KEITHLEY

Application Note Series

Number 2900

Debugging with Forte Workshop

Introduction

This application note discusses how to debug Keithley Interactive Test Tool (KITT) macros and Keithley User Library Tool (KULT) modules using the Forte Workshop Debugger. Two debugging scenarios are outlined.

The Forte Workshop Debugger tool provides many functions for troubleshooting coding problems that may arise when using KITT and the Keithley Test Execution Engine (KTXE). These functions include, but are not limited to, setting break points, single-stepping through code, and analyzing run-time values.

The Debugger is a standard part of the Forte Development Tool set on Keithley's Series S400 and S600 parametric testers.

Coding problems that occur while developing parametric test programs can cause test programs to crash or abort. These test programs can be at the KITT macro or KTXE level.

This application note assumes familiarity with the Keithley Test Environment (KTE) tools discussed here, as well as a working knowledge of the C programming language.

Debugging Scenarios

This application note discusses the following debugging scenarios:

- When running a KITT macro, the macro aborts.
- The KTXE test program crashes.

Scenario 1: The KITT macro aborts

The KITT macro shown in ***Figure 1*** calls a KULT test module (*vth*) to measure the drain current at the threshold voltage.

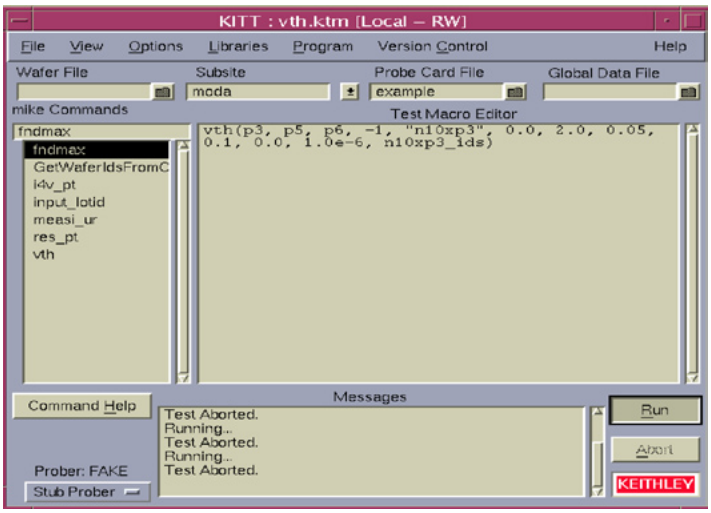

Figure 1

During the execution of this macro, the macro aborts, as shown in the Messages box in ***Figure 1***.

If the macro was executed successfully, the software would normally return a "Done 1" message and the drain current (n10xp3_ids) value would be displayed in the Results window (not shown). However, the message returned is "Test Aborted," indicating a problem somewhere in the macro. This macro only contains one test module (*vth*), but most test macros contain many more modules, which complicates the process of locating the problem.

This macro aborted due to some coding problem within the *vth* routine. One method of locating the source of the problem is to use the C language **printf** statement throughout the *vth* module. The **printf** statement would generate output to the terminal window, making it possible to determine the execution flow of this routine.

This note will discuss how the Debugger can be used to locate the problem. To gain a better understanding of the process involved, we recommend you create a stand-alone C program, called a Practice Task. From the Program menu, select the "Practice Task…" function as illustrated in ***Figure 2***.

You will be asked to enter a UNIX file name. KITT will create this file, which will contain all the C code required and macro contents needed for execution. This file will be compiled and executed using the Workshop Debugger.

When the Debugger is started, it will display the C code from the Practice Task file using a text editor, as shown in ***Figure 3***.

