Using the Real-Time-Clock Library

Lane Westlund *MSP430 Applications*

ABSTRACT

This document will serve as an overview describing the use of the RTC library within an existing C or assembly project. The RTC library encapsulates commonly used routines for keeping track of time. These functions are written in assembly to be optimized for the MSP430, but can be called from any C program that includes the 'RTC.h' header file. To be easily displayable on an LCD all variables are encoded in BCD. All variables, except the year, use one byte. The year variable uses two bytes.

Introduction

It is often the case that along side its main purpose, an MSP430 application will also keep track of and display the current time. The purpose of this library is to encapsulate the features commonly associated with keeping track of time, and making it easy to display on an LCD screen. This encapsulation means the user only needs to make simple function calls with well defined parameters, and not worry about all the computational steps needed to ensure correct time keeping.

The RTC library includes all files necessary to setup a periodic interrupt, and keep real-time. It includes functions to handle the periodic-time interrupt and increment all time-keeping variables. Time-keeping variables include seconds, minutes, hours, days of the week, months, years, and even leap-years.

The RTC is structured to use one-second time intervals by calling the incrementSeconds function. Inside this function the variable TI_second is updated. When it increments to a count of 60, it will be set to zero and the TI_minute variable will be incremented. When TI_minute reaches 60 it is zeroed and the TI_hour variable is incremented, and so on. All clock and calendar variables update automatically with the one function call to incrementSeconds.

There are two files included in the library for time-keeping: RTC.s43 and RTC_Calendar.s43. The first is for use only when time is desired but not calendar functions like day-of-the-week, month, etc. The second implements the full calendar including leap-year adjustments. To implement an RTC in an application, one of these files, but not both is simply included in the project and the appropriate function is called at one-second intervals.

The library also includes functions to setup the required one-second interval. Several timers can be used to generate a one-second interrupt, including Timer_A, the Watchdog timer, and the Basic Timer. Functions to setup each for a one-second interrupt are included in the files named RTC_TA.s43, RTC_WDT.s43, and RTC_BT.s43.

Also included in this application report are a series of code examples. The purpose of these code examples is to illustrate the setup needed to use the library in various ways.

Usage from C

A typical clock type application that uses the RTC library would look like this:

```
#include  "RTC.h"
void main ( void )
{
  WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer
  setTime( 0x12, 0, 0, 0);               // initialize time to 12:00:00 AM
  P1DIR |= 0x01;                         // Set P1.0 to output direction
  CCR0 = 32768-1;
  TACTL = TASSEL_1+MC_1;                 // ACLK, upmode
  CCTL0 |= CCIE;                         // enable CCRO interrupt
  _EINT();

  while( 1 )
  {
    LPM3;                                // enter LPM3, clock will be updated
    P1OUT ^= 0x01;                       // do any other needed items in loop
    _NOP();                              // set breakpoint here to see 1 second int.
  }
}
// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  incrementSeconds();
  LPM3_EXIT;
}
```

The file ‘RTC.h’ must be included in order to gain access to both the variables and functions when using the RTC library from a C program.

In the example above, the writer chose to implement their own Timer_A Interrupt Service Routine (ISR), and set up Timer_A for a 1 second interrupt. If they did not want to write their own ISR and were using a 32.768kHZ crystal, they could have used the RTC_TA library:

```
#include  "RTC.h"
#include  "RTC_TA.h"

void main ( void )
{
  WDTCTL = WDTPW + WDTHOLD;              // Stop watchdog timer
  setTime( 0x12, 0, 0, 0);               // initialize time to 12:00:00 AM
  TA_1sec_wake();                        // configure TA for 1 second update
  LPM3;                                  // enter LPM3, clock will be updated
}
```

By calling the function ‘TA_1sec_wake()’, Timer_A is set up to run in continuous mode with 32768 in CCR1. For this reason a 32.768kHz crystal must be supplied on LFXT1. The ISR supplied in RTC_TA simply calls the function ‘incrementSeconds’ and returns. After exiting the ISR, the device will return to its previous state. This means it will return back into LPM3 or any other low-power mode it was previously in. To return from the ISR into active mode (in order to complete a loop) the User must write a custom ISR as shown earlier.

