. .eescale Semiconductor
Application Note

AN2675
Rev. 1, 11/2004

Integrated EOnCE Examples

by Iantha Scheiwe

Freescale's Enhanced On-Chip Emulation (EOnCE™) module is a powerful debugging and testing tool for monitoring StarCore™ SC140 core activity. This application note provides examples that integrate the use of multiple EOnCE detectors and register settings for specific debugging tasks. Not every programmable feature of the EOnCE is considered. Rather, the examples highlight each functional module of the EOnCE to provide a programming framework.[1] It is assumed that you are familiar with EOnCE capabilities. In addition to the code for these examples, this application note provides the equivalent Metrowerks® CodeWarrior® settings for the EOnCE configurator.

CONTENTS

1. For details on the EOnCE, consult the *SC140 DSP Core Reference Manual* (MNSC140CORE/D). Also, the application note entitled *Differences Between the EOnCE and OnCE Ports* (AN2073) shows examples for using each type of EOnCE functionality.

1 Monitoring Addresses to Enter Debug

Monitoring the SC140 address buses for an access within a specific address range is a typical use of the EOnCE for debugging. You can monitor either a single address location or a range of addresses. For the example discussed in this section, the EOnCE is programmed to watch for write accesses between address 0x0 and 0x320 inclusive and to place the SC140 core into Debug mode when this event trigger occurs. Because there are two internal address buses (XABA and XABB), you must program the EOnCE to watch for the 0x0–0x320 range for both buses.

For a single address location, a single detector can set its two comparators, A and B, equal to the watched location on XABA and XABB. However, for an address range, two of the six available event detectors are required. One detector watches the XABA bus for accesses in the specified range, and the second watches the XABB bus. Watching a single address range requires two of the six available event detectors. The example discussed in this section uses event detectors 2 and 3 for this purpose.

The Address Event Detection Channel i Control Registers (EDCAi_CTRL) can be programmed to allow watching for accesses less than, greater than, equal, or not equal to a specified address (via the EDCAi_CTRL[CBCS] and EDCAi_CTRL[CACS] bits). If a single detector were used to watch an address range, comparator A searches for accesses to addresses greater than the beginning of the range (with no upper limit) on the XABA bus. Comparator B searches for accesses less than the top of the range (with no bottom limit) on the XABB bus. The event in this case is not completely specified for both buses and can be missed or mistakenly detected. Using two detectors provides complete monitoring of both buses for the entire range.

For addresses ranges in the middle of memory, two detectors are adequate. The watched addresses are set to one less and one more than the range boundaries, and detector control is set to watch for *greater than* and *less than*, respectively. In the example discussed here, the upper bound of the range is programmed to one address location greater than the range of interest, and the control register is programmed to watch for addresses *less than* the reference address. EDCAi_REFB is programmed to 0x321 instead of 0x320. EDCAi_REFA is programmed to 0x0 and watches for accesses *greater than* 0x0. However, it misses the *equal to* 0x0 case, so an additional detector is required. In **Example 1**, event detector 1 is programmed to watch for an access equal to address 0x0. In **Example 1** three possible events are tracked:

- Address range access on XABA
- Address range access on XABB
- Address 0x0 access on XABA or XABB

Any one of these events triggers the SC140 core to enter Debug mode. Therefore, these events are ORed in the event selector control. This is the last step in the code shown in **Example 1** when the ESEL_CTRL and ESEL_DM registers are programmed.

Example 1. Monitoring Three Events to Trigger Debug Mode

```
#include "eonce.h"
#define RBA_VIA_BASE 0x00EFFE00
t_EOnCEMM *EOnCE;

void main(void)
{
   EOnCE = (t_EOnCEMM *)(RBA_VIA_BASE); /* Pointer to EOnCE memory map */

   /****************************************************/
   /* Program EOnCE External Signal Control Registers */
   /****************************************************/
   EOnCE->vusiEE_CTRL = 0x0000;         /* EE signals not in use in this example */
```

```
    /*****************************************************/
    /* Program EOnCE Event Detection Control Registers */
    /* EDCA0 not used                                    */
    /* EDCA1 watches XABA and XABB for equal to 0x0      */
    /* EDCA2 watches XABA bus for 0x0 to 0x320           */
    /* EDCA3 watches XABB bus for 0x0 to 0x320           */
    /* EDCA4 not used                                    */
    /*****************************************************/
    /* EDCA1 enabled, compare A OR B, address equal to REFA/REFB, write access, XABA&XABB */
    EOnCE->vusiEDCA1_CTRL = 0x3f06;

    /* EDCA2 enabled, compare A AND B, address<REFB/address>REFA, write access, XABA */
    EOnCE->vusiEDCA2_CTRL = 0x3ee4;

    /* EDCA3 enabled, compare A AND B, address<REFB/address>REFA, write access, XABB */
    EOnCE->vusiEDCA3_CTRL = 0x3ee5;

    /***********************************************/
    /* Program EOnCE Reference Registers and Masks */
    /***********************************************/
    /* EDCA1 checks XA and XB bus for address equal to 0x0 */
    EOnCE->vuliEDCA1_REFA = 0x00000000;
    EOnCE->vuliEDCA1_REFB = 0x00000000;
    EOnCE->vuliEDCA1_MASK = 0xffffffff;

    /* EDCA2 checks XA bus for addresses between 0x0 to 0x320 */
    EOnCE->vuliEDCA2_REFA = 0x00000000;
    EOnCE->vuliEDCA2_REFB = 0x00000321;
    EOnCE->vuliEDCA2_MASK = 0xffffffff;

    /* EDCA3 checks XB bus for addresses between 0x0 to 0x320 */
    EOnCE->vuliEDCA3_REFA = 0x00000000;
    EOnCE->vuliEDCA3_REFB = 0x00000321;
    EOnCE->vuliEDCA3_MASK = 0xffffffff;

    /***************************************/
    /* Program EOnCE Control Configuration */
    /* Enter debug mode when any of the    */
    /* programmed events occurs.           */
    /***************************************/
    EOnCE->vucESEL_CTRL = 0x00;        /* enter debug mode on trigger, OR all sources */
    EOnCE->vusiESEL_DM = 0x000e;       /* EDCA1, EDCA2, EDCA3 are trigger sources */
 };
```

1.1 EOnCE Status Register (ESR)

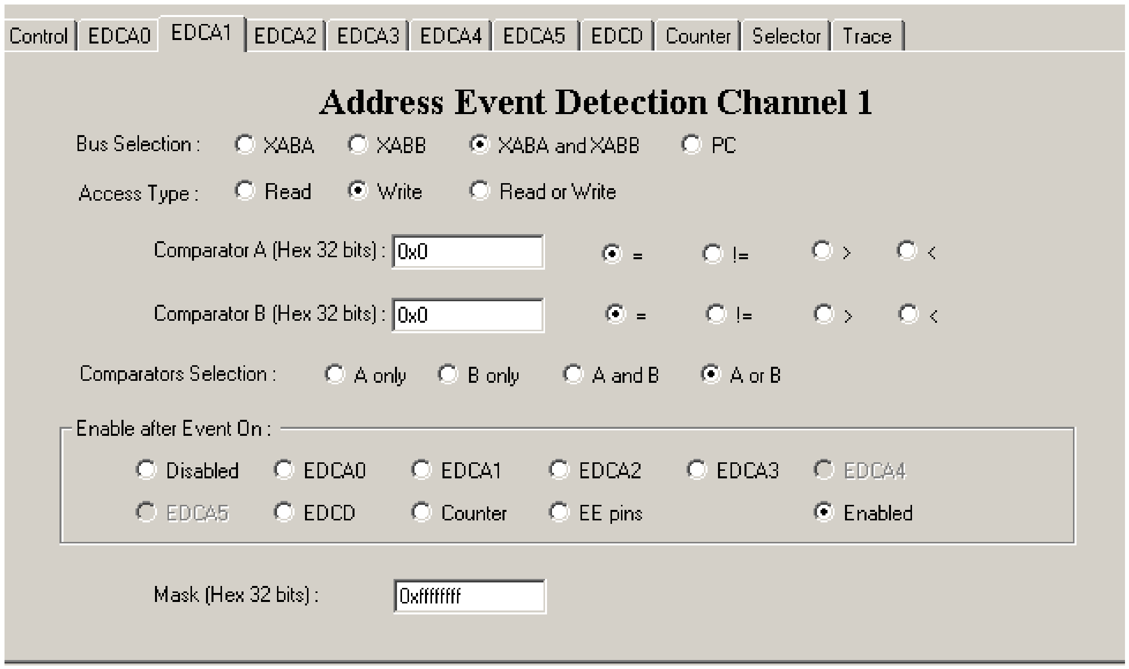
The EOnCE Status Register is extremely useful when multiple events are programmed. In **Example 1**, the events are all related to a single address range, so this register may not be as relevant. However, there are three additional event detectors. If they were to be programmed for different address ranges, it would be necessary to check the ESR to determine which event was detected and then continue debugging as necessary for that case.

