Embedded System Timers

Details of independent timers build into the Blackfin

Watchdog Timer can be used to prevent the OS system from locking up with one task hogging all the CPU time
Review for Midterm

- The use of timers is new material
- BUT -- The way you have to bit-bash the timer registers is the same as you bit bashed the coffee pot control register bits in Assignment 1 and the GPIO register bits in Lab. 1

- So treat a lot of this material as review for Midterm, especially the need to AND and OR to manipulate bit patterns
3 Timers available on Blackfin
Details available in hardware notes

- Watchdog timer – Hardware Reference 15-49
- Core timer – Hardware Reference 15-45
  - Assignment 2 – to automate UpDateSimulator
- General purpose timers 15-1
  - Pulse Width Modulation
  - Pulse Width Count and Capture – Lab. 3
  - External Event
- Application of timers to provide code safety
- Introduction to timer interrupts
Basic concept – watchdog timer

- You are responsible for developing code for an embedded system running unattended in the field.
- Something goes unexpectedly wrong -- This external signal from some equipment NEVER ARRIVES.
  - This one piece of code (a subroutine) keeps running, and running and running
  - Other parts of the system self-destruct as there are no control signals sent to them
- Fix this issue with a “WatchDog Timer”
  - The use of software and hardware watchdog timers are the subject on one TTCOS article
WatchDogTimer
Demonstrate (code provided)

```c
int main( ) {
    SetUpWatchDogTimerASM( );       // Function names change from year to ye
    StartWatchDogTimerASM( );        //  see uTTCOS_Help.txt
    ResetWatchDogTimerASM( );        // or use CNTRL-SPACE autocomplete

    for ( ; ; ) { // For-ever loop
        CriticalTask1( );
        ResetWatchDogTimerASM( );
        CriticalTask2( );
        ResetWatchDogTimerASM( );
        CriticalTask3( );
        ResetWatchDogTimerASM( );
    }
}
```

If any task hangs up (does NOT return in time to Reset the WatchDog Timer)
then the system can be made to reboot, or send error message etc.
HANGS UP – because an expected external equipment signal does not arrive
because of equipment failure (SW5 connection “broken” in Lab. 2 Task 1)
int main(void) {
    int maxNumberTasks = 20;

    printf("Dr. Smith's audio demo for Lab. 3 Tasks 1 and 2 -- 9th October 2009\n");

    TTCOS_511Init();

    TTCOS_InitializeSystemWatchDogTimer(); // Set up the watchdog timer registers
    TTCOS_StartSystemWatchDogTimer(); // Start the watchdog time counting down from a maximum

    TTCOS_InitSwitches();
    TTCOS_InitLED();

    TTCOS_AddTask(ClearLED, NO_DELAY, RUN_ONCE);
    TTCOS_AddTask(ReadPFSwitches, NO_DELAY, EXECUTE_EVERY_SECOND);
    TTCOS_AddTask(LED1and2_Show, NO_DELAY, EXECUTE_EVERY_SECOND);

    TTCOS_Start(maxNumberTasks);
    // Execution time of Dispatch( ) and Update( ) improved by specifying maxNumberTasks

    while (1) {

        // Wait, in low power mode, for an interrupt
        // The interrupt service routine calls TTCOS_Update( )
        TTCOS_GoToSleep();

        // Run all the tasks in the system according
        // to whether their delays have expired
        TTCOS_DispatchTasks();

        // If one of the tasks "hang" then the code will never get to
        // this point and the watchdog ISR will take over control of the embedded system
        TTCOS_FeedSystemWatchDogTimer(); // Re-Start the watchdog time counting down from a maximum

        // QUESTION -- WHAT SHOULD THE WATCHDOG ISR DO ONCE IT HAS TAKEN OVER??????
    }
}
Watchdog Timer Operation

To use the watchdog timer:

1. Set the count value for the watchdog timer by writing the count value into the Watchdog Count register (WDOG_CNT). Note that loading the WDOG_CNT register while the watchdog timer is not enabled will also pre-load the WDOG_STAT register.

