# Using the SC140 Enhanced OnCE™ Stopwatch Timer

Application Note

*by*
*Kim-Chyan Gan*
*and Yuval Ronen*

AN2090/D
Rev. 0, 11/2000

MOTOROLA

**How to reach us:**

**USA/EUROPE/Locations Not Listed**: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado, 80217
1-303-675-2140 or 1-800-441-2447

**JAPAN**: Motorola Japan, Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu, Minato-ku,
Tokyo 106-8573 Japan. 81-3-3440-3569

**ASIA/PACIFIC**: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852-26668334

**Customer Focus Center: 1-800-521-6274**

HOME PAGE: http://motorola.com/semiconductors

# Abstract and Contents

In software application development on embedded devices, optimization is critical. Optimized code provides not only faster processing speed but also lower power consumption and longer battery life. Optimized code also minimizes CPU load, enabling more applications to fit into a single chip. In selecting a processor for a system, it is important to understand and compare the speed of execution of different processors. All of these applications may require a stopwatch as a criterion for measurement. This application note presents the stopwatch timer application developed on the Software Development Platform (SDP). The stopwatch timer is implemented using the Enhanced OnCE™ resource, which is non-intrusive to SC140 operation, therefore, timing can be correctly measured. Two stopwatch techniques, with and without source code modification, are presented. This application note also explains how to port stopwatch applications to other SC140 based devices.

## 1 Introduction

A stopwatch timer is an apparatus for measuring the exact duration of an event. Measuring time during execution of code on a Digital Signal Processor (DSP) is useful to identify opportunities for code optimization, to understand system loading, and to compare execution speeds of different processors.

This application note presents techniques to implement a stopwatch timer on the StarCore, SC140, using the built-in features of the DSP's Enhanced On-Chip Emulation (OnCE) module.

Although many devices with embedded DSPs provide application-specific timer blocks, these timers cannot be used for debugging purposes without interference from the application itself. The ability to use the Enhanced OnCE as a stopwatch timer allows for non-intrusive timing capabilities and is available on any system that integrates the SC140 core.

Two techniques to set up the stopwatch timer are described in this application note:

1. Setting up the stopwatch timer using the SC140 code within an application.
2. Setting up the stopwatch timer using the Metrowerks Code Warrior debugger[1].

The examples given in this application note describe the use of the stopwatch timer in the StarCore, SC140's Software Development Platform (SDP). Minor configuration changes are needed to apply the techniques to other SC140-based devices. The necessary modifications are also described in this application note.

This document is organized to present these techniques as follows: Section 2, "SC140 Enhanced OnCE Stopwatch Timer Capabilities," describes the specific capabilities of the stopwatch timer for the SC140 Enhanced OnCE. Section 3, "Setting Up the Stopwatch Timer Within an Application," describes instrumenting the SC140 application code to use the stopwatch timer. Section 4, "Setting Up the Stopwatch Timer Within the Debugger," presents the set up of the stopwatch timer within the Metrowerks Code Warrior debugger.

Supporting information is included in the remainder of the document. Section 5, "Setting Up the System Clock Speed," describes how to set the clock rate of the SC140 in the SDP by configuring the PLL in hardware and software applications. Section 6, "Verifying Correct Setup," describes techniques to verify that the system is set up correctly and that the measurements are reasonable. The Section 7, "Conclusion," provides a summary of this document. Section 8, "References," lists additional documentation to help the reader and Appendix A, "Complete Example of Profiling," provides a complete profile using the Enhanced OnCE stopwatch timer in the SC140.

1. At the time this application note was written, the release of Metrowerks Code Warrior is Beta v.1.0. All the figures related to screen capture of Code Warrior tools may vary in future versions.

# 2 SC140 Enhanced OnCE Stopwatch Timer Capabilities

This section presents the features of the Enhanced OnCE stopwatch timer and the resources required for implementation. The capabilities of the implementation of the stopwatch timer are also explained.

## 2.1 Features

The Enhanced OnCE stopwatch timer provides the following features:

- A 64-bit counter, incrementing on each DSP clock cycle. The counter is less susceptible to overflow with 64-bit precision.
- The stopwatch timer can be used repeatedly during the execution of an application.

Conversion between clock cycles and absolute time, based on the operating clock frequency of the DSP, is described in Section 3.4, “Converting Cycles to Actual Time,” on page -6.

## 2.2 Resources

The Enhanced OnCE stopwatch timer requires use of these resources:

- One Enhanced OnCE event detector (of the six available on each DSP)
- The Enhanced OnCE event counter
- Program memory of 724 bytes

Because the Enhanced OnCE supports only one event counter, the stopwatch timer cannot be used if the event counter is required for other debugging purposes (such as; the set up of a debugger breakpoint that requires counting occurrences of events).

## 2.3 Implementation

The stopwatch timer implementation allocates a variable in memory which serves as the target for memory write operations. The Enhanced OnCE event detector is set up to detect writes to this flag variable. When setting up of an Enhanced OnCE event detector to detect memory access operations, it is necessary to specify which of the two data memory buses should be “snooped”. Because the selection of the bus to be used is performed dynamically, the event detector is set up to snoop both buses (XABA or XABB). Upon detecting the write to the flag variable, the event detector enables the Enhanced OnCE event counter, which starts counting down.

The Enhanced OnCE event counter can be configured as either a 64-bit counter or a 32-bit counter. Configuring the counter to use 64-bits eliminates the danger of counter overflow, at a negligible extra cost.

To stop the stopwatch timer, an appropriate value is written into the Enhanced OnCE memory-mapped event counter control register. When this operation is completed, the cycle countdown halts. At this point it is possible to read out the values of the Enhanced OnCE event counter registers, and translate them into elapsed number of cycles, or elapsed absolute time.

## 3 Setting Up the Stopwatch Timer Within an Application

This section describes the operations necessary to initialize, start, and stop the stopwatch timer within an application. The sequence of operations is shown in Figure 1. Additionally, this section presents the conversion of cycles to actual time and puts all the application code together. Finally, this section explains how to adapt the stopwatch timer code to other SC140-based devices.

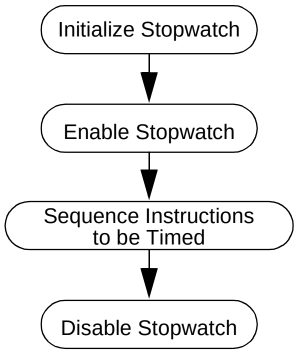


**Figure 1. Sequence of Operations**

### 3.1 Initializing the Stopwatch Timer

The C code to set up the stopwatch timer is shown in Code 1.

**Code 1. Event Detector Setup Code**

```
/*
* Header file contains definitions of EOnCE memory-mapped register addresses,
* and definition of the WRITE_IOREG() macro.
*/
#include "EOnCE_registers.h"

static volatile long EOnCE_stopwatch_timer_flag;           /*Global dummby variable*/

void EOnCE_stopwatch_timer_init()
{
             WRITE_IOREG(EDCA1_REFA,(long)&EOnCE_stopwatch_timer_flag);
                                 /* Address to snoop for on XABA */
             WRITE_IOREG(EDCA1_REFB,(long)&EOnCE_stopwatch_timer_flag);
                                 /* Address to snoop for on XABB */
             WRITE_IOREG(EDCA1_MASK,MAX_32_BIT);
                                 /* No masking is performed in address comparison */
             WRITE_IOREG(EDCA1_CTRL,0x3f06);
                                 /* Detect writes on both XABA and XABB */
}
```

The header file "EOnCE_registers.h" contains the macro definitions, such as EDCA1_MASK, which provides each of the Enhanced OnCE memory-mapped register's corresponding memory address.

This header file also defines the C macros: READ_IOREG() and WRITE_IOREG(). These macros simplify the read and write operations on memory-mapped registers.

Initializing the stopwatch timer consists of setting up the Address Event Detection Channel (EDCA). The role of EDCA in the stopwatch timer implementation is to trigger the commencement of the cycle countdown. The Enhanced OnCE supports six EDCAs. The implementation presented in this application note uses EDCA1, though this choice is arbitrary.

