STM32 gotchas

This is a collection of more or less unexpected or surprising behaviour of STM32 microcontrollers, whether documented or not. What constitutes a gotcha is subjective, and some of the items here are not gotchas at all, just poiting out a feature or idiosyncracy, which may be surprising for some. Also, some of the items are not STM32-specific at all, but apply more broadly to ARM Cortex-M-based mcus, or are related to C or programming generally.

This collection will grow, check back or subscribe to the RSS if you want to follow. Oh yes, no social media, learn how to use real information sources.

Many items are a result of discussion at STM32 forum at community.st.com. I'll try to give relevant links to forum threads where applicable, but they have changed in the past and may do so in the future too, and I won't be able to go and fix all of them. ST assisted me to migrate most of the links lost during migration of forum in 2023.

There's no particular ordering or sorting, it would be hard to find out something half-sane, so just browse through and enjoy.

NEW! All gotchas up to g156 now also in a single pdf.

Legalese: STMicroelectronics registered STM32 as a trademark in various countries, although I don't exactly know what are the ramifications of this, I am not a lawyer. I am not affiliated with ST. In what I write here I may be completely wrong, use below information at your own risk.

Comments are welcome, email them to stm32 at efton dot sk.

---

1. Peripheral clock must be enabled in RCC (else registers cannot be written and read as 0)
2. Timers' clock is 2x APB clock, if APB divider > 1
3. Advanced timers (TIM1, TIM8, ...), to enable output, need to set TIMx_BDTR.MOE
4. Debugging is intrusive (e.g. UART/SPI Rx may not work when debugging)
5. 'F4/'F2 CCM RAM is not good for DMA nor for bit-banding, nor for code
6. RTC seemingly does not run if only time is read
7. Interrupt called without reason (late interrupt source clear)
8. Timer does not work if ARR=0
9. `TIMx->SR &= ~TIM_SR_flag` results in lost interrupts (don't RMW or bit-band on status registers)
10. Strange behaviour after increasing system clock frequency (FLASH latency has to follow system clock)
11. DAC output does not go rail-to-rail and the output buffer makes it worse
12. UART parity bit missing as it counts up to data bits
13. Delay is 1ms longer than required (SW/library, not STM32-specific)
14. GPIO is not toggling at a rate promised by datasheet
15. Timer appears to run slower than is set (unrealistically high interrupt rate)
16. Interrupt does not fire - some troubleshooting hints
17. Program freezes after enabling an interrupt
18. ADC readings not as expected because of high signal impedance
19. Debugger keeps jumping into interrupt when single stepping on 'F746
20. NDTR problematic when DMA used as Rx FIFO
21. SPI master NSS unusable
22. Writing one byte to SPI transmits two bytes (because of *data packing*)
23. STM32 hangs in UART interrupt if Rx overrun is not handled
24. UART Rx stops working (Rx overrun revisited)
25. STM32 are not microcontrollers, but SoC causing various timing-related issues
26. Don't use `printf()` in interrupt or any other lengthy operation
27. In 'F4, after enabling backup domain access in PWR, wait before accessing RCC_BDCR or BKPSRAM
28. SPI master does not work as NSS switches it to slave
29. Processor (and debugger) hangs while debugging OTG USB Host
30. Only DMA2 can be used to transfer to/from GPIO in 'F4 and 'F2/'F7
31. Timer - if compare is enabled, setting CCRx > ARR still causes interrupts and DMA and TRGO
32. SPI generates too many clocks while Rx in Bidir more, and in Rx-only mode
33. Smaller packages require port remapping for certain functions namely the USB pins PA11 PA12 on 'F042 in TSSOP20
34. ADC in 'L47x/48x has an extra switch in GPIOx_ASCR

35. 'F1 don't have the GPIO AF matrix unlike all other STM32 families
36. Peripherals don't have unique AF number in GPIO matrix they do in 'L1 and mostly do in 'F2/'F4/'F7
37. GPIO pin input state can still be read, if it's set to Out or AF and EXTI works, too
38. Analog switch voltage booster (for ADC and other) to be switched on at lower VDD on certain STM32
39. Software or watchdog reset does not work if NRST is pulled-up hard externally
40. First TIM period is too short, prescaler does not work - PSC is preloaded; and so is RCR
41. One TIM period is too long/one PWM pulse is shorter/longer than expected - use preload when needed
42. First TIM Update ("PeriodElapsed") interrupt fires immediately after configuration when using Cube/SPL
43. No bit-banding in Cortex-M0/M0+ but not in Cortex-M7 either
44. Some STM32 models after first programming don't run and need power-on reset (AN2606 pattern6)
45. Not only frequency but also VOS setting implies FLASH waitstates
46. Some pins on my Disco/Nucleo/EVAL board don't work as expected
47. TIMx_CHxN is not inverted if TIMx_CHx is not enabled
48. TIMx_CHxN cannot be used as input to timer
49. F3 CCM RAM can't be used for DMA but it can be used for code
50. In 'F4 CRC, data are ignored if written too soon after reset
51. Not all TIM_CHx are created equal
52. TIM - in Encoder mode, prescaler must be 0 (otherwise the counted value is nonsense)
53. TIM does not run if ARR was changed from 0 to nonzero due to ARR preload
54. In 'F1, RDP also sets write protection on first sectors
55. 'F2/'F4/'F7/'H7 unreliable at higher frequencies due to incorrect VCAP capacitor
56. Internal temperature sensor on ADC indicates surprisingly high temperature but it's most probably correct
57. 'H7 lifetime may be surprisingly short at elevated working temperatures
58. Changing clock frequency requires careful readjustment of peripherals
59. PF4 on 'F303 does not work because it's missing on 'F303RD/E
60. TIM DMA burst (DMAR/DCR) does not work properly if TIMx_DMAR observed in debugger
61. In 'F42x/43x, erase sector does not work due to confusing sector numbering
62. Backup domain is not reset by setting RCC_BDCR.BDRST unless PWR_CR.DBP is set, too
63. Compatibility within the Value Line (0-ending) models
64. Standard C library function does not work not STM32-specific
65. Standard C library function (e.g. printf()) fails in interrupt or RTOS not STM32-specific
66. SPI master Tx outputs 16 clocks, even if its Data Size set to 8 bit keyword: data packing
67. Where are GPIO AF numbers given for individual pins? And where are the NVIC registers described?
68. SPI_SR.BSY is unusable
69. In 'F746/756, ADC triggering from TIM does not work properly but for some cases this can be fixed by switching on DAC
70. Pins don't work even if set in GPIO due to nonzero default values in 'L0/'L4/'Gx/'H7 GPIOx_MODER
71. RTC does not work because of RCC_APBxENR.RTCAPBEN bit in some models
72. No DMA to/from GPIO in 'L0/'G0 i.e. where the Cortex-M0+ core is used
73. Always read the Errata and not only those directly given for your STM32 model
74. Write Protection (WRP) granularity may be non-uniform in large-FLASH 'F1/'F0/'F3
75. Unexpected ETH interrupts after longer uptime due to MAC management counters interrupts being enabled by default
76. RTC digital calibration does not apply to wakeup period with a specific wakeup source selection
77. ETH does not work with certain APB2 clock settings due to internal delays between modules
78. Using TIM input channel filter may produce confusing results if fed with periodic signal
79. If TIM is set to downcounter, 0% PWM is not possible
80. When processor stopped in debugger, peripherals and DMA if used by those peripherals keep running
81. Crash due to stack content getting corrupted DMA writes to buffer improperly allocated on stack
82. On the RTC readout lock mechanism and its deficiencies
83. Cannot access BKPSRAM in CubeProgrammer unless register values are manipulated
84. GPIO pin does not output enough current if set to low GPIO_OSPEEDR setting
85. Not all GPIO pins are created equal e.g. PA10/PB12 pullup/down in 'F2/'F4/'F7 is only 10kΩ
86. 'G0/'G4 USB-C Dead Battery conundrum pin influenced by other pin, unexpected pulldowns
87. UART Tx is not driven immediately after enabling UART and UART_BRR has to be set properly
88. The Y2k38 bug not STM32-specific
89. How to set TRGO in TIM16/TIM17? Also, TIM channel needs output enabled if controls another module, e.g. ADC.
90. Some time after peripheral (TIM, SPI, etc.) function finished, pin state changes resulting in unwanted clock/transition
91. Some interrupts cannot be disabled, and attempts to do so may lead to HardFault e.g. SysTick
92. Peripheral connected to other than a single pin in GPIO AF matrix #GPIO_is_poorly_documented
93. Timers don't support DIR/STEP encoders except in 'G4/'U5 - but a trick may exist
94. 'G4 lifetime may be surprisingly short at elevated working temperatures
95. How to achieve consistent RTC readout?
96. TCM don't support highest clock frequencies in 'H723/725
97. 5V-tolerant pins don't always tolerate 5V especially if mcu is powered down
98. FLASH is not erased to 0xFF in 'L0 and 'L1 but to 0x00
99. FLASH is 0xFF but cannot be written if there's ECC and granule has been already written

100. In lower-end 'F0xx, GPIOC/GPIOD don't have AFR[2] registers
101. V_{DDA} is pulled high when ADC is enabled if V_{DDA}<2.4V and booster not enabled
102. STLink-V3 8MHz MCO output is not derived from crystal partial workaround available
103. After being cleared in NVIC outside an ISR, interrupt reappears if pending flag in NVIC is cleared too soon after removing the interrupt source
104. USB packet buffer memory in 'F042 is not bytewise writable
105. Clicked ETH/USB in CubeIDE/CubeMX, and it does not work
106. Bit-banding cannot be used with DMA
107. No atomic access through bit-banding to GPIO registers' bits except in 'F1
108. Toggling GPIO in 'H7 is slow due to complex bus structure
109. PF0/PF1/PC14/PC15 don't work GPIO overriden by oscillator enabled in RCC
110. Using interrupt or DMA, only first few data are transmitted correctly due to local variable as buffer - not ST-specific
111. Some interrupts invoke incorrect ISR if VTOR is not properly aligned
112. After setting GPIOA things stop to work due to overwriting SWD pins settings
113. Used this Cube/SPL function and it works weird uninitialized struct as parameter
114. Some TIM register fields are not continuous in newer STM32 families
115. Incorrect values from internal temperature sensor due to errorneous values in 'G4 datasheet
116. RTC loses half a second per powerup/reset Cube bug, fixed
117. Current consumption in sleep modes unexpectedly high during debugging due to debugging in sleep enabled by DBGMCU_CR bits
118. DWT_CYCCNT does not work in some sleep modes
119. ETH IPv4 checksum offload does not work if checksum fields in the packet submitted to ETH/MAC are not zero
120. Unique ID (UID) is not available in some STM32 families
121. 96-bit Unique ID cannot be shortened and still considered unique
122. Unique ID is not at continuous addresses in memory in some STM32 families
123. INITS flag may not be the best indicator of RTC having been already initialized if date has not been set
124. Calibration values (UniqueID, temperature, internal VREF) are not at the same address not even within the same family
125. USB/DFU bootloader in 'F4/'F7 may start unreliably with higher frequency crystals
126. Not all STM32F730 are created equal and some don't have the built-in HS USB PHY
127. Built-in USB HS PHY in 'F723/'F730 requires certain crystal frequencies
128. In 'F2, subseconds don't work in RTC nor smooth calibration (and same is for 'L1 Cat.1)
129. In 'F1, PA15/PB3/PB4 don't work they are assigned to JTAG by default
130. In 'F1, PA15/PB3/PB4 stop working after remapping unrelated peripherals AFIO_MAPR.SWJ_CFG bits are write-only
131. Undefined AF in GPIO may turn pin to output
132. DMA vs. DMAMUX numbering discrepancy
133. PA2 unexpectedly outputs a 32kHz signal? RTC behaves strangely? Maybe consequence of VBAT brownout
134. Internal VREFBUF reference not available in lower-pin-count packages where VREF+ is bonded to VDDA
135. Using the internal VREFBUF reference requires a specific value capacitor connected to VREF+ pin
136. Cannot reprogram FLASH after inadvertent write access to FLASH area, and if Cube/HAL is used
137. Communication errors in prototype on breadboard/with flying leads - on importance of signal integrity and proper ground/return (not STM32-specific)
138. Built-in bootloader works unreliably or not at all if there is activity on other bootloader-related pins
139. BOOT0 pin pulled up, but bootloader does not start in 'F1/'L1/'F2/'F4 you need to pull down BOOT1, too
140. Not all TIM2/TIM5 are created equal in some STM32 they are not 32-bit
141. On UART IDLE frame insertion upon UART_CR1.TE being set
142. Not all TIMx_RCR are created equal some are 8-bit, some 16-bit
143. V_{REFINT} and temperature calibration values not available in 'F1 and 'F2
144. Timer trigger interrupt/DMA does not work until TIMx_SMCR.SMS is set to nonzero
145. Not all TIM input polarity selection created equal in non-XL 'F1 there's no both-edges selection
146. Only one EXTI per pin number, this means routing which is different in 'F1 and 'G0/'C0/'L5/'U5
147. Slow input signal slew rate may cause multiple interrupts especially with EXTI, which is asynchronous
148. Touchscreen flipped or does not work on 32F429IDISCOVERY
149. Incorrect ADC readout due to missing calibration
150. Incorrect ADC readout in some families due to the "wrong result after 1ms delay" erratum
151. Missing ADC codes in 'H7 due to missing calibration
152. Internal temperature sensor returned unexpected value so here's some troubleshooting guide
153. Internal temperature sensor, VREFINT and VBAT can't be part of a fast ADC sequence in families where ADC has a common sampling-length register
154. In 'L07x, some TIM3_CHx pins don't work unless remapped in TIM3_OR
155. After programming/reset/power cycle, program does not work and debugging ends up at 0x1FFFxxxx addresses
156. USB and PG2..PG15 don't work on some STM32 they have a dedicated power pin and an isolation switch

157. In 'U5, ADC calibration hangs if VDDA isolation is not removed
158. STM32F407 runs at cca 20x lower frequency than set a RCC/HSE/PLL puzzle
159. PLLI2S/PLLSAI outputs unexpected frequency due to prescaler shared amongst PLLs
160. LSI in 'U5 does not work unless Backup Domain access is enabled before enabling LSI
161. No connectivity from ETH and OTG-HS to FLASH in 'F407 and 'F446 so ETH/USB won't transmit const data
162. LPUART Rx clocked from LSE corrupts bytes at 9600 Baud unless LPUART_CR3.UCESM is set
163. UART Rx corruption if glich occurs during 2nd half of stopbit
164. Strange characters received in terminal in PC parity or baudrate mismatch, or RS232 converter was used for UART
165. UART/USART conundrum
166. FMC/FSMC conundrum
167. 'H7 longevity revisited
168. Inconsistent numbering considered harmful especially mixing 0-based and 1-based numbering
169. Inconsistent naming considered harmful especially between RM and CMSIS-mandated device headers
170. Programming 'G0/'C0 from built in bootloader has to be well-planned as default bootloader entry depends on empty FLASH
171. 'G0/'G4 bricked after programming RDP1 if BOOT_LOCK is set
172. UART receives zeros? Using pullup with UART_Rx may be a good idea not STM32-specific
173. Some remarks on UART used in halfduplex mode
174. Anecdotes about missing VDDx
175. Data register reads zero in 'H7 SPI/I2S if read width is narrower than data width set for SPI/I2S
176. SPI/I2S set to PCM runs twice the expected sample rate PCM is MONO, I2S is STEREO
177. I2SxExt allows not only fullduplex SPI, but also dual-Tx/dual-Rx
178. I2SxExt_SD and SPIx_MISO are different AF in 'F4, but the same AF in 'F3
179. I2S clock numbering is confusing in 'F446 and 'F413
180. In TIM, TRGI signal is generated only if TIMx_SMCR.SMS is nonzero
181. On the Commutation event (COM) in some timers
182. TIM channel can't be enabled in TIMx_CCER if TIMx_CR2.CCPC is set by mistake
183. After enabling clock in RCC, delay may be needed before accessing some peripherals
184. DMAMUX clock how to enable
185. What's all this kernel clock stuff, anyhow?
186. Some kernel clocks are not selected/enabled in RCC
187. Where's that clock enable? It's tied to some other peripheral.
188. ... and then there's also no clock enable at all...
189. ... or the clock is enabled by default...
190. ... or the clock enable bit is just named confusingly
191. Datecode in chip marking is not unique as year is single digit
192. What's up with these -P and -Q suffixed STM32? They have different pinout than the non-suffixed ones.
193. My STM32 board does not work. Are all these pins properly connected?
194. PLL does not output expected frequency or does not work at all physical ratio not identical to value in registers
195. PLL output does not work if its enable bit is not set
196. Not all RTC are created equal
197. Problems with day rollover in 'F1 RTC if SPL/Cube are used
198. RTC at midnight does not roll over, but goes to 24:00:00 and counts up
199. RTC clock source can be set only once
200. LSI is not a good clock to keep real time in RTC
201. STM32 with SMPS+LDO may be hard to recover, if inadvertently set to SMPS upon startup

Last change: 23.March 2024

(C)2020-2024 EFTON s.r.o. - All rights reserved.

STM32 gotchas

1. Peripheral clock must be enabled in RCC

This is a common feature for all 32-bitters - to control consumption, clocks to most peripherals are gated and have to be enabled individually.

In STM32, this happens in the RCC module (RCC does not have a clock gate so is available all the time). The vast majority of modules' clock is disabled after reset, see the individual RCC_AHBxENR/RCC_APBxENR registers' description in the respective Reference Manual (RM).

Using a module without enabling its clock, is a quite common mistake when programming STM32. However, it can be easily detected when debugging: if a module's clock is not enabled, all of its registers read as zero[1], [2], and they can't be written.

See also gotcha 183: After enabling clock in RCC, delay may be needed before accessing some peripherals, gotcha 184: How to enable DMAMUX clock, gotcha 185: What's kernel clock ?, gotcha 186: On kernel clocks, which are not selected/enabled in RCC, gotcha 187: Some clock enables are tied to some other peripheral, gotcha 188: Some peripherals don't have clock enable at all, gotcha 189: Some peripherals have clock enabled by default, gotcha 190: Some peripherals have clock enable bit named confusingly.

---

1. Not necessarily zero, although apparently that is the most usual case.
STM32 forum user MMust.5 noticed, that when reading GPIOA and GPIOB registers after reset in STM32F407, before enabling GPIOA/GPIOB clock in RCC, all GPIOA registers return 0xA800'0000 and all GPIOB registers return 0x0000'0280. These are the reset values of the GPIOA_MODER and GPIOB_MODER registers, respectively.
Experimentation on STM32L476 revealed, that after enabling GPIOA clock, manipulating then reading out any GPIOA register, and disabling GPIOA clock, subsequently reading any GPIOA register returns the value of the register which was last read, but not that read value, but its value at the moment when GPIOA clock was disabled (this was tested on GPIOA->IDR: it was read once while clock enabled, then externally one of the GPIOA was pulled to the opposite polarity, then clock was disabled, and the value which is then returned for all GPIO registers reflects the grounded pin, not the value last read from IDR). So, the APB read interface of GPIOA probably consists of an input latch, which latches the read address, and an output latch, which upon each APB clock (gated by the enable bit in RCC) latches content of the addressed internal register.
This all is an entirely useless information, though, as there is no practical use of it, and also this arrangement may be entirely different for other peripherals or for GPIOx in other STM32 families. In fact, users are not supposed to read/write to disabled peripherals. Some families' RM (e.g. for 'G0) use the following warning: *When the peripheral clock is not active, the peripheral registers read or write accesses are not supported.* The 'H7 RM has the following: *When the bus interface clock is not active, read or write accesses to the peripheral registers are not supported. A read access will return invalid data. A write access will be ignored and will not create any bus errors.*

2. Some peripherals appear to have an even more complex access scheme than described in footnote 1. Here's an example:
In STM32F427, PWR has two registers, CR and CSR. After the first powerup, they read 0xC000 and 0x0000 respectively (the set bits are PWR_CR.VOS being set to Scale 1 which is the reset default). After enabling PWR clock in RCC, this reading does not change, but this step allows us to set PWR_CR.DBP (enabling access to backup domain) and also set PWR_CSR.BRE (which is enable of the low-power LDO of the Backup SRAM, which maintains its content while in VBAT mode). After this, PWR registers read as 0xC100 and 0x0208, the extra bit which reads set is bit 3 in PWR_CSR, PWR_CSR.BRR, which is the bit indicating that the low-power LDO works.
Now if we perform a "pushbutton" reset without a power cycle (i.e. we reset PWR clock in RCC, but we retain the backup-domain bits setting - namely PWR_CSR.BRE here), PWR registers read as 0xC100 and 0x0200, i.e. PWR_CSR.BRE can be read, but PWR_CSR.BRR not. After enabling PWR clock in RCC, PWR registers read as 0xC000 and 0x0208.
Corollary is, that reading a peripheral which does not have clock enabled in RCC, does not result in zeroes, but in possibly surprising values.

Created: 24.May 2020
Footnote 1. added: 29.May 2022
Footnote 2. added: 1.Oct 2022
Link to g183 added: 19.August 2023

2.Timers' clock is 2x APB clock, if APB divider > 1

Most peripherals in STM32 are attached to what's called an Advanced Peripheral Bus (APB), and work at whatever clock this bus works. There may be one or several (usually two) such buses in an STM32, and their clock is derived from the AHB bus to which it is attached (and clock of that is ultimately derived from the system clock).

As there may be many peripherals "hanging" out of one such APB bus, which means that the bus is physically large/long, thus its maximum clock frequency is limited and possibly below the maximum system clock frequency. Where two buses are present, often there is a "shorter" and "faster" one, and a "longer" and "slower" one.

This means, that for higher-speed STM32 models (such as the 'F4/'F7/'H7/'G4), when system clock is run at maximum, the APB clock may be required to be divided down, by /2 or /4 or more. But also in other models, APB bus clock may be divided down, e.g. to reduce power consumption, if fast access to peripherals and fast clocking of the peripherals is not a requirement.

Most peripherals use the clock of APB bus, to which they are attached, as their working clock. However, with timers, this is different in probably all STM32 models - if the related APB clock is divided from AHB clock by /2 or more, the timer's core clock, i.e. the clock which is used as "internal clock" to the timer's prescaler/counter/input filters/deadtime generator etc., is 2x as fast as the APB clock (the register interface is still clocked at the APB clock, but that's rarely noticed by the user and in the vast majority of cases negligible).

The consequence of this is an often seen complaint, that the timer runs twice faster than expected.

In some STM32 models - e.g. the 'F446 - there's one more twist: while by default the behaviour is the same as described above, the user can switch on an extra feature (in said 'F446 by setting RCC_DKCFGR.TIMPRE), which allows to use 4x APB clock, should its divider be /4 and larger (and still 2x APB if the divider is /2).

Created: 30.May 2020

3.Advanced timers (TIM1, TIM8, ...) need to have TIMx_BDTR.MOE set to enable output

The timers in STM32 are relatively complex. They come in several flavours, although the basic design is always the same: a single counter with a prescaler, a "slave mode controller" which is mainly a steering unit for clocks and other signals such as reset, one to four (or more in some models) capture-compare (CC) units which interface to external pins, and interface to interrupts and DMA and other signals interconnecting from and to other modules within the mcu.

This is the "normal" "model"; then there are a few stripped-down timers which don't have any CC units nor connectivity to external pins or options for external clocks, serving mostly as timebase; and then there are the "advanced" timers, which have extra features.

ST in the various STM32 models mostly sticks to a the same scheme in numbering timers, i.e. Advanced timers are numbered TIM1, TIM8, TIM20 where available. While ST does not call TIM15, TIM16 and TIM17 as Advanced timers, and while they have only one or two CC channels, they share many of the characteristics of Advanced timers, and the gotcha described here applies to them, too.

As the vast majority of STM32 timers, the Advanced timers are 16-bit, i.e. they have 16-bit counter and the CC registers (CCRx) are 16-bit too. In STM32 models where there is a "slow" and "fast" APB bus, Advanced timers tend to be located on the "fast" bus.

The purpose of the Advanced timers is mainly in providing PWM or PWM-like output for power converters and motor controllers, usually driving one or several half-bridges. This is why Advanced timers have an additional unit on the first three CC channels, providing complementary output, i.e. each CC channel can control two complementary outputs (onto pins TIMx_CHy and TIMx_CHyN).

To be able to "kill" all complementary channels of one timer simultaneously, there is a common enable bit, MOE (Master Output Enable) in the TIMx_BDTR register (together with a couple of other features common to all complementary channels of the timer, e.g. deadtime). This bit has to be set in software, to enable the complementary channels' output towards the pins[1]; it can be cleared in software, but its main purpose is, that it can be cleared by hardware (a.k.a. BREAK), upon various signals, depending on the particular STM32 model.

Often, users, who migrate code from the non-advanced timers, are not aware of the TIMx_BDTR.MOE bit thus don't set it, leading to a frustrating "PWM does not work" problem.

1.Advanced timers typically have 4..6 channels, of which only 3 are complementary. The remaining non-complementary channels are not affected by TIMx_BDTR.MOE, so they control output pin (if set as Output Compare, and enabled by their respective TIMx_CCER.CCxE bit) even if TIMx_BDTR.MOE = 0.

Created: 6.Jun 2020
Footnote added: 17.Feb 2024

4.Debugging is intrusive

Using the on-chip debugging facilities (together with a debugging "pod" such as the ST-Link, and associated software) is a great tool for debugging through breakpoints, single-stepping and observing variables in memories. It also provides direct experience with the hardware, through observing and directly manipulating the individual modules' registers from "inside" the mcu.

However, this is not without consequences. The debugging module has no magical access to the individual modules; it is part of the processor itself, and uses the processor's buses to access whatever resource is attached to the processor, including memories and peripheral modules. One of the consequences of this is, that if there's some hardware which is influenced by being accessed (mainly its registers being read), it will behave differently than if it's not under debugger access.

And yes, there's quite a couple of modules which can be impacted by this. The most common related complaint is, that when debugging, UART or SPI Rx loses incoming bytes (frames). This is quite easy to understand, as these modules usually have a status bit (in the status word), indicating, that the Rx data register is not empty (RxNE). Once the debugger reads out the data register, this status bit gets cleared - thus the program (i.e. processor) - and it may happen that DMA too - will subsequently "see" a cleared RxNE thus won't proceed reading and processing the incoming data.

Most debuggers won't access registers, which are not currently displayed; and also most debuggers won't access registers if not stopped (on a breakpoint or single-stepping). However, there may be exceptions. Debugger behaviour is toolchain-specific, so for the details, read your toolchain's documentation.

Related:

- RTC lock mechanism (BYPSHAD=0) may result in old/stopped RTC registers reading
- Processor (and debugger) hangs while debugging OTG USB Host
- TIM DMA burst (DMAR/DCR) does not work properly, if TIMx_DMAR observed in debugger

Created: 13.Jun 2020, Modified: 27.Jun 2020

5. 'F4/'F2 CCM RAM is not good for DMA, nor for bit-banding, nor for code

In the upper-end STM32F4* family and in STM32F2, there are two or three SRAMs connected to the busmatrix, available for all purposes, including DMA; and they can be also accessed quasi-bit-wise using the Cortex-M4 bit-banding facility. These SRAMs are mapped in one continuous address space, so that many users are not aware that these are separate memories.

However, many users notice, that their size does not add up to the advertized total SRAM size. The reason is, that there is one more 64kB chunk of RAM, which is NOT connected through the bus-matrix, but it is directly connected to the processor's D (Data) port.

This has several consequences:

- CCM RAM cannot be used for DMA - as it is not connected to the bus-matrix, neither of the two DMA units can access it
- similarly, none of the modules using its own DMA can use it - that includes ETH, OTH_HS (although using DMA is not mandatory for that one) and LTDC (where present)
- CCM RAM is not in continuous address space with other SRAMs
- CCM RAM cannot be used for bit-banding (the bit-band attachment is on the S-port of the processor; this is also reflected in the address space - CCM RAM is addressed at 0x1000'0000 whereas bit-banding RAM area starts at 0x2000'0000)
- code cannot be executed from CCM RAM (as processor fetches code only through its I-port)
- as a special case of the previous constraint, interrupt vector table cannot be placed to CCM RAM (to speed up interrupt execution, upon interrupt entry the vector is fetched through I-port simultaneously with stacking of registers through D- or S-port) - this issue was reported by user with nick *bson* on eevblog forum
- there's also a positive one - all accesses to CCM RAM are always single-cycle, contrary to accesses to SRAMs through the bus-matrix (S-port imposes a 1-cycle penalty for reads, bus-matrix may impose delays due to arbitration policy and collisions with other masters on the matrix, writes may be delayed due to buffered writes to slow peripherals, etc.)

In particular, if one unknowingly attempts to use CCM RAM at the memory port of DMA, DMA will fail with throwing Transfer Error.

So what's CCM RAM good for? It's really great to keep all the "pure" data in it. The simplest thing to do is to set stack and heap into it; but also data which are not intended for input/output, and/or data upon which extensive calculations are to be made.

---

* In ST nomenclature, this includes both the Foundation lines and Advanced lines, but excludes the 'F446. It means, 'F405/407/415/417, 'F427/429/437/439, and 'F469/476, all have CCM RAM.

Created: 20.Jun 2020
comment on vector table added 30.Oct 2021

6. RTC seemingly does not run, if only time is read

In order to present the software with a consistent set of the timekeeping registers (RTC_SSR, RTC_TR, RTC_DR), the RTC (except the 'F1 family, which has a significantly simpler RTC) implements a mechanism, where reading subseconds from RTC_SSR or time from RTC_TR will lock ~~all three registers~~ the "higher rank" registers, until date is read from RTC_DR. The "true" timekeeping registers will continue running, only the readout interface towards APB/processor is locked.

Depending on the particular order in which a debugger reads out the RTC registers, the lock mechanism may also influence the way how the RTC timekeeping registers are presented in debuggers - they may display an old value, or inconsistent values.

The lock mechanism results in the most common RTC-related complaint - an unaware user may want to read only the time from RTC_TR, but once he reads RTC_SSR (e.g. because that's what a time-reading "library" function does), RTC_TR readout stops changing, making the impression of a stopped RTC.

While it's easy enough to avoid this "problem" by reading RTC_DR after each reading of RTC_TR, the RTC has an option to suppress this mechanism, by setting RTC_CR.BYPSHAD=1. Then, it's upon the user to ensure registers' consistency (the RM recommends to read all registers twice, and repeat if they don't match).

Sadly enough, the RTC locking mechanism does not work properly in most STM32s (this appeared in errata of all STM32s starting appearing there by the end of 2016, so some newer STM32s may have this fixed, check your STM32's errata). This means, that for consistency, the three registers have to be read out twice and compared, regardless of RTC_CR.BYPSHAD setting. There may be a difference in timing of the read with different setting of RTC_CR.BYPSHAD, though.

Created: 27.Jun 2020, Modified: 1.Sep 2021

STM32 gotchas

7. Interrupt called without reason

Sometimes an interrupt service routine (ISR) is called without any apparent reason.

This happens immediately after a "legitimate" interrupt, if the flag in peripheral causing the interrupt is cleared late in the ISR

The root cause is, that it takes time, until a write from processor actually reaches the peripheral (from processor through its write buffer, the busmatrix, the AHB/APB bridge with its write buffer, getting synchronized in that bridge to the APB frequency) and then more time until the change from the peripheral propagates all the way through to NVIC (again through synchronizers from the APB clock domain to system/processor clock domain)[1]. This is exacerbated by the fact that peripherals may sit at slow APB buses, with clocks divided significantly down from the system clock.

Consider, for example, the case, where timer has enabled its Update interrupt, and upon timer rollover, the TIMx_SR.UIF gets set by hardware. This propagates through resynchronizers into NVIC, where it triggers the interrupt process. Processor executes the ISR entry stacking and then executes the instructions within the ISR. Let's suppose, that just before ending the ISR, there is an instruction which clears TIMx_SR.UIF, by writing a word which has 0 at this flag's position[2]. This word, passing from processor, has to acquire bus from the bus-matrix's arbiter, then it hits the AHB/APB bridge where it has to wait until synchronization to APB clock, when it gets to the timer. Then the timer's cleared interrupt output signal has pass through synchronizers into NVIC. This may take a couple of system clocks, but the processor won't stay and it starts to exit the ISR, and as due to the tail-chaining feature it starts to check for new interrupts quite early, it still "sees" the timer interrupt active, so it kicks in again.

The reason why this process is so complicated is, that these chips are not microcontrollers built tightly around a processing core and sharing its clock. Rather, they are systems-on-chip, SoC, equivalent to a whole computer from the past, stitched together more or less loosely from a processor core, peripherals, and an interconnection mesh (data and signalling).

Unfortunately, this problem is difficult to remove entirely.

Symptomatically, it's relatively easy - each interrupt source has to be qualified, i.e. the ISR has always to check, whether the flag causing the interrupt is indeed set in the peripheral and the program has to return from ISR if not. This is what the vast majority of users should do.

However, if one, by early clearing the interrupt source, would like to avoid the unnecessary calls to the interrupt entirely (and possibly avoiding the test, if there's only one single source for the interrupt), there's no much help - exact documentation is non-existent, the best one can do is test/benchmark and hope that he/she won't make any mistake. Note, that adding elements to the program, which potentially cause bus conflicts (e.g. DMA), can increase the latencies between processor and peripheral, thus re-introducing the problem which had possibly been removed in a simpler version of the program. This in most cases is best to be avoided, even at the cost of ineffective ISR, using the above "always verify the interrupt source" method. Lengthy ISR may almost safely guarantee that the verification is not needed; but then a bit of inefficiency usually won't hurt lengthy ISRs...

Using barrier instuctions (DSB or ISB) in the processor won't yield the required safety - the root cause is outside the scope of the processor, and dependent on the particular implementation of the busmatrix and peripherals.

This article in ARM/Keil's support describes the same problem. Note, that they don't acknowledge existence of any other source of delay, and the "second, system level buffer" mentioned there is assumed to be immediately on the processor's port, on the entry to the busmatrix. This is rarely the case, that's why the solution described there (performing *any* write) is not likely to help in most of the cases.

This gotcha describes a related problem, which occurs when pending interrupts are cleared outside an ISR.

---

1. A nice example illustrating the second part of the problem, i.e. propagation from peripheral (here: EXTI) to NVIC, is here.
2. You don't want to do this by RMW i.e. using &= or |=, and you don't want to use bit-banding to do this on Cortex-M3/M4-based mcus either. Here's why.

Created: 5.Jun 2020 Modified 8.Jun 2020

8. Timer does not work if ARR=0

That timer does not run when TIMx_ARR=0 is a well-documented feature. Nevertheless, it occasionally happens that users set ARR to 0 (sometimes just by leaving the respective field in the "libraries"' init struct at its default zeroed vale) to find in surprise that it is not running.

Not setting ARR at all won't result in this, as ARR is initialized to all-ones (i.e. 0xFFFF for the 16-bit timers, 0xFFFF'FFFF for the 32-bit timers).

Created: 11.Jul 2020

9.TIMx->SR &= ~TIM_SR_flag results in lost interrupts (don't RMW or bit-band on status registers)

This gotcha is by no means limited to the timer status registers, but pertains to **all peripherals' status registers, bits of which are of rc_w1 or rc_w0 type**.

Status registers in peripherals are not "memory-like" registers, where one can simply write a value which then impacts the way how the peripheral works, but otherwise reads back the same value as written. Instead, they are "changed from inside", i.e. hardware changes their values (usually single bits) indicating various state changes of the hardwaare. Often, the change is "acknowledged" by the user by writing a certain value to the same register.

As the various peripherals' IPs in STM32 have various origins (some are purchased from 3rd parties, others are developed in-house but for historically various purposes), the exact method how the status bits are cleared varies. This is indicated by both the narrative in description of the given register, and also the marking of the regster bits: *rc_w0* indicates a bit cleared by writing 0 to it (e.g. bits in TIMx_SR); *rc_w1* indicates bit cleared by writing 1 to it (e.g. bits in EXTI_PR).

When writing to registers with *rc_w0* bits, these bits are not affected by writing 1, similarly *rc_w1* bits are unaffected by writing 0. Thus, the correct way to clear a given flag in such register is to write a mask, e.g.

```
TIM1->SR = ~TIM_SR_UIE;    // rc_w0
EXTI->PR = 1<<10;          // rc_w1
```

Unfortunately, many users want to treat these registers are normal "memory-like" registers, using in C the "compound" |= and &= operators. This results in a read-modify-write (RMW) operation, where the original value is read out from peripheral register into the processor, the required logic operation is performed, and the result is written back. Performing **RMW (&= or |=) with status registers is incorrect and dangerous**, as this code can inadvertently clear also bits which were not intended to be cleared.

For example, let's assume that in timer, Update and Capture-Compare 1 interrupts are enabled; Update interrupt was detected and is about to be cleared using &= just before a CC1 event happens. As first part of &=, the processor reads TIMx_SR where only UIF is set. It performs the & which results in all-zeros, but at the same time, the CC1 occurs, so in the actual TIMx_SR both UIF and CC1F are set. Processor writes the calculated all-zero value into TIMx_SR by which it clears both UIF and CC1F, and this means, that the processor will never "see" thus won't handle that CC1 event.

There are also status registers which are cleared by writing to a different register, see DMA status registers; and some bits are cleared by a certain sequence of reads/writes to other registers, e.g. notoriously, the UART RXNE register is cleared by reading the UART data register. These schemes may have their own quirks, but generally are not affected by the problem described here. It's not unusual to see users to RMW into the separate-clear-registers, but while it's useless as those registers usually read constant 0, it's also harmless there.

It may be surprising, that using bit-banding is subject to the same RMW problem. The following article goes into details with that, reiterating partially what has been written above, from a slightly different perspective.

Following article was published originally at community.st.com; but due to software migration on that site it ended up somewhat crippled. This is a reconstruction, perhaps slightly better readable.

Bit-banding is dangerous when used on hardware-set status registers

On the ST's FAQ page, the following could be read some time ago:

> Use Cortex-M3 Bit-banding feature for interrupt clearing since it is an atomic operation and NVIC pending interrupts will be ignored during this operation, however Read-Modify-Write is not.

Now this FAQ page is gone already, but the quote is perpetuated on the web and in some materials. The problem with it is, that it is not entirely true. It's true that NVIC pending interrups will be ignored during bit-banding, but it's not true that it's a good method to clear interrupts. In fact, it's dangerous, don't do it unless you exactly know what are you doing.

So let's get this straight.

Bit-banding is a feature of ARM Cortex-M3 and Cortex-M4 processors, allowing certain portions of memory space (including a portion which is usually mapped to peripherals) to be accessed in bit-wise manner. This feature was introduced to attract programmers used to bit-addressable memory from other mcu architectures, most prominently the x51. It is present only in Cortex-M3 and M4, i.e. not present in M0

and M0+, nor in M7. Even in M3 and M4 it is an optional feature and implementers (semiconductor manufacturers) may chose whether to implement it or not - ST's implementation always do implement it, i.e. bit-banding is available on the 'F1, 'F3, 'F4, 'G4, and 'L1 and 'L4 subfamilies.

The bit-wise access is realized through a respective alias region, where every single bit in the original memory./peripheral address space has assigned a corresponding word (32-bits). Reading that word return 0x00000000 or 0x00000001, depending on what is the state of the corresponding bit. When writing to that word in alias region, the lowermost bit will be actually written into the original bit, not affecting other bits in the word containing the original bit.

This is how things look like from the processor's (and thus the programmer's) point of view. But to understand, what is going on, we need to get down to the nasty details of how is this feature implemented in hardware.

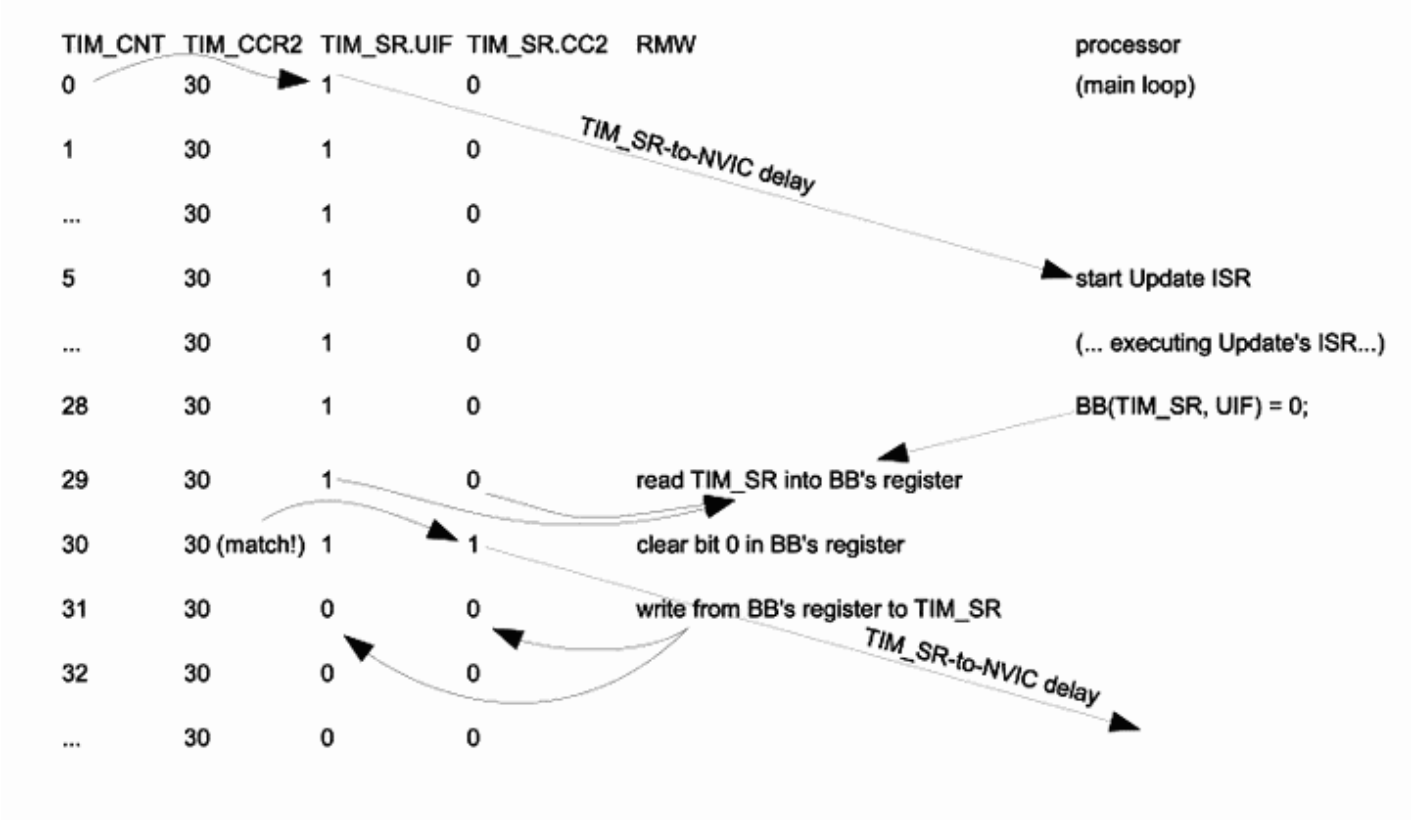
The truth is, that contrary to x51, there is no special hardware allowing to flip individual bits. The processor is still interfaced through a 32-bit bus matrix to 32-bit memories and peripherals, so it can only manipulate data in 32-bit chunks (more precisely, it can also do it in 8-bit and 16-bit chunks, if the attached memory or peripheral implements the byte-select signals of the AHB bus; but never in single-bit). So the trick lies in a simple attachment between the processor's S-port and the bus matrix: when the processor attempts to read from the bit-addressable area, the attachment converts the bit-address to the basic word's address, reads from the bus matrix at that address, takes the read word and rotates it the required number of bits and submits that as result to processor (the processor is stalled by the attachment all that time). Writing is slightly more tricky: the attachment issues first a read on the real word address, then takes the read data, masks the required bit, replaces it with the written one, and then performs the writeback through the bus matrix.

So, a bit-banding write is in fact a read-modify-write operation on a whole 32-bit word, from the point of view of the attached memory or peripheral. During this time, the AHB bus is locked down (there's a special signal for that in the bus), so no other master (such as DMA) can interfere. The processor is left to run until it attempts to access the S-port again, when it is stalled until the operation ends.

This means, atomicity of the operation is preserved, as far as the program is concerned (in this the quote is true); and also the possibility of other busmasters interfering has been taken care of. So what could possibly go wrong?

The peripheral itself.

In many peripherals, there are status words containing individual status bits indicating the states through which the internal state machine of the peripheral has passed. As these are set by hardware, they are usually of the clear-by-writing-1 (c1) or clear-by-writing-0 type (c0) - in the former, writing 1 clears such bit but writing 0 leaves it unaffected (and in the latter it's exactly the opposite), so the proper operation to clear certain bits in such register is to write a mask, not to read-modify-write. And this applies not only to software RMW (i.e. register |= mask or register &= ~mask, depending on whether it's c1 or c0 type which many users already know is no-no), but also to the hardware RMW. If the hardware sets a bit while other bit is being cleared through RMW, the writeback clears the newly set bit, too. The following scheme may perhaps illustrate this better on the case of TIM_SR register (bits of which are c0 type):



The write from BB's internal register clears unexpectedly the CC2 interrupt flag. I made up the particular numbers - I don't know what will be the latencies exactly, so the "sweet spot" for the bitbanding write instruction timing for the problem to occur will be most likely different from 30. Note, that even then the CC interrupt *will* happen as the signal has already started to been passed to NVIC; except that in that ISR, when checking for interrupt source, none will be found.

I tried to visualize this risk in a simple example (to be compiled with augmented device headers) for the 'L476 DISCO. The whole system is run on a slow system clock, MSI set to 100kHz, so that the result is visible on blinking LEDs. There are no AHB/APB prescalers nor prescalers in the timer, as that's the simplest possible setting directly converting to the scheme above. A timer (TIM1) is run with ARR set so that it overflows roughly at a 10Hz rate. There are two interrupts set, one from Update and the other from Capture2. Green LED is toggled at the update rate (in fact it is toggled by hardware through CH1; I might've do it in the Update ISR by software, the result would be the same); red LED is toggled in the CC2 interrupt. To find the "sweet spot", the CC2 event is delayed from the start of cycle more and more in each update cycle, simply by incrementing the CCR2 content (shadowing is switched on for the changing CCR2 to be accepted correctly). The fact that CC2 interrupts are missed because of the bit-banding clearing of Update flag, when the "sweet spot" is reached, is visualized by red LED stopping to toggle from time to time, while green LED toggles continuously:

In the isrCnts struct-array there are counters counting the occurence of Update ISR with Update flag set (.up), occurence of CC ISR itself (.cc) and occurence of that ISR with CC2 flag set (.cc2). This is how the vicinity of the "sweet spot" in this counter looks like:

```
{up = 28, cc = 28, cc2 = 28},
{up = 29, cc = 29, cc2 = 29},
{up = 30, cc = 30, cc2 = 30},
{up = 31, cc = 31, cc2 = 30},
{up = 32, cc = 32, cc2 = 30},
{up = 33, cc = 33, cc2 = 30},
{up = 34, cc = 34, cc2 = 30},
{up = 35, cc = 35, cc2 = 30},
{up = 36, cc = 36, cc2 = 31},
{up = 37, cc = 37, cc2 = 32},
{up = 38, cc = 38, cc2 = 33},
```

The same program compiled with

```
// #define USE_BITBAND
```

commented out, i.e. the TIM_SR flags clear is performed by straight write of the mask:

```
#ifdef USE_BITBAND
    PERIPH_BB(TIM1->SR, TIM_SR_UIF_Pos) = 0;
#else
```

```
    TIM1->SR = ~TIM_SR_UIF;
#endif
```

results in the three counters in isrCnts match perfectly, while counting up, all the time. No ISR missed.

Boring... ;-)

Created: 18.Jul 2020

10.Strange behaviour after increasing system clock frequency

Program in STM32 usually runs from the built-in FLASH memory. As all FLASH memories, this one is relatively slow, allowing to read out data at around 25MHz. STM32 after reset run from an internal RC oscillator (HSI or MSI) at a frequency lower than that, so they safely run directly from the FLASH. But users usually want to utilize the higher possible system clocks, ranging from 48MHz in 'F0 through 150-200MHz in 'F4/'F7 all the way to 480MHz in 'H7. How could possibly the slow FLASH feed instructions at this rate?

There are several techniques employed in the various STM32 families to allow this - prefetch, jumpcache a.k.a. ART accelerator, conventional cache - which are aimed to mitigate the "slowness" of FLASH. We are not going to discuss their respective merits and drawbacks here, because they won't change the fact that the FLASH is physically slower than the available system clock.

The FLASH controller thus has to be "told", how faster the system clock is. This ratio is commonly called **FLASH latency**.

There is basically a divider of the system clock towards the FLASH clock, and the user has to set it *before* increasing the system clock frequency, otherwise the FLASH hardware won't be able to deliver the subsequent instructions after the frequency switch. As a consequence, the FLASH subsystem imposes waitstates to the processor, whenever the processor attempts to fetch an instruction or read data, which is not already present in the prefetch registers or caches.

The latency divider ratio is usually set in **LATENCY field of FLASH_ACR register**. There may be a prescribed procedure to follow for both increase and decrease of system clock; read your STM32 Reference Manual's FLASH chapter.

Created: 25.Jul 2020

11.DAC output does not go rail-to-rail

Users sometimes observe, that they are unable to reach one of the rails (VREF-=VSSA and VREF+) or both, with setting the DAC to its extreme values.

In the older STM32 models, there's an inherent limitation - the DAC output, as per datasheet, can swing between 5mV and VREF+-1LSB (1LSB in 10-bit DAC with VREF around 3V is around 3mV). However, this limitation is not present on the newer 'Gx lines, and also users usually report more than that - sometimes up to tens of LSB, or tens of mV.

But reason for this is, that DAC in STM32 has a maybe **surprisingly high output impedance**. This *is* documented in the datasheet, the typical output impedances of DAC being around 15kΩ (exact value depending on particular STM32 model). The described effect is then just the consequence of excessive loading of the DAC towards some particular potential.

The STM32 DMA has also an optionally switchable output buffer. Those, who are aware of the high output impedance, might switch it on, just to find that it worsened the situation with reaching the rails. Indeed, according to datasheet, the output buffer's swing is within 0.2V of the rails. This is given by the limitation of used technology, which is primarily aimed at digital, rather than analog performance.

As an additional effect with this constraint, if the output buffer is overdriven, i.e. if DAC is set to output closer to rails than 0.2V, the buffer's opamp gets saturated, and coming out of this saturation results in a longer than expected transition.

So, if rail-to-rail operation of the DAC is desired, with higher loads, an external buffer has to be employed, or external DAC with desired parameters has to be used.

Created: 1.Aug 2020

12. UART parity bit missing

Users find it surprising, that they set UART to N data bit plus a certain (odd or even) parity, but the UART transmits (and receives, too, although that's not that conspicuous) only N bits altogether... so the parity bit is missing, right?

Reason for this is, that the set number of bits **includes** the parity bit. In other words, if parity is set, the highermost (latest transmitted) bit is not taken from the data register (DR or TDR), but is replaced by the calculated parity. Similarly, in the received frame, the last bit is not stored to the data register (DR/RDR), rather, it's checked against the parity calculated from previously received bits.

This is different from most other UARTS out there, where the parity bit is appended to whatever number of data bits is set. This is why it is surprising, even if clearly documented in the USART/UART chapter of RM.

As the older STM32 UARTs are limited to 8 or 9 bits (and the newer ones to 7, 8 or 9), this also means, that when parity is used, the maximum number of data bits is 8.