2. In the Watchdog Control register (WDOG_CTL), select the event to be generated upon timeout.

3. Enable the watchdog timer in WDOG_CTL. The watchdog timer then begins counting down, decrementing the value in the WDOG_STAT register. When the WDOG_STAT reaches 0, the programmed event is generated. To prevent the event from being generated, software must reload the count value from WDOG_CNT to WDOG_STAT by executing a write (of any value) to WDOG_STAT, or must disable the watchdog timer in WDOG_CTL before the watchdog timer expires.
Example code -- Post-Lab Quiz 1 Review
Setting the WATCHDOG TIMER registers

```c
SetUpWatchDogTimerCPP( ) {
    *pWDOG_CNT = (unsigned long) 0x08000000
    ssync( ) ;  How long is 0x0800 0000 ticks?
    // Watchdog Count Register and Watchdog Register
}
```

Assignment 2 –or Midterm question – convert to assembly code

Watchdog Count Register (WDOG_CNT)

![Watchdog Count Register Diagram]

Figure 15-30. Watchdog Count Register
The Watchdog Count register holds the programmable count value. A valid write to the Watchdog Count register also preloads the Watchdog counter. For added safety, the Watchdog Count register can be updated only when the watchdog timer is disabled. A write to the Watchdog Count register while the timer is enabled does not modify the contents of this register.

```c++
SetUpWatchDogTimerCPP(void) {
    // Review C++ access of registers
    *pWDOG_CTL = (unsigned short) 0x0AD0
    ssync( ) ;  Watch dog disable

    *pWDOG_CNT = (unsigned long) 0x08000000
    ssync( ) ;  How long is 0x0800 0000 ticks?

    ssync( ) ;  // System synchronize – Count read/write ops.
    // Finish all uncompleted R / W operations
}
```
Watchdog Status Register
WDOG_STAT

WDOG_STAT Register

The 32-bit Watchdog Status register (WDOG_STAT) contains the current count value of the watchdog timer. Reads to WDOG_STAT return the current count value. When the watchdog timer is enabled, WDOG_STAT is decremented by 1 on each SCLK cycle. When WDOG_STAT reaches 0, the watchdog timer stops counting and the event selected in the Watchdog Control register (WDOG_CTL) is generated.

Values cannot be stored directly in WDOG_STAT, but are instead copied from WDOG_CNT. This can happen in two ways:

- While the watchdog timer is disabled, writing the WDOG_CNT register pre-loads the WDOG_STAT register.

- While the watchdog timer is enabled, writing the WDOG_STAT register loads it with the value in WDOG_CNT.

When the processor executes a write (of an arbitrary value) to WDOG_STAT, the value in WDOG_CNT is copied into WDOG_STAT. Typically, software sets the value of WDOG_CNT at initialization, then periodically writes to WDOG_STAT before the watchdog timer expires. This reloads the watchdog timer with the value from WDOG_CNT and prevents generation of the selected event.

The WDOG_STAT register is a 32-bit unsigned system memory-mapped register that must be accessed with 32-bit reads and writes.
My actual working code (in ASM)
Make sure you can read for midterm

```asm
#include <defbf533.h>
#include <macros.h>

.section program;
global _SmithSetUpWatchDogTimerASM__Fv;
_SmithSetUpWatchDogTimerASM__Fv:
  LINK 16;
  P0.H = hi(WDOG_CTL);
P0.L = lo(WDOG_CTL); // Get the control register address into P0
  R1.L = 0x0AD0;
  W[P0] = R1; // Watchdog control register -- disable
  SSYNC;
  CSYNC;
  R1.L = 0x8AD6; // Need to do write one to clear bit 15
  W[P0] = R1; // Set Watchdog -- for no event (bits 1 and 2)
  SSYNC;
  CSYNC;
  P0.H = hi(WDOG_CNT);
P0.L = lo(WDOG_CNT); // Get the address into P0
  R0.L = 0x0000;
  R0.H = 0x0800;
  [P0] = R0; // Watchdog Count Register
  SSYNC;
  CSYNC;
  P0 = [FP + 4];
UNLINK;
_SmithSetUpWatchDogTimerASM__Fv.end:
  JUMP (P0);
```

**NOTE:** WDOG_CTL bit 15 is a W1C bit

Write one to clear means Write one to make 0

NOT 0x0AD6 but 0x8AD6 to clear bit 15 (W1C)
Starting the watchdog timer

Watchdog Control Register (WDOG_CTL)

0xFFC0 0200

TRO - W1C
0 - Watchdog timer has not expired
1 - Watchdog timer has expired

ICTL[1:0]
00 - Generate reset event
01 - Generate NMI
10 - Generate GP interrupt
11 - Disable event generation