When the trigger occurs, the ESR bit for the trigger event changes to a value of 1. For **Example 1**, if the access occurs to address 0x0, ESR = 0xC0nn0002. This value indicates that the SC140 core has entered Debug mode as triggered by event detector 1. If the access occurs in the programmed range on XABA, the ESR = 0xC0nn0004. If the access occurs in the range on XABB, ESR = 0xC0nn0008.

Note: The *nn* value shown in the ESR is SC140 core revision dependent, so a specific value is not shown here. ESR bit values are described in the *SC140 DSP Core Reference Manual*.

1.2 CodeWarrior EOnCE Configuration

CodeWarrior includes an EOnCE configurator to give access to all EOnCE programming parameters via the CodeWarrior graphical user interface (GUI). In addition, the configurator allows you to save a defined configuration and bring it up again for future debug sessions. The steps required to set the EOnCE parameters using the EOnCE configurator shown in this section are equivalent to the steps in the code shown in **Example 1**. If the code is used, you do not need to use the configurator. **Figure 1** shows that EDCA1 watches for write accesses to XABA and XABB at address 0x0. **Figure 2** shows that EDCA2 watches for write accesses on XABA to addresses greater than 0x0 to less than 0x321. **Figure 3** shows that EDCA3 is configured to watch XABB for write accesses to addresses greater than 0x0 and less than 0x321. **Figure 4** shows that events 1, 2, and 3 are ORed to put the SC140 core into Debug mode. No other events are configured. When the steps shown in **Figure 1** through **Figure 4** are complete, the configuration for **Example 1** is complete. The settings in the remaining EOnCE configurator tabs can remain in their default states.



Address Event Detection Channel 1

Bus Selection : XABA XABB [✓ XABA and XABB] PC

Access Type : Read [✓ Write] Read or Write

Comparator A (Hex 32 bits) : 0x0 [✓ =] != > <

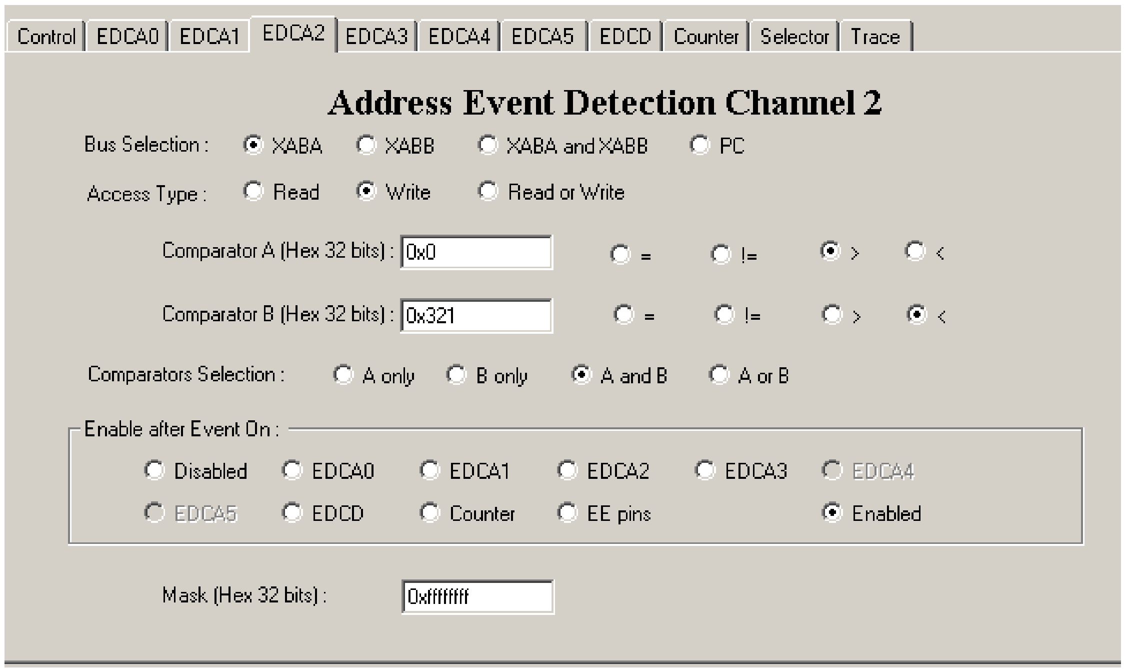
Comparator B (Hex 32 bits) : 0x0 [✓ =] != > <

Comparators Selection : A only B only A and B [✓ A or B]

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

Figure 1. Configure EDCA1 for Example 1



Address Event Detection Channel 2

Bus Selection : [✓ XABA] XABB XABA and XABB PC

Access Type : Read [✓ Write] Read or Write

Comparator A (Hex 32 bits) : 0x0 = != [✓ >] <

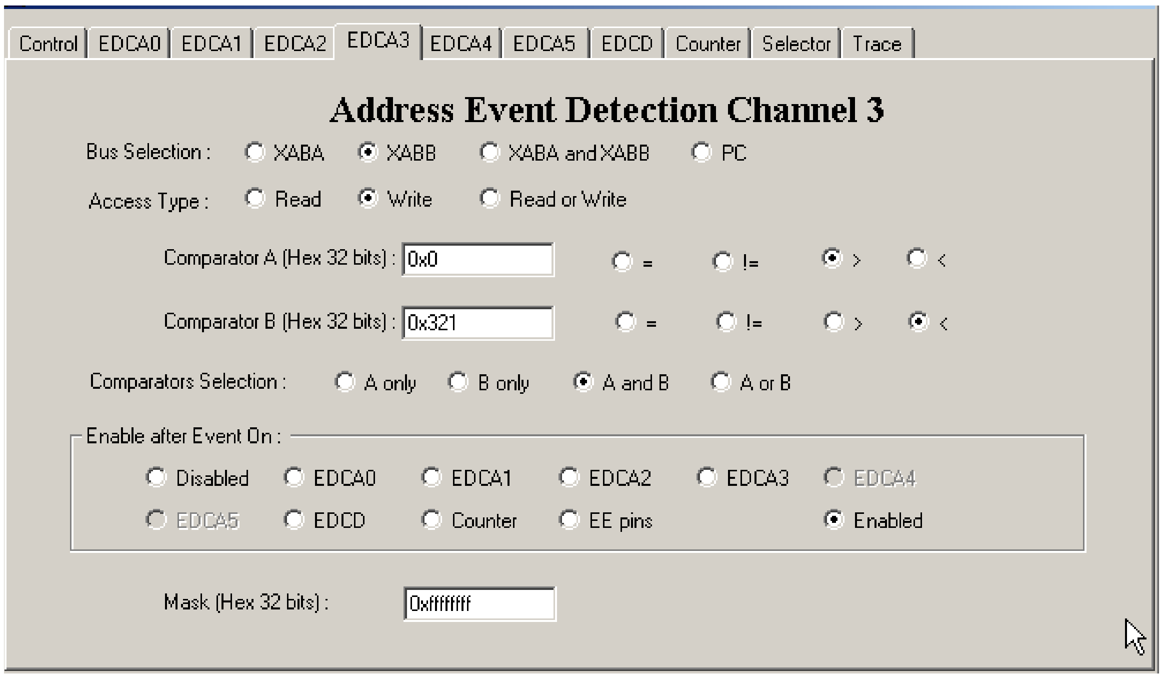
Comparator B (Hex 32 bits) : 0x321 = != > [✓ <]

