STM32 gotchas

82. On the RTC readout lock mechanism (and its deficiencies)

The RTC in STM32 (except the 'F1 family) maintains time in three separate registers - date in RTC_DR, time in RTC_TR and subseconds (i.e. the current state of synchronous prescaler) in RTC_SSR. These registers are in the RTC's clock domain, which is typically asynchronous to clock of APB bus through which processor reads these registers, so reading them involves internal synchronization, which implies 2 APB-cycle wait states when reading them[1].

Other problem with separate timekeeping registers is, that reading them one by one may result in inconsistent set of data (as an incrementing pulse may arrive out of the asynchronous prescaler during that reading sequence).

In the STM32 RTC, both problems are addressed using a set of shadow registers in the APB clock domain (i.e. reading them does not impose the waitstates), into which the timekeeping registers' content is automatically copied by hardware every 2 RTC clocks[2]. Reading RTC_SSR, RTC_TR and RTC_DR then reads these shadow registers.

To provide consistent set of timekeeping registers, this copy process is stopped (shadow registers are "locked") as soon as RTC_SSR or RTC_TR is read; and the copy process is resumed (shadow registers are "unlocked") when RTC_DR is read. But this description is not entirely true: instead of stopping the copy process altogether, it is stopped selectively for individual registers - when RTC_SSR is read, RTC_TR and RTC_DR are locked (but RTC_SSR is never locked itself); when RTC_TR is read, only RTC_DR is locked. Or, as the RM puts it:

To ensure consistency between the 3 values, reading either RTC_SSR or RTC_TR locks the values in the higher-order calendar shadow registers until RTC_DR is read.

This behaviour can be easily checked in debugger (such that enables reading individual registers, without reading other RTC registers, e.g. gdb): reading RTC_SSR always returns a changing value. After RTC_SSR was read, reading RTC_TR returns the same value until RTC_DR is read, After that, repeated reading RTC_TR will return a changing value. After reading RTC_SSR or RTC_TR, and waiting long enough, first read of RTC_DR will return the old value which has been locked at the reading of RTC_SSR or RTC_TR, subsequent read will return changing (current) value [3].

Unfortunately, this locking mechanism is faulty, as witnessed by the erratum *RTC calendar registers are not locked properly* (for parts introduced before cca 2016). According to this erratum:

When reading the calendar registers with BYPSHAD = 0, the RTC_TR and RTC_DR registers may not be locked after reading the RTC_SSR register. This happens if the read operation is initiated one APB clock period before the shadow registers are updated.

As workaround[4], erratum recommends to read RTC_SSR second time after reading out all three registers, and repeat if the two RTC_SSR reads don't match. This is basically the same procedure as it would be for BYPSHAD=1 (i.e. as if there would be no shadow registers). The basic idea is, that if the two RTC_SSR reads return identical value, there was no decrement[5] to it, thus RTC_DR and RTC_TR could not have changed either. This of course assumes the two RTC_SSR reads are not more than one second (taking into account RTC_SSR decrement period) apart, i.e. there must be no lengthy interrupt in between, but that's a reasonable requirement for normal microcontroller applications.

However, the erratum does not mention that it is **necessary to read RTC_DR after the second RTC_SSR read**, otherwise RTC_TR and RTC_DR would remain locked, and upon the next read of the timekeeping registers, old values would be read from these registers.