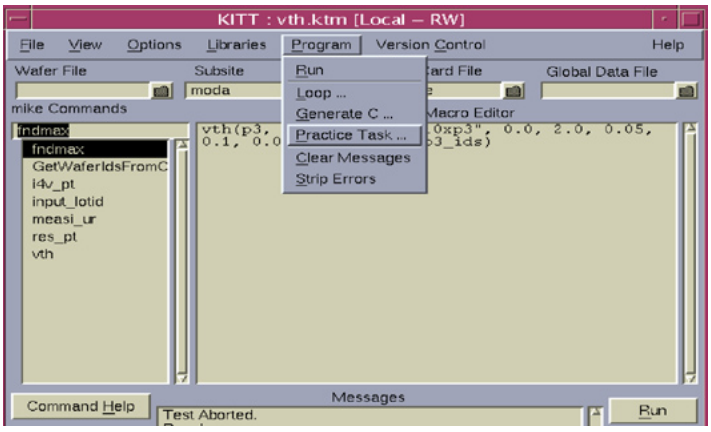

Figure 2

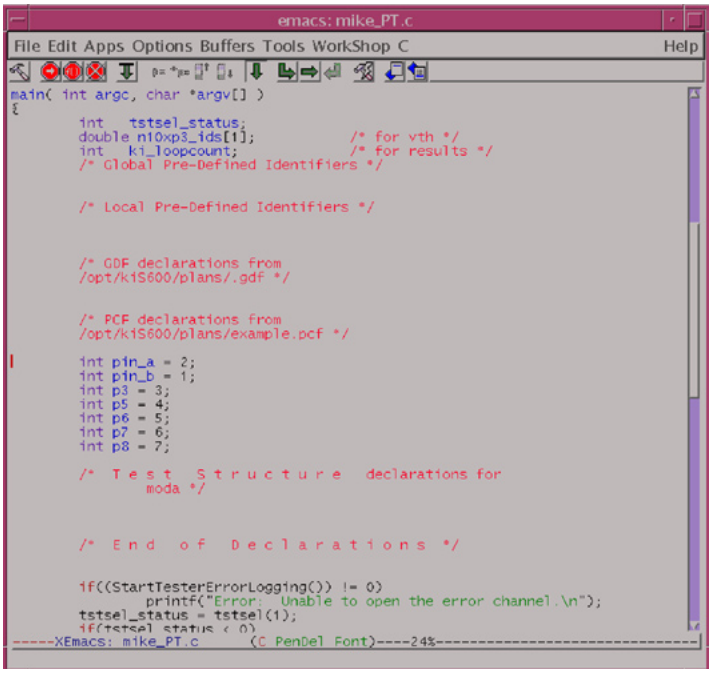

Figure 3

The text editor used here is Emacs, but the Debugger can be configured to use any text editor, such as NEdit.

To execute the Practice task from the beginning, click either green down arrow (located at the top of window along with the Debugger icons).

Now the Practice Task will execute. In this example, a SIGEGV fault (segmentation violation) has occurred at line 59 in the code (see ***Figure 4***).

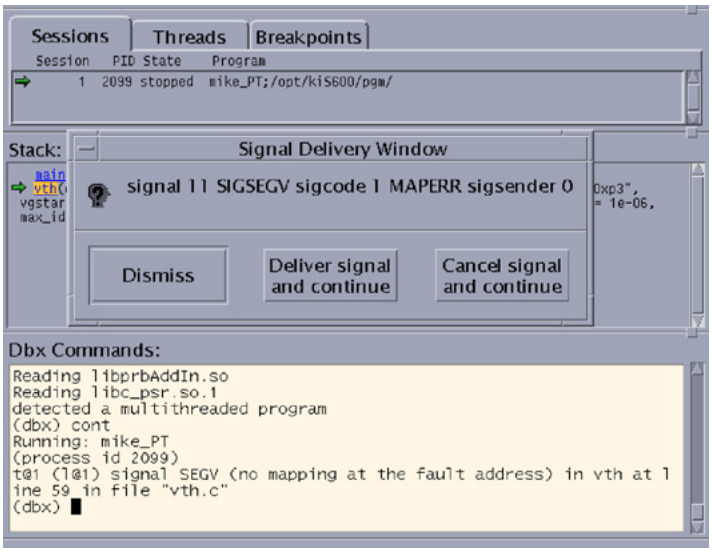

Figure 4

As ***Figure 5*** illustrates, the green right arrow in the Debugger will identify the line of code that generated this fatal condition.