Comparators Selection : A only B only [✓ A and B] A or B

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

Figure 2. Configure EDCA2



Address Event Detection Channel 3

Bus Selection : XABA [✓ XABB] XABA and XABB PC

Access Type : Read [✓ Write] Read or Write

Comparator A (Hex 32 bits) : 0x0 = != [✓ >] <

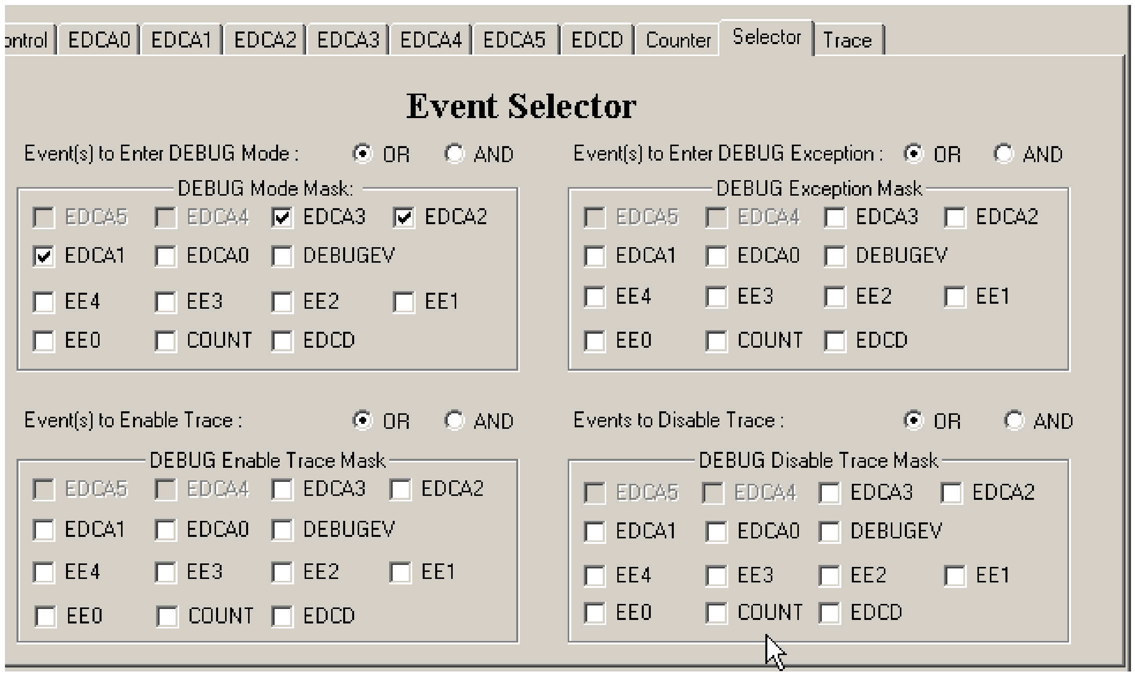
Comparator B (Hex 32 bits) : 0x321 = != > [✓ <]

Comparators Selection : A only B only [✓ A and B] A or B

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

Figure 3. Configure EDCA3


ontrol | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace

Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask:

EDCA5	EDCA4	[✓ EDCA3]	[✓ EDCA2]
[✓ EDCA1]	EDCA0	DEBUGEV	
EE4	EE3	EE2	EE1
EE0	COUNT	EDCD	

Event(s) to Enter DEBUG Exception : [✓ OR] AND

DEBUG Exception Mask

EDCA5	EDCA4	EDCA3	EDCA2
EDCA1	EDCA0	DEBUGEV	
EE4	EE3	EE2	EE1
EE0	COUNT	EDCD	

Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask

EDCA5	EDCA4	EDCA3	EDCA2
EDCA1	EDCA0	DEBUGEV	
EE4	EE3	EE2	EE1
EE0	COUNT	EDCD	

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask

EDCA5	EDCA4	EDCA3	EDCA2
EDCA1	EDCA0	DEBUGEV	
EE4	EE3	EE2	EE1
EE0	COUNT	EDCD	

Figure 4. Configure Event Selector for Example 1

2 Monitoring an Address and Data Combination to Cause a Debug Exception

Sometimes it is necessary to monitor more than just the address of a transaction. It may be necessary to monitor when an address is accessed with a particular data value. For the example discussed in this section, the combination is the long value 0x12345678 written to address location 0x70000. Since a single address is monitored, a single address event detector is used, EDCA0. In addition, the data event detection channel (EDCD) must be used.

In some cases, you may not want the system to enter Debug mode when a trigger event occurs. It may be necessary to keep the device running and execute additional code to handle the case when it occurs. In this case, a debug exception can be used by programming an interrupt service routine (ISR) at vector base address (VBA) offset 0xC0 to run when the triggered event occurs. 0xC0 is the debug exception vector address. DEBUG_ISR is included at the end of the code. The code in the ISR is application-dependent. To demonstrate what can be done in the ISR, the code in **Example 2** clears the EOnCE monitor and control register (EMCR) and then performs a simple move before returning to normal application flow. In addition to programming the ISR, using a Debug exception requires initialization of stack pointers, the status register, and VBA.

The data event detector can trigger on a byte access, word access, or long access, according to the setting of the EDCD_CTRL[AWS] field. For the EOnCE a word is 16 bits and a long access is 32 bits. If the data event detection channel (EDCD) is programmed to trigger on a word access of value 0x1234, but a long access of 0x56781234 occurs, the event does not trigger. If a double word access (move.2w) of 0x5678,0x1234 (0x56781234 on the bus) occurs, the event triggers. The EOnCE compares the 16 least significant bits of the reference register with both words of the transaction. The access width trigger occurs when the programmed EOnCE access width and the access transaction type are the same.

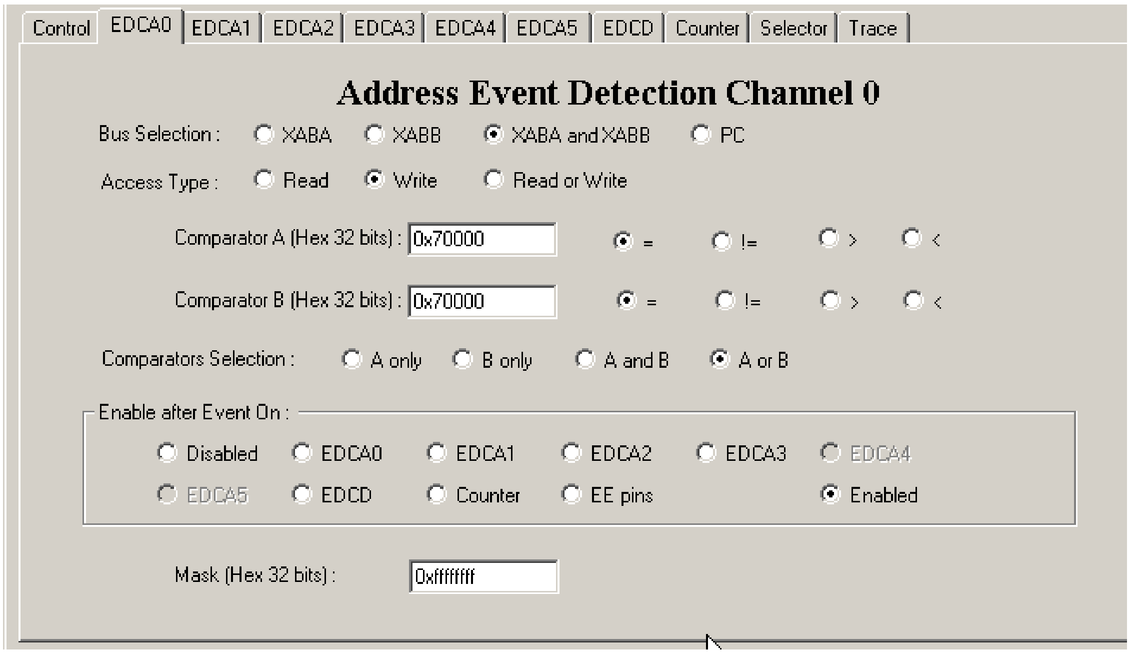
In contrast to **Example 1**, which triggers an entry to Debug mode when any one of the three programmed events occurs (OR), **Example 2** triggers a debug exception when all programmed events occur simultaneously (AND). The event selector for debug exception (ESEL_CTRL[SELDI]) is programmed to AND the sources programmed in ESEL_DI.

Note: Because **Example 2** uses none of the same event detectors as **Example 1** and the examples enable different trigger reactions (**Example 1** enters Debug mode, and **Example 2** jumps to a debug exception), we can enable the previous memory range check and this example's address/data combination check together. After both pieces of code are combined, the only change required is to use the ESEL_CTRL value programmed for **Example 2**. Therefore, two trigger events with different trigger reactions can be enabled for a single application.

Example 2. Debug Exception

```
#include "msc8101.h"
#include "eonce.h"
#define QBUS_BASE 0x00F00000
#define RBA_VIA_BASE 0x00EFFE00
t_QBusMM *pstQbus;
t_EOnCEMM *EOnCE;

void main(void)
{
   pstQbus = (t_QBusMM *)(QBUS_BASE);    /* MSC8101 internal register map */
   EOnCE = (t_EOnCEMM *)(RBA_VIA_BASE); /* Pointer to EOnCE memory map */

   /* Initialize Interrupts */
   asm(" di ");
   asm(" move.l #$5000,vba ");
   asm(" move.l #$68000,r7 ");
   asm(" move.l #$62000,r6 ");
   asm(" bmclr #$00FC,sr.h ");         /* Allow all interrupt levels */
   asm(" tfra r6,osp ");               /* Set ESP to $62000 */
   asm(" tfra r7,sp ");

   /* Enable interrupts */
   asm(" ei ");

   /****************************************************/
              /* Program EOnCE External Signal Control Registers */
   /****************************************************/
   EOnCE->vusiEE_CTRL = 0x0000;         /* EE signals not in use in this example */

   /****************************************************/
   /* Program EOnCE Event Detection Control Registers */
   /* EDCA0 wathces XABA and XABB for equal to 0x70000*/
   /* EDCD checks for data writes of 0x12345678       */
   /****************************************************/
   /* Long transaction, enabled, equal to data, write access */
   EOnCE->vusiEDCD_CTRL = 0x0279;

   /* EDCA0 enabled, compare A OR B, address equal to REFA/REFB, write access, XABA&XABB */
   EOnCE->vusiEDCA0_CTRL = 0x3f06;

   /************************************************/
   /* Program EOnCE Reference Registers and Masks */
   /************************************************/
   /* Check data buses */
   EOnCE->vuliEDCD_REF = 0x12345678;
   EOnCE->vuliEDCD_MASK = 0xffffffff;

   /* EDCA0 checks access to pointer location */
   EOnCE->vuliEDCA0_REFA = 0x00070000;
   EOnCE->vuliEDCA0_REFB = 0x00070000;
   EOnCE->vuliEDCA0_MASK = 0xffffffff;

   /****************************************/
   /* Program EOnCE Control Configuration */
   /* Enter debug mode when any of the    */
   /* programmed events occurs.           */
   /****************************************/
   /* Debug exception on trigger - AND all DI sources */
   EOnCE->vucESEL_CTRL = 0x02;
   EOnCE->vusiESEL_DI = 0x0101;         /* EDCA0, EDCD trigger sources for exception */
};
--------------------------------
   org $50c0
DEBUG_ISR
   clr d0
       move.l d0,EMCR
   move.l (r0),d0    ; when event is triggered move data from location at r0 to d0
   rte
```