Set up of the EDCA requires initializing the following four registers:

- The 32-bit EDCA reference value register A (EDCA1_REFA).
- The 32-bit EDCA reference value register B (EDCA1_REFB).
- The 32-bit EDCA mask register (EDCA1_MASK).
- The 16-bit EDCA control register (EDCA1_CTRL).

## 3.1.1 Event Detector Control

The register, EDCA1_CTRL, controls the behavior of the EDCA. The fields of the EDCA1_CTRL register are shown in Figure 2.

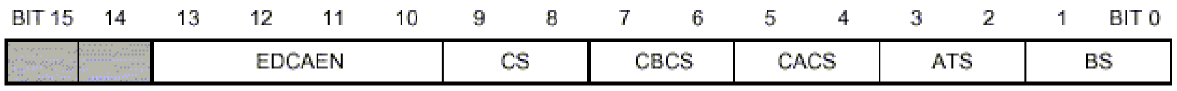


**Figure 2. EDCA Control Register (EDCA1_CTRL)**

Table 1 describes the settings of these fields in the stopwatch timer implementation.

**Table 1. EDCA_CTRL Settings**

| Field | Setting (binary value) | Description |
|---|---|---|
| EDCAEN | 1111 | This channel is enabled |
| CS | 11 | Trigger event if address matches either comparator A or comparator B |
| CBCS | 00 | An “address match” is detected when sampled bus value **equals** value in EDCA_REFB |
| CACS | 00 | An “address match” is detected when sampled bus value **equals** value in EDCA_REFA |
| ATS | 01 | Detect write access only |
| BS | 10 | The sampled buses are XABA and XABB |

The EDCA becomes enabled as soon as these values are written into the control register. The EDCA stays enabled for the duration of the program execution to enable repeated use of the stopwatch timer.

## 3.1.2 Address Comparison Setup

The purpose of the address comparison in the EDCA is to detect writes to the stopwatch timer flag variable. Because writes may take place on either of the two data memory buses, both registers, EDCA1_REFA and EDCA1_REFB, are set up to contain the address of the stopwatch timer flag variable.

The register EDCA1_MASK allows masking of address bits when comparing the sampled address with those in the EDCA1_REFA and EDCA1_REFB registers. In the implementation of the stopwatch timer all

32-bits of the flag variable’s address are used. Thus, EDCA1_MASK is set to 0xffffffff, meaning all address bits will be compared.

## 3.2 Starting the Stopwatch Timer

The C code to start the stopwatch timer is shown in Code 2.

**Code 2. C Code to Start the Stopwatch Timer**

```
#include “EOnCE_registers.h”

void EOnCE_stopwatch_timer_start()
{
    WRITE_IOREG(ECNT_VAL,MAX_32_BIT);  /* Countdown will start at (2**32)-1 */
    WRITE_IOREG(ECNT_EXT,0);           /* Extension will count up from zero */
    WRITE_IOREG(ECNT_CTRL,0x12c);
                          /* Counting will be triggered by detection on EDCA1 */
    EOnCE_stopwatch_timer_flag = 0;
                          /* This write to the flag triggers the counter */
}
```

Before triggering the stopwatch timer, it is necessary to initialize the counter registers. Initializing the event counter requires set up of the following three 32-bit registers:

- Event counter value register (ECNT_VAL)
- Extension counter value register (ECNT_EXT)
- Event counter control register (ECNT_CTRL)

Once these initializations are complete, the C code triggers the stopwatch timer and cycle counting commences.

### 3.2.1 Event Counter Control

This section describes the initialization of the event counter registers. The register, ECNT_CTRL, controls the behavior of the event counter. The fields of the ECNT_CTRL register are shown in Figure 3.

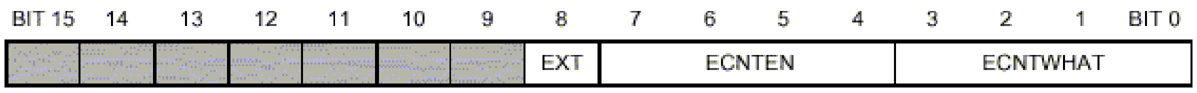


**Figure 3. Event Counter Control Register (ECNT_CTRL)**

Table 2 describes the settings of these fields in the stopwatch timer implementation.

**Table 2. ECNT_CTRL Settings**

| Field | Setting (binary value) | Description |
|---|---|---|
| EXT | 1 | Event counter operates as a 64-bit counter |
| ECNTEN | 0010 | The event counter is disabled, and will be enabled when an event is detected by EDCA1 |
| ECNTWHAT | 1100 | The counter will advance on each core clock cycle |

### 3.2.2 Counter Registers

ECNT_VAL is a countdown counter, and ECNT_EXT is a countup counter. ECNT_VAL is decremented on each occurrence of an event, as specified in the control register. ECNT_EXT is incremented each time there is an underflow in ECNT_VAL.

For maximum cycle counting capacity, the stopwatch timer implementation initializes ECNT_VAL to the largest possible value which is 4294967295, or 0xffffffff.

ECNT_EXT is initialized to zero.

## 3.3 Stopping the Stopwatch Timer

The C code to stop the stopwatch timer is shown in Code 3.

**Code 3. C Code to Stop the Stopwatch Timer**

```
#include “EOnCE_registers.h”

void EOnCE_stopwatch_timer_stop(unsigned long *clock_ext, unsigned long *clock_val)
{
    WRITE_IOREG(ECNT_CTRL,0);                /* Disable event counter */
    READ_IOREG(ECNT_VAL,*clock_val); /* Save ECNT_VAL in program variable */
    READ_IOREG(ECNT_EXT,*clock_ext); /* Save ECNT_EXT in program variable */
    *clock_val = (MAX_32_BIT-*clock_val); /* Adjust for countdown */
}
```

To stop the stopwatch timer, the ECNT_CTRL register is set to zero. After stopping the stopwatch timer, the routine copies the values of ECNT_VAL and ECNT_EXT registers into program variables. This allows putting the stopwatch timer into use again without losing the result of the previous measurement.

Because ECNT_VAL contains the result of a countdown process, the routine converts that result into the actual number of cycles elapsed by subtracting the ECNT_VAL from the value to which it was initialized, specifically, 4294967295.

## 3.4 Converting Cycles to Actual Time

The stopwatch timer measures durations in units of core clock cycles. Most often the units of interest are units of absolute time, such as milliseconds or microseconds.

Conversion from core clock cycles, as measured by the event counter registers, to milliseconds is computed in Equation 1.

$$\mathrm{Time(ms)} = (\mathrm{EXT} \times \mathrm{0xffffffff} + \mathrm{VAL}) \times \frac{1000}{\mathrm{ClockSpeed}} \qquad \textbf{\textit{Eqn. 1}}$$

In this equation, EXT is the value in ECNT_EXT, VAL is the value in ECNT_VAL, and Clock Speed is measured in hertz.

Code 4 on page -7 shows the C code for clock-cycle-to-time conversion.

The C code depends on setting the value of the constant CLOCK_SPEED in EOnCE_stopwatch.c to match the clock speed as set in the PLL.

The conversion routine distinguishes between three different output units: seconds, milliseconds, and microseconds. Handling each unit separately allows the computations to be performed using integer arithmetic without loss of accuracy.

**Code 4. C Code for Clock-Cycle-to-Time Conversion**

```
typedef enum { EONCE_SECOND, EONCE_MILLISECOND, EONCE_MICROSECOND } tunit;

unsigned long Convert_clock2time(unsigned long clock_ext, unsigned long clock_val,
short option)
{
    unsigned long result;
    switch(option)
    {
        case EONCE_SECOND:
            result= clock_ext*MAX_32_BIT/CLOCK_SPEED + clock_val/CLOCK_SPEED;
            break;
        case EONCE_MILLISECOND:
            result= clock_ext*MAX_32_BIT/(CLOCK_SPEED/1000)
                    + clock_val/(CLOCK_SPEED/1000);
            break;
        case EONCE_MICROSECOND:
            result= clock_ext*MAX_32_BIT/(CLOCK_SPEED/1000000)
                    + clock_val/(CLOCK_SPEED/1000000);
            break;
        default: result=0;                          /* error condition */
            break;
    }
    return result;
}
```

## 3.5 Putting it All Together

Code 5 depicts the use of the stopwatch timer using the routines that have been described so far.