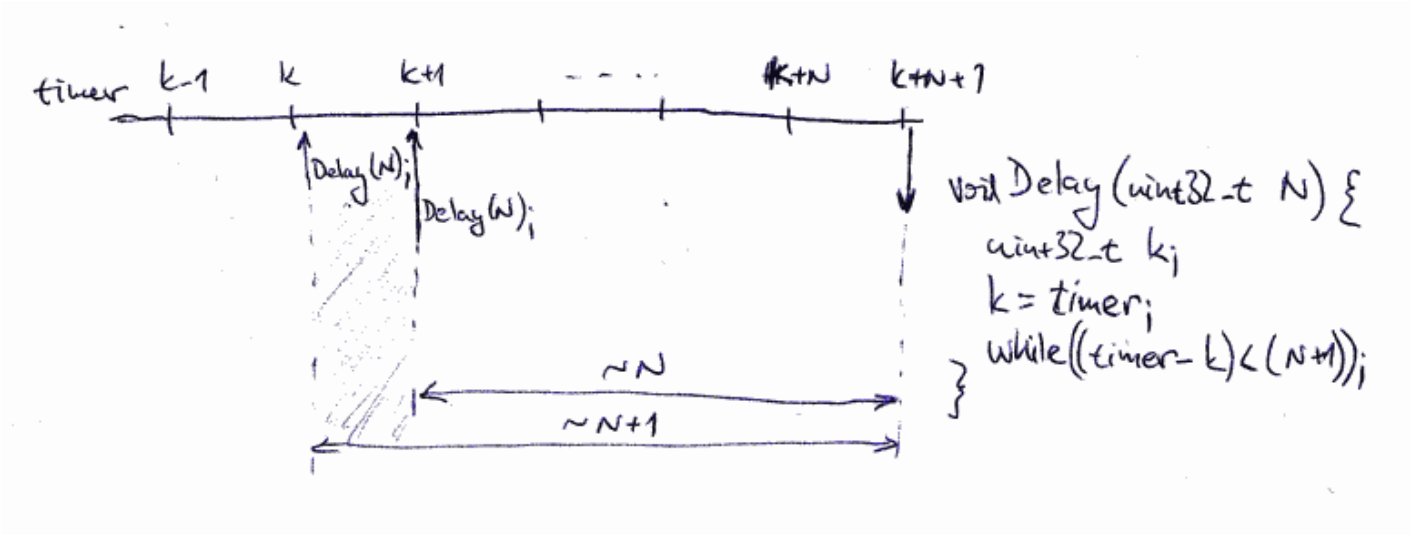
Created: 8.Aug 2020

13. Delay is 1ms longer than required

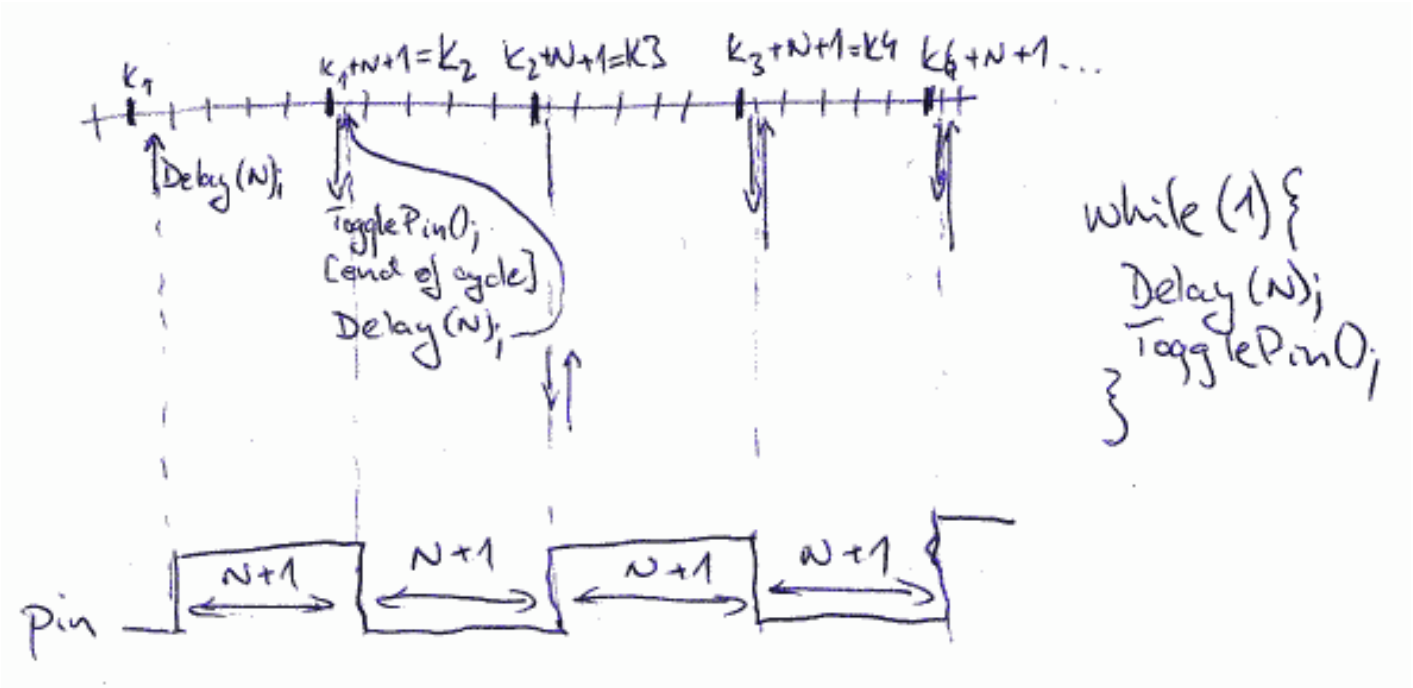
Software delays in various libraries are often derived from a free-running timer, i.e. the timer is not reset/started/stopped with the delay routine call, as it potentially provides timing for other parts of the software, too. In the ARM Cortex-M realm, such delays are often derived from the SysTick timer, which is part of the processor/core. In the STM32/Cube/HAL environment, HAL_Delay() is exactly such a delay.

Users sometimes find out with surprise, that these delays last one time unit (usually millisecond) longer than their prescribed duration.

However, this is quite naturally given by the granularity of the underlying timer. The delay routine, when called with parameter N, looks at that timer's current value, say it's K, and then waits until that timer rolls over from K+N to K+N+1. If the call happens just moments before the timer rolls over from K to K+1, the delay will be just slightly longer than N time units. If it's called slightly after it rolls over from K-1 to K, the delay will be just slightly shorter than N+1 time units.



Often, a basic test consists of repeatedly toggling an output pin and calling the delay. In this case, the delay's call will happen shortly after the previous delay routine ended, i.e. it's shortly after the timer's rollover. This is then exactly the scenario where the user would see the delay lasting consistently around one time unit longer than expected.



Also, if there are any interrupts run in the microcontroller, execution of the ISRs can increase the delay's duration.

So, these delays have to be understood as lasting "at least the prescribed period".

If delay with more precision is desired, a timer running at higher clock is to be used, possibly employing

interrupts of appropriate priority (so they can interrupt other, potentially lengthy ISRs). In 32-bit mcus, interrupts and software loops may take dozens of machine cycles, so that's also the limit of the achievable precision. If even higher precision is desired, hardware has to be used, e.g. output of a timer's compare unit.

Time-wasting delays lasting milliseconds are normally used only in quick testing setups. In production code, only delays lasting a few machine cycles are constructed as time-wasting, as it's inefficient to solve these using timers/interrupts. Also in this case, the delay routine/sequence is usually presented as "lasts at least xxx cycles/time units". In the past, NOP instructions have been used for this sort of delays, but it's hard to properly develop such delays with the increasing complexity of 32-bit mcus, as these delays have to defeat all the measures added to increase the execution speed (caching of all forms, pre-execution NOP removal, branch prediction, etc.), and do it in a defined manner.

Created: 15.Aug 2020

14. GPIO is not toggling at a rate promised by datasheet

In datasheet, GPIO "output frequency" is given, often at a rate equal to the maximum system clock (in *I/O AC characteristics* table, depending on the respective GPIOx_OSPEEDR setting). Sometimes, users complain that this frequency cannot be achieved, not even close, off by perhaps an order of magnitude.

This results from a misunderstanding of what the given table/characteristics means. It does not mean, that there is a way to toggle a pin from software at the given rate. Rather, it means, that the output signal's edges are fast adequately to a waveform with given frequency. Also, as the same set of transistors drives the output when set to GPIO Out and when set to Alternate Function (in GPIOx_MODER), if some peripheral is set to output a waveform of given frequency (e.g. Timer outputting PWM), this parameter ensures that with appropriate setting of GPIOx_OSPEEDR the given frequency is attainable.

Nonetheless, it is not excluded that software can toggle the pin at that rate, or at least make several successive transitions on one or more pins at that rate; but there are several obstacles to be overcome. The usual reason, why users want to achieve that, is an attempt to bit-bang some interface. Users often write a simple test code similar to this:

```
while(1) {
  HAL_GPIO_TogglePin(SOME_PIN);
}
```

This is a function call in C, which can compile to quite many instructions, i.e. it takes many machine cycles to execute. Even a loop written in assembler won't allow toggling pin at the maximum frequency in most mcus, given the jump at the end of loop takes some machine cycles to execute, too. There may be also issues associated with executing from potentially slow memory, contention on buses, etc.

As is the case in the 32-bitters, tight timing is best to be solved entirely using hardware. If bit-banging is necessary and tight timing is desired, the user must be aware of all the possible pitfalls and must accept the fact that a certain timing may be subject to jitter, or in a given mcu model, may not be achievable at all.

Related:

- 108. Toggling GPIO in 'H7 is slow

Created: 22.Aug 2020
Modified: 11.June 2022

15. Timer appears to run slower than is set

This assertion is usually brought up after an attempt to toggle a pin at MHz rate in a timer-triggered interrupt; often using "library" facilities, which further increase the execution time of the interrupt service routine (ISR). Usually, the execution time of processor is completely exhausted by the interrupts, and their execution rate (and thus the observed pin toggling) is then not given by the timer anymore, but by the mere execution time of the ISRs being continuously executed back-to-back.

Interrupts in the 32-bit mcus have a relatively high overhead (and, somewhat associated, relatively high latency), especially when compared to 8/16-bit mcus. The entry of interrupt alone, performed by hardware, takes 12 cycles in the Cortex-M mcus. Add to this the usual C function prologue and epilogue, and some code to perform the required action (e.g. toggling a pin), using "library" code and compiling at low or no optimization setting (usually to facilitate debugging), and an apparently minimal ISR can easily run several dozens or even a few hundreds of cycles long.

In STM32 running at tens of MHz, this means, execution of a typical ISR may last in the order of one to few tens of µs, i.e. maximum reasonable ISR repetition rate is a few hundreds of kHz.

Generally, high rate interrupts are better to be avoided. Faster processes are preferrably be solved entirely by hardware, either using native facilities of the given peripheral modules (e.g. a pin to be toggled by the Output Compare/PWM mechanism of the Timer), or combining peripheral modules (e.g. using timer-triggered DMA), whenever possible.

Related:

- 26.Don't use printf() (or other lengthy functions) in interrupts.

Created: 29.Aug 2020

STM32 gotchas

16. Interrupt does not fire (some troubleshooting hints)

What to do, when interrupt is enabled in a program, but the Interrupt Service Routine (ISR) does not run?

As usually, the most straighforward method is to follow the path of execution in hardware, i.e. in the mcu itself. This is contrary to the usual wishes of the novices, who don't want to go beyond the comfort of the IDE and software-writing process, using some sort of "libraries". Sorry, but that's not how it works, the mcu does not care about libraries nor software.

One more "inconvenience" is, that successful debugging requires understanding of the underlying mechanisms, which is achieved by reading the available documentation. For interrupts, this involves the Reference Manual (RM), mainly *Interrupts and Events* chapter (in some newer RM named differently, or even split e.g. into *Nested vectored interrupt controller (NVIC)* and *Extended interrupts and events controller (EXTI)* chapters); plus the chapter related to the particular peripheral from which the particular interrupt signal originates. However, the real working of NVIC is not described there, as it is part of the processor/core, so the related Programming Manual (PM) has to be consulted (and perhaps other related ARM material). And, note, that from ARM's point of view, interrupts fall into the somewhat broader group of "exceptions", so you want to read everything about "exception priorities" there.

As usually, the most convenient tool to use is the built-in debugging facility plus the associated external hardware (debugging "pod") and software (often IDE, sometimes bare gdb with associated interfacing layer (like openOCD) or other similar tools). The following lines are written mostly with this method in mind. However, as usually, this is not a necessity, the good programmer has a whole range of tools and methods at hand - some of them avoiding the issue that the original problem may actually be caused by the built-in debugger being intrusive to a certain extent.

Here are some steps to contemplate:

- Do you properly check that the ISR does not run? Place a breakpoint directly into beginning of the ISR (not just into any "callback" from a "library", or any other secondary function call). Maybe better, toggle a pin at that place, observing it using an oscilloscope or logic analyzer (LA).
- Are other interrupts at this point running successfully? If yes, that excludes some modes of failure (e.g. global interrupt disable).
- Check, that the desired interrupt is enabled at its source, in the peripheral's registers.
- Check in the peripheral's status bits, that the desired state leading to interrupt has indeed been achieved (don't forget, that in some peripherals (e.g. SPI and UART Rx), observing registers using debugger may actually clear the desired interrupt).
- Check in NVIC_ISERx, that the given interrupt is enabled. If not, check if you've called `NVIC_EnableIRQ()` with the proper interrupt number.
- Check - best in mixed C/disasm view - that the proper vector to the ISR is inserted to proper position inside the interrupt vector table. If not, check if the ISR name matches exactly (noting that C is case sensitive) that in the vector table (usually in the startup code; the interrupt chapter in RM contains the reference list of interrupt vectors)[1].
- Check, that SCB_VTOR points to the proper interrupt vectors table.
- Aren't interrupts disabled globally? By default, they are not, so watch out for any occurence of `__disable_irq()` (or `CPSID I`) in your program (it acts on a special-purpose ARM register, PRIMASK, some debuggers can display its content, too).
- Isn't the program being stuck an interrupt with the same or higher priority? This should be easy to find out by stopping the program and observing where it currently executes; or observe SBC_ICSR.VECTACTIVE.

Also note, that the debugger will usually prevent interrupts being invoked, while single-stepping.

---

1. Users of C++ often don't realize, that in C++ function names are mangled by the compiler, so the linker cannot match the name in the vector table to the user-written ISR and falls back to the default "weak" handler, usually implemented as an empty infinite loop. The canonical way to avoid name mangling in C++ is to enclose the given function (ISR) into `extern "C"{}` block:

```
extern "C" {
    void TIM2_IRQHandler() {
      [...]
    }
  }
```

Created: 5.Sep 2020, footnote 1 added: 16.Oct 2021

17. Program freezes after enabling an interrupt

What to do, when enabling interrupt in a program causes other effects of the program (e.g. "heartbeat" blinking in main loop) to freeze?

To certain extent, this situation is similar to the previous issue, and some of the steps mentioned there are usable here, too. Debugging now might start perhaps with stopping the program in the debugger and observing, "where did it go".

Some of the root causes may be:

- ISR does not clear the flag in the peripheral, which invoked the interrupt
- ISR takes too long to execute (often inappropriately calling routines such as `printf()`) while the peripheral invokes it too often
- ISR contains a software bug, leading to infinite looping inside the ISR (a nasty variant of this bug is a loop waiting for a hardware flag, which won't be changed under certain circumstances, e.g. when debugger is not connected)
- ISR contains a software bug, resulting in the program ending in some of the fault handlers (usually HardFault)
- ISR name doesn't match the prototype in the ISR table, and falls into the "catch all unspecified interrupts" routine.
- SCB_VTOR is not set appropriately in a program which is not located at the default position in FLASH (0x0800'0000) - usually in an environment with custom bootloader

---

Created: 12.Sep 2020

18. ADC readings not as expected

The ADC input, from external circuit's point of view, can be modeled as a cca 6-8pF capacitor, charged to an unpredictible voltage[1], switched to the input pin at the moment when software (or ADC hardware in case of some of the "automatic" modes) starts the sampling process. While this capacitor is sometimes labeled as a Sample-and-Hold circuit, it's in fact part of the ADC structure itself; but that's just a matter of terminology, not influencing its outward effect.

This capacitor together with the signal's source impedance and the the switch's ON resistance[2] represents an RC circuit with its associated exponential charging process. Whatever is the charge state by the end of sampling period (as specified by software in ADC_SMPRx) will result in the value converted by the ADC. In other words, if the RC circuit is not settled by the end of the sampling period, the read out value will be different from the expected one. This process can be easily seen by observing the ADC input pin using oscilloscope, see e.g. this page (with exahustive explanation, albeit in Russian).

The net effect may be sometimes surprising, for example a signal slowly crossing the mid-range may appear to be "stuck" at a certain level - see "flat" sections inside the circles on this picture from this thread.

Another surprising effect occurs, when the same input signal source is sampled by several ADC channels. For example, ADC1 measures several inputs through its input multiplexer in succession, including one particular signal in question, with settings adjusted to determine their precise value; while ADC2 runs continuously conversions only on that one particular shared input signal, set as an ADC watchdog, in order to react quickly to sudden changes of its value. If ADC2 starts to sample at the moment when ADC1 finishes its sampling phase on that shared signal, the "dip" caused by switching of the ADC2's capacitor severely disturbs the value converted by ADC1; and vice versa. In this case, both ADCs have to be carefully synchronized to avoid this effect.

The most straighforward way to mitigate this effect is to avoid high impedance input source (e.g. voltage dividers with high ohmic value resistors). Amplifiers can be employed to decrease impedance, although they may introduce errors (offsets, drifts) on their own. Another method is to simply increase the sampling time

An often mentioned solution is to add a relatively large capacitor in parallel to the ADC input (large compared to the sampling capacitor, i.e. at around a few nF or higher). While this decreases indeed the source signal impedance at the cost of decreased bandwidth, if the RC constant of this setup is shorter than the sampling period, the capacitor may not be recharged from the primary signal source fast enough and the measured value may be off again; so here the sampling period has to be chosen carefully[3]

Internal structure and working of STM32 ADC, together with concise guidelines to its usage (including discussion of the effect described here) are detailed in AN2834.

---

1. Probably VREF+/2. However, the exact working of ADC is unpublished, so proper design has to assume the worst case, i.e. that the initial voltage of capacitor may be at either end of 0-VREF+ range.

2. Value given by datasheet, a few kΩ. This switch is part of the ADC's input multiplexer. In some STM32, this switch requires to activate a voltage booster, if supply voltage is below a certain level. In the 'L47x/48x, there is one extra switch at the ADC inputs.

3. Note that there is a difference between sampling time and sampling period. The former is given by setting in ADC_SMPRx register and represents the time while the sampling switch connects the sampling capacitor to input pin. The latter is given by the method how the ADC is triggered, and may be significantly longer, especially if the ADC samples many inputs within its input multiplexer in succession, or if the sampling is sparsely triggered by timer or software.

Created: 19.Sep 2020

19.Debugger keeps jumping into interrupt when single stepping

Of the features of debugger perceived by users as one of the most useful, is the ability to stop at breakpoints or upon command from the host PC, and then single-step code and observe changes as individual code lines are executed. Of course this has limitations, especially when it comes to optimized code; most users are aware of that and we are not going to discuss these principial limitations here.

Now during single-stepping, the processor is stopped between the individual steps. However, only the code execution in the processor itself is stopped, the rest of the mcu - its peripherals and interconnections - keep ticking and working[1]. So, most of the interrupt sources work, too, and indeed, they trigger interrupts in NVIC. This would make the program very likely jump directly into an interrupt when taking the first step, and that would make the whole single-stepping unusable. To avoid this, the built-in debugging interface (which is part of the processor core) has a direct link to NVIC (which is also part of the processor core), using which it can disable interrupts while the processor is stopped and is being single-stepped.

Unfortunately, this link is broken in the early versions of the Cortex-M7 (revision r0p1), see erratum #702596 in ARM's *Cortex-M7 (AT610) and Cortex-M7 with FPU (AT611) Software Developer Errata Notice*. In mcus using this revision of Cortex-M7, single-stepping is a pain, and users have to jump through hoops (essentially stopping all interrupts and/or avoiding connecting peripherals causing them) to be able to single-step some particularly intricate piece of code. Debugger makers also tried hard to work around the problem (see e.g. here and here), basically by issuing a global interrupt disable when the processor is stopped before single-stepping; however, contrary to the direct link between the debugging interface and NVIC, here it takes time until the host PC issues the interrupt disable command and it propagates all the way through NVIC, so if an interrupt happens to trigger in that time window, the first single-step steps into that ISR anyway, so the outward experience may be of a not very reliable operation.

In STM32, only the first Cortex-M7-based model, the 'F746 (and its crypt-enabled counterpart 'F756) are affected by this bug. The 'F72x and 'F76x subfamilies, as well as the 'H7 family, use newer revisions of the processor core, where the bug has already been fixed.

---

1. Timers and some other peripherals can be made to stop during debugging by settings in DBGMCU registers, but that's another story. That indirectly can mitigate issues with timer-based interrupts. There are still many interrupt sources e.g. external interrupts, which generally can't be avoided during single-stepping.

Created: 26.Sep 2020

20.NDTR problematic when DMA used as Rx FIFO

This is one of the "not confirmed by ST" issues, treat it as such. Also, having used mostly the dual-port DMA (as found in 'F2/'F4/'F7 and covered by AN4031), I don't have extensive experience with the single-port STM32 DMA (covered by AN2548), so I am not entirely sure how exactly this issue pertains to that one.

In order to keep up with data stream arriving[1] to communication devices such as UART or SPI, they have to be serviced by the processor timely, usually in interrupts. With higher loads on the processor, or if multiple ports to be handled, the processor might not be able to keep up with the incoming data stream, resulting in data loss. Traditionally this problem was coped with by adding FIFOs to the transceiving chips, to store up to several received frames before they were read out by the processor. We've seen this for example in the PCs in the 1990s, when the original 825x/16450 UART (having only the basic Rx holding register for one frame) got replaced by 16550 (with a 16-byte FIFO).

Why this works is, that interrupts (or even polling) may involve serious overhead in both executed number of instructions (ISR/functions entry/prologue and exit/epilogue) and the speed how they are executed (program flow disruption resulting in caches miss/purge/reload). Neither of these is really a problem on the 8-bit microcontrollers, (and FIFOs are rare indeed on those); but as mcus get faster, by employing all sorts of tricks from the "big" computers' world, these issues get more and more aggravated. So then it pays off being interrupted only seldom, and process bigger chunks of data at once.

The FIFOs on communication chips attempted to mimic the working of the original FIFO-less chip as close as possible, so that the original software drivers would continue to work. The same interrupt signaling and same register structures was employed, perhaps with a few added control bits in the registers. For Rx, the above benefit to materialize, this meant that the interrupt (signaling originally that the Rx holding register was not empty anymore) could not go up sooner than several frames were received into the FIFO (i.e. a certain threshold was reached in FIFO "fullness"), but then when read, it would not go down until the FIFO was drained completely. If only a few frames (below the threshold) are received, these would "hang" in the FIFO indefinitely; so to avoid this, there is also an idle timer needed to invoke the interrupt. This all results in a nontrivial circuitry (and indeed, the FIFO-containing UARTs have seen their fair deal of early design flaws, programmers had to learn to cope with).

In STM32, the original approach was to build simple communication modules, with the general-purpose DMA serving in the role of FIFO[2]. This makes sense, as there are many communication peripherals on a single STM32, yet usually only a few of them is used at once, so this approach spares silicon area avoiding FIFO memory and control circuitry in each peripheral, since most of them would get unused in a typical design.

To use DMA for reception, the Rx-nonempty signal of communication module (UART, SPI, I2C) is not used to interrupt the processor directly, but serves as request (trigger) signal for the DMA, which in turn transfers the received frame from the Rx buffer to a buffer in memory. The basic idea works nicely, however, the signaling described above is missing entirely. So this has to be worked around somehow. We here won't deal with the methods how to "alert" (interrupt) the processor of frames present in "FIFO"[3]; we will assume the processor is already aware of this and wants to pick and process the frames from the buffer.

To find out how many frames have been transferred by DMA, there's only one choice - to use the NDTR register. DMA decrements this register with each transfer. (DMA also increments the address registers if it is set so, and normally the memory side is; but the real "acting" address register is strangely internal to DMA and unreadable by the processor. The visible address registers never change and always read the value last written.)

The problem is, that **NDTR** counts the peripheral-side transactions, and it **gets decremented at the moment, when the DMA unit** internally arbitrated the request from peripheral and **started to transfer the frame from peripheral to its internal FIFO** [4] [5]. But it takes time until DMA succeeds to store that frame into the memory. If the processor reads the already decremented NDTR, and based on that, reads the frame from memory before DMA writes it, the **processor reads an incorrect value**.

The problem gets exacerbated by using FIFO in DMA. FIFO does not get stored into memory until its set threshold is reached; and also finding out from NDTR that FIFO threshold has been reached does not mean the data are already stored in memory, it means only that the process leading to storing has begun.

Many users see DMA transfer as an instantaneous process (and this notion is supported by the correct recommendation to avoid the relatively slow transfers involving the processor if DMA is available for the same task), but it is not. First, the frame has to be picked from the peripheral, and that may be slowed by both arbitration for APB access between DMA and other busmasters (usually processor) in the dual-AHB-to-APB bridges which are usually present with the dual-port DMA, and the sheer fact that APB bus is usually slower than AHB in the processors with the dual-port DMA. Then, on the memory side, the internal

arbitration within DMA has to be performed, and then DMA has to arbiter for the target memory in the busmatrix.

Unfortunately, I know of no good method how to find out, when are data indicated by NDTR already stored in memory. In direct mode, if NDTR indicates that transfer of next frame has begun, the previous frame is surely stored in memory; but method relying on that fact is usable only for a continuous incoming stream. The only official way to purge FIFO is to disable DMA and wait until its enable bit gets cleared (the very reason for this wait is the fact that the FIFO-to-memory store has to be accomplished) - but that would defeat the very purpose of using DMA. A "reasonable" time between NDTR read and data read from memory could be established by benchmarking for the worst case within the given application and then adding some "safety margin"; but it is far from being trivial to construct a benchmark which ensures the worst case to be achieved.

This issue has been discussed also on community.st.com (thread mingled due to forum software change).

---

1. Transmission (Tx) of course involves similar problems, too, but, usually, gaps in transmission can be tolerated, so this is an issue only in the relatively rare case when back-to-back communication is required. This is usually seen only when it comes to audio/video.

2. In newer STM32, as transistors get cheaper with increasing density of used technology, communication peripherals gradually get more and more FIFOs built into them. They then bring in their own gotchas, of course.

3. We will deal with this particular issue separately. Stay tuned.

4. Internally, the dual-port DMA uses FIFO (to store a single frame) also when FIFO is disabled, i.e. in so called Direct Mode. ([rant mode on] Do you also find it dumb, that the register bit, which means FIFO Enable, and is in the DMA FIFO control register, is called Direct Mode Disable? [rant mode off]). The proof is, under certain circumstances. FIFO error may occur also in Direct Mode.

5. Unfortunately, the ST documentation is not famous for documenting the hardware concisely and meticulously. For example, in RM0090 (for the "classical" 'F40x/'F42x line) rev.17, ch.*10.3.2 DMA transactions says this*:

> Each DMA transfer consists of three operations:
> • A loading from the peripheral data register or a location in memory, addressed through the DMA_SxPAR or DMA_SxM0AR register
> • A storage of the data loaded to the peripheral data register or a location in memory addressed through the DMA_SxPAR or DMA_SxM0AR register
> • A **post-decrement of the DMA_SxNDTR register**, which contains the number of transactions that still have to be performed

which is clearly not the case (emphasise was mine).

Created: 3.Oct 2020

21.SPI master NSS unusable

SPI is a simple and popular serial protocol to connect peripherals communicating at a relatively high speed. It is based on a simple shift registers, so it consists of one data line per direction (MOSI, MISO) and a common clock line (SCK).

There is usually at least one more signal, providing framing, but as SPI is more a defacto than a formal standard, different devices treat this signal differently. Usually, it is generated by master, it is active low, usually starts and stops slave's communication, i.e. resets bit counters and also provides higher-level framing, and often also control's slave's threestateable MISO signal, so that MOSI of multiple slaves can simple be tied together.

In STM32's SPI, this signal is called NSS (N stands for Negative i.e. active low, SS stands for Slave Select). When SPI is set as slave, it indeed is an input providing framing/bitcounter reset and threestating of MISO. When set as master, NSS can be set as input and then it provides a means for an external device to switch the module to slave. This is intended for multmaster communication; but the mechanism is not very robust (there's no provision to prevent masters to collide in starting communication at the same time) so it's not very usable in this way.

The more usual mode of operation is, when SPI master uses NSS as output and generates framing on it. Users sometimes expect that when NSS in STM32 SPI is set to output (SPI_CR2.SSOE=1), the SPI will generate framing signal on this NSS. However, there's no such hardware in the SPI. If SSOE=1, NSS is simply set to low as long as the SPI module is enabled (SPI_CR1.SPE=1). If SPI module is disabled (SPI_CR1.SPE=0), the NSS pin is not driven actively. If a pullup is switched on for this pin in GPIOx_PUPDR, the pin will be pulled up, but this may take time, depending on the capacitive loading on the pin, given the pullup is relatively weak (nominally 40kΩ).

Since the 'F0/F3 families, a new feature has been introduced in SPI, NSS pulse mode (see SPIx_CR2.NSSP). While this generates bit-counter-reset-framing automatically on NSS, this usually is not what most of the users expects, i.e. higher, protocol-level framing is still not provided. Moreover, NSS pulsing generates gaps between frames, resulting in discontinuous clock and effectively slowing down the communication.

So the general recommendation is, for framing/NSS towards a slave requiring framing, use any pin as GPIO Output, and handle it "manually"[1]. If more automation is needed, users with good command of the STM32 peripherals may attempt to use timers to generate framing (and sometimes also clock, at the cost of one extra pin for external interconnect between TIMx_CHx and SPIx_SCK).

---

1. When doing so, it may be not a good idea to time the end of NSS based on the BSY signal.

Created: 10.Oct 2020
Footnote added: 1.Apr 2022

STM32 gotchas

22.Writing one byte to SPI transmits two bytes

While SPI is a relatively simple communication interface, it is also the one which offers the highest transfer speed amongst the "traditional" communication interfaces (SPI, I2C, UART). It is not uncommon to see users pushing SPI clock rates to tens of MHz, making it a significant portion of the overall processor power to keep feeding the SPI with data continuously.

This is more pronounced, if the SPI is set for some reason to 8-bit[1] frames. The rest of this article assumes SPI to be set in this way.

ST's original plan was to use the DMA to help with this, but as price per transistor got lower with improving silicon technology, starting with the 'F0, the SPI has a 4-byte FIFO built in for both Tx and Rx. With that, another opportunity for optimization has been open: as it's not efficient to transfer individual bytes through a 32-bit bus, SPI set to 8-bit frames can now accept/supply 2 bytes at once, when the SPI_DR is accessed as 16-bit register (even if the processor is 32-bit, most of the STM32 peripherals are in fact 16-bit - this may have historical reasons, but this also may help reduce complexity and silicon area).

ST calls this feature **data packing** and it's well documented in the SPI chapter of respective Reference Manuals.

However, this feature for users accustomed to the older SPI versions resulted in a surprising behaviour, where one write resulted in transmitting two bytes (with the second byte being zero). The reason is, that in the device headers, the data register is defined as being 16-bit, so a simple

```
SPIx->DR = data;
```

line is compiled into a STRH instruction, i.e. a 16-bit write. Writing a single byte in C has to be accomplished by type punning, i.e.

```
*(volatile uint8_t *)&SPI->DR = data;
```

Consequence of data packing on data reading (i.e. Rx) are less conspicuous, but still can lead to confusing errors. Using SPI with DMA is less of a problem as users unaware of the data packing feature naturally implement the "correct" data access by setting DMA_CR/CCR.PSIZE to byte.

The lesson here is, that even if there's a great deal of compatibility in the STM32 peripherals among the families, and the newer versions tend to be reasonably backwards compatible, having the new features switched off by default; yet this is no hard rule and it pays off to read and re-read the manual and follow the instructions set there.

1.Actually, the frame data width in the newer SPI with FIFO can be set to any bit number between 4 and 16. Data packing applies to any bit number between 4 and 8.

In microcontrollers programming, problems usually have multiple possible solutions, all with their individual advantages and drawbacks, these being usually context-dependent. Markus H. suggested to use a union:

```
typedef struct
{
    volatile uint32_t CR1;         /* SPI Control register 1,                          Address offset: 0x00 */
    volatile uint32_t CR2;         /* SPI Control register 2,                          Address offset: 0x04 */
    volatile uint32_t SR;          /* SPI Status register,                             Address offset: 0x08 */
    union                           /* SPI data register,                               Address offset: 0x0C */
    {
        volatile uint32_t DR16;
        volatile uint8_t  DR;
    };
    volatile uint32_t CRCPR;       /* SPI CRC polynomial register,                     Address offset: 0x10 */
    volatile uint32_t RXCRCR;      /* SPI Rx CRC register,                             Address offset: 0x14 */
    volatile uint32_t TXCRCR;      /* SPI Tx CRC register,                             Address offset: 0x18 */
    volatile uint32_t I2SCFGR;     /* SPI_I2S configuration register,                  Address offset: 0x1C */
    volatile uint32_t I2SPR;       /* SPI_I2S prescaler register,                      Address offset: 0x20 */
} SPI_TypeDef;
```

While modifying the ST-supplied CMSIS-mandated device headers may pose problems e.g. when porting code or collaborating, and also anonymous union is not part of C99 but a gcc-specific extension which only later made its way into C11; this solution also makes the resulting source code cleaner and tidier...

Created: 17.Oct 2020
Note with alternative solution added: 5.Jan 2023

23.STM32 hangs in UART interrupt

Sometimes users experience a "mysterious" hang of their system. They might find out that the processor permanently "loops" into the UART interrupt. Why is that?

The gotcha lies in the fact that UART RX overrun (i.e. when as byte arrives to Rx while the previous has not been read already), indicated by USART_SR/ISR.ORE being set, does trigger an interrupt if USART_CR1.RXNEIE is set. Note, that this is the same bit which enables the "normal" Rx interrupt, too.

So, proper handling of the UART interrupt, when USART_CR1.RXNEIE is set, must include **checking both RXNE and ORE** in the status register (USART_SR/ISR), and using any procedure needed to **clear** them.

While RXNE is cleared automatically by hardware when reading the data register; clearing ORE differs between the UART versions. In the older version of UART, reading both status register and data register clears also ORE in hardware, very similarly to clearing RXNE (to be precise, clearing RXNE does not require reading the status register, but that normally is read in the ISR prior to reading the data register, anyway). Clearing ORE in the newer version of USART ('F0 and newer, having separate Rx and Tx data registers and ISR/ICR registers) requires writing 1 to USART_ICR.ORECF.

Note, that it may happen, that ORE is set while RXNE is not set (see *Overrun error* subsection of *Receiver* subchapter of *UART* chapter in RM, note starting *"if RXNE=0 [...]"*), so the interrupt handler must be written with this possibility in mind.

The newer USART has a USART_CR3.OVRDIS bit, which suppresses generating ORE. In cases, where data consistency is ensured by other means (e.g. higher-level packetization/framing/checksumming of data), this allows to write a simpler (thus faster) ISR without running into the problem described here.

Created: 24.Oct 2020

24.UART Rx stops working

In the previous gotcha, we've discussed the consequence of not handling UART Rx overflows in the UART interrupt. But what happens, if we don't handle overruns in polled, or DMA-based UART Rx implementations?

Well, the Rx simply stops working.

This is not well documented in the RM, but UART stops generating RXNE upon arriving valid bytes, when USART_SR/ISR.ORE is set.

So, it's vital, that this particular error is properly handled, both in polled, and DMA implementations. In the former, it will probably be part of the polling procedure, e.g.

```
uint_least8_t UsartWaitRxByte(USART_TypeDef * USART) {
  uint32_t sr;

  do {
    sr = USART->SR;
    if (sr & USART_SR_ORE) {
      (void)USART->DR;  // in older USARTs with SR, ORE is cleared by reading SR and then DR
    }
  } while (!(sr & USART_SR_RXNE)) ;
  return USART->DR;
}
```

For DMA-based implementations, it makes more sense to detect and handle overruns in interrupt.

As mentioned already,, in newer USART, overruns can be also ignored, by setting USART_CR3.OVRDIS.

There may be two possible implementations of this behaviour internally, with the same outward effect: the UART may receive but discard all incoming frames as long as ORE is set; or it may stop receiving altogether until ORE is cleared. It appears, the actual STM32 implementation does the latter, as it does not update other error flags (parity error, framing error) either. This behaviour is different from many other UART implementations. A minor, but interesting consequence was, where a nonstandard protocol used already on other mcus was reported to be unimplementable on STM32. This protocol stipulated ignoring incoming bytes (possibly in a sleep mode) until the transmitter transmitted a byte with deliberately crafted parity error, causing error interrupt, serving as a wakeup and framing element.

Created: 31.Oct 2020

STM32 gotchas

25.STM32 are not microcontrollers, but SoC

System-On-Chip (SoC) was a term used maybe a decade ago for chips, where a general-purpose computing core was attached to various peripherals (often heavyweight such as memory controllers or display controllers), originally intended to reduce need for pins for connections between the processor and peripherals. SoC often represented a chip-scale reduction of what was prevously built from individual chips on a board. SoC were generally seen as a counterpart to microcontrollers (mcu); SoC serving the high-end applications, mcus the low-end.

Conventional microcontrollers (mainly 8-bit, such as the venerable 8051 and its derivatives, 8-bit PICs, or the original AVRs; but mostly also the 16-bit microcontrollers) are built tightly around the processor core, with all the buses and peripherals sharing a single clock. This greatly simplifies understanding of the mcu's timing for the users. However, this approach has drawbacks, too: peripherals are designed in a dedicated way to a particular mcu and are not portable to other mcus, increasing development costs. Also when the chip crosses a certain size (physically, but also in terms of clock fanout), clock distribution becames troublesome, in some cases clock skew between different parts of the chip becomes a serious issue.

The 32-bit microcontrollers were created in a different way - they are based around slightly modified general-purpose 32-bit processor cores (ARM, MIPS), but the memories and peripherals are connected to it through a fabric of buses and bus matrices, augmented by quite a few dedicated inter-module signals (e.g. interrupts signaling). This allows to split clocks to several domains, interlocked or mutually independent; adding buffers to allow continued execution after writing to slow peripherals; accomodating slow memories and peripherals through waitstates signaling; multiplexing access to memories and peripherals from several busmasters (processor(s), DMA units).

This is exactly the way how the SoC were created. It's all that "good stuff" which allowed to generate complex circuitry from discrete processors/memories/peripheral chips, on a board, and it continues to allow integrating very complex circuitry on a single chip, while maintaining high speed of operation. Not only that, but it also simplifies design: as bus interfaces are standardized, individual modules (usually called IP for "Intellectual Property") can be purchased from various vendors and "slapped together" relatively easily.

The downside of this approach is, that timing becomes a complex issue, usually beyond the point where it could be described in simple terms. One of the practical consequences is, that programmers accustomed to write bit-banged protocols timed by instruction cycle counting, or to calculate performance of key routines in instruction cycles, in practice can't do that anymore. Even the seemingly simple GPIO toggling won't necessarily be directly related to the speed, at which the mcu operates.

As an additional problem, timing of some of those "extra" internal signals, which are not tied tightly to the bus structure, may cause surprising effects. Some of the better known are the "interrupt clear arrives late and causes the ISR to reenter for no reason" and "clock enable arrives later than a read/write to the same peripheral, which then fails". These are usually brought up when one tries to write tight and fast code, and are usually covered up by the usual bloaty vendor-supplied libraries.

As the raw clock frequency of the processing core increases, so does complexity discussed here, and more. High speeds are achieved thanks to parallelism, high level of pipelining, out of order execution, speculative execution, heavy involvement of caches and buffers and various strategies how they are used. Also the number of interconnecting elements such as bus matrices and inter-bus/inter-clock-domain interfaces increases, rarely being properly documented; concerns are being adressed by extensive handwaving and pointing to benchmarks which barely ever truly represent real-world applications. Users are often genuinely surprised to find out that while the raw clock increased by almost 2 orders of magnitude compared to the 8-bitters, the "worst-case reaction" or "control granularity" remained almost the same, with significantly increased uncertainty in timing. As these issues are not something discussed widely and, understandably, manufacturers are not keen to put this up front in their marketing nor engineering materials, this leads to much frustration among users these days.

Consequences:

- Interrupt seemingly called without reason (if its source is cleared late)
- Due to time it takes until interrupt source change propagates into NVIC, if NVIC pending bit cleared outside an ISR, interrupt reappears, if cleared too soon after removing the interrupt source.
- Peripherals keep running, while processor is stopped by debugger.
- After enabling clock in RCC, delay may be needed before accessing some peripherals .
- In 'F4, after enabling backup domain access in PWR, wait before accessing RCC_BDCR or BKPSRAM.
- Processor and debugger can hang, when attempting reading from undefined register of a carelessly implemented peripheral (OTG USB).
- ETH does not work with certain APB2 clock settings, as MII/RMII switch in SYSCFG (clocked from APB2) may not propagate into ETH/MAC (clocked from system clock) before software initiates ETH/MAC reset.
- Complex bus systems may take some signals surprisingly long to pass from busmaster (processor,

DMA) to intended peripheral.
- Combining IP components purchased from various sources leads to inconsistent numbering, registers/peripherls naming and overall documentation style.

Created: 7.Nov 2020
Updated 19.August 2023

26.Don't use `printf()` in interrupts.

In most programs interrupt service routines (ISR) are assumed to run only a very short piece of code, and exit early, as they prevent other parts of code to run.

So, for example in interrupts from communication peripherals such as I2C or UART, the reason for interrupt is to be established; if error, it has to be cleared or dealt with in proper way, perhaps setting a flag or error/status variable for the main program to be handled later; if transmit buffer empty, the software FIFO/buffer has to be checked, if nonempty, data have to be picked from the software FIFO/buffer and transmitted, otherwise transmitter has to be disabled; if receiving buffer nonempty, the received byte should be read out and stored into the software FIFO/buffer, for "main" program to deal with these data later.

This all should not take more than a couple dozens of instructions/machine cycles. In STM32 running at tens of MHz, this means, execution of a typical ISR should last in the order of ~10µs or less.

Some novices, especially having been trained on PC rather than microcontrollers, tend to insert debugging printouts (usually using `printf()` or its variants, often encouraged by the "semihosting" feature offered by IDEs) into various parts of code. This is usually harmless in "main" part of code; however, in ISR, it's often problematic. The vast majority of `printf()` implementations are based on UART, and are blocking, i.e. they wait until the "printed" characters are transmitted.

Transmitting one character at 115200 Baud takes around 86µs, so transmitting a typical message may easily make the ISR last longer by two orders of magnitude than normally. As a result, ISR may fail to keep up with any external or internal stimulus; in extreme cases it may take up all the processing time, preventing "main" to run normally.

Yes, there may be non-blocking implementations of printf() and in some applications lengthy ISR are not problematic - but these are rare. It's better to stick to the safe method - short ISRs, with no `printf()`.

There may be one more reason not to use printf() in interrupts: if printf() is used both in "main" (non-interrupt context) and interrupt, it may simply fail/crash, as printf() may be non-reentrant.

Related:

- Timer appears to run slower then is set

Created: 14.Nov 2020
Reentrancy note added: 19.Aug 2022

27.In 'F4, after enabling backup domain access in PWR, wait before accessing RCC_BDCR or BKPSRAM

In STM32F407, to be able to access (read or write) resources in the so-called Backup Domain (i.e. which are powered from the VBAT power pin, when VDD is off), a as a safety measure, PWR_CR.DBP bit has to be set first.

PWR module is on the APB1 bus, which is the "slow peripheral" bus, clocked at max. 42MHz. This means, that in the usual setup, where AHB is clocked at its maximum 168MHz, APB bus clock has to be prescaled by at least 4.

As a consequence, those Backup Domain resources, which are on the AHB bus - i.e. bits in RCC_BDCR register and the BKPSRAM - may be unaccessible until the write to PWR_CR.DBP actually reached the PWR_CR register. Due to buffering on the AHB/APB bridge, the processor may attempt to access those resources sooner, and this access will fail quietly (read will probably return 0, write will be ignored).

The remedy is relatively easy: it's enough to read back the PWR_CR value after write (the read result can be thrown away, there's no need to check if write was successful: it certainly was).

This issue bears outward similarity to the issue described in "Delay after an RCC peripheral clock enabling" erratum; however, the exact mechanism is somewhat different (the latter involves extra AHB cycles, too). It belongs to the group of issues caused by the SoC nature of the 32-bitters.

The issue is pertinent also to 'F405/'F415/'F417, which are an identical chip to the 'F407. I don't know how is it pertinent to other 'F4 family members, but given the underlying mechanism and overall similarity, I believe it is. It may also be present in other families with similar PWR/Backup Domain structure.

This issue has been discussed on the STM32 Forum back in 2013/2014; ST "discovered" it in 2019 as witnessed by "Possible delay in backup domain protection disabling/enabling after programming the DBP bit" erratum in the 'F4 errata rev.11 dated 05-Dec-2019.

Created: 21.Nov 2020

28.SPI master does not work (as NSS swithes it to slave)

The NSS (Slave Select, the leading N denotes "negated", i.e. active low) signal of SPI causes lots of confusion.

In slave mode, it's quite straigforward - it's an input signal, which gates the MISO input and threestates MISO when high (so that multiple slaves can be simply wired together), and when going from high to low, it resets the internal clock counter, so that it serves as a byte-level framing signal.

In master mode, NSS may have two very different roles: the more usual is, that it is set as an output by setting SPIx_CR2.SSOE, when it is supposed to control the slaves - although in a not very well designed manner.

If SPIx_CR2.SSOE = 0, NSS works as an input; when it is low, the module automatically switches from master to slave. User is notified of this by SPIx_SR.MODF bit being set (while SPIx_CR1.MSTR bit gets cleared simultaneously). The reason for this behaviour is an attempt for a relatively cumbersome multi-master arrangement: the idea is, that multiple masters have their NSS set to Open Collector, pulled up and wired together; then when any of them desires to transmit, it pulls NSS down, which turns all others to slaves. However, there are no clear guidelines in RM (nor is there an appnote dealing with this), how exactly to proceed after this, and perhaps more importantly, there's no mechanism to prevent two masters attempting to go to "active transmitting" simultaneously; so, at the end of the day, this is not a very useful feature.

But it is already there, and it may be confusing for those, who don't want to use the NSS pin in master mode (given it's not very useful either, they want to drive the select pin of slave(s) "manually", using any pin in GPIO Out mode) - if they leave SPIx_CR2.SSOE = 0, SPI very likely won't work. The reason is, that if there is no pin configured in GPIO as AF for the SPIx_NSS, the internal NSS input node is set to 0[1]. And this, as explained above, will switch SPI surprisingly to Slave.

The remedy is easy, set SPIx_CR2.SSOE = 1 even if the actual SPIx_NSS pin is not intended to be used as output, and no pin is set as AF for SPIx_NSS in GPIO.

Another option is to set both SPIx_CR1.SSM = 1 and SPIx_CR1.SSI = 1 -- the former will replace the internal NSS input signal by the value of SPIx_CR1.SSI, and the latter being 1 prevents the switching to slave mode.

---

1. This is one of the features which have never been acknowledged by ST - AFs which are not assigned to any pin in GPIO, are input as 0.

Created: 28.Nov 2020

29.Processor (and debugger) hangs while debugging OTG USB Host

While debugging OTG USB host on 'F407, debugger hung, ST-Link ceasing to communicate (and blink the two-color LED). Simultaneously, the processor hung, too, as the "heartbeat" LED toggled in main loop ceased to blink.

This is the sort of hang caused by the SoC nature of the chip - if one of the peripheral modules fail to correctly end a read or write transaction, and hold WAIT infinitely, processor will wait infinitely too. As the debug module is part of the processor and uses the processor's "connection to outer world" (i.e. it's bus system), when it's the debugger which attempted read/write to such an offending peripheral, both the debugger and processor will hang waiting infinitely.

Upon investigation it turned out, that hang occured due to read to one of the USB OTG modules at offset 0x51C, but only when the device was connected.

This offset belongs to a group of registers related to Host Channels. While this particular offset is not documented by ST and in the headers is marked as Reserved, it is a register which is implemented in only those versions of the Synopsys OTG module which have enabled the Scatter-Gather-DMA feature (at implementation time, i.e. when the chip is designed). This feature is not implemented in the ST's incarnations of this module, so the register is rightly marked as reserved. However, it would be prudent to implement the register so, that reading it won't cause hang. Reading the "neighbouring" offset 0x518 (which is Reserved, too) won't result in hang. The Synopsys module is not famous for being well designed, to put it mildly.

Why did the debugger read a reserved position anyway? Using gdb, when commanded to read a struct, it reads all of it, as it does not know, which parts of it are reserved. Here's the definition of said struct in the respective device header:

```
typedef struct {
  __IO uint32_t HCCHAR;           /*!< Host Channel Characteristics Register    500h */
  __IO uint32_t HCSPLT;           /*!< Host Channel Split Control Register      504h */
  __IO uint32_t HCINT;            /*!< Host Channel Interrupt Register          508h */
  __IO uint32_t HCINTMSK;         /*!< Host Channel Interrupt Mask Register     50Ch */
  __IO uint32_t HCTSIZ;           /*!< Host Channel Transfer Size Register      510h */
  __IO uint32_t HCDMA;            /*!< Host Channel DMA Address Register        514h */
  uint32_t Reserved[2];           /*!< Reserved                                      */
} USB_OTG_HostChannelTypeDef;
```

The offending register is here .Reserved[2].

It would appear, that it's enough to remove the Reserved[2] field from the struct. In ST's "libraries" code, as they use macros to access these registers, calculating the offsets ad-hoc, it might work; however, I've written a proper struct encompassing all OTG registers; that uses this particular struct in an array (as there are several Host Channels), so removing items from struct would make it's length different, rendering such array unusable. The only solution for me thus is to exercise care and avoid reading the whole struct, read only its individual named items.

"Clicky" debuggers built into IDEs may not suffer from this problem, as they for registers layout don't use information propagated from headers through .elf/.dwarf.

Created: 5.Dec 2020

30.Only DMA2 can be used to transfer to/from GPIO in 'F4

In the 'F4 family (as well as 'F2 and 'F7), two DMA units are implemented. Both have two ports - a "Peripheral port" and a "Memory port" - and a FIFO. The internal units - called streams - can access simultaneously the two ports - while one stream accesses the Peripheral port, other can access the Memory port. Besides the respective RMs, this DMA unit is further described in AN4031.

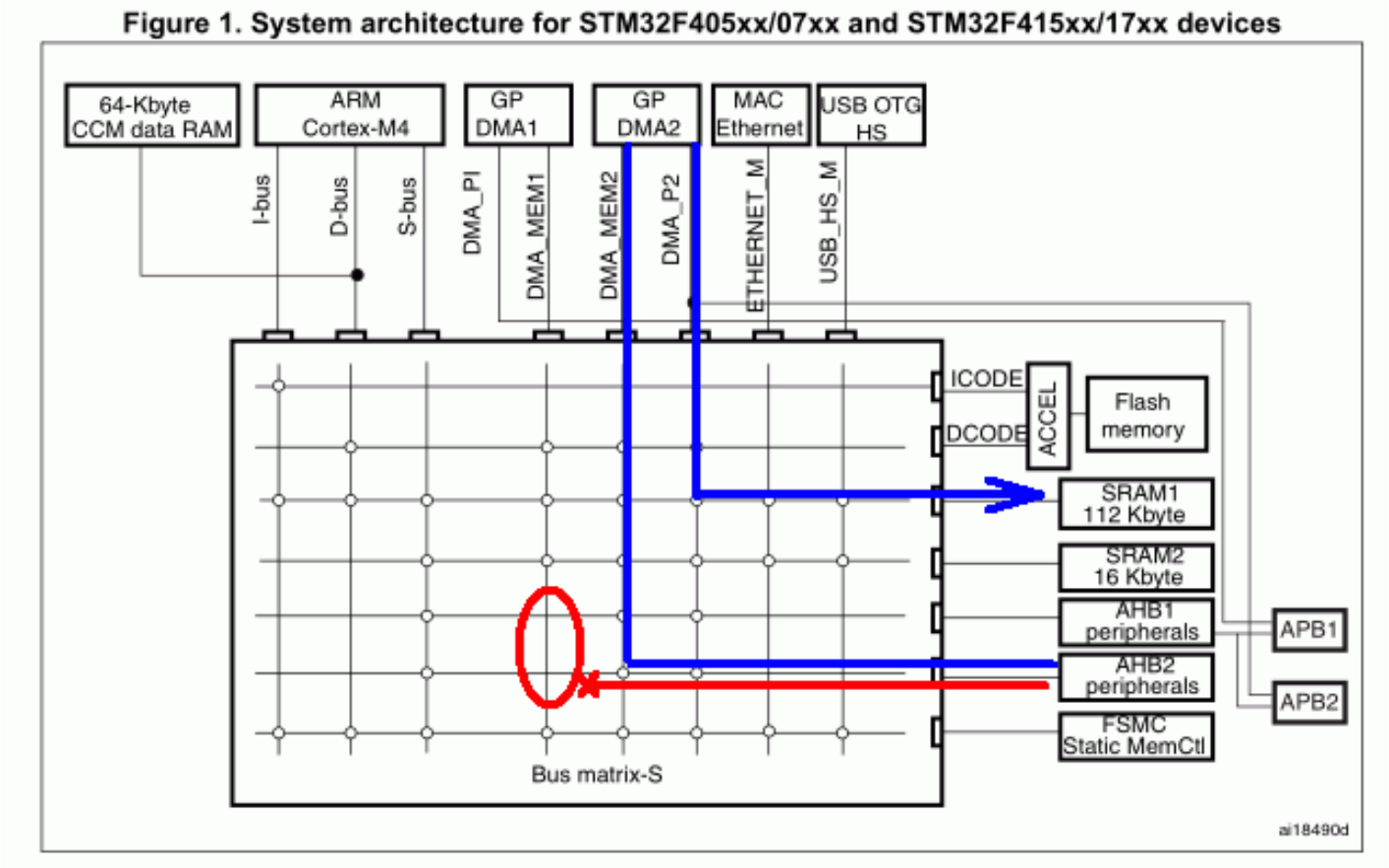
However, even if the two DMAs are almost identical in their working and have identical registers, their connection within the system is not the same: the Peripheral port of DMA1 has no connection into the busmatrix, but is connected directly to the APB1 bus through a special dual-AHB-to-APB bridge, arbitrating between accesses from the busmatrix (through AHB1) and DMA1.

DMA2 does have this direct connection from its Peripheral port to APB2, but at the same time it has also a connection from Peripheral port to the bus-matrix; so if the peripheral address of given stream is outside of the APB2 address range but inside of any other memory or peripheral's range, that access will be steared towards the busmatrix, rather than to APB2.

