Using Advanced Event Triggering to Find and Fix Intermittent Real-Time Bugs

Dennis Kertis *Softtware Development Systems*

ABSTRACT

Finding intermittent bugs in an embedded system can be a challenge. In a real-time embedded system, the challenge increases. In addition to setting breakpoints at the precise occurrence of the problem, the user must be careful to not interfere with the timing characteristics of the system. By changing the timing of the system, the problem may either go away, only to surface later, or new problems may be introduced. Transferring state information to examine the system from the target board requires additional code and may affect the real-time aspects of the system. Advanced Event Triggering (AET) provides a way to examine the system while it is in operation, and to trigger conditional events with no overhead. This document describes how to use these powerful tools to debug a system.

Contents

List of Figures

1 Introduction

Advanced Event Triggering is a powerful new technology consisting of a group of tools that make advanced hardware debugging easier than ever before. The two main components in AET are the Event Sequencer and the Event Analysis tools.

The Event Analysis tool is designed to handle most of your needs. It allows you to:

- Set breakpoints and watchpoints
- Set actionpoints to drive external signals
- Count system events, such as cache hits and misses, pipeline stalls, etc.

The Event Sequencer tool allows you to check for events in the program and trigger actions when the events are detected. Events can be combined with Boolean operators to form complex events. To detect a series of events throughout the flow of the program, you program each state to detect an event in the series, and then proceed to the next state. Event detection all takes place in hardware, so no penalties are incurred on real-time operation of the system.

2 Event Analysis Tool

The Event Analysis tool uses a simple interface to help you configure common hardware debug tasks. Setting breakpoints, watchpoints, actionpoints, and counters is easy with a drag-and-drop interface and right-click menu.

2.1 Setting Hardware Breakpoints

With the Event Analysis tool, you can set hardware breakpoints. Setting hardware breakpoints is advantageous for many reasons. A hardware breakpoint halts the CPU at a specific location. For example, you can set a hardware breakpoint when the code is in ROM, something you cannot do with software breakpoints. This functionality is useful if your boot software is not setting up constants correctly. During the early stages of the development cycle, setting constants is typically handled by the debugger when loading code into RAM. Consider the source code in Figure 1 as an example.

```
#include <std.h>
#include <swi.h>

Void filter(Int);
Void resetInputs();

const int N_MAX = 8; // MAX taps
// additional code here …

Void main()
{
// initialize
   resetInputs();
}
```

Figure 1. Call to resetInputs Function of main.c

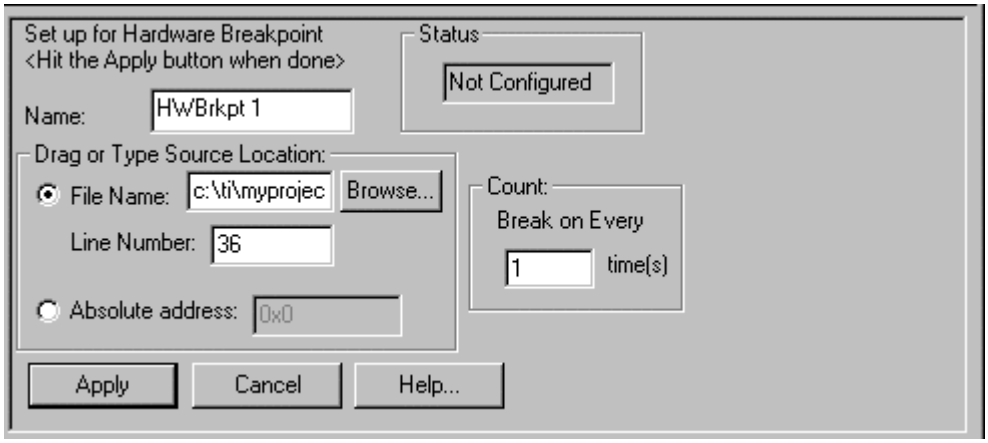
A breakpoint can be set on the resetInputs function to make sure the code is executing correctly out of the boot sequence. For example, it may be useful to break in the resetInputs routine to check a constant such as N_max. The constant can be checked to ensure that it is copied from the .cinit section to the .data section.