**Code 5. Use of Stopwatch Timer Functions**

```
long clock_ext,clock_val,clock_cycle,time_sec;
             ...

EOnCE_stopwatch_timer_init(); /* Execute once per program execution */
      ...
EOnCE_stopwatch_timer_start();
/*
 * Code whose execution time is measured goes here
 */
EOnCE_stopwatch_timer_stop(&clock_ext, &clock_val);
      ...
time_sec = Convert_clock2time(clock_ext, clock_val, EONCE_SECOND);
      ...
      ...
EOnCE_stopwatch_timer_start();
/*
 * Code whose execution time is measured goes here
 */
EOnCE_stopwatch_timer_stop(&clock_ext, &clock_val);
      ...
time_sec = Convert_clock2time(clock_ext, clock_val, EONCE_SECOND);
```

The calls to the stopwatch timer functions can be made from different locations in the application code, not necessarily from within one subroutine. The `EOnCE_stopwatch_timer_start`() and `EOnCE_stopwatch_timer_stop`() can be called more than once to measure the time consumed in different modules as desired.

## 3.6 Adapting Stopwatch Timer Code to Other SC140 Devices

The stopwatch timer implementation controls the Enhanced OnCE by writing to its memory-mapped registers. The addresses of these registers are determined in the memory map of the device in which the SC140 core is embedded. The offset between the base address of the memory-mapped peripherals and the addresses of the Enhanced OnCE registers is the same across SC140-based devices.

Therefore, when adapting the stopwatch timer code for a specific device it suffices to set the value of REG_BASE_ADDRESS in the header file “EOnCE_registers.h.” In the Software Development Platform (SDP), the base register of Enhanced OnCE is 0x00effe00. Consult the user manual of the specific device for the value to set this C macro.

# 4 Setting Up the Stopwatch Timer Within the Debugger

The preceeding section described a technique that requires instrumenting, or modifying, the source code of the application being measured. Occasionally developers are faced with a situation where instrumentation of the code is not possible. For example, the code might be available in object form only.

In such cases it is possible to measure execution times by setting up the stopwatch timer within the Metrowerks Code Warrior SC140 debugger. This section explains this set up.

## 4.1 Initializing the Stopwatch Timer

The technique described in Section 3 triggered the stopwatch timer by detecting a write to the stopwatch timer flag variable. When the stopwatch timer is set up within the debugger, the triggering event is chosen to be the execution of the first instruction in the measured code. Program code disassembly is used to obtain the address of this first instruction.

**NOTE:**

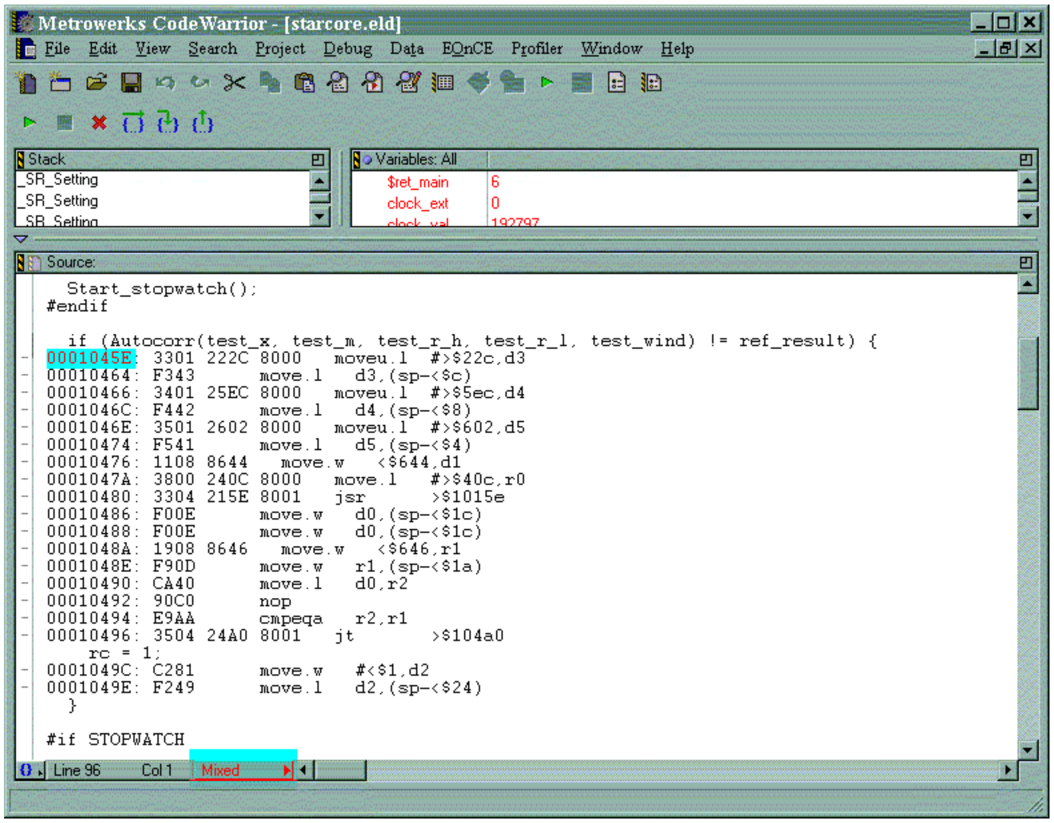
In the following descriptions of selecting options in the debugger windows, the > character separates each command in the menu hierarchy. For example, **EOnCE > EOnCE Configurator > EDCA1** lists the hierarchy to be followed to select the EDCA1. In this example, one would highlight EOnCE in menu bar, then highlight EOnCE Configurator, and finally, choose the EDCA1 tab.

### 4.1.1 Setting Up the Event Detector

The procedure to set up the stopwatch timer is described in this section. To set up the event detector, find the starting address of the measured function or sequence of instructions, as follows:

1. Choose **Project > Debug**.
2. In the debugger window, choose “mixed”.

The starting address can be found in mixed mode, as shown in Figure 4 on page -9.


Metrowerks CodeWarrior - [starcore.eld]
File Edit View Search Project Debug Data EOnCE Profiler Window Help
Stack
Variables: All
Source:
Line 96
Col 1
Mixed

```
_SR_Setting
_SR_Setting
_SR_Setting
```

```
$ret_main    6
clock_ext    0
clock_val    192797
```

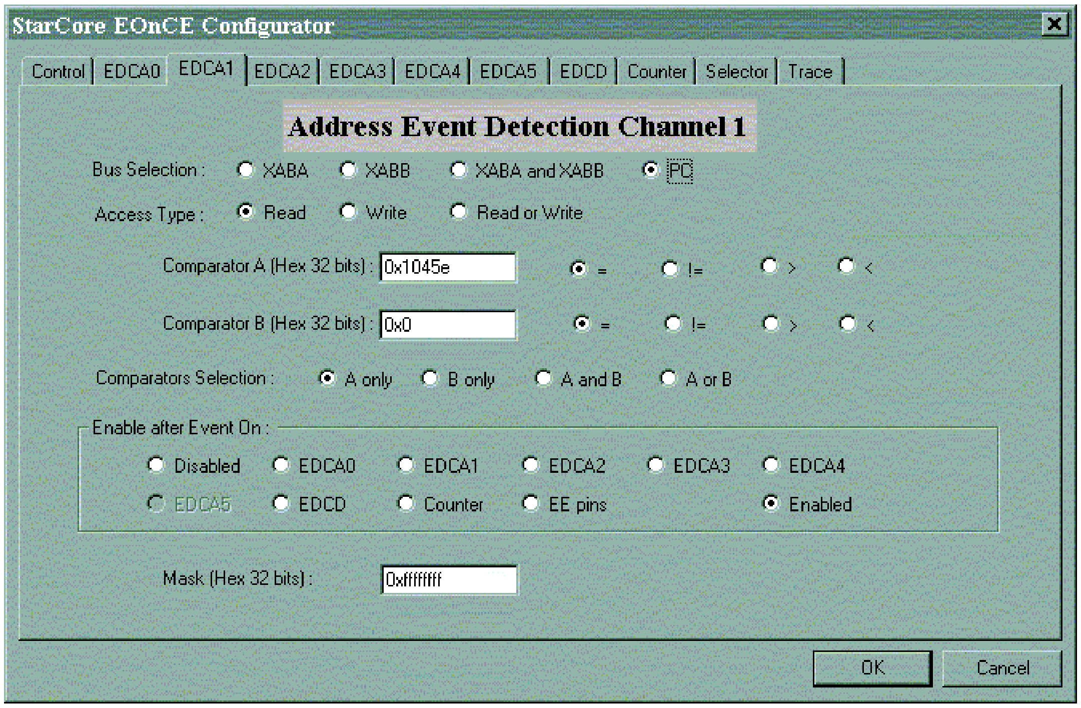
```
  Start_stopwatch();
#endif

  if (Autocorr(test_x, test_m, test_r_h, test_r_l, test_wind) != ref_result) {
0001045E: 3301 222C 8000   moveu.l   #>$22c,d3
00010464: F343             move.l    d3,(sp-<$c)
00010466: 3401 25EC 8000   moveu.l   #>$5ec,d4
0001046C: F442             move.l    d4,(sp-<$8)
0001046E: 3501 2602 8000   moveu.l   #>$602,d5
00010474: F541             move.l    d5,(sp-<$4)
00010476: 1108 8644         move.w    <$644,d1
0001047A: 3800 240C 8000   move.l    #>$40c,r0
00010480: 3304 215E 8001   jsr       >$1015e
00010486: F00E             move.w    d0,(sp-<$1c)
00010488: F00E             move.w    d0,(sp-<$1c)
0001048A: 1908 8646         move.w    <$646,r1
0001048E: F90D             move.w    r1,(sp-<$1a)
00010490: CA40             move.l    d0,r2
00010492: 90C0             nop
00010494: E9AA             cmpeqa    r2,r1
00010496: 3504 24A0 8001   jt        >$104a0
    rc = 1;
0001049C: C281             move.w    #<$1,d2
0001049E: F249             move.l    d2,(sp-<$24)
  }

#if STOPWATCH
```

