Application Report
SPRA387 – May 2002

Using Advanced Event Triggering to Debug Real-Time Problems in High Speed Embedded Microprocessor Systems

Charles Brokish, Jonathan Dzoba, Dennis Kertis, Kim Williams-Smith Advanced EventTriggering

ABSTRACT

As more and more system components become embedded within the system processor chip, new techniques are required for debug. Signals and busses that were historically available for probe connection at external package pins, are now hidden from the users within the processor package. This problem is referred to as *vanishing visibility* meaning that the user can no longer employ traditional techniques to *see* into the system activity. To restore visibility, debug triggers such as breakpoints, watchpoints, counters and other debug tools must now also be embedded to allow access to the embedded signals. This application report instructs the user on how to take advantage of the advanced event triggering (AET) embedded components available on TI's new digital signal processors.

1. Introduction

The terms *embedded system* and *system on a chip* have become so over-used that many forget what the terms really mean. What is actually embedded about the system? To explain this, look at todays typical system designs compared to those of only a few years ago:

Older systems were based on many discrete components. They had a microcontroller chip surrounded by peripherals, all of which were some form of separate DIP package. Debugging such systems was typically a matter of locating the signals on one or more pins within the circuit board, hooking a scope lead onto the appropriate pins, and tracing the signal on a logic analyzer or oscilloscope.

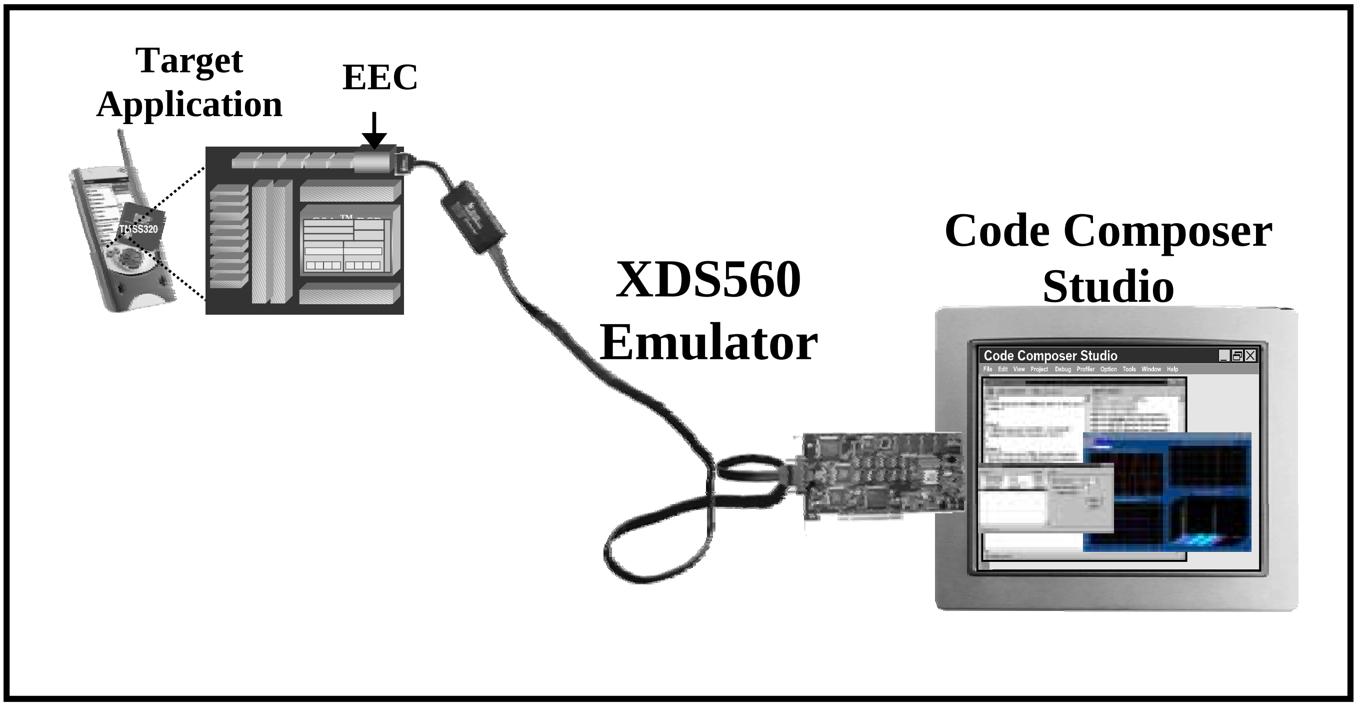
As technology progressed, the components on the printed-circuit board have become surface-mount packages in the form of thin quad flat packages (TQFP) - whose pins are very difficult to probe, or ball grid array packages (BGA) - whose pins are impossible to probe. Additionally, many of the peripherals have been integrated into the microprocessor such that their signals are no longer available at the pins. The only chance of getting trace access to these signals is to view them as they are being passed to/from off-chip memory. Furthermore, the single-bus microcontroller has been replaced with a multi-bus digital signal processor, with ever-increasing amounts of on-chip memory. Many of these busses are not visible to the outside world at the pins.

The entire system has become *embedded* in this single package, and may be truly called a system on a chip (SOC).

Code Composer Studio, XDS510, XDS560, TMS320C6211, TMS320C6711, and TMS320C6400 are trademarks of Texas Instruments.

An SOC offers many advantages to the system developer. These include lower cost, higher performance, higher reliability, lower power consumption, smaller size, and lighter weight. Along with these advantages come disadvantages in regard to system debug, primarily lack of access to embedded signals and busses. TI refers to this problem as *vanishing visibility*. The user can no longer *see* into the system with traditional debug techniques. New instrumentation techniques are required.

To restore visibility, we must now embed debug components also. Event triggers such as breakpoints, watchpoints, counters, and state sequencers are now being designed into the chip, alongside the embedded busses and signals. This restores visibility to the users by physically co-locating the debug event triggers and critical system signals. TI refers to these embedded debug components as embedded emulation components (EEC). The EEC, along with the XDS510™/XDS560™ emulator, plus associated control and user interface software, combine to provide advanced event triggering (AET) capability to the user. User access to AET is provided through TI's Code Composer Studio™ integrated development environment. The integrated AET delivery system is pictured below.



This application note provides a guided tour of the advanced event triggering capability available with TI's TMS320C6211™ DSP generation and TMS320C6711™ generation digital signal processor chips, the XDS510/XDS560 emulator, and Code Composer Studio version 2.0/2.1.

2. On-Chip Debug Capabilities

On-chip capabilities have traditionally consisted of simple comparators, which allow the user to set program or data hardware breakpoints. Advanced capabilities today include counters, event detectors, and state sequencers.

2.1 Hardware Breakpoints/Data Watchpoints

Hardware breakpoints are very useful when debugging embedded systems because they allow the user to monitor bus (data or program) activity for a certain value or a range of values, and break (stop the CPU) when the bus contains that value.

In the case of program hardware breakpoints, on-chip comparators are used to compare a desired program address with the each fetched program address. When they match, the CPU is halted for debug purposes.

Similarly, data hardware watchpoints can use a desired data address and/or data value and compare it to each running memory access. When the address and/or data values meet, the processor is halted for debug purposes.

A good example of when hardware breakpoints can be beneficial is when tracking *system hangs*. Typically, a system hang occurs when the program counter branches into an invalid memory address. Usually, by the time we halt the processor, checking the value of the program counter or any registers would be futile, because they would have already been overwritten by invalid opcodes. But by setting a hardware breakpoint to monitor the program bus, we can detect immediately when the processor tries to execute an invalid section of memory and analyze the state of our system to determine the cause of the invalid branch.

2.2 Complex/Chained Breakpoints

It is common, upon halting the CPU, to interrogate the contents of specific registers, data, or memory and decide whether to continue running until the next breakpoint is hit. This is a form of stop-mode emulation, in which we would rather not really stop the processor until *all of the specific criteria* have been met to indicate a condition in which we would like to halt the processor for deeper debug.

The ability to recognize multiple conditions and/or a sequence of events entails use of an on-chip state sequencer. By combining debug logic and enabling enough capability with the tool itself, we can allow the target CPU to run and detect a series of events, and only halt when all conditions have been met to indicate an error in the application.

