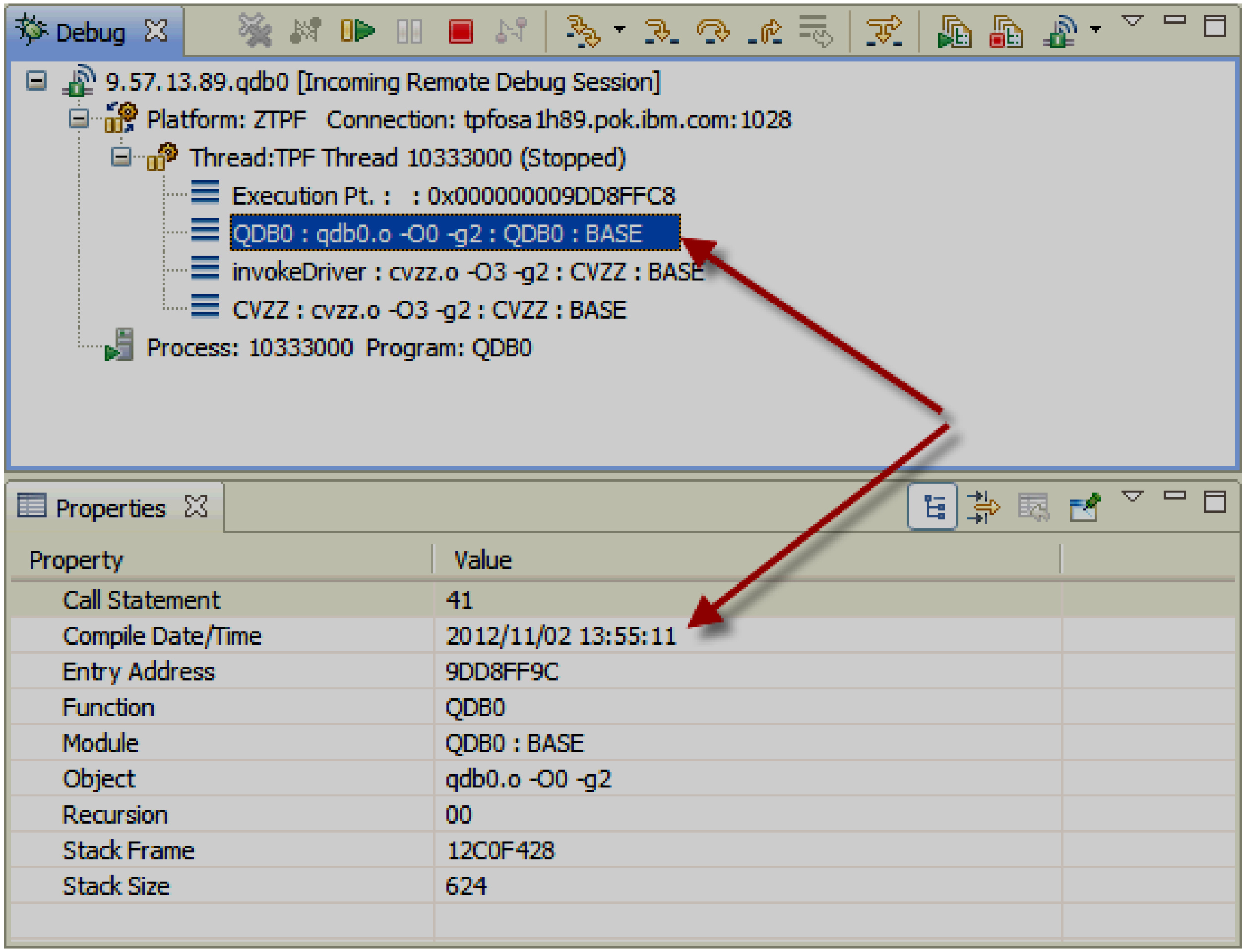
What code am I debugging?

- On the z/TPF system, use the ZDDBG DISPLAY DBGINFO-*prog* command to see what debug information is available on the system for a specific module. The loadset name is provided so you can ensure that your code has debug information.

```
AAES0008I 00 ==> zddbg disp dbginfo-qdb0
CSMP0097I 16.54.03 CPU-B SS-BSS  SSU-HPN  IS-01
CDBS0034I 16.54.03 Debug Info for program QDB0:

VERSION   LOADSET   DBUG    READABLE   DEBUG FILE
-------   -------   -----   --------   ----------------------------------------
_
QDB0      BBBBB     YES     YES        /tpfdbgelf/qd/qdb0/20130319162816.dbgftp
QDB0      BASE      YES     YES        /tpfdbgelf/qd/qdb0/20121102155807

END OF DISPLAY +
```

What code am I debugging?

