Using the SC140/SC1400 Enhanced On-Chip Emulator Stopwatch Timer

By Kim-Chyan Gan and Yuval Ronen

A stopwatch timer is an apparatus for measuring the exact duration of an event. Measuring code execution time on a 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 for implementing a stopwatch timer on DSPs based on the StarCore™ SC140/SC1400 cores using the built-in features of the enhanced on-chip emulation module (known as EOnCE or OCE10), hereafter referred to as the emulator.

Although many devices with embedded DSPs provide application-specific timer blocks, these timers cannot be used for debugging without interference from the application itself. The ability to use the enhanced emulator as a stopwatch timer gives us non-intrusive timing capabilities. This application note describes how to set up the stopwatch timer using the SC140 code in an application or using the Metrowerks® CodeWarrior® debugger.[1]

Code examples illustrate the use of the stopwatch timer on the SC140 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.

1. This application note was written for the Beta v.1.0 release of Metrowerks CodeWarrior. Screen captures of CodeWarrior tools may vary in future versions.

CONTENTS

freescale™ semiconductor

1 Stopwatch Timer Basics

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

1.1 Features

The emulator 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 while an application executes.

Conversion between clock cycles and absolute time, based on the operating clock frequency of the DSP, is described in **Section 2.4**.

1.2 Resources

The emulator stopwatch timer requires use of these resources:

- One emulator event detector (of the six available on each DSP)
- Emulator event counter
- Program memory of 724 bytes

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

1.3 Implementation

The stopwatch timer allocates a variable in memory to serve as the target for memory write operations. The emulator event detector is set up to detect writes to this flag variable. When an event detector is set up to detect memory access operations, it is necessary to specify which of the two data memory buses should be “snooped”. Because the bus is selected 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 event counter, which starts counting down.

The emulator 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 emulator memory-mapped event counter control register. When this operation completes, the cycle countdown halts. Now, you can read out the values of the emulator event counter registers and translate them into an elapsed number of cycles or elapsed absolute time.

2 Setting Up the Stopwatch Timer In an Application

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

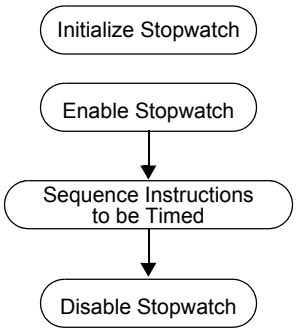

Figure 1. Sequence of Operations

2.1 Initializing the Stopwatch Timer

The C code to set up the stopwatch timer is shown in **Example 1**. This code contains the definitions of the emulator memory-mapped address registers.

Example 1. Event Detector Set-up 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 dummy 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 `EOnCE_registers.h` header file contains the macro definitions, such as EDCA1_MASK, which provides each memory-mapped register's memory address. This header file also defines the C macros: READ_IOREG() and WRITE_IOREG(). These macros simplify the read and write operations on these registers.

To initialize the stopwatch timer, set up the Address Event Detection Channel (EDCA), which triggers the cycle countdown. The emulator supports six EDCAs. The implementation presented in this application note uses EDCA1, though this choice is arbitrary. To set up the EDCA, you must initialize the following four registers:

- 32-bit EDCA reference value register A (EDCA*i*_REFA).
- 32-bit EDCA reference value register B (EDCA*i*_REFB).

- 32-bit EDCA mask register (EDCA*i*_MASK).
- 16-bit EDCA control register (EDCA*i*_CTRL).

2.1.1 Event Detector Control

The EDCA1_CTRL register controls the behavior of the EDCA. The fields of the EDCA*1*_CTRL register are shown in **Figure 2**.

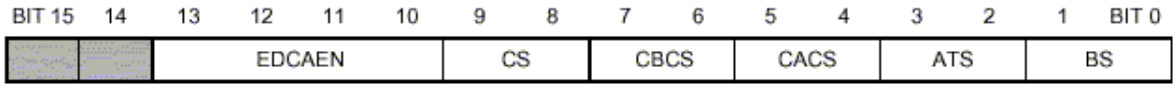

Figure 2. EDCA Control Register (EDCA*1*_CTRL)

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 the address matches on either comparator A or comparator B
CBCS	00	“address match” is detected when the sampled bus value **equals** the value in EDCA_REFB.
CACS	00	“address match” is detected when the sampled bus value **equals** the value in EDCA_REFA.
ATS	01	Detect write accesses only
BS	10	The sampled buses are XABA and XABB

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

2.1.2 Address Comparison Set-up

EDCA address comparison detects writes to the stopwatch timer flag variable. Because writes can occur on either of the two data memory buses, both EDCA*1*_REFA and EDCA*1*_REFB are set up to contain the address of the stopwatch timer flag variable.