For example, if we are trying to detect when a specific memory location is getting corrupted, it may be possible to put a breakpoint in typical debug tools to halt the CPU whenever that memory location is being accessed. Better yet, we could only halt when that memory location is being written to. However, if this memory location is accessed several times within a subroutine, and is getting corrupted somewhere outside of that routine, then we would have to stop the processor every time a write was performed – even when the write is supposed to happen.

Then after stopping the debugger each time to see if the program is within the proper subroutine, you have to restart the processor. Eventually you should catch the problem. But if the problem is intermittent and only occurs after an extended time, halting the processor may prevent the error from ever occurring because real-time processing has been disrupted.

A better means of detecting a problem such as this would be to set up a breakpoint that stops the CPU only if a write is done to the address of concern, when the CPU is outside of the subroutine. This requires a more complex breakpoint, and the tools must be able to accommodate such capabilities.

2.3 Counters/Event Detectors/Action Points

It is sometimes unnecessary and/or undesirable to halt the application when an event occurs. Instead, it may simply be useful to know if an event happens at all, and if so, how often. In this case, it is not breakpoints that are needed, but simply event detectors and counters. For example, when tuning an algorithm, it may be useful to detect such events as cache misses. In the case of trying to debug a system and determine if timing and design objectives are being met, one may only want to detect how often the application is getting cache misses by counting the number of times it happens within a specific code segment.

Counting events can be accomplished by adding a variable and incrementing the variable inside your function. However, this would change the timing of the system and may cause the problem to go away. By using the counters supplied with the debugging hardware, you can count function execution without affecting the performance of your system.

You can also use counters to break only every so often. For example, if you want to stop every tenth time the system hit a specific breakpoint, you can combine counters and breakpoints to perform this action.

Just as counters/event detectors can halt the CPU after certain criteria are met, they can also be used to set other system actions in motion. For instance, you may want to drive a pin high or low after a certain number of times through a range of code.

2.4 State Sequencing

Sometimes the problem we are trying to uncover is more difficult to detect. It may be that a certain memory location gets an incorrect value written to it. However, that value is written from within the proper subroutine. We cannot set a breakpoint to halt the CPU every time the address is written. Instead, we need to try detecting other events that occur coincident with this error.

As control code operates, it can take different paths through our executable program. This results in some code that rarely gets executed. In an attempt to make sure that we have covered all paths within a module of control code, we may want to check the data at the address in question only after it has gone through a specific code sequence. In this case, we would want to detect if a specific set of instructions are executed, then the address in question is written to. This is referred to as a state sequence.

3. TI's Advanced Event Triggering (AET) Debug System

Integrated with Code Composer Studio, TI's AET debug system allows the user to easily program the on-chip debug modules. AET is accessible to the user through three mechanisms:

- Context-sensitive menus in the source code window
- Event analysis plug-in for simple debug tasks
- Event sequencer plug-in for complex sequence set-up and monitoring

3.1 Basic Concepts

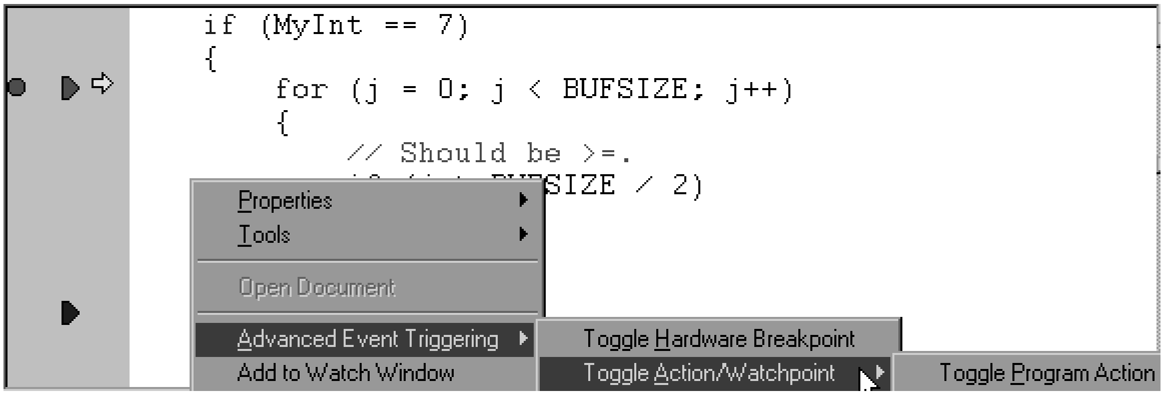
Individual AET debug actions are referred to as *jobs* in the AET menus and plug-ins.

Jobs may be set in any of three places:

- Context-sensitive menus in the source code window
- Event analysis plug-in
- Event sequencer plug-in

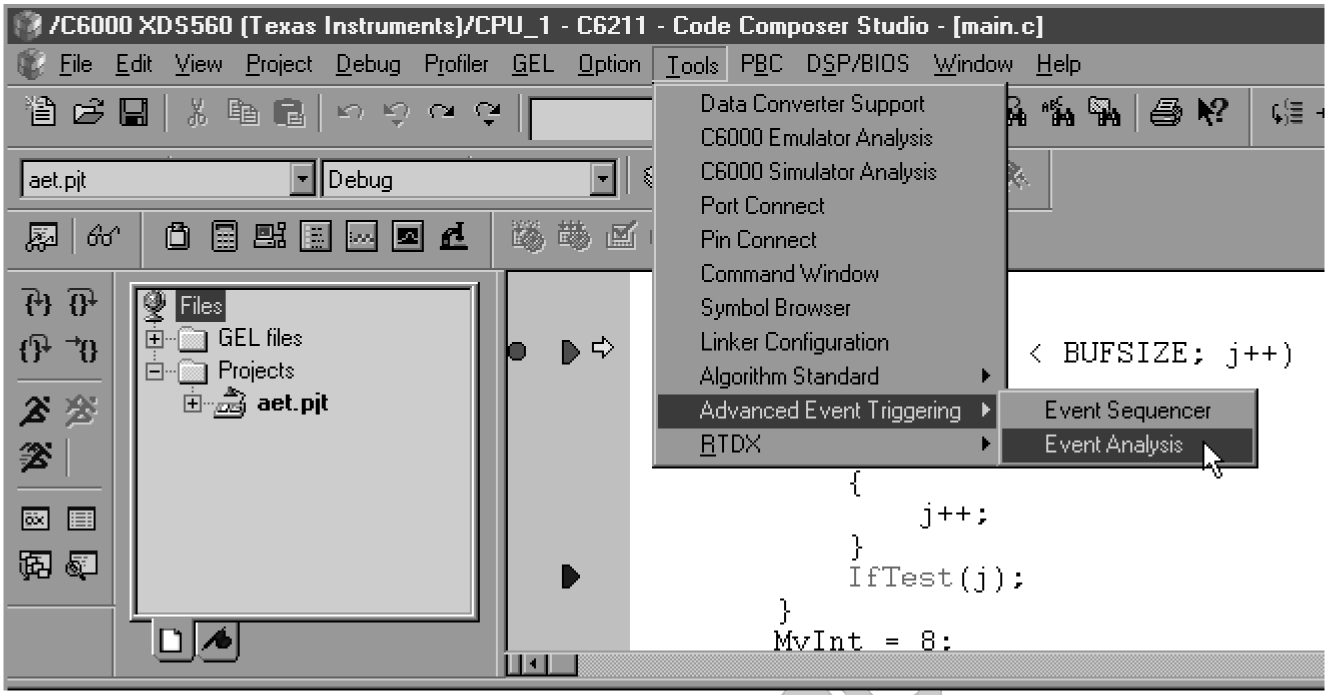
The AET menus and plug-ins abstract the user presentation from the underlying hardware implementation. Using the AET facilities requires no detailed knowledge of underlying on-chip hardware. That is, the user does not have to know how many comparators, counters, etc are on his/her DSP chip in order to setup breakpoints, watchpoints, and other more complex jobs.

