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.

\(^1\) 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 address 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

```c
#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 Address Event Detection Channel i Control Registers */
    EDCAi_CTRL[CBCS] = 1; /* Watch for accesses greater than */
    EDCAi_CTRL[CACS] = 1; /* Watch for accesses less than */
    EDCAi_REFA = 0x0; /* Watch for accesses greater than 0x0 */
    EDCAi_REFB = 0x321; /* Watch for accesses less than 0x320 */
    EDCAi_REFDM = 0x0; /* Watch for accesses equal to */

    /* Program EOnCE Event Selector Control Registers */
    ESEL_CTRL = 0x0000; /* All events are ORed */
    ESEL_DM = 0x0000; /* All event monitors are enabled */

    /* Program EOnCE Comparison Registers */
    EDCAi_CMPA = 0x0; /* Comparator A is not used */
    EDCAi_CMPB = 0x320; /* Comparator B is not used */
}
```

Integrated EOnCE Examples, Rev. 1
Monitoring Addresses to Enter Debug

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 = 0x000000321;
EOnCE->vuliEDCA2_MASK = 0xffffffff;

EDCA3 checks XB bus for addresses between 0x0 to 0x320 */
EOnCE->vuliEDCA3_REFA = 0x00000000;
EOnCE->vuliEDCA3_REFB = 0x000000321;
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.
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.

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*. 

---

**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): 0x001
- =
- ≠
- >
- <

#### 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
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 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

```c
#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(" hmc1r #$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 */
/**************************************************************************/
EOnCE->vusiEDCD_CTRL = 0x00000; /* ECD signals not in use in this example */

/**************************************************************************/
/* Program EOnCE Event Detection Control Registers */
/**************************************************************************/
EOnCE->vusiEDCA0_CTRL = 0x00000; /* 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 */
```
Monitoring an Address and Data Combination to Cause a Debug Exception

```c
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->vuciESEL_CTRL = 0x02;
EOnCE->vuciESEL_DI = 0x0101; /* EDCA0, EDCD trigger sources for exception */
}
```

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.

---

**Figure 5. Configure EDCA0**

### 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
- **Enable after Event On:**
  - Disabled, EDCA0, EDCA1, EDCA2, EDCA3, EDCD
  - EDCA5, EDCD, Counter, EE pins, Enabled
- **Mask (Hex 32 bits):** 0xffffffff

---

Integrated EOnCE Examples, Rev. 1

8 Freescale Semiconductor
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...
Chaining Events and Using the Event Counter

Integrated EOnCE Examples, Rev. 1

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

```c
#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.

![Configuration for EDCA0](image1)

**Address Event Detection Channel 1**

- **Bus Selection:**
  - AXAB
  - XABBB
  - AXAB and XABBB
  - 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

![Configuration for EDCA1](image2)

**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):** 0xffffffff
- **Extension Counter Value (Hex 32 bits):** 0x0

**Figure 9.** Configure EOnCE Counter

Integrated EOnCE Examples, Rev. 1
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**

```c
#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 */
    /*******************************************************************************/
    /* EE signal 2 is input */
    EOnCE->vusiEE_CTRL = 0x0030;

    /*******************************************************************************/
    /* 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;
```
Using the Trace Buffer and External EE Signals

```c
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.