The practical consequence of this fact is, that only DMA2 can perform Memory-to-Memory transfers. Those are transfers, which not only move data from one memory to another (or any other busmatrix slave, including APB1-through-AHB1, and AHB2); but they are also not triggered (requested) by any peripheral, once a stream set to M2M transfer is enabled, it will move data autonomously at a pace given by arbitration within the DMA unit itself and at its ports, until completion.

However, DMA2 allows also another practical transfers: it allows to move data from/to memory to/from GPIO. Here, normally transfers are required to be performed at some regular pace, rather than at the highest possible (and often irregular, given conflicts on buses from other busmasters, e.g. the processor) rate. As triggers (requests) are hardwired from peripherals to DMAs ('F2/'F4/'F7 predate the era of DMAMUX) and the only practically usable request source are timers, in 'F2/'F4, only TIM1 and TIM8 can be used to trigger this sort of transfers. On the other hand, using capture as source of DMA trigger, GPIO-to/from-memory transfers can be accomplished upon external signals, too. Such process is described in AN4666.

There's no way to use DMA1 for this sort of transfer, as it not only does not have connection from its Peripheral port to the busmatrix, but also the connection it has from Memory port to bus matrix does not connect to AHB2, where GPIO ports are located.



Formally, either DMA2 port can be used for the GPIO (and the remaining one for memory). As the trigger(request) always triggers the Peripheral-port-side transfer, it is logical to use Peripheral port for GPIO, where timing usually matters the most.

Of course, this sort of transfer is not constrained to GPIO. An interesting usage is a timer-paced USART or SPI Tx, where the timer (potentially together with other timers, through their internal interconnections) can generate automatic framing/latching signals.

Created: 12.Dec 2020

31.Timer - if compare is enabled, setting CCRx > ARR still causes interrupts/DMA

When timer runs freely, if a channel is set as Output Compare, upon each Compare event - i.e. when TIM_CNT matches TIM_CCRx - the respective CCxIF flag in TIM_DIER is set.

This happens regardless of the setting of respective CCxE enable bit in TIM_CCER - this bit, for a channel set to Output Compare, only enables/disables generation of output signal.

So, if one would like to suppress generation of TIM_DIER.CCxIF for any reason, it would seem to be logical to set TIM_CCRx to a value never attainable by TIM_CNT, i.e. to any value larger than TIM_ARR.

Surprisingly, this does not stop TIM_DIER.CCxIF being generated. This **is** documented in the RM, at the description of TIM_DIER.CC1IF bit:

> *When the contents of TIMx_CCR1 are greater than the contents of TIMx_ARR, the CC1IF bit goes high on the counter overflow (in upcounting and up/down-counting modes) or underflow (in downcounting mode).*

nonetheless, this remark is easy to miss, and the behaviour is not expected.

Setting TIM_DIER.CCxIF results not only in interrupts when enabled, but also in DMA transfers if configured to be triggered by that channel. Also, if the channel in question is CH1, and TRGO is set by TIM_CR2.MMS=0b011 to CC1IF (quite incorrectly called by RM *Compare Pulse*, as it works perfectly also when CH1 is set to capture), TRGO pulses are generated thus can be used as TRGI in coupled timers, or triggers to ADCs when set so.

Note, that TIM channels are set to Output Compare by default, so these signals start to be generated as soon as the timer is set to run freely and enabled. The only way to suppress them is to set given channel to Input Capture and not enable it in CCER.

Created: 19.Dec 2020

32.SPI generates too many clocks in Rx-Only mode, and in Bidirectional mode when turned to Rx

The SPI module in STM32 is relatively complex and features several modes. When set as Master, and set either in Rx-Only mode (SPI_CR1.RXONLY = 1), or in Bidirectional mode turned to Rx (SPI_CR1.BIDIMODE = 1 and SPI_CR1.BIDIOE = 0), it starts to generate clocks on SCK as soon as one of these modes is enabled (and SPI is enabled, SPI_CR1.SPE = 1), and stops generating them only after the given mode is disabled and the current frame is finished.

Naively written code (sometimes seen in "libraries" like Cube) simply turn on these modes and wait until a given number of frames is received, then turn them off. This usually leads to generating more clocks than expected, as the given mode has to be switched off *before* the last frame is received, more precisely - and I quote here from the RM - *between the sampling time of its first bit and before its last bit transfer starts*.

This of course means precise timing control, which is not always easily attainable, especially at higher SPI baudrates. Depending on the particular application, techniques like highest-priority timer interrupts or timer-triggered DMA-to-SPI_CR1 transfers might be needed. For receiving a single frame, a simple trick can be used, where the given mode is enabled and then after a few NOP (or NOP-acting) instructions immediately disabled again[1]; interrupts need to be disabled for this sequence too.

In many applications, extra clocks don't really matter, but there are slaves where this might be unwanted. Sometimes some framing signal has to be toggled just after transmission, without any extra clocks.

In case of Rx-Only, there's a simple workaround, where the SPI is not used in the Rx-Only mode at all, but using it in "regular" fullduplex mode with the MOSI pin simply not set to any pin in the GPIO AF matrix.

Workaround for Bidirectional mode (i.e. halfduplex, where a single data line is alternated for both directions) is not that simple, and comes at the cost of one extra pin - for this, the SPI would be set to Slave mode, and clocks would be generated on that extra pin, connected externally to SCK, by either a timer/PWM (e.g. using repetition mode, or with two timers in master-slave arrangement, one determining the bitrate on SCK and the other gating it to control the number of pulses); or simply bit-banging the clocks "manually" on pin set as GPIO Out.

---

1. In Cortex-M7, NOPs may be "gulped" by the pipeline, i.e. they may not contribute to delays. A common workaround is to use DSB. An example of this technique (although without disabling interrupts, probably because there are no interrupts active in that code) can be seen in the example code for reading the LSM303C magnetometer on 32L476GDISCOVERY, which has bidirectional SPI connection.

Created: 26.Dec 2020

33.Smaller packages require port remapping for certain functions

Most of STM32 come in relatively big packages. Not that there are physically big - in most families, there are those tiny Chip-Scale (WLCSP) packages available; however, these are not very manufacturing-friendly and fit only for mass production. For applications, where pin count is not an issue, packages like QFP-24 are TSSOP-20 are more in demand.

While ST does produce these for some families, they contain the same physical chip as their 64+ pin counterparts, except that most of the pins are simply not brought out to the package. The reason is cost - in modern semiconductor production, a set of masks easily goes to $1M+ and it may take several respins to get a particular chip right; adding to this the inevitable development and testing costs leaves the manufacturer with significant NRE per design. Packaging them to different pin count packages to widen the portfolio, thus perfectly makes sense.

One of the drawbacks of this approach is, that sometimes it's necessary to make tradeoffs. Omitted pins means also unusable peripherals, given the limited GPIO AF matrix.

In some cases, ST solved this by a feature called remapping. Particularly, in STM32F042, USB (and CAN) pins are brought out to PA11/PA12, which are not available on the TSSOP20 and QFN28 (UFQFPN28) packages. However, after setting SYSCFG_CFGR1.PA11_PA12_RMP, the PA9/PA10 pins are replaced by PA11/PA12, making USB or CAN available again.

Why this was not solved by mapping in the "regular" GPIO matrix, I don't know.

Created: 2.Jan 2021

STM32 gotchas

34.ADC in 'L47x/48x has an extra switch in GPIOx_ASCR

This rather surprising "feature" is best "explained"by quoting from ST's own presentation:

New GPIOx_ASCR control register has been introduced on STM32L47x/48x devices to control analog interconnection between an I/O pin and ADC input. This register has to be configured before the ADC conversions are started to correctly bring the signal to the ADC input. This register is removed on other Categories of STM32L4 devices and the analog interconnection will be enabled automatically when an ADC channel is selected.

Don't ask me, why.

Created: 9.Jan 2021

35.'F1 don't have the GPIO AF matrix

The STM32F1xx was the first STM32 family, conceived more than a decade ago. It immediately hit a sweet spot on the mcu market with the combination of features, peripherals, became very successful and is still popular today despite its age and shorcomings. Of course, unavoidably, errors were made, as witnessed by the relatively lengthy errata; but that's nothing extraodrinary given the immense complexity of the design.

One of the early decisions was to have a relatively fixed mapping of peripherals' inputs/outputs to physical pins. While developers would welcome the flexibility of a full matrix, this is a relatively expensive choice, given the silicon area such matrix and the interconnections web[1] would occupy is not insignificant.

As it would be unreasonable to produce a separate chip for each package variant of one model, the same chip is packaged to different packages, with simply not bringing out some of the pins[2]. However, there are more peripherals than would fit onto pins on the least-pin-count packages, so there are several peripheral IO mapped onto the same physical pin in some cases. Normally, user should not enable (by setting its clock in RCC) more than one peripheral at a time, where such sharing occurs.

To increase flexibility, ST allows to "remap" some of the peripherals to pins available in the higher-pin-count packages. From the programmer's perspective, a separate peripheral, AFIO, serves this purpose; before writing to its registers its own clock has to be enabled in RCC.

In all newer STM32 models, the arrangement is different and there are several peripherals' inputs/outputs assigned to most of the pins. Once the developer sets a given pin to Alternative Function (AF) mode, he/she then has to choose, which particular peripheral's IO to connect to. This mapping is presented in the Datasheet, in a simple sparse matrix form. While this is still not a full matrix at all[3] and there are shortcomings of this method[4], too; the AF/pin assignment in those families is generally a simpler and cleaner process.

This difference might come as an unpleasant surprise for developers who are already familiar with other STM32 families, and then are asked to work with the 'F1.

---

1. Extensive interconnections might also require extra metallization layers, thus masks, all resulting in increase in both non-recurring and per-device cost.

2. In STM32, the individual ports are deliberately designed so, that the "higher" ports are available only in the higher pin count packages. For example, STM32F103x8/xB in 48-pin package (i.e. 'F103C8/CB) has only GPIOA and GPIOB available; in 64-pin ('F103R8/RB) it has also GPIOC; and in 100-pin ('F103V8/VB) GPIOD and GPIOE become available. There are exemptions to this rule - the oscillator pins (for both HSE and LSE) are on fixed port pins, so those are available in all packages where the oscillator(s) is(are) available.

3. Only the most common peripherals such as timers and UART/I2C/SPI have pins mapped to multiple pins, the complex peripherals have usually a fixed mapping. This has to be borne in mind when designing in the STM32 into a circuit.

4. There are several, but let's pick the one which relates to 'F1: In 'F1 in some cases, two peripherals' inputs are/can be mapped onto one pin, and this may be useful. For example, where both a TIM channel and UART Rx can be mapped onto one pin, the timer with appropriate setting can automatically measure inactivity and throw an interrupt upon long inactivity (this is more flexible than the usual IDLE implementation in UART).

Created: 16.Jan 2021

36.Peripherals don't have unique AF in GPIO matrix

In STM32 - except STM32F1xx - peripheral IOs are assigned to physical pins through a simple setup process in GPIO: Once the developer sets a given pin to Alternative Function (AF) mode in respective GPIOx_MODER register bits, he/she then has to choose, which particular peripheral's IO to connect to, by setting respective bits in the GPIOx_AFR[x] register pair (sometimes marked also as AFRL/AFRH).

The value to be written into GPIOx_AFR[] is given in the Datasheet of each particular STM32 model, as a table titled *STM32xxxx alternate function mapping*. It is a sparse matrix, describing, which AF number corresponds to required peripheral, for each physical pin. The AF number is given by the column in which the given peripheral IO is written; column headers describe the set of peripherals to which the given AF is assigned.

This scheme has been introduced in the 'L1 model, where the AF number uniquely determines the peripheral. For example, wherever there is a TIM2 IO assigned to a pin, in these families, it's invariably AF1; all USART2 pins are AF7, etc.

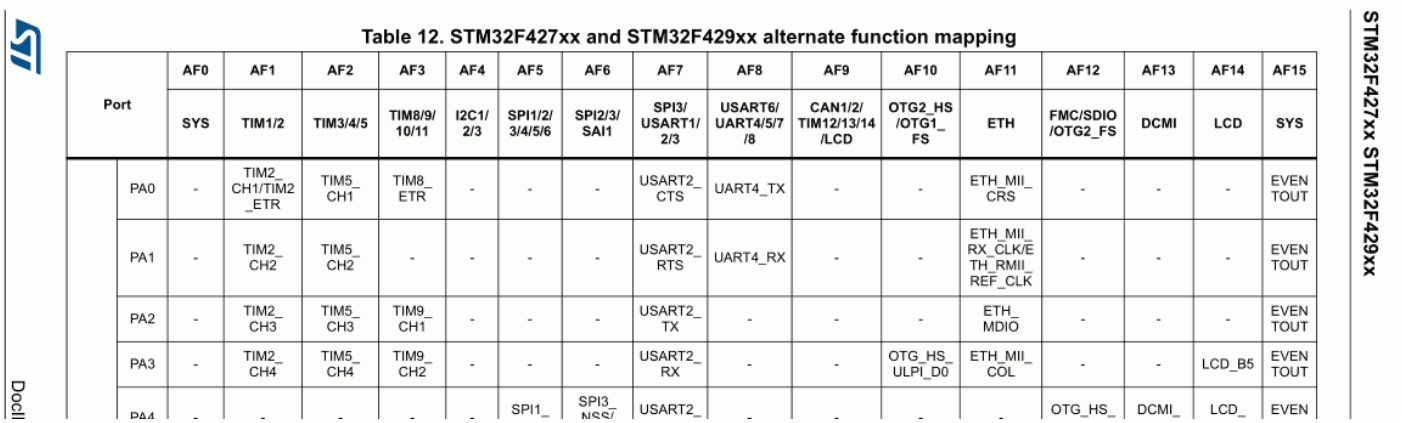


Table 12. STM32F427xx and STM32F429xx alternate function mapping

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/ 10/11	I2C1/ 2/3	SPI1/2/ 3/4/5/6	SPI2/3/ SAI1	SPI3/ USART1/ 2/3	USART6/ UART4/5/7 /8	CAN1/2/ TIM12/13/14 /LCD	OTG2_HS /OTG1_ FS	ETH	FMC/SDIO /OTG2_FS	DCMI	LCD	SYS
	PA0	-	TIM2_ CH1/TIM2 _ETR	TIM5_ CH1	TIM8_ ETR	-	-	-	USART2_ CTS	UART4_TX	-	-	ETH_MII_ CRS	-	-	-	EVEN TOUT
	PA1	-	TIM2_ CH2	TIM5_ CH2	-	-	-	-	USART2_ RTS	UART4_RX	-	-	ETH_MII_ RX_CLK/E TH_RMII_ REF_CLK	-	-	-	EVEN TOUT
	PA2	-	TIM2_ CH3	TIM5_ CH3	TIM9_ CH1	-	-	-	USART2_ TX	-	-	-	ETH_ MDIO	-	-	-	EVEN TOUT
	PA3	-	TIM2_ CH4	TIM5_ CH4	TIM9_ CH2	-	-	-	USART2_ RX	-	-	OTG_HS_ ULPI_D0	ETH_MII_ COL	-	-	LCD_B5	EVEN TOUT
	PA4	-	-	-	-	-	SPI1_	SPI3_ NSS/	USART2_	-	-	-	-	OTG_HS_	DCMI_	LCD_	EVEN

STM32F427xx STM32F429xx

This user-friendly scheme (even with the same assignment of AF numbers to individual peripherals) has been brought over to 'F2/'F4/'F7 models, with a few exceptions mainly for SPI/I2S.

Unfortunately, that was it, and in the rest of STM32 families the AF assignment to pins is more less random. Thus users who don't rely on tools generating automatically code (e.g. from CubeMX) have to check in the DS for each individual pin, which AF number corresponds to the required functionality.

Created: 23.Jan 2021

37.GPIO pin input state can still be read if it's set to Out or AF - and EXTI works, too

As long as a pin is not set as Analog in GPIO_MODER, it is permanently connected to the input Schmitt trigger, and from that to GPIO_IDR, as well as to the EXTI input multiplexer.

That means, that a pin's input state can be read, even if it is set as Out or AF[1]. This is useful especially when the output is set as Open Drain - in that case, level set in ODR for that pin does not necessarily match the level read in into IDR for the same pin.

Similarly, EXTI can be used, even if pin is set as Out or AF. This allows for example to have an external interrupt together with pin acting as the dedicated alternate-function input for some module. One useful application for this is the NSS pin in SPI set as slave, used with DMA - the AF acts as framing/output enable for the hardware, and the rising edge could trigger an interrupt which in software starts the processing of received data.

These facts come as a surprise mainly to the users of "libraries" such as Cube or SPL,as they tend to avoid reading the RM, and also in "libraries" the GPIO initialization routines usually bind EXTI functionality exclusively to pins being set as In.

1.In fact, a pin being set in GPIO_MODER as In means just that the output drivers are permanently disabled.

Created: 30.Jan 2021

38.Analog switch voltage booster is needed for analog features to work properly at lower VDD on certain STM32

While many frown upon the performance of analog blocks in the STM32 such as ADC, DAC, opamps and comparators, mainly based on comparison with their standalone counterparts; it's honestly amazing how well these blocks perform when considering the technology upon which they are built, which is aimed primarily at high-speed digital circuits. Consequently, there are not only fundamental limitations to the expectable performance, but there are also quirks which have to be taken into mind, and also extra requirements when operating these analog blocks. While these requirements are clearly set in the documentation, they are often overlooked - given the "I've just whipped up some libraries/clicked in configurator" and "who reads the documentation anyway" attitude we see so often today - ultimately resulting in less-than-optimal operation and unnecessary disappointment.

One of such details to be observed is the usage of an internal analog switch booster at lower VDDs. The STM32 generally work at a rather wide supply voltage range, roughly from 3.3V down to 1.8V. MOSFET transistors are not fully opened at lower gate voltages. While the associated higher channel resistance generally does not matter much for digital operation, it is detrimental when used as analog switches, e.g. in the ADC input multiplexer, which switches the individual channels (as assigned to physical pins) to the ADC "core".

To decrease MOSFET channel resistance, some STM32 families feature an internal higher-voltage source (boost), to drive these MOSFETs' gates high enough at lower VDD. User has to switch this boost on, if the analog switches are to be operated at lower VDDs. The exact VDD is given by the individual datasheets, for example the 'L476 specifies ADC parameters at VDDA<2.4V with boost ON (and above 2.4V with boost OFF).

Of course, using this boost is not "free" - it involves an increased current consumption. For example, DS for mentioned 'L476 specifies the boost current consumption as 0.5mA at VDD between 2.0V and 2.7V.

Also, it takes some time after having the boost enabled in the given register bit, until the high voltage is actually available; again, this time is specified in the respective DS.

The analog switch booster is controlled by a peripheral register bit, given in the respective RM; for example in 'L4, 'G0 and 'G4 it's SYSCFG_CFGR1.BOOSTEN; in H7 it's SYSCFG_PMCR.BOOSTE.

Analog switch booster may also influence V_{DDA}.

Created: 6.Feb 2021

39.Software or watchdog reset does not work, if NRST is pulled-up hard externally

The NRST pin on STM32 is a bidirectional pin, controlled by an internal open-drain transistor, and also being pulled up by an internal resistor.

So, it has two functions: as input, when pulled down externally (hard enough to override the internal pullup resistor), it generates a system reset; this is what most of users do expect from this pin.

As output, when an internal reset source gets active, the internal open-drain transistor pulls down the NRST pin for a few µs. The system reset is still generated by the *input* circuitry of NRST, so the voltage on pin has to physically go low enough for the system reset to occur. Examples of internal reset sources are the Software reset (result of setting SCB_AIRCRE.SYSRESETREQ, to be used through the CMSIS intrinsic NVIC_SystemReset()) or reset from internal watchdogs (IWDG/WWDG).

Some users connect a reset circuit with push-pull output (such as the venerable industry-standard xxx809) to NRST pin. As the internal reset generator cannot pull down through the open-drain transistor such external load, internal reset sources such as Software reset or the Watchdog resets, simply don't work properly. They won't result in system reset; however, the reset source in RCC_CSR or equivalent register gets registered.

This all is well documented in every STM32 RM, *Power reset* subchapter of *RCC* chapter. The values of internal pullup, required external reset duration and the reset output duration, are documented in the respective DS.

Created: 13.Feb 2021

40.First TIM period is too short, prescaler does not work - PSC is preloaded; and so is RCR

Some timer registers are preloaded, which means, that writing to the given register does not write the actual "working" register, but an "intermediate" register[1]; and transfer value from "intermediate" to "working" register at the Update event. Update event is typically generated at counter overflow/underflow; but it has several other possible sources, some of which can be disabled - for the exhaustive details, see the TIM chapter in RM.

The reason for preloading registers is twofold - changing some of the registers "on the run" may lead to confusing results (e.g. changing ARR to a value below current CNT value would result in CNT to count up all the way to its upper limit, 0xFFFF or 0xFFFF'FFFF depending on counter width). The other reason is, that some registers are best to be changed at once at one given point - e.g. it's usually required to change PWM period and duty cycle at once.

Preloading of some of the registers is optional, e.g. ARR is preloaded only if CR1.ARPE=1, CCRx is preloaded only if the respective CCMRx.OCxPE=1[2]. On the other hand, the prescaler (PSC) and - in timers where it's available - the repetition-count register (RCR) are preloaded unconditionally. This means, that after loading them, their new value won't get "accepted" until an Update event occurs. So, as a practical consequence, unless this "acceptance" is "forced", the first timer period after setting PSC is run with the old PSC setting As PSC after reset is by default 0, i.e. no prescaling, the first timer period appears to be too short for those unaware of the PSC preload.

One way to "force" "acceptance" of new values into PSC and/or RCR is to generate Update "artificially", by setting EGR.UG. Note, that this - as with any Update - results also in setting SR.UIF, so if there's any related interrupt or DMA enabled by DIER.UIE or DIER.UDE, the interrupt gets activated and DMA performs the associated transfer.

---

1. ST uses the term "shadow" register for what is called here "working" register. As probably many users would associate "shadow" with "intermediate" rather than "working", "shadow" is not used here.
2. In some timers (advanced, with complementary outputs), CCER.CCxE, CCER.CCxNE and CCMRx.OCxM bits can be preloaded, but they are not loaded to their "working" counterparts upon Update event, but upon Commutation/Trigger event. This is aimed at changing the output mode of all channels at once, upon a one-off stimulus/signal, mainly to be used in motor-control applications.

Created: 20.Feb 2021

41.One TIM period is too long; PWM pulse is shorter/longer than expected - use preload when needed

While the preloaded character of some TIM registers may result in surprising behaviour, on the other hand, not using preload where it should be used, may result in surprises, too.

In particular, if ARR is not preloaded and is modified (decreased) so that its new value is below CNT (in upcounting mode), CNT will count all the way up to 0xFFFF or 0xFFFF'FFFF, depending on the width of given timer, resulting in a surprisingly long period.

Similarly, if CCRx is not preloaded and is changed while the counter is running, it may happen that the output compare result changes at the moment of the new CCRx value write, resulting in change of output level at an unexpected moment, yielding a nonstandard length of that one particular pulse.

While this effect is easy to understand, it must have confused some of the users so, that ST put this remark to the description of CCMRx.OCxPE bit (which enables the respective CCRx preload):

The PWM mode can be used without validating the preload register [sic!] *only in one-pulse mode (OPM bit set in TIMx_CR1 register). Else the behavior is not guaranteed.*

(ST by "validating the preload register" probably meant "using" or "enabling" the preload register. ST manuals are written by non-native speakers of English.) But this of course is nonsense. Writing to non-preloaded CCRx amidst a one-pulse period results in the same confusing behaviour as writing it in normal non-one-pulse mode; i.e. one-pulse mode is no guarantee for different behaviour. And, vice versa, if in non-one-pulse mode, user knowingly writes non-preloaded CCRx with a value which he is sure that at the given moment/value of CNT does not change the outcome of the compare; or if user simply does not care that there will be one non-standard pulse - that write has no other adverse consequences.

Created: 27.Feb 2021

42.First TIM Update ("PeriodElapsed") interrupt fires immediately after configuration, when using Cube/SPL

As the timer register prescaler is unconditionally preloaded, libraries such as Cube/HAL, Cube/LL or SPL, when initializing a timer, they invariably set the EGR.UG bit to generate an Update event which would force loading the preloaded values of PSC and RCR into the respective "working" registers. As a consequence, after performing such initialization and enabling Update interrupt (sometimes renamed to "PeriodElapsed" or similarly), this interrupt "fires" immediately, sometimes puzzling the users.

The remedy is easy - after the timer initialization and before enabling the respective interrupt in NVIC, TIM_SR.UIF has to be cleared.

Created: 6.Mar 2021

43.No bit-banding in Cortex-M0/M0+ - but not in Cortex-M7 either

When ARM and its mcu-manufacturing partners started to introduce the Cortex-M family some 15 years ago, their first product was aimed at the mid-range, with the Cortex-M3 core. However, they wanted to capture not only the nascent 32-bit mcu market, but they also wanted to pick up parts of the existing 16-bit and the - then still dominant - 8-bit mcu market. As part of this effort, they introduced so called bit-banding into the new processor's design.

This feature was obviously aimed at users of the venerable Intel-originated 8051, and its numerous clones and variants produced by many companies in huge numbers. As in 8051 part of the memory can be accessed both byte-wise and bit-wise; in a similar move, in Cortex-M3, certain portion of the memory map[1] can be accessed again in a different area of the memory map in such a way, that every bit from words in the first area are aliased to the first bit of an otherwise empty word in the second area. Thus, reading words from the second area returns 0x0000'0000 or 0x0000'0001 depending on state of given bit; conversely, writing 0x0000'0000 or 0x0000'0001 will clear or set a single bit in the respective word in the first area[2].

This feature is called bit-banding[3].

Contrary to the 8051, where the bit-addressable area is an integral part of the core design, and individual bits are accessed using dedicated instructions; in Cortex-M3, the designers employed an ingenious trick: bit-banding is performed by an extension unit on the port, through which the processor accesses the rest of the mcu (usually via a bus matrix)[4]. This extension detects accesses to the bit-addressable area and intercepts them, while putting the processor to wait. When a read from bit-addressable area is to be performed, the extension reads the corresponding word from the "normal" area, extracts the required bit, and presents it to the processor as the LSB in an otherwise zeroed word. Write to bit-addressable area is more complex: the extension has to read the corresponding word, insert the new bit value into that word, and write the word back to the memory; however, it does not need to waitstate the processor (except if the processor would want to write again while this process lasts). This trick allows to implement bit-banding without any impact to the instruction set or any other portion of the processor or the rest of the system.

Bit-banding was retained also in the Cortex-M4 core[5] which expanded on the Cortex-M3, adding mainly the FPU and several smaller additional features. However, it did not gain much popularity amongst the users. Maybe the reason lies in lukewarm acceptance also by the toolmakers, who typically provided little to no support for bit-banding, not even a relatively simple guideline for placing variables into the dedicated bit-addressable area using compiler pragmas and/or (already existing) linker features. There is also a couple of issues when using bit-banding in the peripherals without exercising due care[6], leading to further alienation of users.

When ARM proceeded to create the stripped-down Cortex-M0 core for smaller, cheaper end of the whole group of mcus, they decided to omit the bit-banding extension. This is somewhat surprising, as it was created primarily in order to attract existing 8-bit mcu users, and the Cortex-M0 was again intended for mcus directly replacing the 8-bitters. Whether the reason was in the previous weak acceptance of bit-banding, or the attempt to create as compact processor as possible, this strategy was adopted also in the successor: Cortex-M0+ does not have bit-banding either.

Implementation of the bit-banding extension in the high-end Cortex-M7 would probably be cumbersome, given caching and conversion of 32-bit processor bus into the 64-bit AXI bus, maybe even impacting performance; so bit-banding was omitted from Cortex-M7 too[7].

The corollary is that while bit-banding is a useful feature to have bit-wide variables (usually of boolean nature), sparing memory while retaining atomicity[8]; they are available only in Cortex-M3/M4. So, if portability of code onto Cortex-M0/M0+ or Cortex-M7 is desired, bit-banding is better to be given up, in favour of spending a whole byte for the single-bit (boolean) variables. With a bit of planning and proper placement of these variables using linker features, seamless transition may be achieved, but the cost of increased memory consumption still has to be considered.

---

1. In fact, two portions - one is intended to provide some bit-aliased memory, the other is intended to provide bit-aliased peripheral area.

2. This process, the exact location of the mentioned areas and the exact mapping together with formulas to calculate respective addresses from one area to the other, can be found in ARM's *Technical Reference Manuals* to the Cortex-M3 and M4. In ST's materials, refer to *Bit-banding* chapters in PM0056 and PM0214 (Programming Manuals to Cortex-M3 and Cortex-M4 for STM32 mcus). Here is a simple bit-band-addresses calculator.

3. Sometimes confused with the somewhat slangish expression used with microcontrollers: bit-bang**g**ing -

this refers to a non-hardware implementation of certain sequenced protocols such as SPI or I2C, where the individual bits (and respective clock transitions) are output to the port pins (or input from them, i.e. "banged out/in") step-by-step, "manually", in software.

4. In particular, bit-banding extension is only on the S-port (System) of the mcu. That is the von-Neumann-ish port (i.e. port through which both instructions can be fetched and data can be read and written) of the processor, through which addresses above 0x2000'0000 are accessed - this is obvious also from addresses of the particular bit-banded areas. The Harvard-ish I-port (Instruction) nor D-port (Data) (for accesses below 0x2000'0000) are not bit-banded.
The fact that bit-banding is an extension on the processor's memory interface also means, that the bit-addressable area is "invisible" (non-existing) to all other bus-masters, so, for example, DMA cannot write or read the "single bit" areas.

5. Note, that bit-banding is an optional feature of Cortex-M3/M4, and implementers may choose to omit it. There are examples of Cortex-M4-based mcus from other manufacturers which don't have bit-banding implemented, but as of 2020, every STM32 based on Cortex-M3/M4 does have bit-banding implemented.

6. The main problem with this scheme in peripherals is, that while from the process/program point of view accessing the bit-area is atomic, from the hardware point of view it may not be so; so registers, which are set/cleared autonomously by hardware may be dangerous to be accessed through the bit-area.
On an NXP Cortex-M3-based LPCxxxx I've experienced also a different gotcha, which might or might not be consequence of bit-banding, but I've experienced it with bit-banding - there were two writes into a GPIO port back-to-back, one using bit-band and the other not. The memory interface merged these two accesses into a single write (plus obviously one read for the bit-banded write), outputting the result of both writes at once, ruining my bit-banged SPI implementation. This was probably due to the fact that NXP decided - not very wisely - to put GPIO into the bit-banded area intended for memories, which in the default setting of MPU is tagged as "Normal", allowing such merging in write. No STM32 has peripherals in this area, and STM32 do have the BSRR registers, performing atomic writes at the GPIO peripheral itself, obviating the need for placing them into bit-banding area at all.

7. Bit-addressable aliases can actually be implemented at various points of the system, not just at the processor-to-system interface; and this then may have less impact on overall performance and processor complexity. An example is the "Bit Manipulation Engine" of certain NXP/ex-Freescale Cortex-Mx-based mcus, where bits can be not only set and cleared, but also some other boolean operations can be performed over them - all this through seemingly innocuous access to an aliasing area.
However, this is still just an insert into the bus structure, making use of the waitstate/handshake nature of data exchange through the mcu buses (read: the processor waits until the whole read-modify-write process in the "insert" is finished). But, while I haven't seen such yet, I can envisage a piece of truly bit-addressable memory, without alias into byte-wise or word-wise collection of the bits, to be used as boolean variables/arrays. While this would spare silicon area at the cost of wasted address space, there is plenty of the latter in the ARMs.

8. The apparent replacement - C bitfields - notoriously suffer from not being atomic, as they form part of a byte/word in which bits cannot be flipped atomically with a single instruction, at least not in standard memory with standard ARM Cortex-M core.

Created: 13.Mar 2021
Link for DMA in footnote added: 28.May 2022

44.Some STM32 models after first programming don't run, and need power-on reset (AN2606 pattern6)

If a brand new STM32F091 is programmed with a program, which has already be proven to work, it is not enough to perform a software reset or even a NRST-"pushbutton" reset; a power-on reset is needed to run the application. What gives?

This is a surprising, but documented behaviour. STM32F091, when "pristine"[1], upon startup detects empty FLASH and goes directly into bootloader. This may be a pleasant feature (as most other STM32 models require some physical action to enter the bootloader, usually pulling up the BOOT0 pin).

However, this is a somewhat more involved procedure, described in RM0091, *Boot configuration/Empty check* subchapter. The hardware **does not check the whole FLASH**: contrary, it checks only the first word in it (at address 0x0800'0000). If this word is 0xFFFF'FFFF, FLASH is considered empty, a flag is set (user-invisible or undocumented in 'F091, documented in some other models[2]); and if this flag is set, bootloader is subsequently entered.

Problem is, that **this flag is not cleared with most sources of reset**. As the "FLASH-empty" check is performed by hardware at the same moment when the FLASH option bytes are read, i.e. at power-on reset, this is what's required to clear the flag. It means, after programming, power has to be removed and then restored again, to run the application for the first time.

Alternatively, option bytes can be reloaded - thus the "FLASH-empty" check performed and flag cleared - by executing OBL_LAUNCH command (i.e. writing 1 into FLASH_CR.OBL_LAUNCH). There may be programming tools which can perform this through the debug (SWD) interface; I am not aware of them.

There are several STM models which "suffer" from this problem, but it's not easy to identify them. While they have **pattern 6** bootloader entry, as documened by **AN2606**, some - as e.g. the 'F042 - compensate for this problem in the bootloader itself, detecting "non-empty" FLASH and launching the application from software. For each suspected model, the respective RM has to be consulted for the minute details.

---

1. Or, it has been erased and power-on reset has been applied. Or, for whatever weird reason, it has the 0x0800'0000 word unprogrammed, i.e. equal to 0xFFFF'FFFF, at the moment of power-on.

2. For example, see FLASH_SR.PEMPTY in RM0394.

According to this article, the full list of devices which have the "empty FLASH" check implemented is:

- STM32L4+
- STM32WB
- STM32G0
- STM32F04x
- STM32F09x
- STM32F070x6
- STM32F030xC
- STM32L011x/ STM32L021x
- STM32L41x/ STM32L42x
- STM32L43x/ STM32L44x
- STM32L45x/ STM32L46x

Created: 20.Mar 2021, list added 30.Jan 2022

45.Not only frequency but also VOS setting implies FLASH waitstates

The first STM32 - the 'F1 family - were built using a 180nm CMOS technology, some 15 years ago. Since then, newer STM32 models are produced using even tighter silicon process - through 90nm down to 45nm - which makes the incredible level of integration found in these devices possible.

The voltages used commonly in board-level interconnects are relatively high, around 3V, to avoid issues with signal integrity at usual levels of noise. So these are also supply voltages commonly used at todays digital electronics, including the STM32. However, the tiny MOSFET transistors utilizing the shortest gate length (thus smallest dimensions) allowed by the mentioned technology nodes, would break down or have excessive leakage at those voltages, so most STM32 digital circuitry runs internally at lower supply voltage, 1.2 to 1.8V, some even lower, depending on particular model and technology used.

To achieve this "core" voltage while avoiding the extra burden of having one more power supply, most STM32 contain an internal LDO regulator, which drops the supply voltage from 1.8V-3.3V to the required core voltage. As the LDO means some energy loss, for the ultra-power-sensitive applications, some newer STM32 models have a variant with internal LDO switched off, core voltage to be supplied externally; and yet another models have variants with a built-in swithing-mode regulator (SMPS, requiring external inductor). But these are the exception, we won't discuss them here.

Many (perhaps most) STM32 allow the user to select one of several output voltages of the LDO. This allows a tradeoff between speed and consumption: lower LDO voltage means lower core voltage thus lower consumption, but also lower maximum clock frequency. There are usually two or three settings, and they are set by one or two bits in PWR module, usually called VOS.

The core voltage influences not only the logic circuitry, but the FLASH memory's speed depends on it. As FLASH is usually slower than the core logic, waitstates have to be inserted into each read, the number of waitstates has to be set by the user as appropriate.

So, when determining the required number of waitstates, besides system clock frequency, the VOS setting has to be also taken into mind. Details differ between particular STM32 families, so Reference Manual and sometimes also Datasheet for given model have to be consulted.

Created: 27.Mar 2021

46.Some pins on my Disco/Nucleo/EVAL board don't work as expected

The Nucleo/Disco/EVAL boards are valuable resorces as testing/development platforms (although rather undervalued by ST, as seen by their approach to provided documentation, examples and support). Users often build prototypes around these boards, utilizing the pins brought out from the target STM32 to pin-headers.

However, sometimes a pin on the pinheader does not work as expected.

Often, the root cause is a conflict with an on-board connection, for example the serial connection to the on-board STLink. These connections are realized through jumpers and (more often) solder-bridge connections (which, when connected by default, are populated by miniature 0R resistors), so by disconnecting these the given pin is freed up for the given application.

Sometimes, the pin on pinheader is connected to pin on mcu through such a solder bridge, which is by default disconnected, for some reason.

The associated User Manual to given board usually contains a table with all the jumpres and solder bridges and their functions, marking which of them are by default connected.

This problem occurs more often on the Disco boards than on any other boards - the Nucleo boards are usually "barebones" enough (contain almost no on-board resources except the target mcu), and the EVAL boards usually employ the largest package variant of the given family, so that there's usually enough "spare" pins brought out, unconnected to on-board resources.

The ultimate resource is the schematics for the given board, which should be consulted (together with the UM) before using any pin brought to the pinheaders. ST had a good practice appending schematics to the User Manual; unfortunately ST recently ceased to do so (claiming their own convenience as reason), and schematics has to be downloaded separately.

Created: 3.Apr 2021

47.TIMx_CHxN is not inverted if TIMx_CHx is not enabled

Advanced timers intended for halfbridge control (TIM1, TIM8, TIM20 and also TIM16/TIM17 - these are the timers requiring TIMx_BDTR.MOE to be set, to enable their outputs) have two output pins per channel, designated TIMx_CHx and TIMx_CHxN. Normally, these pins provide mutually complementary output (i.e. waveform on TIMx_CHxN is inverted version of TIMx_CHx, with an optional deadtime inserted).

On some smaller packages, or if pins are consumed by some other peripheral, sometimes only TIMx_CHxN is available for a given timer channel. Or, put simply, the user wants to use TIMx_CHxN alone, without using TIMx_CHx, for whatever reason, as an output controlled by Output Compare facilities for given channel (usually as PWM).

In that case, users are often surprised to see, that TIMx_CHxN does not provide the inverted waveform as is in case when TIMx_CHx is used simultaneously; but as standalone output (i.e. if the respective TIMx_CCER.CCxE=0), it outputs the "straight" version exactly as TIMx_CHx would.

This feature is well documented in RM, e.g. in the *Output control bits for complementary OCx and OCxN channels with break feature* table. If desired, the waveform can be easily inverted using the respective TIMx_CCER.CCxNP bit.

Created: 10.Apr 2021

48.TIMx_CHxN cannot be used as input to timer

Timers in STM32 have only a single counting element, but they have several multipurpose channels, which can be set as either capture or compare. Those channels have associated pins through the AF matrix in GPIO, the respective pins being designated as TIMx_CHx (e.g. TIM2_CH1 for the pin corresponding to Channel 1 of Timer 2).

The "direction" of the pins set in GPIO_MODER as AF and assigned to TIMx in respective GPIO_AFR, is controlled from the timer: if the respective channel is set to Input Capture (in TIMx_CCMRx.CCxS), it acts as an input (its output drivers are disabled); if it is set to Output Compare in TIMx_CCMRx.CCxS, it functions as an output (its output drivers are enabled pending other timer control bits, such as TIMx_CCER.CCxE or in case of Advanced timers, TIMx_BDTR.MOE).

In Advanced Timers, such as TIM1, TIM8, TIM20, TIM16/TIM17 etc, some channels have also a complementary output pin, TIMx_CCxN. These usually output an inverted version of TIMx_CCx waveform (with optional deadtime inserted), but they can function standalone, too. While from this fact it would appear that these pins can then be a replacement for respective TIMx_CCx, so, by extension, that they can in some circumstances (e.g. if TIMx_CCx are not used or even not available on some packages) be used as input on that channel, this is not the case.

TIMx_CHxN cannot be used for Input Capture, nor as input to Slave-mode Controller of the Timer.

Created: 17.Apr 2021
Modified: inputs generally, not just Input Capture: 17.Feb 2024

49.F3 CCM RAM can't be used for DMA, but can be used for code

The 'F3 came out later than the 'F2/'F4 (roughly at the same time as 'F0 and sharing much of the details in peripherals), featuring the same Cortex-M4F core than the 'F4, but at a lower maximum clock of 72MHz, smaller memory sizes and without the "power" peripherals such as OTG, ETH and FSMC. With its relatively rich analog-oriented peripheral set, it is mostly intended for motor control, and similar semi-analog real-time control applications.

Probably as a measure to cut the die size (thus unit price), and maybe also to provide the 'F4 a distinguishing element justifying its price, in 'F3 there's no FLASH access accelerator (called ART by ST, basically a jump-cache). This would mean, that the processor would be hampered by the relatively high number of required 2 wait-states when running at the highest speed[1].

Figure 1. System architecture for STM32F405xx/07xx and STM32F415xx/17xx devices

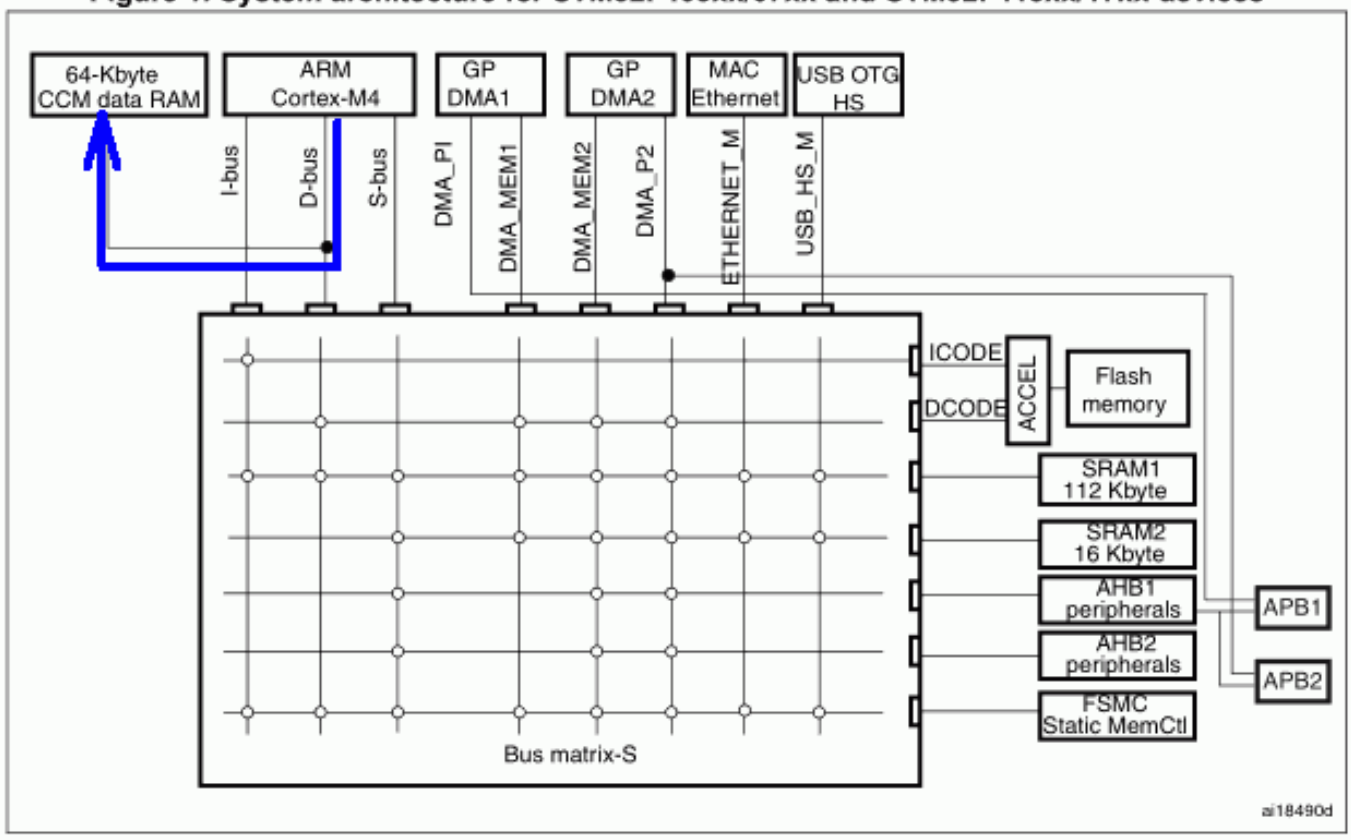
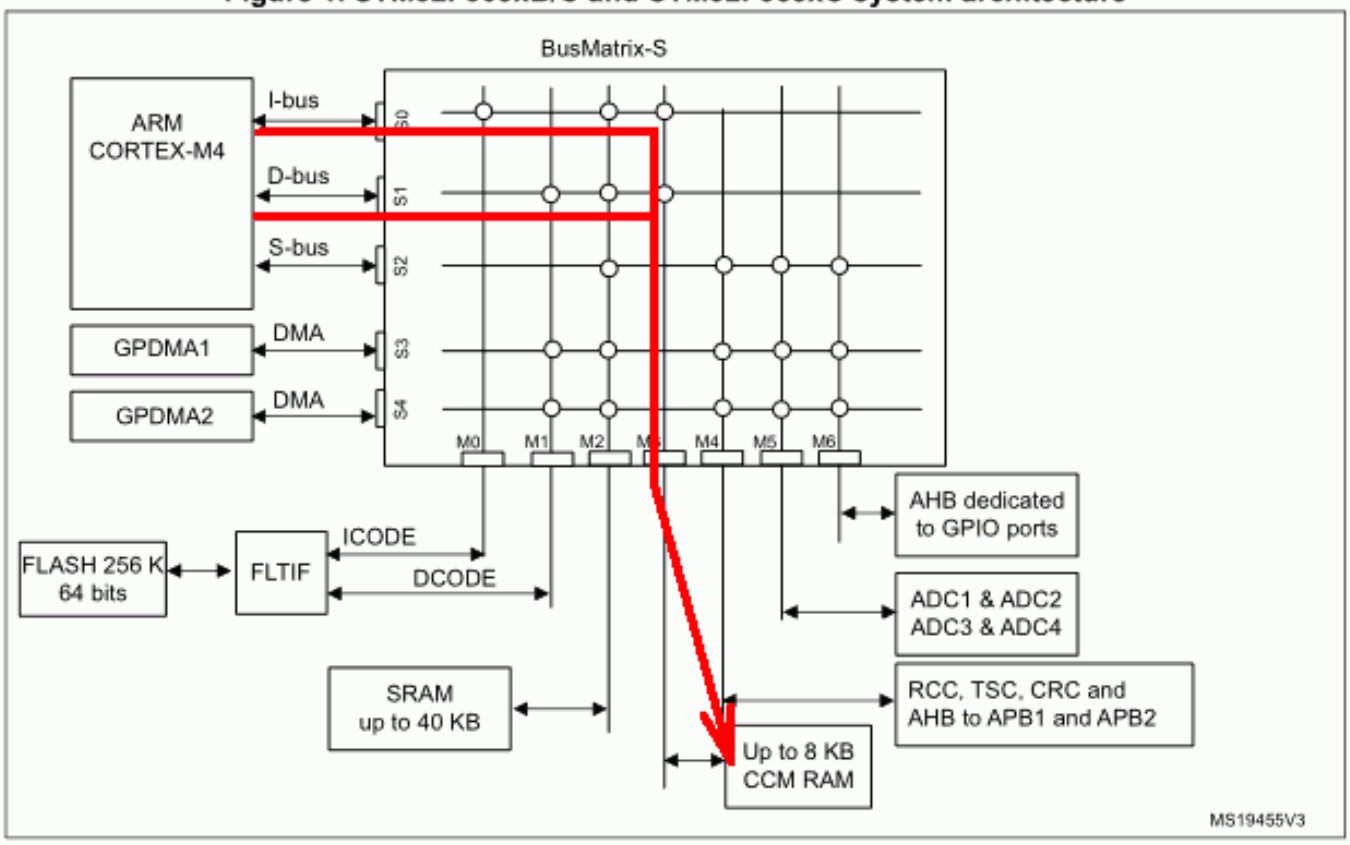


Figure 1. STM32F303xB/C and STM32F358xC system architecture



As users might want to speed up execution of at least some critical parts of code, there's an option to run code from RAM, with 0 waitstates. 'F3 - similarly to 'F4 - besides SRAM available to all masters on busmatrix, provides also a piece of RAM to which the processor has exclusive access, called CCM RAM (Closely-Coupled Memory). Same as in 'F4, this memory cannot be used for DMA, nor can it be target of bit-banding. But, while in 'F4, CCM RAM cannot be used to run code, in 'F3, the designers wisely addedd access to CCM RAM from both the I and D ports of processor, which means, that CCMRAM in 'F3 can actually be used also for running code.

1. The FLASH line is 64-bits wide, there is a 2-line prefetch, the I-port of processor is 32-bit wide while Cortex-M4 runs exclusively in Thumb mode which means 16-bit wide instuction word (with some instructions being 2-word long). The processor itself has a several words worth of prefetch queue. Also, in normal programs, there's enough loads and stores to relatively slow peripherals, allowing all the prefetch queues to fill up enough so that linear code will mostly run just as fast as if it there were no FLASH waitstates. The problem is, when the code branches (jumps) - whether due to normal program flow or due to interrupts - and also when the program reads literals (constants) from the FLASH. There are techniques to somewhat reduce occurence of both these things, but discussing them is outside the scope of this article.

Created: 24.Apr 2021

50.In 'F4 CRC, data are ignored if written too soon after reset

The CRC unit, before performing one round of calculation, should be reset by setting CRC_CR.RESET bit. This is a self-clearing bit, i.e. software sets it, hardware clears. The net result is, that the CRC_DR is set to its default value, 0xFFFF'FFFF. After this, the user can simply dump the data onto CRC_DR and then read out the calculated CRC value from the same CRC_DR. While one calculation step takes 4 AHB cycles, there is no risk of overwhelming the CRC unit, as it automatically inserts waitstates to throttle the writes appropriately.

However, in STM32F407, if a datum is written too soon after setting RESET bit, it gets ignored. Apparently, the reset procedure takes a couple of AHB cycles, and the waitstate mechanism does not cover this case.

In the majority of cases, this problem is covered up by the fact that normally, after writing to CRC_CR, software has to fetch the first datum from the memory before writing to CRC_DR, and time needed for this is apparently sufficient for the CRC unit to finish its reset process. However, in some cases, the first datum to be written is already loaded in some of the processor's registers - e.g. because it has been calculated or fetched beforehand for some other operation, or because it's known to be 0x0000'0001 i.e. identical to the word written into CRC_DR) and the compiler optimalizator spots this and generates two successive instructions to write into the CRC_CR and CRC_DR, thus revealing the problem.

This is one of those issues which has not been confirmed by ST. The suggested workaround (see the thread in STM32 forum linked below) is to read back CRC_DR until RESET bit reads as zero. However, there is no mention of this procedure in RM0090, nor is it followed in any example code (e.g. in Cube). As such, it may work purely coincidentally, wasting enough time to cover up the problem; but then potentially also wasting more than absolutely needed.

As newer STM32 models have extended CRC unit, it is unclear whether this problem pertains to them, too.

This issue has been reported on STM32 forum on April 07, 2017 as "CRC data ignored after CRC reset" and then again here.

Created: 1.May 2021

51.Not all TIM_CHx are created equal

Timers, besides having the counting unit (prescaler and counter) and the control harness, they have several Capture-Compare channels. These don't have the ability to count, yet they represent most of the inputs and outputs of the timer (besides external clock inputs and "emergency" break inputs), which usually also connect to the actual pins (through the GPIO matrix). They also input/generate (through the control harness, or sometimes even directly) the internal signals going from/to other peripherals within the chip.

In STM32 timers, there's a varying amount of channels in various timers (1 to 6, 4 channels being the "gold standard"). The channels also have somewhat varying capabilities in different timers - good example of this are the complementary output and break features of channels in Advanced Timers. Mostly, timers with the same number have similar capabilities and similar number of channels across all STM32 families, but exceptions do exist.