To work with Code Composer Studio™ debug tools, create a Code Composer Studio project, build your program, and then load the .out file onto your board.

The Event Analysis window tracks the breakpoints/watchpoints you set in your hardware. The easiest way to set a hardware breakpoint is to right-click on a selected source code line in an editor window and select Advanced Event Triggering→Toggle Hardware Breakpoint. This command opens the Event Analysis window, along with the Set up for Hardware Breakpoint dialog.

Alternatively, to add a breakpoint to the Event Analysis window:

1. Open the Event Analysis window, select Tools→Advanced Event Triggering→Event Analysis.
2. Right-click in the Event Analysis window and select Set Hardware Breakpoint. This command opens the Set up for Hardware Breakpoint dialog.
3. Drag the source line from the editor into the File Name field of the Set up window, or simply enter the address in the Absolute Address field.
4. Click Apply to add the hardware breakpoint to the Event Analysis window.



2.1.1 Setting Breakpoints at a Numbered Occurrence

You can set a breakpoint to stop only on a particular occurrence of itself. In the example source code shown in Figure 2, the generateSignal function switches from low to high on the 10th occurrence of entry into the function.

Code Composer Studio is a trademark of Texas Instruments.

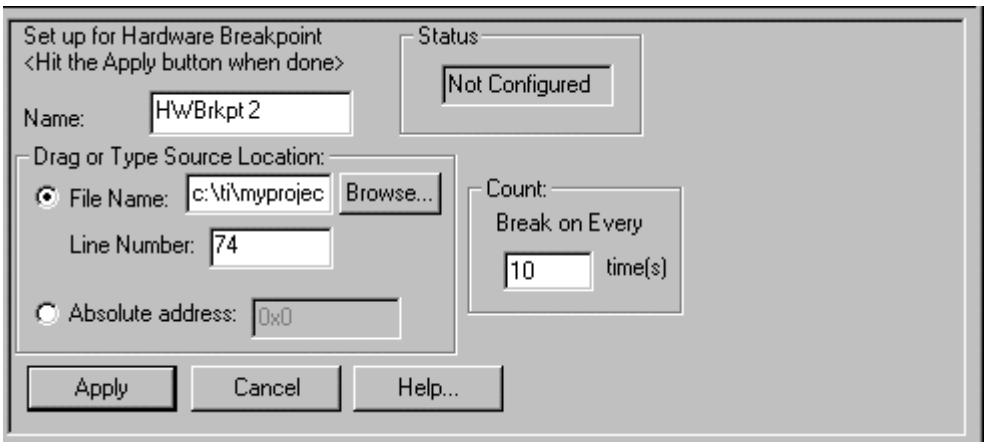
```
Void generateSignal()
{
Int input;
if (counter < 10) { // To test step function, drag this line into // set up dialog
      counter++;
      input = 0;
 }
if (counter == 10) {
      input = 1;
}

x[0] = input;
d = x[0];

SWI_post(&SWI_filter);        // run the filter on the input
}
```

Figure 2. generateSignal Function of main.c

To set a hardware breakpoint to trigger on a particular occurrence of an event:

1. Open the Event Analysis window. Select Tools→Advanced Event Triggering→Event Analysis.
2. Right-click in the Event Analysis window and select Set Hardware Breakpoint. This command opens the Set up for Hardware Breakpoint dialog.
3. Drag the appropriate source line from the editor into the File Name field of the Set up window. In this example, the source line would be:

   if (counter < 10) {
4. Change the Count field to specify the numbered occurrence. In this example, the Count field should be set to 10.
5. Click Apply to add the hardware breakpoint to the event analysis window.



On the 10th occurrence, the input should change from 0 to 1. With this functionality, you can count and compare values with minimal overhead in the software. The logic and breakpoints are carried out non-intrusively through hardware.

2.1.2 Chaining Breakpoints

The Event Analysis tool allows you to chain breakpoints. For example, suppose you want to track a bug in frequently-executing code that only surfaces after the execution of a separate piece of corrupt code. It is not necessary to break every time the frequently-executed code is run. You only need to break after the corrupt code is executed.