**Figure 4. Finding the Starting Address in the Debugger**

After finding the starting address, the event detector can be setup. The procedures are outlined in these steps:

1. Choose **EOnCE > EOnCE Configurator > EDCA1.**
2. Click PC in the "Bus Selection" box.
3. Enter the address of the first instruction in the measured code into the "Comparator A Hex 32-bits" box.
4. Click Enable in the "Enabled after Event On" box.

Figure 5 on page -10 shows the event detection settings after performing these steps. For more detailed information in Enhanced OnCE configuration in Code Warrior tools, refer to [3].


StarCore EOnCE Configurator
Control | EDCA0 | EDCA1 | EDCA2 | EDCA3 | EDCA4 | EDCA5 | EDCD | Counter | Selector | Trace
OK
Cancel

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) : 0x1045e   [✓ =]   !=   >   <

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

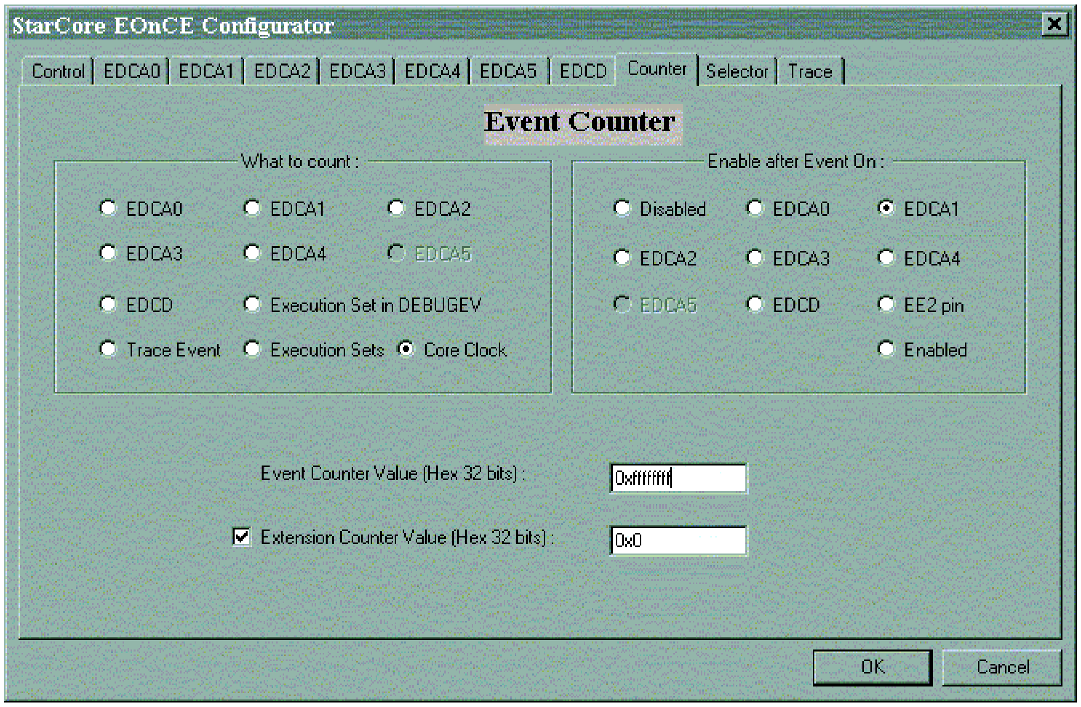
**Figure 5. Event Detection Settings**

### 4.1.2 Setting Up the Event Counter

The event counter is configured to the same mode as described in Section 3.2.1, “Event Counter Control,” on page -5, except this time the configuration is made using the debugger windows:

1. Choose **EOnCE > EOnCE Configurator > Counter**.
2. Click Core Clock in the “What to count” box.
3. Click EDCA1 in the “Enable after Even On” box.
4. Type “0xffffffff” in the “Event Counter Value (Hex 32 bits)” box.
5. Check the box in the left side of “Extension Counter Value (Hex 32 bits)”.

The result of this procedure is shown in Figure 6 on page -11.


StarCore EOnCE Configurator

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

**Event Counter**

What to count :

EDCA0 EDCA1 EDCA2
EDCA3 EDCA4 EDCA5
EDCD Execution Set in DEBUGEV
Trace Event Execution Sets [✓ Core Clock]

Enable after Event On :

Disabled EDCA0 [✓ EDCA1]
EDCA2 EDCA3 EDCA4
EDCA5 EDCD EE2 pin
Enabled

Event Counter Value (Hex 32 bits) : 0xffffffff

[✓] Extension Counter Value (Hex 32 bits) : 0x0

OK Cancel

Figure 6. Event Counter Settings

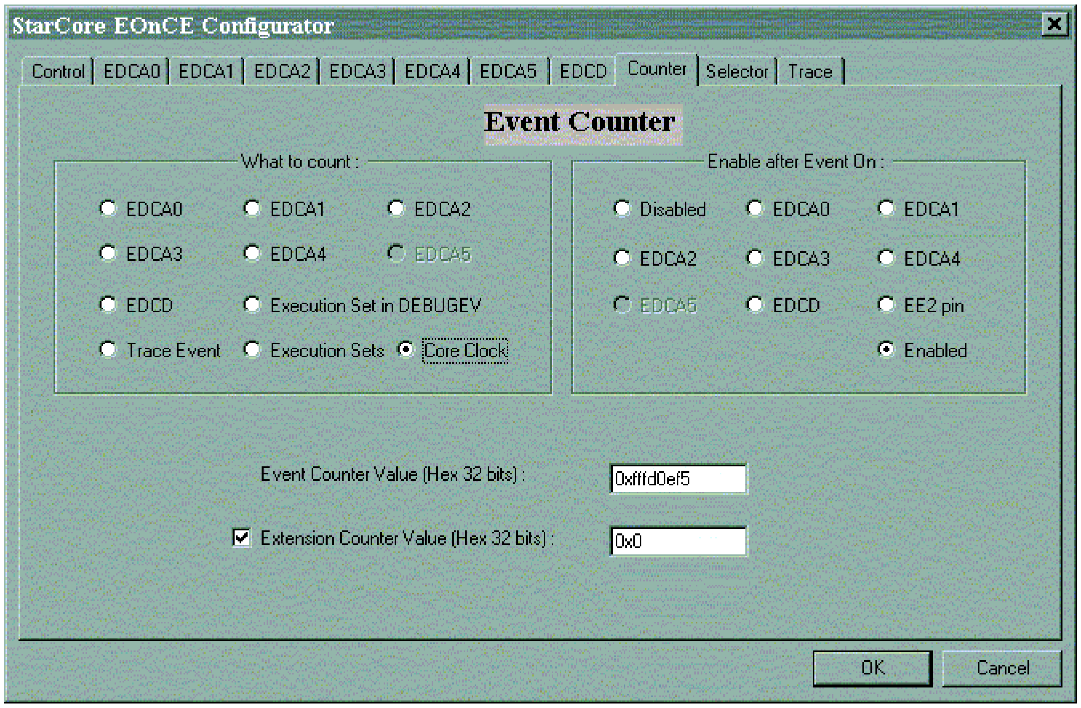
## 4.2 Stopping the Stopwatch Timer

Stopping the stopwatch timer is achieved by halting execution of the application at a breakpoint. Set up the breakpoint at the point in the application at which timing should stop to achieve the desired affect.

