GETTING STARTED WITH THE TMS320F24x PROCESSOR

Tutorial 3: Generating a Time Delay

New Instructions Introduced

LACC
SUB

New Flag Introduced

SXM

New Test Conditions Introduced

EQ NEQ

Introduction

In the previous tutorial we saw how to turn the XF LED on and off under the control of software, as well as how to control its brightness. In this tutorial we shall flash the XF LED on and off at different rates.

Generating a Time Delay

In order to flash the XF LED slowly, one way is to generate a time delay. Example 3-1 shows one way in which C code can be used to do this.

Example 3-1.

```
                unsigned long x;        /* Declare variable x */
                x = 0x7FFF;             /* Initialize x to 7FFFh */
                while ( x != 0)         /* Loop until x is equal to 0 */
                {
                  x--;                  /* Decrement variable x */
                }
```

Note that we have chosen to count down from an initial value of 7FFF hexadecimal to zero, rather than to count up. This is deliberate. When using assembly language, testing for zero is much easier than testing for a specific number.

Let us now implement Example 3-1 using assembly language. There are three separate operations that need to be performed: assign storage for variable *x*, decrement *x* and test for *x* is equal to zero.

Allocating Storage for the Variable

The first task is to allocate storage for the variable *x* in a place where we can manipulate and test it. For this we shall use the 32-bit working register known as the *accumulator*.

To load the accumulator with the value 7FFF hexadecimal we write:

Example 3-2.

```
        LACC #7FFFh     ; Load the accumulator with
                        ; 7FFFh. The accumulator now
                        ; contains 00007FFFh. This is
                        ; equivalent to x = 0x7FFF.
```

The instruction `LACC` (load accumulator) is used with a single operand. Note that the operand `7FFFh` is preceded by the symbol #. The symbol # means that value following immediately afterwards is to be taken as a *number*.

A common mistake to make is to forget to type in the symbol #, which changes the meaning of the code.

The number `7FFFh` has used the letter 'h' to mean hexadecimal (or base 16).

Example 3-3.

```
        LACC #200h      ; Correct syntax to load the
                        ; accumulator with the constant
                        ; 200h.
        LACC 200h       ; Incorrect syntax. Does not load
                        ; the accumulator with the
                        ; constant 200h.
```

Decrementing the Variable

We have now initialized our variable *x,* which we have loaded into the accumulator. The next step is to decrement the contents of the accumulator. The TMS320F243 does not provide a decrement instruction. Instead we must subtract one from the accumulator.

Example 3-4.

	SUB #1	; Subtract the value 1 from the ; accumulator. This is equivalent ; to *x*--;

The instruction SUB (subtract from accumulator) takes a single operand, which is the value to be subtracted. As in Example 3-2, the symbol # must precede the number. In Example 3-4, the value 1 is subtracted from the accumulator and the result put back into the accumulator.

Testing the Value for Zero

The third task is to test the variable *x* for zero. The variable *x* is stored in the accumulator and this can be tested using the instruction BCND (branch conditionally).

Example 3-5.

	.setsect ".text",	8800h
loop:	BCND *done*, EQ	; If the condition that the ; accumulator is equal to zero is ; TRUE, then branch to the code at ; the label *done*.
	B *loop*	; The accumulator does not contain ; zero. The test condition that ; the accumulator contains zero ; has evaluated to FALSE. Go round ; again.
done:	NOP	; This instruction will be ; executed when the accumulator ; contains zero.

The instruction BCND (branch conditionally) takes two operands. The first operand is a *label*. The second operand is the *condition* to be tested. When the *condition* evaluates to TRUE, then a branch occurs to the code at the label.

In this case we use the test condition EQ. When the contents of the accumulator are equal to zero, then the test condition EQ evaluates to TRUE and a branch occurs. On the other hand, when the contents of the accumulator are not zero, then the test condition EQ evaluates to FALSE and no branch occurs.

The Complete Delay Loop

The instructions LACC (load accumulator), SUB (subtract from accumulator) and BCND (branch conditionally) can be combined together to form a delay loop, as shown in Example 3-6.