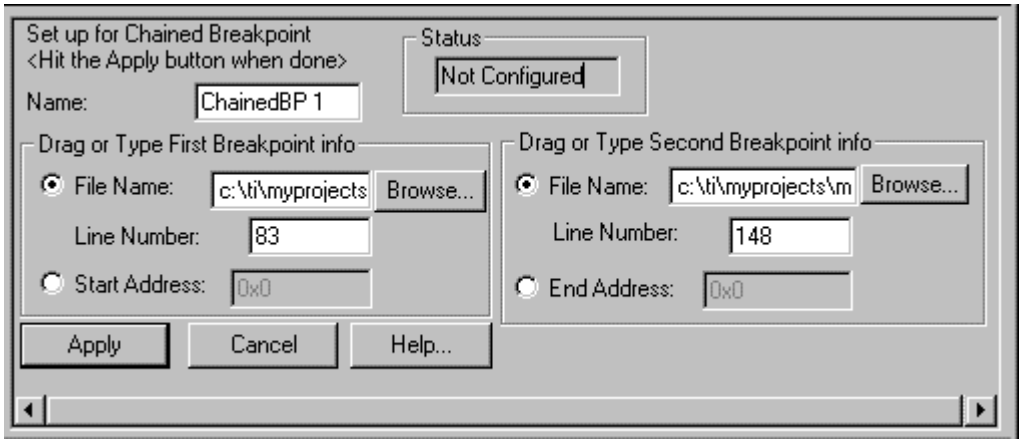
To test the filter in main.c only after the input changes from zero to one, a breakpoint could be set in the generateSignal function at the line input = 1. The breakpoint could then be chained to another breakpoint in the filter function at the line y = 0. With this configuration, the processor will not pause in the filter function until the step response has occurred (input = 1). See Figure 3.

```
Void generateSignal()
{
Int input;
if (counter < 10) {
      counter++;
      input = 0;
 }
if (counter == 10) {
      input = 1;        // This line is first chained brkpt
      }

x[0] = input;
d = x[0];

SWI_post(&SWI_filter);           // run the filter on the input
}

Void filter(Int input)
{
Int i;

y = 0;              // This line is second chained bkpt

for (i=0;i<N;i++) {
      y += x[i]*h[i];
}

// post output interrupt here
SWI_post(&SWI_D2AInterrupt);


// post inturrput to change inputs
SWI_post(&SWI_adapt);
}
```

Figure 3. generateSignal and Filter Functions of main.c

To set chained breakpoints:

1. Remove the breakpoints set previously. Right-click in the Event Analysis window and select Remove All Jobs.
2. Right-click in the Event Analysis window and select Set Chained Hardware Breakpoint. The Set up for Chained Breakpoint dialog appears.
3. Drag the appropriate source line into the File Name field for the first breakpoint. For this example, the input = 1 line from the generateSignal function should be dragged into the field.
4. Drag the appropriate source line into the File Name field for the second breakpoint. For this example, the y = 0 line should be dragged into the field.
5. Click Apply to add the Chained Breakpoints to the Event Analysis window.



Now, when running the program, the processor will only halt in the filter function (the second breakpoint) after hitting the first breakpoint, which is set to trigger when the input changes to 1.

2.2 Setting Hardware Watchpoints

A hardware watchpoint allows you to set a breakpoint on a data value, such as a variable or absolute address. This feature is useful when a data variable is being assigned an illegal value, but is accessed from many spots in the source code. Instead of setting a breakpoint at every location that changes the variable, you can set a hardware watchpoint to trigger when that variable changes. See Figure 4.