<table>
<thead>
<tr>
<th>Control</th>
<th>EDCA0</th>
<th>EDCA1</th>
<th>EDCA2</th>
<th>EDCA3</th>
<th>EDCA4</th>
<th>EDCA5</th>
<th>EDCD</th>
<th>Counter</th>
<th>Selector</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE Ptn 0</td>
<td>output: detection by EDCA0</td>
<td>input: enable EDCA0 event</td>
<td>input: Debug Request</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE Ptn 1</td>
<td>output: detection by EDCA1</td>
<td>input: enable EDCA1 event</td>
<td>output: DebugAck.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE Ptn 2</td>
<td>output: detection by EDCA2</td>
<td>input: enable EDCA2 event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE Ptn 3</td>
<td>output: detection by EDCA3</td>
<td>output: ERTC Receive register is FULL</td>
<td>input: enable EDCA3 event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE Ptn 4</td>
<td>output: detection by EDCA4</td>
<td>output: data in ETRSM Transmit register is READY</td>
<td>input: enable EDCA4 event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE Ptn 5</td>
<td>output: detection by EDCA5</td>
<td>input: enable EDCA5 event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EED Ptn</td>
<td>output: detection by EDCD</td>
<td>input: enable EDCD event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. Configure EOnCE Signal Control
Using the Trace Buffer and External EE Signals

**Figure 11.** Configure Event Selector for Example 4

<table>
<thead>
<tr>
<th>Event(s) to Enter DEBUG Mode:</th>
<th>OR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG Mode Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDCA5</td>
<td>EDCA4</td>
<td>EDCA3</td>
</tr>
<tr>
<td>EDCA1</td>
<td>EDCA0</td>
<td>DEBUGEV</td>
</tr>
<tr>
<td>EE4</td>
<td>EE3</td>
<td>EE2</td>
</tr>
<tr>
<td>EE0</td>
<td>COUNT</td>
<td>EDCD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event(s) to Enter DEBUG Exception:</th>
<th>OR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG Exception Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDCA5</td>
<td>EDCA4</td>
<td>EDCA3</td>
</tr>
<tr>
<td>EDCA1</td>
<td>EDCA0</td>
<td>DEBUGEV</td>
</tr>
<tr>
<td>EE4</td>
<td>EE3</td>
<td>EE2</td>
</tr>
<tr>
<td>EE0</td>
<td>COUNT</td>
<td>EDCD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event(s) to Enable Trace:</th>
<th>OR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG Enable Trace Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDCA5</td>
<td>EDCA4</td>
<td>EDCA3</td>
</tr>
<tr>
<td>EDCA1</td>
<td>EDCA0</td>
<td>DEBUGEV</td>
</tr>
<tr>
<td>EE4</td>
<td>EE3</td>
<td>EE2</td>
</tr>
<tr>
<td>EE0</td>
<td>COUNT</td>
<td>EDCD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events to Disable Trace:</th>
<th>OR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG Disable Trace Mask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDCA5</td>
<td>EDCA4</td>
<td>EDCA3</td>
</tr>
<tr>
<td>EDCA1</td>
<td>EDCA0</td>
<td>DEBUGEV</td>
</tr>
<tr>
<td>EE4</td>
<td>EE3</td>
<td>EE2</td>
</tr>
<tr>
<td>EE0</td>
<td>COUNT</td>
<td>EDCD</td>
</tr>
</tbody>
</table>

**Figure 12.** Configure Trace Buffer

<table>
<thead>
<tr>
<th>Control</th>
<th>EDCA0</th>
<th>EDCA1</th>
<th>EDCA2</th>
<th>EDCA3</th>
<th>EDCA4</th>
<th>EDCA5</th>
<th>EDCD</th>
<th>Counter</th>
<th>Selector</th>
<th>Trace</th>
</tr>
</thead>
</table>

**Integrated EOnCE Examples, Rev. 1**
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.

```c
#include "typedefs.h"
```
typedef struct EOnCEMM
{
    /* EOnCE Registers */
    VUWord32 vuliESR;               /* EOnCE status register */
    VUWord32 vuliEMCR;              /* EOnCE Monitor and Control Register */
    VUWord32 vuliERCVO;             /* EOnCE Receive Register LSB */
    VUWord32 vuliERCV1;             /* EOnCE Receive Register MSB */
    VUWord32 vuliERSMTO;            /* EOnCE Receive Register LSB */
    VUWord32 vuliERSTM1;            /* 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;

#include <eonce.h>
NOTES:
Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. All other product or service names are the property of their respective owners.