The steps to set the EOnCE parameters using the EOnCE configurator for **Example 2** are equivalent to the steps in the code for **Example 2**. If the code is used, then you do not need to use the configurator. **Figure 5** shows that EDCA0 watches for write accesses to XABA and XABB at address 0x70000. **Figure 6** shows that EDCD watches for long write accesses equal to 0x12345678. **Figure 7** shows that the event selector is configured to enter Debug exception mode when both EDCA0 and EDCD occur.



Address Event Detection Channel 0

Bus Selection : XABA XABB [✓ XABA and XABB] PC

Access Type : Read [✓ Write] Read or Write

Comparator A (Hex 32 bits) : 0x70000 [✓ =] != > <

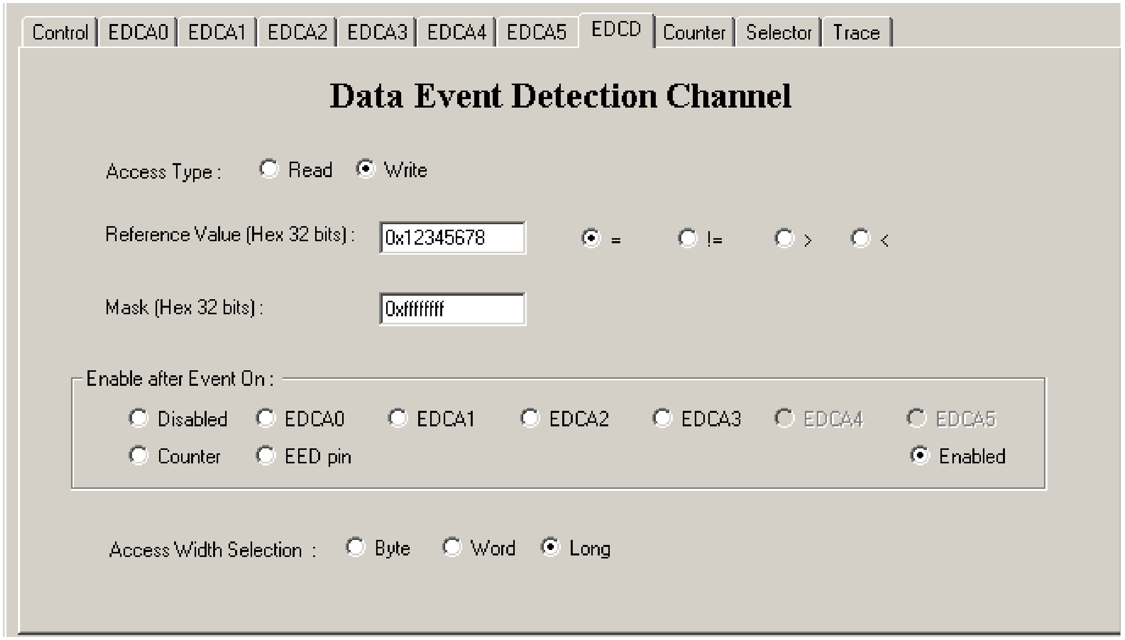
Comparator B (Hex 32 bits) : 0x70000 [✓ =] != > <

Comparators Selection : A only B only A and B [✓ A or B]

Enable after Event On : Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins [✓ Enabled]

Mask (Hex 32 bits) : 0xffffffff

Figure 5. Configure EDCA0



Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | [✓ EDCD] | Counter | Selector | Trace

Data Event Detection Channel

Access Type : Read [✓ Write]

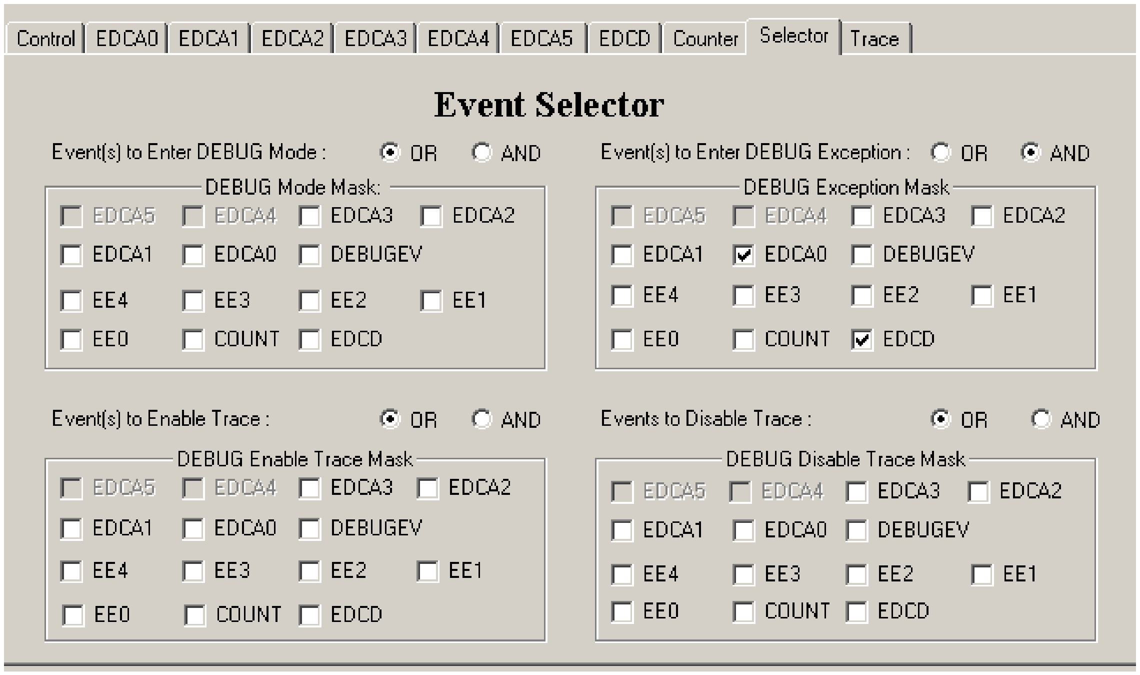
Reference Value (Hex 32 bits) : 0x12345678 [✓ =] != > <

Mask (Hex 32 bits) : 0xffffffff

Enable after Event On :
Disabled EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5
Counter EED pin [✓ Enabled]

Access Width Selection : Byte Word [✓ Long]

Figure 6. Configure EDCD



Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | [✓ Selector] | Trace

Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask:
EDCA5 EDCA4 EDCA3 EDCA2
EDCA1 EDCA0 DEBUGEV
EE4 EE3 EE2 EE1
EE0 COUNT EDCD

Event(s) to Enter DEBUG Exception : OR [✓ AND]

DEBUG Exception Mask
EDCA5 EDCA4 EDCA3 EDCA2
EDCA1 [✓ EDCA0] DEBUGEV
EE4 EE3 EE2 EE1
EE0 COUNT [✓ EDCD]

Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask
EDCA5 EDCA4 EDCA3 EDCA2
EDCA1 EDCA0 DEBUGEV
EE4 EE3 EE2 EE1
EE0 COUNT EDCD

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask
EDCA5 EDCA4 EDCA3 EDCA2
EDCA1 EDCA0 DEBUGEV
EE4 EE3 EE2 EE1
EE0 COUNT EDCD

