```
/*
 / _____)             _              | |
( (____  _____ ____ _| |_ _____  ____| |__
 \____ \| ___ |    (_   _) ___ |/ ___)  _ \
 _____) ) ____| | | || |_| ____( (___| | | |
(______/|_____)_|_|_| \__)_____)\____)_| |_|
    (C)2013 Semtech

Description: MCU RTC timer and low power modes management

License: Revised BSD License, see LICENSE.TXT file include in the project

Maintainer: Miguel Luis and Gregory Cristian

Note: works for OTAA - SV

*/
#include <math.h>
#include "board.h"
#include "rtc-board.h"
#include "em_rtcc.h"
#include "em_rtc.h"
#include "em_cmu.h"
#include "em_emu.h"

#define RTC_TICKS_PER_S             32768
#define RTC_TICKS_PER_MS            32.768
#define RTC_TICK_DURATION_MS  0.030517578


#define RTC_DIVIDER                   ( cmuClkDiv_1 )
#define RTC_CLOCK                     ( 32768U )
#define MS_TO_TICKS_DIVIDER           ( 1000U * RTC_DIVIDER )
#define MS_TO_TICKS_ROUNDING_FACTOR   ( MS_TO_TICKS_DIVIDER / 2 )
#define MS_TO_TICKS( ms )     ( ( ( (uint64_t)(ms) * RTC_CLOCK ) +
MS_TO_TICKS_ROUNDING_FACTOR ) / MS_TO_TICKS_DIVIDER )
#define TICKS_TO_MS_ROUNDING_FACTOR   ( RTC_CLOCK / 2 )
#define TICKS_TO_MS( ticks ) ( ( ( (uint64_t)(ticks) * RTC_DIVIDER * 1000U )
+ TICKS_TO_MS_ROUNDING_FACTOR ) / RTC_CLOCK )


// EFM32 temperature sensor gradient
#define THERMOMETER_GRADIENT  (-6.3)

// RTC variables. Used for converting RTC counter to system time
static uint32_t   rtcOverflowCounter    = 0;
static uint32_t   rtcOverflowInterval   = 0;
static uint32_t   rtcOverflowIntervalR  = 0;


static TimerTime_t   previousTime = 0;

/*!
 * \brief Indicates if the RTC is already Initialized or not
 */
static bool RtcInitalized = false;
```

```
/*!
 * \brief Hold the Wake-up time duration in ms
 */
volatile uint32_t McuWakeUpTime = 0;

/*!
 * \brief Hold the cumulative error in microsecond to compensate the timing
errors
 */
static int32_t TimeoutValueError = 0;


void RtcInit( void )
{
      if( RtcInitalized == false )
      {
            // Reset overflow counter
            rtcOverflowCounter = 0;

            // Calculate overflow interval based on RTC counter width (32
bit) and frequency
            rtcOverflowInterval   = ((0xFFFFFFFF+1) / RTC_TICKS_PER_MS);
            rtcOverflowIntervalR  = ((0xFFFFFFFF+1) - (rtcOverflowInterval *
RTC_TICKS_PER_MS)); // remainder

            RtcInitalized = true;

            RTCC_Init_TypeDef rtccInit = RTCC_INIT_DEFAULT;
            RTCC_CCChConf_TypeDef rtccInitCompareChannel =
RTCC_CH_INIT_COMPARE_DEFAULT;

            rtccInit.cntWrapOnCCV1 = false;   //not reset counter
            rtccInit.presc = rtccCntPresc_1;

            /* Setting the compare value of the RTCC */
            RTCC_ChannelInit(1, &rtccInitCompareChannel);
            RTCC_ChannelCCVSet(1, 0xFFFFFFFF);

            /* Enabling Interrupt from RTCC */
            RTCC_IntEnable(RTCC_IEN_CC1 | RTCC_IEN_OF );
            NVIC_SetPriority(RTCC_IRQn, 4);
            NVIC_ClearPendingIRQ(RTCC_IRQn);
            NVIC_EnableIRQ(RTCC_IRQn);

            /* Initialize the RTCC */
            RTCC_Init(&rtccInit);

      }