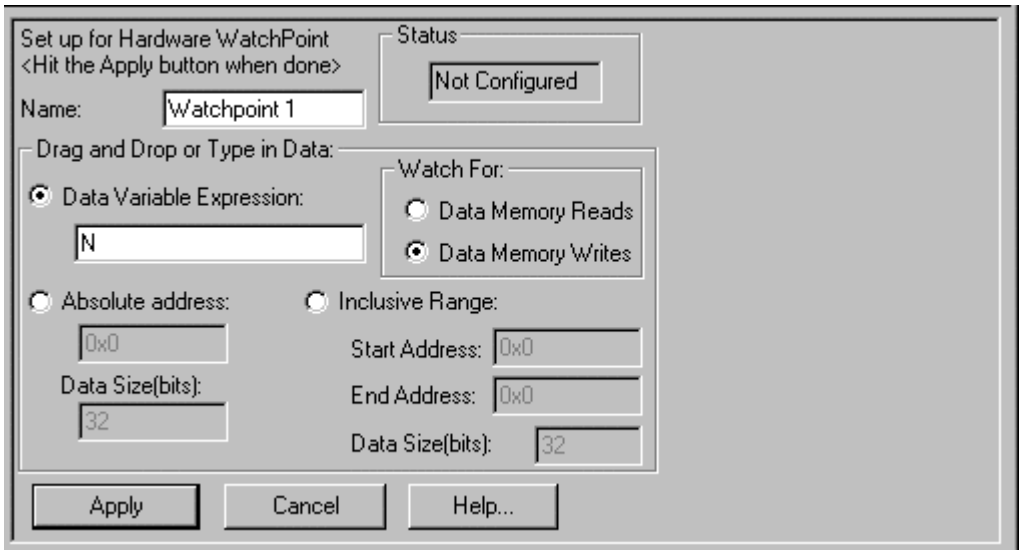
```
const Int N_MAX = 8; // MAX taps

// could be changable by host through RTDX
Int N = 8;           // filter taps
Int hostN;
Float mu;            // parameter declarations
Float alpha;
Float sigma;

Int x[8];      // past inputs
Float h[8];    // filter coefficients
```

Figure 4. Variables in main.c

For example, using the code in main.c, you may want to break whenever the variable N changes.

To set a hardware watchpoint for a variable:

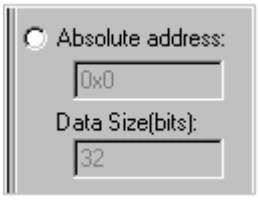
1. Open the Event Analysis window and select Tools→Advanced Event Triggering→Event Analysis.
2. Right-click in the Event Analysis window and select Set Hardware Watchpoint. This command opens the Set up for Hardware Watchpoint dialog.
3. Select the variable in the editor and drag it into the Data Variable Expression field of the Set up dialog, or enter the name of the variable into the field. In this example, variable N would be dragged into the field.
4. In the Watch For field, select Data Memory Writes. This selection halts the processor when the variable changes.
5. Click Apply to add the watchpoint to the Event Analysis window.



2.2.1 Specifying an Absolute Address for a Watchpoint

While debugging your program, you may find that a certain spot of memory is being corrupted, but do not know where it is being changed in the source code. Overwriting an array is a typical cause for corrupting memory such as a pointer. To capture an event like this, you can set a hardware watchpoint at an absolute address.

In the Set up for Hardware WatchPoint dialog, instead of entering a Data Variable Expression, select the Absolute Address field and edit it to contain the address of the memory location being corrupted.



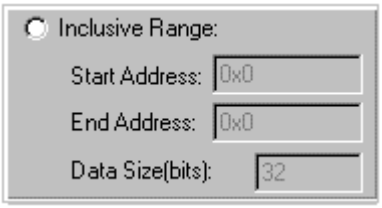
When the value at the address changes, the system halts. You can then analyze the state of the system.

2.2.2 Specifying a Range for a Watchpoint

In the Set up for Hardware WatchPoint dialog, you can specify an absolute range for the watchpoint. This feature allows you to trap a read or write to any variable in an array. For example, in main.c, you may want to halt the processor if a write occurs to any of the coefficient values (h).

To set such a watchpoint:

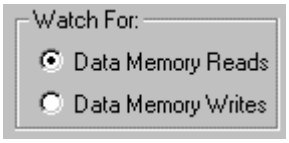
1. Check the map file for the starting address of the coefficients (h).
2. Calculate the size of the array by multiplying the size of the array type by the number of elements (8*4).
3. Add the size of the array to the starting address to get the ending address.
4. Enter the values into the Start Address and End Address fields of the Inclusive Range field.



If any of the coefficients change, the processor halts. You can then examine the state of your program.