In addition to 'incrementSeconds', the library also supports several other 'increment' functions such as 'incrementMinutes', 'incrementHours', 'incrementDays', and 'incrementYears'. These functions are provided so that they may be called in situations where the time is to be set through user interaction (ie. Pushing a button)

Usage from Assembly

The library can also be used from assembly. In this case, no imports are required, but the desired functions must be listed using the **EXTERN** keyword. Care must also be taken to make sure function calls are in immediate mode.

```
#include  "msp430x14x.h"
            EXTERN  TA_1sec_wake
;-------------------------------------------------------------------------------
            ORG     01100h                  ; Progam Start
;-------------------------------------------------------------------------------
RESET       mov.w   #0A00h,SP               ; Initialize 'x1x9 stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
            call    #TA_1sec_wake
            bis.w   #CPUOFF,SR              ; CPU off, interrupts enabled
                                            ;
```

Leap Years

When using the calendar functions, leap years will be correctly calculated except for all years that are evenly divisible by 400. As of this writing, the next year that will meet this criterion is 2400 A.D. which was deemed as an acceptable trade-off in order to reduce code size.

Leap years will be calculated any time an 'increment' or 'setDate' function is called. Should the year variable be changed outside of the RTC_Calendar library, the function 'testLeap' should be called to set the correct number of days in February for that year.

For convenience, the macro LEAP_YEAR has been included in RTC_Calendar.h. After the current year's leap year calculation has been done, C programs can put in the LEAP_YEAR conditional to test whether the current year is a leap year or not. LEAP_YEAR is defined as (FebDays == 0x29)

Daylight Savings

Both EU and US daylight savings is accounted for. By setting the 'dayLightZone' variable to either US_DAYLIGHT_SAVINGS, EU_DAYLIGHT_SAVINGS, or NO_DAYLIGHT_SAVINGS the library will automatically adjust the clock on the following times:

	Begins (hour +1)	Ends (hour -1)
US	First Sunday in April at 2am	Last Sunday in October at 2am
EU	Last Sunday in March at 1am	Last Sunday in October at 1am

Note: The variable TI_dayLightSavings is meant to be used internally for the library. It is designed to prevent the library from being stuck in an endless loop, rolling back the hour each time 2 or 1 am is reached in the last Sunday in October. It is set when daylight savings begins, and cleared the first time 2 or 1 am is reached on the last Sunday in October. **It is always set to 1 when the setDate() routine is called, regardless of the date given**. If the code is calling setDate() to a static date, it is possible to set this variable to a static value in the code, which will be updated correctly

Cycle Count:

	Cycle Count	
Function	RTC	RTC_Calendar
incrementSeconds	14	14
incrementMinutes	14	14
incrementHours	22	35[1]
get24Hour	23	23
incrementDays	NA	37[1]
incrementMonths	NA	14[1]
incrementYears	NA	36[1]
setDate	NA	681[2]

Code Size:

	Size (bytes)
RTC	126
RTC_Calendar	714

Note:

The default state of the variables is 12:00:00 AM (Tuesday, January 1st 2004 when the Calendar library is used). In IAR, these variables will be initially set to this default state when any global C variables are used. If this is not the case, these variables must be explicitly initialized in the application code using the setTime(h,m,s,pm) and setDate(y,m,d) functions.

[1] When incrementing from 1:00PM April 29th 2005. This represents the most common cycle count. Daylight savings days and incrementing into a leap year will require more cycles.
[2] When setting date to April 29th 2005

Example Includes:

The following is a list of hypothetical example projects, which describe which code files should be included for each one.

RTC application with Calendar (non-library based 1 second interrupt):

- RTC_Calendar.s43
- RTC_Calendar.h

RTC application with Calendar using Timer_A from Library

- RTC_Calendar.s43
- RTC_Calendar.h
- RTC_TA.s43
- RTC_TA.h

RTC application without Calendar (non-library based 1 second interrupt):

- RTC.s43
- RTC.h

RTC application without Calendar using Timer_A from Library

- RTC.s43
- RTC.h
- RTC_TA.s43
- RTC_TA.h

Included Files:

Note: In the zip file accompanying this application report there are two directories: source_CCE and source_IAR. The files in these directories are functionally equivalent, and only contain minor changes to allow for compiling using CCE or IAR respectively.

RTC.s43

This file includes all variables and functions needed for a “time only” RTC. Measurements beyond hours of the day are not accounted for. If greater functionality is desired, RTC_Calendar should be included and not this file

RTC_Calendar.s43

This file includes all variables and functions needed for a full RTC. Calendar based functions such as day, month, year, day of week, and daylight savings are included.

