Application Report
SLAA263D–September 2005–Revised April 2013

Advanced Debugging Using the Enhanced Emulation Module (EEM)

Stefan Schauer *MSP430*

ABSTRACT

This document shows the benefits of the Enhanced Emulation Module (EEM) advanced debug features of the MSP430 devices and how they can be used using IAR Embedded Workbench™ version 5.51 software development tool.

The EEM advanced debugging features support both precision analog and full-speed digital debug. The configuration of the debug environment for maximum control and the use of the embedded trace capability are shown. Some techniques that allow rapid development and design-for-testability are demonstrated.

Related code files can be downloaded from www.ti.com/lit/zip/slaa263.

Contents

List of Figures

List of Tables

1 Introduction

Within every MSP430 flash-based microcontroller, there is on-chip debug logic. This Enhanced Emulation Module (EEM) provides different levels of debug features, depending on the device being used. This application report shows how the EEM can be used to solve typical debug problems.

In general, the following features are available:

- Two to eight hardware breakpoints
- Complex breakpoints
- Break on read/write at specified address
- Protection of read/write areas within memory
- All timers and counters can be stopped (device dependent)
- PWM generation may not be stopped on emulation hold
- Single step/step into and over/run in real-time
- Full support of all low-power modes
- Support for DCO dependencies such as temperature and voltage

NOTE: All examples in this application report are based on IAR version 5.51. Many of the other debuggers have the same or similar features. For details about using these other debuggers, see the user's guide of the dedicated debugger.

2 Triggers

The event control in the EEM of the MSP430 system consists of Triggers, which are internal signals indicating that a certain event has happened. These Triggers may be used as simple breakpoints, but it is also possible to combine two or more Triggers to allow detection of complex events. In general, the Triggers could be used to control the following functional blocks of the EEM:

- Breakpoints
- State storage
- Sequencer

There are two fundamental different types of Triggers, one for the address and data bus and the other for the CPU registers. It is also possible to define under which condition the Trigger is active. Such conditions include reading, writing, or fetching of an instruction. These Triggers can also be combined, so that a Trigger gives a signal if a particular value is written into a dedicated address.

3 Breakpoint

Triggers are used to configure breakpoints. This very flexible system allows the definition of various powerful breakpoints.

3.1 Address Breakpoints

A simple code breakpoint in this context would be a Trigger with a certain value (instruction address) on the address bus combined with the fetch signal of the CPU.

For the address breakpoints, one Trigger is used.

3.2 Data Breakpoints

Another type of breakpoints – data breakpoints – can be configured by using one or two Triggers. A data break could be used to check for a certain value on the address bus (memory address of the variable) combined with a read and/or write signal. It could also be enhanced, so that the break only occurs if a specific value is read or written into this address. This value is then checked on the data bus.

For a data breakpoint without a value, one Trigger is used, and for a data breakpoint with a value, two Triggers are used.