In the *vth* routine, the line that caused the crash is:

drange = *rngptr;

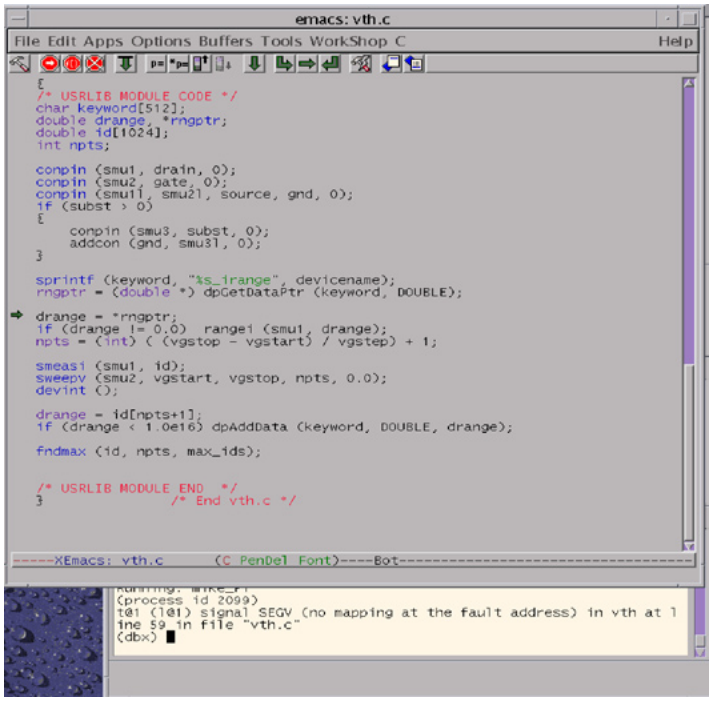

Figure 5

The variable **rngptr** is to contain a memory address that points to an instrument measurement range value previously stored in memory. The **dpGetDataPtr** routine returns the memory address for the tag (identifier) defined in the variable **keyword** and is a DOUBLE data type. In this example, the keyword is equal to: "n10xp3_irange", which is the combination of the name of the device "n10xp3" and the string "_irange".

The variable **drange** is to contain the drain measurement range that will be set using the **rangei** command.

If the tag is not defined in memory, a NULL pointer (address) is returned to **rngptr** and the code tries to assign the value at this address (NULL), which is illegal.

The problem here is "n10xp3_irange" of type DOUBLE was not found in memory and when the NULL address was accessed to retrieve the range value, the C code crashed and the macro aborted. At this point, you need to investigate why "n10xp3_irange" was not stored in memory in the first place.

One suggestion in the *vth* code is to check for the NULL address before trying to access it. You could modify the code as follows:

```
rngptr = (double *) dpGetDataPtr (keyword, DOUBLE);
if (rngptr == NULL)
        drange = 0.0; /* Set to auto-range */
else
        drange = *rngptr;
```

Scenario 2: The KTXE program crashes

The test macro discussed in Scenario 1 is included in a cassette plan. When KTXE executes this cassette plan, a segmentation

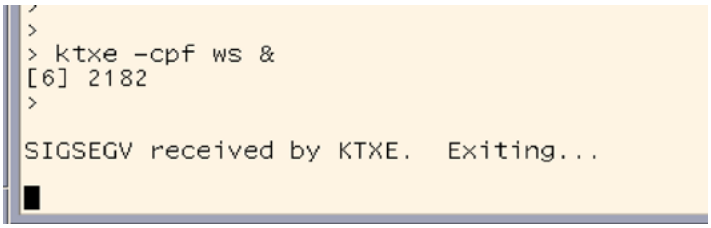

Figure 6

fault could generate the error message shown in ***Figure 6***, and KTXE will abort. Here in Scenario 2, the ws cassette plan (ws.cpf) crashed.