TMR_EN[7:0]
0xAD - Counter disabled
All other values - Counter enabled

Figure 15-32. Watchdog Control Register

**Post-Lab 1 Quiz review**

_StartWatchDogTimerASM__Fv:

P0.H = hi(WDOG_CTL)
P0.L = lo(WDOG_CTL) // Get the address into P0
// Put 0x0000 into R1.L; Will force a "RESET EVENT"
// RESET -- Start the processor code from the beginning
(FORCED TOTAL RESTART AS IF SYSTEM JUST POWER-EDUP
R1.L = 0x0000;
W[P0] = R1; // Watchdog control register – counter enabled
ssync;

This more an INIT than a Start
Example code

Watchdog can trigger many events

```
section program;
global _SmithStartWatchDogTimerASM__Fv;
_SmithStartWatchDogTimerASM__Fv:
    LINK 16;

    P0.H = hi(WDOG_CTL);
P0.L = lo(WDOG_CTL);    // Get the address into P0

    R1.L = 0x0ADO;
W[P0] = R1;            // Watchdog control register -- disable
SSYNC;
CSYNC;

    // Put 0x0000 into R1.L; Will force a RESET EVENT
    // Put 0x0002 will cause NMI,
    // Put 0x0006 normal GP interrupt;
    // Put 0x0006 no event, just count
R1.L = 0x0002;        // Force NMI event
W[P0] = R1;
ssync;
csync;

    P0 = [FP + 4];
UNLINK;
_SmithStartWatchDogTimerASM__Fv.end:
    JUMP (P0);
```

Next lectures

What’s an interrupt and what is the difference to an NMI (Non-maskable interrupts)?

Look in the Blackfin Hardware manual for correct settings
“Flight control”
using Watchdog based security system

InitFlashASM( ); // Set up the Flash memory – Lab. 1
InitFlashPortASM( ); // Set up Flash LED port – Lab. 1
WriteFlashLEDASM(0); // Clear LED panel – Lab. 1

StopCycleCounterASM( ); // Stop and then reset cycle counter
ResetCycleCounterASM( ); // -- Assignment 3

WatchDogTimerHowled = 0;
ActivateWatchDogTimer( );

while (WatchDogTimerHowled != 1) {
    ResetWatchDogTimer( );
    GoControlTheAeroPlane( );
    CheckIfCrewAreAwake( );
}

StopWatchDogTimer( ); // Should never get here unless Watch Dog Timer 'Howled
WriteFlashLEDASM(0); // Clear LED panel – Lab. 1
CrewErrorOccurred( );
Watchdog video game (JFY - just for you!)

```c
volatile long int quit = 0;
volatile long int correctAnswerFound = 0;
volatile extern long int watchdogTimerWon;

void SmithDemoWatchDogTimerConcept(void) {
    Initialize_ProgrammableFlagsASM(); // From your Lab. 1 code
    InitializeLEDInterfaceASM();

    SmithSetUpWatchDogTimerASM(); // Number of processor ticks
    register_handler(ik_nmi, SmithWatchDogTimerISR);

#define SWITCHPOSITIONS 0x0F00
#define NUMBEROFGAMES 8
long int masterKey[NUMBEROFGAMES] = { 1, 2, 3, 5, 7, 11, 13, 17};

puts("You have to press the corresponding switches to LEDs shown");
puts("before the WatchDog Timer counts to zero (4 seconds)");
puts("When you succeed matching the switches and LEDs you get a new life");
puts("Otherwise -- you lose! \n GAME HAS STARTED");

SmithStartWatchDogTimerASM( );

long int whichGame = 0;

while (quit == 0) {
    WriteLEDASM(masterKey[whichGame]);

    long int answer = 0;
    while ((quit == 0) && (correctAnswerFound == 0)) {
        answer = (ReadProgrammableFlagBitsASM() & SWITCHPOSITIONS) >> 8;
        if (answer == masterKey[whichGame]) {
            correctAnswerFound = 1;
            SmithResetWatchDogTimerASM( ); // Correct Answer so give new
        puts("New life added");
    }
```
ResetWatchDogTimerCPP();
Practice for Midterm (Post Lab. 2 Quiz)?