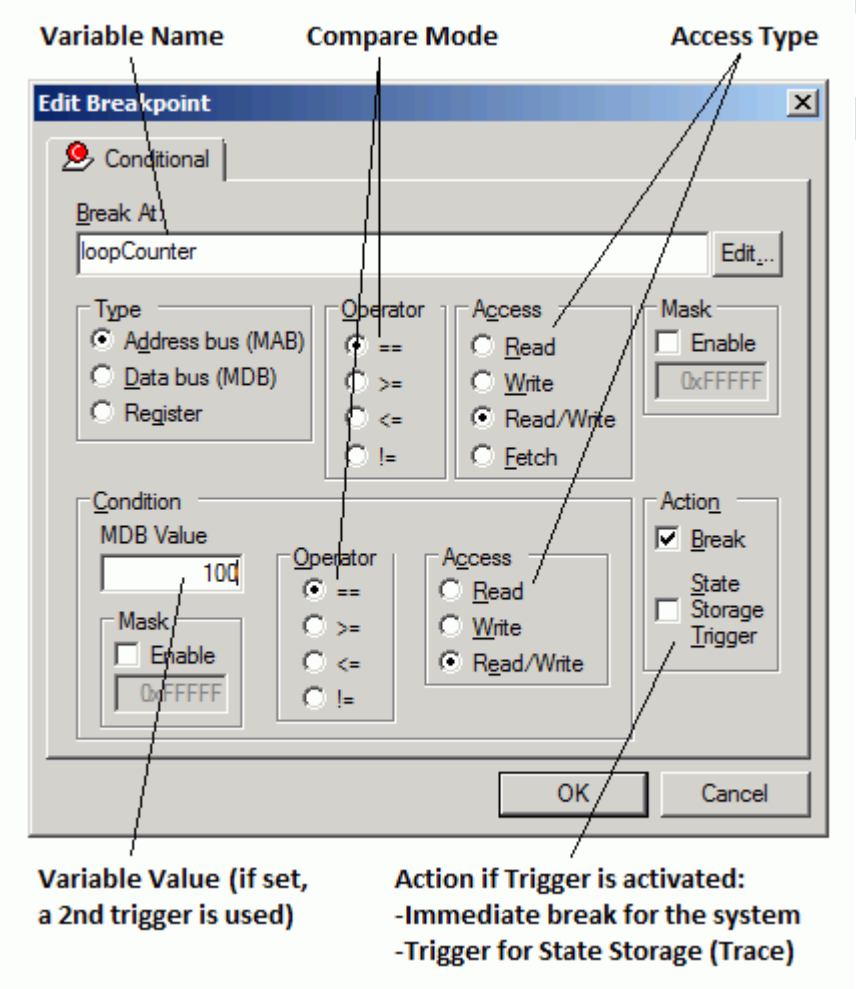

Figure 1. Conditional Breakpoint Dialog

3.3 Register Breakpoints

The same observation methods can be used for the CPU registers. This can be a very powerful tool if the program is written in assembler, where the programmer has complete control over the usage of the registers. Dedicated registers might be used for a variable or a system state flag. One register is very critical and worth observing very carefully, the stack pointer. If there is a problem within the program that allows the stack to run into the data area, it is often very difficult to find the problem with normal debugging features, as the symptoms may change each time program execution is started. A simple breakpoint, which stops the microcontroller when the stack pointer reaches a certain value or is below a certain value, helps detect such problems quite easily. To setup this Trigger, one of the Register Triggers is used.