At this point, you need to debug the entire cassette plan. You can use the Debugger to step to (run up to) the offending line, much as in the previous scenario. Type in the command:

workshop -D $KIBIN/ktxe &

This will start the Debugger and load in all the run-time files needed for the execution of KTXE.

NOTE: You may get a warning message about No symbolic information. If this occurs, click OK.

Now select the "Edit Run Parameters for ktxe" from the Debug menu. Enter the run parameters for KTXE as shown in ***Figure 7*** and click OK.

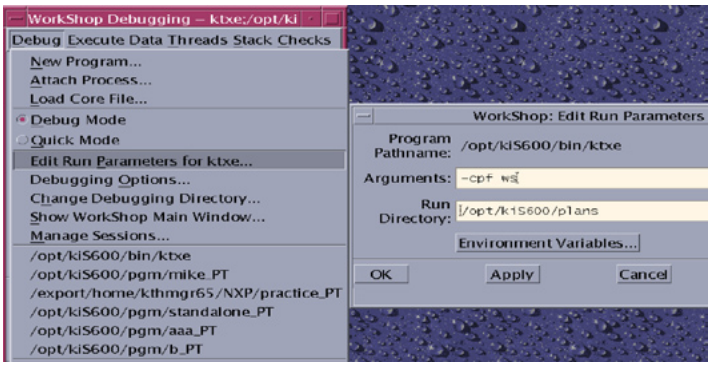

Figure 7

Select the green down arrow to start the execution of KTXE (refer to ***Figure 3***). The Debugger will execute KTXE and will display the offending line (refer to ***Figure 5***, which illustrates using the Emacs editor).

Setting break points and analyzing run-time values

Another feature of the Debugger allows you to set break points at specified locations in the execution flow. Setting a break point allows you to stop execution to perform several debugging functions:

- single-stepping through the code
- stepping into calling functions/subroutines
- analyzing run-time values

This section will illustrate how to set a break point in the *vth* routine, step through the code, and look at the run-time values.

Start the Debugger by typing in this command:

workshop -D $KIBIN/ktxe &

Set the Run Parameters, as shown in ***Figure 5***.

In the Dbx command section, load the KULT library that contains the *vth* routine (refer to ***Figure 8***). Because KTXE has not started execution yet, the KULT libraries have not been loaded and the Debugger needs to know the routine names. To load in a KULT library, type in the command:

loadobjects -p /opt/kiS600/usrlib/libexample.so

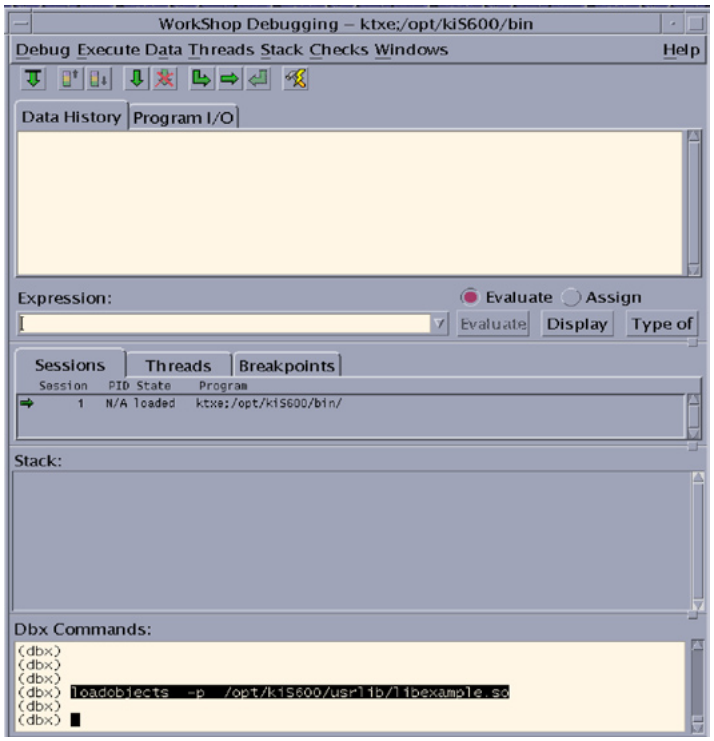

Figure 8

NOTE: The routine *vth* is located in the KULT library **example** and this library file is located in the directory /opt/kiS600/usrlib. The actual KULT library name is **libexample.so**.

Now, you need to set a break point in the function *vth*. From the Execute Menu, select "Set Breakpoints" as shown in ***Figure 9.*** Enter the function name *vth* and click Add. The list of break points will appear at the top of this box.

To start execution of KTXE, select the green down arrow on the Workshop Debugging window, as shown in the top left portion of ***Figure 8***. KTXE will now execute normally and stop at the established break points. Because the *vth* routine was selected as the break point, KTXE will execute up to this routine. The Debugger will display the source code at the break point using an editor (Emacs), as shown in ***Figure 10***.

The break point is the first executable line in the *vth* routine and indicated by a red stop sign (mostly hidden by the green arrow in the left-hand margin). The green arrow indicates which line is to be executed next (the highlighted **printf** statement).