- Click on a stack frame and look at the properties view to see the compile time and other information.



Debugger performance

- **Set up your Edit Source Lookup to perform well:**
  - Choose **TPF project** (limit the definition of project filters to a small set of files) or **Remote folder** since they are known to perform better.
  - Do not specify root directories (such as /ztpf/).
  - Specify directories as close to source as possible.
  - Specify as few paths as possible.
  - Do not search for multiple matches unless it is absolutely needed. This feature will search all directories and sub-directories on all paths for the matching file name and present a list of all matches to the user.
  - Do not search subfolders. Select the folders where your source exists explicitly.
  - If network performance is a drastic issue, copy source code to a local location on the hard drive, remove all network paths and set the path to this single local location. This will give the best performance in locating files but introduces source file synchronization issues.

Debugger performance

- Open fewer debugger views.
- Give focus to views with static data (breakpoints, monitors, modules, etc) to hide more dynamic views.
- Hide complex or costly views until you need them.
  - Variables view if lots of variables are present.
  - SW00SR view
  - ECB Summary view
  - TPF Malloc view (hide the corruption detection column)
- Limit the use of labor intensive features such as perform heapcheck on stop.
- Turn off hover expression evaluation: from the preference option Window menu->Preferences->Run/Debug->Compiled Debug->Allow hover evaluation checkbox.

Debugger performance

- **Define remote debug information directories well.**
  - Specify as few paths as possible
  - Specify a small timeout value. If FTP must timeout on each system and path and the timeout value is set significantly high, the user may need to wait a long time for the timeout to occur for each system and path (accumulating to a long wait time).
  - If the network is performing poorly, load debug information by way of the loaders instead of relying on the remote debug information feature. Or use debugging techniques that do not require debug information to be loaded.

Starting the debugger effectively

- Topics
  - Understand your application
  - Debugging the right ECB
  - Registration types
  - Tips for registering on shared test systems

Understand your application

- The answer to the following questions determines how you must register the debugger to debug the right ECB.
  - How is my ECB started? Is this ECB started by a CREMC, CRETC, TPF_fork, SWISC CREATE or so on? Is this ECB started by a pthread_create? Or is this ECB started from a communications terminal such as an incoming message on an LNIATA, IP or LU.
  - How does my application behave? Does it create ECBs such as CREMC and so on? Does it create threads? Are there events, LOCKCs, signals, waiting for user input (ZPAGE), waiting for responses from another system, and so on?
  - What part of my application do I need to debug? Does it call global constructors? Is a library malfunctioning or is the mainline path? Is a system function or macro not returning the expected result?
  - Where is the right spot in my application to start debugging such that I'm close to the cause of the problem?
  - Maybe you don't know your application in this level of detail. Do you know a main entry point name, a library used, or so on?

Debugging the right ECB

- The z/TPF debugger is an ECB centric debugger meaning that the ECB is debugged regardless of which code that ECB executes.
- As you think about starting the z/TPF debugger, always be thinking in terms of catching the right ECB that will execute the code you need to debug.
- Use the determining code path functionality to understand the application.

Registration types

- The second key to starting the debugger effectively is knowing what features the debugger has, how to use them, and what the limitations are in order to catch the right ECB.
  - **Register by program name –** 4 char module name (wild cards are supported).
  - **Register by function name –** First execution of a function (wild cards are supported).
  - **Register by SVC –** First execution of a macro.
  - **Register by system error –** start the debugger on an application dump.
  - **Register by CTEST –** start the debugger where ever CTEST is coded in your application.
  - **Register by user defined registration –** start the debugger where ever you want under the conditions you define and register.

Registration types

- Most types of registration provide the following options.
  - **TPF terminal** – acts as a filter in that only the ECBs with a matching terminal will be candidates for debugging. To debug created ECBs (CRETC, CREMC, etc), you must register with LNIATA as *.
  - **Conditional registration** – acts as a filter in that only ECBs that meet the condition (register or ECB contents) will be candidates for debugging.
  - **Trace created entries** – indicates to the debugger that you are interested in debugging ECBs that will be created from the initial parent ECB you debug in independent debugger sessions.
  - **Trace global variable initialization** – allows you to debug global constructors and other initialization functions.
- **Debugging threaded applications** – trace created entries is not necessary. All threads created are immediately stopped. Each thread is controlled independent of all other threads. Key is to click on a thread in the debug view and then perform your desired action.