}


void RtcSetTimeout( uint32_t timeout )
{
      RTCC_IntClear(RTCC_IFC_CC1);

      previousTime = RtcGetTimerValue();
```

```
	if (timeout < 1)
		timeout = 1;

	// timeoutValue is used for complete computation
	double timeoutValue = round( timeout * RTC_TICKS_PER_MS );

	// timeoutValueTemp is used to compensate the cumulating errors in
timing far in the future
	double timeoutValueTemp =  ( double )timeout * RTC_TICKS_PER_MS;

	// Compute timeoutValue error
	double error = timeoutValue - timeoutValueTemp;

	// Add new error value to the cumulated value in uS
	TimeoutValueError += ( error * 1000 );

	// Correct cumulated error if greater than ( RTC_ALARM_TICK_DURATION *
1000 )
	if( TimeoutValueError >= ( int32_t )( RTC_TICK_DURATION_MS * 1000 ) )
	{
		TimeoutValueError = TimeoutValueError - ( uint32_t )(
RTC_TICK_DURATION_MS * 1000 );
		timeoutValue = timeoutValue + 1;
	}

	// Rounding errors should not cause us to set the number behind the
current time
	uint32_t value = (previousTime * RTC_TICKS_PER_MS) + timeoutValue;

	if (RTCC->CNT > value && value + RTC_TICKS_PER_MS > RTCC->CNT)
		RTCC_ChannelCCVSet(1, RTCC->CNT + 1);
	else
		RTCC_ChannelCCVSet(1, value);

	RTCC_IntEnable(RTCC_IEN_CC1);
}

TimerTime_t RtcGetAdjustedTimeoutValue( uint32_t timeout )
{
    return  timeout;
}


TimerTime_t RtcGetTimerValue( void )
{
    return TICKS_TO_MS(RtcGetTimerTickValue());
}

uint32_t  RtcGetTimerTickValue(void )
{
	//the RTCC is a 32bit timer
	return  RTCC_CounterGet();
}



TimerTime_t RtcGetElapsedAlarmTime( void )
```

```
{
    TimerTime_t currentTime = RtcGetTimerValue();
    if( currentTime < previousTime )
    {
        return( currentTime + ( rtcOverflowInterval - previousTime ) );
    }
    else
    {
        return( currentTime - previousTime );
    }
}

TimerTime_t RtcComputeFutureEventTime( TimerTime_t futureEventInTime )
{
    return( RtcGetTimerValue( ) + futureEventInTime );
}

TimerTime_t RtcComputeElapsedTime( TimerTime_t eventInTime )
{
    TimerTime_t elapsedTime = 0;

    // Needed at boot, cannot compute with 0 or elapsed time will be equal to
current time
    if( eventInTime == 0 )
    {
        return 0;
    }

    elapsedTime = RtcGetTimerValue();
    if( elapsedTime < eventInTime )
    { // roll over of the counter
        return( elapsedTime + ( rtcOverflowInterval - eventInTime ) );
    }
    else
    {
        return( elapsedTime - eventInTime );
    }
}


void RtcEnterLowPowerStopMode( void )
{
      EMU_EnterEM2(false); //you don't need to restore clocks
}


void RTCC_IRQHandler(void)
{
      uint32_t flags = RTCC_IntGetEnabled(); //get flags for only the enabled
interrupts
      RTCC_IntClear(RTCC_IFC_OF | RTCC_IFC_CC1);

      // Check for overflow
      if (flags & RTCC_IF_OF)
      {
            rtcOverflowCounter++;
```

```
	}
	// Check if timer expired
	if (flags & RTCC_IF_CC1)
	{
		RTCC_IntDisable(RTCC_IEN_CC1);
		TimerIrqHandler();
	}
}


void HAL_Delay (uint32_t ms)
{
	uint32_t startTime = RtcGetTimerValue();
	while((RtcGetTimerValue() - startTime) < ms); // busy wait until at
least ms ticks have passed
}
```