Source code menus are made visible by right-clicking in the source code window and selecting *Advanced Event Triggering*:



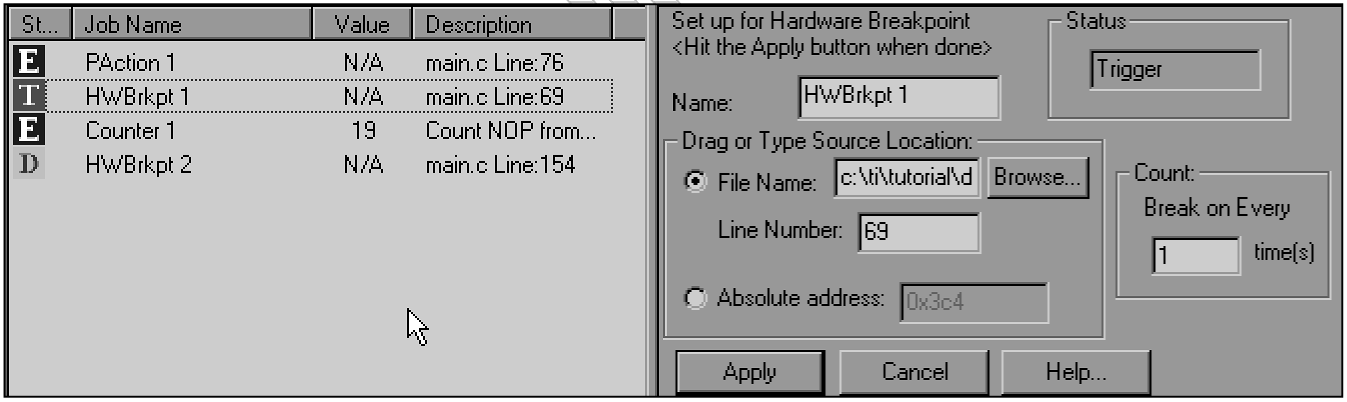
```
if (MyInt == 7)
{
    for (j = 0; j < BUFSIZE; j++)
    {
        // Should be >=.
        if (j > BUFSIZE / 2)
```

The event analysis plug-in and event sequencer plug-in are accessed by selecting *Tools -> Advance Event Triggering* from the code composer studio *Tools* menu:



```
< BUFSIZE; j++)
    {
        j++;
    }
    IfTest(j);
}
MyInt = 8;
```

All jobs are tracked in the event analysis plug-in. Here they may be added, tracked, enabled, disabled, removed, or renamed to suit the user's needs:



St...	Job Name	Value	Description
E	PAction 1	N/A	main.c Line:76
T	HWBrkpt 1	N/A	main.c Line:69
E	Counter 1	19	Count NOP from...
D	HWBrkpt 2	N/A	main.c Line:154

Set up for Hardware Breakpoint
<Hit the Apply button when done>

Status: Trigger

Name: HWBrkpt 1

Drag or Type Source Location:

● File Name: c:\ti\tutorial\d Browse...

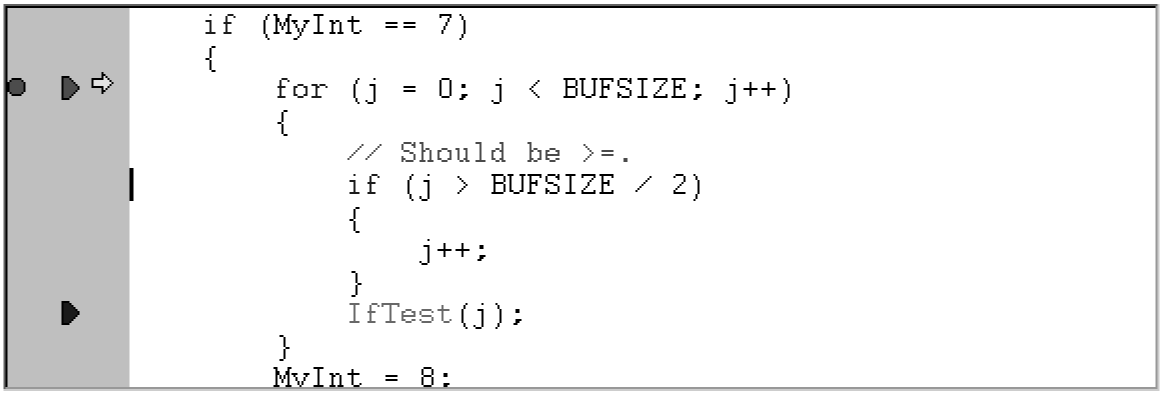
Line Number: 69

○ Absolute address: 0x3c4

Count: Break on Every 1 time(s)

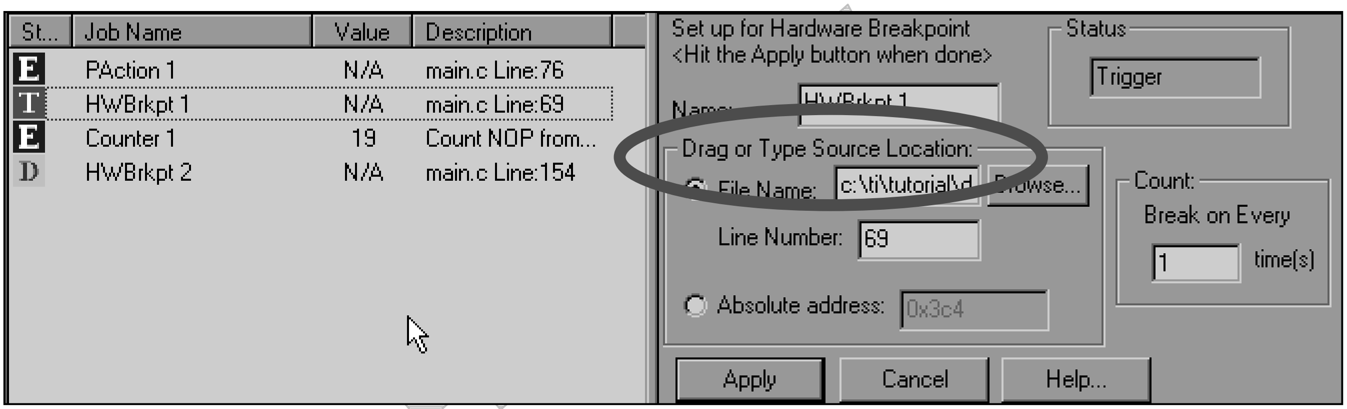
Apply Cancel Help...

AET icons appear in the left margin of the source code window to indicate AET job locations and status:



```
    if (MyInt == 7)
    {
        for (j = 0; j < BUFSIZE; j++)
        {
            // Should be >=.
            if (j > BUFSIZE / 2)
            {
                j++;
            }
            IfTest(j);
        }
        MyInt = 8;
```

Drag and drop of source code line numbers, data variable names, and desired job actions is enabled in both event analysis and event sequencer plug-ins. This minimizes the amount of typing needed to set up a job.



St...	Job Name	Value	Description
E	PAction 1	N/A	main.c Line:76
T	HWBrkpt 1	N/A	main.c Line:69
E	Counter 1	19	Count NOP from...
D	HWBrkpt 2	N/A	main.c Line:154

Set up for Hardware Breakpoint
<Hit the Apply button when done>

Status: Trigger

Name: HWBrkpt 1

Drag or Type Source Location:

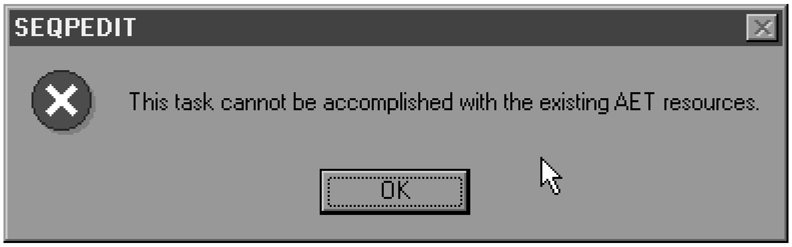
● File Name: c:\ti\tutorial\d

Line Number: 69

○ Absolute address: 0x3c4

Count:
Break on Every 1 time(s)

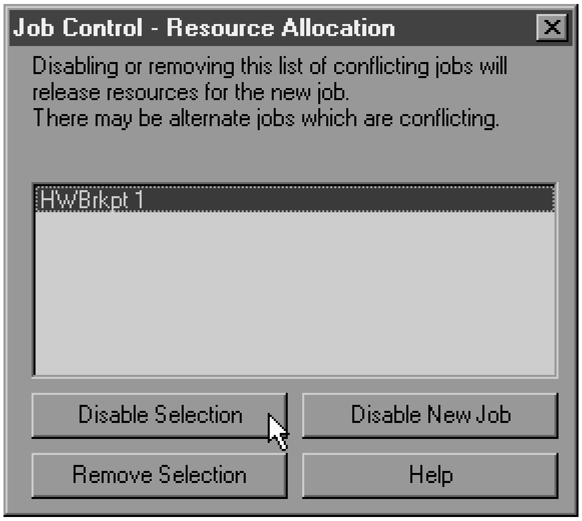
Because on-chip hardware adds gates (and therefore cost) to the chip, these facilities are naturally limited in quantity. The user is only able to setup a few jobs at a time. If you try to set up a job that exceeds the capability of your chip, the AET software displays an error message:



SEQPEDIT

This task cannot be accomplished with the existing AET resources.