2.2.3 Watching for Data Reads

In addition to watching for data changes, you can also set a watchpoint to trigger when data is read. To watch for data reads, in the Watch For field, select Data Memory Reads.



You may want to watch for data reads while trying to determine when you are using memory that may be corrupted. For example, if you know you have a pointer that is invalid, you want to find the location in code where it is crashing your system. Setting a watchpoint on a read from this variable allows you to track this sort of information. Errant pointers that read from I/O addresses can be detected in a similar manner.

2.2.4 Checking for Stack Overflow

You can use watchpoints to check for a stack overflow. Set a watchpoint to watch a single address or range of addresses located at, or just below, the last memory location at which the stack is linked into memory. When the program accesses that memory, you know that your stack has overflowed, and you either need to change your program or allocate more memory to the stack.

If you try to watch a variable by selecting a range of memory just below the stack, note that other variables may be using that memory. There may be some confusion about what is actually changing that variable. Check your map file before specifying an absolute address or an address range. Alternately, you can leave some memory unused in the section allocated after the top of the stack. Since your program does not use this memory by convention, only stack overflow operations cause memory writes to that area.

2.3 Setting Actionpoints

Actionpoints are similar to breakpoints and watchpoints. They allow you to monitor various data and program events, such as memory reads/writes from a data or program address. However, instead of halting the processor, they allow you to specify a particular action to be performed as a result of the event.

2.3.1 Setting Program Actionpoints

A program actionpoint monitors the program bus for a specific address. You can set an actionpoint on a specific line of source code or on an absolute address in memory. When your processor reaches the code you selected, it performs the action specified by the actionpoint.

A program actionpoint is useful for driving the trace on a logic analyzer through an external pin. For example, to drive an external pin whenever the filter interrupt of main.c is posted, you can set an actionpoint on the call to SWI_post in the generateSignal function. See Figure 5.

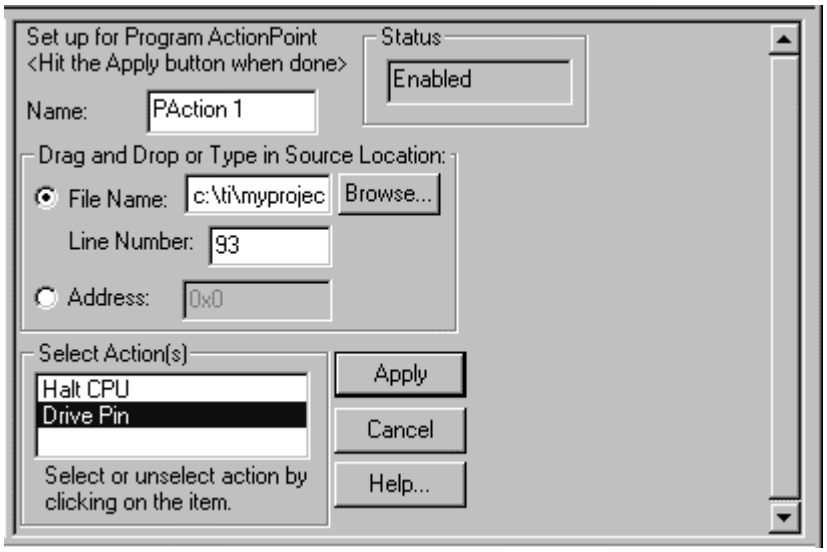
```
Void generateSignal()
{
 // code shown in Figure 2 ...

 SWI_post(&SWI_filter);            // run the filter on the input
}
```

Figure 5. Call to SWI_post in generateSignal Function

To set a program actionpoint:

1. Open the Event Analysis window. Select Tools→Advanced Event Triggering→Event Analysis.
2. Right-click in the Event Analysis window and select Set Program Actionpoint. This command opens the Set up for Program ActionPoint dialog.
3. Select the source code line in the editor and drag it into the File Name field of the Set up dialog. In this example, SWI_post(&SWI_filter); would be dragged into the field.

4. In the Select Action(s) field, select Drive Pin.
5. Click Apply. This opens the Choose Pin Option dialog.
6. Select the appropriate pin and click OK.