Figure 7. Configure Event Selector for Example 2

3 Chaining Events and Using the Event Counter

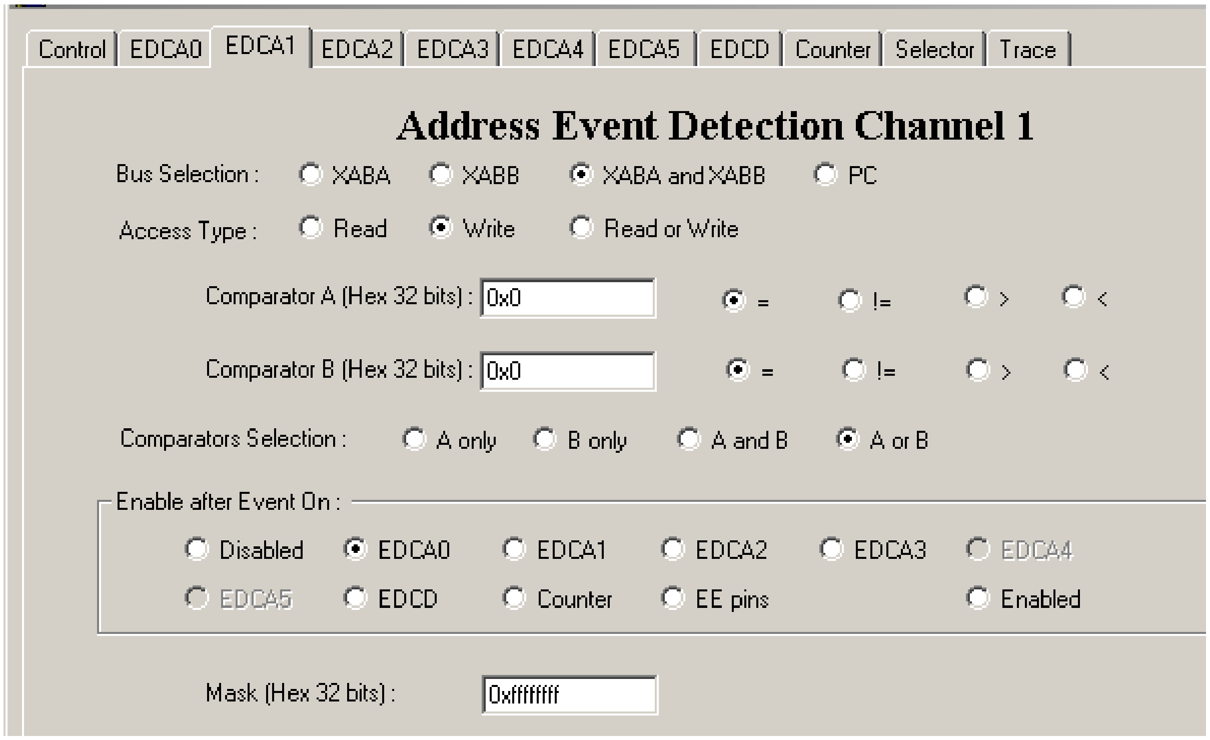
This section presents an example in which we program the EOnCE so that the SC140 core counts the number of times the address 0x0 is accessed after a pointer at location 0x70000 is written. This example uses some address detector settings from previous examples and adds EOnCE event chaining and counting. It uses the event detector reference settings for EDCA0 and EDCA1 from **Example 1** and **Example 2**. Changes in this example are required in the event detector control registers, which allow an event to occur after a previous event so that the events can be chained. Here, the first event is EDCA0 (writing to pointer at location 0x70000). The subsequent events is EDCA1 (write to address 0x0). An event detector can be enabled after an event on any of the other detectors (EDCA0–5, EDCD). When chaining EOnCE events, ensure that the events do not occur in the same core clock cycle because if this happens, the chained event is ignored.

Example 3 does not place the SC140 core into Debug mode or cause a debug exception. Therefore, no event combinations are required in the event selector control register. Instead, the EOnCE counter is used to count the number of times that EDCA1 occurs, as programmed in the ECNT_CTRL register. The EOnCE Counter Value Register (ECNT_VAL) should be initialized before the application is started. The application can then check the count value periodically, if desired, and act accordingly. Note that ECNT_VAL is a down-counter with a maximum value of 0x7FFFFFFF counting down to 0x0. An extension to the counter register is available if additional counting is required, but the extension is not used in this example.

Example 3. Event Chaining and Counting

```
#include "eonce.h"
#define RBA_VIA_BASE 0x00EFFE00
t_EOnCEMM *EOnCE;
void main(void)
{
   EOnCE = (t_EOnCEMM *)(RBA_VIA_BASE); /* Pointer to EOnCE memory map */

   /****************************************************/
   /* Program EOnCE External Signal Control Registers */
   /****************************************************/
   EOnCE->vusiEE_CTRL = 0x0000;         /* EE signals not in use in this example */

   /****************************************************/
   /* Program EOnCE Event Detection Control Registers */
   /* EDCA0 watches XABA and XABB for equal to 0x70000*/
   /* EDCA1 watches XABA and XABB for equal to 0x0    */
   /****************************************************/
   /* EDCA0 enabled, compare A OR B, address = REFA/REFB, write access, XABA&XABB */
   EOnCE->vusiEDCA0_CTRL = 0x3f06;

   /* EDCA1 enabled after event 0, compare A OR B, address=REFA/REFB, write, XABA&XABB */
   EOnCE->vusiEDCA1_CTRL = 0x0706;

   /************************************************/
   /* Program EOnCE Reference Registers and Masks */
   /************************************************/
   /* EDCA0 checks access to pointer location */
   EOnCE->vuliEDCA0_REFA = 0x00070000;
   EOnCE->vuliEDCA0_REFB = 0x00070000;
   EOnCE->vuliEDCA0_MASK = 0xffffffff;

   /* EDCA1 checks XA and XB bus for address equal to 0x0 */
   EOnCE->vuliEDCA1_REFA = 0x00000000;
   EOnCE->vuliEDCA1_REFB = 0x00000000;
   EOnCE->vuliEDCA1_MASK = 0xffffffff;

   /****************************************/
   /* Program EOnCE Counter Control       */
   /* Count EDCA1 events.                 */
   /****************************************/
   /* EOnCE counter is enabled, counts EDCA1 occurrences */
   EOnCE->vusiECNT_CTRL = 0x00f1;
   EOnCE->vuliECNT_VAL = 0x7fffffff;
};
```

Figure 5 shows the configuration for EDCA0 in the EOnCE Configurator. It is unchanged for **Example 3**. **Figure 8** shows the configuration for EDCA1. Note that only the selection for “Enable after event on” differs from **Figure 1**. The event selector is not used for this example since the SC140 core does not enter debug mode, debug exception, or enable the trace buffer. **Figure 9** shows the EOnCE event counter configuration for **Example 3**.



Address Event Detection Channel 1

Bus Selection : XABA XABB [✓ XABA and XABB] PC

Access Type : Read [✓ Write] Read or Write

Comparator A (Hex 32 bits) : 0x0 [✓ =] != > <

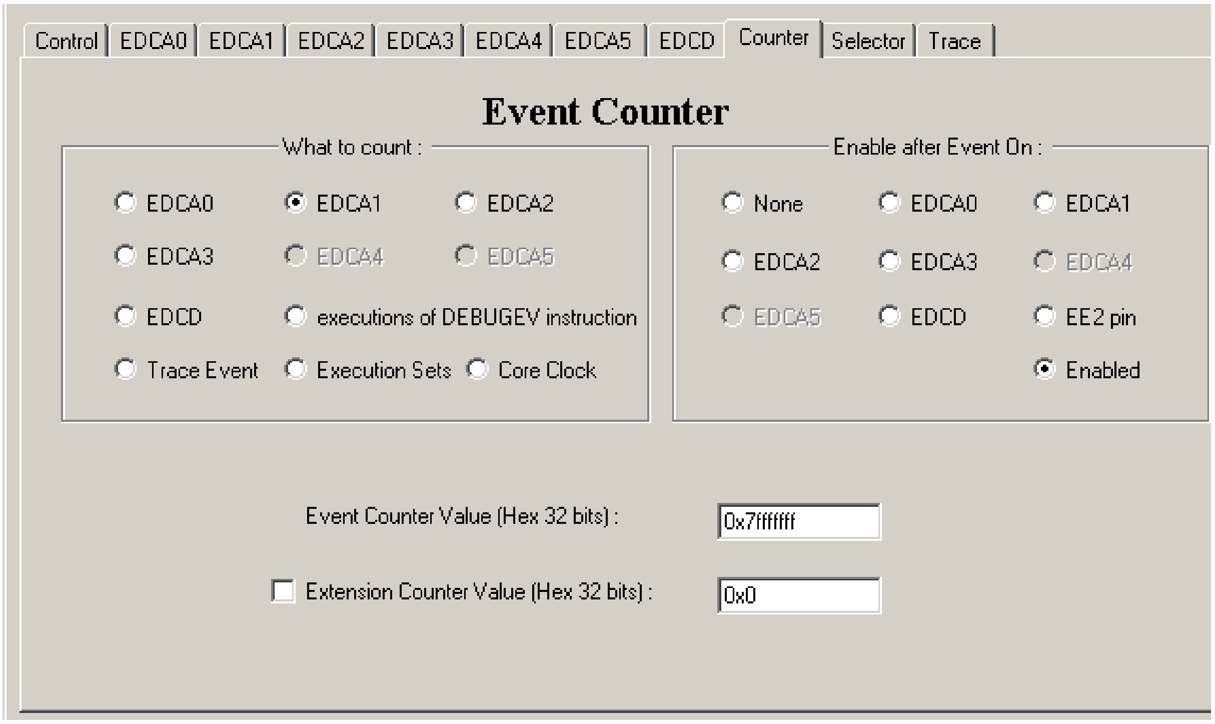
Comparator B (Hex 32 bits) : 0x0 [✓ =] != > <