The EDCA*1*_MASK register allows masking of address bits when the sampled address is compared with those in the EDCA1_REFA and EDCA1_REFB registers. Our implementation of the stopwatch timer uses all 32-bits of the flag variable address, so EDCA*1*_MASK is set to 0xFFFFFFFF.

2.2 Starting the Stopwatch Timer

The C code to start the stopwatch timer is shown in **Example 2**. This code contains the write commands that trigger the event counter.

Example 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 */
}
```

The counter registers must be initialized before the stopwatch timer is triggered. 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)

When these initialization are complete, the C code triggers the stopwatch timer and cycle counting commences.

2.2.1 Event Counter Control

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

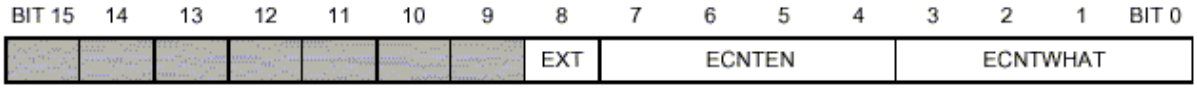

Figure 3. Event Counter Control Register (ECNT_CTRL)

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 is enabled when EDCA1 detects an event.
ECNTWHAT	1100	The counter advances on each core clock cycle.

2.2.2 Counter Registers

ECNT_VAL is a countdown counter, and ECNT_EXT is a count-up counter. ECNT_VAL decrements on each occurrence of an event, as specified in the control register. ECNT_EXT increments 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.

2.3 Stopping the Stopwatch Timer

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

Example 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 cleared to zero. After the stopwatch timer stops, the routine copies the values of the ECNT_VAL and ECNT_EXT registers into program variables. Therefore, the stopwatch timer can be used 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.

2.4 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**.

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

EXT is the value in ECNT_EXT, VAL is the value in ECNT_VAL, and Clock Speed is measured in MHz. **Example 4** 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.

Example 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;
}
```

2.5 Putting it All Together

Example 5 shows the stopwatch timer using the routines described so far.

Example 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.

2.6 Adapting Stopwatch Timer Code to SC140 Devices

The stopwatch timer implementation controls the emulator 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 emulator registers is the same across SC140-based devices. Therefore, when the stopwatch timer code is adapted to 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 the emulator is 0x00EFFE00. Consult the user manual of the specific DSP for the value of this C macro.

3 Setting Up the Stopwatch Timer In the Debugger

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, execution times can be measured by setting up the stopwatch timer within the Metrowerks Code Warrior SC140 debugger, as explained in this section.

3.1 Initializing the Stopwatch Timer

When the stopwatch timer is set up within the debugger, the triggering event executes 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 →symbol 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 the menu bar, then highlight EOnCE Configurator, and finally choose the EDCA1 tab.

3.1.1 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 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**.