When you set an actionpoint in this way, you have the option to specify an absolute address.

Alternatively, you can set a program actionpoint as follows:

1. Select the appropriate line of code in the editor. In this example, SWI_post(&SWI_filter); would be selected.
2. From the right-click menu, select Advanced Event Triggering→Toggle Action/Watchpoint →Toggle Program Actionpoint.
3. In the Select Action(s) dialog box, select Drive Pin for the action.
4. Click OK.
5. In the Choose Pin Option dialog, select the pin to drive.
6. Click OK.

2.3.2 Setting Data Actionpoints

A data actionpoint monitors the data bus for a read or write from either a specific address or a range of addresses. Data actionpoints can trigger upon accessing a specific variable, an absolute address, or a range of data. When the processor accesses the specified data, it performs the action specified by the actionpoint.

Typically, when debugging with a scope or logic analyzer, a timing diagram of the system is needed. This diagram can provide information such as when specific variables or signals change, and in what order they change.

Advanced Event Triggering can be used to trigger a scope and provide this information. Set a data actionpoint on the variables to be monitored. Specify whether you are monitoring a read or write, and select a pin to drive when the event occurs. You can then hook a multi-channel scope to each pin and view the state changes as your program is executing in real time. For example, using main.c, you can drive a pin when the global variable y changes.

To set the data actionpoint:

1. In the editor, highlight the appropriate variable. In this example, you would highlight y.
2. From the right-click menu, select Advanced Event Triggering →Toggle Action/Watchpoint →Toggle Data Action Point.
3. In the Select action(s) dialog, select Drive Pin for the action.
4. In the Watch For field, select Data Memory Write.
5. Click OK.
6. In the Choose Pin Option dialog, select the pin to drive.
7. Click OK.

When all the variables have changed and each channel in the analyzer is configured, you can set a program actionpoint to halt the processor. This gives you time to examine the state diagram.

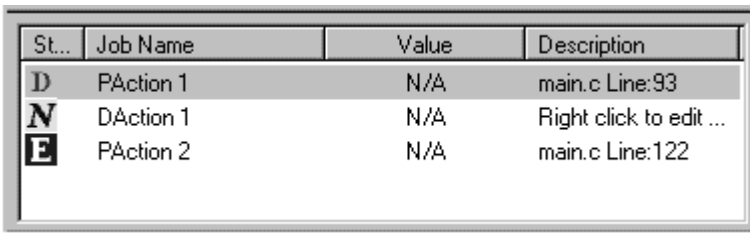
2.4 Job Management

Every breakpoint, watchpoint, actionpoint, and counter is contained in a job. A job contains all the information about the event, such as its absolute address, whether it monitors the data or program bus, etc. The job also keeps track of the information sent to the hardware.

A job has one of five states:

- Enabled. The job is configured and programmed correctly.
- Disabled. The job is configured correctly, but not currently programmed.
- Not configured. A new job has been created, but the parameters have not yet been entered.
- Triggered. The job is currently active, the event has been detected, and the action has taken place. For example, a breakpoint job has triggered when the specified location has been reached (the event has been detected), and the CPU has halted (the action has taken place).
- Error. A new job has been created, but the parameters are incomplete or incorrect.

The number of jobs that can be performed is limited by the available hardware. To work around this limitation, create several jobs and disable the jobs that are not currently needed. Jobs are displayed in the Event Analysis window.



To disable a job, simply right-click on it and select Disable This Job. To enable the job, right-click on it and select Enable This Job.

If you no longer need a particular job, you can remove it. Right-click on the job and select Remove This Job.

To enable, disable, or remove all the jobs, right-click in the Event Analysis window. Select Enable All Jobs, Disable All Jobs, or Remove all Jobs, as desired.

Every job has a Job Name field. Both the job name and description are given default values that are appropriate for the type of job. The name provides a description of the job's purpose. To change a name, click on the job and edit the name in the Set up dialog box that appears in the right pane of the Event Analysis tool.

You may also enter a description for your job to provide more detailed comments. To edit the job description, right-click on the job and select Edit Job Description.

To save the jobs created in the Event Analysis tool, simply save your Code Composer Studio workspace. When you restart Code Composer Studio, reload your workspace. The Event Analysis tool restores all the jobs present when the workspace was saved.