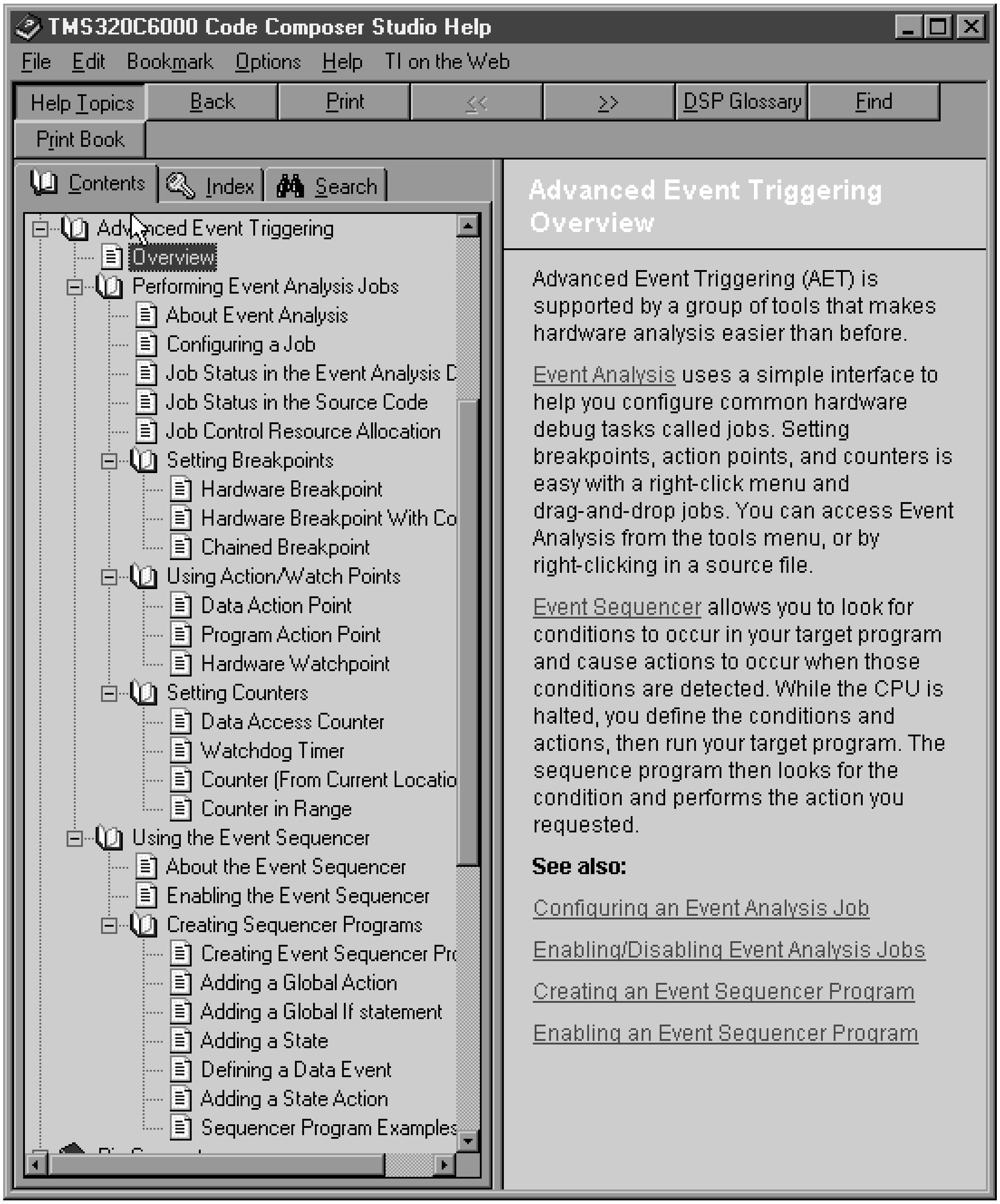
When possible AET suggests disabling other jobs to free enough resources for the current request:



Disabling or removing this list of conflicting jobs will release resources for the new job.
There may be alternate jobs which are conflicting.

Getting Detailed Help:

Detailed online help and a tutorial is available in code composer studio version 2.0 and later. To access these, select *Help -> Contents or Help -> Tutorial* in code composer studio and look for the *Advanced Event Triggering* section. An example program is also included in the <root>:\ti\tutorial\aet folder.


TMS320C6000 Code Composer Studio Help
File
Edit
Bookmark
Options
Help
TI on the Web
Help Topics
Back
Print
<<
>>
DSP Glossary
Find
Print Book
Contents
Index
Search
Advanced Event Triggering
Overview
Performing Event Analysis Jobs
About Event Analysis
Configuring a Job
Job Status in the Event Analysis D
Job Status in the Source Code
Job Control Resource Allocation
Setting Breakpoints
Hardware Breakpoint
Hardware Breakpoint With Co
Chained Breakpoint
Using Action/Watch Points
Data Action Point
Program Action Point
Hardware Watchpoint
Setting Counters
Data Access Counter
Watchdog Timer
Counter (From Current Locatio
Counter in Range
Using the Event Sequencer
About the Event Sequencer
Enabling the Event Sequencer
Creating Sequencer Programs
Creating Event Sequencer Pro
Adding a Global Action
Adding a Global If statement
Adding a State
Defining a Data Event
Adding a State Action
Sequencer Program Examples

Advanced Event Triggering Overview

Advanced Event Triggering (AET) is supported by a group of tools that makes hardware analysis easier than before.

Event Analysis uses a simple interface to help you configure common hardware debug tasks called jobs. Setting breakpoints, action points, and counters is easy with a right-click menu and drag-and-drop jobs. You can access Event Analysis from the tools menu, or by right-clicking in a source file.

Event Sequencer allows you to look for conditions to occur in your target program and cause actions to occur when those conditions are detected. While the CPU is halted, you define the conditions and actions, then run your target program. The sequence program then looks for the condition and performs the action you requested.