However, even channels within one timer are not equal when it comes to their capabilities. Here is an incomplete list:

- only CH1 can generate TRGO from Input Capture (with the entirely incorrectly named Compare Pulse setting of TIMx_CR2.MMS)
- only CH1 and CH2 can be used as input to the slave-mode controller, and only CH1 with both edges (i.e. when the edge detector is to be used)
- as a consequence of previous item, only CH1 and CH2 can be used as inputs for encoder mode
- inputs in input capture mode can be "crossed" only between CH1/CH2 and CH3/CH4 couples
- in newer STM32 where some timers feature Asymmetric and Combined modes on output, only CH1/CH2 and CH3/CH4 pairs are coupled for this feature (in newer STM32 models' (e.g. 'G4) Advanced timers, CH5 can be combined with CH1/CH2/CH3, too - this still omits CH6 and CH4 from the mix)
- where available, the XOR function combines specifically CH1+CH2+CH3 and inputs to CH1
- in Advanced Timers, CH4 does not have complementary output, nor the BREAK logic (only CH1, CH2, CH3 have these)
- where available, CH5 and CH6 have restricted usability:
  - they don't connect to pins through GPIO
  - they can't generate TRGO
  - even if they do generate corresponding flags as TIMx_SR.CCIF, they can't generate interrupts nor DMA requests
  - as they came to be as an afterthought and ST rightly sticks to compatibility of registers layout within a peripheral, the CCRx for CH5 and CH6 are not continuous with other CCRx, which makes updating them through the TIMx_DCR/DMAR mechanism tricky

---

Created: 8.May 2021

STM32 gotchas

52.TIM - in Encoder mode, prescaler must be 0

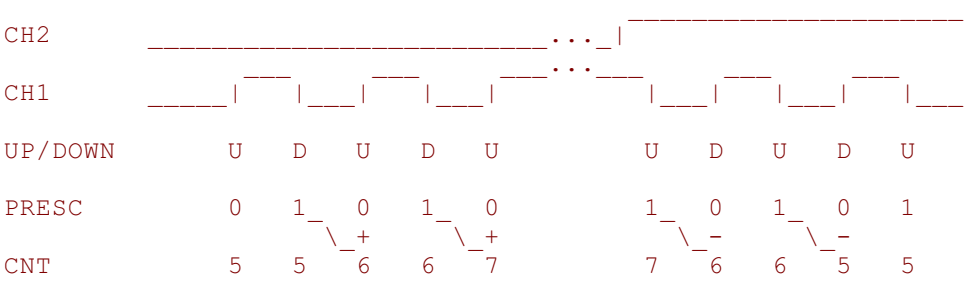
Many (although not all) of the STM32 timers have an Encoder Mode option in the Slave-mode Controller. When set, the timer's counter counts according to edges on CH1 and CH2[1] up and down, changing direction (as indicated by the TIMx_CR1.DIR bit) according to the order of edges, so that it allows connecting a quadrature encoder.

There are essentially two modes available[2]. In one, edges of only one channel contribute to counting - this mode is often called "x2", as there are two counts per the quadrature cycle. In the other mode, edges of both channels are counted, this is "x4" mode.

Sometimes, users wish to use the "x1" mode (or would like to have counted even less edges, for whatever reason), where only one edge is counted per quadrature cycle (e.g. only upgoing in one direction of movement and only downgoing in the other direction, only on one channel). This is often required for encoders used in manually turned "knobs" with detents, as their mechanism usually generates one full quadrature cycle per detent. It appears, that using prescaler (set to 1, i.e. dividing by 2) would solve the problem.

However, after setting prescaler[3] to nonzero, the counter starts to behave strangely. While the encoder is turned/moved only to one direction, it works quite OK, but as soon as direction changes, counts may start to be missing, or behave "strange" and "unpredictible".

The root of the problem is in the fact, that the prescaler counter counts only in one direction. While the encoder-mode logic decodes the order of edges and changes the TIMx_CR1.DIR bit accordingly, this applies only to the "main" counter (CNT), but not to the prescaler counter. Prescaler counter counts always down, regardless of TIMx_CR1.DIR. The following diagram illustrates this effect for TIMx_SMCR.SMS=0b001 (Encoder mode 1 - Counter counts up/down on TI1FP1 edge depending on TI2FP2 level) - counter increments/decrements according to indicated direction, when prescaler counter transitions from 1 (= value set in TIMx_PSC) to 0:



Note, that at the beginning the encoder was moving only one edge up and down, yet it produced an ever increasing value in CNT; and after moving one more edge in the middle of the diagram, again moving up and down produces an ever decreasing value.

There is only one remedy to this problem: if "x1" or other "divided" mode is desired, prescaler can't be used, rather, the resulting count has to be post-processed (divided) after reading it out from TIMx_CNT, in software.

This problem has been discussed in a local conference.

---

1. Only CH1+CH2 are available for Encoder mode.

2. Three, but two of them are only "mirrored" versions of the same mode, changing roles of the two channels. In newer STM32 Timers, there are also other modes available for different models of rotary/linear movement encoders, e.g. Direction/Pulse. For details, resort to the given model's RM.

3. ... and considering that prescaler is preloaded, e.g. followed the setup by a forced update by setting TIMx_EGR.UG...

4. Encoders in manually turned control "knobs" often output "noisy" or "bouncy" output, where there are several transitions instead of one clean edge, even if turned in one direction. This is equivalent to encoder being moved one edge up and down repeatedly, i.e. the output count of "normal" decoder would increment and decrement one count repeatedly, rapidly, until the "bouncing" stops and levels stabilize in the new state.
This is normally not a problem, as mechanical detents stop the movement in between edges; but with the prescaler set to nonzero, the above described effect may be seen due to "bouncing" even if the encoder is turned in one direction i.e. without explicit mechanical reversal. In the STM32 timer, filters can be switched on input channels (see TIMx_CCMRx.ICxF), somewhat mitigating this problem; but with prescaler set to 0, there's no real need to use them, as the effect of "bouncing" on the final count is

always effectively cancelled (by the up-down-increment-decrement logic) at the moment the encoder comes stationary between edges.

Created: 15.May 2021

53.TIM does not run if ARR was changed from 0 to nonzero (due to preload)

The TIMx_ARR may be preloaded, by setting TIMx_CR1.ARPE bit. This means, that writes to this register don't change the timer's behaviour immediately, but will get active only after the next Update event, usually the next timer overflow.

If TIMx_ARR is set to 0, timer will stop at the next update. Alternatively, if this is set while timer is not running, then after enabling timer it won't run.

If this is combined with preload being set, there will never be an Update due to overflow, so setting TIMx_ARR to a non-zero value won't get "active" and the timer won't run (unless Update is subsequently forced by setting TIMx_EGR.UG, or from the slave-mode controller set to Trigger mode).

While this may seen as obvious, the TIM initialization may be performed by a library such as Cube or SPL, and the user may not be aware of the TIMx_CR1.ARPE being set, or the consequences of this.

An example of possible confusion here.

Created: 22.May 2021

54.In 'F1, RDP also sets write protection on first sectors

The basic mechanism which prevents unauthorized reading/copying of firmware in FLASH in STM32 is called RDP (ReaD Protection). In newer STM32 families there are also additional protection mechanisms (Proprietary code readout protection (PCROP), sophisticated secure mechanisms in 'H7 and 'L5; but RDP is the staple security mechanism, which is present since the very first 'F1 family.

RDP has 3 levels:

- Level 0 is no protection, this is the factory setting;
- Level 1 is a basic protection (known to be "leakable" in some families) which disables reading of FLASH though debug (SWD) but still allows debug itself and also allows reverting to Level 0 (while performing a FLASH bulk-erase);
- Level 2 being the ultimate protection, completely disabling SWD, with no way to revert.

In all settings, program running from FLASH still can both read and write FLASH - unless, that is, FLASH write is disabled, through a different mechanism.

Most newer families have a separate protection mechanism to disable erase/write individual sectors (or groups of sectors, e.g. sector pairs in 'F0/'F3). However, this is not the case of 'F1.

In 'F1, there is no separate write protection mechanism; however, setting RDP to other than Level 0 automatically enables write protection on the first couple of sectors:

> *Once the protection byte has been programmed: [...] Pages 0-3 (for low- and medium-density devices), or pages 0-1 (for high-density and connectivity line devices) are automatically write-protected. The rest of the memory can be programmed by the code executed from the main Flash memory (for IAP, constant storage, etc.), but it is protected against write/erase (but not against mass erase) in debug mode or when booting from the embedded SRAM.*

This for the 'F1 family is documented in PM0075, "Flash Programming Manual". Again, such a "Programming Manual" is unique for the 'F1 family - in all subsequent families the FLASH-related details were simply included in the respective Reference Manual.

However, there's an exception to this exception, too. The largest-FLASH STM32F103xF/xG (>=512kB FLASH), aka *XL-density*, are described in PM0068; and they don't have this quirk.

This issue was brought up here.

Created: 29.May 2021
Note about the PM0068-related 'F1 added: 23.September 2023

55.'F2/'F4/'F7/'H7 unreliable at higher frequencies, due to improper VCAP capacitor

In prototype hardware, unstability is not an unheard of problem. Contrary - although we tend to consider the modern microcontrollers as commonplace, they are still complex high-frequency circuits and deserve attention accordingly. Stability problems tend to be exacerbated with increasing clock frequency.

There are many potential sources for instability, but the power source distribution (including ground) and decoupling belong to prominent ones. However, as increasingly more users are aware of this particular problem, and design accordingly, it appears that in some STM32 families, a different, although related issue comes up surprisingly often. Namely, using inadequate capacitors on VCAP pin(s).

The STM32 are built on technologies requiring a low voltage for the vast majority of digital circuitry - 1.8V and below, depending on family. While on some models this voltage can be supplied externally, this represents a minority, most of the STM32 incorporate an on-chip LDO voltage regulator generating the required "digital" supply voltage from whatever VDD is available (VDD itself determining mainly the IO pins working voltage). The 'F2/'F4/'F7/'H7 families have a relatively high power consumption, so this LDO has an output, marked VCAP, to which an external decoupling capacitor has to be connected.

I believe the reason for this being so often the root of the problem is caused by users misreading the datasheet. While it clearly states that VCAP pins have to be connected to 2.2uF/4.7uF low-ESR (ceramic) capacitors placed closely to the chip, users tend to inadvertently connect the "ceramic" word with nF capacitances, thus populating incorrectly small capacity, resulting in high digital-supply voltage fluctuations and, ultimately, stability problems.

Sometimes, of course, it happens also that the problem is caused by using improper (electrolytic) capacitor, incorrectly routed (long) connections (including ground), or with hand-assembled prototypes, bad solder joints.

This is a recurring theme on the forum, but sometimes it happens that the same issue occurs twice in rapid succession, independently of each other.

Created: 5.June 2021

56.Internal temperature sensor on ADC indicates surprisingly high temperature

One of the channels in most if not all STM32 families' ADC is connected to an internal temperature sensor. Users, when trying to measure the chip's temperature, often report unexpectedly high temperature readouts.

While there are many potential sources for error - including incorrect temperature calculation from the raw ADC readout - most of the time the measured temperature is correct. Users usually don't realize, that the sensor is part of the chip, thus it indicates temperature of the chip itself, which can significantly differ from the ambient temperature. Differences of 15°C and more in an 'F4/'F7 running at maximum clock with many peripherals enabled, are perfectly normal.

An approximate estimate of this difference follows from the supply voltage, supply current and the thermal resistance of given chip (a more precise calculation would need to include also other factors, such as impact of the method of mounting of given chip/thermal properties of the PCB, surrounding sources of heat, air circulation, fluctuations in the power consumption, considering the power dissipated by currents flowing in and out of IO pins, etc.)

As an example, let's take the datasheet value for typical current consumption of STM32F407 running at 168MHz with all peripherals enabled, that's 87mA; as this value is given at VDD=3.3V, the power dissipation is 287.1mW. Let's assume we use the LQFP100 package for which the junction-to-ambient thermal resistance is given by DS as 43°C/W, this results in the basic estimate of difference between junction (chip) and ambient temperature of 12.3°C.

Created: 12.June 2021

57.'H7 lifetime may be surprisingly short (at elevated working temperatures)

Microcontrollers are generally considered to be industrial-grade equipment with decades of useful working life. Usually, FLASH retention time is seen as the major limiting factor, with normal retention times at usual working temperatures being usually quoted between 20 and 100 years for various microcontroller models and makers.

Enter the ultra-high performance 'H7 family, and this is no longer the case.

Contrary to most other microcontrollers, the 'H7 family with its blazingly fast multi-hundred-megahertz performance, and huge megabytes of memories, is built using an advanced 45nm silicon technology. However, small feature size not only increases the number of transistors which can be crammed onto a chip, and increases their speed, but also increases complexity of manufacturing and decreases all margins for potential errors. This also increases impact of fearsome effects like electromigration and unwanted diffusion, which change behaviour of both conductors and semiconductors on the chip.

These effects result in reduction of effective lifetime of the chip, depending on voltage of the digital power supply voltage (usually given by setting of the internal voltage regulator as VOS0..VOS3) - which in turn determines maximum operation frequency; and junction temperature. Note, that junction (chip) temperature may be significantly different from the ambient temperature.

This issue is discussed in AN5337. At the maximum digital power supply voltage - VOS0 - and 90°C junction temperature (which at the highest frequencies and many peripherals running can be realistically reached at ambient temperatures around 50°C, nothing unusual in less-than-well ventillated enclosures) the expected lifetime is around 5 years, rapidly decreasing to 2 years with a mere 15°C increase of junction temperature to the maximum allowed 105°C.

High performance comes at a cost. Welcome to the 21th century.

See also g167.

Created: 19.June 2021

58.Changing clock frequency requires careful readjustment of peripherals

Microcontrollers typically work at one working (system, bus) frequency all the time, with stopping the clocks altogether when power saving is needed. However, sometimes there's a requirement to change the working clock frequency, for whatever reason - it may be sustained function while saving power, or an emergency situation when a primary clock source (e.g. crystal oscillator) fails and a backup clock (e.g. internal RC oscillator) is brought in.

It may sound obvious that in such situation those peripherals, which functionality depends primarily on the system clock frequency, e.g. UART. However, as this is a seldom used operation, there may be less than obvious sources of error or confusion in way.

For example, if "libraries" such as SPL or Cube are used for the setup, the system clock frequency is sometimes calculated from a value which is usually given by a #define as a constant, so it may require some understanding of the involved mechanisms to be able to coax the usual setup routines to use a variable instead.

Some routines may be written so that they assume peripherals' control registers to be at their reset value (e.g. using RMW operations such as |= or &= to modify only certain bits of a bitfield in a register); these may need to be rewritten for correct operation.

Some peripherals don't allow changing timing-related registers while operating, or such change may cause the peripheral to end up in an unknown, non- or partially-functioning state. Such peripheral may then require a deeper readjustment, maybe requiring a complete reset (possibly employing the respective RCC_AxBRSTRx bit) and a new setup.

Also, peripherals, which are clocked from external clock (e.g. TIM in external clock mode, or I2C in slave mode), still may employ internally clocked filters on the signals, which may need readjustment.

Created: 26.June 2021

59.PF4 on 'F303 does not work (because it's missing on 'F303RD/E)

STM32 are fairly mutually compatible. There is a great deal of compatibility across all families at least in positioning of the power, reset and oscillator pins, allowing a common board such as the original Nucleo-64 to cater for multiple chips with a dozen or so solder bridges. There are notable exceptions (e.g. LQFP-100 between 'F4 and 'F7 has a whole row of pins shifted by 1; the relatively new chips with the embedded SMPS break compatibility even with the same model number, differing only in suffix), and new 'G0 and 'G4 families deliberately break that sort of compatibility for an optimized power/ground arrangement leaving more signal pins available.

Where the basic pin compatibility is maintained, there's also some compatibility in assigment of peripherals to individual pins. As the different families have different set of peripherals, this is of course far from being universal, but within a family, the "downwards" compatibility is usually maintained quite well.

And, one would expect, that STM32, which have the same basic model number and differ only in suffix, would have the same pin/peripheral layout and would have only different memories sizes and pin number/package, as designated by the suffix.

However, there are surprising exceptions to this.

Perhaps the less surprising are the various variants of the venerable 'F103 model, and the 'F030 model, both in fact representing several different chips, with slightly differing layout of peripherals. These have to be observed carefully in the respective datasheets, should one try to replace one variant of given model with another, either because of need for more memory, or simply because of current (un)availability of a given variant.

One of the most surprising differences within one model can be found in the 'F302/'F303 model, namely between the "lower" B/C-suffixed variant and the "higher" D/E-suffixed variant. While there are slight differences in the behaviour of peripherals (e.g. comparators in the "higher" variant do not have adjustable speed and hysteresis and miss some of the possible input pins available in the "lower" variant), the difference which is a true gotcha, is the absence of PF4 pin in the D/E as compared to the B/C, in LQFP packages smaller than the largest one. The PF4 pin in B/C - for example pin 18 on the LQFP-64 package - is replaced by VSS in the D/E.

Meticulous comparison of the respective datasheets of course reveals this difference, and there's even a "migration" application note available, AN4807, where the two pinouts are highlighted. Nonetheless, changing pin from GPIO to power/ground within one model is not something most users would expect and may present an unpleasant surprise when attempting to use a replacement for any of the reasons already mentioned.

Created: 3.July 2021

60.TIM DMA burst (DMAR/DCR) does not work properly, if TIMx_DMAR observed in debugger

Most timers (except those which don't support DMA, e.g. the rudimentary TIM6/TIM7) provide a mechanism through which multiple TIM registers can be filled (or read out, although that's probably a rare application) in one go, sometimes called a "TIM DMA burst". It consists from two registers - TIMx_DCR, to which the index of first target register, and number of registers to be accessed in one "burst", is written; and TIMx_DMAR, which is not a true register, but a "window", through which the "currently indexed" register can be accessed. The "current index" is in fact an internal register, inaccessible directly to the user.

DMA's Peripheral pointer is to be set to TIMx_DMAR. The first DMA transfer of the "burst" is triggered by usual Update event (with TIMx_DIER.UDE being set). The subsequent transfers are then triggered by the "burst" mechanism upon accessing TIMx_DMAR (DMA writing to); up until an another internal "counter" reaches the count written to TIMx_DCR.DBL.

TIM can't distinguish between accesses from DMA, from processor (debugger is part of processor), or any other busmaster. If the TIMx_DMAR register is observed in debugger, this advances the internal "current index" and counter registers, thus messing with the DMA burst mechanism as it's intended to work. This can be tried experimentally: after setting up TIMx_DCR to access several registers previously written with various values, in a stopped processor with no DMA running, reading out TIMx_DMAR by debugger ("refreshing" the registers view, or setting the debugger to automatic readouts, "live mode") results in varying value in TIMx_DMAR, as its "window" shifts across the indexed registers.

The runtime results are then surprising, with the TIM registers getting "out of sync" with source data in RAM.

This is yet another, although maybe less expected, consequence of the fact, that debugging is intrusive.

Created: 10.July 2021

61.In 'F42x/43x, erase sector does not work (due to confusing sector numbering)

The STM32F427/249/437/439 comes in two memory sizes: 1MB FLASH and 2MB FLASH. As with all other STM32, individual sectors this FLASH can be erased and rewritten by the application itself.

Sectors in 'F42x/43x are non-uniform, with sizes increasing from 16kB at the beginning of the FLASH to 128kB up to the 1MB boundary, where this pattern starts again. When erasing a sector, the number of sector to be erased has to be written into FLASH_CR.SNB. The varying size of sectors makes calculation of number of sector from memory address a nontrivial task; however, there's one more gotcha ahead.

While in the table denoting the partitioning of 2MB FLASH into sectors, these are numbered sequentially from 0 to 23, the actual value to be written to FLASH_CR.SNB is different. Sectors 0 to 11 are written as 0-11; but starting from sector 12, the value written into FLASH_CR.SNB has to be increased by 4.

As if this would not be confusing enough, there's one more twist on this numbering. The FLASH in 'F42x/43x is dual-bank, which means that it is possible to erase/program one bank while run code and read data from the other[1]. In the 2MB model, simply the first 1MB is one bank, the second 1MB is the other.

In the 1MB 'F42x/43x model, the picture is more complex: by default, the 1MB is a single-bank FLASH and also behaves as such. However, there's is a bit in the nonvolatile Option bytes, DB1M, which allows the 1MB model to function with dual-bank FLASH. When this bit is set, partitioning of FLASH changes entirely: the first 512kB is one bank and it is partitioned as expected, i.e. exactly like the lower 512kB of the entire FLASH. At the 512kB boundary, the partitioning is restarted (i.e. again starts with 16kB sectors, sector sizes increasing as we go higher with addresses), but also the sectors numbering jumps to 12, as if it would be the 1MB boundary between the two banks in the 2MB model.

And with this, the need to add 4 for sector numbers above and including 12, when writing to FLASH_CR.SNB, still applies.

---

1. In single-bank FLASH, which is in the vast majority of STM32 models, attempt to read/fetch from FLASH being programmed/erased leads to processor stall until the write/erase finishes, so programs which need to program/erase while running have to run from RAM or external memory where available.

Created: 17.July 2021

62.Backup domain is not reset by setting RCC_BDCR.BDRST (unless PWR_CR.DBP is set, too)

This is one of the issues which is not documented by ST, only found by observation. While observed on an STM32F427, it probably pertains to most or all of the 'F4 family, and most likely to some or all of the other families, too.

Most STM32 families (with exceptions like the STM32L0 family) do have a separate backup power domain, which is powered through a separate VBAT pin, when the main VDD is off or low (and from VDD through an internal switch, when VDD is above the powerup threshold). This domain contains primarily the RTC, the related LSE oscillator, backup registers (which in some families are a separate block, in others they form part of the RTC), and in some families also a small portion of RAM.

To protect registers in the backup domain from inadvertent writes from the "normally-powered" domain, there is a protection bit (PWR_CR.DBP), which is cleared by normal system reset, and has to be set before any write access to the backup domain.

As the backup domain is intended to be powered all the time, registers of peripherals in the backup domain are not reset by the normal system reset, only if there was a powerdown in the backup domain, i.e. when the battery is disconnected from VBAT while VDD is powered down.

There is also an optional method to reset the backup domain, by setting the RCC_BDCR.BDRST bit. Unlike other bits in the RCC_BDCR register, this bit is not write protected by the PWR_CR.DBP bit. And, indeed, it is possible to set RCC_BDCR.BDRST while PWR_CR.DBP=0, as witnessed it being still set when reading RCC_BDCR back.

However, setting RCC_BDCR.BDRST will not reset the backup domains, as can be seen on other bits of the very same RCC_BDCR register. They get reset only after PWR_CR.DBP is set.

This behaviour is not documented in the respective STM32 user manuals.

There is an additional, potentially confusing, side effect to resetting the backup domain using RCC_BDCR.BDRST. In RTC, if BYPSHD=0 (which is the default setting), the calendar registers (RCC_TR, RCC_DR, RCC_SSR) which are in the backup domain, are not accessed by the processor directly, rather, they are periodically copied into shadow registers in the normal power/APB clock domain, and then these shadow registers are visible by the processor. When backup domain is reset using RCC_BDCR.BDRST, this copying cedes to work, but the shadow registers - not being in the backup domain - retain the last copied value. The outward behaviour is, as if the RTC would "freeze" instead of being reset. This is not anything severe or problematic, but may come as a surprise.

Created: 24.July 2021

63.Compatibility within the Value Line (0-ending) models

We've written previously, that while mutual compatibility of STM32 is quite reasonable in between the families (at least the older ones, the 'G0/'G4 deliberately breaking this sort of compatibility) and then a bit better within a family, yet it's not perfect even within a single model number. That single model number may consist of variants, seemingly differing only in memory sizes and available packaging options, but in fact being different chips, with minor but sometimes significant deviations, e.g. in availability of a certain peripheral, or functionality of certain pins.

Therefore, one has to be very careful when migrating even between variants of the same model number, for whatever reason.

This problem gets more pronounced with the so called "Value Line" models, which are present almost in each STM32 family, with model number ending with 0 (e.g. STM32F03**0**). These models are in fact "downgraded" models from the same family, which see less production testing, have some features/peripherals unavailable, less memories available, and have reduced operating temperature span. These all contribute to lower manufacturing cost hence lower unit price. While the "unavailable" features may in fact be present, they are not guaranteed to work properly; ST may take steps to disable them; and also, would there be enough demand, ST may in the future create new masksets for given models, genuinely omitting those features to achieve optimized silicon area.

However, what ST likes to do is, that they "gather" several models under one Value Line model number. This means, that under the same model number, several different models can be found as "memory variants". The differences may be relatively minor as in the case of 'F030; dramatic, as in case of 'F730 where the models in smaller packages stem from 'F722 whereas the larger packages originate from 'F723, and these models differ significantly in that the 'F723 (and thus the larger-than-100-pin 'F730) is the only STM32 incorporating USB HS PHY; all the way to surprising, as in case of 'L010, where a model with larger memories have much worse availability of certain peripherals through the GPIO matrix than the "smaller-memory" model, in the same package.

The Value Line parts represent a very useful portion of the STM32 family, as there are many applications where the large temperature range and some of the peripherals are not needed, while the smaler price is a bonus. However, when migrating "within model" for whatever reason (availability, need for larger memory), the newly chosen model has to be carefully compared to the orginal one, to avoid any unpleasant surprise.

In the 'F1 family, while ST designates the 'F100 as Value Line, in fact, 'F100 are not "downgraded" versions of 'F101/102/103; rather, they are different chips. While 'F100 indeed have the lowest rated system clock of all 'F1 (24MHz, vs. 36/38/72MHz), and they share most of the basic peripherals with the rest of 'F1 family, they also have a HDMI-CEC peripheral, which is not present in other members of 'F1 family. OTOH, 'F102 and 'F101 are both "downgraded" versions of 'F103, so in the spirit of this article those can be seen as the "Value Line" of the 'F1 family.

Created: 31.July 2021
'F1-specific footnlote added: 29.4.2023

64.Standard C library function does not work (not STM32-specific)

This issue comes up from time to time, such as recently on the STM32 forum in the form of atanf() not working.

The reason for this problem is a somewhat surprising requirement of the C standard, that **all symbols used as names for standard C library functions**, as well as all macros defined in the standard C library **are reserved.**

Thinking of it, this is not an unreasonable requirement. At the end of the day, the compiler and its environment (which includes also the linker and the default libraries) has to have a certain amount of keywords which have a special utility. The user is then not supposed to use these keywords as user identifiers, i.e. variable or function names.

However, what may be surprising is, that there is no requirement by the C standard for error or even warning to be issued by the compiler, when this inadvertently happens.

In the above case what happened was, that the user wrote a function called fabsf() which overrode the stanard fabsf() function, yet functioned in a different way. The GNU linker accompanying gcc during linking looks up the symbols in the user's code first, (and also the libraries don't have all symbols resolved internally - discussion of the minute details involved here is outside the scope of this article) before looking them up in the libraries, that's how the user's fabsf() function ended up used by the library's atanf() function instead of the library fabsf() function.

While this description may indicate that library functions will always be overriden by user functions of the same name, this is far from being the case, as some of the library functions are actually integrated into the compiler itself, and also the linking process is highly toolchain dependent.

In short, simply obey the rule and treat all standard library symbols as reserved, as the standard prescribes.

Unfortunately, the standard C library (defined in chapter 7 of the standard) is extensive and a quite big number of such symbols adds up to the list of reserved keywords already defined by the C language's core (defined in 6.4.1).

Created: 7.August 2021

65.Standard C library function (e.g. printf()) fails in interrupt or RTOS (not STM32 specific)

It is not a widely known fact, that **standard C library functions are not reentrant** (C99 7.1.4#4). It means, among other things, that they cannot be used both in main() and in interrupts, simultaneously. It also means, that if some form of multitasking (usually called RTOS in the realm of microcontrollers) is used, the same library function cannot be used in several processes/tasks/threads simultaneously.

Although this is a generic issue, it usually does not exhibit itself for most of the standard library functions. Compilers/libraries authors recognize the importance of reentrancy and usually give explicit guarantee for reentrancy at least for the math part of the library; and also provide clues or methods to make also rest of the library reentrant. This information of course toolchain dependent, i.e. non-portable, and it has to be looked up in the documentation accompanying the toolchain.

printf() and its family is an extremely complex function, if it attempts to implement all the features imposed by the standard, and its newlib implementation uses some form of dynamic memory allocation, which represents the non-reentrant element. While newlib does implement some techniques to ensure reentrancy, these require close attention, they are dependent on the environment where the library is used, and are not always properly implemented or utilized. This resulted in issue reported on the STM32 forum, with this printf() implementation failing when printing floating-point numbers in FreeRTOS environment.

Created: 14.August 2021

66.SPI master Tx outputs 16 clocks, even if its Data Size set to 8 bit

SPI module even in its original form in the early STM32 families is relatively complex, however, its basic operation follows the expected pattern. In its Master mode, after setting the proper mode, selecting baudrate, and deciding the way how NSS is used, it is enough to select whether transmitted frame is 8-bit or 16-bit wide; and then writing the Data register when it's empty simply makes the SPI module to output 8 or 16 pulses on SCK together with shifting out the data onto MOSI (and simultaneously shifting in data from MISO).

However, starting with the 'F0/'F3 families, users often find, that following this simple procedure while setting 8-bit frame, sometimes results in transmitting 16 bits instead of 8 bits, per each written piece of data. MSB of the transmitted frame is usuall all 0. Why is that?

Of the 3 basic serial communication interfaces - USART, SPI, I2C - SPI is the one generally capable of the highest transfer speeds. As such,it makes sense to try to reduce any bottleneck in moving data between SPI module and the rest of the chip. So, one of the improvements to the SPI module was, what ST calls **data packing**: whenever the frame size is set to 8 bits (or less), it is possible to read or write two bytes at once, simply by writing/reading a whole 16-bit halfword at once[1].

This behaviour is in the new SPI module implicit and cannot be disabled. It makes use of the fact, that within the 32-bit bus system (AHB/APB) used in the STM32, besides data and addresses, there are also signals indicating, what is the width of data being transferred through the buses. It is the busmasters - processor, DMA units - which control these signals, although users may not always be aware of them.

In DMA units, the transfer width (incorrectly named "alignment" in the Cube realm) is explicitly set in the control register.

In processor, load and store instructions have data width encoded directly in the instruction, this is reflected in the respective mnemonics (STB/LDB for byte, STH/LDH for halfword). So, to avoid data packing to send out 16 bits where the user intends to send only 8 bits, one has to use STB to write the data to SPI data register. But most users don't use assembler, they use C instead, so how to accomplish that there?

As SPI registers are defined in the CMSIS-mandated device headers so that the data register is defined as 16-bit type,simply writing to it as

```
SPI->DR = data;
```

results exactly in the 16-bit transfer, and thus SPI transmitting 16 bits, which we wanted to avoid. The universally accepted way to do this is use type punning[2] by taking pointer to the SPI data register, casting it as pointer to 8-bit type, and then writing to it by dereferencing the pointer[3]:

```
*(volatile uint8_t *)&SPI->DR = data;
```

---

1. The next logical question is: hey, but why going just half way far, why does not use this great feature the fact that the STM32 are 32-bits, so by writing a whole word, you could transfer 2 halfwords and 4 bytes at once? The anwer is twofold: one reason is, that at the end of the day, even if the features of these chips are incredibly small, distributing a full-fledged 32-bit bus across major portions of chips (read: APB buses connecting dozens of peripherals) still results in high complexity, silicon area, reduced yield, and thus increased cost. The other reason is historical - many of the STM32 peripherals are IPs taken over from older 16-bit ST mcus, so they still bear some of their legacy.

2. C, being a high(-ish) level language, does not define a good, universally portable method of prescribing, how exactly the hardware should work. Type punning in any form (either using pointer casting or unions) is a feature with undefined behaviour thus broadly speaking can't guarantee the expected goal. However, we here are talking about a very particular usage of the language, where the handful of existing compilers make conscious effort to behave in the "expected way" in this particular regard.

3. ARM in CMSIS headers define *__IO* as an alias to *volatile*, to provide no benefit while violating C naming rules (C99 7.1.3#1).

Created: 21.August 2021

67.Where are GPIO AF numbers given for individual pins? And where are the NVIC registers described?

Users are often confused by the way how STM32 documentation is organized. This is partially caused by the fact, that the processor core - ARM Cortex_M series - is not an ST product. It's just one of the modules ("intellectual property", IP) from which the chip is "assembled", and the documentation for it comes from ARM (the company). They provide a boilerplate template for all their licencees, who are supposed to generate a customized version of it, as the core can be "configured" (e.g. the floating-point unit in Cortex-M4 and Cortex-M7 is optional, Cortex-M7 can optionally have a double-precision FPU, etc.). In the STM32 world, these come as Programming Manuals (PM), e.g. PM0214 for STM32 with Cortex-M4 core.

One of the peripherals, which is described in these Programming Manuals is the interrupt module, NVIC, as it is part of the processor core as provided by ARM. However, PM describes only the general working of NVIC, not the individual signals - i.e. the interrupt lines - which come from the rest of the chip to the NVIC. In other words, for the particular number of individual interrupt sources you have to look elsewhere - namely into the Interrupts chapter of Reference Manual (RM).

Reference Manual is the largest document of the whole collection, which describes the detailed working of the individual modules, and also the working of the chip as a whole. It is usually common to a whole sub-family of STM32, for example, RM0090 pertains to 'F405/407/415/417/427/429/437/439.

Most STM32 (the ancient 'F1 being the notable exception) have a (sparse) matrix between physical pins and individual modules' IO signals. This is being programmed in the GPIO module, after setting given pin to Alternate Function, by setting the appropriate function number into the GPIO_AFR field assigned to the pin. While this general mechanism is described in the GPIO chapter of RM, the particular numbers assigning the particular functions are not described there. So where are they?

This information can be found in the "Alternate function mapping" table in Datasheet (DS). Datasheets contain particular information for several memory-size and package options of one individual STM32 model, or a very small group of models. In the DS, after an introductory overview, and table of memory/peripherals/package arrangement for individual modes, a very general overview of peripherals is given. That is followed by the pinout table and mentioned AF table, then an extensive chapter with electrical characteristics follows, and finally package drawings and numbering/ordering information is given.

The fourth member of the "canonical" set of documents is Errata Sheet (ES), again individual to a small group of STM32 models.

This set of documents (PM/RM/DS/ES) is then augmented by Application Notes (AN), Technical Notes (TN), and various marketing materials. All these are usually listed in the given STM32 model's web-folder.

On the "upper end", the information given in the PM is expanded by ARM's documents (to be found on ARM's webpage) - Technical Reference Manual (TRM) to given Cortex-M core (with accompanying Errata), and the broader Architecture Reference Manual (ARM) to the given ARM architecture version and variant.

Created: 28.August 2021

68. SPI_SR.BSY is unusable

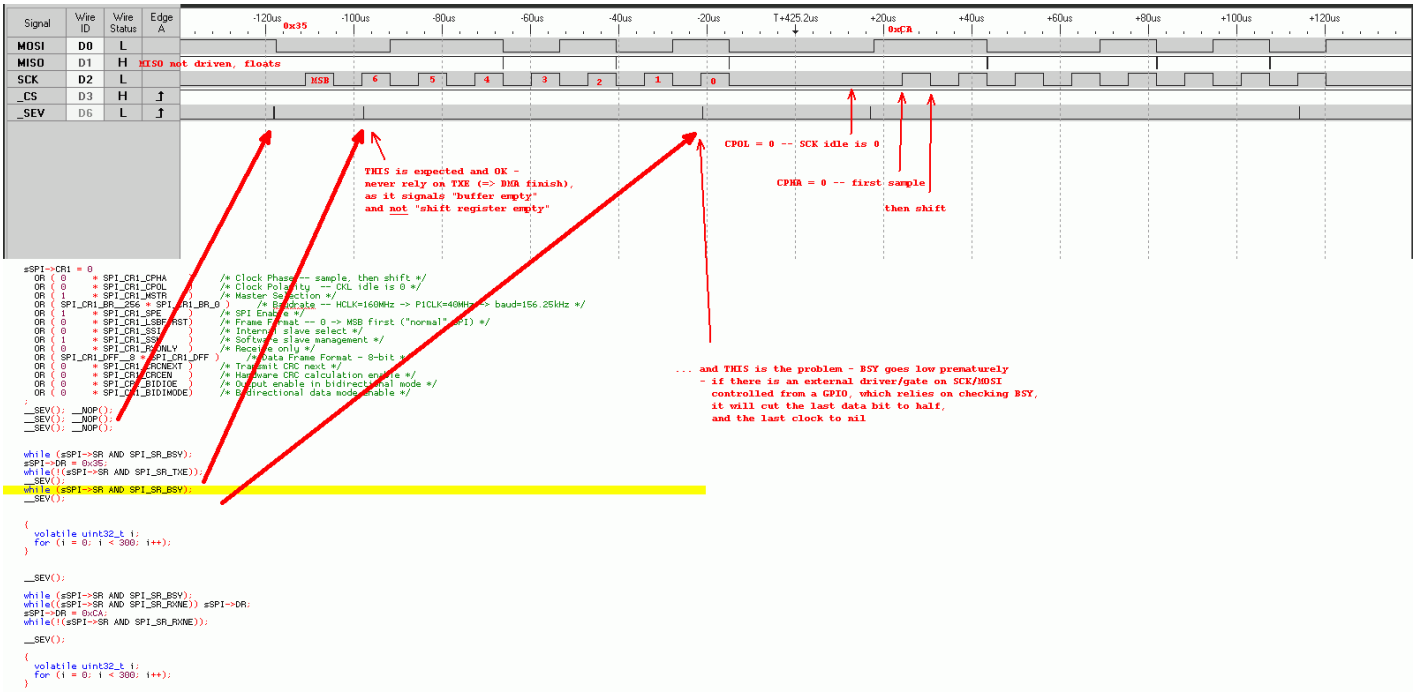
As the SPI module underwent significant changes across various STM32 families, the following information may not be relevant to the particular model you use.

In the SPI module, BSY flag is supposed to signal the period, while the SPI module transmits or receives data. Users often rely on this signal when generating framing signals, e.g. NSS for slaves, and are also often disappointed by the exact behaviour. ST does not document the exact timing of BSY signal for various settings properly.

What follows is anectdotal evidence, that generally, BSY is not suitable for the purpose it is intended for.

Users start transmission by writing data to the SPI_DR register. The general expectation is, that as soon as data are written, BSY goes active. When writing transmission routine for a single frame so that immediately after writing to SPI_DR a loop waits until BSY goes inactive, users are unpleasantly surprised to find out that the program exits the loop immediately (example here). The reason is, that after writing SPI_DR it takes some undocumented time until the frame gets transferred from the holding to the shift register, and BSY gets active only after that.

On the other end of transmission, when last bit of the last frame gets transmitted, at least in case of CPOL=0 and CPHA=0, on an STM32F407, BSY was seen to get inactive early, before the last SCK clock[1]:




```
pSPI->CR1 = 0
  OR ( 0     * SPI_CR1_CPHA      )    /* Clock Phase -- sample, then shift */
  OR ( 0     * SPI_CR1_CPOL      )    /* Clock Polarity  -- CKL idle is 0 */
  OR ( 1     * SPI_CR1_MSTR      )    /* Master Selection */
  OR ( SPI_CR1_BR__256 * SPI_CR1_BR_0 )    /* Baudrate -- HCLK=160MHz -> P1CLK=40MHz -> baud=156.25kHz */
  OR ( 1     * SPI_CR1_SPE       )    /* SPI Enable */
  OR ( 0     * SPI_CR1_LSBFIRST  )    /* Frame format -- 0 -> MSB first ("normal" SPI) */
  OR ( 0     * SPI_CR1_SSI       )    /* Internal slave select */
  OR ( 1     * SPI_CR1_SSM       )    /* Software slave management */
  OR ( 0     * SPI_CR1_RXONLY    )    /* Receive only */
  OR ( SPI_CR1_DFF__8 * SPI_CR1_DFF )    /* Data Frame Format - 8-bit */
  OR ( 0     * SPI_CR1_CRCNEXT   )    /* Transmit CRC next */
  OR ( 0     * SPI_CR1_CRCEN     )    /* Hardware CRC calculation enable */
  OR ( 0     * SPI_CR1_BIDIOE    )    /* Output enable in bidirectional mode */
  OR ( 0     * SPI_CR1_BIDIMODE  )    /* Bidirectional data mode enable */
;
__SEV(); __NOP();
__SEV(); __NOP();
__SEV(); __NOP();

while (pSPI->SR AND SPI_SR_BSY);
pSPI->DR = 0x35;
while(!(pSPI->SR AND SPI_SR_TXE));
__SEV();
while (pSPI->SR AND SPI_SR_BSY);
__SEV();


{
  volatile uint32_t i;
  for (i = 0; i < 300; i++);
}


__SEV();

while (pSPI->SR AND SPI_SR_BSY);
while(pSPI->SR AND SPI_SR_RXNE)) pSPI->DR;
pSPI->DR = 0xCA;
while(!(pSPI->SR AND SPI_SR_RXNE));

__SEV();

{
  volatile uint32_t i;
  for (i = 0; i < 300; i++);
}
```

If NSS signal towards a slave is derived from BSY, the slave may be aborted before the last frame is completed, thus slave may not receive that last frame at all.

One way to work around these problems in SPI transmitter, and avoid using BSY as a signal from which framing is derived, is to use the receiver's RXNE signal instead. This is working also in case when MISO pin is not assigned in the GPIO matrix; however, its timing (which again is not something properly documented by ST) might not be suitable in all CPOL/CPHA combinations.s

Using BSY in slave mode also has its share of issues, see for example the *BSY bit may stay high at the end of data transfer in slave mode* erratum in many STM32 models' errata sheet.

---

1. This "early BSY end" actually is displayed in waveform drawings, e.g. in RM0090 rev.17 *Figure 253. TXE/RXNE/BSY behavior in Master / full-duplex mode (BIDIMODE=0 and RXONLY=0) in case of continuous transfers* and some subsequent figures; however, the description does not mention it.

Created: 4.September 2021
Footnote added: 9.April 2022

69.In 'F746/756, ADC triggering from TIM does not work properly

The ADC design in STM32 allows conversion to be started (triggered) from one of the timers, to allow regular ADC conversions. However, in the STM32F746, under certain APB bus/timer clock setting combinations, this triggering does not work.

The STM32F746/756 is ST's first implementation of the Cortex-M7 core, and as such it probably had seen a rushed development, as witnessed by a significant number of errata. Perhaps the the most annoying of them is *Debugger keeps jumping into interrupt when single stepping*, although to be fair, that comes directly from ARM. Nonetheless, this TIM-ADC error is just another in the list, which may make some users to consider other, newer members of the 'F7 family where these problems have been already fixed, or an entirely different family.

What is maybe even more surprising than the triggering bug itself is, that for some of the triggering sources there is a workaround in that after enabling the apparently unrelated DAC clock in RCC, triggering ADC from TIM starts to work.

While this issue is properly documented in the *Missed ADC triggers from TIM1/TIM8, TIM2/TIM5/TIM4/TIM6/TRGO or TGRO2 event* erratum, it is nevertheless surprising, as witnessed by repeated threads on the forum dealing whith this problem, e.g. here, here, here and here.

Example in Cube - and the fact that "step 3" is after "step 4" and the DAC clock enable - may hint, that a vital part of the workaround is to enable the DAC clock before starting the timer. This is not described by the erratum, but then some of the workarounds in errata are known to be not working, or perhaps work only under undescribed circumstances (such as particular sequencing, as may be the case here).

Created: 11.September 2021
More links, rant about workaround added: 17.February 2023

70.Pins don't work even if set in GPIO (due to nonzero default values in 'L0/'L4/'Gx/'H7 GPIOx_MODER)

Sometimes, users who were used to work with models in some of the STM32Fxx family, especially employing normal register-level approach, when migrating to 'L0/'L4/'Gx/'H7[1], find with surprise, that the pins initialized in GPIO using an already well-proven routine or method, simply don't work at all.

The reason is, that users often write code such as

```
GPIOB->MODER |= (0b10 << (2 * 10));  // set PB10 to AF
```

which relies on the fact, that in 'Fxx, most GPIOx_MODER fields are set to 0 after reset. Thus, in these families, after reset, most pins are set to Digital Input mode[2] [3].

However, in the 'L0/'L4/'Gx/'H7 families, the default value of GPIOx_MODER fields for the majority of pins is 0b11, i.e. almost all pins are set as Analog. This presumably is aimed at lowering the consumption of unconfigured pins, as in this setting, the Schmitt trigger digital input buffer is disconnected, so should a pin be unused and unconnected, any noise induced to it does not flip state of any internal digital gate.

ORing any value to a field already set to all-ones of course makes no change, so the GPIO initialization routine for these families has to be modified accordingly.

---

1. For some unknown reason, reset value of GPIOx_MODER for most pins in 'L1 is 0b00, as is in the 'Fxx families. Maybe it is related to the fact that 'L1 was the first low-consumption STM32 family, and in fact the GPIO structure used ever since was introduced with this family.

2. Exceptions to this are generally the pins related to JTAG/SWD, but there may be more in particular models. One of such exception is PA4 in the 'L0 family, which is set by default as (digital) Input, whereas all other pins (except SWD pins) are set as Analog.

3. The ancient 'F1 family has a different GPIO structure, we won't be discussing that here.

Created: 18.September 2021
Remark on PA4 in 'L0 added: 24.March 2022

STM32 gotchas

71.RTC does not work - because of RCC_APBxENR.RTCAPBEN bit in some models

RTC in most STM32 is in the backup power domain (BD), i.e. powered from the switch between VDD and VBAT (controlled by the power voltage detector). To protect registers in the BD from inadvertent changes, there's a protection bit PWR_CR.DBP, which is cleared by reset, and has to be set by user before accessing any register in BD.

So, to enable the RTC, one has to go through a somewhat involved procedure - first enable PWR clock in RCC, then set PWR_CR.DBP, make sure that clock chosen for RTC (LSE or LSI or HSE/N) is up and running, make the selection of that clock in RCC_BDCR and enable RTC using the RTCEN bit in that same RCC_BDCR.

This gets RTC running in most STM32 models, but not in all. The problem is, that somewhere around 2016, ST started introducing a new enable bit, RCC_APBxENR.RTCAPBEN, which has to be set, otherwise there's no access to RTC registers from the processor.

[rant mode on] Btw. a nice way to notify the developer/user community of this kind of changes would be the Technical Updates, ~~but ST stubbornly refuses to renew them. They supply us with ample marketing stuff without material content, instead~~ finally renewed in 2024. [rant mode off]

The exact purpose of this bit is unclear. Apparently, it gates access to the non-backup part of RTC which interfaces to the APB bus, but the only conceivable reason to do that is to spare some power. It's hard to tell how much, as there's no APB-part-of-RTC-specific current in the current consumption tables in datasheets[2].

To increase confusion, ST added this bit even to new models of already existing lines[3]; for example, this bit is present in 'F410, 'F412, 'F413/423, but it is absent in any of the "upper-line" 'F4 and, more notably, it's absent also in 'F401 and 'F411. Similarly, it's absent in the "original" 'F746, but it's present in newer 'F722/723 and 'F767; absent in "original" 'L476 but present in 'L496, the 'L4x2/4x3 and the 'L4+. And it's also present in all the newest lines, i.e. 'H7/'G0/'G4.

ST obviously attempted to mitigate consequences by making this bit being set by default after reset. But they failed to be consistent in this, too: in the 'F4 line, it's not set in 'F410; in the 'F7 it's not set in 'F722/732; in the new lines, it's not set in 'G0.

To add insult to injury, in the CMSIS-mandated device headers, for the 'F4, the symbol for this bit is called - logically and consistently - RCC_APB1ENR_RTCAPBEN, whereas in the 'F7, it's called RCC_APB1ENR_RTCEN, incorrectly and potentially confusing it for the RTCEN bit in RCC_BDCR.

---

1. In the ancient 'F1 family, which has a different backup-domain/RTC arrangement from all other STM32, there already was a similar enable bit, RCC_APB1ENR.BKPEN.

2. Together with RCC_APBxENR.RTCAPBEN bit, there's also a new low-power equivalent of that bit, RCC_APBxSMENR.RTCAPBSMEN or RCC_APBxLPENR.RTCAPBLPEN depending on the particular model ([rant mode on] as naming of low-power RCC enable registers is a mess, too, and so is their relationship to the "run-time" enable registers and default content [/rant mode off]). Purpose of this bit is even more unclear, as in low-power modes there's no processor available which would access the RTC through its APB interface, anyway.

3. ST otherwise has a very reasonable policy not to make "enhancements" to peripherals in a particular family (except when fixing what's clearly a hardware bug). For example, all the 'F4 models including the abovementioned "junior" models share the same relatively old SPI module which was in the first 'F4 - the 'F405/407, even if when most of other 'F4 models came out, the SPI featuring data packing had been already available on 'F0/'F3.

Created: 25.September 2021
'F1-related footnote and Technical Updates renewal added: 24.February 2024

72.No DMA to/from GPIO in 'L0/'G0 (i.e. where the Cortex-M0+ core is used)

Developers on various STM32 sometimes use DMA transfers triggered by timer, where data are moved from memory to GPIO (either GPIO_ODR or, if atomic access is desired, GPIO_BSRR), or vice versa, from GPIO_IDR to memory. The purpose is to output random waveform onto one, or many pins at once, or conversely, sample the input state of one or more pins. As timer-triggered DMA can originate also from a timer capture, this also provides a way to do such waveform output or pins sampling upon external signal. There is even an application note - AN4666 - demonstrating this technique.

However, this method fails on the STM32L0xx and STM32G0xx families. Why?

These families are built around the ARM Cortex-M0+ processor core. When ARM designed this core, they attempted to improve on the minimalistic Cortex-M0 core (as the designation shows), and one of the improvements was that they included directly to the core a dedicated port called IOPORT, GPIO modules to be connected onto this port.

It's not clear what are the benefits of this design, but one of the consequences is, that GPIO mapped onto IOPORT are available only from the processor core, and there's no way all other potential busmasters in the system - i.e. DMA - can access them.

Created: 2.October 2021

73.Always read the Errata - and not only those directly given for your STM32 model

The Errata form a vital part of the documents stack needed to understand and be able to use a particular STM32 model.

While the Reference Manual deals with the details of the design of particular modules in the mcu, this is mostly a design target description. Inevitably, when implementing this design target, errors occur, resulting in the actual chip behaving differently from the RM's description - and Errata are there to describe these deviations. Sometimes, an erratum is more a documentation clarification rather than error description.

It is not rare to see a developer to struggle with an inexplicable behaviour, just to find out later, that this is consequence of a known deviation from the RM described in the Errata. Therefore, it is a good practice to read the Errata, before starting work on any particular mcu model, and also to follow any changes and updates to those Errata throughout the work.

However, sometimes it pays out to read Errata not only to the given STM32 model, but also to other similar, maybe older models. The reason is, that ST does not update the Errata very timely, especially for "marginal" models which don't see much usage. Even if there is a known problem with an IP module which is present in several models of the same STM32 family and often also in other families, ST does not rush to update all related Errata at once.

Sadly, sometimes it even happens that a known bug of a given IP does not propagate into Errata of a different STM32 family using the same IP at all. For example, in this particular case, an erratum known for the (rather complex and hard to use) I2C module in the 'F1 family did not propagate into errata of 'F4 family using the same I2C IP. According to the 'F1 erratum changelog, the given problem was already known in 2011, the year when 'F4 family was introduced.

Created: 9.October 2021

74.Write Protection (WRP) granularity may be non-uniform (in large-FLASH 'F1/'F0/'F3)

Write Protection (WRP) of FLASH is a feature, which prevents accidental overwrite of portions of FLASH from both program running in mcu, and external programmer. It is useful for example to protect a bootloader from being inadvertently corrupted, resulting in the device controlled by the mcu to be "bricked" from the user point of view.

The various STM32 families approach this problem in different ways, but WRP always works on portions (sectors, pages - terminology varies across families) of FLASH, each portion being several kilobytes large - this is called WRP granularity. Whether some FLASH portion is write protected or not, is controlled by Option Bytes dedicated to WRP.

Granularity of WRP in the historically first 'F1[1] family is 4kB (4 pages in lower-end 'F1 and 2 pages in higher-end 'F1, but the pages' size is different so that it always results in 4kB WRP granularity). Each "granule" is write protected, if the corresponding bit in the WRP Option Bytes is programmed (= cleared, as is the usual confusion with the Option Bytes). However, the largest-FLASH 'F1 have 256kB and 512kB FLASH, which would mean 64/128 WRP "granules", while there are only 4 WRP Option Bytes i.e. 32 bits; so how does this work?

Easily: in these 'F1, the first 31 option bits write protect the first 31 WRP "granules", i.e. the first 124kB in 4kB granularity; and the 32th option bit simply write protects whatever portion of FLASH is left (i.e. size of the last "granule" is 132kB/388kB).

This same scheme is used also in 'F0 and 'F3. While their respective Reference Manual do describe this at the description of the WRP Option Bytes, they fail to mention it in the dedicated *Write protection* subchapter of the *Embedded Flash memory* chapter, leaving the reader confused.

In 'L1, there are truly uniform WRP "granules", at the cost of implementing up to 16 WRP bytes (i.e. 4 WRP words, 128 bits).

The 'F2/'F4/'F7 families use a different scheme, where WRP copies the non-uniform erase sectors. As these sectors are mostly large (up to 128kB), there are fewer than 32 of them even in the largest-FLASH models.

The FLASH in 'L0 is different from all other models (maybe consequence of the effort to implement true EEPROM as a portion of it). Its WRP is an alternative option to the more complex PCROP protection, with uniform granularity and 48 related option bits.

In 'L4, again a different scheme was introduced: instead of bit-mapping WRP "granules", two areas (consisting of an integer number of "granules") can be selected for write protection using two pairs of "boundary" registers within the Option bytes. This scheme is used also on 'G0 and 'G4.

---

1. Contrary to other families where descripiton of FLASH is part of the Reference Manual, 'F1 has the FLASH and its features described in a separate document, Programming Manual PM0075. In this family, confusingly, write protection of first few pages of FLASH is also switched on, whenever Readout Protection is switched on.
However, there's an exception to this exception, too: the largest-FLASH STM32F103xF/xG (>=512kB FLASH), aka *XL-density*, are described in PM0068, and they don't have this quirk.