Example 3-6.

```
                .setsect ".text", 8800h

                LACC #7FFFh     ; Load the accumulator with the
                                ; constant 7FFFh. The accumulator
                                ; now contains 00007FFFh.
loop:           BCND done, EQ   ; Test if the accumulator is zero.
                                ; If TRUE, then branch to the code
                                ; at the label done. If FALSE, then
                                ; execute the next instruction.
                SUB #1          ; Subtract 1 from the contents of
                                ; the accumulator.
                B loop          ; Go round again
done:           NOP             ; This instruction will be executed
                                ; when the value in the accumulator
                                ; has reached zero.
```

In Example 3-6, the instruction NOP (no operation) is the next instruction to be executed when the loop terminates.

Tracing the Code Execution

How exactly does the code in Example 3-6 work?

If we were to step through the instructions one at a time, we would see the order in which the instructions are executed.

Example 3-7.

```
                LACC #7FFFh     ; Accumulator contains 00007FFFh.
loop:           BCND done, EQ   ; Accumulator does not contain 0.
                                ; Do not branch. Execute next
                                ; instruction.
                SUB #1          ; Accumulator contains 00007FFEh.
                B loop          ; Go to code at label loop.
loop:           BCND done, EQ   ; Accumulator does not contain 0.
                                ; Do not branch. Execute next
                                ; instruction.
                SUB #1          ; Accumulator contains 00007FFDh.
                B loop          ; Repeat loop until accumulator
                                ; contains zero.
                                ; More instructions
```

Finally, when we have been round the loop 7FFFh (32767) times, the value in the accumulator will be zero and the loop will terminate.

A More Code-Efficient Delay

We can write the `while` loop in Example 3-1 as a `do-while` loop, as shown in Example 3-8.

Example 3-8.

```
        unsigned long x;        /* Declare variable x */
        x = 0x7FFF;             /* Initialize x to 7FFFh */
        do {
          x--;                  /* Decrement variable x */
        } while (x != 0);       /* Loop until x is equal to 0 */
```

This is similar to the `while` loop, but instead of testing the value of *x* at the beginning of the loop, we test it at the end. The `do-while` loop requires less instructions to implement than does the `while` loop.

Example 3-9.

```
        .setsect ".text", 8800h

        LACC #7FFFh     ; Load the accumulator with the
                        ; constant 7FFFh. The accumulator
                        ; now contains 00007FFFh.
loop:   SUB #1          ; Subtract 1 from contents of the
                        ; accumulator.
        BCND loop, NEQ  ; Test if the accumulator is zero.
                        ; If FALSE, then execute the code
                        ; at the label loop. Otherwise
                        ; execute the next instruction.
        NOP             ; This instruction will be executed
                        ; when the value in the accumulator
                        ; has reached zero.
```

There are two differences between Example 3-6 and Example 3-9. The `do-while` construction in Example 3-9 does not use the instruction `B` (branch unconditionally), which saves an instruction.

Also, the test condition has changed from branch when accumulator equals zero (`EQ`) to branch when the accumulator is non-zero (`NEQ`). The condition `NEQ` evaluates to TRUE when the contents of the accumulator are not zero. When the accumulator contains zero then the condition `NEQ` evaluates to FALSE.

Calculating the Delay Time

How long a delay does the code in Example 3-9 generate? Let us first write out this example again, but this time putting in the numbers of cycles per instruction.

Example 3-10.

```
                .setsect ".text", 8800h

                LACC #7FFFh         ; 2 cycles.
loop:           SUB #1              ; 1 cycle.
                BCND loop, NEQ      ; 4 cycles when TRUE. 2 cycles when
                                    ; FALSE.
```

The instruction `BCND` will execute 7FFFh -1 = 7FFEh (32766) times. Each time the instruction `BCND` is executed it will take 4 cycles, except the last time when it does not branch and will take 2 cycles.

On the TMS320F243 DSK, each cycle takes 0.4µs (400 ns).

The execution time will be (2 + 32766 * 5 + 1 + 2) * 0.4µs = 65535 µs = 0.0655 seconds.

Out of Range Value

The code in Example 3-10, which implements a `do-while` loop will work correctly when any value is loaded into the accumulator between 1 and 7FFFh. However, if we were to load the accumulator with zero, then:

Example 3-11.

```
                .setsect ".text", 8800h

                LACC #0h            ; Load accumulator with 0
loop:           SUB #1              ; Accumulator now contains
                                    ; FFFFFFFFh
                BCND loop, NEQ      ; 4 cycles when TRUE. 2 cycles when
                                    ; FALSE.
```

After the first instruction `SUB` (subtract from accumulator), the accumulator will contain FFFFFFFFh. Consequently the loop will execute FFFFFFFFh times, giving a very long delay indeed!