After the breakpoint is in place, the debugger can be instructed to run the application.

When execution of the application reaches the breakpoint, the value of the stopwatch timer counters can be retrieved by opening the **EOnCE > EOnCE Configurator > Counter** dialog box. Figure 7 on page -12 shows an example of an event counter dialog box after the debugger halt at the breakpoint. Because the countdown counter is initialized to maximum value (0xffffffff), the difference between maximum value and the value in the “Event Counter Value” column yields the real SC140 clock counts. To convert the values of the real SC140 clock counts to absolute time use the computation described in Section 3.4, “Converting Cycles to Actual Time,” on page -6.

After the debugger stops at the breakpoint, the counter, which is setup in sleep mode and is enabled by the events at the first instruction of the measured code, is now in enabled mode and continues to count (see “Enabled after Event on” column in Figure 7 on page -12). However, continued counting is irrelevant because the SC140 clock count has been achieved.


StarCore EOnCE Configurator

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

**Event Counter**

What to count :

| | | |
|---|---|---|
| EDCA0 | EDCA1 | EDCA2 |
| EDCA3 | EDCA4 | EDCA5 |
| EDCD | Execution Set in DEBUGEV | |
| Trace Event | Execution Sets | [✓ Core Clock] |

Enable after Event On :

| | | |
|---|---|---|
| Disabled | EDCA0 | EDCA1 |
| EDCA2 | EDCA3 | EDCA4 |
| EDCA5 | EDCD | EE2 pin |
| | | [✓ Enabled] |

Event Counter Value (Hex 32 bits) : 0xfffd0ef5

[✓] Extension Counter Value (Hex 32 bits) : 0x0

OK Cancel

Figure 7. Event Counter Dialog Box When Debugger Halts at Breakpoint

## 5 Setting Up the System Clock Speed

Every SC140-based device contains a Phase Lock Loop (PLL) block, which controls the operating frequency of the device. The frequency of the device is governed by the frequency control bits in the PLL control register, as defined in Equation 2.

$$\text{Fdevice} = \frac{\text{Fext} \times \left(\text{MFI} + \frac{\text{MFN}}{\text{MFD}}\right)}{\text{PODF} \times \text{PDF}} \qquad \textit{Eqn. 2}$$

In this equation:

- MFI (multiplication factor integer), MFN (multiplication factor numerator), MFD (multiplication factor denominator), and PODF (post division factor) are defined in the PCTL1 register.
- PDF (pre-division factor) is defined in the PCTL0 register.
- Fext is external input frequency to the chip at the EXTAL pin.
- Fdevice is the operating frequency of the device.

The range of values of these terms are describe in [1].

Figure 8 and Figure 9 on page -13 illustrate the PLL control registers; PCTL0 and PCTL1, respectively.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MFI1 | MFI0 | MFD9 | MFD8 | MFD7 | MFD6 | MFD5 | MFD4 | MFD3 | MFD2 | MFD1 | MFD0 | PD3 | PD2 | PD1 | PD0 |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pen | rcp | * | * | MFN9 | MFN8 | MFN7 | MFN6 | MFN5 | MFN4 | MFN3 | MFN2 | MFN1 | MFN0 | MFI3 | MFI2 |

**Figure 8. Programming Model of PCTL0**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | * | * | * | * | * | * | * | * | * | * | * | * | PODF2 | PODF1 | PODF0 |

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | COE |

**Figure 9. Programming Model of PCTL1**

## 5.1 Setting Up the PLL in Software

The clock frequency of the SC140 can be set up either in software or in hardware. This section describes how to set the SC140 in the Software Development Platform (SDP) to operate at 300 MHz using these two alternatives. The C code to set up the PLL to 300 MHz is shown in Code 6.

**Code 6. C Code to Set Up the PLL to 300 MHz**

```
#include "EOnCE_registers.h"
void PLL_setup_300MHz()
{
            asm("move.l #$80030003,PCTL0");
            asm("move.l #$00010000,PCTL1");
}
```

To set up the SC140 for operation at 300 MHz, the registers PCTL0 and PCTL1 should be set to the values 0x80030003 and 0x00010000, respectively. These settings are explained in Table 3.

**Table 3. Settings of PCTL0 and PCTL1**

| Field | **Setting** (binary value) | Description |
|---|---|---|
| PCTL0.PEN | 1 | PLL enabled. The internal clocks are derived from the PLL output |
| PCTL0.RCP | 0 | PLL locks with respect to the positive edge of the reference clock |
| PCTL0.MFN | 000000000 | MFN = 0 |
| PCTL0.MFI | 1100 | MFI = 24 |
| PCTL0.MFD | 000000000 | MDF = 1 |
| PCTL0.PD | 0011 | PD = 4 |
| PCTL1.COE | 1 | Clock out pin receives output |
| PCTL1.PODF | 0 | PODF = 1 |

With these configurations, the Fchip is calculated as expressed in Equation 3.

$$\mathrm{Fchip} = \frac{50\mathrm{MHz} \times \left(24 + \frac{0}{1}\right)}{4 \times 1} = 300\mathrm{Mhz} \qquad \textbf{\textit{Eqn. 3}}$$

The PLL should be configured so that the resulting PLL output frequency is in the range specified in the device’s technical data sheet.

## 5.2 Setting Up the PLL in Hardware

During the assertion of hardware reset, the value of all the PLL hardware configurations pins are sampled into the clock control registers (PCTL0 and PCTL1). Thus, it is possible to set up the core frequency at reset by configuring the jumpers on the SDP board.

To set up the PLL for operation at 300 MHz, the jumpers for PLLEN, PDF1, PDF0, MFI3 and MFI2 should be removed (thereby causing these bits to be asserted).

For more detail information jumper configuration of SDP, refers to [2].

# 6 Verifying Correct Setup

The previous sections described how to set up the Enhanced OnCE stopwatch timer and the DSP clock speed. This section describes techniques to verify that the system is set up correctly and that the Enhanced OnCE stopwatch timer measurements are reasonable as described in Section 3, “Setting Up the Stopwatch Timer Within an Application,” on page -3.

The verification process is based on measuring a specified time period, while also creating an external behavior (turning on and off an LED) that can be measured independently by a “wall clock” (that is; an independent stopwatch, such as; an oscilloscope).

## 6.1 Using the LED on the SDP

The implementation described in this section is based on the configuration of the Software Development Platform (SDP). In SDP, each of EE1 pins of the Enhanced OnCE is connected to an LED.

The following implementation is based on the ability to program the Enhanced OnCE to toggle the output value on its pins whenever an event is detected by one of the Enhanced OnCE event detection channels. The implementation below toggles the output value on the pin EE1 whenever the stopwatch timer starts or stops running. This capability requires just a small enhancement to the stopwatch timer software that is presented in Code 6 on page -13.

### 6.1.1 Setting Up EE1

The functionality of the Enhanced OnCE pins is controlled through the Enhanced OnCE pins control register (EE_CTRL). Figure 10 displays the structure of this register.

| BIT 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EEDDEF | ▨ | ▨ | ▨ | ▨ | EE5DEF | EE4DEF | | EE3DEF | | EE2DEF | | EE1DEF | | EE0DEF | |

**Figure 10. EE Pins Control Register (EE_CTRL)**

In EE_CTRL, the EE1DEF field is set to 00, which signifies output signal when detected by EDCA1. The remaining fields in EE_CTRL are irrelevant because they are not used.