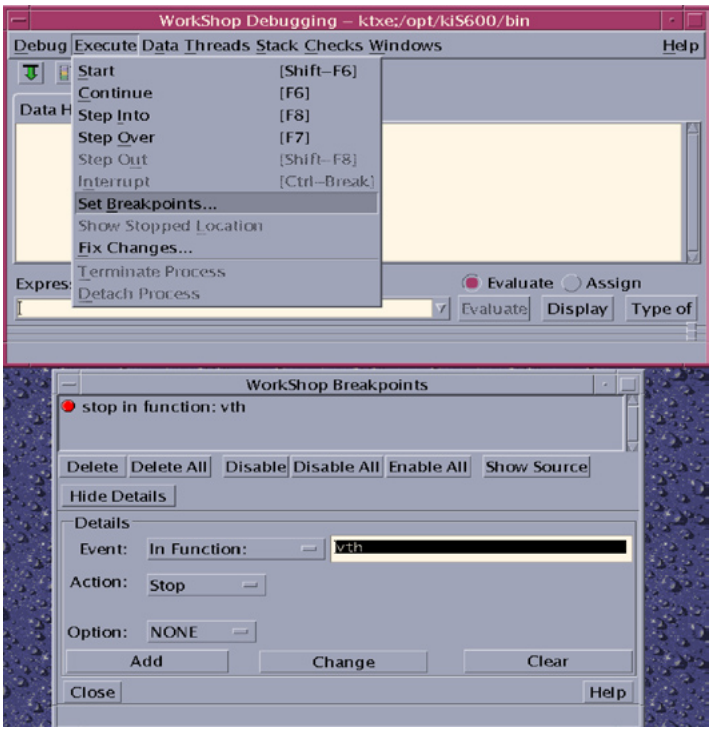

Figure 9

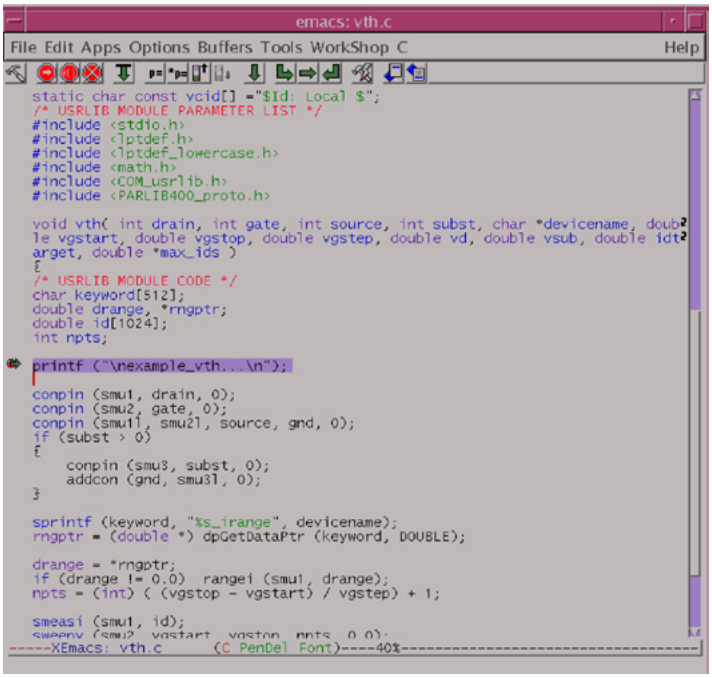

Figure 10

At this point, you can single-step through the code by selecting the down/right green arrow. Each time this button is selected, the next statement in the execution flow is performed, and the green arrow moves to the line that will be executed next. The break point (the red stop sign) will remain in the same location for future executions.

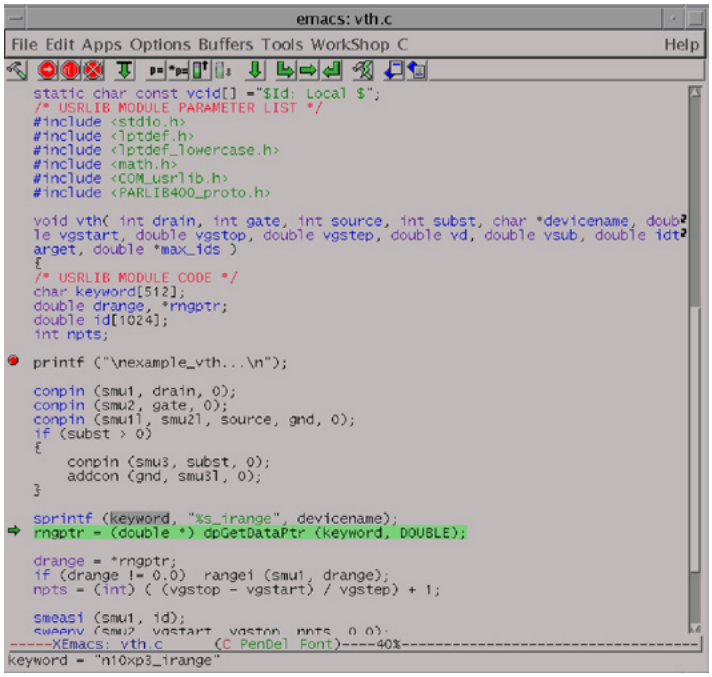

Figure 11

Figure 11 indicates we have stepped down through the code to the **dpGetDataPtr** statement. Because the **sprintf** line has completed, the variable, **keyword**, should have some value (a string of characters). You can select this variable by highlighting it; click the "**p=**" button to view its contents. The bottom of the editor box will display the contents:

keyword = "n10xp3_irange"

You can now verify whether this value is correct.

Summary

Although the Workshop Debugger offers many other useful features, the ones discussed in this note are the most useful for debugging coding problems.

KEITHLEY

KEITHLEY INSTRUMENTS, INC. ■ 28775 AURORA ROAD ■ CLEVELAND, OHIO 44139-1891 ■ 440-248-0400 ■ Fax: 440-248-6168 ■ 1-888-KEITHLEY ■ www.keithley.com

 Printed in the U.S.A. No. 2900 Jan 08