3 Event Sequencer

The Event Sequencer helps you to debug your program when a complex sequence of events occurs before the bug occurs. The Event Sequencer also allows you to create more complex versions of the jobs in the Event Analysis tool. For example, with the Event Sequencer, you can set a watchpoint with a count.

The Event Sequencer can provide a debug solution for the following sample scenarios:

- Trigger at the nth occurrence of an event, but only if another event has occurred.
- Trigger if data changes when a Boolean value is set (or not set).
- Trigger if the program writes to a pointer and the index value is out of range.
- Trigger if a value changes inside a range of code.
- Trigger illegal instruction access by setting a breakpoint on PC access outside of a valid range of code.

To open the Event Sequencer in Code Composer Studio, select Tools→Advanced Event Triggering→Event Sequencer.

The event sequencer is made up of two panes. The left pane contains the State Machine program you will create. The right pane contains a list of actions.

There are three types of statements you can create in the Event Sequencer:

- Global Actions
- Global If
- State If

All three statements are evaluated as soon as execution of your program begins or resumes. Global Actions take place immediately. Actions that are conditional to events in Global If or State If statements take place only when the specified conditions are true.

3.1 Using the Event Sequencer to Set a Complex Breakpoint

The Event Sequencer can be used with the breakpoint example described in Section 2.1.1. In the example, a breakpoint was triggered on the 10th occurrence of the generateSignal function. Suppose that the coefficients are not reinitialized to zero after every 300 samples. The second time the step response occurs, the response would be different because the initial values of the coefficients would not be zero. To examine this, it is necessary to break only on the second step response. See Figure 6.

```
Void outputSignal()
{
 // resets filter every 300 samples, if new parameters are read from host
 // they would be updated now
 if (outCounter == 300) {

       hostN = 8;          // simulate 8 sent through RTDX
        N = hostN;
       SWI_post(&SWI_resetInputs);
}

outBuffer[outCounter++] = y;
}

Void generateSignal()
{
 Int input;
 if (counter < 10) {
        counter++;
        input = 0;
}
if (counter == 10) {
        input = 1;
}
// additional code here
}
```

Figure 6. outputSignal and generateSignal Functions

The Event Sequencer can be used to set such a breakpoint:

1. Right-click in the Event Sequencer window and select Add State.
2. In the Editor, select the line hostN = 8 (in outputSignal) and drag it into the if Undefined Event field in the Event Sequencer.
3. Right-click in the Event Sequencer window and select Add State.
4. Drag the line input = 1 (in generateSignal) into the If Undefined Event field.
5. Select the Undefined Action field in the State #1 event. From the Action dialog, drag Goto State into the Undefined Action field.
6. In the State Number dialog box, select 2 and click OK.
7. Select the Undefined Action field of the state #2 event. From the Action dialog, drag Halt CPU into the Undefined Action field.

When the code is executed, the execution is interrupted only after both states in the sequence have been satisfied.

3.2 Handling Sequencer Programs

The Event Sequencer allows you to save and load sequence programs. Simply right-click in the Event Sequencer window and select Save Sequencer program. To load the sequence, right-click in the Event Sequencer and select Open Sequencer Program.

To clear your sequencer program or just temporarily disable it, right-click in the Event Sequencer and select Disable Sequencer Program. When the sequencer program is disabled, other jobs can be programmed into the hardware. After disabling the sequencer program, you can re-enable it. Right-click in the Event Sequencer and select Enable Sequencer Program.

Saving the Code Composer Studio workspace also saves the current sequencer program. When you restart Code Composer and reload your workspace, the Event Sequencer tool restores the program that was present when the workspace was saved.

4 Summary

Advanced Event Triggering provides you with a powerful debugging tool by giving you non-intrusive access to, and control over, your system. The most frequently used debug tasks are presented in the source code menus. For more complicated bugs, the Event Sequencer provides a mechanism to quickly gather only the information you need to help you determine your software problem. Both tools allow you to analyze your system design by gathering statistics about your system, such as cache hits/misses and pipeline stalls, with no software overhead.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265