- TAKE-HOME EXERCISE
- You write the required code – base on standard Blackfin Assembly language Stub

`ResetWatchDogTimerCPP:`

My original version – 24 lines of assembly code. A lot less in “C / C++”

However, I then re-read the manual – resetting the timer can be done in 4 lines of ASM code excluding LINK / UNLINK code
Functionality Test of Wand Wave thread

- Wait while Goldilock’s wand is raised and then lowered and then start (launch) a new Task
- A key issue with a task “waiting” – did you stop the other tasks from running?
- Once you have got the `wand wave’ thread working (at least demoed three times)
  - SW4 pressed for a short time – Kill Christine-the-Young
  - SW4 pressed for a long time – Kill Christine-the-Agressive
  - SW4 pressed for “just the right amount of time” -- Kill Christine-the-very-angry
Temperature Sensor -- Lab 3 “event driven

How can you time “High” time?

ANALOG DEVICES
TMP03
Temperature Sensor

Part of Lab. Kit
DON’T HOOK UP HOT
(Turn power off)

DON’T MIX UP
POWER (+5V) and
GROUND (0V)
CONNECTIONS

SIGNAL FROM TMP03 has this shape

HIGH

LOW

SIGNAL TO BLACKFIN -- hook into PF8 (SW1). Check using the same “when switch SW1 pressed, when released” code as in Lab. 1

+5V

GROUND
Counting amount of time thermal sensor signal is high -- TTCOS illegal

```c
while (1) {
    int countHigh = 0;    // Why must this be inside and not outside the loop
    int countLo = 0;

    while (SW1 is high) /* do nothing */;
    while (SW1 is low) /* do nothing */;    // Get to the START of the low-to-high transition

    while (SW1 is high) {    // Relative time high
        UseUpTime( );
        countHigh++;
    }
    while (SW1 is low) {    // Relative time high
        UseUpTime( );
        countLo++;
    }

    int temperature = CalculateTemperature(countHigh, countLow);
    DisplayTemperatureOnLEDs(temperature);
}
```

4 while loops needed – NOT 2 (key lab / exam concept)

Post Lab Quiz 1 Hints
We discussed doing IF and WHILE loops in ASM

Uses same ASM files
For reading PF flags as in Lab. 1

Uses WriteLEDASM( )
Why is this style of code uTTCOS illegal? How do you rewrite?

```c
while (1) {
    int countHigh = 0;
    // Why must these variables be inside and not outside the loop
    int countLo = 0;

    while (SW1 is high) /* do nothing */;  // Get to the START of the low-to-high transition
    while (SW1 is low) /* do nothing */;

    while (SW1 is high) {    // Relative time high
        // Relative time high
    }
    while (SW1 is low) {    // Relative time high
        // Relative time high
    }

    int temperature = CalculateTemperature(countHigh, countLow);
    DisplayTemperatureOnLEDs(temperature);
} 

Uses WriteLEDASM( )
```

4 while loops needed – NOT 2
(key lab / exam concept)
A C++ version of code -- based on "(interval counter) – Software part of Lab 1"

- Develop a routine `UseUpFixedAmountTimeCPP()` that uses up a fixed amount of time
- Use `My_ReadSwitchesASM()` to find when input signal goes high
  - When input goes high, call this Interval Counter routine
  - Keep calling the Interval counter until the input goes low
- Count how many times this routine must be called from `main()` while the temperature signal is HIGH

Could be constructed using “for-loop” construct

```c++
void UseUpFixedAmoutTimeCPP(unsigned long int timeToUse) {
    unsigned short int counter = 0;

    for (int num = 0; num <= timeToUse; num ++) {
        counter = counter; // Waste time
    }
    // Spin the wheels on the processor
}
```

Post Lab. 1 Question – how would you use `ReadLEDASM()` to wait until the input signal goes high – Do in either C++ or assembly code.
You set Core timer register TSCALE to 0 (decrement clock rate by 0 + 1)
You set register TPERIOD to 0x2000
You set register TCOUNT to 0x4000
You enable timer using control register TCNTL

TCOUNT is decreased by 1 until it reaches 0 (0x4000 system clock ticks)
When TCOUNT reaches 1, timer interrupt is caused and TCOUNT is reloaded with TPERIOD (0x2000) – counts down again
void UseUpFixedAmountTime(unsigned long int timeToUse) {
    1. Load the core timer counter register with parameter “time_to_use”
    2. Start the core timer – which causes the core-timer register to count down to zero
    3. While the core timer counter register != 0
        continue counter
    4. When the core timer count register equals 0 -- Return
}

Core timer changes at 500 MHz compared to uTTCOS changes of 1 or 2 kHz – so can get very accurate timing values
TCOUNT and TPERIOD registers

The Core Timer Count register (TCOUNT) decrements once every TSCALE + 1 clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated and the TINT bit of the TCNTL register is set.