See also:

Configuring an Event Analysis Job

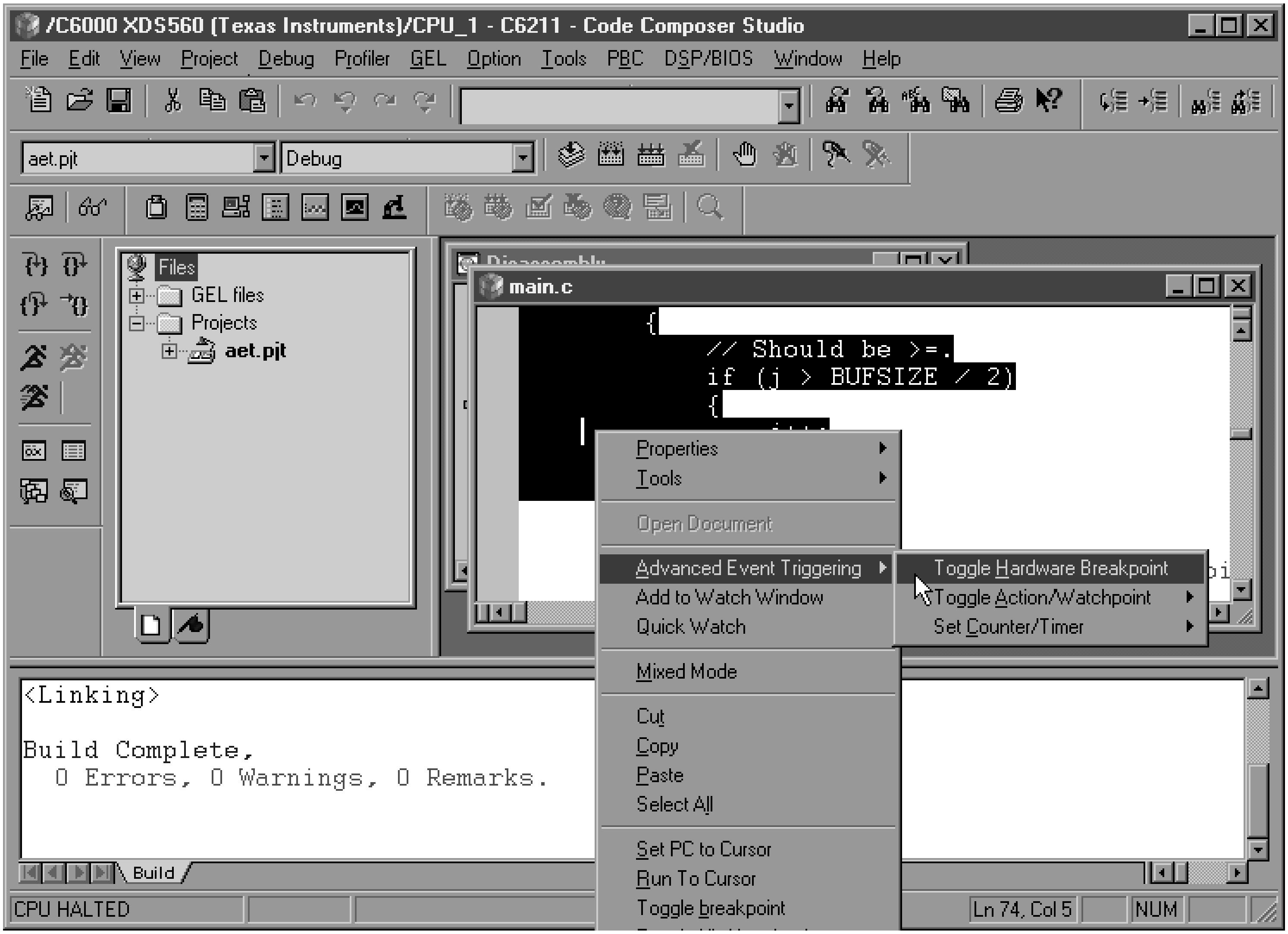
Enabling/Disabling Event Analysis Jobs

Creating an Event Sequencer Program

Enabling an Event Sequencer Program

3.2 Context-sensitive menus in the Source Code Window

Because the user spends most of his/her debug time working with code in the source code window, TI has placed the ability to access most common jobs here. To access the AET context sensitive menus, right click anywhere in the source code window and select *Advanced Event Triggering*.



```
        {
            // Should be >=.
            if (j > BUFSIZE / 2)
            {
```

```
<Linking>

Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

Some examples of the menu items are:

Program menu items appear as appropriate when one or more source code lines are selected:

- Add Chained HW Breakpoint... (Wait for cpu to get to location 1, then when cpu reaches location 2, halt.)
- Add Program HW Action Point… (Wait for cpu to get to location 1, then perform a debug action - halt CPU, Drive a pin high or low, etc.)
- Watchdog Timer… (Tell me if it takes more than N cycles to go through a range of code.)

Data menu items appear when a variable name or expression are selected:

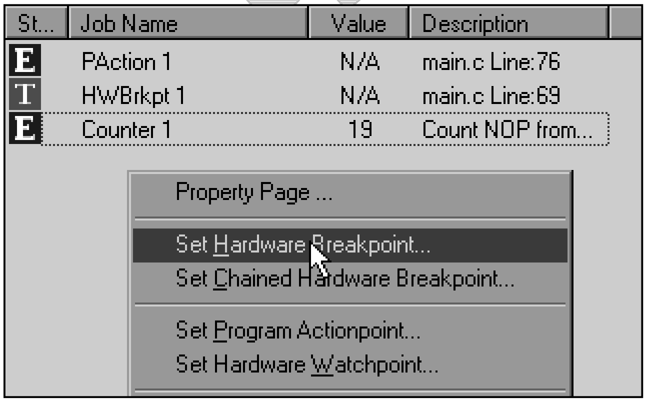
- Add HW Watch Point… (Halt the CPU if a variable changes, a commonly used action.)
- Add Data HW Action Point… (Perform one or more actions if a variable changes.)
- Count Data Accesses… (Count Read or Write accesses to a variable.)

Counter/Timer Menu Items are available at all times, regardless of code selected:

- Count… (Count the #occurrences of or #cycles spent in Cache Misses, Cache Hits, NOPs, etc., then optionally perform one or more actions.)
- Timer… (Wait a specified amount of time, then perform one or more actions.)

3.3 The Event Analysis Plug-In

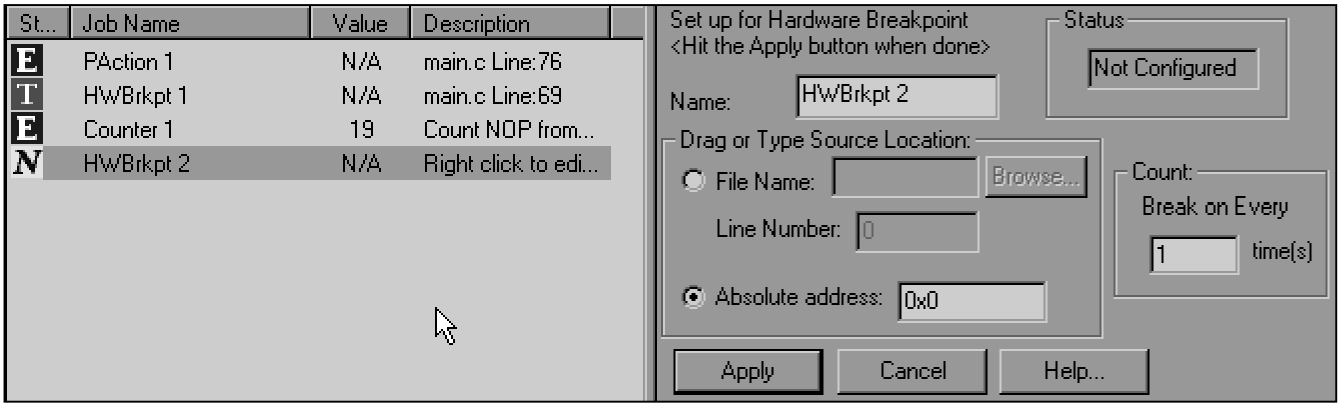
Jobs are set up in the event analysis plug-in by right-clicking in the job name field and selecting the desired job type. Below, a new hardware breakpoint job is being added:



St...	Job Name	Value	Description
E	PAction 1	N/A	main.c Line:76
T	HWBrkpt 1	N/A	main.c Line:69
E	Counter 1	19	Count NOP from...

Property Page ...
Set Hardware Breakpoint...
Set Chained Hardware Breakpoint...
Set Program Actionpoint...
Set Hardware Watchpoint...

After adding the job, it is configured by making appropriate selections in the right hand pane. In this case, the user must either type in an absolute address for the breakpoint, or drag and drop the source code line number from the source code window. The user also has the option of breaking on the first or nth time the breakpoint is hit:



St...	Job Name	Value	Description
E	PAction 1	N/A	main.c Line:76
T	HWBrkpt 1	N/A	main.c Line:69
E	Counter 1	19	Count NOP from...
N	HWBrkpt 2	N/A	Right click to edi...

Set up for Hardware Breakpoint
<Hit the Apply button when done>

Name: HWBrkpt 2

Status: Not Configured

Drag or Type Source Location:

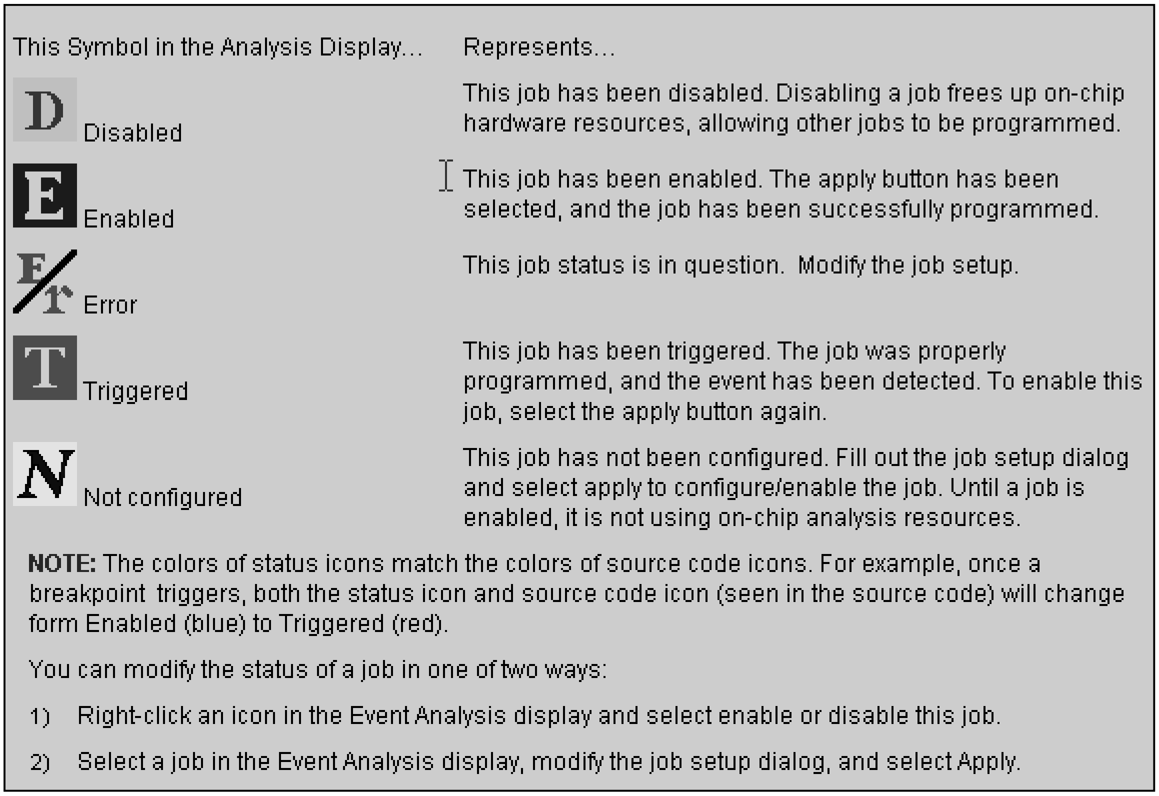
○ File Name:

Line Number: 0

● Absolute address: 0x0

Count: Break on Every 1 time(s)

All job are statused and tracked within the event analysis window. Below are the explanations of the job status codes:



This Symbol in the Analysis Display...	Represents...
D Disabled	This job has been disabled. Disabling a job frees up on-chip hardware resources, allowing other jobs to be programmed.
E Enabled	This job has been enabled. The apply button has been selected, and the job has been successfully programmed.
E/T Error	This job status is in question. Modify the job setup.
T Triggered	This job has been triggered. The job was properly programmed, and the event has been detected. To enable this job, select the apply button again.
N Not configured	This job has not been configured. Fill out the job setup dialog and select apply to configure/enable the job. Until a job is enabled, it is not using on-chip analysis resources.

NOTE: The colors of status icons match the colors of source code icons. For example, once a breakpoint triggers, both the status icon and source code icon (seen in the source code) will change form Enabled (blue) to Triggered (red).

You can modify the status of a job in one of two ways:

1) Right-click an icon in the Event Analysis display and select enable or disable this job.

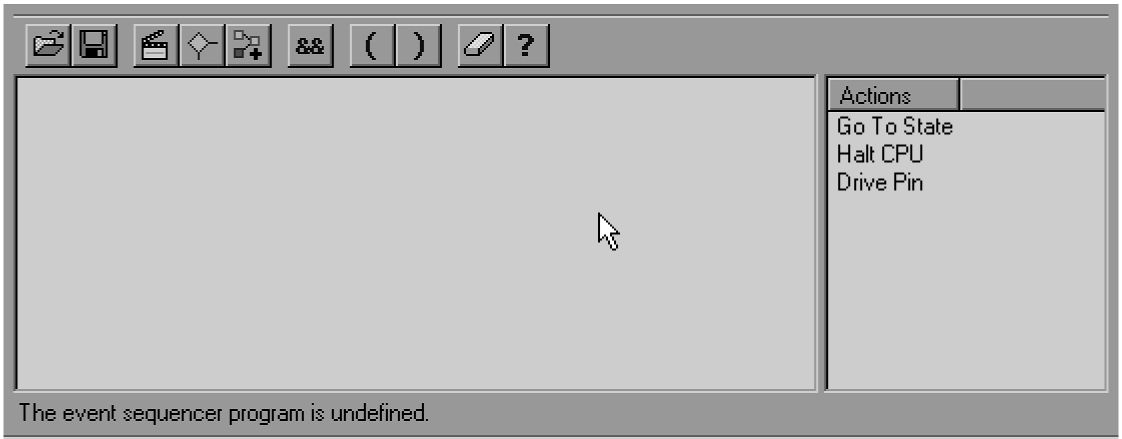
2) Select a job in the Event Analysis display, modify the job setup dialog, and select Apply.

3.4 The Event Sequencer Plug-In Features:

The sequencer plug-in is used to perform the more complex debug functions that involve sequences of events and/or program history. Every sequencer state is essentially an *If-Then* statement.

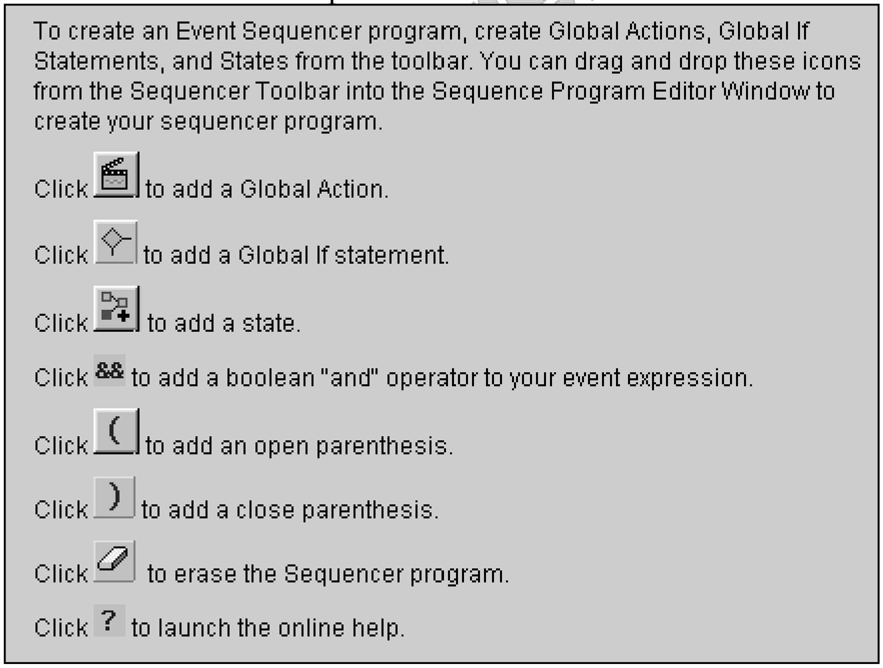
If (some event occurs) Then (perform some action)

The event sequencer window display opens with a blank program window as shown below:



The event sequencer program is undefined.

The toolbar button descriptions are as follows:



To create an Event Sequencer program, create Global Actions, Global If Statements, and States from the toolbar. You can drag and drop these icons from the Sequencer Toolbar into the Sequence Program Editor Window to create your sequencer program.

Click to add a Global Action.

Click to add a Global If statement.

Click to add a state.

Click && to add a boolean "and" operator to your event expression.

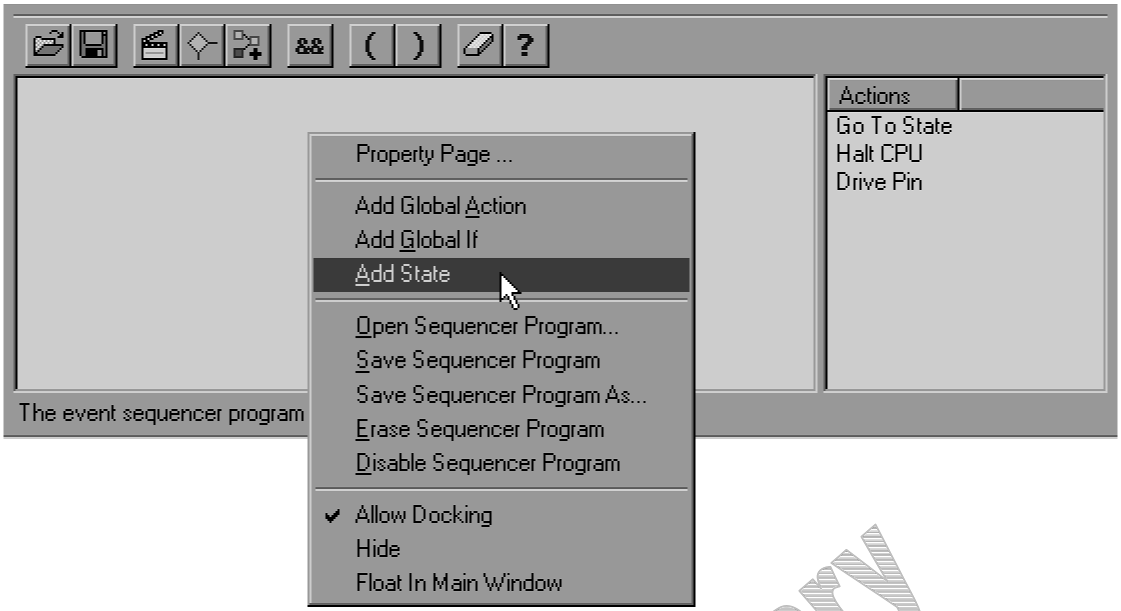
Click (to add an open parenthesis.

Click) to add a close parenthesis.

Click to erase the Sequencer program.

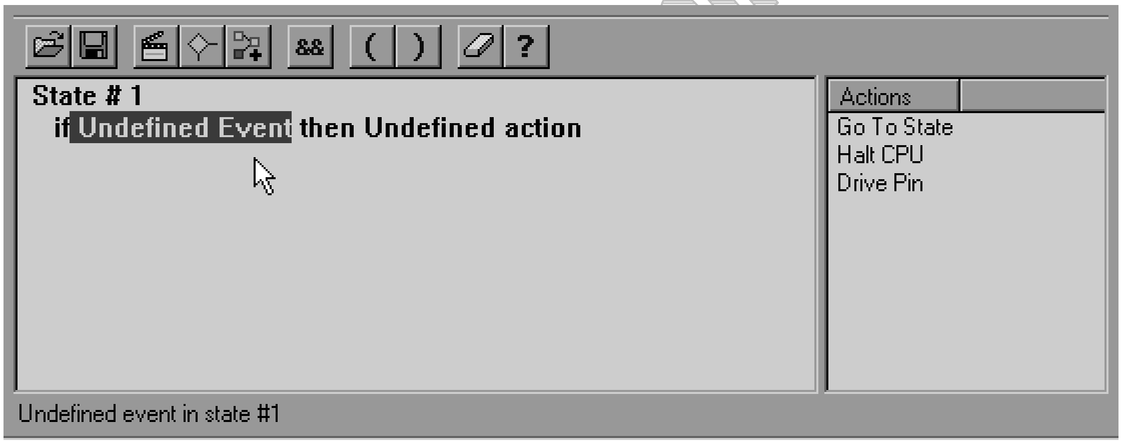
Click ? to launch the online help.

The first step in creating a sequencer program is usually adding the first step in a series of program actions. This is done by clicking the *add a state* icon or by right-clicking in the window, and selecting *Add State*:



The event sequencer program

Now, we have the skeleton of our first if-then statement:

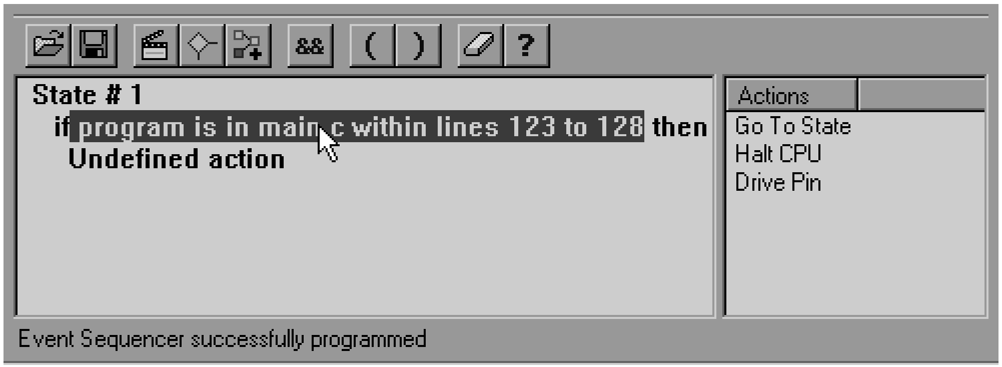


State # 1
if Undefined Event then Undefined action

Undefined event in state #1

Next, we must fill-in-the blanks.

Here is a simple case: The user has noticed that a certain data variable is getting corrupted, but only after going through a particular loop in his program. Set *State #1* to trigger after going through the loop in lines 123 to 128 of our source code:

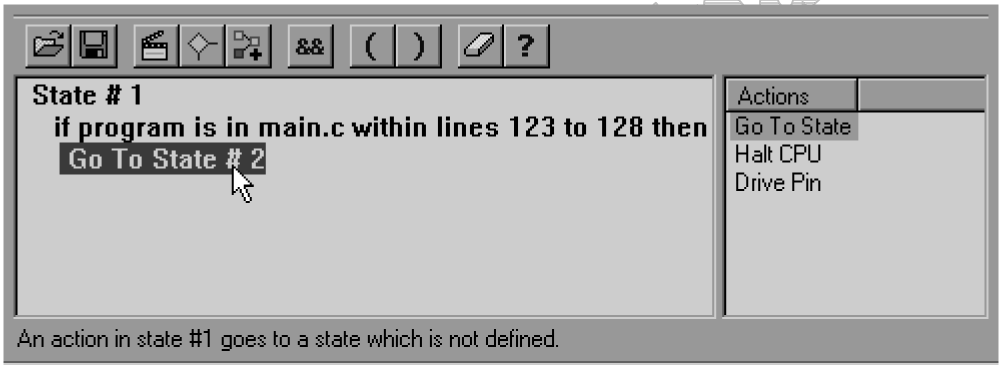