Comparators Selection : A only B only A and B [✓ A or B]

Enable after Event On : Disabled [✓ EDCA0] EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD Counter EE pins Enabled

Mask (Hex 32 bits) : 0xffffffff

Figure 8. Configure EDCA1 for Example 3



Event Counter

What to count : EDCA0 [✓ EDCA1] EDCA2 EDCA3 EDCA4 EDCA5 EDCD executions of DEBUGEV instruction Trace Event Execution Sets Core Clock

Enable after Event On : None EDCA0 EDCA1 EDCA2 EDCA3 EDCA4 EDCA5 EDCD EE2 pin [✓ Enabled]

Event Counter Value (Hex 32 bits) : 0x7fffffff

☐ Extension Counter Value (Hex 32 bits) : 0x0

Figure 9. Configure EOnCE Counter

4 Using the Trace Buffer and External EE Signals

In addition to event detection and counting, the EOnCE provides a trace buffer and the ability to trigger on external signal activity, as described in this section. **Example 4** enables the trace buffer after the chained events in **Example 3** (write to address 0x70000 followed by a write to address 0x0). The trace buffer is programmed to trace the issue of execution sets after it is enabled. The trace buffer is disabled when an event on EE2 is detected. The trace buffer stores addresses based on its programming. It can be programmed to store addresses of change-of-flow instructions, interrupts, hardware loops, and/or execution sets. The Trace Buffer Control Register (TB_CTRL) defines trace buffer operation. The simplest method for disabling the trace buffer is to clear the TB_CTRL[TEN] bit. However, the EE2 signal is used here to provide an example of using an external event on an EE signal to disable the buffer.

The EOnCE has six external signals, EE[0–5], that can be used as event triggers for EOnCE monitoring or as outputs after the EOnCE detects an internal event. For **Example 4**, the EE2 signal acts as an input to trigger EOnCE events (disable trace buffer). Operation of the EE signals is defined in the EE_CTRL register.

The EDCA0 event enables EDCA1, which enables the trace buffer and is programmed as shown in **Example 2**. EE2 can disable the trace buffer directly without use of event detector 2.

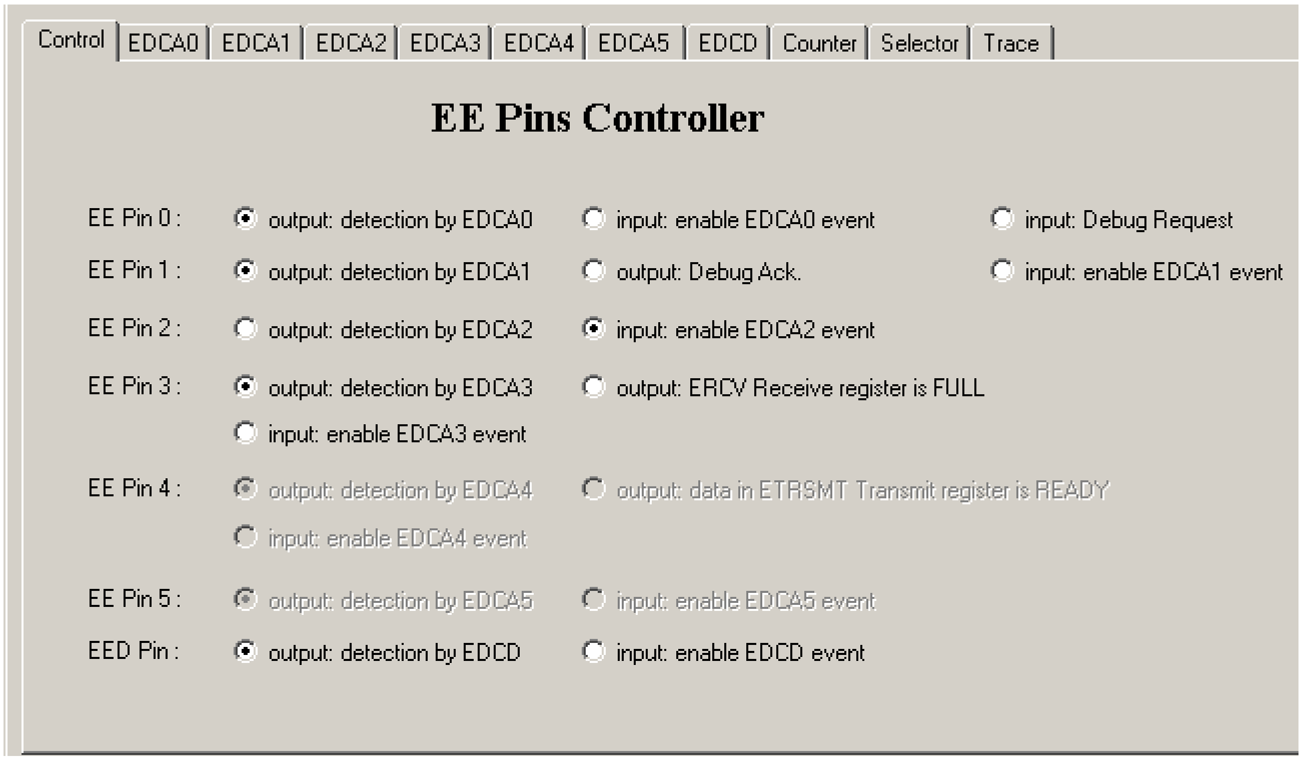
The event selector is used in **Example 4** to define which events enable and disable the trace buffer. The Event Selector Mask Enable Trace Buffer Register (ESEL_ETB) defines the events that enable the trace buffer. The Event Selector Mask Disable Trace Buffer Register (ESEL_DTB) defines the events that disable the trace buffer. The Event Selector Control Register (ESEL_CTRL) defines whether the events are ORed or ANDed similar to previous examples.

Example 4. Enable/Disable the Trace Buffer

```
#include "eonce.h"
#define RBA_VIA_BASE 0x00EFFE00
t_EOnCEMM *EOnCE;

void main(void)
{
   EOnCE = (t_EOnCEMM *)(RBA_VIA_BASE); /* Pointer to EOnCE memory map */

   /***************************************************/
   /* Program EOnCE External Signal Control Registers */
   /***************************************************/
   EOnCE->vusiEE_CTRL = 0x0030;         /* EE signal 2 is input */

   /***************************************************/
   /* Program EOnCE Event Detection Control Registers */
   /* EDCA0 triggered on write to location 0x70000    */
   /***************************************************/
   /* EDCA0 enabled, compare A OR B, address = REFA/REFB, write access, XABA & XABB */
   EOnCE->vusiEDCA0_CTRL = 0x3f06;
   /*Enable EDCA1 after event0,compare A OR B,address=REFA/REFB,write access,XABA & XABB*/
   EOnCE->vusiEDCA1_CTRL = 0x0706;

   /* EDCA0 checks access to pointer location */
   EOnCE->vuliEDCA0_REFA = 0x00070000;
   EOnCE->vuliEDCA0_REFB = 0x00070000;
   EOnCE->vuliEDCA0_MASK = 0xffffffff;
/* EDCA1 checks XA and XB bus for address equal to 0x0 */
   EOnCE->vuliEDCA1_REFA = 0x00000000;
   EOnCE->vuliEDCA1_REFB = 0x00000000;
```

```
        EOnCE->vuliEDCA1_MASK = 0xffffffff;

        /****************************************/
        /* Program Trace Buffer Control        */
        /****************************************/
        EOnCE->vucTB_CTRL = 0x04;            /* trace issue of execution sets */

        /****************************************/
        /* Program EOnCE Control Configuration */
        /* Enter debug mode when any of the    */
        /* programmed events occurs.           */
        /****************************************/
        EOnCE->vucESEL_CTRL = 0x00;          /* OR all sources */
        EOnCE->vusiESEL_ETB = 0x0002;        /* EDCA1 enables trace buffer */
        EOnCE->vusiESEL_DTB = 0x1000;        /* EE2 disables trace buffer */
    };
```

Figure 5 shows the EDCA0 EOnCE configurator settings for **Example 4**. **Figure 8** shows the EDCA1 EOnCE configurator settings for this example. **Figure 10** shows the EOnCE configurator control panel. This panel defines EE signal functionality. It shows that EE2 is an input. **Figure 11** shows the event selector. We select EDCA1 to enable the trace buffer and specify that EE2 disables the trace buffer. **Figure 12** shows the trace buffer configuration. We want to trace issuance of execution sets only for this example.