Code 7 shows the setup code for EE control registers.

**Code 7. EOnCE_LED_init()**

```
void EOnCE_LED_init()
{
             *((long *)EE_CTRL) &= ~(3<<2); /* Toggle EE1 when event1 happens */
}
```

### 6.1.2 Toggling EE1

The initialization previously discussed the set up of EE1 to toggle each time an event is detected by EDCA1. The same channel is also used to trigger the stopwatch timer to count. Therefore, all that remains is to create an EDCA1 event when the stopwatch timer stops. This is achieved by writing to the Enhanced OnCE stopwatch timer flag variable.

Caution must be taken to perform this write only after execution of `EOnCE_stopwatch_timer_stop()`.

**Code 8. Turn LED Off**

```
void EOnCE_LED_off(){
    EOnCE_stopwatch_timer_flag = 0; /* Create an EDCA1 event */
}
```

## 6.2 Testing the Stopwatch Timer

The program depicted in Code 9 on page -16 sets up the stopwatch timer and then measures the time it takes to execute two loops whose duration is built into the program.

The measured code sequences are constructed to take 10 seconds and 7.5 seconds, respectively. These durations are constructed by having the code sample the Enhanced OnCE counter and loop until the expected number of clock cycles have passed.

It is recommended to try this code prior to measuring the target application, as a means of verifying correct setup of the system. If the times measured are not correct, it is recommended to check the PLL setup and the values of the clock speed and the memory-mapped register base (as set in the header files).

If the SC140 compiler beta 1.0 V is used, COMPILER_BETA_1_BUG must be defined to make the code run correctly. In the macro WRITE_IO(REG,VAL) or READ_IO(REG,VAL), REG is deemed as 16-bit even though it is 32-bit in macro definition. It is fixed by explicitly declaring a long variable if COMPILER_BETA_1_BUG is defined.

**Code 9.   Testing Code**

```
#include <stdio.h>
#include "EOnCE_stopwatch.h"

#ifdef COMPILER_BETA_1_BUG
extern long ECNT_VAL;
#else
#include "EOnCE_registers.h"
#endif

void PLL_setup_300MHz()
{
    asm("move.l #$80030003,PCTL0");
    asm("move.l #$00010000,PCTL1");
}



void main(){
    unsigned long clock_ext,clock_val,clock_cycle,cycle_req;
    unsigned long time_sec;
    extern unsigned long CLOCK_SPEED;

    PLL_setup_300MHz();
    EOnCE_stopwatch_timer_init();             /* Setup to event detector 1 for any write to
                                                 dummy variable. Setup EOnCE event counter
                                                 to count if event 1 happens */
    EOnCE_LED_init();                         /* Setup LED to toggle in detection of
                                                 event 1 */

    cycle_req = CLOCK_SPEED*10;               /* Calculate total clock cycles required
                                                 by 10 sec */
    EOnCE_stopwatch_timer_start();            /* Event 1 happens, counter & LED on */

    do {
        READ_IOREG(ECNT_VAL,clock_cycle);     /* Read bit 31-0 counter value */
        clock_cycle = MAX_32_BIT - clock_cycle;/* Minus max value due to count down */
    } while (clock_cycle <= cycle_req);

    EOnCE_stopwatch_timer_stop(&clock_ext, &clock_val);  /* Stop timer, return bit 63-0
                                                counter value */
    EOnCE_LED_off();                           /* LED off */
    time_sec = Convert_clock2time(clock_ext, clock_val, EONCE_SECOND);
    printf("duration = %u sec\n", time_sec);



    cycle_req = CLOCK_SPEED*7.5;             /* Calculate total clock cycles required
                                                 by 7.5 sec */
    EOnCE_stopwatch_timer_start();           /* Event 1 happens, counter & LED on */

    do {
        READ_IOREG(ECNT_VAL,clock_cycle);     /* Read bit 31-0 counter value */
        clock_cycle = MAX_32_BIT - clock_cycle;/* Minus max value due to count down */
    } while (clock_cycle <= cycle_req);

    EOnCE_stopwatch_timer_stop(&clock_ext, &clock_val);  /* Stop timer, return bit 63-0
                                                counter value */
    EOnCE_LED_off();                          /* LED off */
    time_sec = Convert_clock2time(clock_ext, clock_val, EONCE_MILLISECOND);
    printf("duration = %u ms\n", time_sec);

    return;
}
```

## 7 Conclusion

This application note presented two techniques to measure the speed of execution of software running on an SC140 device, using the SC140 Enhanced OnCE. The first technique requires instrumenting the application code with calls to stopwatch timer routines. The second technique controls the stopwatch timer within the Metrowerks Code Warrier debugger.

In addition, examples were provided in this application note to demonstrate setting up the SC140 Phase Lock Loop, and software control of the LED available on the Software Development Platform.

## 8 References

[1] *StarCore SC140 DSP Core Reference Manual* (order number MNSC140CORE/D), Motorola Inc., 2000.

[2] *StarCore 140 Software Development Platform*, Hardware Reference Manual (order number SC140SDPRM/D), Motorola Inc., 1999.

[3] *Code Warrior Targeting StarCore* (Beta 1.0 Release), Metrowerks Corp., 1993.

# Appendix A
## Complete Example of Profiling

A complete example of profiling using the Enhanced OnCE stopwatch timer in the SC140 platform is presented in this appendix.

**Code A-1. EOnCE_stopwatch.h**

```
typedef enum { EONCE_SECOND, EONCE_MILLISECOND, EONCE_MICROSECOND } tunit;

#define READ_IOREG(reg,val) val = *((volatile long *)reg)
#define WRITE_IOREG(reg,val) *((volatile long *)reg) = val
#define MAX_32_BIT 0xffffffff

void EOnCE_stopwatch_timer_init();
void EOnCE_stopwatch_timer_start();
void EOnCE_stopwatch_timer_stop(unsigned long *clock_ext, unsigned long *clock_val);
void EOnCE_LED_init();
void EOnCE_LED_off();
unsigned long Convert_clock2time(unsigned long clock_ext, unsigned long clock_val, short option);
```