```
State # 1
  if program is in main.c within lines 123 to 128 then
    Undefined action
```

Event Sequencer successfully programmed

The event statement: *program is in main.c within lines 123 to 128* was created by selecting lines 123-128 in the source code window and *dragging and dropping* those lines into the sequencer window on top of the previous *undefined event* statement.

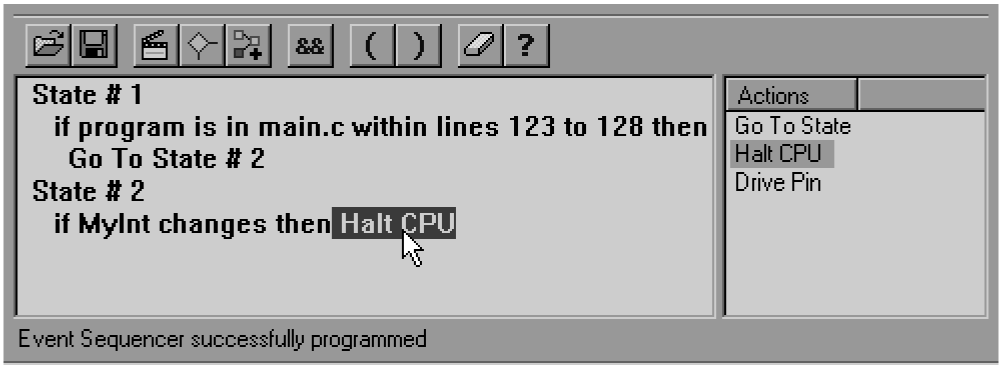
Now we set up the action that we want to happen after the loop is hit. In this case, we want to go to a second state, so the action is *go to state 2*. The go to state is created by dragging and dropping *Go To State* from the right-hand pane a*ctions* list.