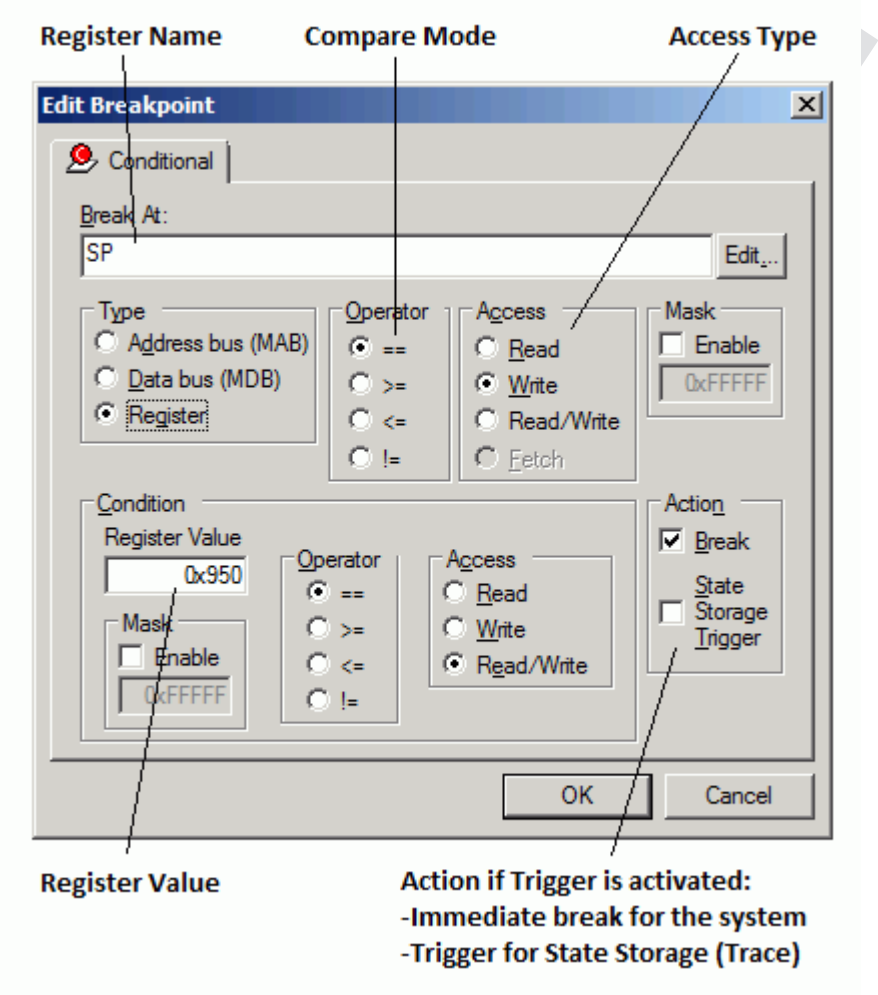

Figure 2. Conditional Breakpoint Dialog for Register

NOTE: When using an MSP430X CPU (MSP430FG46x), a second breakpoint is required for observing the stack pointer (SP) due to the speed improvement on the MSP430X CPU. Set a Conditional Breakpoint on an MAB write access with the address of the SP limit.

3.4 Mask Register

The Mask Register in the Conditional Breakpoint dialog can limit the check to only certain bits (according to the mask) of the register. This could be used, for example, to check if only a certain bit has been set/reset in the specified register.

3.5 Range Breakpoints

Range breakpoints are necessary to check an access to the dedicated memory range. Some possible conditions could be:

- Break on Write to Flash – This allows a check to determine if, during program execution, a write access into the flash memory area occurs. In many cases, this is not allowed (or is allowed only under certain circumstances) and, therefore, could be considered as an erroneous write.
- Break on Read/Write to Invalid Memory – This check could be used to evaluate if, during program execution, any attempt to access data in invalid memory range occurs.
- Break on Instruction Fetch out of Range – This breakpoint could stop the CPU if it fetches an instruction from a memory address where no program is stored.
- Break on Data out of Range – This Trigger gives a signal if the value at the data bus is inside or outside the specified range. This could be used if the value in a certain variable should be observed for this data range. To use this feature the Data Range Trigger must be combined with a write or read Trigger on the variable address or the Trigger Sequencer. Otherwise, any value that appears on the data bus and is in this range (for example, an instruction) could stop the CPU.

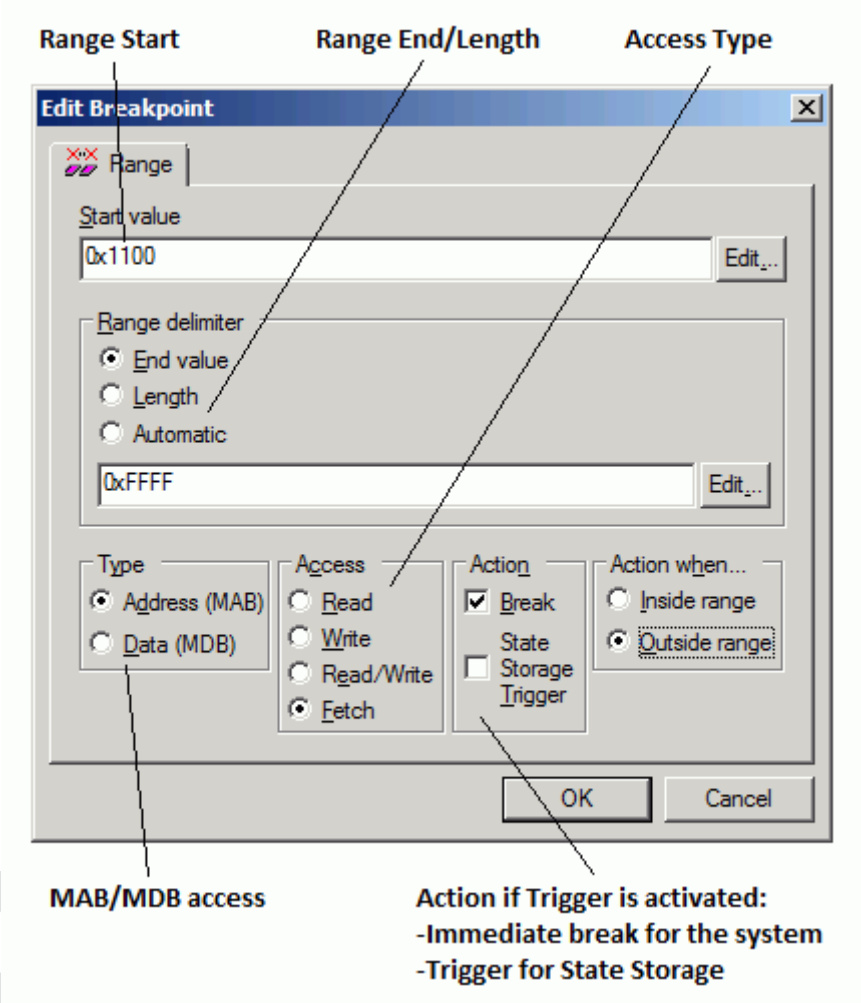

Figure 3. Setting a Range Breakpoint

4 State Storage

State storage can be used to save the information that is on the address and data bus. Additionally, some flags of the CPU, such as Read/Write or Instruction Fetch are stored. There is a total depth of eight entries available in the state storage buffer. The flexible configuration of this system makes it possible to record the required information into the state storage buffer very efficiently, so that the required information can be saved and displayed.