Tips for registering on shared test systems

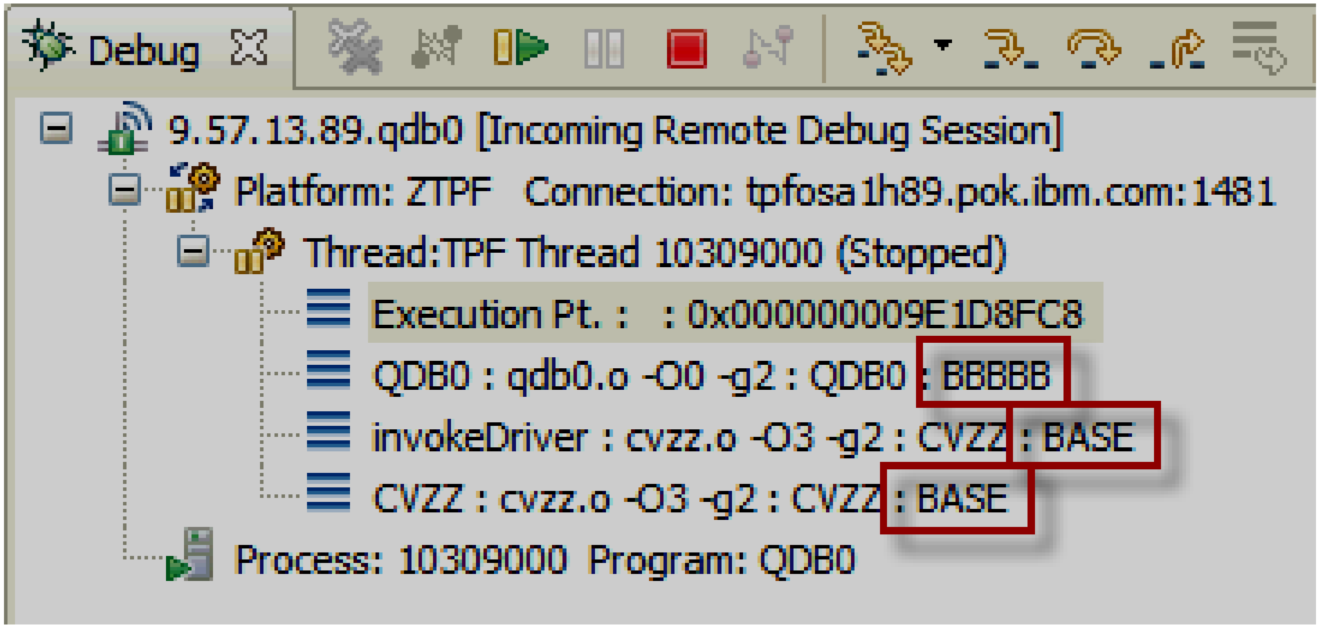
- Registering on a shared test system can present a number of challenges.
- You must define a registration entry such that you do not debug someone else's application.
- You must start your application such that it is not debugged by someone else's registration entry.

Tips for registering on shared test systems

- Use the trace by terminal feature by specifying a TPF Terminal instead of using * for the LNIATA when registering the debugger and have the traffic of each individual started from a different LNIATA.
- Use conditional registration to differentiate ECB from ECB. For example test against a unique value in the ECB such as EBROUT or so on.
- User Defined Registration can work well for these situations. You can define a field to test to be the ID of a user and as part of your application traffic embed the ID of the user in the ECB so that you can test against it.
- Every registration type allows you to pass in a user token. As part of your application traffic embed the ID of the user in the ECB. And every registration trace by program type of registration calls user exit UCCDBTS in cusr.cpy for verification that a debugger session can be started on that ECB. In UCCDBTS you can code a test to compare the user token in the IPROG entry to the ID of the user embedded in the application. Or instead of embedding the ID of the user in the application, you can equate the user token passed in on the registration entry with the IP address or another user unique feature in ECB. You can do a very similar sort of user token comparison for trace by terminal in the tpf_terminal_user_exit in cdbuxt.c.
- Use selective activation. The debugger will work in selectively activated programs without making any accommodations.

Tips for registering on shared test systems

- Another common problem in debugging on a shared test system is debugging code that someone else loaded. While the debugger cannot know if you are debugging the right code, it does show you which loadset your code was loaded in the debug view in each stack frame. If something is not behaving properly, confirm that you are debugging your code.



Q & A

Trademarks

- IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.
- *(Include any special attribution statements as required – see Trademark guidelines on https://w3-03.ibm.com/chq/legal/lis.nsf/lawdoc/5A84050DEC58FE31852576850074BB32?OpenDocument#Developing%20the%20Special%20Non-IBM%20Tr)*

Notes

- Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.
- All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.
- This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.
- All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
- Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
- Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.
- This presentation and the claims outlined in it were reviewed for compliance with US law. Adaptations of these claims for use in other geographies must be reviewed by the local country counsel for compliance with local laws.