Created: 16.October 2021
Note about the PM0068-related 'F1 added: 23.September 2023

75.Unexpected ETH interrupts after longer uptime (due to MAC management counters interrupts being enabled by default)

The ETH module is rather complex, and among other things it contains a set of counters counting good and bad frames in each direction. This sub-module is called MAC management counters (MMC). These registers serve to debug/benchmark the ETH communication in a given setup. Besides the counters themselves there is also a set of registers enabling interrupt, when a respective counter reaches half its count (i.e. 2^31).

Now the first confusing thing is, that these registers disable (in Synopsys's and ARM's lingo, "mask") given interrupts when their respective bits are set.

And by reset default these registers are cleared, so the related interrupts are enabled. This is again confusing, as in all other modules all interrupts are disabled by default.

It's also not that obvious during usual debugging that these interrupts are enabled - they kick in only after a prolonged time, as at least 2^31 frames has to be exchanged.

(On STM32 Forum, user Piranha provided a description of the variants of the problem for several families, and proposed respective fixes).

Created: 23.October 2021

76.RTC digital calibration does not apply to wakeup period (with a specific wakeup source selection)

RTC in STM32 can compensate for inaccuracy of the used crystal by so called digital calibration. This is achieved by a unit, which counts pulses in parallel with the main RTC counter, and when a set number of pulses is counted, it removes (gates) a clock pulse or inserts an extra clock pulse to the main counter, while resetting itself.

The exact working of this unit, and where it is placed within the clock chain of RTC, varies from family to family; at one point (in some 'L1 and in 'F4/'F3) there even were two such units ("coarse" and "fine") present simultaneously.

RTC has also a Wakeup unit, which allows periodic interrupt intended (as its name suggests) to wake up the mcu put into some of the low-power modes (sleep). This is basically a configurable auto-reloading counter, which also has a configurable clock source, selecting and prescaling clocks coming from various points of the RTC clock-chain.

Some of the wakeup unit clock sources are tapped from before the calibration unit. If such clock source is used, wakeup is then "not precise", as it does not include the corrections from the calibration unit.

This issue was brought up here.

Created: 30.October 2021

77.ETH does not work with certain APB2 clock settings

This issue has been brought up here and here.

STM32F4xx for connection between ETH/MAC and external PHY supports both the MII and RMII standard (the latter requiring less pins, at the expense of 50MHz clocking of the interface instead of 25MHz). This is accomplished by a relatively small "insert" between pins and the ETH/MAC module itself, which demultiplexes the RMII signals onto MII signals and also generates MII clocks from RMII clock, so that the ETH/MAC module is entirely unaware of the RMII/MII "duality" and works purely on MII interface.

This "insert" is controlled by a single control bit, SYSCFG_PMC.MII_RMII_SEL. SYSCFG is a unit which does not have any inherent functionality, it's just a collection of registers to "gather" such "orphan" control bits from various "intermediate" modules too small to grant them status of standalone module (i.e. assign a memory range for registers, assign clock enable bit(s) in RCC, etc.).

SYSCFG is clocked from APB2, i.e. if APB2 prescaler is set to a high value, it may take so many system clocks until a write to its register actually results in change of the related internal signal.

In this particular case, the Cube "library" code switched MII/RMII by writing SYSCFG_PMC.MII_RMII_SEL, and immediately after that initiated an ETH/MAC reset by writing into a self-clearing register bit in ETH/MAC itself. ETH/MAC is clocked directly from system clock, the MII/RMII clocks have not been properly switched in the RMII/MII "insert" by the time the ETH/MAC reset was initiated, thus the reset did not work properly, resulting in the whole (rather complex) ETH/MAC module not to work properly.

Workaround consisted of inserting a dummy read from SYSCFG before initiating ETH/MAC reset, which is a method to ensure that the write to SYSCFG register was actually finished.

This is an example of subtle errors stemming from timing details within the complex SoC fabric of STM32.

Created: 6.November 2021

78.Using TIM input channel filter may produce confusing results (if fed with periodic signal)

The TIM channels at their inputs incorporate a digital filter. This helps to filter out glitches and other unwanted signals, especially if the signal source is "real world" rather than other digital circuitry. The input signal is then used either to Capture the timer's counter current state, or as an input to the Slave-Mode controller.

The filter samples the input signal continuously, and if the input level is about to change, only if it sees N samples of this new level consecutively, the filter outputs this new level, otherwise its output does not change. N and the sampling frequency can be set for each channel separately in TIMx_CCMRx.ICxF.

N ranges from 2 to 8, and the sampling frequency is either equal to the internal clock of the timer, or is a fraction (1/2 to 1/32) of output of a prescaler called DTS (dead-time and sampling clock), derived from the same internal clock of timer, being equal to that clock or divided to 1/2 or 1/4, according to TIMx_CR1.CKD.

This works quite well for spurious oscillations coming from bouncing mechanical contacts of switches. However, if the input signal changes periodically, e.g. due to electromagnetic interference from a neighbouring circuit into a high-impedance circuit of a disconnected switch, the results might be strange to confusing.

If the input frequency is an exact multiple of the sampling frequency (which may easily happen if the interfering signal is derived from the same mcu clock than the timer's clock), the filter's output may be a steady level. For example, if the closed switch results in low input, but after opening the switch the interference would cause the filter sample low at every its sampling period, the net output is the same as if the switch would never open.

An even more confusing result stems from an input signal which period is slightly different from exact multiple of the sampling frequency. In such a case, the output of filter would change at a low rate, corresponding to the beat frequency between sampling and input signal, possibly misleading the program/observer to believe that the input signal is of a much lower frequency than it really is.

The corollary is, that while the input filter on timer channels may be a good tool to filter out spurious oscillations at signal's edges; where periodic input signal is expected, an external analog low-pass filter has to be included into the signal path.

Created: 13.November 2021

79.If TIM is set to downcounter, 0% PWM is not possible

The STM32 timers are primarily up-counters. In that capacity, when a channel is set to Output Compare and PWM1 mode is selected (PWM2 being simply inversion of PWM1), setting respective TIMx_CCRx register to 0 results in 0% PWM and setting it to value higher than is content of TIMx_ARR results in 100% PWM.

Some "better equipped" timers however feature also an up-down ("center aligned") counting mode. This in case of multiple channels being set to PWM, allows them to have their pulses "spread out" around one center point, for whatever that might be needed.

In this mode, a period lasts twice the number of clocks set in ARR. The designers - again for unknown reason - decided, that they need also the generated PWM pulse to be an even number of clocks. Therefore the condition to change the output level is different for counting up and counting down: if TIMx_CCRx is set to N, while counting up, output level is changed when TIMx_CNT changes from N-1 to N; whereas while counting down, output level is changed when TIMx_CNT changes from N+1 to N. Or, as in RM the TIMx_CCMR1.OC1M description for PWM1 mode states:

In upcounting, channel 1 is active as long as TIMx_CNT<TIMx_CCR1 else inactive. In downcounting, channel 1 is inactive (OC1REF=0) as long as TIMx_CNT>TIMx_CCR1 else active (OC1REF=1).

In the marginal case, when in up-down (center-aligned) mode TIMx_CCRx is set to 0, output duty is 0%. It may appear, that when downcounting, output level should change from 0 to 1 when counting from 1 to 0, but - again somewhat arbitrarily - the direction bit changes at the same moment, making the period while CNT==0 being already in the upcounting portion, thus preserving the output level at 0.

Timers, which have the up-down (center-aligned) mode, already have the downcounting mechanism, so it's logical that they can also be set to down-only ("edge-aligned", as opposed to "center-aligned") counting mode. The consequence of the described mechanism is, that if in that mode TIMx_CCRx is set to 0, the transition of TIMx_CNT from 1 to 0 indeed makes the output to go high for one clock period. In other words, there is no way to set 0% PWM in the downcounting mode.

This behaviour is explicitly documented in *Downcounting configuration* section of *PWM mode* subchapter of the *TIM* chapter in RM, but it's surprising nonetheles.

Created: 20.November 2021

80.When processor stopped in debugger, peripherals (and DMA if used by those peripherals) keep running

As debuggers allow to stop code execution (whether "manually" upon pressing the appropriate "button", or at a previously set breakpoint) and display a static snapshot of memory (and peripheral registers') content, users sometimes expect that the whole mcu is in a completely stopped state. They may find with surprise, that a single step, or even just making the debugger to re-display content of memory or peripheral registers, results in large changes.

Explanation again lies in the SoC character of these chips. Debugger is part of the processor, and stopping execution means exactly that: the processor stops processing instructions. The rest of the chip still continues to do what it is set to do: clocks keep clocking, memory system keeps working (e.g. prefetch/caches keep prefetching/linefilling/evicting until they stop due to processor inactivity), peripheral keep working[1].

Consider for example a fully configured UART, with its Rx set to trigger DMA, transferring received data to memory. As long as characters keep arriving at the UART_Rx pin, the UART receiver deserializes them, upon completion triggers the DMA which then moves them to the memory. This happens regardless of whether the processor is stopped by debugger or not.

As with some peripherals this would lead to surprising and hard to debug states, STM32 provide a mechanism to stop certain peripherals - usually timers, CAN, I2C, watchdogs - while the processor is stopped by debugger. This can be done by setting respective bits in DBGMCU_APBx_FZ or similarly named register in the MCU submodule of DBG module (ST is not good at maintaining register naming consistent across STM32 models).

This may also lead to a "reverse" surprise: some debuggers may have settings which allow to set bits in this register when connecting. If user of such debugger is not aware of that feature (or has used it, setting some of the bits, and later has forgotten about it), it may find by surprise that some peripheral is not working while the debugger is in stopped mode.

---

1. Don't confuse running peripheral with interrupt service routine (ISR) which is intended to serve it. When processor is stopped, the running peripheral keeps setting flags resulting in interrupts, but the ISR cannot be executed (so its actions cannot have visible consequences) as the processor is stopped. When single-stepping, the interrupt flags being set would result in ISR being entered immediately; this would in most practical cases prevent single-stepping non-interrupt code thus defeat the purpose of single-stepping, which is why debuggers have a switch in hardware wich prevents interrupts to be entered during single stepping (except in the flawed versions of Cortex-M7 core).

Created: 27.November 2021

81.Crash due to stack content getting corrupted (DMA writes to buffer improperly allocated on stack)

Users sometimes inadvertently allocate a memory-side buffer for DMA as a local non-static variable, i.e. on the stack. They expect this buffer to be filled by a peripheral-to-memory DMA, and then process the data in an interrupt invoked upon completion of DMA.

If they do this in a fuction which subsequently ends while DMA is still running, which is the usual case, this may lead to a surprising crash or any other unexpected program behaviour.

Naturally, the stack gets reused in following function calls, so the running DMA will then corrupt the content of stack.

Counterpart to this issue, related to Tx rather then Rx with perhaps less dramatic consequences, is described here.

One of the instances of this gotcha reported here.

Created: 4.December 2021

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 ot
#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  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 regi
    // 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/
  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 r
    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
  // 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
  RTC->PRER = (uint32_t)0x7;  // sync
  RTC->PRER |= (uint32_t)(0x3 SHL 16);  // async -- RTC_PRER register must be written i

  // 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

  // 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
  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 cc
    // so this delay makes sure that "real" TR will change
```

```
    // 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
      __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" cond
    // 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

STM32 gotchas

83.Cannot access BKPSRAM in CubeProgrammer (unless register values are manipulated)

This article is a summary of a related discussion on STM32 Forum. Thanks to ST employee with nick Mike_ST for the support!

The backup SRAM (BKPSRAM) is a portion of few kilobytes of static RAM, which is supplied from the VBAT power domain, thus retains its content even if VDD is completely removed. It is present only in some STM32 families ('F2, 'F4, 'F7, and in 'H7, where it is confusingly called BKPRAM).

As with other circuits in the VBAT (backup) domain such as RTC (with backup registers and tamper protection) and LSE, write access to BKPSRAM is protected by PWR_CR.DBP bit. However, the backup domain reset through setting RCC_BDCR.BDRST does not affect BKPSRAM content, and after powering down and up the backup domain (by removing both VDD and VBAT) the BKPSRAM content is random.

BKPSRAM shares also some characteristics with FLASH - it is essentially non-volatile (provided VBAT is present and the BKPSRAM-associated low-power regulator is enabled), it is protected by RDP in the same way as FLASH is, it's even cleared when RDP changes from Level 1 to Level 0.

It might be a reasonable expectation, then, that programming tools - such as ST-supplied CubeProgrammer - support writing and reading of BKPSRAM in the same way as writing and reading FLASH, for purposes of both development and production. Turns out, this is not entirely the case.

While CubeProgrammer allows to read and write any area of the memory space of target STM32, thus including BKPSRAM; it also maintains all the registers in its reset state. As access to BKPSRAM is subject to enabling its APB clock (by setting RCC_AHB1ENR.BKPSRAMEN, which is zero after reset), to be able to read BKPSRAM in CubeProgrammer, this bit has to be set "manually". As said, CubeProgrammer allows to access any address in memory space, this includes also RCC registers, so this can be accomplished for example for STM32F405/407 by selecting an area starting e.g. at 0x4002'3800 (this is starting address of the RCC registers area), selecting the word at 0x4002'3830 which is RCC_AHB1ENR (this has default value of 0x0010'0000 as CCM RAM's clock is enabled after reset), and writing the value 0x0014'0000 into it. After this, selecting the BKPSRAM area at 0x4002'4000 displays the BKPSRAM content.

Should we want to write to BKPSRAM, we would need to enable PWR clock first (by setting RCC_APB1ENR.PWREN, i.e. writing 0x1000'0000 into 0x4002'3840), then enable backup domain write by setting PWR_CR.DBP (writing 0x0000'4100 into 0x4000'7000), and to retain written values in VBAT mode, the BKPSRAM's low-power regulator has to be enabled by setting PWR_CSR.BRE (writing 0x0000'0200 into 0x4000'7004).

For other STM32 models, the particular register addresses and values may be different, but the principle remains.

Created: 18.December 2021

84.GPIO pin does not output enough current (if set to low GPIO_OSPEEDR setting)

For each IO pin, the GPIO module has a set of registers, which determine their exact behaviour. One of these is the OSPEEDR register determining the "output speed" of the pin (when set as Output, or when set as Alternate Function and the controlling peripherals turns it to Output). More precisely, OSPEEDR influences the slew rate of the output signal when switching, by activating one or more MOSFET output transistors of varying dimensions (hence varying resistance). The resulting slew rate of the signal's edge is then given by multiple of the resulting total channel resistance and the loading (usually parasitic) capacitance on the given pin, connected tracks and other circuits.

While some high-speed circuits demand to have very high slew rates, setting all pins indiscriminately to high slew rate is a bad idea, as these generally result in increased high frequency emissions, transmission-line effects such as reflections, increased signal crosstalk and effects arising from imperfect ground. STM32 thus generally have by default the OSPEEDR settings set to the lowest range (there are individual exceptions, e.g. the SWI/JTAG pins).

However, OSPEEDR setting has also a different, maybe less expected effect: as OSPEEDR effectively influences total output resistance, it also means that it also determines the maximum output current to be drawn for given pin. The DS does not account for this effect and for most STM32 (except some pins like those which are supplied through the backup-domain switch) it gives an universal figure of maximum output current of 8mA or 20mA at degraded output voltage, without mentioning that this current is achievable only at higher OSPEEDR settings.

This may come as a surprise if the user attempts to supply higher current from an IO pin, e.g. to drive a LED in an optocoupler.

Created: 25.December 2021

85.Not all GPIO pins are created equal

Most of the IO pins in STM32 have the same structure, controlled by GPIO registers:

- direct or switched input to analog features related to the pin
- Schmitt-trigger digital input, which can be switched off if pin is set to Analog mode
- totem-pole Push-Pull output, where number of parallel-connected transistors can be chosen to select slew rate, and the upper transistors can optionally be held off for Open-Drain configuration
- individually switchable pullup and pulldown, both consisting from a true resistor, and a switching transistor
- ESD protection

For most pins, the characteristics are roughly the same: the Schmitt trigger has a leakage of max. 1uA; the output stage can source cca 8mA (or 20mA with degraded output voltages) at high enough OSPEEDR setting, driving a few MHz signals at lowest OSPEEDR setting and frequencies up to the maximum system clock at higher OSPEEDR setting; and the nominal value of pullup and pulldown resistors is cca 40kΩ.

However, exceptions exist, sometimes leading to nasty surprises in a late stage of design, after boards have been designed and populated, and just some pin does not behave as expected.

There are obvious differences in the ESD protection between 3V-only pins (generally marked as TTx in the pinout table in datasheet) and 5V-tolerant (FTx) pins; with the caveat of some FT pins turning non-5V-tolerant, as soon as they are switched to Analog mode (e.g. PA4/PA5 in 'F4, when they are switched to be DMA outputs).

Among the less obvious and thus potentially more surprising exceptions are:

- higher IO leakage of FT pins when voltage on them exceeds their respective supply voltage
- asymmetrically higher (low-level only) current on FTf pins, allowing Fm+ mode of I2C (and high-current IR LED connection)
- asymmetrically lower maximum current on pins which go through the backup-voltage switch (usually PC13-PC15, but may be others)
- surprising and quite poorly documented pullups/pulldowns on the USB-C-PowerDelivery-related pins on 'G0 and 'G4; these pins also tend to have significantly higher leakage
- significantly lower values (nominally 10kΩ) of pullups and pulldowns of PA10 and PB12 in 'F2/'F4/'F7 - related to these pins being OTG_ID pins

Generally, when designing a new circuit, it pays out to read the datasheet to given STM32 carefully, including the GPIO characterization data, even if it is undoubtedly a boring reading.

Created: 1.January 2022

86.'G0/'G4 USB-C Dead Battery conundrum (pin influenced by other pin, unexpected pulldowns)

The STM32G0xx and STM32G4xx families feature USB-C Power Delivery support. Part of this is so called Dead Battery support, where a battery-powered device signals readiness to receive power even if it is not powered/running, due to discharged battery. This is simply a nominally 5k1 resistor to ground.

In 'G0/'G4, this resistor is present in the UCPDx_CCx pins (see *Pin assignment and description* table in Datasheet to identify which pins have this functionality). It is connected through a transistor, gate of which is brought out to respective UCPDx_DBCCx pin. The purpose of this arrangement is for the users to be able to disable the 5k1 pulldown, by tying the UCPDx_DBCCx pin low (otherwise, if dead-battery support is desired, UCPDx_DBCCx pins should be connected to their respective UCPDx_CCx pins).

Remember, that this functionality has to be present even while the mcu is not running, so this is implemented purely in hardware. Once the application is running, it can disable the pulldown by setting UCPDx_STROBE bit in SYSCFG1_CFGR1 in 'G0/UCPDx_DBDIS in PWR_CR3 in 'G4.

There are several confusing consequences from this, for users who are not aware of this functionality:

- presence of unexpected pulldown during reset may cause unexpected behaviour of hardware
- changing level on UCPDx_DBCCx switches the pulldown, potentially influencing voltage levels on weakly driven UCPDx_CCx pins
- pulldown is present also in the 'G0x0 "Value line" models, where USB-C is formally not present

Assorted threads from the STM32 forum dealing with the problem: here, here, here, here, here, here, here.

Created: 8.January 2022

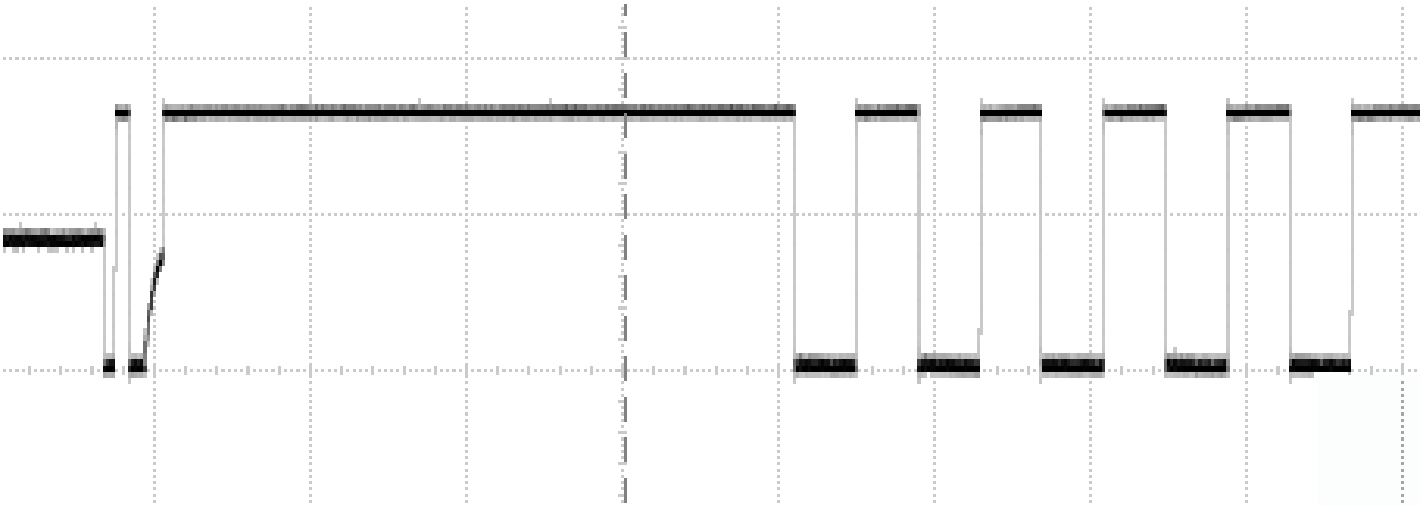
STM32 gotchas

87.UART Tx is not driven immediately after enabling UART

[EDIT February 2023]
While investigating an issue with Cube generating a spurious pulse on UART_Tx pin during initialization with Tx inversion enabled, I realized that there's a difference between the older and newer version of UART with regard of the issue described below. The newer UART - which appeared in the 'F0/'F3 (and can be easily distinguished from the older as having separate data registers - UART_TDR for Tx and UART_RDR for Rx):

- does enable the UART_TX pin directly after setting UART_CR1.TE, even if UART_CR1.UE is still zero
- does not exhibit the one-bite-wide delay with floating Tx pin, see waveform below
- the stopped UART when UART_BRR <= 7 or UART_BRR >= 0xFFF8, and surprising baudrate when 8 <= UART_BRR <= 0xF, is still present as described in the second part of the original text.

The following waveform is taken from an STM32F030 in the same setup as the 'F411 in the original text (with resistive divider on the Tx pin pulling it to "mid-level", when threestated), with the same code as in the first experiment below. Compare it to the waveform for 'F411 to see that the initial one-bit-delay is missing (repaired?). The initial manually generated "marker" pulse and the subsequent threestate period until TE gets set i.e. Tx starts to drive, are longer than in case of 'F411, as the 'F030 run at half the clock frequency and has less efficient instruction set.



Following is the **original text** of this gotcha, which was written based on observation of behaviour of the **older version of UART**.

---

*This minor issue has been spotted and reported by user with nick **il-2** on a russian-language forum.*

In STM32, if a pin is set to Alternate Function in GPIO_MODER and assigned to given peripheral in GPIO_AFR[], that pin's output circuitry is controlled from the given peripheral. For example, if pin is assigned to a timer, if given timer channel is set to Input Capture, the pin's output circuitry is disabled (i.e. it functions as an input), if it is set to Output Compare and enabled, it's output circuitry is enabled and pulls up or down depending on the state of given channel (or pulls only down if set to Open Drain in GPIO_OTYPER).

Most of the peripherals have one or more "enable" bits, which directly influence the respective pin's output state. In UART, transmission - thus Tx pin's output state - is driven by UART_CR1.TE, but UART_CR1.UE has to be enabled, too (UE presumably controls the overall clocking of the UART). So, one would expect, that as soon as UART_CR1.TE is set, the given UARTx_TX pin will be driven high (to UART idle state).

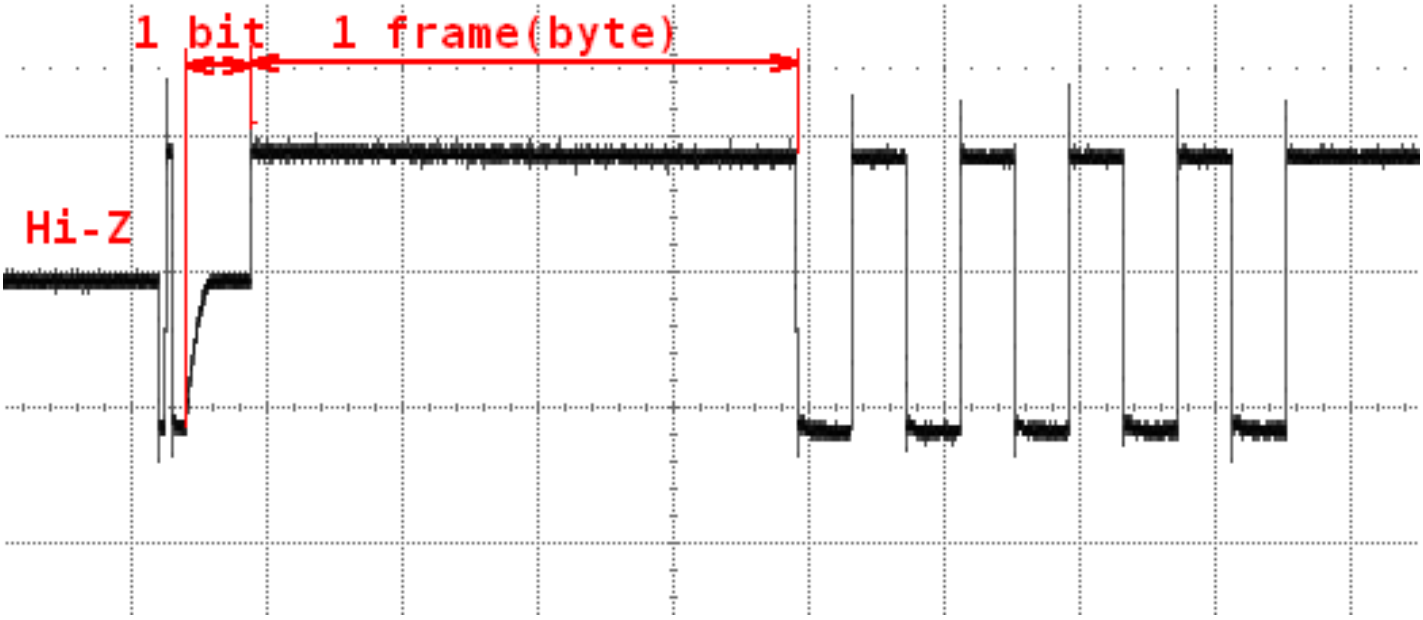
But this is not how UART behaves. The following code is aimed at testing that ('F411 is used, PA9 as USART1_TX loaded by a voltage divider so that the Hi-Z state shows as cca 2V; USART1's clock in RCC and PA9's GPIO_AFR[] have been already set):

```
    GPIOA->MODER = (GPIOA->MODER & ~(3 << (2 * 9))) | (1 << (2 * 9));  // generate a marker by setting PA9 to Output
    GPIOA->ODR |= (1 << 9);                                            // pulse PA9
    GPIOA->ODR &= ~(1 << 9);
    GPIOA->MODER = (GPIOA->MODER & ~(3 << (2 * 9))) | (2 << (2 * 9));  // set PA9 as AF, i.e. place it under control of USART1

    USART1->BRR = 0x20;  // set baudrate

    USART1->CR1 = 0      // enable Tx and enable USART
      | USART_CR1_TE
      | USART_CR1_UE
    ;
    (void)USART1->SR;    // clear TC flag
    USART1->DR=0x55;     // transmit S10101010P -- in reality this gets transmitted after the documented 1-byte delay
```

and this is how the Tx pin waveform looks like:



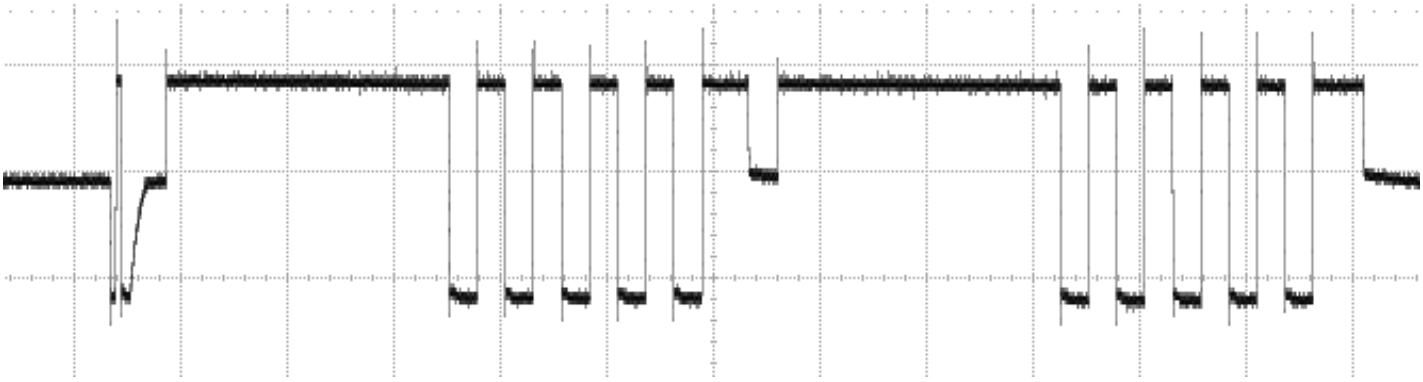
As the waveform shows, after TE is enabled, there is **one bit long delay, during which Tx pin is in Hi-Z state**, before it pulls high. (There is also one frame length Idle transmitted before the first frame - this feature **is** documented in RM.)

As the discussion in the source forum points out, the output pin is not driven directly by the TE bit, rather, it's controlled from the baudrate generator.

There's also a **similar delay after TE is cleared**, i.e. Tx pin does not go into Hi-Z immediately. However, this delay appears to be somewhat **shorter, maybe half-bit long**. For testing that, the above code has been amended by: resulting in the following waveform (the delay in question exhibits itself as a prolonged stopbit, before the pin goes threestate):

```
while ((USART1->SR & USART_SR_TXE) == 0);
while ((USART1->SR & USART_SR_TC) == 0);
USART1->CR1 = USART_CR1_UE;     // clear UE, Tx pin should go to threestate but does so only after delay added to stopbit

USART1->CR1 = 0                 // enable transmitter again, which results in the 1-bit long Hi-Z delay
  | USART_CR1_TE
  | USART_CR1_UE
;
(void)USART1->SR;  // clear TC
USART1->DR=0x55;   // again this gets transmitted after the documented 1-byte delay
while ((USART1->SR & USART_SR_TXE) == 0);
while ((USART1->SR & USART_SR_TC) == 0);
USART1->CR1 = USART_CR1_UE;  // again Tx pin goes to Hi-Z after delay added to stopbit
```



As an additional gotcha, the baudrate generator has quirks, if UART_BRR is set to some extreme values:

- if UART_BRR <= 7, there's no baudrate generation, i.e. after enabling UE/TE, Tx pin remains threestated indefinitely. This may quite easily occur in environments where baudrate is expected to be set "afterwards", e.g. from USB; or if baudrate is set from a possibly corrupted source, e.g. an EEPROM.
- similarly, if UART_BRR >= 0xFFF8, there's also no baudrate generation
- if 8 <= UART_BRR <= 0xF (with OVER8 = 0), baudrate is surprisingly low; for example with UART_BRR = 8 it's (UART_CLOCK / 0x8000)

While these features are unlikely to be important in the vast majority of usual applications, there may be niche applications where behaviour other than "expected" may cause unpleasant surprises. For example,

Lessons to be taken from these observations:

- UART_BRR is to be set to proper value early
- if Tx pin state has to be high at all times when there's no transmission, and UART_CR1.TE is cleared/set for whatever reason (e.g. in order to inject an Idle frame), enable pullup or use appropriate external pullup
- if Tx is connected to other driving pin and it's anticipated that it will go threestate after UART_CR1.TE is cleared, either the extra (half)bit has to be taken into account, or the pin should be set to Input in GPIO_MODER[1]
- the extra bit may should be taken into account in cases where tight timing is expected

---

1. While manipulating GPIO_MODER runtime and in interrupt/multitasking context, one has to observe atomicity if several such operations are present throughout the program. While this may seem "self-evident", manipulating this sort of registers runtime is unusual and this requirement may be easy to be overlooked resulting usually in rarely occuring and hard to debug issues.

Created: 15.January 2022
Intro for the new version of UART added: 19.February 2022

88.The Y2k38 bug (not STM32-specific)

Seasoned STM32 programmers may remember the Y2k frenzy around year 2000, when some older programs which assumed they will never operate beyond the 20th century and e.g. used 2-digit BCD coding for calendar year, were in risk of failing. While such programs were comparatively rare, it was not easy to tell without closer inspection, whether a program was "safe" in this regard or not.

A similar boundary is slowly approaching, as the UNIX epoch (seconds since 1.Jan 1970) will reach 2^31 on 19.Jan 2038, overflowing in systems which maintain time in 32-bit (signed) int. This epoch is used for keeping date in many systems using the standard C functions, including compiler suites for STM32 based on gcc, which usually employs some variant of the newlib standard C library. So, is this setup vulnerable to the Y2k38 problem?

Checking sizeof(time_t) on most not-that-old installations of such system may reveal, that time_t type which holds the UNIX-epoch time, is declared as 64-bit integer, so it should be completely safe for many centuries to come. However, the nature of problem is not that simple, and it turned out, that one particular function, **mktime()**, used to convert calendar-style date/time to UNIX-epoch based time, still suffered from the Y2k38 problem, due to the C-language implicit type conversions.

This latter problem was already fixed, too; so users of newest versions of gcc/newlib should be safe.

Created: 22.January 2022

89.How to set TRGO in TIM16/TIM17? Also, TIM channel needs output enabled if controls another module, e.g. ADC.

Many timers in STM32 have a mechanism through which they can output a signal (called TRGO, Trigger Output) towards other modules within the chip. This is then used for the inter-timer synchronization, but also in triggering ADC and DAC.

There are several sources from which TRGO can be chosen (in TIMx_CR2.MMS): Reset, Enable and Update, which are internal signals of the timer and its slave-mode controller; outputs of all 4 channels' output comparator (OCxREF); signal which sets channel 1 interrupt flag (regardless of whether CH1 is set as Capture or Compare - this signal is notoriously incorrectly called Compare Pulse throughout ST's materials and software).

However, in some STM32 models, other timers or modules such as ADC are triggered from timers (typically TIM16/TIM17), which don't have the TRGO setting facility (i.e. don't have the TIMx_CR2.MMS field, or don't have TIMx_CR2 register at all). How is TRGO generated in those timers?

Here, the signal output from timer instead of TRGO is identical to the signal driving the respective pin of such timer's (usually single) output channel. It is not merely OCxREF of given channel, but it includes also the enable signalling, i.e. the respective TIMx_CCER.CCEx has to be set. And, in timers which do have TIMx_BDCR, also bits from this register, impacting the enabled state of given pin, are considered (MOE, OSSI, etc.).

In other words, the effective triggering signal is result of AND of the output level signal and the enable signal, going from given timer channel into the AF matrix of GPIO. There is no need for any pin to be assigned to given timer channel in GPIO, though.

This description applies to all instances where triggering comes directly from a timer's channel, rather than TRGO (e.g. when an ADC is triggered by TIMx_OCx).

Discussion of this issue with ST participation here.

Created: 29.January 2022

90.Some time after peripheral (TIM, SPI, etc.) function finished, pin state changes, resulting in unwanted clock/transition.

In STM32, if a pin is set to Alternate Function in its GPIOx_MODER setting, and a connected peripheral is selected in the respective GPIOx_AFR[] field, the pin's digital input is connected to the selected peripheral's input (if the associated peripheral pin's function is input). Similarly, the associated peripheral's output (if there's any) is connected to the pin's output driver.

There are two output signals, though, going from peripheral to GPIO: **output level and output enable**. The peripheral controls **both**. Usually, when the peripheral's clock is disabled in its respective RCC_AxBx_ENR, output enables for the peripheral's outputs are also disabled - this functionality is not properly documented, though, so cannot be relied upon, but has to be taken into account as a possibility. In some peripherals, outputs' enables are disabled when peripheral's "global enable" bit is cleared.

Some peripherals even have explicit output enable bit/bits - e.g. in timers, TIMx_CCER.CCxE/CCxNE function in this way, and in Advanced timers, the whole BREAK-related circuitry is geared towards output enable/disable.

Some library functions are aimed at delivering functionality which has a definitive duration, e.g. a function for communication interface may transmit a single packet. Some of those functions, after having accomplished their action, simply switch off the given peripheral. There also may be interface functions which are directly aimed at "stopping" a given peripheral. All these may result in output pins assigned to given peripheral to be threestated.

And, as such, they may change their state after some time, depending on associated leakages and parasitic capacitancies. Usually, voltage on them slowly decrease, but that's not necessarily always so. If there's strong enough crosstalk from a neighbouring track, the signal may even oscillate.

This may result in surprising behaviour of connected external peripheral, as signal which is output from a mcu usually controls some functionality, or serves as a clock.

There are several remedies to this issue:

- don't use library functions which switch off given functionality
- use internal or external pullup/pulldown (considering the parasitics and also increase of consumption while output is active)
- use such functionality of controlling peripheral which does not result in threestating, e.g. if in timer one desires to stop PWM output, some of the "set on match" or "forced" modes of Output Compare can be used instead of switching the channel off in TIMx_CCER
- before switching off the peripheral, change the given pin to Output in respective GPIOx_MODER, with output state in GPIOx_ODR set to the desired "idle" level

---

Created: 5.February 2022

STM32 gotchas

91.Some interrupts cannot be disabled, and attempts to do so[1] may lead to HardFault.

Interrupts in STM32 are controlled by NVIC, a module which is part of the Cortex-Mx processor. So, while individual interrupts are listed in the Interrupt chapter of RM, the internal working of the NVIC module is documented in the Programming Manual to the given Cortex-M core, and in ARM's materials.

Besides interrupts for individual peripherals in STM32, there are also "system" interrupts (or, in ARM's lingo, exceptions), related to the processor's internals. These include the Non-Maskable Interrupt (NMI), Faults[2] (including HardFault), Debug interrupt, and also the SysTick interrupt, as SysTick is also part of the processor core.

There are 16 positions reserved for "system" interrupts, after which the "normal" interrupts follow. Not all of the 16 are used, though, the actually implemented "system" interrupts vary between various Cortex-M cores, but ARM tends to maintain the numbers for the analogous interrupts between the variants.

Confusingly, ARM uses two kinds of numbering for interrupts. One is "natural", starting with 0, so the "system" interrupts are all between 0 and 15. This is reflected e.g. in the Vector Table, and also in indicating current interrupt in processor's core Program Status register PSR (IPSR), and also processor's System Control Block register SCB_ICSR in VECTPENDING and VECTACTIVE fields.

The second type of interrupt numbering is shifted so, that "system" interrupts are negative (from -16 to -1). This reflects the fact that the system interrupts are controlled directly from processor's SCB registers (whereas "normal" interrupts are controlled in a uniform way using NVIC registers). Most of the "system" interrupts cannot be disabled, and some have fixed priority. In particular, Faults (except HardFault) can be disabled in SCB_SHCSR[2,3]. SysTick interrupt can be enabled/disabled in SysTick's own control register using the SYST_CSR.TICKINT bit[4,5]. Priority of some of the "system" interrupts can be set in SCB_SHPRx. NMI has fixed priority of -2 and HardFault has fixed priority of -1 (smaller number means higher priority)[6].

For user convenience, ARM-provided CMSIS (in the core_cmX.h headers) offers several functions to access NVIC registers, including among others also the often used NVIC_Enable() and NVIC_Disable(). As a parameter, these functions generally take the given interrupt's number, in its second type (CMSIS requires the manufacturer to provide in the device header a list of symbols related to individual peripherals - e.g. WWDG_IRQn - mapped to the interrupt numbers, to be used with these functions). As the "system" interrupts are not controlled by NVIC, these functions generally don't take negative interrupt numbers. This is usually documented in doxygen-formatted comments to the given functions, but often overlooked by users.

So, system interrupts such as SysTick, cannot be disabled by calling NVIC_Disable() (and most of them cannot be disabled at all, anyway). As the negative parameter is unexpected by these functions, such calls may even result in surprising HardFault.

This issue has been brought up in this thread on the STM32 forum.

---

1. ... using the CMSIS-provided functions (NVIC_DisableIRQ() and related)... (as explained further in text)

2. Cortex-M0/M0+ does not have individual faults, only HardFault.

3. In addition to individual interrupts enable/disable, all interrupts except NMI can be disabled at once through single bit of the PRIMASK system register. CMSIS provides the __disable_irq()/__enable_irq() function pair to facilitate this.

4. Thanks to Alex (ataradov) for pointing out my mistake.

5. Not only ST cannot keep registers naming consistent between documentation and headers; but ARM, too. In ARM's own CMSIS headers the SYST_CSR register is called SysTick->CTRL.

6. ARM has this strange fixation on negative values in numbering things. These values are "virtual", though, they are not used in any register nor in any function call, they just express the fact that some interrupts have higher priority than any other interrupt. In some respects, even Reset is considered to be an interrupt (exception) with priority -3 (i.e. highest of all).

Created: 12.February 2022
Footnote 1 and SysTick disable added: 6. January 2023

92. What happens if peripheral is connected to other than a single pin in GPIO AF matrix?

When a peripheral is to use a (non-analog) pin, this pin has to be configured as Alternate Function in a bitfield in GPIOx_MODER, respective to given pin. This connects the pin to the AF matrix. Then, the appropriate peripheral is to be chosen in this matrix in the given pin's GPIOx_AFR[] bitfield.

Normally, it is anticipated, that exactly one pin is connected to a given internal pad of the peripheral. However, in the AF matrix, often several alternative pins can be chosen for one peripheral pad. What happens if several pins are connected to one pad simultaneously? And what happens, if *no* pin is connected?

When **multiple** pins are assigned to a single peripheral pad, experimental evidence indicates, that if this pad is **input**, the individual pin's states are simply **ORed**. Similar experiment indicates, that state of **output** pads (both enable and state) are simply **fan-out** to all assigned pins.

When **no** pin is assigned to a peripheral pad, outputs are simply ignored, and **inputs** - according to experimental evidence - are **set to 0**.

Unfortunately, the *I/O alternate function input/output* subchapter of *GPIO chapter* of some STM32 Reference manuals has this to say (quoting from RM0090 rev.19 for STM32F4xx):

Note: The application is allowed to select one of the possible peripheral functions for each I/O at a time.

While this note is absent in many if not most STM32 families' RM, it clearly indicates that ST did not design the GPIO matrix consciously and meticulously to allow connecting none or multiple pins to a single peripheral pad. In other words, while the experimental evidence indicates that the "most logical" solution was chosen, there may be surprising exceptions to this, and ST is not intending to document it, so it is not to be used in real-world applications.

Unfortunately.

Created: 19.February 2022
Revised with the quote from RM prohibiting such usage 27.August.2022

93.Timers don't support DIR/STEP encoders (except in 'G4/'U5) - but a trick may exist

Timers in STM32 support quadrature encoders - and they count up/down as expected, according to the order of edges in the two quadrature signals. Not all timers support this, and those which do (usually TIM1 to TIM5, TIM8 and TIM20), cannot be used for any other useful purpose, as their slave-mode controller and the single counting element is used for this feature; and only CH1 and CH2 can be used as inputs of quadrature signals. Nonetheless, this still is a very useful and sought after feature.

However, there are also other types of encoder out there, which don't output quadrature signal, rather, they output one signal determining direction, and second signal pulsing upon each step. Such encoder is not supported by most STM32, as there is no direct way to change the direction of counting (indicated by TIMx_CR1.DIR bit) from a pin. The newest STM32G4xx and STM32U5xx lines are the exception, as the timers there have been enhanced and they support DIR/STEP style encoders directly in hardware.

However, on older STM32 models, there may be a trick available, as described by berendi on the STM32 forum. The trick is based on generating a signal from DIR/STEP, which together with STEP behaves similarly than quadrature signal, by XORing DIR and STEP together and delaying it slightly.

For XOR, the internal XOR of timer is used. As this XOR uses 3 inputs - CH1, CH2 and CH3 - one of the inputs has to be kept on an unchanging level. For this purpose, either one pin has to be tied to GND or VCC externally, or the undocumented property that unconnected peripheral pad is at steady 0 level, can be used.

As an adjustable delay, the filtering feature of timer channel input can be used. This then determines the maximum clock rate of STEP signal.

Created: 26.February 2022
'U5 added 19.February 2023

94.'G4 lifetime may be surprisingly short (at elevated working temperatures)

Similarly to the 'H7 family, the STM32G4xx also have a somewhat limited lifetime at elevated working temperatures. This fact is documented in AN5738.

This is not surprising, given 'G4 are built in the same 45nm technology than the 'H7, enabling very high integration i.e. (sometimes overly) complex peripherals on a small silicon area, resulting in relatively small unit cost.

Unfortunately, ST does not reveal the underlying mechanism for this limitation. It might be reduced flash retention, diffusion-related issues, electromigration, or any other effect.

While the limitation is not that dramatic as with 'H7 and lifetime even at relatively high temperatures achieves 10 years (contrary to 2 years with 'H7), this is something to be kept in mind when designing with this family.

Created: 5.March 2022

STM32 gotchas

95.How to achieve consistent RTC readout?

This article is based on findings of EngyCZ, while attempting to achieve consistent subsecond-precision time reading, reported in this thread, and by end of this thread.

This is hopefully the last installment of the RTC-data-readout saga, with previous issues dicussing effects of BYPSHAD=0: on the need to read DR after SSR/TR, and on the exact mechanismus how TR and DR are "locked" when SSR/TR are read.

Those articles also describe to which STM32 families this issue pertains - i.e. not 'F1 and not the newest 'G0/G4 and newer families. This is equivalent to the **version 2 of the RTC**, as described in **AN4759**.

Please note, that issues described here are equally relevant to use cases where subseconds are not needed, i.e. the time (TR) and date (DR) readout may be equally mutually inconsistent than SSR/TR readouts. However, as there are nearly a hundred thousand seconds in a day, with equal frequency of readout and readouts not intentionally synchronized to RTC clock, the probability of observing inconsistency between TR/DR is much lower than probability of observing inconsistency between SSR/TR, and in many practical cases negligible[1].

The main aim here is to achieve a consistent set of SSR/TR/DR readouts. As DR changes rarely, we concentrate on SSR/TR readouts. A sequence of normal readouts may look like the following (with the usual synchronous prescaler of 256 i.e. PREDIV_S = 255; and remembering that SSR is a downcounter):

```
sec:22 subsec: 2
sec:22 subsec: 1
sec:22 subsec: 0
sec:23 subsec: 255
sec:23 subsec: 254
sec:23 subsec: 253
```

Issues influencing consistent SSR/TR/DR readout from RTC with **BYPSHD=0** (i.e. the lock/unlock mechanism in place) include:

1. **erratum-described late-lock** - this demonstrates itself for example as the following sequence of readouts:

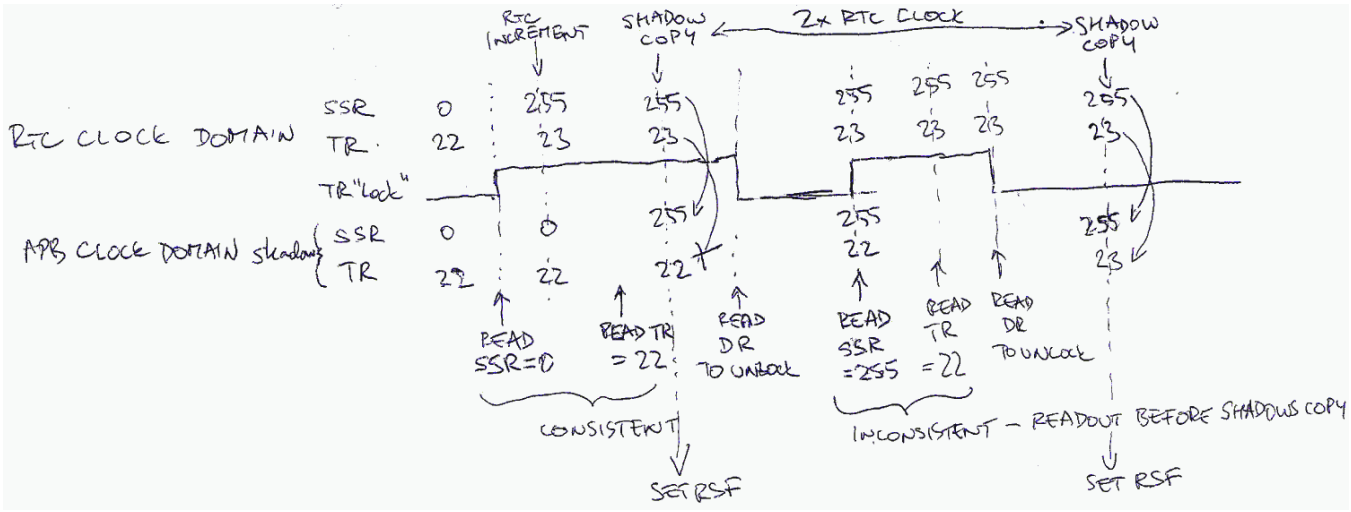
   ```
   sec:22 subsec: 2
   sec:22 subsec: 1
   sec:22 subsec: 0
   sec:23 subsec: 0
   sec:23 subsec: 255
   sec:23 subsec: 254
   sec:23 subsec: 253
   ```

   as readout of subseconds may happen before the increment, and reading of subseconds may fail to lock the old pre-increment value of seconds exactly as the erratum describes it.

2. **early reading after an unlock** - the datasheet requires to wait at least 2 RTC clocks after reading DR in order to "unlock", before next readout happens. The consequence of not fulfilling this requirement is the following sequence (note, that here we show only readouts which changed; in fact, there were multiple readouts with the same value in between those which are shown):

   ```
   sec:22 subsec: 2
   sec:22 subsec: 1
   sec:22 subsec: 0
   sec:22 subsec: 255
   sec:23 subsec: 255
   sec:23 subsec: 254
   sec:23 subsec: 253
   ```

   where subseconds display the value after increment (again, SSR is downcounter) while seconds (TR) are from the old "locked" value (more precisely, TR has not been refreshed). As we've learned already, SSR is never "locked", but reading it "relocks" TR/DR.



A bullet-proof method to ensure consistent readout here consists of using the RSF mechanism: user clears the RSF bit and waits, until hardware sets it, which indicates that the shadow registers have been copied from RTC domain[2]. Of course, users can come up with other methods based purely on observation of time outside of RTC, e.g. using a common timer.

3. **RSF is a write-protected bit**. What that means is, that when implementing the RSF-based readout delay as per the previous item, and if we use the RTC write protection (as we should in most of the applications), before writing 0 into RSF, we must first un-protect RTC writes using the key sequence written into RTC_WPR.

   Otherwise, zero does not gets written into RSF, and subsequent check for RSF being one is fulfilled immediately, without actually waiting for the shadows refresh to happen, leading ultimately to inconsistent readouts again.

An example bullet-proof code to read out RTC with BYPSHAD = 0 (within code utilizing the RTC write protection, which is the usual case) provided by EngyCZ is[3]:

```
    volatile uint32_t drreg1, trreg1, ssrreg1;
    volatile uint32_t drreg2, trreg2, ssrreg2;

    do
    {
      // Unlock RTC registers
      RTC->WPR = 0xCA;
      RTC->WPR = 0x53;
      // Clear the RTC_ISR_RSF
      RTC->ISR &= (uint32_t)~RTC_ISR_RSF;
      // Lock RTC registers
      RTC->WPR = 0xFF;
      // Wait for RSF
      while ((RTC->ISR AND RTC_ISR_RSF) == 0);

      ssrreg1 = RTC->SSR;
      trreg1  = RTC->TR;
      drreg1  = RTC->DR;

      // Unlock RTC registers
      RTC->WPR = 0xCA;
      RTC->WPR = 0x53;
      // Clear the RTC_ISR_RSF
      RTC->ISR &= (uint32_t)~RTC_ISR_RSF;
      // Lock RTC registers
      RTC->WPR = 0xFF;
      // Wait for RSF
      while ((RTC->ISR AND RTC_ISR_RSF) == 0);

      ssrreg2 = RTC->SSR;
      trreg2  = RTC->TR;
      drreg2  = RTC->DR;
    } while (ssrreg1 != ssrreg2);
```

The basic drawback of this code is, that it inevitably waits for the shadow registers reload (through the RSF mechanism), resulting execution of this code to last for cca 60us in the usual case. Some optimization may be possible, by clearing RSF at the end of the function and then checking it at its beginning (assuming there's no other code working with RTC meantime); nonetheless there's always at least one shadow reload which has to be waited for.

Now, let's look at the case of **BYPSHAD=1**, i.e. the case, when shadow registers are not used, and processor reads the RTC registers directly.

It can be anticipated, that an RTC increment may happen in the course of reading individual registers, so a single readout won't result in mutually consistent values. However, the surprising fact EngyCZ found (and I've reproduced) was, that not just registers were mutually inconsistent, but that sometimes **bits within registers were inconsistent**, too! Clearly, in that case, the read from APB clock domain came in the middle of bits flipping in the RTC-clock-domain-registers as consequence of increment.

This, while undoubtedly surprising for many users, *is* documented in the RM:

Additionally, the value of one of the registers may be incorrect if an RTCCLK edge occurs during the read operation.