Care should therefore be taken using the `do-while` construct not to use an initial value of zero. This is not a problem with a `while` loop because the test is done before the decrement, so zero will cause immediate termination.

Generating a Longer Delay

In order to generate a longer delay that shown in Example 3-9, we require a larger starting value than 7FFFh in the accumulator.

Loading a value between 8000h and FFFFh into the accumulator is not as easy as it might first seem. If we were to load the value FFFFh into the accumulator using the instruction `LACC`, then we can obtain one of two different outcomes, depending upon the value of the `SXM` (sign-extension) flag.

Example 3-12.

```
                .setsect ".text", 8800h

                CLRC SXM        ; Turn off sign-extension mode.
                LACC #0FFFFh    ; The accumulator now contains
                                ; 0000FFFFh
                SETC SXM        ; Turn on sign-extension mode.
                LACC #0FFFFh    ; The accumulator now contains
                                ; FFFFFFFFh.
```

Why is there a difference in behavior between the two applications of the instruction `LACC`?

When sign-extension mode is turned off, all numbers are taken to be *unsigned* i.e. positive only. This means that FFFFh is taken to be 65535 decimal.

On the other hand, when sign-extension mode is turned on, the value loaded into the accumulator is taken to be a *signed* number. This means that FFFFh is taken to be –1 decimal. When expressed as a 32-bit number, -1 decimal is FFFFFFFFh.

Note that positive values of 7FFF or less are not affected by sign-extension mode.

Flashing the XF LED on and off

Example 3-13 shows how we can use a time delay to flash the XF LED on and off.

Example 3-13.

```
        .setsect ".text", 8800h

start:  SETC SXM        ; Turn on sign-extension mode.
        SETC XF         ; Turn on XF LED.
        LACC #7FFFh     ; Load initial value of 7FFFh into
                        ; the accumulator. The accumulator
                        ; now contains 00007FFFh.
loop1:  SUB #1          ; Decrement the accumulator.
        BCND loop1, NEQ ; Test the accumulator. If the
                        ; accumulator does not contain
                        ; zero (the condition NEQ
                        ; evaluates to TRUE) then branch
                        ; to the label loop1.
        CLRC XF         ; Turn off XF LED.
        LACC #7FFFh     ; Load initial value of 7FFFh into
                        ; the accumulator. The accumulator
                        ; now contains 00007FFFh.
loop2:  SUB #1          ; Decrement the accumulator.
        BCND loop2, NEQ ; Test the accumulator. If the
                        ; accumulator does not contain
                        ; zero (the condition NEQ
                        ; evaluates to TRUE) then
                        ; branch to the label loop2.
        B start         ; Go round again from beginning.
```

The XF LED will be on for 0.0655 seconds and off for 0.0655 seconds. This corresponds to a frequency of 7.6 Hz.

Flashing the XF LED more Slowly

To flash the XF LED more slowly than in Example 3-13, we need to load a value larger than 7FFFh into the accumulator at the beginning of each delay loop. This requires that sign-extension mode be turned off.

Example 3-14.

```
        .setsect ".text", 8800h

start:  CLRC SXM            ; Turn off sign-extension mode.
        SETC XF             ; Turn on XF LED.
        LACC #0FFFFh        ; Load initial value of FFFFh into
                            ; the accumulator. The accumulator
                            ; now contains 0000FFFFh.
loop1:  SUB #1              ; Decrement the accumulator.
        BCND loop1, NEQ     ; Test the accumulator. If the
                            ; accumulator does not contain
                            ; zero (the condition NEQ
                            ; evaluates to TRUE) then branch
                            ; to loop1.
        CLRC XF             ; Turn off XF LED.
        LACC #0FFFFh        ; Load initial value of FFFFh into
                            ; the accumulator. The accumulator
                            ; now contains 0000FFFFh.
loop2:  SUB #1              ; Decrement the accumulator.
        BCND loop2, NEQ     ; Test the accumulator. If the
                            ; accumulator does not contain
                            ; zero (the condition NEQ
                            ; evaluates to TRUE) then branch
                            ; to loop2.
        B start             ; Go round again from beginning.
```

Note that the labels *loop1* and *loop2* are different.

Each loop will take 1 + 1 +1 + (5 * 65534) + 2 = 327675 cycles = 0.131 seconds. This corresponds to a frequency of 3.8 Hz.