EE Pins Controller

EE Pin 0 : [✓ output: detection by EDCA0] input: enable EDCA0 event input: Debug Request

EE Pin 1 : [✓ output: detection by EDCA1] output: Debug Ack. input: enable EDCA1 event

EE Pin 2 : output: detection by EDCA2 [✓ input: enable EDCA2 event]

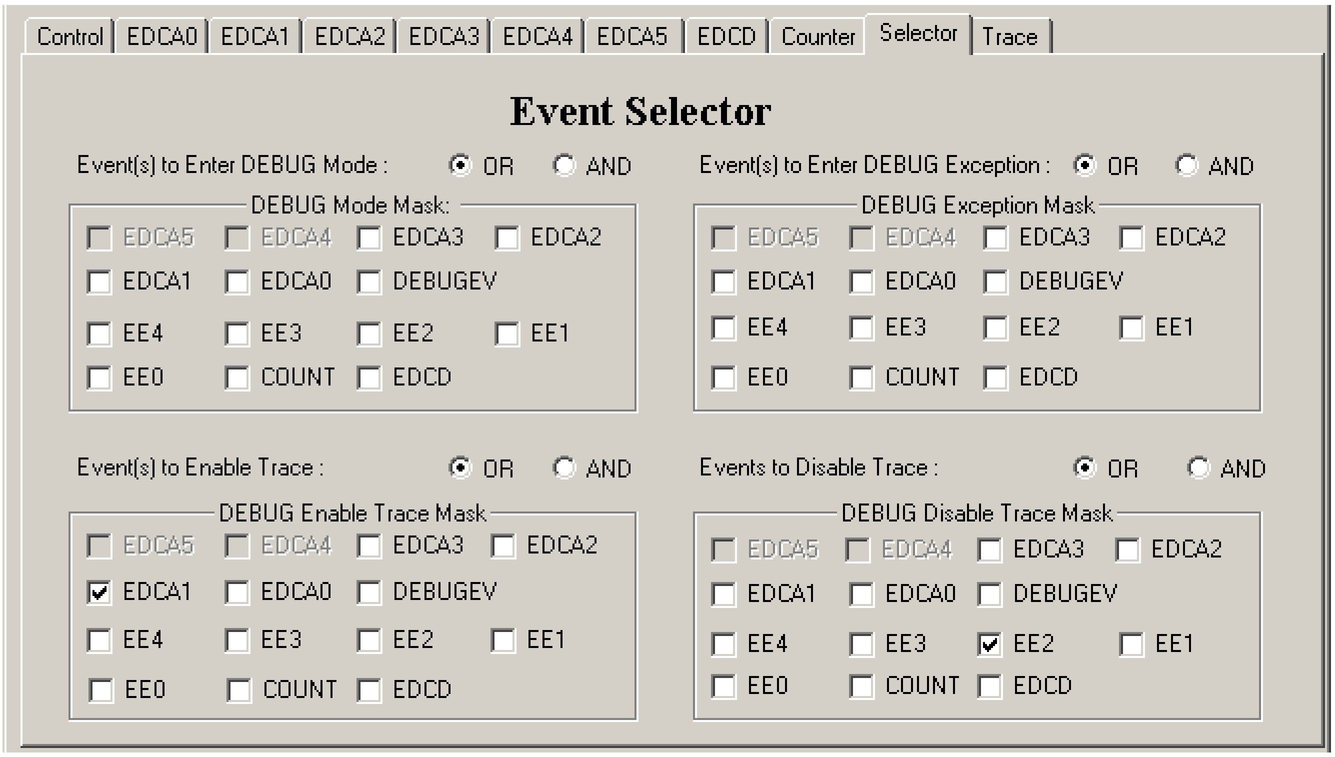
EE Pin 3 : [✓ output: detection by EDCA3] output: ERCV Receive register is FULL input: enable EDCA3 event

EE Pin 4 : [✓ output: detection by EDCA4] output: data in ETRSMT Transmit register is READY input: enable EDCA4 event

EE Pin 5 : [✓ output: detection by EDCA5] input: enable EDCA5 event

EED Pin : [✓ output: detection by EDCD] input: enable EDCD event

Figure 10. Configure EOnCE Signal Control



Event Selector

Event(s) to Enter DEBUG Mode : [✓ OR] AND

DEBUG Mode Mask: EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

Event(s) to Enter DEBUG Exception : [✓ OR] AND

DEBUG Exception Mask: EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

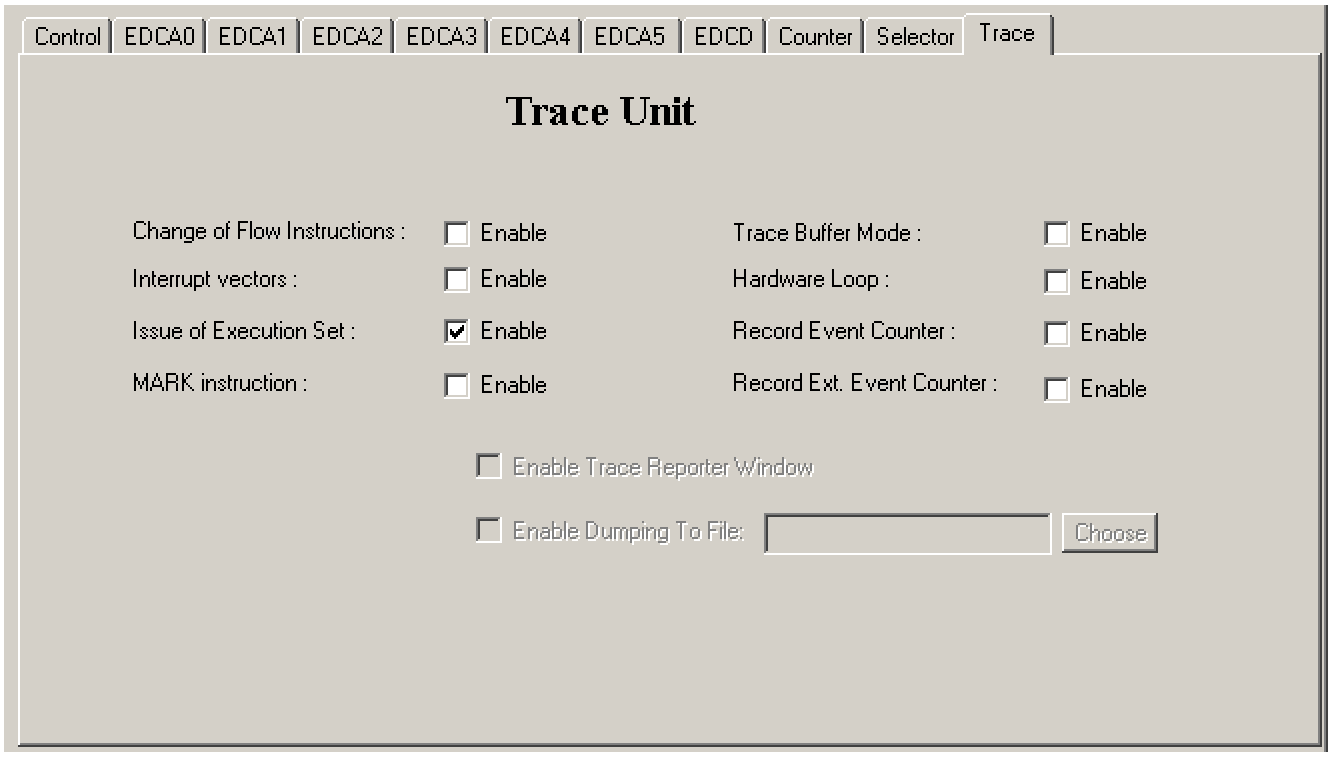
Event(s) to Enable Trace : [✓ OR] AND

DEBUG Enable Trace Mask: EDCA5, EDCA4, EDCA3, EDCA2, [✓ EDCA1], EDCA0, DEBUGEV, EE4, EE3, EE2, EE1, EE0, COUNT, EDCD

Events to Disable Trace : [✓ OR] AND

DEBUG Disable Trace Mask: EDCA5, EDCA4, EDCA3, EDCA2, EDCA1, EDCA0, DEBUGEV, EE4, EE3, [✓ EE2], EE1, EE0, COUNT, EDCD

Figure 11. Configure Event Selector for Example 4



Trace Unit

Change of Flow Instructions :	Enable	Trace Buffer Mode :	Enable
Interrupt vectors :	Enable	Hardware Loop :	Enable
Issue of Execution Set :	[✓ Enable]	Record Event Counter :	Enable
MARK instruction :	Enable	Record Ext. Event Counter :	Enable

Enable Trace Reporter Window

Enable Dumping To File: Choose

Figure 12. Configure Trace Buffer

Trace buffer contents are read via the trace buffer register, TB_BUFF. The trace buffer can trace additional types of core actions and can be triggered by different types of internal and external events than the ones shown here. **Example 4** provides a simple framework for initializing the EOnCE to perform SC140 core monitoring tasks. Note that when the trace buffer is enabled, its contents are reset. Therefore, if the chained events 0 and 1 occur again to enable the trace buffer, the contents of the previous trace are lost.