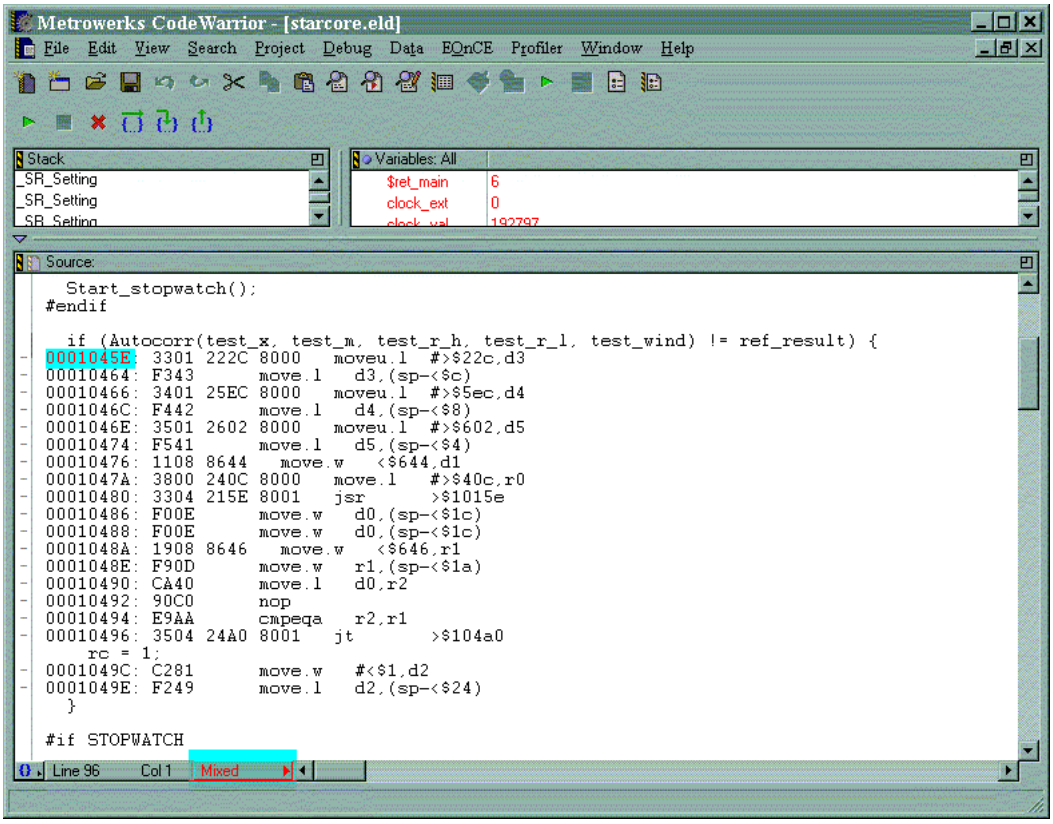

Figure 4. Finding the Starting Address in Debugger

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

1. Choose **EONCE →EONCE CONFIGURATOR > EDCA1.**
2. Click PC in the **BUS SELECTION** box.
3. Enter the starting address of the function or a sequence of instructions into the **COMPARATOR A HEX 32-BITS** box.
4. Click **ENABLE** in the **ENABLED AFTER EVENT ON** box.

Figure 5 shows the event detection settings after these steps are performed. For details on emulator configuration in the CodeWarrior tools, refer to [3].