The following code by EngyCZ represents a bulletproof implementation of workaround described in the RM:

```
void ReadDateTime(void)
{
  volatile u32 subsec,  timereg,  datereg;
  volatile u32 subsec2, timereg2, datereg2;

  do
  {
    datereg  = RTC->DR;
    timereg  = RTC->TR;
    subsec   = RTC->SSR;

    datereg2 = RTC->DR;
    timereg2 = RTC->TR;
    subsec2  = RTC->SSR;
  } while (datereg != datereg2 ||
           timereg != timereg2 ||
           subsec  != subsec2
          );

  ... process subsec,timereg & datereg ...
```

A slight optimization may be possible, as it should be enough to read SSR twice - before and after reading DR/TR - and compare only the two SSR readouts to determine, whether an increment happened during reading or not (and loop if it did, i.e. in case of mismatch).

While RM warns that in this case an RTC register read lasts 2 APB cycles rather than 1, this method is still far quicker than the code with BYPSHAD=0, which requires to wait for shadow copy because of the Erratum, as described above.

User Piranha in this thread proposed an intriguing method of RTC readout: read TR/DR into one set of variables, read SSR, then read TR/DR into a second set of variables. If SSR is above half of PREDIV_S accept the second set of variables, else accept the first set (note, that SSR is a downcounter!) Basic requirement for this algorithm to work is, that its total duration (including potential interrupts in between readouts) must be below half a second - but this requirement can usually be fulfilled relatively easily in typical (micro)controller applications. It was proposed, that this algorithm should work correctly for both BYPSHAD settings and even in presence of the Erratum.

The value of this algorithm is mainly in the fact that it avoids the loop, with all its consequences.

Unfortunately, for BYPSHAD=1 the fact, that SSR itself may be read out corrupted means, that this method can't be used. And for BYPSHAD=0, the requirement for the lengthy wait for shadows reload between the two readouts makes this algorithm unatracctive..

1.With entirely random/unsynchronized, and relatively sparse readouts, the "basic" probability of hitting Erratum-described inconsistency is given by the fact that the "not locking" leading to inconsistency happens - according to the Erratum itself - only if the readout occurs one APB cycle before RTC increment. As in usual setting, RTC increment which rolls over seconds (i.e. increment where inconsistency can be detected) happens once per second, and APB frequencies are in tens of MHz; the baseline probability of such inconsistency is one in several tens of millions of readouts.
Assuming average readout frequency in the range of 0.1s, one inconsistency would occur in a few hundreds of days. Also, without explicit rigorous method to find such inconsistency (e.g. if readouts are used only to log events, which are then observed only to find some particular event - which is the usual usage of such readouts), probability of spotting it is near zero. Both these explain, why it took roughly 7 years since introduction of STM32 models with this RTC, until the Erratum appeared.
Introducing intended or unintended synchronicities and/or increasing relative frequency of readouts (which includes decreasing of RTC prescalers to achive higher RTC resolution at the cost of losing real-time-ness; and using unusually low APB frequencies to achieve ultralow power consumption), together with such usage of readout which highlights the inconsistency (e.g. taking differences between successive readouts and processing them) may result in dramatically different observations.

2.Note, that if BYPSHAD=1 (e.g. in code which handles both BYPSHAD cases), the shadow-copy mechanism is not working and RSF will never be set by the hardware.

3.We don't discuss code resilient to hardware failures here. Users who are concerned about possible hardware failures may want to implement timeouts or count limits to the loops in presented code. In particular, the loop waiting for RSF may keep looping if LSE fails. Conversely, the outer loop checking for identical SSR/TR/DR readouts may keep looping if LSE starts to run unexpectedly fast, or the RTC module gets internally damaged and starts to return unexpectedly varying values - both these events appear to

be quite unlikely, though.

Created: 12.March 2022

96.TCM don't support highest clock frequencies in 'H723/725

This issue has been brought up in this thread on ST forum.

The STM32H7 family brought us brutal raw computing power, expressed mostly in raw megahertzs of system clock, but also in a complex bus system, built around an extensive 64-bit AXI bus matrix. To support this computing power, the ARM Cortex-M7 core relies heavily upon L1 cache interfacing the AXI matrix to the processor.

This setup of course brings also hard-to-control variations in details of program execution timing and resulting jitter in outwards signals. This in the vast majority of applications is negligible, thanks to both the high computing power and powerful peripherals hardware, which can handle many signals without need for immediate processor intervention.

However, as a legacy feature, the Cortex-M7 core supports also tightly-coupled memories (TCM), which allow better-controlled code execution than executing the caches/AXI system. TCM memories usually support zero wait state access.

The STM32H723/725 subfamily feature the highest system clock frequency amongst all STM32, up to 550Hz. However, as it turns out, the 'H723/725 don't support ECC for TCM at frequencies above 520MHz.

This information is conveyed indirectly: according to datasheet, frequencies above 520MHz require to have an innocently-sounding CPU_FREQ_BOOST option bit to be set. This bit is not explained in DS itself, and only RM clarifies that this bit switches off the TCMs' ECC.

As the 45nm technology used for the 'H7 family apparently requires ECC to guarantee data integrity in RAM, TCM is practically unusable above 520 MHz in that family.

---

a

Created: 19.March 2022

STM32 gotchas

97.5V-tolerant pins don't always tolerate 5V

In STM32, IO pins are usually in two groups: those, which are 5V-tolerant (marked in the pin table in datasheet as FT), and those which are not (TT). There are also a few special pins such as NRST; and the two major groups have also sub-variants, marked by additional small letters, e.g. FTa for 5V-tolerant pin with additional analog functionality.

TT pins simply have protection diodes to both VDD and GND (=VSS). FT pins don't have the protection diode to VDD, and that allows external voltages higher than VDD (i.e. +5V} to be connected to FT pins.

However, there are several limitations to this:

- Even if there is no diode directly from FT pin to VDD, there *is* an ESD protection structure connected, which limits the **maximum FT input voltage to VDD+4.0V** (as if it would be a reverse polarized Zener diode in series with another diode preventing current flow from VDD to pin - the actual circuit is different and uses a relatively complex structure pantented by ST). Exceeding this limit may damage the pin or the entire chip.
  As VDD is typically min 1.8V and 1.8V+4.0V=5.8V, it may appear that there's enough headroom for 5V signals. However, if the chip is powered down, VDD=0V, so **connecting a 5V signal to an FT pin of unpowered STM32 may result in damaged chip**.
- If FT pin is set as **push-pull output** (including AF set by peripheral to output), **or pullup is switched on**, there will be a flow of current ("injection") from pin to VDD. The amount of current depends on resistances in the entire path, so they must be set so that the current won't exceed the pin's injection limit (usually a few mA). Also, power supplies are usually not designed to handle reverse current flow, so in case of excessive current the VDD rail may rise above expected levels and cause damage to STM32 and/or any other circuit connected to the same power supply.
- FT **pins which have also analog functionality (FTa) become usually non-5V-tolerant, if their analog functionality is switched on**. This is usually accomplished by setting given pin to Analog in its respective GPIOx_MODER, although in certain models there may be additional switches, which isolate the analog function from the pin itself.
  Note, that some STM32 families have almost all pins set to Analog by default. Unfortunately, the documentation is often vague in explaining what are exactly the conditions for such pins to be 5V-tolerant; so unless documented otherwise, the failproof method is to switch on any external circuitry which would set voltage on FT pins above VDD should be accomplished only after all such pins have been properly set to some of the non-Analog setting in GPIOx_MODER.

There may be additional limits to certain pins, and also values mentioned above may not be valid for all STM32 models, so always consult the datasheet to your particular STM32 model.

Created: 26.March 2022

98. FLASH is erased to 0x00 rather than to 0xFF in 'L0 and 'L1

FLASH (as well as its predecessors, EPROM and EEPROM) is traditionally erased to all-ones state (i.e. after erasing FLASH, all bytes are 0xFF). This is not a necessity, just a custom, and in some cases this convention is simply not used.

In the world of STM32, the STM32L1xx and STM32L0xx families are the exception, where FLASH is erased to all-zero state (i.e. after erasing FLASH, all bytes are 0x00). These two families feature also a portion of FLASH which behaves like EEPROM, i.e. can be erased/rewritten byte-wise.

Why did ST decide for this unusual step is unknown. Sure, formally the particular erased state does not matter, but it is a surprise nonetheless, and care must be exercised when algorithms potentially depending on the value of erased FLASH (e.g. internal-FLASH-based file systems) are ported from different STM32 families to these two families.

Created: 2.April 2022

99. FLASH is 0xFF but cannot be written (if there's ECC and granule has been already written)

For sake of simplicity, in this text we will assume that erased state of FLASH is all-ones, ie. all bytes are 0xFF. This is not the case for 'L0 and 'L1 as has been described in Gotcha 98. As 'L0 does have ECC, the issue itself still pertains to this family, too.

Newer STM32 families are built using technologies with extremely small features, benefiting from the respectively increased circuit density. However, smaller transistors in FLASH also means a smaller volume of the floating gate where the charge representing programmed value of given bit is trapped, hence it means increased probability that noise during readout causes an incorrectly read bit.

To mitigate this problem, FLASH in these families is equipped with error detection and correction circuitry (ECC). FLASH is divided into granules (smallest portions which can be written, 32, 64 or 128 bits, depending on particular STM32 model). These granules are augmented by additional FLASH bits, invisible to the user, into which the error-check code is written, when the granule itself is written.

When the FLASH is read - regardless of whether code to be run is read from it, or data - the FLASH controller always reads the whole granule, performs the ECC check, if there is a single bit error, corrects it, and then returns to the system the requested portion of the granule.

If two or more bit error occurs, the error is uncorrectable. What happens in this case depends on the particular STM32 family/model. In most families, the uncorrectable error causes an NMI (nonmaskable interrupt), and then it's up to the user how this error is handled. In most families, also the single-bit error may throw a particular interrupt, if enabled.

There are several consequences of ECC on FLASH, which may be surprising to unaware users:

- In erased state, both visible and ECC portion of granule is all-ones, but while the particular ECC algorithm ST uses is not published, it is known that all-ones is not a valid ECC for all-ones data. In other words, reading data from erased unwritten FLASH granule results in ECC error. For most families the datasheet states, that this particular case results in a single-bit error, i.e. it does not throw NMI, but if the single-bit error interrupt is enabled, it will be invoked.
- Granules cannot be rewritten (or partially written).
  Some STM32 models disable rewriting already written granules during FLASH programming and throw a FLASH-write error. Other models enable rewriting a granule, but as the hidden ECC portion was not all-ones, the newly written ECC is in fact bitwise-AND of the previous ECC and ECC belonging to the newly written data, thus it is likely to have zeros where it should have ones. In other words, subsequent reading of such rewritten granule very likely results in ECC failure and NMI.
  This fact precludes schemes, where only a few bytes/bits of a granule are written from 1 to 0 at once, and later remaining 1s would be written to 0, e.g. implementing a "usage counter", with the aim to decrease FLASH wear and reduce needed erase cycles, increasing chip lifetime. This is not possible anymore.
- Granule written with all-1 (0xFF) is **not** an erased granule, and cannot be written again. This is surprising especially if user in program allocates an array into FLASH, initialized by 0xFF, with the intention to write to it later from the program. This array gets written using the programmer, and for the user it is indistinguishable from erased portion of FLASH; however, it is not writable without erasing it.

STM32 families can be roughly divided into the following groups, as far as their FLASH is concerned:

- 'F1, 'F0, 'F3 - no ECC
- 'F2, 'F4, 'F7 - no ECC, has OTP
- 'L4, 'G0, 'G4, 'L5 - has ECC, has OTP
- 'H7 - has ECC
- 'L1, 'L0 - erased at 0x00, has EEPROM, 'L0 has ECC, 'L1 RM mentions ECC only with EEPROM(?)

Created: 9.April 2022

100. In lower-end 'F0xx, GPIOC/GPIOD don't have AFR[2] registers

In STM32 (except the ancient 'F1 family), the individual peripherals are assigned pins through a sparse matrix in the GPIO module. To assign a pin to a peripheral, first pin has to be set to Alternate Function (AF) in its respective GPIOx_MODER field, and then the individual peripheral is selected by setting its assigned value in one of the GPIOx_AFR[2] registers (sometimes marked also as AFRL and AFRH).

In the lower-end STM32F0xx families ('F031, 'F042, 'F051, and the "value-line" derivatives of these, i.e. 'F030x4/6/8 and 'F070x6), in GPIOC and GPIOD, there is only one or no AF per pin. So, ST decided simply to omit the AFR[] registers from GPIOC and GPIOD.

This is properly documented in the initial section of the GPIO chapter in RM0090/RM0360, although it is formulated "inversely" (i.e. it says, that only GPIOA and GPIOB have the AFR registers).

There are also no *Alternate functions* tables in the respective datasheets for GPIOC and GPIOD.

Created: 16.April 2022

101. V_{DDA} is pulled high when ADC is enabled (if V_{DDA}<2.4V and booster not enabled)

We are here revisiting the need to use the analog switch voltage booster. This is needed in some families, if analog features (e.g. ADC) are used. However, it appears that signal speed is not the only reason to use the analog booster.

As reported in this thread, if V_{DDA}<2.4V, (while V_{DD} is presumably at around 3.0V or higher), and the analog switch voltage booster is - incorrectly - not enabled, some current starts to flow into V_{DDA} when ADC is enabled.

As usual voltage regulators can only source current, not sink, this may result in unexpectedly elevated V_{DDA} voltages. In smaller packages, where V_{DDA} is bonded together with VREF+, this in turn results in unexpectedly lower ADC readings.

Created: 23.April 2022

102. STLink-V3 8MHz MCO output is not derived from crystal

On development boards like Nucleo and Disco, for target processor a HSE crystal is usually not populated. For a more precise clock source than the internal HSI, these boards rely on the MCO output of the on-board STLink, which in its V2 (and V2-1) variants outputs a precise 8MHz based on its own crystal/HSE oscillator.

In the newer V3 version of STLink, instead of the venerable STM32F103 used in older versions, the STM32F723 is used. The main reason for this change is, that STM32F723 is the only STM32 model which features a High-Speed USB interface with an on-chip High-Speed USB PHY. This allows higher download speeds, which becomes increasingly important with STM32 models featuring extensive LCD/GUI support, as graphical assets stored typically in external (usually SPI/QSPI-connected FLASH) memories are relatively large.

However, using the on-chip HS PHY requires usage of crystal in HSE oscillator with one of few particular frequencies. ST made a relatively bad decision, choosing 25MHz, probably because that crystal has been already used in many of their ETH-enabled devboards.

Unfortunately, the clock tree on STM32F723 does not allow to generate 8MHz from the 25MHz HSE onto its MCO pin. So, what it does instead is, that it uses its HSI RC oscillator to output a - not very precise and relatively jittery - 8MHz onto the MCO pin, which is then fed to the HSE input of target processor.

This is an unpleasant surprise for users, which on related Nucleo or Disco boards wish to develop an application which requires precise primary clock.

As a partial workaround, using the STLink update feature in CubeProgrammer, the MCO pin on STLink-V3 can be switched to a mode where it outputs 25MHz/3 = 8.333MHz.

This issue came up in this thread, but has been discussed on the STM32 forum several times since.

Created: 30.April 2022

STM32 gotchas

103. After being cleared in NVIC outside an ISR, interrupt reappears (if pending flag in NVIC is cleared too soon after removing the interrupt source)

Sometimes, we want to enable an interrupt in NVIC, source of which has been active during the time it has been disabled. It means, that the given interrupt is set as pending in NVIC (as can be observed in NVIC_ISPRx registers), and as soon as it is enabled, it will "fire" and run the associated ISR.

If we don't want this, rather, we want an interrupt to occur only when the interrupt source gets active in the future, we need to clear the "old", pending interrupt, Using the NVIC_ICPRx register is intended exactly for this purpose[1].

However, we might not know, when was the interrupt source removed - it might've been just before we want to enable the interrupt. Or, it might've been our deliberate action which removed the interrupt source (e.g. clearing some status bit in a peripheral), just before enabling it. And, as we already know, it takes time until changes of interrupt source propagate to NVIC, so we might expect that clearing the pending bit too soon may not work.

It may seem that this is simple to work around: it should be enough to clear the NVIC pending bit in a loop until it reads as being cleared, right?

Unfortunately, this is not the case. After clearing the pending bit only a few cycles after removing the interrupt source, it reads zero for a couple more cycles, but then it gets set again. The following snippet has been run on an STM32F407 ~~just after reset (i.e. with no clock tree setup)~~ (see part 2 below):

```
int main(void) {
  RCC->APB1ENR |= RCC_APB1ENR_TIM3EN; // enable TIM3 clock

  TIM3->DIER |= TIM_DIER_UIE;  // enable interrupt from update event

  testA[0] = NVIC_GetPendingIRQ(TIM3_IRQn);  // there's no pending IRQ at the moment
  testA[1] = TIM3->SR;

  TIM3->EGR = TIM_EGR_UG;      // generate an update event

  testA[2] = NVIC_GetPendingIRQ(TIM3_IRQn);  // pending IRQ in NVIC? not yet
  testA[3] = TIM3->SR;                       // but we should see status register being
    __asm("nop");     // okay, it may take time until the interrupt flags "propagate" t
    __asm("nop");
  testA[4] = NVIC_GetPendingIRQ(TIM3_IRQn);  // there should already be a pending IRQ i
  testA[5] = TIM3->SR;                       // and also status register of the timer

  ~~TIM3->SR &= ~TIM_SR_UIF;  // this clears the status bit~~ -- see part 2 below
  TIM3->SR = ~TIM_SR_UIF;  // this clears the status bit

  // the key to the problem is the number of cycles elapsed between these two events
  // - more than 6 is needed to remove the problem in this snippet
    __asm("nop");
    __asm("nop");
    __asm("nop");
    __asm("nop");
    __asm("nop");
    __asm("nop");
  // __asm("nop");  // uncommenting this line makes delay long enough to remove the pro

  NVIC_ClearPendingIRQ(TIM3_IRQn);  // and this should clear the pending interrupt

  testA[6] = NVIC_GetPendingIRQ(TIM3_IRQn);  // there should be no pending IRQ in the N
  testA[7] = TIM3->SR;                       //

    __asm("nop");
    __asm("nop");
    __asm("nop");
    __asm("nop");

  testA[8] = NVIC_GetPendingIRQ(TIM3_IRQn);  // and it should remain so
  testA[9] = TIM3->SR;                       //


  while(1) {
    __asm("nop");
  }
}
```

Running this code results in the following content of testA array:

```
(gdb) p /x testA
$26 = {0x0, 0x0,  0x0, 0x1,  0x1, 0x1,  0x0, 0x0,  0x1, 0x0}
```

As witnessed by nonzero testA[8], the pending interrupt "reappeared" and would cause ISR to be entered immediately after enabling the interrupt in NVIC. Only after the 7th nop was uncommented before clearing the pending bit, it did not reappear again.

This is one more of those pesky timing problems, which are not documented at all, and even if experimentally determined, given we don't know what do they exactly depend upon (e.g. APB prescalers), they have the nasty potential to backfire if not handled with utmost caution.

Part 2

Soon after this article was published, I was contacted by Herman Roebbers, who pointed out, that the code I posted violates the rule not to use RMW on status bytes. He also suggested, that that the operation in question, TIM3->SR &= ~TIM_SR_UIF; might've been reponsible for the observed effect.

I've corrected the status register bit clear, and then also tested on an 'L476 Disco (which has unfortunately been discontinued by ST). I found out that with RCC left completely in reset state, the testA array was:

```
(gdb) p /x testA
$23 = {0x0, 0x0, 0x0, 0x1, 0x1, 0x1, 0x0, 0x0, 0x0, 0x0}
```

which means, that the interrupt did not "come back", and this remained so even when I removed every nop between TIM3->SR = ~TIM_SR_UIF; and NVIC_ClearPendingIRQ(TIM3_IRQn);.

Does this mean that my original test was incorrect? Turns out, that it's enough to set the APB1 divider to 4, and the described effect is back (and remains with insertion of up to 4 nop).

```
(gdb) p /x testA
$26 = {0x0, 0x0, 0x0, 0x0, 0x1, 0x1, 0x0, 0x0, 0x1, 0x0}
```

(Note, that the first TIM3_SR readback after TIM3_EGR.UG being set indicates, that UIF hasn't been set yet, due to slower APB bus thus TIMx operation.)

Conclusion?

My original test was done quite a couple of years ago and the original code and its compilation environment and binary did not survive. As we are dealing with issues related to a few clock cycles, the exact machine code generated by compiler, and possible minute differences in the bus/clock structure between 'F4 and 'L4, do matter. I now might spend time on digging deep and trying to reconstruct the original setup under which the interrupt "reappearance" occured back then, but that was not my point.

In demonstrating, that there *are* circumstances under which the cleared interrupt might "reappear", I wanted to alert the users on this possibility, so that they are not caught by surprise when attempting to perform the disable-clear-reenable sequence with interrupts in Cortex-M3/4/7-based mcus.

Originally reported here.

1.In the "normal" case, after clearing the interrupt source within executing an ISR, the hardware automatically clears the associated pending bit in NVIC.

Created: 7.May 2022
Part 2: 16. July 2022

104.USB packet buffer memory in 'F042 is not bytewise writable

This issue has been reported here, and has not been confirmed by ST.

There are two types of USB modules in the various STM32: the device-only module (which only recently was expanded also by host mode in the 'G0Bx), and the device-host-OTG module licenced from Synopsys. The former is to be found in the lower-end 'F1 and 'L4, and in all 'L0, 'F0 and 'F3 which have USB at all.

This module has a heavy 16-bit legacy feel, underpinned by the fact that all its registers are 16-bit, although they are placed at 32-bit addresses. This may be also the reason for relatively weird access to the packet buffer memory, i.e. memory where outgoing data are stored by user and incoming data are stored by the hardware.

For this packet buffer memory, RM0091 rev.9 states:

The packet memory should be accessed only by byte (8-bit) or half-word (16-bit) accesses.

(and it also forbids word (32-bit) accesses). While reading it bytewise is OK, experiments on STM32F042 revealed, that writing the packet buffer memory byte-by-byte does not work as expected. Regardless of whether the byte is written to the lower or upper address of a halfword, it is in fact written simultaneously to both halfs of that halfword.

It is possible, that this behaviour can be found also in other STM32 models which feature the device-only USB module.

Created: 14.May 2022

105. Clicked ETH/USB in CubeIDE/CubeMX, and it does not work.

Microcontroller manufacturers go to great lengths to make their products attractive to users. One way how they do this is, that they prepare environments - libraries, IDEs, point-and-click configuration utilities - which are designed to look programming microcontrollers easy. This then includes also very complex interfaces such as ethernet/TCP/IP and USB.

These are incredibly complex interfaces with multiple layers, and with many interactions with hardware, other pieces of software and the whole slew of various devices they connect to. As a consequence, there are also millions of ways how they can break down. It should be understood, that this is not primarily fault of badly designed configuration utilities or buggy underlying firmware, although that's the prevailing view. But the truth is much simpler: there are just too many potential points of failure, which are outside of scope of those, who wrote those utilities and firmwares.

So, it's no wonder, that users often come to places like the STM32 forums with the question similar to the title of this article. There is no good answer to this question, as there are too many things which could have gone wrong, and only the user can find out, what was the problem.

While the genuine answer is - just don't use the pre-chewed version, go out and write your own, maybe inspired by, or partially adopting, what's available - this indeed takes lots of effort, learning, reading, and lots and lots of time. So, it may be worth try and fix the existing "clicked-out" version.

So here are some simple hints, what to try:

- start with a known-good hardware and software: use a Nucleo/Disco/EVAL board. Use provided software example; if possible, use provided binary, to exclude compilation issues.
- if this does not work, check the board's configuration, is it identical with the factory default, haven't there been jumpers removed/flipped
- if this still does not work, assume outside problems: replace cables; for USB try a different USB hub, port, or even a different PC; for ETH check, what IP configuration does the example expect (e.g. a running DHCP server, or network accepting a particular IP address)
- the TCP/IP examples from ST traditionally require ETH cable to be connected (to a switch) when the software is started (because the software expects PHY reporting the appropriate detected mode/speed immediately after reset)
- at this point, if still no success, start debugging as usually; start with checking presence of clocks - USB usually requires an internal 48MHz clock, usually from a dedicated PLL tap, to be set and running (and HS USB also a dedicated 60MHz clock, associated with the PHY); ETH requires a 25/50MHz clock be present on PHY, and then being fed also to dedicated pins on STM32
- these clocks have to be enabled also in RCC; for ETH, the RMII/MII switch has to be set appropriately. Read out the respective registers and check.
- in both interfaces, there is a basic level of setup which is still manageable without having to dig too deep into the details, and that can be checked. In USB, verify, if register controlling the DP pullup is switched on, if it's not in some low-power mode, if packet memory buffers are set up properly. In ETH, check the MDIO/SMI communication between STM32 and the PHY, looking both at physical signals, and content of registers, as software goes through communication with the PHY.
- the OTG USB module in some variant (usually the High Speed one, even if not used as HS) incorporates DMA, usage of which is optional; it may be worth trying it both with or without DMA
- if the interfaces are basically running, and the problem is in some particular detail not working or not working satisfactorily, it is incredibly helpful to have a way to look at the resulting communication from outside - for USB, this means bus protocol analyzers, which may come in the form of dedicated devices (and there are some not-that-extremely-expensive ones around mainly for Full-Speed) or as protocol analyzers in some oscilloscopes and logic analyzers; for ETH/TCP/IP this is usually accomplished with a PC-based analyzer software (Wireshark is a very popular one), in some cases coupled with managed switch which can mirror communication between given ports.

Created: 21.May 2022

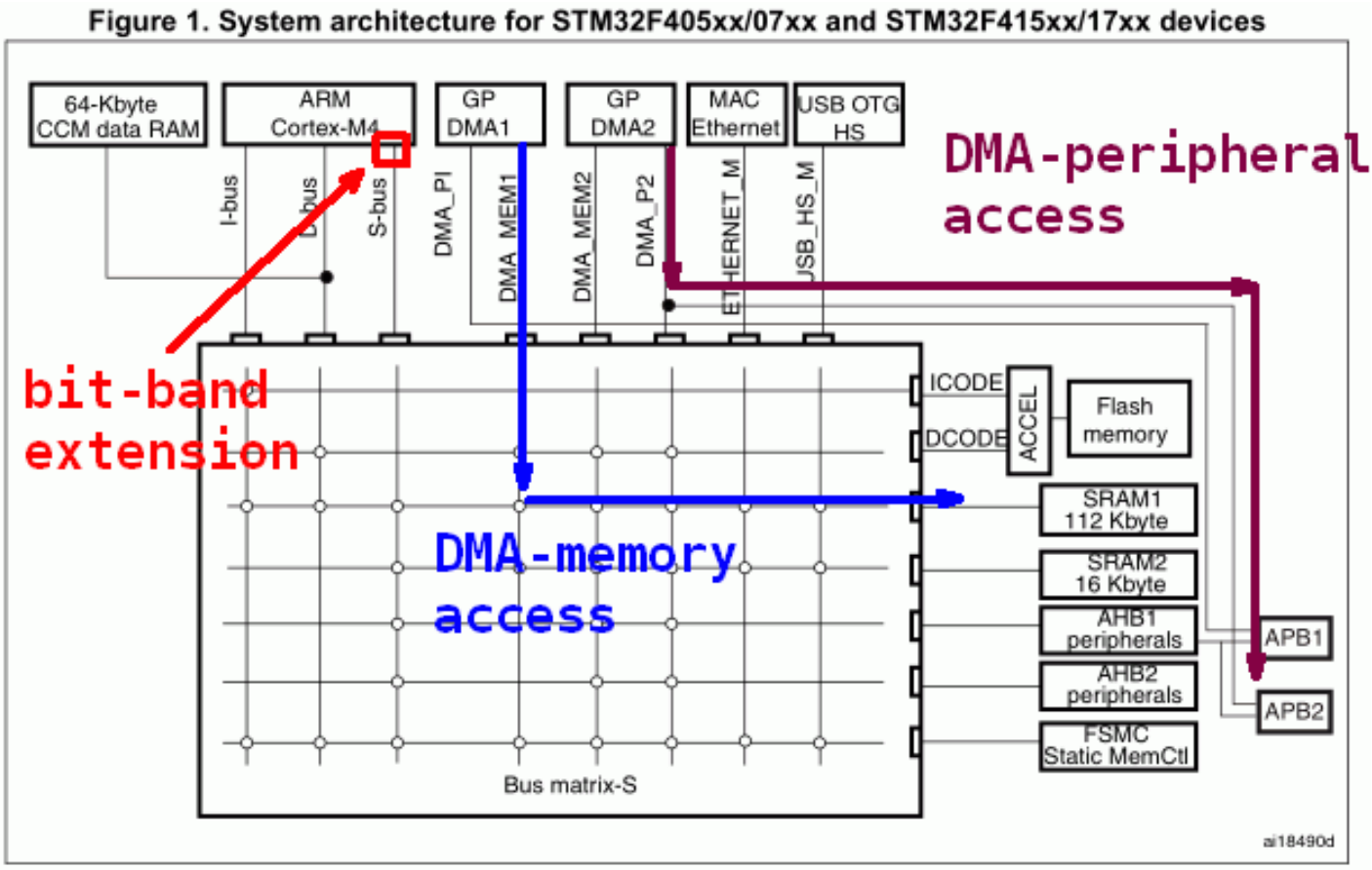
106. Bit-banding cannot be used with DMA

Bit-banding is an interesting addition to the Cortex-M3/M4 processors (used in STM32F1/L1/F2/F4/F3/L4/G4), which allows atomic access[1] to a portion of memory and peripherals.

In some cases, it is tempting to use this access to read or write single bits also from DMA (e.g. to flip polarity of TIMx_CHx output by writing to its respective TIMx_CCER.CCxP).

This does not work, as bit-banding is performed by an extension on the S-port of the processor. This extension intercepts accesses from processor targeted to the bit-addressable regions, "translates" the address from the bit-addressable to the "real" memory/peripheral address, generates the appropriate bitmask, and converts the bit-banding access to either a RMW operation modifying the required bit if it is a write, or to a read followed by mask-and-shift to return only the required bit on the data bus for the processor.

However, accesses from DMA don't go through this extension.



In other words, for DMA (and other bus-masters), the bit-addressable alias area is simply nonexistent, and accessing it results in DMA transfer error.

---

1. Bit-oriented access through bit-banding is atomic from the processor's perspective, i.e. it cannot be interrupted by any other process within the processor. However, if the target of access is a bit within a peripheral register with other bits changed directly by hardware, the consequences are the same as with any other read-modify-write accesses to such peripheral registers. See detailed discussion of this issue here.

Created: 28.May 2022

107. No atomic access through bit-banding to GPIO registers' bits (except in 'F1)

There are two bit-banded areas in mcus based on the Cortex-M3/M4 processors - one intended to map a portion of SRAM, and onther intended to map peripherals. In STM32, connection to most peripherals is implemented through one or more "secondary/slower" APB bus connected through a bridge to an AHB bus within the bus matrix, and it's these APB buses which are mapped into the peripheral bit-banded area.

In STM32F1xx, GPIO was connected to such APB bus, so bits within its registers were bit-addressable. However, in all subsequent STM32 based on Cortex-M3/M4 (i.e. 'L1/'F2/'F4/'F3/'L4/'G4), GPIO resides on one of the AHB buses, presumably to allow faster access to them. As this AHB bus in all mentioned models is outside of the bit-banding areas, there is no bit-banding access to GPIO registers.

For the vast majority of applications this does not matter, as the most used GPIO register which requires atomic access is the output register, GPIOx_ODR; and for that, atomic writes are provided inside the GPIO module through the GPIOx_BSRR register (and in some cases, the mostly redundant GPIOx_BRR register). However, in some applications, atomic write into other GPIO registers would be benefitial, too. As this is unavailable through hardware (bit-banding), applications must apply the usual care, when accessing GPIO registers both from "main" and "interrupt".

Created: 4.June 2022

STM32 gotchas

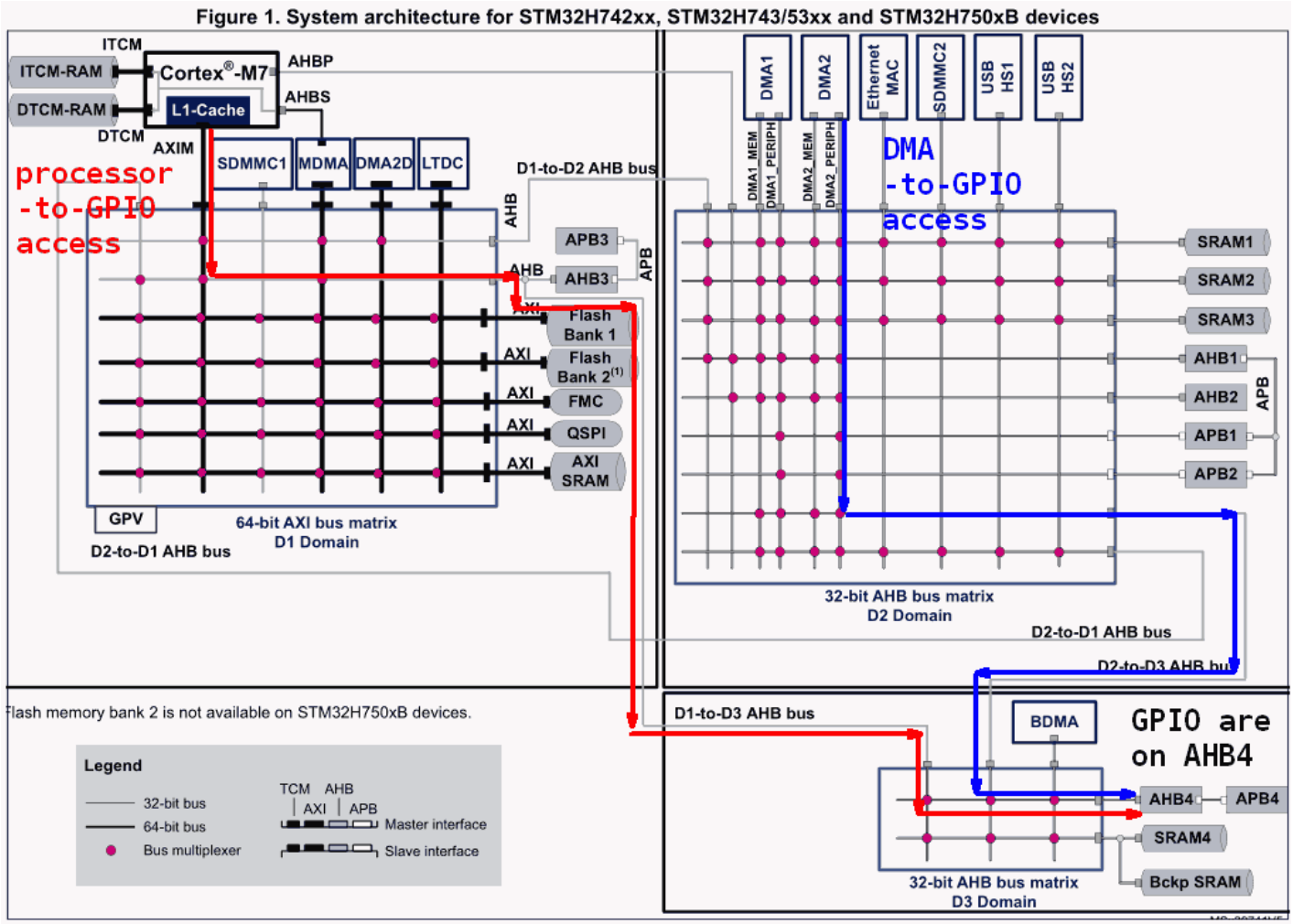
108. Toggling GPIO in 'H7 is slow (due to complex bus structure)

We've talked about how frustrating it is for the novices who come to the 32-bit mcu world, writing their very first test program, which does nothing but toggles a pin as rapidly as it can be toggled, in a loop, to see the pin toggling much slower than they anticipated.

Nowhere is this frustration bigger than with the 'H7 family, where the processor is clocked at hundreds of MHz, yet a GPIO pin toggles only at low tens of MHz. Why?

To start, let's realize, that in the 'H7 it is only the processor core which runs at the advertised highest frequencies. All main buses run at maximum half that frequency, and most peripherals at maximum quarter of that.

Moreover, the 'H7 is an incredibly complex design, based on three bus matrices. This is given by several facts: one single bus matrix would be physically too big, resulting to be too slow and too power consuming. The "first" matrix is a 64-bit AXI matrix, which is unnecessarily extensive/wasteful for the slower "normal" peripherals. The second matrix is mostly derived from the 'F2/'F4/'F7 family and partially caters to the secondary Cortex-M4 processor core; and even if that is present only in some high-end members of the 'H7 family, the impact is still seen in other members, which largely copy the general layout for simpler overall family design.



Bus matrix separation also allows simpler clocks separation between various parts of the chip, which in turn allows a whole bunch of power consumption saving modes - and those are desperately needed in 'H7 should it survive more than just a few years of operation.

This latest requirement is probably why the GPIO modules are placed at the AHB4 bus, which is part of the relatively small D3-domain bus matrix. This matrix is intended to be "alive" at the lowest-power-consumption, yet stil peripherally-active modes, thus contains also the low-power asynchronous timers and UART (as well as one SPI and one I2C unit) and a small single-port DMA unit to be able to handle these peripherals partially without processor intervention. It also contains the optionaly battery-driven RTC, comparators, watchdog and the EXTI unit - all these related to low-power-run and wakeup functionalities.

The consequence of this arrangement is, that accessing GPIO registers from processor and the general-purpose DMA always crosses at least two bus matrices. So this involves not only delays due to two bus-

matrix arbitration processes, but as different bus matrices potentially run at different clocks, which means, that there are resynchronization units between the matrices, requiring additional clock periods for their working.

These delays cumulate up to the point where toggling GPIO from the processor (and from the general-purpose DMA) results in frustratingly slow changes on the pin. This is not to say that the pins themselves can't toggle faster than that - indeed they do, when set to Alternate Fuction in GPIOx_MODER and then driven directly from the associated peripherals.

Hand in hand with this comes also jitter i.e. uncertainties in timing - so e.g. duration of a single pulse generated by two consecutive writes to GPIO from processor can result in variable pulse length, depending on load on the bus matrices from other bus masters, and also on the momentarily mutual state of synchronicity between involved bus matrices.

The exact working of inter-matrix synchronizators is undocumented, so calculation of these timing effects is mostly impossible.

Created: 11.June 2022

109. PF0/PF1/PC14/PC15 don't work, if overriden by oscillator enabled in RCC

ST has a good habit of locating the HSE oscillator pins onto PF0 and PF1, and the LSE oscillator pins onto PC14 and PC15, in all STM32 families. This goes that far, that GPIOF is present in the smallest STM32 models such as the 'F0, 'L0 and 'G0, which may even not have GPIOD and GPIOE.

However, if HSE is switched on in RCC (by setting RCC_CR.HSEON or similar), PF0 and PF1 cease to be controlled by GPIO. Most commonly a crystal (together with loading capacitors) is then connected between these two pins.

Sometimes an external clock source is used, in which case it is connected to OSC_IN = PF0, and RCC_CR.HSEBYP is set. In this case, the OSC_OUT = PF1 pin behaves differently in various families - in some families (e.g. 'F3 or 'G0) it is freed up and can be used as GPIO; in others (e.g. 'F4 or 'L0), the RM requires them to be left Hi-Z. So, **availability of PF1 with external HSE clock source has to be checked for each family/model individually**.

This is very similar for PC14 and PC15. Again, they are overriden by enabling LSE in RCC_BDCR.LSEON (RCC_CSR.LSEON in 'L0, which has no backup-domain so no Backup-Domain Control Register, BDCR); and again, in case of using external 32.768kHz clock source, in some families with setting of LSEBYP bit the OSC32_OUT = PC15 is freed up to be used as GPIO, and in other it has to be left Hi-Z.

Besides of that, if used as GPIO, PC14 and PC15 are in the low-power or VBAT-power-domain, so they usually have lowered drive capability as compared to other GPIO pins.

Created: 18.June 2022

110. Using interrupt or DMA, only first few data are transmitted correctly (due to local/nonstatic variable used as buffer) - not ST-specific

Libraries such as Cube provide functions, which transmit an array of data (a packet) through communication interfaces such as UART, SPI, I2C etc. These functions come in various flavours, usually there are "polling"/blocking versions available, as well as non-blocking versions based on interrupts or DMA. As one of the parameters, these function take pointer to an array, which contains the data to be transmitted.

Sometimes users define the array, which holds the data (buffer), as a local variable in a function. As local non-static variables are placed at the stack, content of the buffer may get overwritten after program execution leaves the function in question and executes other functions. As data transmission using interrupts/DMA happens "at the background", while other code is executed, the data transmitted come from the already modified buffer, i.e. are different from the intended ones.

This does not happen necessarily with local buffers. For example, if execution dwells long enough in given function for all data to be transmitted; or if the function is the furthest on the execution tree (more precisely if its stack frame is the deepest in the stack usage tree), or if execution never exits from given function (as is case of main()), using local buffers is seemingly "safe". The danger then lies in potentially excessive stack use, and in potential problems if transmitting piece of code is migrated to a different function, or if the execution tree changes due to other functions being rewritten/reordered.

As occurence of symptoms depends on the code execution path, may be not apparent immediately and may "bite" surprisingly some time later, it is important that users are aware of this problem and avoid consciously it when writing the code.

Counterpart to this issue, related to Rx rather then Tx, which potentially results in program crash, is described here.

Created: 25.June 2022

111. Some interrupts invoke incorrect ISR (if VTOR is not properly aligned)

This issue has been discussed here for 'H7 and here for 'F4.

The interrupt vector table can be relocated in Cortex-Mx (except Cortex-M0) using the processor-core register SCB->VTOR (Vector Table Offset Register). This is used e.g. when interrupts have to be moved to RAM (for higher execution speed, or because of FLASH unavailability e.g. during FLASH programming), or to a different than default area of FLASH, often when custom bootloaders are used.

VTOR has 128-byte granularity and its lowermost 7 bits are fixed at zero. According to ARM's materials, when interrupt occurs, the appropriate vector is picked by adding the interrupt's number multiplied by four, to VTOR, then the interrupt vector is picked from the resulting address. *ARM® v7-M Architecture Reference Manual ARM DDI 0403E.b (ID120114)* gives the following pseudocode for this process:

```
ExceptionTaken(integer ExceptionNumber)
[...]
bits(32) VectorTable = VTOR<31:7>:'0000000';
tmp = MemA[VectorTable+4*ExceptionNumber,4];
BranchTo(tmp AND 0xFFFFFFFE<31:0>);
```

where MemA[] roughly means "check address for alignment and priviledges, and then return the word from that address".

This process implies, that the vector table might start at any address of 0xXXXX'XX00 or 0xXXXX'XX80.

However, from the reported behaviour it is obvious, that the VTOR value is not *added* to ExceptionNumber*4, rather, it is ***bitwise-OR-ed*** to it. From hardware point of view this makes sense - bitwise OR is faster than addition, and also implements in a significantly lower gate count; while the result is the same, as long as the vector table fits into the "granularity" of its location.

Unfortunately, practical vector tables in any Cortex-M3/M4/M7 implementation never fits into 128 bytes. Consequence of using VTOR value not having enough zeros at its lowermost bits to fit the offset of all vectors is, that for interrupts with higher numbers, incorrect ISR will be invoked.

This fact - without explanation - is documented in ST's Programming Manuals to individual Cortex-Mx cores in the following sentence below VTOR description:

When setting TBLOFF, the user must align the offset to the number of exception entries in the vector table.

where TBLOFF is the only bitfield in VTOR.

Confusingly, the rest of VTOR description (incorrectly) states that the lowermost 9 bits are always zero, implying 128-word (512-bit) alignment requirement, which in some implementations with larger vector table is insufficient.

Created: 2.July 2022

112. After setting GPIOA things stop to work (due to overwriting SWD pins settings)

STM32 are debugged and programmed predominantly through the SWD interface, which consists primarily of two signals, clock SWCLK and data SWDAT. Probably in all STM32 these two signals are mapped to pins PA14 and PA13, and represent one of the possible alternate functions of those pins. This means, that those two pins after reset are not set as Digital or Analog Input, as most of other GPIO pins, rather, they are set as Alternate Function in GPIOA_MODER (and AF0 is selected for them in GPIOA_AFR[1], but that is also the reset value of GPIOx_AFR registers). They also have non-default settings in GPIOA_OSPEEDR and GPIOA_PUPDR registers.

If user rewrites the **GPIO_MODER** register (usually at the beginning of program) without taking into account settings of PA13 and PA14, they **cease to function as SWD**. This has the logical, but often surprising consequence, that once such program is run, the debugging interface almost immediately stops working. The chip also cannot be reprogrammed "on the fly", which is also may be surprising for those who are not aware of this issue. **It has to be reprogrammed under reset** - either by instructing the debugger/programmer to program under hardware reset, while making sure the programmer is connected to target's NRST, or by manually holding the NRST pin of the chip low.

Overwriting settings for PA13 and PA14 in GPIOA_AFR[1] would have the same consequence, but it is less likely to happen, as users would overwrite it most probably to the same value (0). Overwriting the high-slew-rate setting for PA13=SWDAT in **GPIOA_OSPEEDR** may result not only in not-working SWD, but also in **surprisingly unreliable SWD**, depending on exact details of connection and debugger's function.

Overwriting the pullups/pulldowns settings is on the other hand unlikely to have any noticeable effect with connected debugger, it mostly prevents inadvertent effects when the debugger is not connected.

Created: 9.July 2022

113. Used this Cube/SPL function and it works weird (due to uninitialized struct as parameter)

This problem is in fact a generic C-usage problem, which is linked to STM32 only coincidentally, through a programming practice favored by ST's programmers.

In the various libraries produced by ST (SPL, Cube/HAL, Cube/LL), function calls often use, as a parameter, a pointer to a struct, which contains several members detemining behaviour of the peripheral to be controlled by given function. The function then combines those struct members into one ore more values to be written into actual peripheral registers.

Sometimes users unaware of this problem **define such struct as a local variable, and they fail to fill in all its members**, leaving some members uninitialized. As local variables are located at stack, their content is not defined. This is mostly due to the fact, that users rarely review the definition of the given struct or read the documentation to functions they are using, and they usually focus only on members which are straighforwardly required. It also happens that users copy an example code, while defining their own local struct, being unaware of the fact that the original had the struct defined as global or explicitly initialized at definition, or not being aware of the distinction between the various forms of variable definiton.

Struct fields with undefined content then get combined into the resulting values written into the peripheral registers, resulting in wrong and very confusing behaviour of given peripheral.

If this problem happens, it may be not easy to debug. The wrongly set struct field may for example result in overwriting a field in target peripheral register entirely unrelated to the intended usage of the uninitialized struct member, as the value in that struct member may be outside of its intended range. Debugging of this problem is worsened by the fact that users use "libraries" often in order to avoid working with the "low-level" hardware, so they are naturally prejudiced against searching problems by looking at content of hardware registers, not even at the machine code which sets them.

Things are made even worse by the fact, that uninitialized local variables don't have an entirely random value; rather, their value depends on the previous history of program execution. Under certain circumstances, it may quite well happen, that value of an uninitialized struct member will be consistently 0 or any concrete number. It means, that the problem may result in a "mostly working", or "it works for me" kind of application.

A minor secondary issue with uninitialized struct members is, that it prevents the optimizer to do its job properly.

They are several potential remedies; however, the user has to be aware of the root problem:

- ST library functions have implemented sets of assertions, intended to ensure that no parameter departs from its intended range.
  How meticulously are these assertions implemented, is left on conscience of ST's developers.
  Also, undefined struct members may have values falling into the allowed range, yet still resulting in unwanted behaviour.
  As values of undefined struct members depend on previous run history, it may also happen, that assertions won't be able to catch such problem ever in a testing environment, resulting in unpleasant surprises when deployed.
  Moreover, using assertions is costly in terms of program memory and execution time. Granted, they can be switched on only during development.debugging, but also conceptually the idea of developing/debugging a different binary than is finally deployed, is patently wrong.
- Structs can be defined as global variables. This results in increased consumption of RAM, especially if the struct is only for a single use, which is often the case (e.g. at mcu startup).
- Structs can be defined as explicitly initialized when defined. As initialization of a single member of function causes implicit zeroing the whole struct, it is enough to initialize one member at the struct definition, other members may be assigned later in the program. This is what CubeMX-generated functions do.
- Structs can be defined as explicitly initialized constant variable. This usually results in struct being placed into FLASH, sparing precious RAM; sometimes optimizers may be able to remove the struct itself even from FLASH.
- Compound literals can be used as parameters to those functions. This is perhaps the best solution, both in terms of cost and methodology. However, compound literals are a relatively new feature (if 20+ years of C99 can be "new") and many users are simply not aware of their existence.

Since CubeMX started to generate functions with such structs explicitly initialized to {0}, frequency of this kind of problem reported on the STM32 forum significantly decreased (e.g. an example here); but this makes it perhaps even more surprising.

Created: 16.July 2022

114. Some TIM register fields are not continuous

Timers are one of the few peripherals in STM32 which is featured in every STM32 family, yet maintains a consistent set of features and registers across the entire STM32 realm (in contrast with GPIO and RTC which in 'F1 are very different from the rest, the two different incarnations of I2C, or the ADC which is different in perhaps every family). Of course, timers have also seen development, and that means, that some of the bitfields in its registers have been expanded and don't fit into the slot they have been assigned originally.

So in newer versions, ST simply added bits to these bitfields elsewhere in the same register, thus creating a non-continuous field of bits.

For examples, let's have a look at STM32G4, which is the newest STM32 family intended for mixed-signal applications e.g. motor control, where timers play a prominent role:

- TIMx_CR2.MMS occupies bits 4..6 and 25
- TIMx_SMCR.TS occupies bits 4..6 and 20..21
- TIMx_SMCR.SMS occupies bits 0..2 and 16
- TIMx_CCMRx.OCxM occupies bits 4..6 and 16 for CH1/CH3, and 12..14 and 24 for CH2/CH4

These additional bits are of course OK, but an unaware user might be caught by surprise, if trying to use them as a continuous field. And, unfortunately, not all RMs point out clearly this fact in the description of given bitfield, e.g. as of writing this text, RM0316 rev.8 for the STM32F303, which was probably the first family to feature 4-bit TIMx_SMCR.SMS).

When writing to these bits, the usual AND/OR/SHIFT scheme still can be used - and the CMSIS-mandated device headers indeed define the respective masks to conform to this scheme, e.g.

```
  #define TIM_SMCR_TS_Pos           (4U)
  #define TIM_SMCR_TS_Msk           (0x30007UL << TIM_SMCR_TS_Pos)              /*!< 0x00300070 */
```

however, the bitfield values have to be constructed equally carefully

```
// legacy TS values
  #define  TIM_SMCR_TS__ITR0                  0  // internal interconnections, concrete values defined elsewhere
  #define  TIM_SMCR_TS__ITR1                  1
  #define  TIM_SMCR_TS__ITR2                  2
  #define  TIM_SMCR_TS__ITR3                  3
  #define  TIM_SMCR_TS__TI1F_ED               4  // both edges on CH1 input
  #define  TIM_SMCR_TS__TI1FP1                5  // TI1FP1 with the same polarity as if CH1 used for capturing in CC1
  #define  TIM_SMCR_TS__TI2FP2                6  // TI2FP2 with the same polarity as if CH2 used for capturing in CC2
  #define  TIM_SMCR_TS__ETRF                  7  // external trigger input
// extended TS values
  #define  TIM_SMCR_TS__ITR4                  ((1 << 20) + 0)  // internal interconnections, concrete values defined elsewhere
  #define  TIM_SMCR_TS__ITR5                  ((1 << 20) + 1)
  #define  TIM_SMCR_TS__ITR6                  ((1 << 20) + 2)
  [... etc....]

// product/family-specific interconnects for 'G4
  #define  TIM2_SMCR_TS__TIM1_TRGO                TIM_SMCR_TS__ITR0
  #define  TIM2_SMCR_TS__TIM3_TRGO                TIM_SMCR_TS__ITR2
  #define  TIM2_SMCR_TS__TIM4_TRGO                TIM_SMCR_TS__ITR3
  #define  TIM2_SMCR_TS__TIM5_TRGO                TIM_SMCR_TS__ITR4
  #define  TIM2_SMCR_TS__TIM8_TRGO                TIM_SMCR_TS__ITR5
  [... etc....]

// and this allows to write
  TIM2->SMCR = (TIM2->SMCR & ~TIM_SMCR_TS_Msk) | (TIM2_SMCR_TS__TIM5_TRGO << TIM_SMCR_TS_Pos);
```

A maybe less important implication of these extra bits is, that they change the respective registers from 16-bit to 32-bit. This has to be borne in mind for example if the registers are written using DMA.

Created: 23.July 2022

115. Incorrect values from internal temperature sensor, due to errorneous values in 'G4 datasheet

This is just a narrated version of this STM32-forum thread.

User MHerr.3 came across a problem, where calculation of temperature from readings of internal temperature sensor of STM32G474VE were relatively close to actual temperature of the chip's body measured by an external thermometer, but the higher the actual temperature rose, the more the internally read out temperature deviated from the actual one. And, unexpectedly, at higher temperatures the indicated temperatures were lower than the actual.

Around one year later, user JGosd.1 posted the solution:

The datasheet for the STM32G474 has been recently (Nov 2021) updated to Rev 6 to show TS_CAL2_TEMP as 130°C, rather than the 110°C it shows in rev5.