Core Timer Count Register (TCOUNT)

0xFFF0 300C

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Reset = Undefined

Count Value[31:16]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Count Value[15:0]

Figure 15-27. Core Timer

TPERIOD Register

When auto-reload is enabled, the TCOUNT register is reloaded with the value of the Core Timer Period register (TPERIOD) whenever TCOUNT reaches 0.

Core Timer Period Register (TPERIOD)

0xFFF0 3004

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Reset = Undefined

Period Value[31:16]

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Period Value[15:0]
The Core Timer Scale register (TSCALE) stores the scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in the TSCALE register is 0, the counter register decrements once every clock cycle. If TSCALE is 1, the counter decrements once every two cycles.

Core Timer Scale Register (TSCALE)

0xFFE0 3008

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Reset = Undefined

Core Timer Control Register (TCNTL)

0xFFE0 3000

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Reset = Undefined

TINT - W1C
Sticky status bit
0 - Timer has not generated an interrupt
1 - Timer has generated an interrupt

TAUTORL
0 - Disable auto-reload feature. When TCOUNT reaches zero, the timer generates an interrupt and halts
1 - Enable auto-reload feature. When TCOUNT reaches zero and the timer generates an interrupt, TCOUNT is automatically reloaded with the contents of TPERIOD and the timer continues to count

TMPWR
0 - Puts the timer in low power mode
1 - Active state. Timer can be enabled using the TMREN bit

TMREN
Meaningful only when TMPWR = 1
0 - Disable timer
1 - Enable timer
New ASM instructions – Can be used to “safe-guard” OS lock-changes

- The INTERRUPT mask contains information (bits set or clear) about which devices are allowed to interrupt the processor
- CLI Rx
  - Save a copy of the “INTERRUPT” mask into data register Rx and then CLEAR the mask (The mask bit must be set for interrupts to occur – Clearing blocks all interrupts)
  - Post-Lab Quiz 1 Review – How done on GPIO interrupts
- STI Rx
  - Copy the bit pattern from data register Rx into the “INTERRUPT” mask, effectively reactivating all the devices that were allowed to interrupt the processor
void *UseUpFixedAmountTimeASM*

*(unsigned long int timeToUse) - 1*

Stop interrupts  -- Why is this needed

CLI R2;        // Clear interrupts  -- re-enable interrupts STI R2;

Stop the timer by changing the bits in the timer control register

P0.H = hi(TCNTL)
P0.L = lo(TCNTL)  // Get the address into P0
R1 = 0;
[P0] = R1;
SSYNC;

Post-Lab 1 Quiz review

Was [P0], W[P0] or B[P0] needed here?

Load the core timer counter register with parameter “time_to_use (R0)”

P0.H = hi(TCOUNT)
P0.L = lo(TCOUNT)  // Get the address into P0
[P0] = R0;
SSYNC;

Was [P0], W[P0] or B[P0] needed here?

Start the core timer by changing the bits in the timer control register

P0.H = hi(TCNTL)
P0.L = lo(TCNTL)  // Get the address into P0
R1 = 3;
[P0] = R1; Was [P0], W[P0] or B[P0] needed here?
SSYNC;
void UseUpFixedAmountTime
(unsigned long int timeToUse) - 2

Stop interrupts
Stop the timer
Load the core timer counter register with parameter “time_to_use”
Start the core timer

P0.H = hi(TCNTL)
P0.L = lo(TCNTL)  // Get the address into P0
R1 = 3;          // Bit pattern %0000 0000 0000 0011
[P0] = R1;
SSYNC;

While the core timer counter register != 0 continue

P0.H = hi(TCOUNT)
P0.L = lo(TCOUNT)  // Get the address into P0
SSYNC            // Necessary or not?

TIMER_LOOP:
R0 = [P0];       // Keep reading the timer counter
CC = R0 == 0;    // TIMER COUNTER IS VOLATILE
IF !CC JUMP TIMER_LOOP

DE-ACTIVATE TIMER
 VALUES STORED IN R2
RETURN

REACTIVATE INTERRUPTS – Old
void UseUpFixedAmountTime (unsigned long int timeToUse) - 3

While the core timer counter register ! = 0 continue
    P0.H = hi(TCOUNT)
    P0.L = lo(TCOUNT)   // Get the address into P0
    SSYNC              // Necessary or not?