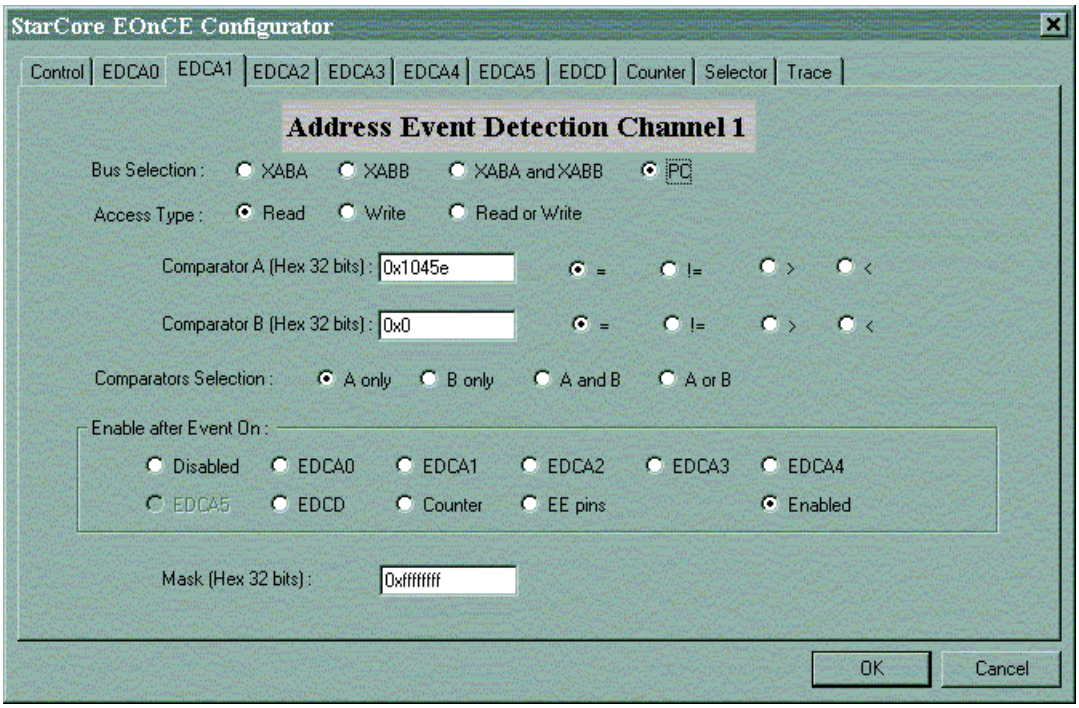

Figure 5. Event Detection Settings

3.1.2 Setting Up the Event Counter

The event counter is configured to the mode described in **Section 2.2.1**, *Event Counter Control,* on page 5, except that this time the configuration occurs in the debugger windows:

1. Choose **EONCE →EONCE CONFIGURATOR →COUNTER**.
2. Click on **CORE CLOCK** in the **WHAT TO COUNT** box.
3. Click **EDCA1** in the **ENABLE AFTER EVENT ON** box.
4. Type “0xFFFFFFFF” in the **EVENT COUNTER VALUE (HEX 32)** box.
5. Check the box in the left side of **EXTENSION COUNTER VALUE (HEX 32 VALUE)**.

The result of this procedure is shown in **Figure 6**.

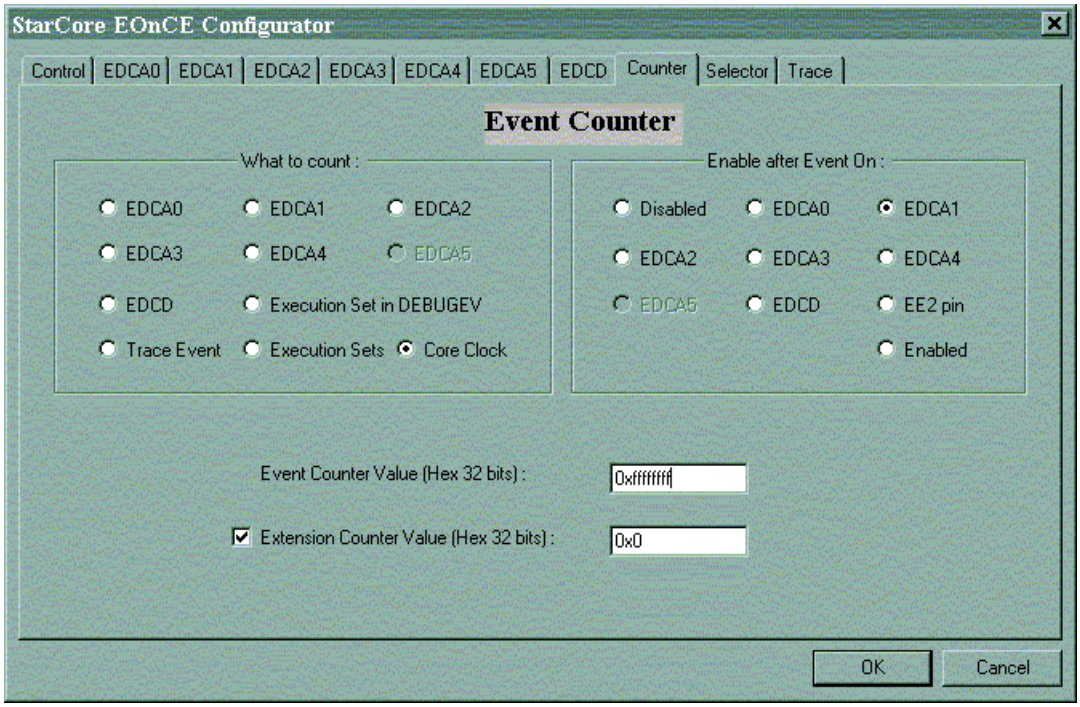

Figure 6. Event Counter Settings

3.2 Stopping the Stopwatch Timer

The stopwatch timer stops when execution of the application halts at a breakpoint. Set up the breakpoint at the point in the application where you want such halts to occur. When the breakpoint is in place, the debugger can be instructed to run the application.