5 Include File, eonce.h

The examples in this application note include the `eonce.h` file, which is shown in this section. `eonce.h` includes another file, `typedefs.h`, which is in the Metrowerks CodeWarrior stationery, along with the `msc8101.h` file used in **Example 2**. Those files are not shown in this application note.

```
/****************************************************************************
* Property of Freescale, RF&DSP Infrastructure Division
*****************************************************************************
* ANSI C source code
*
* FREESCALE GENERAL BUSINESS INFORMATION
*****************************************************************************/
/*****************************************************************************
* MODULE NAME: eonce.h
* VERSION:     v0.0, 21 Nov 2003.
* COMPILER:    Metrowerks, CodeWarrior for Starcore
* TARGET:      SC140
* REVISION HISTORY:
* 21 Nov 2003  ies    initial version
*****************************************************************************/
#ifndef _EOnCE_H
#define _EOnCE_H
/*****************************************************************************
* Include files
*****************************************************************************/
#include "typedefs.h"
/*------------------------------------------------------------------------*/
/*                DEFINITION OF EOnCE MEMORY MAP                          */
/*------------------------------------------------------------------------*/
/*****************************************************************************
* EOnCE Registers
*
* EOnCE registers are defined in a data structure called "EOnCE_MM".  This
* is not part of the t_8101IMM data structure and should thus not affect it.
* To access the EOnCE registers using this structure do the following:
*
* 1. Declare a global EOnCE pointer (just as the IMM pointer).
*    t_EOnCEMM *EOnCE;                      // EOnCE pointer
*
* 2. Initialize the EOnCE pointer (just as you would the IMM pointer):
*    EOnCE = (t_EOnCEMM *)(RBA_VIA_BASE);   // Pointer to EOnCE registers
* where
*    #define RBA_VIA_BASE 0x00EFFE00        // EOnCE map base address for MSC8101
*
* 3. Access the EOnCE registers in the application code using pointer to the
* EOnCE structure, example:
*   pstEOnCE->vuliEMCR = 0x00000000;
*   pstEOnCE->vusiEE_CTRL = 0x0003;
*   pstEOnCE->vucESEL_CTRL = 0x02;
*****************************************************************************/
```

```
typedef struct EOnCEMM
{
    /* EOnCE Registers */
    VUWord32 vuliESR;                /* EOnCE status register */
    VUWord32 vuliEMCR;               /* EOnCE Monitor and Control Register */
    VUWord32 vuliERCV0;              /* EOnCE Receive Register LSB */
    VUWord32 vuliERCV1;              /* EOnCE Receive Register MSB */
    VUWord32 vuliETRSMT0;            /* EOnCE Receive Register LSB */
    VUWord32 vuliETRSMT1;            /* EOnCE Receive Register MSB */
    VUWord16 vusiEE_CTRL;            /* EOnCE EE pins Control Register */
    VUByte   avucReservedE1[2];
    VUWord32 vuliPC_EXCP;            /* EOnCE Exception PC Register */
    VUWord32 vuliPC_NEXT;            /* EOnCE PC of Next Execution set */
    VUWord32 vuliPC_LAST;            /* EOnCE PC of Last Execution set */
    VUWord32 vuliPC_DETECT;          /* EOnCE PC Breakpoint Detection Register */
    VUByte   avucReservedE2[20];
    VUWord16 vusiEDCA0_CTRL;         /* EOnCE EDCA #0 Control Register */
    VUByte   avucReservedE3[2];
    VUWord16 vusiEDCA1_CTRL;         /* EOnCE EDCA #1 Control Register */
    VUByte   avucReservedE4[2];
    VUWord16 vusiEDCA2_CTRL;         /* EOnCE EDCA #2 Control Register */
    VUByte   avucReservedE5[2];
    VUWord16 vusiEDCA3_CTRL;         /* EOnCE EDCA #3 Control Register */
    VUByte   avucReservedE6[2];
    VUWord16 vusiEDCA4_CTRL;         /* EOnCE EDCA #4 Control Register */
    VUByte   avucReservedE7[2];
    VUWord16 vusiEDCA5_CTRL;         /* EOnCE EDCA #5 Control Register */
    VUByte   avucReservedE8[10];
    VUWord32 vuliEDCA0_REFA;         /* EOnCE EDCA #0 Reference Value A */
    VUWord32 vuliEDCA1_REFA;         /* EOnCE EDCA #1 Reference Value A */
    VUWord32 vuliEDCA2_REFA;         /* EOnCE EDCA #2 Reference Value A */
    VUWord32 vuliEDCA3_REFA;         /* EOnCE EDCA #3 Reference Value A */
    VUWord32 vuliEDCA4_REFA;         /* EOnCE EDCA #4 Reference Value A */
    VUWord32 vuliEDCA5_REFA;         /* EOnCE EDCA #5 Reference Value A */
    VUByte   avucReservedE9[8];
    VUWord32 vuliEDCA0_REFB;         /* EOnCE EDCA #0 Reference Value B */
    VUWord32 vuliEDCA1_REFB;         /* EOnCE EDCA #1 Reference Value B */
    VUWord32 vuliEDCA2_REFB;         /* EOnCE EDCA #2 Reference Value B */
    VUWord32 vuliEDCA3_REFB;         /* EOnCE EDCA #3 Reference Value B */
    VUWord32 vuliEDCA4_REFB;         /* EOnCE EDCA #4 Reference Value B */
    VUWord32 vuliEDCA5_REFB;         /* EOnCE EDCA #5 Reference Value B */
    VUByte   avucReservedE10[40];
    VUWord32 vuliEDCA0_MASK;         /* EOnCE EDCA #0 Mask Register */
    VUWord32 vuliEDCA1_MASK;         /* EOnCE EDCA #1 Mask Register */
    VUWord32 vuliEDCA2_MASK;         /* EOnCE EDCA #2 Mask Register */
    VUWord32 vuliEDCA3_MASK;         /* EOnCE EDCA #3 Mask Register */
    VUWord32 vuliEDCA4_MASK;         /* EOnCE EDCA #4 Mask Register */
    VUWord32 vuliEDCA5_MASK;         /* EOnCE EDCA #5 Mask Register */
    VUByte   avucReservedE11[8];
    VUWord16 vusiEDCD_CTRL;          /* EOnCE EDCD Control Register */
    VUByte   avucReservedE12[2];
    VUWord32 vuliEDCD_REF;           /* EOnCE EDCD Reference Value */
    VUWord32 vuliEDCD_MASK;          /* EOnCE EDCD Mask Register */
    VUByte   avucReservedE13[20];
    VUWord16 vusiECNT_CTRL;          /* EOnCE Counter Control Register */
    VUByte   avucReservedE14[2];
    VUWord32 vuliECNT_VAL;           /* EOnCE Counter Value Register */
    VUWord32 vuliECNT_EXT;           /* EOnCE Extension Counter Value */
    VUByte   avucReservedE15[20];
    VUByte   vucESEL_CTRL;           /* EOnCE Selector Control Register */
    VUByte   avucReservedE16[3];
    VUWord16 vusiESEL_DM;           /* EOnCE Selector DM Mask */
    VUByte   avucReservedE17[2];
    VUWord16 vusiESEL_DI;           /* EOnCE Selector DI Mask */
    VUByte   avucReservedE18[6];
    VUWord16 vusiESEL_ETB;          /* EOnCE Selector ETB Mask */
    VUByte   avucReservedE19[2];
    VUWord16 vusiESEL_DTB;          /* EOnCE Selector DTB Mask */
    VUByte   avucReservedE20[10];
    VUByte   vucTB_CTRL;            /* EOnCE Trace Buffer Control Register */
    VUByte   avucReservedE21[3];
    VUWord16 vusiTB_RD;             /* EOnCE Trace Buffer Read Pointer */
    VUByte   avucReservedE22[2];
    VUWord16 vusiTB_WR;             /* EOnCE Trace Buffer Write Pointer */
    VUByte   avucReservedE23[2];
    VUWord32 vuliTB_BUFF;           /* EOnCE Trace Buffer */
    VUByte   avucReservedE24[176];

} t_EOnCEMM;

#endif /* _EOnCE_H */
```

NOTES:

NOTES:

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2675
Rev. 1
11/2004