TIMER_LOOP:
    R0 = [P0];         // Read the timer counter
    CC = R0 == 0;
    IF !CC JUMP TIMER_LOOP

DE-ACTIVATE TIMER       // You provide the required code

REACTIVATE INTERRUPTS – Old values stored in R2

POST-LAB 1 QUIZ REVIEW -- USE OF CC LOOP CONTROL

RETURN
Problem -- CODE IS “WAITING” and uses up processor power uselessly

While the core timer counter register ! = 0 continue
P0.H = hi(TCOUNT)
P0.L = lo(TCOUNT)  // Get the address into P0
SSYNC                    // Necessary or not?

TIMER_LOOP:
R0 = [P0];               // Read the timer counter
CC = R0 == 0;
IF !CC JUMP TIMER_LOOP

Here the processor is waiting, which means that the processor can’t be calculating other values (computer graphics) or servicing other requests
Fixed by using interrupts to do the counting in the background

Perhaps there are no other values to calculate. In that case we need to put the processor in a low power mode, and then wake it up
Fixed with IDLE instruction and then wake-up enable register
Concepts of interrupt code

NOT PART OF POST-LAB 1 QUIZ

Task 1 – file 1

volatile int foo_flag = 8;

int main( ) {
    SetUpTimerInterrupts(timeBeforeISRHappens);
    StartTimerInterrupts( );

    while (foo_flag != 0) {
        WriteLEDASM(foo_flag);
        DoSomething Complicated( );
    }
    StopTimerInterrupts();
}

Task 2 – file 2 (C++ or ASM)

extern volatile int foo_flag;

SPECIAL C++ CODE NEEDED TO
Tell “C++” that I am not a function or subroutine that is called within the program. I am an ISR – interrupt service routine and proud of it. I can happen at any time because of an outside external signal

????declare??? ISR_count( ) {
    foo_flag--;

    Tell the timer that the interrupt has been serviced (handled)
How it is supposed to work

First task 1 (main( )) – starts

Sets up the timer so that it will cause an interrupt
When this interrupt starts – task 1 will stop, task 2 will start

start doing some processing – spend NO time looking at the timer

NOTE – if the ISR – interrupt service routine – does not happen (because external hardware not working) then task 1 will never stop

When Task 1 stops – turn off the interrupts
Task 2 and Task 1 in this code

- Task 1 and Task 2 communicate via the “volatile foo_flag” variable – “message”

- Every time the timer counts down to zero
  - The TCOUNTER register is reloaded with TPERIOD register value
  - The timer counts down again
  - An interrupt is issued and latched
Unanswered questions

1. What does “volatile” mean?
2. Why will “optimized code” probably not work if volatile is not used?
3. How do you tell C++ that this function is an ISR and not a standard function?
4. Why do you need to tell C++ that this function is an ISR and not a standard function?
5. What is the difference (in coding) between an ISR and a standard function?
6. How does an interrupt get latched, why and where?
7. Why do I have to tell the timer that the interrupt has been serviced, and how do I do it?

Task 2 – file 2 (C++ or ASM)

extern volatile int foo_flag;

Tell “C++” that I am not a function but I am an ISR – interrupt service routine

????declare as ISR??? ISR_count( )
{
    foo_flag--;
    Tell the timer that the interrupt has been serviced
}
Tackled today

- Watchdog timer – Hardware Reference 15-49
- Core timer – Hardware Reference 15-45
- General purpose timers 15-1
  - Pulse Width Modulation
  - Pulse Width Count and Capture
  - External Event
- Application of timers to provide code safety and improved version of `UseFixedAmountTimeASM()`
- Introduction to timer interrupts
Information taken from Analog Devices On-line Manuals with permission
http://www.analog.com/processors/resources/technicalLibrary/manuals/

Information furnished by Analog Devices is believed to be accurate and reliable. However, Analog Devices assumes no responsibility for its use or for any infringement of any patent other rights of any third party which may result from its use. No license is granted by implication or otherwise under any patent or patent right of Analog Devices. Copyright © Analog Devices, Inc. All rights reserved.