As these values enter the formula using which temperature is calculated from ADC readings, it explains the deviation.

As user TDK noted, using the simple temperature coefficient + offset values from Datasheet instead of the factory-calibration values stored in system memory would probably indicate the correct temperature even if with somewhat higher error, as the calibration-temperature parameter does not enter the simple calculation's formula. This would have helped to reveal the root of the problem.

This story highlights the fact that while the DS and RM are the ultimate source of information, they are not infallible, and especially with newer models, copy-paste errors are not unseen.

Created: 30.July 2022

116. RTC loses half a second per chip reset (Cube bug, fixed)

This issue has been reported/discussed on the STM32 forum many times during the last years; but this thread probably sums it the best, and also contains the report of the fix (below "More answers", if it's not there in the 10th post thanks to ST using salesforce instead of proper forum software, reload the page).

In the unmistaken logic of Cube/HAL/CubeMX, CubeMX-generated code unconditionally calls HAL_RTC_Init(), which in turn enters the INIT mode of RTC and rewrites its basic setup registers (the clock dividers). As the values written are always the same, and the date/time registers are not overwritten in HAL_RTC_Init(), the RTC seemingly continues to keep time, but the effect of entering INIT mode is, that the subseconds register gets cleared.

The net effect is, that with resets (including power cycles) occuring at random times, in average, half a second is lost, i.e. time kept by RTC lags half a second per every reset/powerup experienced.

To unaware user, this can mimic LSE frequency deviation, the user may mistakenly go the path of attempting to correct it.

In 2022, ST posted a fix to CubeF4 (which is to be propagated to other families' Cubes), where in HAL_RTC_Init(), the RTC INIT mode is entered only if INITS flag is not set. However, this may be not the best solution either, given the INITS flag only indicates that the year field in RTC_DR is different from 0, and not all applications wish to set date. It may be a better idea to use RCC_BDCR.RTCEN as basis for this test (or equivalent in STM32 families where the register controlling backup-domain clocks are named differently, e.g. RCC_CSR in 'L0).

Created: 6.August 2022

117. Current consumption in sleep modes unexpectedly high during debugging (due to debugging in sleep enabled by DBGMCU_CR bits)

All STM32 feature a set of power-saving modes. In these, clocks to various portions of the mcu are stopped, and in the lower-consumption modes, internal power regulators are switched off.

Switching off clock to the processor core would disable debugging, as the debugger modules are part of the processor core. Therefore, STM32 offer an option to keep part of the processor core running during debugging, by setting one of the related bits in DBGMCU_CR register (DBG_SLEEP, DBG_STOP, DBG_STANDBY). These are documented in the DBG chapter of RM to given STM32. Depending on particular low-power mode employed, clock to processor core may be supplied from the normal system clock, or an internal RC oscillator may be switched on for this purpose.

Using any of these options of course partially defeats the working of the low-power modes, and STM32 with these bits set have higher power consumption in the low-power modes than expected. In the lowerst-power modes, the difference is significant, milliamperes instead of microamperes may be drawn.

Sometimes users are not aware of this feature, as the sleep-related bits in DBGMCU_CR may be manipulated directly by the debugger. Usually, there are switches in the debugging environment to disable this behaviour, but they may not be displayed conspiciously.

Created: 13.August 2022

118. DWT_CYCCNT does not work in some sleep modes

DWT (Debug Watchpoint and Trace) unit is one of the debug units of the Cortex-M processor core. In Cortex-M3/M4/M7, it contains a 32-but counter, DWT_CYCCNT, primarily intended to provide a source for timestamps for the tracing functions. However, as it provides 32-bit resolution and runs at system clock frequency, and as it is accessible to the processor (with some added obstacles in the 'H7 family, but we are not going to discuss that here), users like to use it for general timing functions, too.

In this thread on STM32 forum, user VMiki.1 reported, that DWT_CYCCNT does not run, while the microcontroller is in some of the low-power (sleep) modes.

Experiment on STM32F429 revealed, that if SCB_SCR.SLEEPDEEP=0, DWT_CYCCNT continues to run at the system clock frequency even after executing WFE. If SCB_SCR.SLEEPDEEP=1, after WFE DWT_CYCCNT stops running until wakeup; but if DBGMCU_CR.DBG_STOP is set (which enables debugger to keep running while debugging sleep modes at the cost of higher power consumption), DWT_CYCCNT continues to run at cca 16MHz, probably HSI clock.

Created: 20.August 2022

119. ETH IPv4 checksum offload does not work - if checksum fields in the packet submitted to ETH/MAC are not zero

The ETH module in 'Fxx7/'Fxx9 (within 'F1/'F2/'F4/'F7 families) is an incredibly complex machine with lots of more or less useful options.

One of its features is, that it can calculate in hardware several header checksums for some of the common protocols of the TCP/IP suite. This significantly increases the performance of TCP/IP stack running through ETH, which is undoubtedly the most common usage combination.

As user SHers reported on STM32 forum, this mechanism does not work, if the respective fields in data submitted to ETH/MAC are not zero. In that case, the ETH/MAC unit, instead of calculating the proper checksum, inserts zeros in those fields.

Created: 27.August 2022

120. Unique ID (UID) is not available in some STM32 families

Most STM32 have a 96-bit UniqueID, programmed in factory into in the system ROM. To explain, what the "uniqueness" of this number means, ST is saying this:

The 96-bit unique device identifier provides a reference number which is unique for any device and in any context.

While bold, this is not very specific. Nevertheless, the UniqueID is surely unique within specimens of the same STM32 model, and most probably unique amongst all STM32, too.

However, not all STM32 do have the UniqueID. This is especially true for the Value-line devices, i.e. those STM32 for which the base device number ends with 0 (i.e. STM32xxx0). The Reference Manual for these devices then does not contain the *Unique device ID register (96 bits)* subchapter of *Device electronic signature* chapter.

Some of these devices, confirmed by ST not to have UniqueID available, are for example STM32F070 and STM32F030.

Note, that this does not mean, that these devices may not have some data stored at addresses which are expected to hold the UniqueID in analogous devices of the same family. Rather, the number not guaranteed to be there or to be unique.

This is one of the drawbacks of using these cheaper devices, and an unpleasant surprise for those who wish to move from "standard" to value-line devices.

Created: 3.September 2022

121.96-bit Unique ID cannot be shortened, and still considered unique

The UniqueID number factory-programmed to the system ROM of STM32 promises to be a valuable way to distinguish individual devices, e.g. when addressing them on multi-drop communication bus. However, the uniqueness is bound to a relatively long 96-bit (12-byte) number, while for most practical uses a shorter, 24- or 32-bit "serial"/unique number is required.

Users are thus tempted to somehow reduce the UniqueID, while still maintain its uniqueness. One attempted method is to generate a shorter "hash" of the UniqueID, using whatever method available, CRC being a popular option. This, of course, does not provide unique-ness by any means, as no hash does, although the resulting output appears to be tantalizingly "random".

Other users noticed, that the UniqueID contains structured data - and indeed, some RM even explain that the UniqueID consists from the X/Y position of given die on the wafer, of the wafer number within lot, and the manufacturing lot marking. Clearly, this results in much less information than 96 bits. The X/Y position and wafer number appears to be stored as ASCII numbers. However, this is not documented by ST, and also there is no description on what are the expected ranges of these numbers, nor the structure of the lot identifier. So, at the end of the day, the whole 96 bits have to be treated as a black-box, providing unique identification only if taken as it is.

So, there's no good way to reduce the UniqueID length. If a shorter serial/unique number is required for whatever purpose, users have to generate them themselves by the usual methods, at the moment when the STM32 is being programmed at device manufacturing. Some STM32 families provide a OTP (One-Time Programmable) section of memory, which is a good place to store this kind of numbers/information.

One notable exception, where the 96-bit UniqueID can be used as it is, is in the role of Serial Number Descriptor of USB Device. This is transmitted and interpreted as a string, so its particular length is not critical.

Created: 10.September 2022

122. In some STM32 families, Unique ID is not at continuous addresses in memory

In most models, the UniqueID occupies a continuous piece of system memory. However, there are models ('L1, 'L0, but maybe there are other such), where UniqueID consists of several pieces which are not after each other.

In such models, if user is not aware of this gotcha and reads out 96 bits from system memory continuously starting at the given base address, some bits - which are not part of the UniqueID - may contain the same value (e.g. all-FFs) for several or all STM32 of the same model. As all 96 bits are needed for the uniqueness, this can lead users to a mistaken conclusion, that the UniqueID is not that unique, as is described e.g. here.

Following is a very incomplete table of placement of UniqueID in a random selection of STM32 models:

STM32 (sub)family	RM	UniqueID base address	offset 2	offset 3
'F0	RM0091	0x1FFF'F7AC	0x04	0x08
'F1	RM0008	0x1FFF'F7E8	0x04	0x08
'F3	RM0316	0x1FFF'F7AC	0x04	0x08
'F4	RM0090	0x1FFF'7A10	0x04	0x08
'F7	RM0385	0x1FFF'F420	0x04	0x08
'G0	RM0444	0x1FFF'7590	0x04	0x08
'G4	RM0440	0x1FFF'7590	0x04	0x08
'H7	RM0433	0x1FF1'E800	0x04	0x08
'L0	RM0367	0x1FF8'0050	0x04	0x14
'L1 Cat.1/2	RM0038	0x1FF8'0050	0x04	0x14
'L1 Cat.3/4/5/6	RM0038	0x1FF8'00D0	0x04	0x14
'L4	RM0351	0x1FFF'7590	0x04	0x08
'L5	RM0438	0x0BFA'0590	0x04	0x08
'U5	RM0456	0x0BFA'0700	0x04	0x08

Created: 17.September 2022

123. INITS flag may not be the best indicator of RTC having been already initialized

To determine, whether RTC has been already initialized or not (e.g. to start/initialize the RTC at the first run of device or after backup battery change), RTC offers the INITS flag. It is a read-only bit of the RTC_ISR register in RTC v2, quite unnecessarily renamed/reorganized in RTC v3 as RTC_ICSR (e.g. in 'G0/'G4).

However, the INITS flag only indicates, that the year field in RTC_DR register is nonzero.

As user Alec Davis noted in this thread, if date (thus year) is not set during RTC initialization (e.g. if the user does not care about date), or if year 2100 is reached, INITS will incorrectly indicate, that RTC has not been initialized yet.

So, for this purpose, it is probably better to use the RTC_BDCR.RTCEN flag (or its equivalent RCC_CSR.RTCEN in 'L1/'L0).

Created: 24.September 2022

124. Calibration values (UniqueID, temperature, internal VREF) are not at the same address - not even within the same family

When manufacturing the STM32, at the testing step, certain unique numbers are programmed into the system memory of individual chips. Among these are the UniqueID; individually measured temperature sensor readouts at two temperatures (TS_CAL1 and TS_CAL2); and readout of VREFINT, the internal voltage reference (VREFINT_CAL).

Some STM32 don't have some or all of these informations programmed at manufacturing. These are usually models of the "Value lines" (those with the principal model number ending with 0, e.g. STM32F03**0** or STM32F73**0**), where the price decrease is achieved mostly by sparing down testing steps.

Note, that where available, these individual numbers are placed at different addresses within the system memory for different STM32 models. Even individual models within one family may have these addresses different, so it is important to check these addresses in the **datasheet for the particular STM32 model** and set them properly, when migrating related code from one STM32 model to other.

Created: 1.October 2022

125. USB/DFU bootloader in 'F4/'F7 may start unreliably with higher frequency crystals

The default bootloader in system memory conveniently provides several ways to load a program into an STM32. In most models with USB, the bootloader incorporates also an option to use USB DFU bootloader.

However, as USB requires a higher clock precision than that provided by the internal RC oscillators of STM32 (HSI, MSI), in models which don't support "crystal-less USB", the DFU bootloader requires a crystal to be connected to the HSE pins. Frequency of the crystal cannot be arbitrary, and AN2606 describes the particular requirements on crystal frequencies for individual STM32 models.

For the 'F4/'F7 families, the crystal has to have frequency which is integer multiple of 1MHz, between 4MHz and 26MHz. The exact mechanism how bootloader detects, which of the possible crystals is present, is the following:

- use HSI with PLL to set system frequency to 24MHz
- start HSE, set the HSE-to-RTC prescaler to 31 (maximum value of RCC_CFGR.RTCPRE)
- enable TIM11 clock in RCC (APB dividers are 1 so timer clock is 24MHz), in TIM11_OR, set TIM11_CH1 to be connected to HSE-RTC
- set it to run freely and capture CH1 with capture prescaler set to 8 (maximum)
- capture two consecutive HSE/31/8 edges, calculate difference of captured values D
- calculate 24'000'000/D*31*8, that's the measured HSE frequency
- divide by 100'000, add 5, divide by 10, use result as M in setting up PLL to generate 60MHz system clock and 48MHz USB clock from HSE

This method involves several rounding and granularity errors. As the basis of measurement is the ratio between HSE/HSI, the errors get more pronounced as this ratio decreases i.e. HSE frequency increases. As long as HSI is within the 1% precision given by factory calibration, this method correctly measures all HSE values between 4MHz and 26MHz. However, the 1% factory calibration is guaranteed only for temperature of 25°C. HSI frequency varies quite significantly with temperature (which is the temperature of the silicon die, and that may be surprisingly higher from ambient by quite a couple of degrees very quickly).

At around 2% of HSI error, measurements of the highest frequencies (25MHz, 26MHz) start to fail. The consequence is, that the bootloader sets up PLL incorrectly, thus USB module is not clocked at 48MHz and the difference is too big for USB to work successfully.

ST recognizes this problem, as in AN2606, the descriptions for 'F4 and 'F7 bootloaders contain the following footnote:

Due to HSI deviation and since HSI is used to detect HSE value, the user must use low frequency rather than high frequency HSE crystal values (low frequency values are better detected due to larger error margin). For example, it is better to use 8 MHz instead of 25 MHz.

Indeed, the measurement for 8MHz HSE is good up to cca 6.5% error of HSI. For the maxumum error of HSI given by datasheet as -8% at the highest temperature range, only 4MHz and 5MHz are "safe" values.

This issue has been mentioned here, in the context of using the DFU bootloader as indication of whether the hardware USB connection for a 'F411 is OK. Interestingly - and unfortunately - this problem with the USB/DFU bootloader makes this an unreliable "test"/indicator, if a higher-frequency crystal is used (25MHz in this case). The original manufacturer of the popular "Black pill" board in question also recognizes the problem and recommends to "heat up" the chip in case of problems with the DFU bootloader to 25°C.

The problem of granularity of measurement in bootloaders is not new/unique. Here for example is description of autobauding problem in bootloader of a venerable 8051 derivative, resulting in similar unreliability of starting bootloader, puzzling users maybe 20 years ago...

Created: 8.October 2022

126.Not all STM32F730 are created equal, and some don't have the built-in HS USB PHY

The STM32**F722** is a "lower-end", but still very capable member of the 'F7 family, featuring the Cortex-M7 core. As other "performance" STM32 ('F2, upper-end 'F4, all 'F7 and 'H7), it also features two USB OTG modules, which both have connected a built-in Full-Speed PHY, but one of them - usually marked OTH_HS[1] - has also an ULPI interface, to which an external High-Speed PHY can be connected.

In the STM32**F723**, there is an internal High-Speed PHY, connected to this OTG_HS module internally. This makes it the only STM32, which offers a native single-chip HS USB solution.

ST also makes a "Value-line" version of these chips, marked STM32**F730**. As with other "Value-line" models, they are simply the same chips which has seen less testing at smaller temperature range, so parts of memories and some peripherals are not guaranteed to work and thus are effectively unavailable. It is not uncommon for ST to offer several models of the same family under one "Value-line" model number - for example, the STM32F030 consists in fact of 3 different models, depending on their memory size.

Nonetheless, in this case, the difference is quite significant - in STM32F730 in **64-pin and 100-pin** packages the **internal HS PHY is absent**, so they are equivalent of the 'F722; whereas in the **144-pin and 176-pin** packages the **internal HS PHY is present**, so they are equivalent to 'F723.

---

1. ST does not maintain consistent marking of modules across families, for example in the 'H743 RM it calls both OTG modules as OTG_HS, despite the fact that one of them is capable only of FS, exactly as in other families.

Created: 15.October 2022

127.Built-in USB HS PHY in 'F723/'F730 requires certain crystal frequencies

The STM32F723 (and it's "value" counterpart, the 'F730, but only in the 144- and 176-pin packages), has a built-in High-Speed USB PHY, connected to the OTG_HS module internally.

This PHY needs 60MHz and 480MHz clocks, which it generates internally directly from HSE. Unfortunately, this fact is not described in the dedicated USBPHYC chapter in RM0431 as it ought to be (instead, this chapter confusingly states both in narrative and in the schematic diagram, that only APB clock PCLK2 is connected to it). The real connection from HSE to the USB PHY's internal PLLs is depicted on the Clock tree diagram in RCC chapter.

As the USBPHYC_PLL1.PLLSEL description states, there's only a selected set of HSE frequencies from which the PHY's PLLs can generate the required 60MHz/480MHz clocks: 12MHz, 12.5MHz, 16MHz, 24MHz and 25MHz.

This means, that using the 8MHz crystal which is so widely used on ST's devboards won't allow to use the internal HS USB PHY in the 'F723. This same limitation together with bad design decisions also resulted in the STLink3 not being able to supply stable 8MHz clock to its connected target STM32.

Created: 22.October 2022

128. In 'F2, subseconds don't work in RTC, nor smooth calibration

The STM32F205/207 models were concieved as improved versions of the then high-end STM32F105/107 models. The 'F2 received improvementes in many peripherals (e.g. the GPIO busmatrix), more FLASH and RAM (including battery-backup SRAM) and higher clock frequency.

Soon after STM32F205/207, the STM32F405/407 appeared, boasting an improved Cortex-M4 core with single-precision floating-point unit, and even more increased clock frequency. Contrary to the 'F4 line, the 'F2 line has never seen any more newer/better models; but given that the 'F2 is in almost all aspects (except the core and clock frequency) identical to 'F4, the 'F2 is considered a "somewhat worse 'F405/'F407" since then.

However, those who "downgrade" to the 'F205/207 from 'F405/407 for cost, availability or any other reason, may stumble upon one of the rare differences in peripherals of those two models: the lack of subseconds in the 'F2 RTC. **Subseconds** (and the related registers for alarm subseconds etc.) **are simply not implemented in the 'F2 RTC**.

Another relatively often used feature also not implemented in the 'F2 RTC is the smooth calibration. Btw. this may be the reason why coarse calibration *is* implemented in 'F4 - for upward compatibility.

There are more "defficiencies" in the 'F2 RTC as compared to 'F4 RTC, in the less-used functionalities: missing 1Hz calibration output, synchronization feature, shadow registers bypass, filtering/sampling on tamper input etc. Similar "reduced capability" RTC module is in the Cat.1 'L1 family, but that's less likely to cause surprise when migrating, as 'L1 "subfamilies" have higher mutual differences in several other peripherals/aspects than 'F4 vs. 'F2.

AN4759 contains an excellent **overview of capabilities of RTC** modules implemented in various STM32 families (except the simplistic rev.1 RTC in the ancient 'F1 family).

Created: 29.October 2022

129. In 'F1, PA15/PB3/PB4 don't work (they are assigned to JTAG by default)

In the STM32F1xx family, behaviour of IO pins - whether they are digital Input, Output, are connected to some peripheral, or act as Analog - is given by setting them in the GPIO module. However, setting PA15/PB3/PB4 in GPIO alone does not work.

The reason is, that by default these pins are part of the JTAG debug interface[1]. This is the same as in all other STM32 families - but the surprising thing for those who migrate to 'F1 from other STM32 is, that the GPIO setup does not override the default settings. Instead, users have to **remove mapping of these pins to JTAG usage by setting the AFIO_MAPR.SWJ_CFG bitfield**[2].

There is another group of pins related to the TRACE debug functionality - PB3 (in a different function than when used as part of JTAG) and PE2..PE6 - where mapping these pins to TRACE (by setting DBGMCU_CR.TRACE_IOEN and DBGMCU_CR.TRACE_MODE) overrides the GPIO setting. However, these bits are by default mapped to GPIO, so this does not cause any unpleasant surprise.

---

1. Also pins PA13 and PA14 are part of SWD/JTAG interface and mapped as such, but that's less surprising as usually the chips are debugged/programmed through SWD so these pins often remain dedicated to this functionality. However, if users wish to use these pins as GPIO, they have to set AFIO_MAPR.SWJ_CFG accordingly, too.

2. Working with AFIO_MAPR.SWJ_CFG bifield, and consequently AFIO_MAPR register, has its own set of issues.

Created: 5.November 2022

130. In 'F1, PA15/PB3/PB4[1] stop working after remapping unrelated peripherals

In STM32F1xx family, there is no GPIO AF matrix as in other STM32 families. Instead, the mapping of peripherals to individual pins is more-less fixed. However, some peripherals have a limited flexibility in that their internal IO pads can be remapped to a different set of pins than is the default mapping. This is accomplished by writing to the respective bit-field in AFIO_MAPR register[2].

Besides these remaps, AFIO_MAPR contains also the SWJ_CFG field intended to remap pins to the JTAG/SWD debug interface. Surprisingly, these SWJ_CFG bits are write-only and read as zero. This means, that any **read-modify-write (RMW) operation on any other field in AFIO_MAPR writes all SWJ_CFG bits to zero, and thus remaps all JTAG/SWD pins to their debug function**.

As bit-banding is by hardware converted to RMW, this problem occurs also if the periphepral remap bits are adjusted using bit-banding.

Solution is, that each time AFIO_MAPR register is to be written, the SWJ_CFG field has to be explicitly set to the bit combination required by the user for the affected pins to work as intended.

This has been brought up here and subsequently implemented by ST in Cube.

1. This problem pertains to PA13/PA14, too, as they are also part of SWD/JTAG interface remapped by AFIO_MAPR.SWJ_CFG, but this is less likely to be observed as those pins are usually dedicated to SWD debugging/programming.

2. A few peripherals remap using AFIO_MAPR2 register, these remaps of course don't cause the problem described here.

Created: 12.November 2022

STM32 gotchas

131.Undefined AF in GPIO may turn pin to output

STM32 GPIO (except in the 'F1 family) features a matrix, through which for individual pins, if set to Alternate Function in the respective GPIOx_MODER field, one of the available peripheral functions for that pin is connected. This selection is performed by a field in GPIO_AFRL/AFRH registers.

GPIO_AFRL/AFRH has 4 bits per pin, so there are maximum 16 peripheral functions per pin available. However, the matrix is sparse, and there is probably no single pin in any STM32 which would have all 16 functions implemented.

The original intention was, that each of the 16 options will be assigned to a group of functions - e.g. AF0 is for system functions such as JTAG/SWD or MCO, AF1 is for TIM1, AF2 for TIM2-TIM5, AF4 for all the I2C, etc. This scheme can clearly be seen in 'L1, where the bus matrix was first introduced, and the also for the first 'F2/'F4; but starting with 'F0/'F3 the functions got more mixed up; and by now in the newer STM32 if there still is some hint of a scheme it has so many exceptions that it can't be relied upon anyway and each function's AF for a pin has to be looked up individually.

What happens, if one inadvertently[1] assigns a "reserved" AF number to a pin set to Alternate Function?

This (similarly to other details of GPIO matrix implementation) is not documented. However, experiments indicate, that the matrix has been optimized during the design process, and there is no "neutral/parking" position of the matrix multiplexer, rather, if an "unassigned" AF number is selected in GPIOx_AFR[], then one of the "existing" peripheral connections is chosen. And, if that peripheral pad happens to be an output or switched to output, it may hapen, that the pin in question turns to output, resulting in surprising behaviour and potential damage.

---

1. For example, if pin is set to Alternate Function in GPIOx_MODER, before the respective GPIOx_AFR[] field has been set. Default value of the majority of GPIOx_AFR[] fields is 0b0000 i.e. AF0 is selected, which may be "unassigned" for given peripheral. How do I know? :-)

Created: 19.November 2022

132. DMA vs. DMAMUX numbering discrepancy

In various STM32 families, there are two[1] types of DMA: single-port and dual-port. Here, we are not going to discuss all the differences between these two types - which there are quite a few - only a confusing numbering issue.

Both DMA types consist of several (6-8) transfer elements and common steering logic. Each transfer element consists of holding register or FIFO for transferred data, source/destination address register and number-of-transfer register, and logic to load/increment these registers when a request (trigger) arrives for this transfer element. The common logic provides access to external bus/buses through which DMA reads/writes data, and arbitrate between transfer elements in case of simultaneous requests.

In **single-port DMA**, transfer elements are called **Channels** and they are **numbered starting at 1**. Individual requests from several periferals for one transfer element are either simply ORed together (where it's the user's responsibility to enable DMA only in one of the peripherals), or there is a multiplexer controlled by a separate register in the DMA.

In **dual-port DMA**, transfer elements are called **Streams** and they are **numbered starting at 0**. Requests to one stream are selected using a multiplexer, which is controlled by a field in the given stream's control register. Confusingly enough, individual requests are called channels...

Starting from the 'L4+ family, a new feature appered: the DMAMUX unit, which is in fact a DMA request multiplexing matrix with some additional functions such as synchronization and repetitions. The nature of matrix means, that each of the DMA transfer elements can be triggered by any of the dozens of available requests, so there is no more limitation of conflicting requests in one DMA transfer element.

The output side of DMAMUX consists of channels, with one channel for each DMA transfer element. However, DMAMUX channels are **numbered starting at 0**.

As most of the models with DMAMUX have the single-channel DMA, there is a confusing mapping from DMAMUX channels to DMA channels: DMAMUX ch0 triggers DMA ch1, DMAMUX ch1 triggers DMA ch2, etc.

There are STM32 models with DMAMUX and two DMA ('L4+, 'L5, 'G4 and high-end 'G0 ('G0Bx)). The channels of single DMAMUX are then split to serve the two DMAs. The DMAMUX channel numbering is continuous, so it depends on number of channels in the DMAs in individual models, how exactly the mapping splits. For example, in 'L4R5, DMAMUX channels 0-6 trigger DMA1 channels 1-7 and DMAMUX channels 7-13 are connected to DMA2 channels 1-7.

The 'H7 comes with an added bonus, as it contains both single- and dual-port DMA (plus the MDMA mentioned in footnote). It has two DMAMUX units - DMAMUX1 serves the dual-port DMA1 and DMA2, where DMAMUX1 channels 0-7 trigger DMA1 streams 0-7 (making it the only case where the numbering matches) and DMAMUX1 channels 8-15 trigger DMA2 streams 0-7; whereas DMAMUX2 serves the single-port BDMA, DMAMUX2 channels 0-7 triggering BDMA channels 1-8.

---

1. In 'H7, there is a third type, MDMA, with complexity shooting over the roof (boasting among other things scatter-gather capability). In 'U5, there are two more types, GPDMA and LPDMA, both being significantly different from the two "original" DMA types present in all other STM32 families. Neither of these uses DMAMUX.
In display-oriented STM32 (usually those which have also LTDC) there's a DMA2D unit; but that is not considered here as it is always triggered by software, rather than by any signal from other modules.

Created: 26.November 2022
'L5 and high-end 'G0 added to models where two DMA share one DMAMUX: 26.August 2023

133. PA2 unexpectedly outputs a 32kHz signal? RTC behaves strangely? Maybe consequence of VBAT brownout.

In STM32G071 errata (ES0418), the following erratum appeared, dated 17-Oct-2022:

Corrupted content of the RTC domain due to a missed power-on reset after this domain supply voltage drop

The RTC domain reset may be missed upon a power-on following a power-off, if its supply voltage drops during the power-off phase hitting a window, which is few mV wide before it starts to rise again. In this critical window, the flip-flops are no longer able to safely retain the information and the RTC domain reset has not yet been triggered. This window is located in the range between 100 mV and 700 mV, with the exact position depending mainly on the device and on the temperature.
This missed reset results in unpredictable values of the RTC domain registers. This may cause a spurious behavior (such as driving the LSCO output pin on PA2 or influencing RTC functions).

It is not clear, which STM32 are affected, but based on anecdotal observation of "mysterious failures" which match this description, reported on the forum, it at least affects also some of the STM32L4. A similar report by user arnold_w indicates, that 'F446 is likely be affected, too - and in this case, it affected PA0. ST does not always propagate such finding to all affected models' errata timely, so it may quite well be that all STM32 are affected.

What might be the worst-scenario consequences of this issue?

It appears, that if the VBAT brownout (i.e. VBAT voltage decreasing to the critical window and then rising again) occurs, the backup-domain[1] registers' content may end up in any random state. These registers are mainly registers of RTC (and TAMP in families where it's a separately described unit), but also the backup-domain control register, RCC_BDCR[1]. It may well be that in certain STM32 a few spurious bits scattered across peripherals also belong to backup-domain, always check your STM32 RM.

In applications, which do use the backup-domain facilities (RTC, backup registers, LSE to clock asynchronous peripherals such as LPTIM or LPUART) the consequences of spurious change of these registers/bits is obvious. However, there are also consequences to applications which don't use the backup-domain facilities, or use only some functionalities of them while the spurious change sets other functionalities.

One such consequence may be unexpectedly higher power consumption, e.g. due to inadvertently enabled LSE; other may be unexpected interrupt or register clear through tamper input. But maybe the most dangerous consequence is, that some backup-domain registers bits enable to override the GPIO settings of certain pins, changing their intended functionality or even causing hardware conflict by unexpectedly turning them to output. Some of such pins are PC13 (the common RTC-output-tamper-etc. pin), PC14/PC15 (OSC32 pins for LSE), PA2 (LSCO pin), and they may be more - the *Additional functions* column of *Pin definitions* table in DS to given STM32 model may be helpful in identifying such pins.

There are three groups of applications, regarding VBAT connection/usage:

1. **Backup domain (mainly RTC) is not used at all**, VBAT connected to VDD either through internal bond in the low-pin-count devices, or externally as per datasheet guidance.
   The recommended workaround is to **explicitly reset backup domain** (using RCC_BDCR.BDRST). This is probably the most common usage model, and perhaps the **most dangerous** for this particular issue, as users who don't use actively backup-domain (RTC) features are **not likely to associate any problem with backup domain/RTC**, nor to actively seek issues related to the backup domain/RTC. Thus, the problem violates the unwritten "if you don't use it, simply don't touch it, the default state is OK" rule.
   I expect ST will be adding such reset quietly to all CubeMX-generated code which did not have backup-domain facilities clicked, but users who don't program by clicking in CubeMX ought to be warned, too. Given above analysis the erratum is not appropriate enough for this, ST should consider adding conspicuous warnings directly to RM (probably to the Reset subchapter of RCC chapter).
2. Backup domain (mainly RTC) is used **only while VDD is present**, VBAT connected to VDD in the same way as above.
   The recommended workaround is again explicit backup-domain reset in the startup code, however with one exception: if Standby mode is utilized, the user may want to avoid the backup-domain reset. The erratum in ES0418 contains a long, winding and relatively complex description of the possible workaround (i.e. discrimination of power-on reset from reset-to-exit-standby).
3. Backup domain is **actively used with battery (or capacitor/supercap) connected to VBAT**.
   This case is not mentioned in the erratum, but this may be one of the most problematic cases, especially if the battery is not maintained actively or if a capacitor (often supercap) is used in place of battery on VBAT. In case of battery, using only batteries from reputable sources with well-known characteristics, regular measurement of VBAT voltage through the facility built into ADC, and/or counting usage time and estimating future lifetime may be part of the solution. In any other case,

comprehensive registers checksumming, estimates based based on on- and off-time timing, and similar techniques can/should be employed.

---

1. The STM32L0 family does not have a separate VBAT pin, nonetheless it does have a supply-domain which behaves in the same way as the backup-domain in other STM32 families - here it's called *RTC domain*. Into this belong the RTC registers, and selected bits in RCC_CSR ('L0 does not have a separate RCC_BDCR register). It's not clear, if 'L0 is affected by the same brownout problem, but pending further information from ST it's reasonable to err at the safe side and assume that it is.

Created: 4.December 2022

134. Internal VREFBUF reference not available in lower-pin-count packages

Reference voltage to the ADC[1] in STM32 comes from a dedicated VREF+ pin[2]. Users are expected to provide a stable and clean reference voltage to this pin from some external source.

In newer STM32 families (starting with 'L4[4]), there's an optional internal reference voltage source, VREF**BUF**[3], which can typically provide two output voltages, nominally 2.048V and 2.5V[5] (provided VDDA is high enough).

If VREFBUF is enabled, its output is connected directly to VREF+, internally.

In most (although not all) STM32 models in smaller packages (typically below 64 pins), VREF+ is internally bonded to VDDA, to allow for more GPIO pins. In such models, of course, VREFBUF output would clash with VDDA, so it is not allowed to switch it on, effectively making it unavailable.

---

1. ... and some other analog-related functions...

2. This has also a counterpart, VREF-. However, this pin is expected to be at VSSA (analog ground) potential all the time, and in all but the largest packages it is indeed internally bonded to VSSA.

3. Not to be confused with VREF**INT** reference source, which is present in all STM32, connected internally to one of the ADC input multiplexer's inputs. VREFINT thus provides only an indirect reference in case that VREF+ voltage is relatively stable but unknown (e.g. coming from a battery).

4. Not all newer models have VREFBUF, e.g. some Value Line models (i.e. those with primary marking ending with 0) don't. Check your STM32 model's Datasheet.

5. Newer STM32 families/models provide additional/different voltage levels, e.g. 1.5V and 1.8V. Check VREFBUF characteristics section in your particular STM32 model's Datasheet.

Created: 10.December 2022

135.Using the internal VREFBUF reference requires a specific value capacitor

VREFBUF in the newer STM32 families provides a convenient reference voltage source for ADC and other analog functions. Where available, it is internally connected to the VREF+ pin, which otherwise would serve as an input to external voltage reference.

As user _andreas pointed out, using VREFBUF requires to have a capacitor connected between VREF+ pin and (analog) ground[1]. Datasheets call for a rather specific value of this capacitor, 0.5 µF - 1.5 µF, typ. 1 µF. The value of this capacitor determines also the startup time of VREFBUF. In parallel to this capacitor, the datasheet also requires a 100nF capacitor to suppress high-frequency noise.

---

1. Some STM32 families - namely 'L1 and 'L0 - **don't** have VREFBUF, but still have some form of VREF output, marked VREF_OUT. This is a buffered version of the internal VREFINT reference, which is not connected to VREF+ pin, but brought out through switches (controlled by COMP_CSR.VREFOUTEN and RI_ASCR1.CH8/CH9 in 'L1; and by SYSCFG_CFGR3.SEL_VREF_OUT in 'L0) to one or two GPIO pins. According to datasheets, VREF_OUT has only 1uA current capability, and must not be loaded by more than 50pF. This may cause some confusion when migrating across STM32 families.

Created: 17.December 2022

136. Cannot reprogram FLASH (after inadvertent write access to FLASH area, and if Cube/HAL is used)

In a fairly complex piece of software, due to a bug, a pointer variable was zeroed inadvertently. This pointer normally serves to access a struct in RAM, so after this event, inadvertent writes started to occur to low addresses, just above 0x0000'0000. It means, processor attempted to write into FLASH, which was mapped to this area as it is by default. These writes of course did not modify the content of FLASH, as these were plain writes, no proper FLASH-write sequences.

Upon these writes, the FLASH controller set FLASH_SR.PGSERR, as it considered these to be FLASH-write-sequencing errors - exactly as documented, since FLASH_CR.PG/FSTPG had not been set previously. Depending on particular write width and address, FLASH_SR.SIZERR and FLASH_SR.PGAERR were set, too.

In other parts of the program, data were to be written into FLASH. For this, Cube/HAL functions were used: HAL_FLASHEx_Erase() and HAL_FLASH_Program(). These both call FLASH_WaitForLastOperation() as the very first thing, and don't proceed if it does not return HAL_OK. FLASH_WaitForLastOperation() is there to wait for any outstanding FLASH operation from previous run, so it loops with a timeout waiting for FLASH_SR.BSY to clear. But then, for good measure, it also checks wheter some of the programming-failure-indicating flags in FLASH_SR is not set - and since they were set, it returned HAL_ERROR.

The net effect was, that "legitimate" writes to FLASH stopped to work.

This issue has been observed on STM32L431. Other models/families may behave differently.

Created: 24.December 2022

137. Communication errors in prototype on breadboard/with flying leads - on importance of signal integrity and proper ground/return

A popular prototyping technique, especially among hobbyists, is to use development boards such as Nucleo or Disco, which have brought out mcu pins onto pin headers, and connect them to peripheral circuits (often mounted on similar development boards with pin headers) using flying leads. Sometimes a breadboard is also used as a basis for such prototype circuit.

Sometimes users complain of surprising/spurious errors in communication with the connected peripheral. In extreme situations, these problems occur also when the circuit is designed on a PCB.

Often, the common denominator to these errors is improper signal ground/return, less commonly other problems with signal integrity (crosstalk/interference). Users are usually not aware of the source of problem, and they often expect (not unreasonably) the cause of it being error in software.

While using the concept of ground as a single reference potential is a technique which simplifies understanding and thus design of circuits a lot, at the same time it is also one of the most common source of "second order" errors as described. Ground interconnections in real world have nonzero impedance (in both its real and imaginary part), thus any current flowing through them results in voltage (potential) difference, exactly as Georg Ohm ruled. Long wires, multiple and often poor quality or worn out connectors increase the impedance, while high currents and conducting currents from several sources through a single interconnection contribute to the current-part of equation.

A good common practice (besides liberal application of decoupling capacitances) is to use as short interconnections as possible, and to have a dedicated ground/return wire for each signal wire (or at least for a small group of them), routed next to the signals. Ideally returns should be routed between signals, or twisted together with individual signals, to provide shielding, i.e. decrease mutual capacitive couplig between the signals.

Also note, that long interconnections tend to have relatively high imaginary component of the impedance, in other words, they are troublesome with high frequency signals. Also note, that the "high-frequency-ness" of digital signals is not given by their digital repetition rate, but rather, by the slew rate of the edges, which in STM32 is given by setting of the respective GPIO_OSPEEDR register. In other words, never use higher signal "speed" than necessary.

While improper ground/return influences all signals, with interference it's slightly different: a signal is more prone to interference if it's source is of relatively high impedance. This is one of the reasons why we see relatively more complaints of this kind with I2C connections, as impedance of I2C signals (in the "recessive state) is given by the pullup, which is usually in the order of kiloohms, rather than impedance of an output driver, which is usually in the order of ohms (this is compounded with the unfortunate fact that I2C, while conceptually simple, is relatively hard to get right in all its details, and errors in both hardware and software are far from being rare).

Another (and maybe extreme) example of a quite often used high impedance circuit is the LSE crystal oscillator, which while it's probably never brought out of a single PCB, may be too easily influenced by improperly routed grounds and high current high frequency/slew-rate signals routed at its vicinity.

Whatever the source of faulty signals is, some communication protocols are inherently less prone to errors than others. Namely, UART with its self-clocking and multiple-sampling principle is far less vulnerable to "random noise" than say SPI, where a single spike on clock throws the whole communication astray. However, this also goes hand-in-hand with decreased maximum data rate. I2C occupies a middle ground here, as there are both analog and digital filters on its receiver; however, as is the usual case with I2C, it takes a relatively high understanding of the underlying principles and thorough reading of the manual to properly and fully employ all the benefits these features provide while avoiding all the pitfalls at the same time.

A random selection of related threads on STM32 forum: here, here, here.

Created: 31.December 2022

138.Built-in bootloader works unreliably or not at all, if there is activity on other bootloader-related pins

All STM32 have a built-in bootloader, which facilitates simple firmware update through one of several communication interfaces. This bootloader - which is a program residing in the "system memory" (part of FLASH, written in factory and not rewritable by users) - is started upon system reset, when certain pins or option bits are set in a certain way.

The exact combination of pins and option bits needed to start the bootloader is specific for each STM32 (sub)family and described in AN2606. This application note also contains the basic information about interfaces and pins available for bootloading (and also some other related information, such as internal resources used, or value of crystal/oscillator required for USB/CAN bootloaders, etc.). Details of communication protocols used by individual interfaces are published in separate application note for each interface.

The key information here is, that after the bootloader starts, it does not know, which interface is going to be used. Thus, it initially just "listens" for any activity on relevant pins (e.g. USART_RX pins), and if it detects such activity, it activates hardware of the given interface and starts to await the prescribed initialization sequence for the given interface.

This means, that if there is any external signal activity on *any* of the "listening" pins (or these pins are floating and there is enough "noise"/interference so that the bootloader "sees" activity on them), the bootloader will stop listening on the one particular interface the user intends to use. In other words, unless the user makes sure that *all* bootloader-related pins are at a steady level during bootloader entry, the chosen bootloader will enter unreliably or not at all.

For list of bootloader interfaces and related pins for your particular STM32, resort again to AN2606.

Created: 7.January 2023

139. BOOT0 pin pulled up, but bootloader does not start (in 'F1/'L1/'F2/'F4 you need to pull down BOOT1, too)

Various STM32 (sub)families utilize different methods to enter the built-in bootloader upon reset. The traditional method employed in the primeval STM32F1xx is to have a defined level on **two** pins: BOOT0 at logic 1 and BOOT1 at logic 0. Reason for using two pins instead of one is, that there are three kinds of memory which can be mapped at address 0 from where the Cortex-Mx processor actually boots: the user FLASH, the "system memory" (piece of FLASH programmed at factory and not user-reprogrammable), and RAM.

The gotcha lies in the fact, that while BOOT0 is a dedicated pin, **BOOT1 is a GPIO (PB2)**. If user is unaware of the requirement of BOOT1 being at a certain level, results may be surprising, ranging from incongruent "randomly" unreliable bootloader entry, through a consistent but mysterious pattern e.g. "works at first powerup but not with pushbutton reset" all the way to entirely "unoperational" bootloader.

In particular, if there is enough parasitic capacitance on PB2/BOOT1 to "hold" its state reliably for some time, then when being powered up after a longer off period the pin is at logic 0 allowing relatively reliable bootloader entry. However, if it has been reset for whatever reason (soft-reset deliberately invoked to enter bootloader, pushbutton reset, rapid powerdown/up) while the pin was held at logic 1 being set as GPIO output, the parasitic capacitance will hold the level at logic 1 long enough to prevent bootloader entry. This is a very confusing behaviour.

ST used this original scheme (called Pattern 1 in AN2606) also in subsequent STM32 families, up until 'F4. In the newer 'F4 models ST started to combine the pins with option bits, and since the 'F0/'F3 era, BOOT1 is not used anymore, being entirely replaced by option bits.

Created: 14.January 2023

140. Not all TIM2/TIM5 are created equal (in some STM32 they are not 32-bit)

All STM32 models (but the smallest 'L0) have a selection of at least half a dozen of timers. They vary in capabilities, but ST tends to maintain the same name (number) for similar capability timer across all families.

For example, in most STM32 models, TIM2 and TIM5 are 32-bit timers - this is in contrast to the fact that most timers are 16-bit. However, there are exceptions to this unwritten "rule" - and when migrating between families, these may come as an unpleasant surprise.

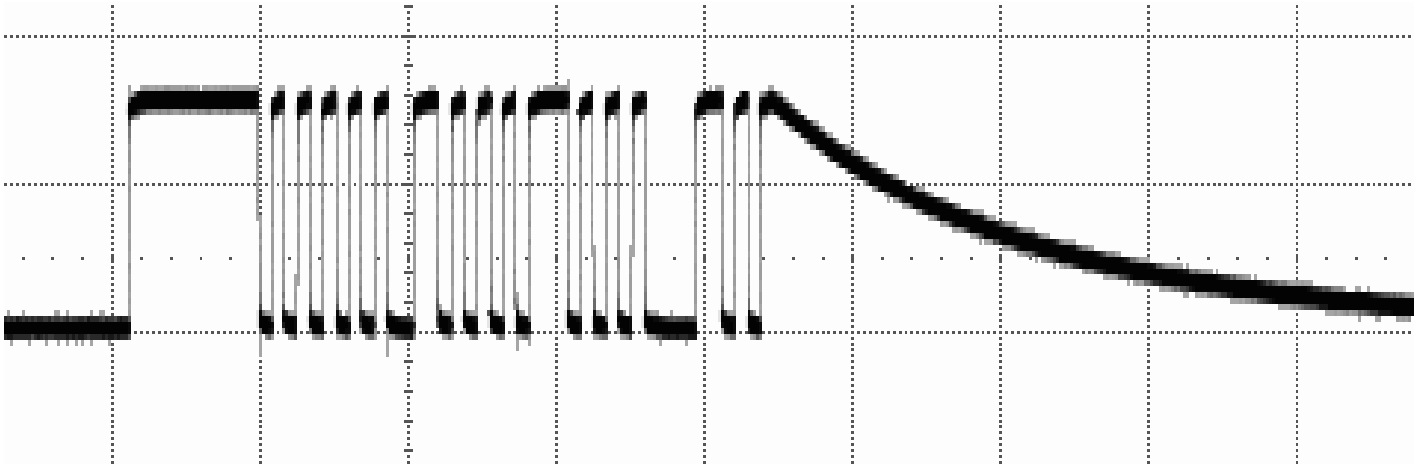
In this particular case, in the 'L0 and 'F1 families, TIM2 and TIM5 are 16-bit too, and there are no 32-bit timers in 'L0 and 'F1. In 'L1, TIM2 is 16-bit, and TIM5 is 32-bit, but TIM5 is not available in all 'L1 models.

A great overview of timers' capabilities, and their availability in all STM32 families, is in AN4013.

Created: 21.January 2023

141. On UART IDLE frame insertion upon UART_CR1.TE being set

The STM32 UART has a separate control bit, UART_CR1.TE, to enable the transmitter portion of the UART. The most visible effect of the TE bit is, that it enables/disables output on the UART_Tx pin assigned to this UART through the GPIO matrix. This is visible on the following waveform, where immediately after TE was set, 4 bytes were transmitted (0x55, 0x56, 0x57, 0x58) and then TE was cleared again:

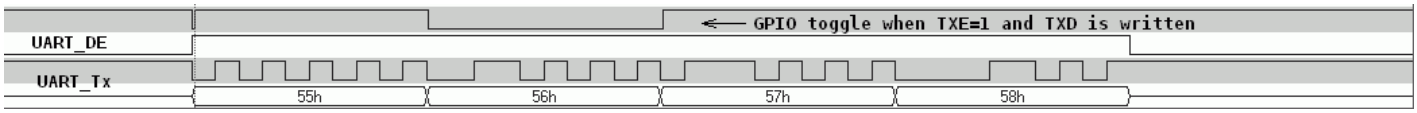


The gradual "decay" is consequence of leakage on UART_Tx pin together with parasitic capacitances on it. But there is one more notable feature on that waveform: after TE is enabled, Tx pin goes to high, which is the expected idle level, but even if the first byte 0x55 was written to UART's data register UART_TXD immediately after TE being set, there is a one-byte long delay before Tx starts to transmit this byte.

This is a deliberate and documented feature of the STM32 UART and can be used to insert an IDLE frame for example in multi-drop buses such as RS485. IDLE frames allow for end-of-frame detection, and/or to resynchronize UART receivers when a listener is brought up amidst an ongoing transaction.

The following waveform displays the same transaction as above, taken by a logic analyzer. The top waveform is from a GPIO output which was toggled each time UART_ISR.TXE was tested as being set and subsequently USART_TDR was written; the second from top is taken from UART_DE pin i.e. it is a hardware indication of transmission in progress[1]:

And this is how the same transaction looks like, when TE is never being cleared after it had been set upon UART initialization, far before this transaction. Note, that upon start of transmission the top waveform indicates that there are two writes to USART_TDR immediately after each other: the first byte after being written to the UART Tx buffer, gets immediately transferred to the UART Tx shift register as Tx starts immediately, so the TXE flag indicates that the Tx buffer is empty and the second byte is allowed to be written just after that:



It may appear, that in order to insert the IDLE frame, it is enough to toggle TE before packet transmission start, to insert an IDLE frame. However, in the newer version of UARTs which are used since 'F3/'F0, the following warning appeared in the description of TE bit in RM:

In order to generate an idle character, the TE must not be immediately written to 1. In order to ensure the required duration, the software can poll the TEACK bit in the USART_ISR register.

This is quite easy, the whole sequence may look like this:

```
USART2->CR1 &= ~USART_CR1_TE;
while((USART2->ISR & USART_ISR_TEACK) != 0);
USART2->CR1 |= USART_CR1_TE;
```

But why is this poll important, why was this requirement not present in older STM32, and what delay does it actually involve?

One notable difference between the older and newer version of UART is, that while the older UART was

entirely clocked from the related APB clock, in newer UARTs only the register interface is clocked by the APB clock, UART itself (the "kernel") is clocked from one of several possible clock sources (including APB, too) selected in RCC, usually by a field in RCC_CCIPR register.

While writing TE bit happens on the register i.e. APB-clock side, the actual insertion of IDLE frame is part of the UART logic i.e. it's clocked by the kernel clock, which may be asynchronous to APB and may be also slower than the APB clock. Thus, after writing TE bit, it takes time while the synchronizer between the two clock domains writes this bit into the UART logic on kernel-clock side, and the TEACK bit comes through another synchronizer back to the APB-clock domain, as a confirmation that TE bit has indeed been registered by the kernel-side logic.

Here is a simple program testing this behaviour, written for the STM32L4 Disco board. There are several options to clock the UART, enabled by uncommenting one of the USART2_USE_DIVIDED_APB, USART2_USE_HSI16, USART2_USE_LSE defines, or leaving all of them commented for the default, plain APB clock. The system clock is the default 4MHz MSI, so the HSI16 option tests asynchronous USART kernel clock which is faster than APB clock, and LSE tests USART kernel clock which is significantly slower than APB clock (namely 32.768kHz).

Only with USART2_USE_LSE the plain TE toggle failed, leading to waveform identical to that without TE being touched. That's where the TEACK test as outlined above restored the IDLE frame. Another simple experiment, where a GPIO pin was toggled before and after the TEACK test, indicated a delay between cca 34us and 64us, which is one-to-two periods of the slow LSE (32.768kHz) clock, plus a couple of APB clocks, spent in the synchronizers; and a few more system clocks to actually toggle the GPIO pin.

However, TE toggling is used here only to illustrate the potential problem and its nature and source. In real-world programs, e.g. with RS485, the most appropriate approach is to switch off TE after transmission ends (see commented-out line in USART2_IRQHandler() in the example), and switch it back on just before next transmission is to commence. This leaves ample time while TE is off, during reception. That UART_Tx pin goes threestate after TE being switched off is in this case not a problem, as the RS485 transceiver is not set to drive the bus during that period either.

---

1. Unfortunately, as pointed out also by user Gael Le Moing, DE is not active during the initial IDLE. This leaves it only to the usual RS485 "failsafe" (usually a pullup and a pulldown) to ensure that the bus stays at 1 during IDLE.

Created: 28.January 2023
Footnote 1 added: 2.May 2023

142. Not all TIMx_RCR are created equal: some are 8-bit, some 16-bit

One of the central features of STM32 timers is the Update Event. This is basically the moment, when the timer's counter overflows. This event is then used to update all preloaded registers, resulting in a well-defined synchronous operation of all channels and other features of the timer.

However, in some "better" timers, the Update Event does not necessarily happen at every overflow of the counter. These have an additional register, Repetition Count Register - TIMx_RCR, and if this is set to nonzero, the Update Event occurs at every (TIMx_RCR + 1)-th overflow of the counter.

TIMx_RCR in itself is again a "holding" register, so its value is always what was written to it by the user (or the reset value which is 0). It is loaded into the actual repetition downcounter (which is internal and not user accessible) upon each Update Event, making it effectively unconditionally preloaded.

TIMx_RCR is present only in those timers which have channels with complementary output - TIM1, TIM8, TIM20 called collectively Advanced Timers, and TIM15, TIM16, TIM17. Of course, not all of these timers are present in all STM32 families/models, AN4013 gives a good overview of the available options.

As most STM32 timers (and their counting-related registers) are 16-bit, it may come as a surprise to learn that **in many STM32, TIMx_RCR is only 8-bit wide**.

In the 'F3, 'F7, 'G0, 'G4, 'H7, 'L4 and 'L5 families, TIM1, TIM8 and TIM20 (where available) do have 16-bit wide TIMx_RCR. However, even in those families, TIMx_RCR in TIM15, TIM16, TIM17 is still only 8-bit wide.

Created: 4.February 2023

143. V_{REFINT} and temperature calibration values not available in 'F1 and 'F2

The internal voltage reference V_{REFINT} in STM32 provides a stable voltage, using which the absolute voltage of analog reference VREF+ (bonded to VDDA or even VDD in low-pin-count packages) can be determined. While V_{REFINT} is quite stable across VDD and temperature variations, its absolute voltage has a certain spread across individual devices, given manufacturing tolerances. For example, the STM32L476 datasheet gives *Internal reference voltage spread over the temperature range* as max. 7.5mV, and from *Average voltage coefficient* of max. 1200ppm the the maximum spread over operating voltage range is around 2.5mV; however, the nominally 1.212V reference has range from min. 1.182V to 1.232V, i.e. 50mV. Other models/families may have different values, but the principle is the same and the ratios are very similar.

To compensate for this, for most STM32, the precise value of V_{REFINT} is determined at manufacturing, in the form of ADC readout at given "standard" supply voltage and temperature; and this readout is stored at a predetermined address (usually called VREFINT_CAL and given in individual datasheets) in the system memory. This value can then be used to determine VREF+ voltage and thus all ADC readouts more precisely.