When the executing 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** shows an event counter dialog box after the debugger halts at the breakpoint. Because the countdown counter is initialized to the maximum value (0xFFFFFFFF), the difference between the 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 2.4**, *Converting Cycles to Actual Time,* on page 6.

After the debugger stops at the breakpoint, the counter, which is set up in sleep mode and is enabled by the events at the beginning of the code, is now enabled and continues to count (see the **Enabled after Event on** column in **Figure 7**). However, continued counting is irrelevant because the SC140 clock count has been achieved.

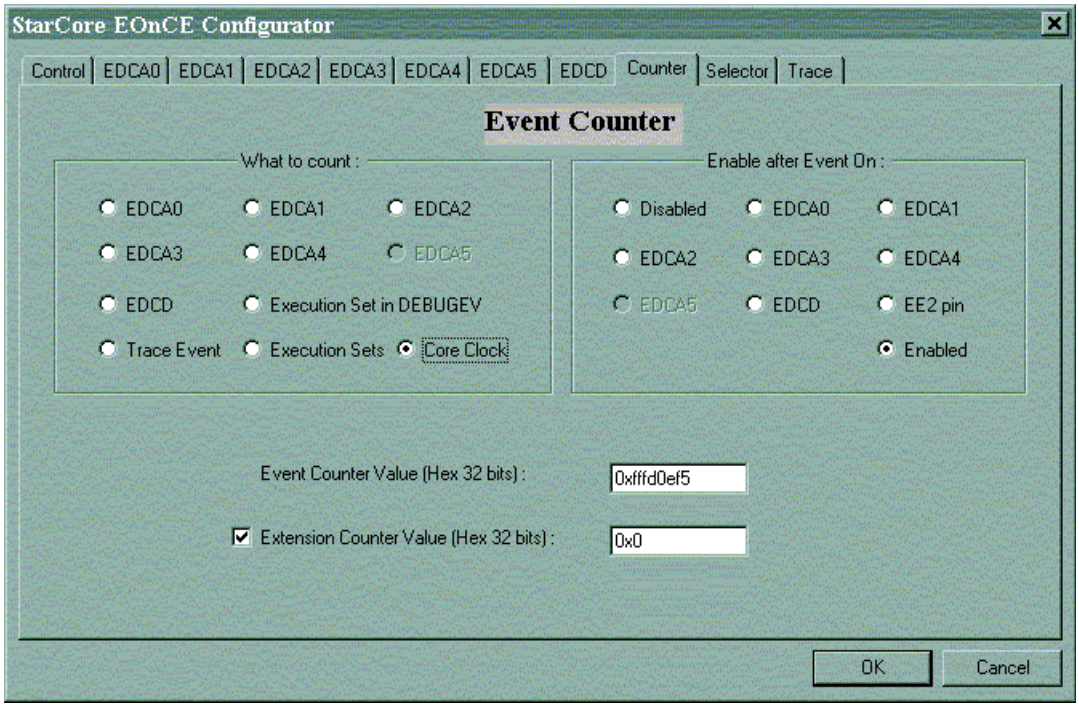

Figure 7. Event Counter Dialog Box When Debugger Halts at Breakpoint

4 Setting Up the System Clock Speed

Every SC140-based device contains a phase lock loop (PLL) to control operating frequency. The frequency of the device is governed by the frequency control bits in the PLL control register, as defined in **Equation 2**.

$$Fdevice = \frac{Fext \times \left(MFI + \frac{MFN}{MFD}\right)}{PODF \times PDF} \qquad \textbf{Equation 2}$$

Where:

- MFI (multiplication factor integer), MFN (multiplication factor numerator), MFD (multiplication factor denominator), and PODF (post division factor) are defined in the PCTL1 register.
- PDF (predivision factor) is defined in the PCTL0 register.
- F_{ext} is external input frequency to the device at the EXTAL pin.
- F_{device} is the device operating frequency.

The range of values of these terms is described in [1].

Figure 8 and **Figure 9** illustrate the PLL registers; PCTL0 and PCTL1, respectively.