TMS320F243 DSK EXPERIMENTS

Equipment Required

TMS320F243 DSK
Frequency meter or oscilloscope

Experiment 3-1.

Objective: To Flash the XF LED on and off.

Enter the code in Example 3-13 into a text editor, save it, then assemble it and run it on the TMS320F243 DSK. This program turns on the XF LED, waits until the delay is done, turns off the XF LED and then waits until the second delay is finished. The sequence then repeats.

Measure the frequency between the output on the XF pin (Connector P2 pin 21) and ground (Connector P2 pin 19 or pin 20). Alternately, a probe may be connected between one end of the XF LED and the other pin to ground.

The XF LED should be on for 0.0655 seconds and off for 0.0655 second, corresponding to a frequency of 7.6 Hz.

Experiment 3-2.

Objective: To Change the Delay Time

Again using Example 3-13 as the starting point, replace the references to 7FFFh with 0FFFFh. Save the program, assemble it and run it on the debugger. Measure the frequency on the XF pin (Connector P2 pin 21) and ground (Connector P2 pin 19 or pin 20).

Experiment 3-3.

Objective: To show the effect of Sign-Extension Mode

Enter the code in Example 3-14 into a text editor, save it then assemble and run it on the TMS320F243 DSK. Measure the output frequency on the XF LED.

This program is similar to that in Example 3-13, only the reference to `SETC SXM` has been replaced with `CLRC SXM`.

Experiment 3-4.

Objective: To Flash the XF LED on and off 5 times per second

Modify Example 3-13 to flash the XF LED on and off 5 times per second (0.1 second time delay to give 0.1 seconds on and 0.1 seconds off). Hint. It will be necessary to turn off sign-extension mode.

Design Problem 3-1.

Objective: To Compare the `do-while` loop with the `while` loop

Enter Example 3-6 into a text editor. Replace the comments with the number of cycles. Calculate the time required to execute the delay loop.

How does the delay time differ between the `while` loop in Example 3-6 and the do-`while` loop in Example 3-10?

Design Problem 3-2.

One of the comments in Example 3-15 is incorrect. What should it be?

loop:	SUB #1	; Subtract 1 from contents of the ; accumulator.
	BCND *loop*, NEQ	; Test if the accumulator is zero. ; If TRUE, then execute the code ; at the label *loop*. Otherwise ; execute the next instruction.

Self-Test Questions..... *Click Here To View* Answers!

1.	What is the *accumulator*?
2.	The instruction `LACC` means which of the following? a) Left align counter b) Load accumulator c) Load and clear counter d) Load auxiliary carry
3.	Which of the following instructions loads the accumulator with a constant? a) `LAC #3` b) `LACC 35h` c) `LACC #140h` d) `LAK 43h`
4.	The instruction `SUB #1` carries out which operation? a) Subtract 1 from the accumulator b) The accumulator is subtracted from 1
5.	The instruction `BCND label, EQ` will branch when the condition `EQ` is TRUE. Here `EQ` means equal to what?
6.	In which of the following cases will the condition `EQ` evaluate to TRUE? a) The accumulator contains a negative value b) The accumulator contains a positive value c) The accumulator contains zero.
7.	Why might we implement a `do-while` construction in preference to a `while` loop?
8.	In which of the following cases will the condition `NEQ` evaluate to TRUE? a) The accumulator contains a negative value b) The accumulator contains a positive value c) The accumulator contains zero.
9.	The `while` loop has an advantage over the `do-while` construction. What is it?
10.	What will the accumulator contain after the following two instructions are executed?
	`CLRC SXM ; Turn off sign-extension mode` `LACC #0FFFFh`
11.	What will the accumulator contain after the following two instructions are executed?
	`SETC SXM ; Turn on sign-extension mode` `LACC #0FFFFh`
12.	What is the maximum positive value that can be loaded into the accumulator using the instruction `LACC`, which is not effected by `SXM`?

References

TMS320F/C24x DSP Controllers. CPU and Instruction Set. Reference number SPRU160.

TMS320F/C240 DSP Controllers. Peripheral Library and Specific Devices. Reference Number SPRU161.

TMS320F243, TMS320F241 DSP Controllers. Reference Number SPRS064.

CLICK TO VIEW
Tutorials
1 2 3 4 5 6 7 8 9 10

Click here to view..... Route Map