**Code A-2.   EOnCE_registers.h**

```
/*   EOnCE registers */
#define EXCP_TABLE 0x7000
#define REG_BASE_ADDRESS 0x00effe00         /* EOnCE Status register */

#ifdef COMPILER_BETA_1_BUG
long  EE_CTRL       =         REG_BASE_ADDRESS+0x18; /* EOnCE EE pins Control register */
long  EDCA1_CTRL    =         REG_BASE_ADDRESS+0x44; /* EOnCE EDCA #1 Control register */
long  EDCA1_REFA    =         REG_BASE_ADDRESS+0x64; /* EOnCE EDCA #1 Reference Value A */
long  EDCA1_REFB    =         REG_BASE_ADDRESS+0x84; /* EOnCE EDCA #1 Reference Value B */
long  EDCA1_MASK    =         REG_BASE_ADDRESS+0xc4; /* EOnCE EDCA #1 Mask Register */
long  ECNT_CTRL     =         REG_BASE_ADDRESS+0x100; /* EOnCE Counter Control register */
long  ECNT_VAL      =         REG_BASE_ADDRESS+0x104; /* EOnCE Counter Value register */
long  ECNT_EXT      =         REG_BASE_ADDRESS+0x108; /* EOnCE Extension Counter Value */
#else
#define  EMCR                   REG_BASE_ADDRESS+0x4 /* EOnCE Monitor and Control register */
#define  ERCV                   REG_BASE_ADDRESS+0x8 /* EOnCE Receive register */
#define  ETRSMT0                REG_BASE_ADDRESS+0x10 /* EOnCE Receive register */
#define  ETRSMT1                REG_BASE_ADDRESS+0x14 /* EOnCE Receive register */
#define  EE_CTRL                REG_BASE_ADDRESS+0x18 /* EOnCE EE pins Control register */
#define  PC_EX                  REG_BASE_ADDRESS+0x1c /* EOnCE Exception PC register */
#define  EDCA0_CTRL             REG_BASE_ADDRESS+0x40 /* EOnCE EDCA #0 Control register */
#define  EDCA1_CTRL             REG_BASE_ADDRESS+0x44 /* EOnCE EDCA #1 Control register */
#define  EDCA2_CTRL             REG_BASE_ADDRESS+0x48 /* EOnCE EDCA #2 Control register */
#define  EDCA3_CTRL             REG_BASE_ADDRESS+0x4c /* EOnCE EDCA #3 Control register */
#define  EDCA4_CTRL             REG_BASE_ADDRESS+0x50 /* EOnCE EDCA #4 Control register */
#define  EDCA5_CTRL             REG_BASE_ADDRESS+0x54 /* EOnCE EDCA #5 Control register */
#define  EDCA0_REFA             REG_BASE_ADDRESS+0x60 /* EOnCE EDCA #0 Reference Value A */
#define  EDCA1_REFA             REG_BASE_ADDRESS+0x64 /* EOnCE EDCA #1 Reference Value A */
#define  EDCA2_REFA             REG_BASE_ADDRESS+0x68 /* EOnCE EDCA #2 Reference Value A */
#define  EDCA3_REFA             REG_BASE_ADDRESS+0x6c /* EOnCE EDCA #3 Reference Value A */
#define  EDCA4_REFA             REG_BASE_ADDRESS+0x70 /* EOnCE EDCA #4 Reference Value A */
#define  EDCA5_REFA             REG_BASE_ADDRESS+0x74 /* EOnCE EDCA #5 Reference Value A */
#define  EDCA0_REFB             REG_BASE_ADDRESS+0x80 /* EOnCE EDCA #0 Reference Value B */
#define  EDCA1_REFB             REG_BASE_ADDRESS+0x84 /* EOnCE EDCA #1 Reference Value B */
#define  EDCA2_REFB             REG_BASE_ADDRESS+0x88 /* EOnCE EDCA #2 Reference Value B */
#define  EDCA3_REFB             REG_BASE_ADDRESS+0x8c /* EOnCE EDCA #3 Reference Value B */
#define  EDCA4_REFB             REG_BASE_ADDRESS+0x90 /* EOnCE EDCA #4 Reference Value B */
#define  EDCA5_REFB             REG_BASE_ADDRESS+0x94  /* EOnCE EDCA #5 Reference Value B */
#define  EDCA0_MASK             REG_BASE_ADDRESS+0xc0  /* EOnCE EDCA #0 Mask Register */
#define  EDCA1_MASK             REG_BASE_ADDRESS+0xc4  /* EOnCE EDCA #1 Mask Register */
#define  EDCA2_MASK             REG_BASE_ADDRESS+0xc8  /* EOnCE EDCA #2 Mask Register */
#define  EDCA3_MASK             REG_BASE_ADDRESS+0xcc  /* EOnCE EDCA #3 Mask Register */
#define  EDCA4_MASK             REG_BASE_ADDRESS+0xd0  /* EOnCE EDCA #4 Mask Register */
#define  EDCA5_MASK             REG_BASE_ADDRESS+0xd4  /* EOnCE EDCA #5 Mask Register */
#define  EDCD_CTRL              REG_BASE_ADDRESS+0xe0  /* EOnCE EDCD Control register */
#define  EDCD_REF               REG_BASE_ADDRESS+0xe4  /* EOnCE EDCD Reference Value */
#define  EDCD_MASK              REG_BASE_ADDRESS+0xe8  /* EOnCE EDCD Mask register */
#define  ECNT_CTRL              REG_BASE_ADDRESS+0x100  /* EOnCE Counter Control register */
#define  ECNT_VAL               REG_BASE_ADDRESS+0x104  /* EOnCE Counter Value register */
#define  ECNT_EXT               REG_BASE_ADDRESS+0x108  /* EOnCE Extension Counter Value */
#define  ESEL_CTRL              REG_BASE_ADDRESS+0x120  /* EOnCE Selector Control register */
#define  ESEL_DM                REG_BASE_ADDRESS+0x124  /* EOnCE Selector DM Mask */
#define  ESEL_DI                REG_BASE_ADDRESS+0x128  /* EOnCE Selector DI Mask */
#define  ESEL_RST               REG_BASE_ADDRESS+0x12c  /* EOnCE Selector RST Mask */
#define  ESEL_ETB               REG_BASE_ADDRESS+0x130  /* EOnCE Selector ETB Mask */
#define  ESEL_DTB               REG_BASE_ADDRESS+0x134  /* EOnCE Selector DTB Mask */
#define  TB_CTRL                REG_BASE_ADDRESS+0x140  /* EOnCE Trace Buffer Control register */
#define  TB_RD                  REG_BASE_ADDRESS+0x144  /* EOnCE Trace Buffer Read Pointer */
#define  TB_WR                  REG_BASE_ADDRESS+0x148  /* EOnCE Trace Buffer Write Pointer */
#define  TB_BUFF                REG_BASE_ADDRESS+0x14c  /* EOnCE Trace Buffer */
#define  TRAP_EXCP              EXCP_TABLE  /* trap instruction exception */
#define  ILL_EXCP              EXCP_TABLE+0x80  /* illegal set or illegal instruction exception */
#define  DBG_EXCP               EXCP_TABLE+0xc0  /* debug exception (eonce) */
#define  OVFL_EXCP              EXCP_TABLE+0x100  /* overflow exception */
#define  AUTO_NMI_EXCP          EXCP_TABLE+0x180  /* default nmi exception vector */
#define  AUTO_EXT_EXCP          EXCP_TABLE+0x1c0  /* default external exception */
#define  NMI_EXCP               EXCP_TABLE+0x280  /* nmi exception vector (arbitrary address) */
#define  EXT_EXCP               EXCP_TABLE+0x2c0  /* external exception  (arbitrary address) */
#endif
```

**Code A-3. EOnCE_stopwatch.c**

```
/*
* Header file contains definitions of EOnCE memory-mapped register addresses,
* and definition of the WRITE_IOREG() macro.
*/
#include "EOnCE_registers.h"
#include "EOnCE_stopwatch.h"

unsigned long CLOCK_SPEED = 300000000;

static volatile long EOnCE_stopwatch_timer_flag;    /*Global dummby variable*/


void EOnCE_stopwatch_timer_init()
{
        WRITE_IOREG(EDCA1_REFA,(long)&EOnCE_stopwatch_timer_flag);
                                 /* Address to snoop for on XABA */
        WRITE_IOREG(EDCA1_REFB,(long)&EOnCE_stopwatch_timer_flag);
                                 /* Address to snoop for on XABB */
        WRITE_IOREG(EDCA1_MASK,MAX_32_BIT);
                                 /* No masking is performed in address comparison */
        WRITE_IOREG(EDCA1_CTRL,0x3f06);
                                 /* Detect writes on both XABA and XABB */
}

void EOnCE_LED_init()
{
        *((long *)EE_CTRL) &= ~(3<<2);                  /* Toggle EE1 when event1 happens */
}

void EOnCE_stopwatch_timer_start()
{
    WRITE_IOREG(ECNT_VAL,MAX_32_BIT);             /* Countdown will start at (2**32)-1 */
    WRITE_IOREG(ECNT_EXT,0);               /* Extension will count up from zero */
    WRITE_IOREG(ECNT_CTRL,0x12c);
                         /* Counting will be triggered by detection on EDCA1 */
    EOnCE_stopwatch_timer_flag = 0;
                         /* This write to the flag triggers the counter */
}

void EOnCE_stopwatch_timer_stop(unsigned long *clock_ext, unsigned long *clock_val)
{
    WRITE_IOREG(ECNT_CTRL,0);                    /* Disable event counter */
    READ_IOREG(ECNT_VAL,*clock_val); /* Save ECNT_VAL in program variable */
    READ_IOREG(ECNT_EXT,*clock_ext); /* Save ECNT_EXT in program variable */
    *clock_val = (MAX_32_BIT-*clock_val); /* Adjust for countdown */
}

void EOnCE_LED_off()
{
    EOnCE_stopwatch_timer_flag = 0;                 /* Create an EDCA1 event */
}

unsigned long Convert_clock2time(unsigned long clock_ext, unsigned long clock_val, short option)
{
    unsigned long result;
    switch(option)
    {
        case EONCE_SECOND:
            result= clock_ext*MAX_32_BIT/CLOCK_SPEED + clock_val/CLOCK_SPEED;
            break;
        case EONCE_MILLISECOND:
            result= clock_ext*MAX_32_BIT/(CLOCK_SPEED/1000)
                    + clock_val/(CLOCK_SPEED/1000);
            break;
        case EONCE_MICROSECOND:
            result= clock_ext*MAX_32_BIT/(CLOCK_SPEED/1000000)
                    + clock_val/(CLOCK_SPEED/1000000);
            break;
        default: result=0;                          /* error condition */
            break;
    }
    return result;
}
```