4.1 Default Configuration

A useful default configuration would be an instruction trace of the last few cycles before a breakpoint. To enable this, select the state storage action on instruction fetch and set State Storage to Stop on trigger. The breakpoint must also be configured to as State Storage Trigger in addition to Break. The result can then be seen in the State Storage Window.

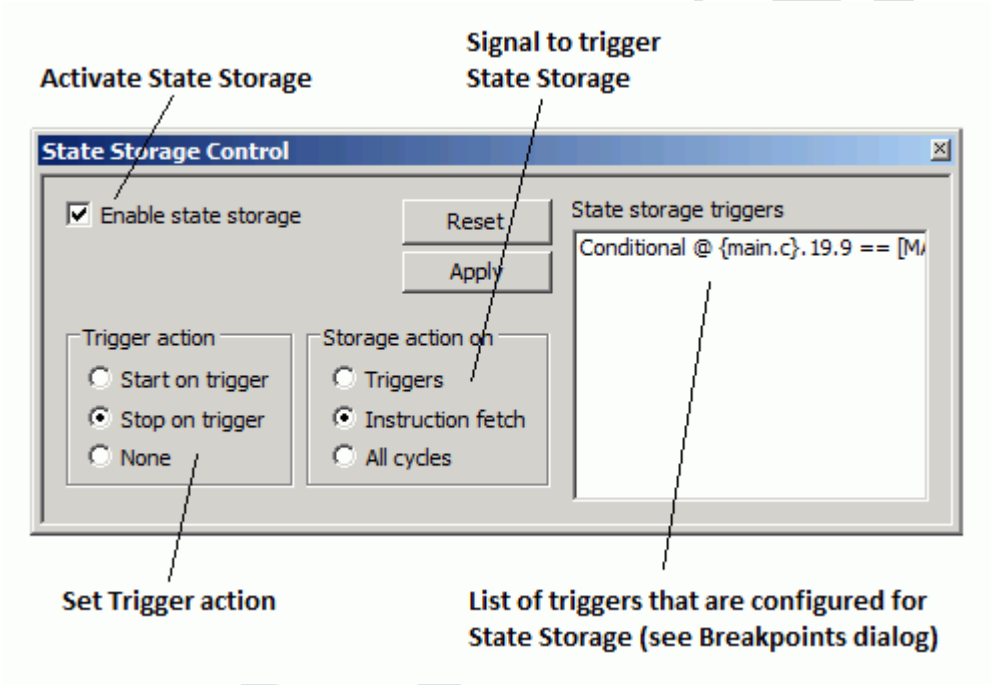

Figure 4. State Storage Control

5 Advanced Trigger Options

5.1 Manually Combining Triggers

With the Breakpoint Combiner dialog (Emulator | Advanced menu), two or more individual Triggers can be combined. When defining a complex Trigger with the manual combination of Triggers, the following points should be considered:

- One Trigger (Sub-Trigger) is added to another Trigger (Main-Trigger) with an AND combination. The Main-Trigger then contains the combination of the two Triggers.
- In a combination, only the reaction of the Main-Trigger is used. Sub-Triggers will no longer trigger a reaction on their own.

5.2 Trigger on DMA Events

The Triggers are also able to differ between a memory access hosted by the CPU or by the DMA. If a Trigger should be setup for the DMA, this must be done within the Advanced Trigger dialog. It is possible to select from all possible access types and gain full control of all features of the Trigger.

6 Clock Control

A very important part of the debugging system is the flexible clock control. The clock should, of course, be stopped on emulation hold, especially the main clock for the CPU but, depending on the application, there could be different requirements to the clock for the peripheral modules, such as the UART module that might transfer a character or a timer that is generating a PWM signal for a motor. Merely stopping these peripherals could cancel a communication or even destroy the high-power circuit of the motor control unit. The different clock control modules are listed below, and it is shown how they could be used. Table 1 shows which device has which implementation.