Figure 8. Programming Model of PCTL0

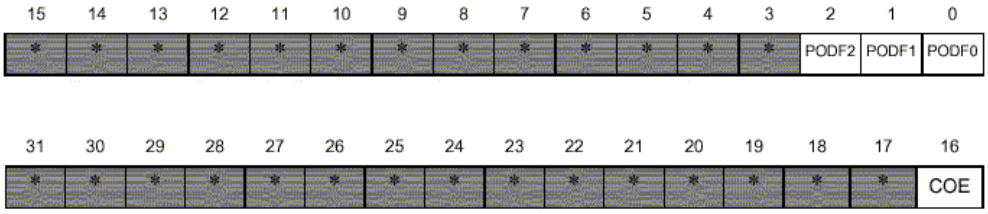

Figure 9. Programming Model of PCTL1

4.1 Setting Up the PLL in Software

The clock frequency of the SC140/SC1400 core can be set up in either software or hardware. This section describes how to set up the core 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 **Example 6**.

Example 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 core for operation at 300 MHz, the PCTL0 and PCTL1 registers 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 F_{chip} is calculated as expressed in **Equation 3**.

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

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

4.2 Setting Up the PLL in Hardware

During the assertion of hardware reset, the values of all the PLL hardware configuration pins are sampled into the clock control registers (PCTL0 and PCTL1). Thus, the core frequency can be set up 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 details on jumper configuration of SDP, refer to [2].

5 Verifying Correct Set-up

This section describes how to verify that the system is set up correctly and that the emulator stopwatch timer measurements are as described in **Section 2**, *Setting Up the Stopwatch Timer In 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).

5.1 Using the LED on the SDP

The implementation described in this section is based on the configuration of the SDP. In SDP, each EE1 pin of the emulator is connected to an LED. The following implementation is based on the ability to program the emulator to toggle the output value on its pins whenever an event is detected by one of the emulator event detection channels. Our implementation toggles the output value on the EE1 pin when the stopwatch timer starts or stops running. This capability requires just a small enhancement to the stopwatch timer software that is presented in **Example 6**.

5.1.1 Setting Up EE1

The functionality of the emulator pins is controlled through the EOnCE pins control register (EE_CTRL). **Figure 10** displays the structure of this register.

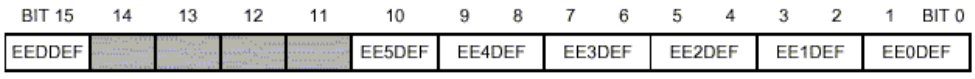

Figure 10. EE Pins Control Register (EE_CTRL)

In EE_CTRL, the EE1DEF field is set to 00, which signifies an output signal when detected by EDCA1. The remaining fields in EE_CTRL are irrelevant because they are not used. **Example 7** shows the set-up code for EE control registers.

Example 7. EOnCE_LED_init()

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

5.1.2 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 should be taken to perform this write only after execution of EOnCE_stopwatch_timer_stop().

Example 8. Turn LED Off

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

5.2 Testing the Stopwatch Timer

The program in **Example 9** sets up the stopwatch timer and measures the time it takes to execute two loops whose duration is built into the program. The measured code sequences are constructed to take 5 seconds and 3 seconds, respectively. These durations are constructed as the code samples the emulator counter and loop until the expected number of clock cycles have passed. Trying this code prior to measuring the target application is recommended as a means of verifying correct system set-up. If the times measured are not correct, check the PLL set-up and the values of the clock speed and the memory-mapped register base (as set in the header files).

Example 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;
}
```

6 Conclusion

This application note presents two techniques for measuring the execution speed of software running on a DSP device based on the StarCore SC140 or SC1400 core, using the enhanced on-chip emulator. 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. Examples in this application note demonstrate setting up the SC140 Phase Lock Loop, and software control of the LED available on the Software Development Platform.

7 References

The following documents are available at the Freescale web site listed on the back cover of this document:

[1] SC140 DSP Core Reference Manual (MNSC140CORE)

[2] Device Application Development System (ADS) Reference Manual.

[3] SC1000-Family Processor Core Reference Manual.

[4] OCE10 On-Chip Emulator Reference Manual.

NOTES:

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

Document Order No.: AN2090
Rev. 1
1/2005

freescale™
semiconductor