**Code A-4. autocorr.c**

```
#include "EOnCE_stopwatch.h"
#include <prototype.h>
#include <stdio.h>

#define L_WINDOW 240
# define SIGN_32 (1U << 31U)

Word16 Autocorr (Word16 x[], Word16 m, Word16 r_h[], Word16 r_l[],
                 const Word16 wind[]);
void L_Extract (Word32, Word16 *, Word16 *);

Word16 test_x[240] =
{
  -116, -179, -258, -128, 321, 714, 757, 640, 648, 867, 840, 1181,
  1071, 786, 760, 617, 803, 697, 559, 64, 164, 492, 437, 413, 167,
  -170, -75, 97, 27, 546, 618, 103, 51, 371, 104, -410, -387, -178,
  -151, -180, -213, -295, -430, -369, -268, -407, -387, -171, -217,
  -121, 74, -385, -651, -651, -296, -215, 178, 135, -127, -210, -195,
  105, 78, -270, -141, 271, 99, 135, 204, -8, -302, -160, -82, -389,
  -737, -616, -435, -521, -500, -396, -248, -293, -121, 263, 129, 203,
  584, 654, 659, 826, 452, -263, -468, -345, -208, -204, 0, 502, 518,
  323, 600, 422, -2, -6, 224, 206, 110, 4, -159, -478, -762, -806,
  -1008, -912, -412, 160, 374, 240, 154, 93, -71, -101, 84, 14, 107,
  71, -239, -365, -337, -718, -1042, -312, 188, 509, 647, 413, 442,
  530, 525, 295, -160, -538, -326, 388, 658, 396, 211, 242, 424, 250,
  310, 425, -21, -190, 362, 510, 68, -210, -361, -668, -269, 33, -126,
  -259, 4, 109, 327, 509, 220, -216, -404, -132, 151, -68, -248, -185,
  -170, -257, -160, -252, -301, 282, 575, 557, 897, 835, 373, 58,
  -192, -249, -347, -390, -198, -169, -206, -123, -91, -17, 54, -48,
  281, 484, 377, 318, 377, 470, -15, -461, -360, -445, -464, -682,
  -759, -529, 85, 232, 198, 485, 907, 426, 319, 810, 1056, 648, 188,
  1, -578, -790, -568, -84, -7, -188, -74, -166, -38, 213, 13, -168,
  101, -47
};

Word16 test_m = 10;

Word16 test_r_h[11] =
{
  28714, 21991, 11202, 4479, -494, -4007, -5339, -6829, -8122, -6165, -1943
};

Word16 test_r_l[11] =
{
  11232, 3256, 3463, 16354, 6929, 13451, 31455, 6034, 19039, 21655, 2132
};

Word16 test_wind[240] =
{
  2621, 2623, 2627, 2634, 2644, 2656, 2671, 2689, 2710, 2734, 2760,
  2789, 2821, 2855, 2893, 2933, 2975, 3021, 3069, 3120, 3173, 3229,
  3288, 3350, 3414, 3481, 3550, 3622, 3697, 3774, 3853, 3936, 4021,
  4108, 4198, 4290, 4385, 4482, 4582, 4684, 4788, 4895, 5004, 5116,
  5230, 5346, 5464, 5585, 5708, 5833, 5960, 6090, 6221, 6355, 6491,
  6629, 6769, 6910, 7054, 7200, 7348, 7498, 7649, 7803, 7958, 8115,
  8274, 8434, 8597, 8761, 8926, 9093, 9262, 9432, 9604, 9778, 9952,
  10129, 10306, 10485, 10665, 10847, 11030, 11214, 11399, 11586,
  11773, 11962, 12152, 12342, 12534, 12727, 12920, 13115, 13310,
  13506, 13703, 13901, 14099, 14298, 14497, 14698, 14898, 15100,
  15301, 15504, 15706, 15909, 16112, 16316, 16520, 16724, 16928,
  17132, 17337, 17541, 17746, 17950, 18155, 18359, 18564, 18768,
  18972, 19175, 19379, 19582, 19785, 19987, 20189, 20390, 20591,
  20792, 20992, 21191, 21390, 21588, 21785, 21981, 22177, 22372,
  22566, 22759, 22951, 23143, 23333, 23522, 23710, 23897, 24083,
  24268, 24451, 24633, 24814, 24994, 25172, 25349, 25525, 25699,
  25871, 26042, 26212, 26380, 26546, 26711, 26874, 27035, 27195,
  27353, 27509, 27664, 27816, 27967, 28115, 28262, 28407, 28550,
  28691, 28830, 28967, 29102, 29234, 29365, 29493, 29619, 29743,
  29865, 29985, 30102, 30217, 30330, 30440, 30548, 30654, 30757,
  30858, 30956, 31052, 31146, 31237, 31326, 31412, 31495, 31576,
  31655, 31730, 31804, 31874, 31942, 32008, 32071, 32131, 32188,
  32243, 32295, 32345, 32392, 32436, 32477, 32516, 32552, 32585,
  32615, 32643, 32668, 32690, 32709, 32726, 32740, 32751, 32759,
  32765, 32767, 32767, 32097, 30112, 26895, 22576, 17333, 11380, 4962
};

Word16 ref_result = 6;

void PLL_setup_300MHz()
{
    asm("move.l #$80030003,PCTL0");
    asm("move.l #$00010000,PCTL1");
}

int main(void)
{
  int rc = 0;
  unsigned long clock_ext, clock_val,time_us;

  PLL_setup_300MHz();
  EOnCE_stopwatch_timer_init();
  EOnCE_LED_init();
  EOnCE_stopwatch_timer_start();

  if (Autocorr(test_x, test_m, test_r_h, test_r_l, test_wind) != ref_result) {
    rc = 1;
  }

  EOnCE_stopwatch_timer_stop(&clock_ext, &clock_val);
  EOnCE_LED_off();
  time_us=Convert_clock2time(clock_ext, clock_val,EONCE_MICROSECOND);
  printf("time consumed by function is %d us\n", time_us);
  return rc;
}

Word16 Autocorr (Word16 x[], Word16 m, Word16 r_h[], Word16 r_l[],
          const Word16 wind[])
{
  Word16 i, j, norm;
  Word16 y[L_WINDOW];
  Word32 sum;
  Word16 overfl, overfl_shft;

  for (i = 0; i < L_WINDOW; i++) {
    y[i] = mult_r (x[i], wind[i]);
  }
  overfl_shft = 0;

  do {
    overfl = 0;
    sum = 0L;

    for (i = 0; i < L_WINDOW; i++) {
      sum = L_mac (sum, y[i], y[i]);
    }

    if (L_sub (sum, MAX_32) == 0L) {
      overfl_shft = add (overfl_shft, 4);
      overfl = 1;

      for (i = 0; i < L_WINDOW; i++) {
        y[i] = shr (y[i], 2);
      }
    }
  } while (overfl != 0);
  sum = L_add (sum, 1L);

  norm = norm_l (sum);
  sum = L_shl (sum, norm);
  L_Extract (sum, &r_h[0], &r_l[0]);

  for (i = 1; i <= m; i++) {
    sum = 0;

    for (j = 0; j < L_WINDOW - i; j++) {
      sum = L_mac (sum, y[j], y[j + i]);
    }

    sum = L_shl (sum, norm);
    L_Extract (sum, &r_h[i], &r_l[i]);
  }
  norm = sub (norm, overfl_shft);

  return norm;
}

void L_Extract (Word32 L_32, Word16 *hi, Word16 *lo)
{
  *hi = extract_h (L_32);
  *lo = extract_l (L_msu (L_shr (L_32, 1), *hi, 16384));

  return;
}
```