6.1 No Clock Control (e.g., 'F11x1,'F12x,'F13x/14x)

Modules may be clocked while the CPU is stopped by reading and writing to memory. So if, for example, a timer interrupt is enabled while the program is executed in single step, the program is permanently in the ISR.

NOTE: The only solution to single step with enabled timer interrupt is to clear the GIE bit in the status register before starting to single step.

6.2 Standard Clock Control (e.g., 'F41x)

Standard clock control stops clocks for selected modules completely; other modules maintain a continuously running clock. Clock control selection is hardwired. This means that it is possible to select if all of ACLK, MCLK, or SMCLK on the device should be stopped on emulation hold or not.

6.3 Extended Clock Control (e.g., 'F15x/16x,'F43x/44x)

Extended clock control allows the same control as the standard clock control but additionally the clock could be controlled on the module level. A generally recommended setting for this system is to stop all clocks except the USART, ADC, and flash modules. This setting allows a data transmission, ADC measurement, or write into the flash memory that was already started to be finished by the system, while all other peripheral modules are stopped on a break condition.

7 Cycle Counter

The cycle counter can be used to profile code and count the number of clock cycles required to execute a piece of code. It is displayed along with the regular CPU registers (View | Register | CPU Registers).

NOTE: On devices with a hardware cycle counter ('F(R)5xx/6xx), the measured cycles can be slightly off due to releasing the device from and taking it back under debug control. Devices without a hardware cycle counter will require single stepping.

8 Considerations

- If the JTAG fuse has been blown, access to the emulation logic is disabled.
- When using a complex breakpoint, the CPU is stopped after the instruction causing the break has been executed.
- When a break occurs, the current instruction is always completed before stopping execution.
- The EEM logic cannot prevent an invalid value from being written into a given address or register.
- Hardware registers, like the Timer_A counter (TAR), cannot be used for Triggers unless the CPU is accessing this register during the time the required value are stored in the register.

9 Emulation Module Implementation Summary

Table 1 shows in a summary the differences of the implementations of the Emulation modules in the different devices.

Table 1. Emulation Module Overview(1)

Device	F11x1 F12x	F12x2	F13x F14x	F15x F16x F16xx F26xx	F20xx F21x1 F22xx F23xx	F24x	F41x F42x FE42x FW42x F42x0	FG43x	F43x F44x	FG46xx	F543x F543xA F552x F551x	F5509 F5510	CC430 F61xx
Triggers (MAB/MDE)	2	2	3	8	2	3	2	2	8	8	8	3	3
<=/>=(2)	–	–	X	X	–	X	–	–	X	X	X	X	X
R/W	–	–	–	X	–	X	–	–	X	X	X	X	X
DMA	–	X	–	X	–	–	–	X	–	X	X	X	X
16/20-bit Mask	–	–	–	X	–	X	–	–	X	X	X	X	X
Reg.-Write-Trigger	–	–	–	2	–	1	–	–	2	2	2	1	1
<=/>=(2) (3)	–	–	–	X	–	X	–	–	X	X	X	X	X
16/20-bit Mask	–	–	–	X	–	X	–	–	X	X	X	X	X
Combination	2	2	3	8	2	4	2	2	8	8	8	4	4
Trigger Sequencer	–	–	–	1	–	–	–	–	1	1	1	–	–
Reactions													
Break	X	X	X	X	X	X	X	X	X	X	X	X	X
State Storage	–	–	–	X	–	–	–	–	X	X	X	–	–
Trace													
Internal	–	–	–	X	–	–	–	–	X	X	X	–	–
Cycle Counter (HW)	–	–	–	–	–	–	–	–	–	–	2	1	1
Clock Control													
Global	–	–	–	X	X	X	X	X	X	X	X	X	X
Modules	–	–	–	X	–	X	–	X	X	X	X	X	X

(1) Flash devices only
(2) <=/>= is Compare for ==, <>, <=, and >=
(3) Standard comparison within all devices

10 C-Spy Internal Breakpoint Usage

C-SPY itself consumes breakpoints. C-SPY will set a breakpoint if:

- the debugger option Run to has been selected, and any step command is used.
  These are temporary breakpoints which are only set when the debugger system is running. This means that they are not visible in the Breakpoint Usage window.
- the linker options With I/O emulation modules has been selected.
  These types of breakpoint consumers are displayed in the Breakpoint Usage dialog box as C-SPY Terminal I/O & libsupport module. The number of options used for this purpose depends on which library you are using. Then CLIB is used, three breakpoints are used, on putchar, getchar and exit. The user can disable these in the Breakpoint options dialog box. The DLIB is used, only one breakpoint is required for terminal I/O etc and this breakpoint can not be disabled.
- C-SPY plugin modules, for example modules for real-time operating systems, can also consume additional breakpoints. Specifically, by default, the Stack window consumes one breakpoint. This can be disabled.
- some user breakpoints consume several low-level breakpoints and conversely, several user breakpoints can share one low-level breakpoint. User breakpoints are displayed in the same way both in the Breakpoint Usage dialog box and in the Breakpoints window, for example Data @[R] callCount. Breakpoints are treated in the same way on all MSP430 devices.

11 References

1. MSP430 Product Brochure (SLAB034)

Appendix A Examples

The appendix contains some samples of the EEM features that are discussed in this application report. The code provided with this report (www.ti.com/lit/zip/slaa263) does not necessarily present a good coding style or have practical application for a specific part. The code has been designed for easy use of the EEM features. Some features, particularly allowing of nested interrupts and a delay loop inside an Interrupt Service Routine should never been used in a production application.

The menus and dialogs mentioned in the following sections can be accessed at these locations:

- Breakpoint
  View | Breakpoint
- New Breakpoint
  Right click in the Breakpoint window and select New Breakpoint.
- State Storage Configuration
  Emulator | State Storage Control
- State Storage Window
  Emulator | State Storage Window

A.1 Break on Write to Address

1. Create a Conditional Breakpoint (New Breakpoint | Conditional...)
2. Enter name of the variable in field Break At: wLoopCounter
3. Select Address bus, operator '==' and Write Access
4. Action should be Break
5. In the Condition Field set a Value of 50, Operator '==' and Write Access (do not change the mask value)
   - System stops if a value of 50 is written into the variable wLoopCounter

A.2 Break on Write to Register

1. Delete or disable previous Breakpoints
2. Create a Conditional Breakpoint (New Breakpoint | Conditional...)
3. Enter name of the register in field Break At: SP (for Stack Pointer)
4. Select Register, operator '<=' and Write Access
5. Action should be Break
6. Set Register value to 0x4350
7. No change is required in the Condition or Mask fields
   - System halts if the stack pointer decreases to a value below 0x4350

A.3 Break on Write to Flash

1. Delete or disable previous Breakpoints
2. Create a Range Breakpoint (New Breakpoint | Range...)
3. Enter Start Value: 0x4400 and End Value: 0xFFFF
4. For write protection select:
   - Type: Address
   - Access: Write
   - Action: Break
   - Action when: Inside range

NOTE: To test, reset the device via the GUI and execute GO. After a delay of approximately 5 seconds, the break occurs.

A.4 Break on Access of Invalid Memory

1. Delete or disable previous Breakpoints
2. Create a Range Breakpoint (New Breakpoint | Range...)
3. Enter Start Value: 0x1000 and End Value: 0x1800
4. For read protection select:
   - Type: Address
   - Access: Read/Write
   - Action: Break
   - Action when: Inside range

NOTE: To test, reset the device via the GUI and execute GO. After a delay of approximately 10 seconds, the break will occur.

A.5 Break if Fetch is Out of Allowed Area

1. Delete or disable previous Breakpoints
2. Create a Range Breakpoint (New Breakpoint | Range...)
3. Enter Start Value: 0x4400 and Length: 0xFFFF
4. For read protection select:
   - Type: Address
   - Access: Fetch
   - Action: Break
   - Action when: Outside range

NOTE: To test, reset the device via the GUI and execute GO. A break will occur in the Delay function.

A.6 State Storage: Trace

1. Delete or disable previous Breakpoints
2. Create an Advanced Breakpoint (New Breakpoint | Advanced...)
3. Enter Break at: 0x4440
4. Actions should be Break and State Storage Trigger
5. Open State Storage Dialog via Emulator menu to configure the state storage function
6. Switch on via Enable state storage
7. Configure Trigger Action to Stop on Trigger
8. Configure how the State Storage function should Trigger to save events- Instruction Fetch
9. Run the code. The last eight instructions before the break are stored in the buffer
   - Open State Storage Window via the Emulator Menu and examine the data after the CPU was running and is stopped.