```
State # 1
  if program is in main.c within lines 123 to 128 then
    Go To State # 2
```

An action in state #1 goes to a state which is not defined.

State 2 is set up to stop the CPU for debug after the data variable *MyInt* changes:



```
State # 1
  if program is in main.c within lines 123 to 128 then
    Go To State # 2
State # 2
  if MyInt changes then Halt CPU
```

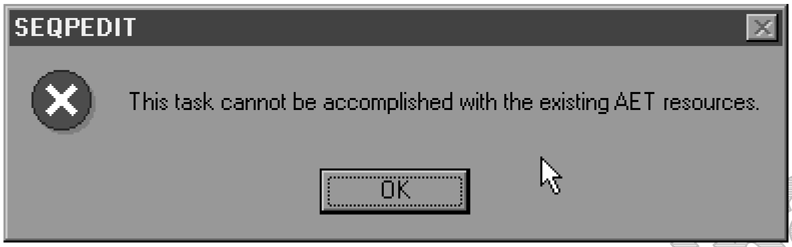
Event Sequencer successfully programmed

Once again, drag and drop is used to pull the *MyInt* symbol from the source code and the *halt CPU* action from the right hand pane *actions* list.

By setting up this simple sequence and running our program, we can halt the program for debug precisely when the data variable is being corrupted.

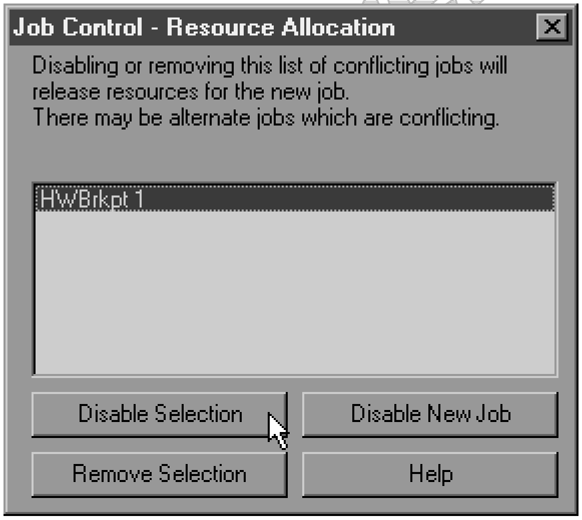
4 Special Notes regarding available on-chip resources

It is important to note that early instantiations of AET on-chip resources, such as those available on the C6211^{TM} DSP and C6711^{TM} DSP devices are relatively limited. Many useful debug operations can be performed with these devices, but complex state sequence equations are not possible. This is due to the small number of comparators, counters, etc available at the time of development of these devices. When requesting a job that is outside of the capability of your particular device, the user sees error messages such as:



This task cannot be accomplished with the existing AET resources.

When possible, AET suggests disabling other jobs to free enough resources for the current request:



Disabling or removing this list of conflicting jobs will release resources for the new job.
There may be alternate jobs which are conflicting.

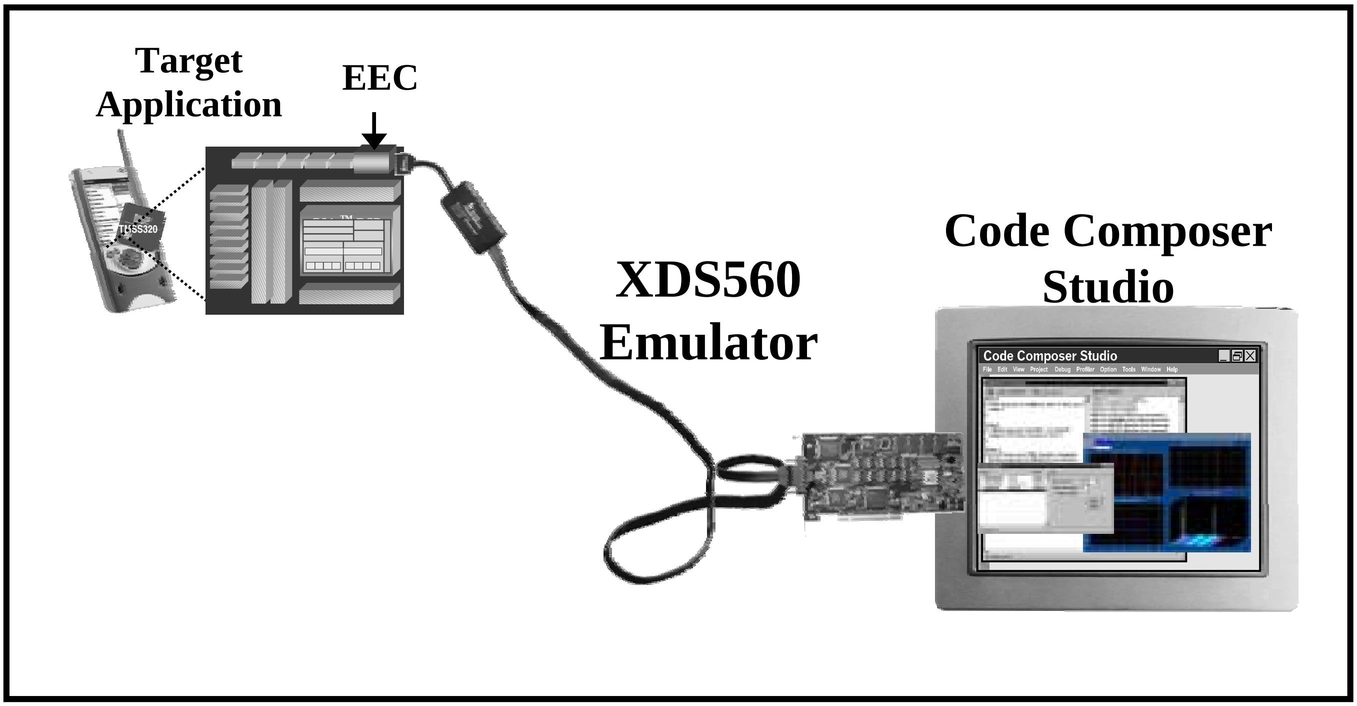
HWBrkpt 1

Newer devices such as TMS320C6400^{TM} DSP platform versions 1.1 and later, should contain much richer implementations of the AET capabilities.

5 Summary

Embedded systems cannot be debugged with traditional tools. System busses and probe points are now hidden within the device, causing *vanishing visibility*. To restore debug visibility, the debug tools must also be embedded.

TI's advanced event triggering debug system provides these embedded debug tools, and gives the user access to them through the XDS510/XDS560 emulator and the Code Composer Studio IDE.



The advanced event triggering system provides the facilities to use the on-chip components, without requiring detailed knowledge of the underlying hardware.

The most common debug jobs, such as hardware breakpoints and data watchpoints can be set directly in the source code window, where the developer spends most of his/her time. More complex jobs are setup using the two AET plug-ins – the event analysis and event sequencer plug-ins.

Early versions of the AET tools are found on TI's C6211^{TM} DSP and C6711^{TM} DSP devices. New device families will build on these and bring broader and richer visibility options to the user

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265