RTC_BT.s43

This file includes the function BT_1sec_wake(). Calling this function will initialize the Basic Timer to wake every 1 second and call the function incrementSeconds(). Please note: an external 32Khz crystal on LFXT1 is assumed.

RTC_TA.s43

This file includes the function TA_1sec_wake(). Calling this function will initialize Timer_A to wake up every one second and call the function incrementSeconds(). CCR1 is used and loaded with 32768. The timer runs in continuous mode so each during each interrupt an additional 32768 is added to CCR1. It is possible to use CCR0 for additional interrupt timing, however the TAR should not be modified and the mode should not be changed, in order to avoid mis-timing. Please note: and external 32Khz crystal on LFXt1 is assumed.

RTC_WDT.s43

This file includes the function WDT_1sec_wake(). Calling this function will initialize the Watchdog Timer to wake every 1 second and call the function incrementSeconds(). Please note: an external 32Khz crystal on LFXT1 is assumed.

Test_Suite.c

This file is included to illustrate all tests used on the RTC library. In this file, all use cases for the Library are tested

Variable Description

Note: all RTC library variables begin with the 'TI_' prefix in order to avoid collision with any other variable names used in an end application

TI-second

The current second count: 0x00-0x59

TI_minute

The current minute count: 0x00-0x59

TI_hour

The current hour count: 0x00-0x12

TI_day

The current day of the month: 0x01-0x31

TI_dayOfWeek

The current day of the week: 0-6 (integer), Sunday == 0. Please see headers for defined day values

TI_month

The current month of the year 0x00-0x11, January == 0. Please see headers for define month values

TI_year

The current year 0x0000-0x2399. Leapyears are not computed properly for 0x2400 (see earlier note)

TI_PM

The AM/PM flag. AM = 0 PM = 1

TI_FebDays

The number of days in this current year's February. Either 0x28 or 0x29

TI_DayLightZone

The current daylight savings time method being used. Can be either "no daylight savings", "US daylight savings", or "EU daylight savings" Please use header file definitions

TI_DayLightSavings

Used to test whether the daylight savings hour has already been rolled back in order to avoid an infinite loop. Please see the earlier note for more information on this variable and its usage.

Function Description

incrementSeconds(void)

This function will add one to the BCD variable 'second'. If 'second' increases to 0x60, incrementMinutes() will be called and 'second' will revert to 0x00

incrementMinutes(void)

This function will add one to the BCD variable 'minute'. If 'minute increases to 0x60, incrementHours() will be called and 'minute will revert to 0x00

incrementHours(void)

This function will add one to the BCD variable ‘hour’. If ‘hour’ increases to 0x12, the variable ‘PM’ will be toggled. If ‘hour’ increases to 0x13 it will revert to 1. If ‘hour’ increases to 0x13 and PM is 0x01 incrementDays() will be called

setTime(char hour, char minute, char second, char pm)

This function sets the current time to the values specified in the values passed in. This function is strictly a macro function so the parameters must be passed in BCD format using 12 hour notation.

Get24Hour()

This function returns a char containing the current hour in BCD 24 hour format. For example 12:00 AM returns as 0x00 and 12:00 PM returns as 0x12, 11:00 PM returns as 0x23

incrementDays(void) – *Calendar Only*

This function will add one to the BCD variable ‘day’. If ‘day’ is greater than the number of days in the current month, it will revert back to 0x01 and incrementMonths() will be called

incrementMonths(void) – *Calendar Only*

This function will add one to the BCD variable ‘month’. If ‘month’ increases to 0x12, incrementYears() will be called and ‘month’ will revert to 0x00. **Note:** this function does not correctly set the ‘dayOfWeek’ variable when called from a source externally to the library

incrementYears(void) – *Calendar Only*

This function will add one to the BCD variable ‘year’. **Note:** this function does not correctly set the ‘dayOfWeek’ variable when called from a source externally to the library

testLeap(void) – *Calendar Only*

This function will correctly set the ‘FebDays’ variable based on the current ‘year’ variable. It computes whether February has 28 or 29 days in the current year. This function is typically only called by setDate(), but could be called after anytime the TI_year variable is changed.

setDate(int year, char month, char day) – *Calendar Only*

This function sets the current date to the values specified in the variables year, month, and day. It will compute whether the year is a leap year and correctly set the ‘FebDays’ variable. It will also correctly set the ‘dayOfWeek’ variable. The variables should be passed in decimal (ie. Not BCD) format. Months should be passed with January == 1 and day starting at 1 as well.