The following snippet for STM32F091 tests the erratum's assertion, that RTC_TR may remain not locked after reading RTC_SSR. After RTC_SSR read, RTC_TR is read, then follows a delay long enough to ensure the "true" RTC_TR to be incremented, and then RTC_TR is read again to find out, whether it has changed due to failure of the lock:

```
#include "stm32f0xx.h"

volatile uint32_t ssr, tr, dr;
volatile uint32_t ssr2, tr2, dr2;

#define  RCC_BDCR_RTCSEL__NONE              0   // cannot be changed to __NONE from other except through backup-domain-reset
#define  RCC_BDCR_RTCSEL__LSE               1   // external 32.768kHz crystal
#define  RCC_BDCR_RTCSEL__LSI               2   // internal low-speed RC oscillator
#define  RCC_BDCR_RTCSEL__HSE_32            3   // high-speed crystal, divided by 32

#define  AND &
#define  OR |


void LoopDelay(volatile uint32_t n) {  // results in delay of cca n*8 clocks
        while(n > 0) n--;
}

int main(void) {
  RCC->APBENR1 |= 0
    | RCC_APBENR1_PWREN    // enable clock to PWR, as we will want to write to its registers
    // some mcus such as the 'G0 require also to enable RTCAPB clock here
  ;


  // init clock - we use default HSI 8MHz system clock,
  // but we also enable HSE (where his board has a 8MHz Xtal) to be used by RTC as HSE/32
  RCC->CR = (RCC->CR AND ~(0
```

```
    // this field by default is zero so we don't bother
  )) OR (0
    OR RCC_CR_HSEON
  );

  PWR->CR |= PWR_CR_DBP;  // enable backup domain access

  RCC->BDCR = 0
    OR (RCC_BDCR_RTCSEL__HSE_32 * RCC_BDCR_RTCSEL_0)    // select the HSE/32, here it results in 250kHz input clock to RTC
    OR (1 * RCC_BDCR_RTCEN)     // enable RTC
  ;

  // unlock RTC write protection after power-on reset
  RTC->WPR = 0xCA;
  RTC->WPR = 0x53;

  // set to init mode and wait until in sync
  RTC->ISR |= RTC_ISR_INIT;
  while ((RTC->ISR AND RTC_ISR_INITF) == 0);

  // set the prescaler
  // we set it to an unusually low value so that we don't need to wait too long to see date to change
  // this results in "seconds" (i.e. RTC_TR) counting at cca 7.8kHz instead of 1Hz,
  // that means one "hour" lasts cca half a second, thus RTC_DR increments cca once in 12 seconds
  RTC->PRER = (uint32_t)0x7;  // sync
  RTC->PRER |= (uint32_t)(0x3 SHL 16);  // async -- RTC_PRER register must be written in two steps according to the datasheet!

  // now set the time/date -- this is meaningless here, but let's just do it
  RTC->TR = 0x000000;  // time: 00:00:00 (BCD HHMMSS)
  RTC->DR = 0x130705 + (5 SHL 13);  // date: 2012 09 12 Wednesday (BCD YYxMDD + DOW SHL 13)

  // exit init mode - this should start the RTC
  RTC->ISR &= (uint32_t)~RTC_ISR_INIT;

  // wait for synchro
  // -- it should take 2 RTCCLK clocks - given similar HSI and HSE frequency and no APB prescaler, this is cca 64 APB clocks
  RTC->ISR &= (uint32_t)~RTC_ISR_RSF;
  while ((RTC->ISR AND RTC_ISR_RSF) == 0);


  while(1) {

    ssr = RTC->SSR;  // lock
    tr = RTC->TR;

    // LoopDelay(300) => roughly 8*300 system clocks = 300us @ 8Mhz HSI, TR ticks at cca 7.8kHz i.e. 128us period (see above),
    // so this delay makes sure that "real" TR will change
    LoopDelay(300);

    if (tr != RTC->TR) {  // would the shadow TR be not locked, it would be changed by now
      __BKPT();
    }

    dr = RTC->DR;  // unlock

    // wait for next copy to shadow registers
    RTC->ISR = ~(RTC_ISR_RSF OR RTC_ISR_INIT);
    while((RTC->ISR AND RTC_ISR_RSF) == 0);
    // add some "random" delay so that the "read starts one APB cycle before copy" condition for the erratum to kick in is met
    // HSI/HSE non-synchronicity also helps in this
    LoopDelay(tr % 100 + 50);
  }
}
```

A "random" wait (together with mutually asynchronous system/RTC clock sources) should ensure that the "one APB clock period before the shadow registers are updated" condition of erratum is fulfilled. One iteration lasts roughly 500us, and assuming enough "randomness" the probability of fulfilling above condition is given by the ratio of shadow registers update (8MHz/32/2) and APB clock (8MHz) i.e. 1:64, so we should see the program to end up in the breakpoint pretty soon.

However, the program did not stop at the breakpoint even when run for several minutes. So what gives?

I believe that the description in the erratum is not precise. In the above program, the main loop has been changed for a simple:

```
  while(1) {
      ssr = RTC->SSR;  // lock
      tr = RTC->TR;
      dr = RTC->DR;  // unlock
      ssr2 = RTC->SSR;  // lock
      tr2 = RTC->TR;
      dr2 = RTC->DR;  // unlock
      if (tr2 != tr) {
        __BKPT();
      }
  }
```

and when breakpoint was reached (i.e. RTC_TR changed), ssr2 was observed. It was found, that while in most cases ssr2 = 7 - which is the expected value, equal to the synchronous prescaler, given RTC_TR has just changed and the program is simple enough to avoid any delays to be introduced - sometimes ssr2 = 0 was seen.

This means, that under the specific timing condition, content of RTC_SSR was read before decrement (and rollover), while RTC_TR was locked after RTC_SSR decrement => RTC_TR increment, in the updated state.

The same program (only modified for register name changes) was run also on a STM32G0B1, with breakpoint subject to ssr2 != 7, and there was no occurence of this during several minutes of run, confirming that the RTC in 'G0 has this particular problem fixed.

Regardless of the exact mechanism of the lock problem, the workaround described in the erratum is valid [EDIT] - although see g95 for a proper implementation [/EDIT].

---

1. Accessing all RTC registers except the three shadow registers impose the same 2 APB-cycle wait states, see *RTC register access* subchapter in RTC chapter of RM. In other words, all RTC registers are located at the backup clock domain. In some applications, this waitstate can have unwanted consequences in blocking the APB bus with this access, potentially resulting in late handling (and thus over/underflow) of other peripherals on the same APB bus.

2. For this copy process to work properly, APB bus frequency has to be 7x the RTC clock frequency. There is a method to monitor this copying process: the RTC_ISR.RSF bit is set by hardware when the copy is performed, so this bit can be cleared by software and then polled. There is also a method to avoid using the shadow registers altogether, by setting RTC_CR.BYPSHAD.

3. As RTC_DR changes only "once a day", this is normally cumbersome to observe. For experiments, RTC can be clocked from a high-speed oscillator (LSE in bypass, or using the HSE/32 option) and the prescaler can be set to a minimal value. The experimental setup described in this article results in RTC_DR being incremented roughly once in 12 seconds.

4. By the end of this discussion (which inspired this article), user with nick Piranha proposes an alternative method for consistent reading of the timekeeping registers, ~~which should work consistently, regardless of the erratum or BYPSHAD setting.~~ [EDIT] Unfortunately, this method does not work because of issues discussed in g95. [/EDIT]

5. RTC_SSR is downcounter.

Created: 11.December 2021
Link to g95 and other small fixes: 27.February 2021