Similarly, the internal temperature sensor has a manufacturing spread, which is compensated by using two readouts - TS_CAL1 and TS_CAL2 - taken by manufacturer at "standard voltage" and two temperatures, again addresses and particular temperatures given in individual datasheets.

However, two of the oldest STM32 families - the **'F1 and 'F2 family - don't have these calibration values determined.** Quite surprisingly, the 'L1 family does have them, even if it is an older family than the 'F2. For 'F2, this is another difference against 'F4 to be taken into account, should 'F2 be used as a "downgrade" for 'F4.

Even in families except 'F1 and 'F2, the lowest-end "value-line" models (principial numbering of which ends with 0) usually don't have all of these coefficients available, often the higher-temperature coefficient is missing.

Created: 11.February 2023

144. Timer trigger interrupt/DMA does not work until TIMx_SMCR.SMS is set to nonzero

Timers in STM32 have several quite poorly described features. One of them is the TGI signal, which can only be seen on the timer block diagram at the beginning of timer chapter in RM. This signal stems from the TRGI input, and triggers interrupt and/or DMA transfer, pending TIMx_DIER.TIE/TDE being set, respectively.

Source of TRGI is selected by TIMx_SMCR.TS bitfield. It can be one of four internal sources (or more in newer STM32 families), which are connected to the TRGO outputs from other timers, thus forming inter-timer interconnection. TRGI can come also from external sources - TIMx_CH1/CH2 or TIMx_ETR pins.

Primary purpose of TRGI is not to trigger interrupt/DMA, but to control the counter's clock logic (called slave-mode controller). The exact behaviour of this is selected by TIMx_SMCR.SMS bitfield, causing TRGI e.g. to reset the counter, one-time enable clock, or gate the clock.

It may appear that once TRGI source is set, selection of TRGI's action in slave-mode controller should not interfere with the TRGI**I**triggered interrupt or DMA. However, from experimentation it appears, that for the TGI interrupt/DMA to work, TIMx_SMCR.SMS should be set to nonzero value.

Created: 18.February 2023

145. Not all TIM input polarity selection created equal (in non-XL 'F1 there's no both-edges selection)

In the STM32 timers, channels selected for Input Capture perform the capture upon edge of the input signal. In the vast majority of STM32 TIM, rising, falling or both-edges can be selected using CCxP and CCxNP bits in TIMx_CCER.

However, in the historically first STM32 family, the **'F1, timers don't have the both-edges option.** Only TIMx_CCER.CCxP bit is used to select one of the rising/falling edge and there is no TIMx_CCER.CCxNP bit.

The so-called XL-density STM32F101xx and STM32F103xx microcontrollers with 768kB and 1MB FLASH (i.e. with xF or xG suffix) are a relative late coming addition to the 'F1 family. This is why in these, TIM9..TIM14, which were also an addition only in this sub-family and are not present in other 'F1, already do have the both-edge selection as other STM32 TIM do.

Created: 25.February 2023

146. Only one EXTI per pin number, this means routing (which is different in 'F1 and 'G0/'C0/'L5/'U5)

Interrupt upon edge (or level) of input signal on an external pin is a feature traditionally present in perhaps all microcontrollers. Usually, the number of pins, which can be used as external interrupt, is limited and much less than all pins present on the given package.

In all STM32, there are **16 edge-triggered external pin interrupts**, which are handled by the EXTI unit[1]. Interrupt upon rising, falling, or both edges can be selected individually, and the **edge detection is asynchronous**, i.e. independent on the system clock. The EXTI functionality on a selected pin is available in parallel to any other functionality of given pin, except if it's set as Analog.

External pin interrupts EXTI0..EXTI15 in STM32 are **not assigned to fixed pins** - there is a certain, although not unlimited, flexibility available: each EXTIn interrupt can be assigned to any of the pin with the same pin number within the port. For example, EXTI3 can be assigned to PA3, or PB3, or PC3 etc.

This means, that there must be some mechanism to select particular pin for given EXTIn interrupt. In most STM32, this selection is through the SYSCFG unit, in bitfields of **SYSCFG_EXTICRx** registers. However, there are exceptions:

- in the ancient **'F1** family, there's no SYSCFG and for EXTI selection **AFIO_EXTICRx** registers were used;
- in the **'G0/'C0, 'L5 and 'U5** families, selection has been (quite logically) moved into EXTI itself, using **EXTI_EXTICRx** registers

---

1.In addition to pin interrupts, EXTI unit handles several internal interrupt sources (mostly wakeup-related), too. Pin interrupts are numbered EXTI0 to EXTI15. Interrupts EXTI16 and higher are from internal sources.

Created: 4.March 2023

147. Slow input signal slew rate may cause multiple interrupts (especially with EXTI, which is asynchronous)

On STM32 pins, digital input circuitry conditions the input signal using a Schmitt circuit, effectively introducing hysteresis of cca 200mV to the input decision level. This means, that a slowly rising or falling signal should not cause several input level transitions, as long as the rise or fall is monotonic.

However, as soon as there is any noise or interference with amplitude of more than cca 200mV superimposed on this slowly rising/falling signal, the input may see several transitions between 0 and 1 in rapid succession.

Hundreds of millivolts of noise are not a rare sight in digital circuits. As long as all signals in the circuit are "perfectly digital", i.e. they are driven between GND and VDD by relatively low impedance drivers, resulting in sharp transitions/edges, such noisy circuit still may function quite well as expected, and the noise may go unnoticed.

Slowly rising/falling signals on the other hand tend to result from being driven by a high-impedance source, making them even more prone to interference through capacitive or inductive coupling from neighbouring circuits/tracks. Additionally, improperly routed and/or decoupled power supply and ground may introduce additional noise to signal through voltage drops on common tracks impedance, resulting from the pulsed power consumption of massive digital circuitry such as the mcu itself.

The surprise is often associated also with the less-known fact, that - in contrast to interrupts associated e.g. with TIM input capture - the EXTI interrupts are asynchronous, i.e. they react on pulses shorter than the maximum system clock frequency of given STM32 model, regardless of the actually used system clock frequency. If EXTI is set to react on both edges, such pulse will the cause both interrupt source to latch, thus resulting in two (or more, if there's more such noise) consecutive ISR execution.

While using for example TIM channels (which feature also an optional digital filter) instead of EXTI may reduce impact of such unwanted noise, it's better to design the circuit so that overall circuit noise is reduced, and slowly rising/falling and/or weakly driven signals are avoided.

Created: 11.March 2023

STM32 gotchas

148.Touchscreen flipped or does not work on 32F429IDISCOVERY

STM32F429 was the first STM32 to feature the LCD controller (LTDC) and SDRAM support (in FMC). The associated DISCO board - 32F429IDISCOVERY - showcases both these features, incorporating a 64-Mbit SDRAM and a 320x240 LCD display with resistive touchscreen.

This board proved to be quite popular and in almost 10 years it has seen several updates. These are marked on the board itself MB1075B through MB1075E (as of Jan 2023). UM1670 contains a list of changes in *Board revision history* chapter.

One of the changes involved swapping the YU/YD pins of the touchscreen. As a consequence, some versions of the board have swapped direction of the touchscreen as compared to others. This has been discussed here and here, those threads also contain links to firmware which can be used to test this phenomenon.

As the original touchscreen controller, STMPE811QTR, is not manufactured anymore, the boards are prepared to use a different controller, SX8651. This results in firmware which was written for the original controller, ceasing to work on boards with the newer controller.

[EDIT] As reported here, in the examples (BSP) in CubeF4, ST has addressed the issue of touchscreen flip. [/EDIT]

Created: 18.March 2023
Notice of flip addressed in examples: 1.May 2023

149. Incorrect ADC readout - due to missing calibration

ADC is perhaps the most varying peripheral from STM32 family to STM32 family, so migration usually requires to compare ADC chapters in RM and modify software as appropriate.

In some STM32 (namely 'L1/'F2/'F4/'F7), ADC are relatively simple as far as single conversions go: after enabling ADC clock in RCC and selecting appropriate clock prescaler, selecting one channel in input multiplexer and setting its sampling time, a single conversion is started by writing appropriate register bit. After the expected time ADC signals conversion complete, and the appropriate result can be read out from the data register, provided that correct voltages are present at VREF+/VDDA pins. That's all. Of course, these ADCs are much more complex and have a various degree of autonomous operation on multiple channels, mutually synchronized multiple ADCs, etc., but the basic single conversion has no special prerequisites.

However, in all other STM32 families, the ADCs require to perform an extra step - **calibration** - in software, **before the first conversion is started**. This step **removes offset** present in the ADC due to manufacturing variations and imperfections. Calibration may be required also when temperatures and/or supply voltages change significantly.

The ADC chapter in relevant RMs always contains a subchapter dedicated to ADC calibration, which describes the required procedure in detail.

Users, who forget to perform ADC calibration before the first conversion, are usually surprised by the readouts being offset quite significantly, usually a few tens of bins.

Created: 25.March 2023

150.Incorrect ADC readout in some families, due to the "wrong result after 1ms delay" erratum

For some STM32 Errata (seen in 'L4, 'L5 and 'G4), the following applies for ADC:

Wrong ADC result if conversion done late after calibration or previous conversion
The result of an ADC conversion done more than 1 ms later than the previous ADC conversion or ADC calibration might be incorrect.

This may result in surprisingly incorrect ADC readouts to the unaware. The suggested workaround is, if there's more than 1ms delay, to perform a dummy conversion and discard its result.

In this thread on STM32 forum, ST employee SimonV sheds more light on the minutiae of the underlying problem. The main takeaways are, that

- the problem causes more-than-specified additional offset/gain error in a small portion of chips due to silicon process variations during manufacturing;
- the problem is independent of power supply voltage and measured voltage;
- for the workaround, it is enough to perform a dummy conversion at any channel with any sampling time.

Created: 1.April 2023
Remarks on details and link to forum thread added: 4.November 2023

151.Missing ADC codes in 'H7, due to missing calibration

In most newer STM32 ADC, before performing the first conversion, a calibration has to be performed by software. This is in order to remove an offset due to manufacturing variations and imperfections.

In the 'H7 family, a new ADC has been implemented, featuring 16-bit resolution in contrast to the 12-bit ADC in other STM32 families. The increased resolution obviously required a change in the internal ADC architecture. This is witnessed by the fact, that if this ADC is used without calibration, it not only produces offset values, but with input voltage varying across the whole ADC input range, values corresponding to up to a hundred of ADC bins around 2^12 boundaries are missing.

This issue has been spotted and reported by user HPiG.

Created: 8.April 2023

STM32 gotchas

152.Internal temperature sensor returned unexpected value (troubleshooting guide)

ADC in STM32 features a temperature sensor on one of its inputs. This is a circuit, which generates voltage linearly proportional to the internal temperature of the chip. Users often test this as this is a relatively convenient test to try the ADC; however, sometimes the results are confusing.

Here are some guidelines how to achieve correct result from temperature sensor:

- Make sure you have correctly connected VSSA/VREF- and VDDA/VREF+ to stable voltage sources within datasheet values.
- Make sure the ADC has enabled clock in RCC, and its frequency after prescaling is within the datasheet limits.
- In STM32 families where applicable, make sure you have performed ADC calibration before taking measurement.
- To make sure the ADC performs conversions properly, you can enable the internal voltage reference's channel and take a measurement on it, comparing the result to the expected value in datasheet, and/or the factory calibration value VREFINT_CAL stored in the system memory.
- Note, that in STM32 models with multiple ADC units, temperature reference is available usually only in one of them.
- Temperature sensor has to be enabled, and the exact method depends on the STM32 family. In most families, this is accomplished by setting a register bit in ADC itself, often ADC_CCR.TSEN. For added confusion, in the 'L4 and 'L5 families, the same bit is called ADCx_CCR.CH17SEL, and simultaneously to enabling the temperature sensor it also flips a switch at given ADC MUX input (from internal DAC connection). In some families, temperature sensor has a common enable bit with the internal VREFINT reference, ADC_CR2.TSVREFE. Uniquely in the 'L0 family, besides ADC_CCR.TSEN, it has to be enabled also in SYSCFG_CFGR3.ENBUF_SENSOR_ADC; and this fact is not mentioned in the Temperature sensor subchapter of ADC chapter.
- After enabling the temperature sensor, a startup time specified in the Datasheet has to be observed.
- Select the proper input in the ADC MUX. In some 'F4 and 'F7, the ADC input used for temperature measurement is shared with VBAT measurement; in that case, make sure VBAT measurement is not enabled.
- Make sure appropriate sampling time is set for given channel, according to minimum sampling time for Temperature sensor, given in Datasheet.
- Calculating converted value to temperature may be confusing, here are some first-principle guidelines. If you use library functions/macros, make sure you understand what are the individual parameters, and supply them in the expected physical units. With macros, note that improperly used types may lead to incorrect results e.g. when mixing unsigned and signed integer variables, and/or surprisingly lengthy calculations and/or unexpected FLASH and RAM consumptiond due to imported libraries, e.g. when *double* are used inadvertently.
- Note, that the TS_CAL1/TS_CAL2 calibration values are taken at different temperatures and different reference voltages for different STM32 families. Refer to the Datasheet for the details. Be cautious, sometimes there are errors in the documentation.
- In some STM32 models (in older families, and in the "Value Line" models), one or both TS_CAL1/TS_CAL2 calibration values are unavailable. In that case, you have to resort to using the Average slope and Voltage at X°C parameters for the Temperature sensor in Datasheet. Note, that these values have a relatively high variation and errors up to several tens of °C may be expected in this case.
- Note, that the temperature sensor is not a precision instrument. Even with using the calibration values, expect +-5°C of deviation. ST recommends to use it as a monitor of relative temperature changes, rather than as an absolute thermometer.
- ADC measurements are subject to noise from internal and external sources, and given the relatively high slope of the ADC characteristics (cca 2 bins/°C at VREF=3V), it has noticeable impact on the calculated temperature. Consider filtering/smoothing of measured values.
- Note, that the temperature sensor measures the internal temperature of the chip, not the ambient temperature. It may be surprisingly quite a couple of degrees higher than ambient, depending on the current consumption and degree of cooling applied to the chip.

Created: 15.April 2023

153. Internal temperature sensor, VREFINT and VBAT can't be part of a fast ADC sequence - in families, where ADC has a common sampling-length register

ADC is a module which varies between STM32 families quite a lot. It means, that nontrivial operations which may be relatively easy to accomplish on one STM32 family may be hard or impossible on other.

All ADCs have a couple of internal channels - internal temperature sensor, voltage reference VREFINT; and where available, measurement of voltage on the VBAT pin and/or some of the VLCD pins. These sources are of relatively high impedance and for proper operation they require a relatively long sampling time.

External ADC signal sources are usually of relatively low impedance and users often require high sampling rates, implying short sampling times.

The STM32 ADC enables to perform a sequence of conversions on various channels automatically. In most STM32 families, each channel in sequence can have assigned a sampling time individually. However, this is not the case of of the **'F0 and 'L0** family, where there is only a single register for the sampling time, and **all channels in a sequence use the same sampling time**. This prevents converting "slow" channels such as the internal signal sources, and "fast" external signal sources, in a single sequence.

The 'G0 and 'C0 families solve this particular issue in yet another fashion: there are two registers for sampling times available, and a single bit for each channel, selecting one of the two sampling times. Thus, mixing "slow" and "fast" channels in a sequence is possible in these families, to a certain degree.

Created: 22.April 2023

154. In 'L07x, some TIM3_CHx pins don't work - unless remapped in TIM3_OR

In some STM32, some of the timer input channels can be switched from the usual GPIO connection (through the AF matrix) into an alternative signal source, usually an internal clock or other singal. This enables to conveniently measure mtual relationship between clocks, or for example to count number of occurences of some event. Such feature is usually controlled by so called remap bits in TIMx_OR (Option Register), their default being the usual GPIO connection.

In STM32L07x, there are some surprising default mappings of some of pins related to TIM3 channels (in the 'L0xx family, only 'L07x has TIM3):

- TIM3_CH1 is by default connected to USB_SOF, and not to PA6/PC6/PB4/PE3
- pin PB5 is by default connected to TIM22_CH2 rather than TIM3_CH2
- pin PC9 is by default connected to USB_NOE rather than TIM3_CH4

To be able to use TIM3_CHx on these pins, the respective remap bits in TIM3_OR have to be set.

Issue spotted by user JZhan.33.

Created: 29.April 2023

155. After programming/reset/power cycle, program does not work - and debugging ends up at 0x1FFFxxxx addresses

The common denominator of these problems is the built-in bootloader; however, there are several mechanisms involved.

The built-in bootloader is located to the so called system memory - an area of FLASH at 0x1FFFxxxx addresses, which is programmed at factory and locked, so that it can't be reprogrammed. When the entry mechanism is activated at mcu reset, this memory is mapped (aliased) by hardware at 0x0000'0000, from where the processor boots. The bootloader entry mechanisms for individual STM32 (sub)families differ, and they (together with other details) are described in AN2606.

If the user program after programming it into FLASH and subsequent reset, does not work, it may be that:

- the BOOT0 pin is left floating - this is a high-impedance input, which may inadvertently get pulled up e.g. through resistance of soldering residui.
- in some newer STM32 models, BOOT0 pin is shared with a GPIO. In this case, the pin is sampled at reset, and if samples as 1, it enters bootloader; else the user program can use it as GPIO. If this feature is not disabled in Option bytes, user has to make sure the pin reads as 0 at reset (e.g. using a suitable pulldown). If this pin is used as input, the connected device must not pull it down during reset. Note, that even in case this pin is used as output, parasitic capacitances may "hold" an unwanted high level long enough to be sampled at the end of a potentially short reset pulse (e.g. internal reset sources such as watchdog or software reset result in a min. 20us reset pulse).
- in some STM32, if previous power-on reset was accomplished while FLASH was empty, the PEMPTY bit keeps entering bootloader upon subsequent resets until a power cycle or executing option-byte reload (OBL_LAUNCH).
- in some STM32 models with dual-bank FLASH (e.g. 'G4 cat.3, or the high-end L4 and L4+), if booting from the second bank is enabled in Option bytes (by setting BLB2 bit), the System memory is mapped at 0x0000'0000 even if program executes from user FLASH after reset. If in that case user leaves the SCB_VTOR register at its reset (zeroed) state, interrupts will be executed according to the vector table in system memory i.e. from the bootloader, often resulting in crash - which ends up in HardFault handler again in the system memory, i.e. at a 0x1FFFxxxx address.

Common remedy to BOOT0-pin-related problems is to set the Option bytes so, that the hardware pin is not sampled at startup, effectively disabling the bootloader entirely.

In cases where the system memory (bootloader area) remains mapped at 0x0000'0000, before enabling any interrupt, the SCB_VTOR register has to be set to 0x0800'0000 to map properly the vector table to user FLASH. Alternatively, (e.g. in 'F0 based on Cortex-M0 which does not have SCB_VTOR), use the remapping mechanism (usually a MEM_MODE field in some of the SYSCFG registers) to map user FLASH to 0x0000'0000.

Created: 6.May 2023

156. USB and PG2..PG15 don't work on some STM32 - they have a dedicated power pin and an isolation switch

In the 'L4, 'L5 and 'U5 families, USB module has its own dedicated power pin, V_{DDUSB}. In the larger packages with PG2..PG15 pins, they also have a dedicated power pin, V_{DDIO2}.

Both V_{DDUSB} and V_{DDIO2} are isolated from the rest of the system, both electrically and logically, by dedicated switches. These switches are controlled by USV and IOSV bits in PWR_CR2 register (or, in case of 'U5, USV and IO2SV bits in PWR_SVMCR register) and are off by default. As a surprising consequence, USB and PG2..PG15 don't work, until these bits are set.

Note, that dedicated V_{DDUSB} and V_{DDIO2} pins exist also in some other STM32 models, but there they don't have the isolation switch. The isolation switch is probably needed for the low-power STM32 families due to technological reasons, to prevent unwanted leakages with vastly different voltages on VDD and the dedicated power pins.

Created: 13.May 2023

157. In 'U5, ADC calibration hangs, if VDDA isolation is not removed

In the low-power STM32, there are several auxiliary power domains,which are isolated from the common VDD domain both logically and electrically through dedicated isolation switches, presumably to reduce leakage when those domains are not in use.

The 'U5 family took this to the next level: even the VDDA power domain has an isolation switch, which has to be switched on, whenever any analog facility is to be used. It is controlled by the PWR_SVMCR.ASV bit.

If this bit is not on i.e. the switch is off, as it is by default, the analog facilities don't work properly. For example when trying to use the ADC, the first step is to perform a calibration. As user ASimo.3 found out, if the isolation switch is off, this calibration will never finish, and naively written loop waiting for calibration to end will hang forever.

Created: 20.May 2023

158. STM32F407 runs at cca 20x lower frequency than set (a RCC/HSE/PLL puzzle)

In a device with STM32F407, system clock is derived from a 25MHz crystal connected to HSE. This in PLL is divided to 1MHz, multiplied by 336 and then post-divided by 2 to provide the 168MHz system clock. This is then subdivided and distributed throughout the chip as usually; among other modules also to timers driving several LEDs with PWM at 260Hz.

In one specimen of the device, a couple of minutes after having passed an initial test and been programmed by the final application, all LEDs started flashing violently and synchronously at a frequency of cca 13Hz. The whole functionality of the device was off, indicating, that it run at a significantly lower system frequency. As the PWM should have been 260Hz and LEDs were blinking at around 13Hz, system frequency at that point was probably around 8MHz instead of 168MHz.

Why? Solution below.

Investigation revealed, that in given specimen, the 25MHz crystal was mechanically damaged (this probably had happened when board had been populated at a contractor), and as the device has been moved around the test positions, it ceased to oscillate.

The surprising thing in this was, that even if there was no HSE oscillation, the system kept running at a surprisingly low frequency (would it have switched to HSI, it would run at around 16MHz, but the observed system frequency was significantly lower). This was probably consequence of using PLL, where output of the phase detector probably went to one of its extremes and the VCO responded by going at its lowest frequency. It is rather surprising that the VCO which nominally operates between 100MHz and 432MHz, can oscillate (albeit in unspecified mode) at as low as 16MHz (PLL had a post-divider of 2).

The STM32F407 does have CSS (Clock Security System) which detects HSE oscillation and in case of failure it switches to HSI and throws an unmaskable interrupt (NMI); however, operation of this device is not critical and it also does not have any reasonable means of backup operation, so the solution in this case was simply exchanging the faulty crystal for a new one.

Created: 27.May 2023

159. PLLI2S/PLLSAI outputs unexpected frequency (due to prescaler shared amongst PLLs)

Some STM32 families - e.g. 'F4/'F7/'L4 but also others - have several PLL units in their RCC module. One of the PLLs - "main" - is dedicated for system clock generation, and the others allow to generate relatively independent clocks, mostly to the audio-related modules such as I2S and SAI.

Details of interconnections of inputs and outputs of these PLLs to other signals within RCC differ significantly from family to family - and even between models within a family - but in many models there is only one path for the input reference signal, shared amongst all PLLs. It means, that there is only one multiplexer which selects usually between HSI and HSE as primary clock source, and one single prescaler, controlled by the PLLM field of the "main" PLL's control register (usually RCC_PLLCFGR).

As soon as any of the PLLs is enabled, the PLLM field is locked and cannot be written anymore. This means, that software shall set up all PLLs at once, or otherwise set them up with this limitation in mind.

However, functions for PLL setup in some libraries - e.g. the Cube/LL - expect, that they are able to write all fields determining the PLL's output frequency: the prescaler M, VCO divider ("multiplier") N, and post-scalers (STM32 PLLs have up to 3 outputs from a single VCO, so there are up to 3 post-scaler fields, P, Q and R). The usual setup is, that the "main" PLL is enabled the first, so calling its setup function sets the PLLM value and locks it. If, subsequently a function attempts to set one of the "auxilliary" PLLs with a different M setting, the new M value is quietly ignored, and as a result that PLL then outputs a surprising and unexpected frequency.

Created: 3.June 2023

160. LSI in 'U5 does not work - unless Backup Domain access is enabled before enabling LSI

In most STM32, the low-frequency (and low-precision) internal RC oscillator LSI is in the VDD power domain. As a consequence, if it is e.g. used as RTC clock, RTC stops working when VDD is off, even if RTC itself is powered from VBAT.

However, in the 'U5 family, LSI has been moved to the VBAT (Backup) power domain. This again has a consequence - LSI cannot be enabled until Backup domain access is enabled by setting the PWR_DBPR.DBP bit.

This gotcha has been spotted by user Linas L.

Created: 10.June 2023

161. No connectivity from ETH and OTG-HS to FLASH in 'F407 and 'F446 (so ETH/USB won't transmit const data)

The ETH and OTH_HS modules incorporate a DMA unit, which after having been set up, transfer data autonomously from/to the interface to/from memory. For this, they have a master connection to the AHB bus matrix, where the memories are connected as slaves.

In 'F407 (and 'F417; also in 'F405/'F415 and in 'F446 which do have OTG-HS but don't have ETH) bus matrix, these modules don't have connectivity to every memory in the system - namely they cannot access the FLASH.

The practical consequence is, that ETH and OTH_HS (in DMA mode) in these STM32 models cannot transmit data from FLASH. As compilers usually place data qualified as *const* into FLASH, those data cannot be transmitted. If the software/drivers for these modules are not written to meticulously check all errors, upon attempting to transmit such data they simply cease to work. In case of ETH, this for example may mean that a device based on STM32F407 ceases to respond to *ping*s, if it attempts to serve a web-page from internal FLASH (as noticed by user DMart, with explanation provided by Mahdy Sghaier (ST Employee)).

In 'F427/'F429 - successors to 'F407 - this limitation is not present.

Created: 17.June 2023

STM32 gotchas

162.LPUART Rx clocked from LSE corrupts bytes at 9600 Baud - unless LPUART_CR3.UCESM is set

The newer type of UART present in STM32 since 'F0/'F3 on, is a peripheral with quite couple of options when it is configured at chip design stage: it can have the synchronous option, the IrDA/smartcard options, RS485 support etc. Various UARTs in various STM32 families are configured by a mix of these options, as described usually in USART chapter in DS, and/or at the beginning of UART chapter in RM.

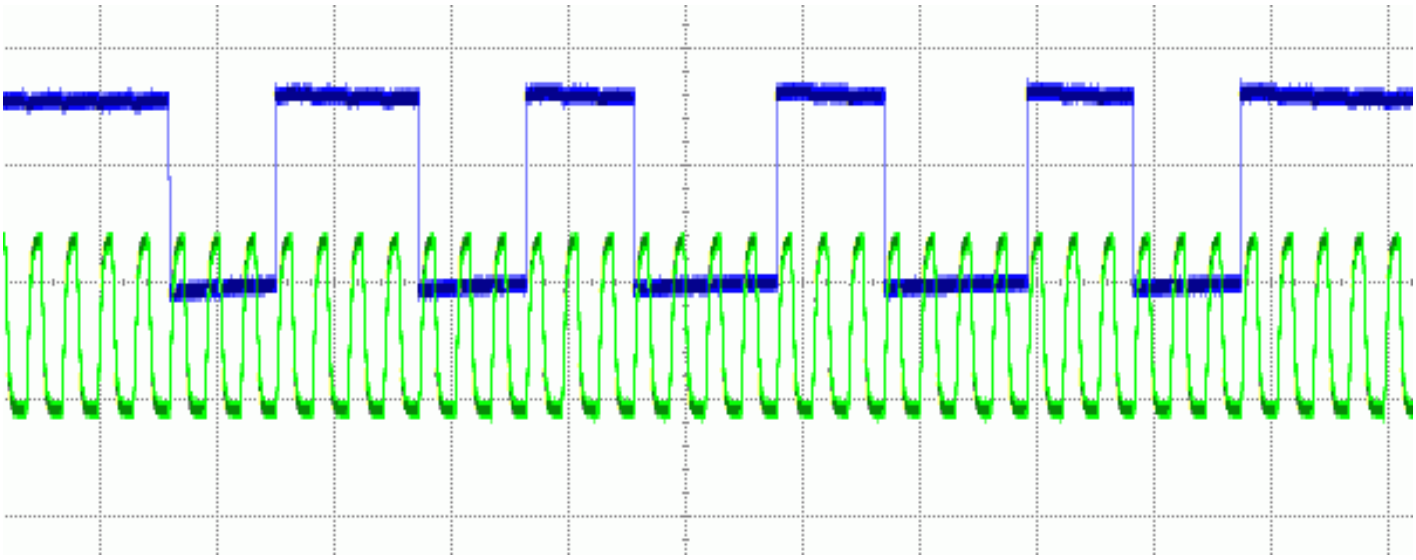
For purposes of this discussion, most notable is the option to have two separate clock domains; the APB-clock domain through which its registers are accessed from the rest of the mcu; and the kernel-clock (in some materials called independent-clock) domain, which contains the actual transmitter/receiver and its surrounding logic. Kernel-clock source is selected in RCC, usually amongst APB/HSI/LSE/LSI; again, the particular details depend on family.

Nonetheless, dual-clock option allows the UART to keep running during low-power modes. It is even able to detect startbit's falling (=leading) edge asynchronously, i.e. it needs no kernel clock while waiting for the startbit, reducing further power consumption. Depending on how its kernel clock is set in RCC, it then either sets a request signal to RCC to start HSI, or un-gates the continuously running LSE, to enable reception of the incoming UART frame.

The Low-Power UART (LPUART) present in newer STM32 families is almost identical to the "general-purpose" UART, and probably are only a particular configuration variant of it. This is underpinned by the fact that it does not have separate register struct nor register-bits/bitfields symbols in the CMSIS-mandated device headers, and the USART-related struct/symbols are used for LPUART, too.

While LPUART is usually able to wake up the chip from a deeper sleep setting than UART (e.g. Stop 2 in 'L476), this is probably only due to power supply distribution choices in given STM32 model.

The one notable difference in LPUART in contrast to UART is, that LPUART does not employ the 16x/8x oversampling, only a single sample in what its baudrate generator marks as middle of the bit. Also, LPUART baudrate generatorhas 8 fractional bits rather than 4 as has UART. This allows LPUART to run at 9600 Baud with the usual 32.768kHz LSE. However, this means in average cca 3.4 clocks per bit, and while the fractional baudrate generator ensures quite precise timing in the long run, individual bits are sampled by either 3 or 4 clocks. This can be seen also on the LPUARTs Tx jitter under this combination (blue waveform is Tx of 0x55, green is LSE)[1]:



The need to generate the Rx sampling point from 3 or 4 clocks results in significant jitter in exact placement of the sampling point, too. In other words, relatively small error in timing of incoming bits (caused e.g. by asymmetry in skew of high-to-low and low-to-high transitions in some transceivers, or simply by imprecise transmitter clocking) may result in incorrectly received frames.

However, even with perfect input signal, LPUART Rx at 9600 Baud clocked from LSE may be prone to errors in reception in the low-power modes, especially in first byte(s) of a packet (as reported here and here). The reason probably is, that ungating LSE after the asynchronous detection of startbit falling edge requires some time (and maybe some synchronization), and that possibly delays the sampling points by one LSE period, worsening the placement of sampling points. In this case, the RM requires to have the LSE clock into LPUART kernel ungated all the time, by **setting the LPUART_CR3.UCESM bit**.

This increases somewhat consumption ('L476 DS gives 2.0µA/MHz, i.e. at 32.768kHz the consumption increase is around 65nA), and that's the price for reliable Rx under these conditions[2].

Even if this requirement is explicitly documented in the LPTIM chapter of RM, it is easy to be overlooked and may result in unpleasant surprise.

1.Tested on 32L476GDISCOVERY (which was inexplicably discontinued by ST), source code here.

2.Similarly to this combination, in case of UART/LPUART kernel clock being HSI, in low-power mode, the HSI startup time may cause startbit to be missed. In 'L476, worst-case HSI16 startup/stabilization is 1.2µs+5µs; comparing that with bit duration (cca 100µs at 9600Baud, cca 8.6µs at 115200Baud) indicates, that higher baudrates would be unreliably received under these conditions, unless HSI is kept running all the time.

Created: 24.Jun 2023

163.UART Rx corruption if glich occurs during 2nd half of stopbit

Byte received by UART may be received as corrupted, i.e. having a different value than received, if a glitch (a short 0 pulse) occurs during second half of stopbit. There's no indication that this occured in the UART's flags, RXNE is set as normally.

This issue pertains probably to the "new" UART used in all STM32 since 'F0/'F3. It is documented in an erratum e.g. for 'G0B1, but was observed e.g. also on 'L431 which errata don't contain this erratum. It takes time until ST propagates a new erratum to all relevant models' errata.

What could be the possible mechanism of this problem? UARTs should stop the Rx process by sampling the stopbit at its middle, as due to possible timing mismatch between transmitter and receiver, this may be in reality the end of stopbit and it's just time to start to wait for the next frame's startbit downgoing (leading) edge. The just received frame should be transferred to the Rx buffer at this point, and the RXNE flag should be risen.

Here, transfer of the received frame appears to be delayed for some reason. The glitch (a short 0 pulse) during startbit detection should be ignored; however, it appears that it also through some error in design impacts the frame which is still in the receiver's shift register (or its equivalent), and then the corrupted frame is transferred to the Rx buffer.

How can such glitch happen? UART transmitters which can control a transceiver (e.g. RS485) often switch the transceiver's direction at the middle of last transmitted byte (again, to accomodate for possible timing mismatch between parties on a shared bus and prevent collision). Some less-than-perfect combination of transceivers and their loads (e.g. cables) may result in such short unwanted glitch to be generated. UARTs are normally designed to ignore short glitches, so such conditions usually go unnoticed.

Created: 1.July 2023

STM32 gotchas

164.Strange characters received in terminal in PC? Parity or baudrate mismatch, or RS232 converter was used for UART.

UART is perhaps the most ubiquitous method of communication between electronic devices, often used also to transmit debug or other information from a microcontroller such as STM32 to PC. Even if it is relatively simple and straighforward to set it up, it is also not uncommon to make a mistake, which usually results in receiving unintelligible characters.

The most usual problem is baudrate mismatch between transmitter and receiver. This happens usually at the mcu side, and may be consequence of incorrectly set clock train, or misunderstanding of what the primary clock source is. It is relatively easy to check the baudrate by transmitting 0x55 ('U') repeatedly, as that results in a continuous train of 0s and 1s of a single bit duration, which then can be checked using oscilloscope or a logic analyzer. The receiver still may receive some characters, but they are "strange", and often there are also errors (framing/noise) in reception (but they may go unnoticed, as e.g. many terminal programs don't display the UART flags).

Similar problem may be caused by mismatched parity and/or bit-per-frame settings, although in this case some characters may come through correctly. In case of STM32 this problem may be caused by the fact, that unlike any other UART, the STM32 UART counts parity bits into the total number of bits.

Sometimes confusing characters are received even if baudrate, parity, bit-number settings match the transmitter, but a RS232 converter is used. While RS232 indeed was the dominant physical protocol used for UART communication two-three decades ago, currently it is rarely used, and to connect mcu to PC, a "straight" USB-to-UART converter has to be used, rather than a USB-to-RS232 (the latter would be used ONLY if there is an UART-to-RS232 converter already connected to the mcu, the two RS232 converters would then "cancel out" each other).

The reason why such connection "sort-of-works" and characters - even if "strange" - are received, is, that **RS232 levels are effectively inverted to UART levels**: logic 0 is signalled by a high voltage (+3V..+15V) and logic 1 is signalled by a low voltage (-3V..-15V)[1]. Even if the mcu's Tx pin cannot produce the negative voltages, RS232 receivers are often built so that their decision level is above 0V (around +1V).

And while it may seem that connection in the other direction (from RS232 to mcu) inevitably results in mcu damage due to high voltage levels, and it indeed does result in damage sometimes; but surprisingly quite often the mcu survives such connection thanks to relatively high output impedance of RS232 transmitters together with the protection structures built into the mcu IO pins.

1.In the RS232 realm, the official nomenclature for logic 0 is "space" and for logic 1 is "mark", but that is rather confusing ("mark" being the idle state) and rarely used.

Created: 8.July 2023

165.UART/USART conundrum

As we've already discussed here, UART and USART (and even LPUART) in STM32 is the same module, except UART is a "stripped-down" version with some of the features simply omitted at chip design time. Even USARTs are not all created equal, individual incarnations have an individual mix of features, again as it was configured by the chip designers.

Fact that UART and USART are just two names for the same module is underpinned by the fact, that UARTs don't have individual structs defined in the CMSIS-mandated device headers, nor symbols for its registers' bits and bitfields - they all use the USART-related structs/symbols (and again, same applies for LPUART).

However, ST stubbornly features UARTs as a different module, eg. in using different naming of modules (UARTx vs. USARTx) and different symbol names (such as RCC_APBxENR.UARTxEN vs. RCC_APBxENR.USARTxEN) - which makes automatic code generation unnecessarily harder. Also in libraries such as Cube, ST provides entirely unnecessarily separate code for UARTs and USARTs, with separate source files, separate function names etc.

This all is entirely unnecessary and superfluous, resulting in confusion and unnecessary extra work. While clearly it's a legacy burden, ST could've easily dealt with this by providing a simple but complete and concise descripition of differences between UART and USART, augmented by a "legacy translation layer" in the CMSIS-mandated device header and in any libraries.

Even ST then makes the shortcut in some documentation, and talks only about "UART" or only about "USART" (e.g.in 'L476 RM, while there is officially USART1..3 and UART4 and UART5, at several places description like "USARTx, x = 1,2,3,4,5" can be found).

Here, in gotchas, systematically, "UART" is used exclusively, to denote both UART and USART.

Created: 15.July 2023

166. FMC/FSMC conundrum

FMC (Flexible memory controller) is the parallel memory interface in the STM32 world. It started under the moniker FSMC (Flexible static memory controller) in the initial 'F1 family. It was designed to handle

- conventional SRAM/pSRAM/EPROM with plain parallel address and data bus and nOE/nWE/nCS style of control, with optional external waitstate input;
- memories with multiplexed address/data buses in the "x51-style", with dedicated bus for upper part of address, where lower part of adderss is output onto data bus together with an address-latch signal, ahead of the actual data cycle;
- synchronous memories, which work in bursts and where lower part of address is generated by a counter from clock signal
- NAND FLASH (which in fact are a special case of synchronous memory), with poorly documented hardware ECC

FSMC supports several memories to be connected simultaneously, and as it requires a large number of pins, it is fully supported only on the larger-pin-count devices - the major limitation is, that lowermost address pins are not present in below and including 100-pin packages. The 100-pin package still contains enough FSMC pins to drive multiplexed-address memories, which includes conventional parallel-SRAM with a single parallel external latch. 100-pin package also allows to have a parallel data bus with the usual SRAM control for devices which require limited addressing, found often in TFT-LCD controllers, which usually require only one address line - ST (ab)uses this fact in marketing FSMC to have a dedicated "LCD-mode".

FSMC in this form was used also in the newer 'F2 family and the first models of 'F4 family ('F405/'F407). However, in the next "power" model of the 'F4 family, the 'F427/'F429, ST introduced an extension of the FSMC module, to support synchronous dynamic RAMs (SDRAMs). These have not only synchronous interface with a relatively complex signal multiplexing and timing, but they also require periodic refresh - and the hardware is expected to handle all this seamlessly for the user to experience plain reads and writes from the system (processor, DMAs, LCD controller (L) point of view.

To honor the fact that FSMC is no longer supports only static memories but also dynamic memories, "static" thus letter S was omitted from the name, confusingly undermining the common expectation, that more letters in module naming means more capabilities. The different naming is reflected also in naming of structs and symbols of course makes also porting or writing universal libraries unnecessary harder.

To cause even more confusion, ST mixes and matches the naming and features very freely, so that capabilites can't be inferred from the name and careful reading of the manuals is needed. For example, in the higher-end-'L4 and 'L4+ manuals, as well as in 'G4 manual, the module is called FSMC in the headlines (and indeed it does not support SDRAMs), but then in the first sentences of FSMC chapter, the following sentence appears:
This memory controller is also named flexible memory controller (FMC).
and subsequently the rest of manual consistently uses the moniker FMC, including naming of registers (and that also in the CMSIS-mandated device header). However, in the various selectors, 'L4/'L4+ appear in the group having FMC (together with those STM32 which do have genuine FMC with SDRAM support) while 'G4 appears in the same selectors as having FSMC.

At the end of the day, FMC/FSMC is a surprisingly complex piece of hardware, which is relatively poorly described in the manuals with plenty of handwaving when it comes to details; featuring quite a couple of errata; and having minor but for some applications important differences between individual incarnations in individual (sub)families. While there's only one FMC/FSMC module per STM32, and there are not so many projects using it, so the problem with naming is perhaps a bit less annnoying than that of UART/USART and maybe could be called minor; FMC/FSMC is still a confusion and a completely unnecessary addition to all the complexities of that module.

Created: 22.July 2023

167.'H7 longevity revisited

In g57, we've discussed the possible lifetime shortening of 'H7 devices operated at elevated temperatures. In 'H74x errata (ES0392 rev.9) there's an interesting erratum, "Possible drift of USB PHY pull-up resistor", which may shed some light at the mechanism which causes the 'H7 lifetime shortening:

When the $V_{DDUSB33}$ equals 3.6 V and the USB Idle resistor (pull-up 1 connected between 3.6 V and ground) remains activated for a long period of time, the pull-up resistor value might drift, reaching the maximum value defined in USB Specification.

Let's put forward, that this erratum is pertinent **only to the older, rev.Y** version of the chip.

Also, the real-world effect is less dramatic than it may sound at first reading - the erratum follows with:

The degradation can be observed after USB PHY pull-up continuous activation in the following conditions:
- *10 years of continuous transmit operations or*
- *A 86 million reset operations (from the USB host) provided that the reset period is 2.5 ms (duration might vary depending on host reset duration).*

The proportion of applications where USB is in Transmit (let alone USB-Reset) state any significant portion of its lifetime is probably miniscule. However, why these strange conditions?

The original USB specification requires a relatively precise, 1.5kΩ±5% resistor as pullup on D+ indicating Full-speed device to the upstream hub. This kind of precision, while trivial with todays discrete resistor, is hard to achieve in integrated circuits. So, an Engineering Change was added to the USB standard, allowing to use two less-precise resistors of different values together with switches, which select the appropriate resistor in various modes of the USB transceiver (this EC is part of the .zip file which forms the current USB2.0 standard, and contains a detailed rationale for this solution). And, so it happens, that the resistor which is used during Transmit, is of the higher value, 1475Ω..3090Ω. To reduce silicon area, instead of using a long resistor, a shorter and thinner resistor has been probably designed.

The nature of this erratum again appears to point to electromigration, effects of which are more pronounced on extremely narrow structures (here: the USB pullup resistor) allowed by the 45nm technology used to build the 'H7. This structure was probably modified to be somewhat wider in the subsequent revisions of the chip, to remove the limitation in question.

This erratum highlights the challenges associated with shrinking features on integrated circuits. While the pressure to produce microcontrollers cheaper but with more features, can still be felt, it is time to realize that we are probably hitting the physical, economical and rational limits of this process. Further development in microcontrollers lies probably in making thing smarter, rather than just making them more complex and adding more memory.

In memory of Gordon Moore - IC pioneer, co-founder of Intel, and author of Moore's Law.

Created: 29.July 2023

168.Inconsistent numbering considered harmful - especially mixing 0-based and 1-based numbering

There are many repeated features in STM32, so it's natural to employ some form of numbering to denote the individual features. In STM32, unfortunately, ST uses - with no particular system - both 0-based and 1-based numbering.

That some of these problems stem from ST compounding STM32 from purchased IP components, is no excuse - it's ST's task to unify style and screen users from malpractices of its suppliers. The current practice made ST's life easier, but it makes the life of thousands of users harder.

Some of particular issues stemming from this problem have been already discussed, e.g. the DMA/DMAMUX numbering mess.

An even more spectacular example is in numbering of the SYSCFG_EXTICR (as experienced painfully e.g by user J.v.W.) - while Reference Manual numbers them 1-based, the CMSIS-mandated header defines them as a standard (thus 0-based) C array; yet the same header also defines symbols which originate from the 1-based numbering. Thus, we then have to write confusing concoctions like:

```
SYSCFG->EXTICR[3] = SYSCFG_EXTICR4_EXTI15_PB;
```

Created: 5.August 2023

169. Inconsistent naming considered harmful- especially between RM and CMSIS-mandated device headers

In both documentation and the CMSIS-mandated device headers, ST sometimes for no particular reason changes name of a given module, register or register-bit/-bitfield, between STM32 families. As a consequence, not only code porting becomes hard, but also studying differences in documentation and code (using textual search), too. One example of this issue is the backup SRAM, which is called BKPSRAM in 'F2/'F4/'F7, yet, inexplicably, called BKPRAM in 'H7.

While this inconsistency is bad enough, ST makes it worse by naming of symbols in the CMSIS-mandated device headers. Not only are they sometimes inconsistent from family to family, some of them also uses different spelling for modules/registers/bits/bitfields than the RM.

Most striking examples of this problem include the FMC/FSMC and UART/USART naming issue. The inconsistent numbering problem is related, too. Some particular issues of this kind have been discussed also here.

Created: 12.August 2023

170.Programming 'G0/'C0 from built in bootloader has to be well-planned, as default bootloader entry depends on empty FLASH

Some users wish to program STM32 only through its built-in bootloader, avoiding the SWD connections - possibly to spare pins and/or PCB area in designs, where some of the interfaces used in bootloader are brought out anyway.

In the 'G0 (and related 'C0) families, the bootloader entry method (described as Pattern 11 in AN2606) in a factory-new chip uses the fact that empty FLASH results in boot from system emory(where the built-in bootloader resides). This allows to program a factory-new chip using the built-in bootloader.

Similar bootloader entry mechanism exists in other STM32 families - most notably, Pattern 6, used in several models of 'F0, 'L0, 'L4 families. However, there is a notable difference between Pattern 6 and Pattern 11: in the factory-new default setting of option bits in Pattern 6 enables using the BOOT0 pin (which is often a dedicated pin in related models) to enter the bootloader - option bit nBoot0_SW, which if cleared, would replace BOOT0 pin by nBoot0 bit, is by default set.

Contrary to that, in Pattern 11, the option bit nBOOT0_SEL steering between BOOT0 pin (which in these models is often shared with a GPIO) and nBoot0 option bit, is by default set which means that nBoot0 option bit is selected - and that in turn is by default set, which means no bootloader entry. In other words, **factory-new default setting of option bits involves no mechanism how to enter the bootloader by outside manipulation of pins**. That means, that after the initial programming of FLASH thus FLASH is not empty anymore[1], bootloader entry is not available.

So, if users wish to have continuing access to bootloader in some form, either the firmware (possibly, a custom bootloader) should be written so that it deliberately enters the bootloader code upon some stimulus; or, should the BOOT0 pin be used, programming the nBOOT0_SEL option bit should be the first step when a factory-new chip is programmed.

---

1. The FLASH empty check mechanism (setting FLASH_ACR.EMPTY bit) is performed only upon power-on reset or explicit reload of option bytes using FLASH_CR.OBL_LAUNCH. So, after initial programming of FLASH, upon "pushbutton reset" without power cycle, the bootloader is entered again. This may result in even larger surprise to the unaware user, that after a subsequent powerdown-powerup cycle there's no bootloader entry anymore.

Created: 19.August 2023

171. 'G0/'G4 bricked after programming RDP1, if BOOT_LOCK option bit is set

The primary protection mechanism against unauthorized code readout/copying in STM32 is Read Protection, accomplished through a dedicated Option byte, RDP. It can be set to 3 levels - RDP0 (short for RDP Level 0) is the default, unprotected mode; RDP1 prevents against code readout, but the chip still can be reprogrammed (while a bulkerase is performed); RDP2 is the highest readout protection where no further programming is possible and also the SWD/JTAG debugging mechanism is completely disabled.

In the 'G0/'G4 families, there is a new option bit, BOOT_LOCK, which is probably intended to increase security by preventing to boot from System memory or RAM, ie. boot is always from the User FLASH. However, combining BOOT_LOCK together with RDP1 has the following effect (quoting directly from RM):

Caution: If BOOT_LOCK is set in association with RDP Level 1, the debug capabilities of the device are stopped and the reset value of the DBG_SWEN bit of the FLASH_ACR register becomes zero. If DBG_SWEN bit is not set by the application code after reset, there is no way to recover from this situation.

So, in effect, BOOT_LOCK=1 and RDP1 has the same effect than RDP2; except that contrary to RDP2, it can be "reverted from inside".

More dramatic of course is the impact to the unaware, if this combination is set inadvertently.

The 'G0x0 "Value line" subfamily oficially does not have the BOOT_LOCK bit; however, as the die is the same as of some of the "full-fledged" 'G0s, this distinction is only in the programming software (e.g. CubeProgrammer) which does not display the state of this bit and does not allow to program it.

Created: 26.August 2023

172. UART receives zeros? Using pullup with UART_Rx may be a good idea (not STM32-specific)

The pin set for UART_Rx is effectively a high-impedance input, and it relies on the connected transmitter to provide the correct idle (usually high) voltage level when there are no frames transmitted. However, this may be not always the case - the transmitter may go into high-Z, e.g. if it is a second microcontroller and it is being reset, or it has a bug in its firmware. In some designs, the transmitter can be physically disconnected (e.g. behind a connector); in others, removing power supply or external (e.g. USB) connectivity from transmitter may leave the connection be floating.

In such cases, the floating line may either drift to zero, generating a surprising BREAK frame (which in STM32 UART is received as 0x00 together with Framing Error), or the UART may receive spurious characters through coupling to neighbouring signal lines. While software ought to be written in such a way that such events won't cause major problems (e.g. UART data might be packetized/checksummed, timeouts may be implemented, received data sanity may be tesed, error flags should be meticulously handled, etc.); it is always better to have also the baseline data as error free as possible.

Perhaps the simplest way to ensure a fairly secure input level in case of potentially floating UART_Rx is to use the humble pullup. It may be an external pullup of any appropriate value, but in most cases it's enough to switch on the internal pullup. In contrast to external pullup, it costs nothing (it came "bundled" with the mcu), and its nominally 40kΩ value is appropriate in most practical cases. There may be implications in mixed-voltage systems, though, as pullups shouldn't be switched on with 5V input signals even if the given STM32 pin is 5V tolerant.

Of course, this solution is not applicable in severely power-constrained designs. The internal pullup increases total power consumption by cca 100µA when the transmitter pulls the line down; if the communication is sparse (which is often the case), the average power consumption is proportionally lower.

Created: 2.September 2023

173. Some remarks on UART used in halfduplex mode

UART in the STM32 features a halfduplex mode, enabled by setting UART_CR3.HDSEL. When enabled, the UART stops listening on the UART_Rx pin (as assigned in GPIO matrix). Instead, it connects its internal Rx node to the Tx node, and stops actively driving the UART_Tx pin unless transmission is in progress. This is a rarely used feature, so here are some remarks about its usage:

- Halfduplex mode enables to connect several microcontrollers using a single signal wire. There's no enforcement of which participant transmits when: transmitter does not check its own receiver and starts transmitting immediately after its data register is written. So, measures have to be taken to prevent pin damage in case of two transmitters pushing against each other.

  RM in description of this mode in the UART chapter requires to set the Tx pin to Open Drain mode with pullup, which is a foolproof method but also limits baudrate (due to RC constant of pullup and parasitics). Another option is to use series resistors to limit current - Ohm's law and datasheet-given constraints apply (and again, the resulting RC constant limits achievable baudrate). Chosen solution also implies level of immunity against induced noise.

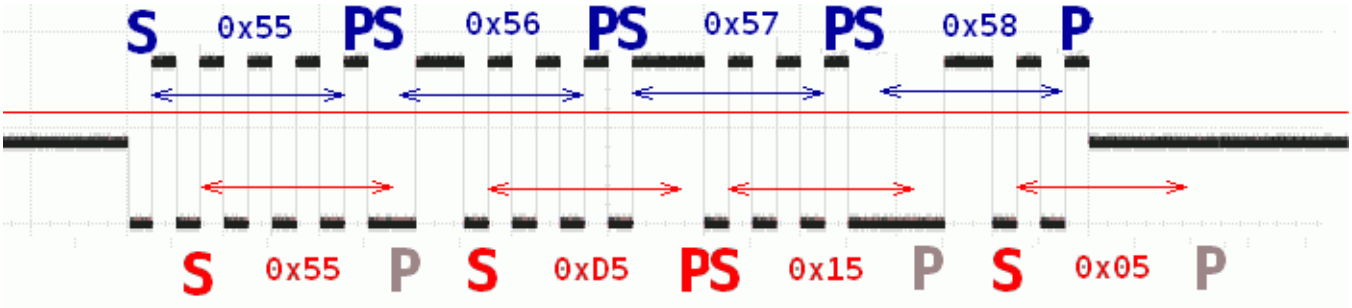
- In case of such multi-drop bus, it may be a good idea to implement in software some form of scheduling (usually a master-slaves scheme, but others exist too; RS485 users are familiar with this concept).

- Receiver listens to transmission (i.e. it "sees the echo"), so software has to be written with this in mind.

- "Direction" (i.e. whether output transistors are enabled or not) of UART_Tx pin, when halfduplex is enabled, is controlled by the UART module. It is automatically turned to input whenever the UART module is not transmitting. The UART_Tx pin has to be set as Alternate Function (with appropriate AF number). This works in the same way also in the GPIO-atypical 'F1 family.

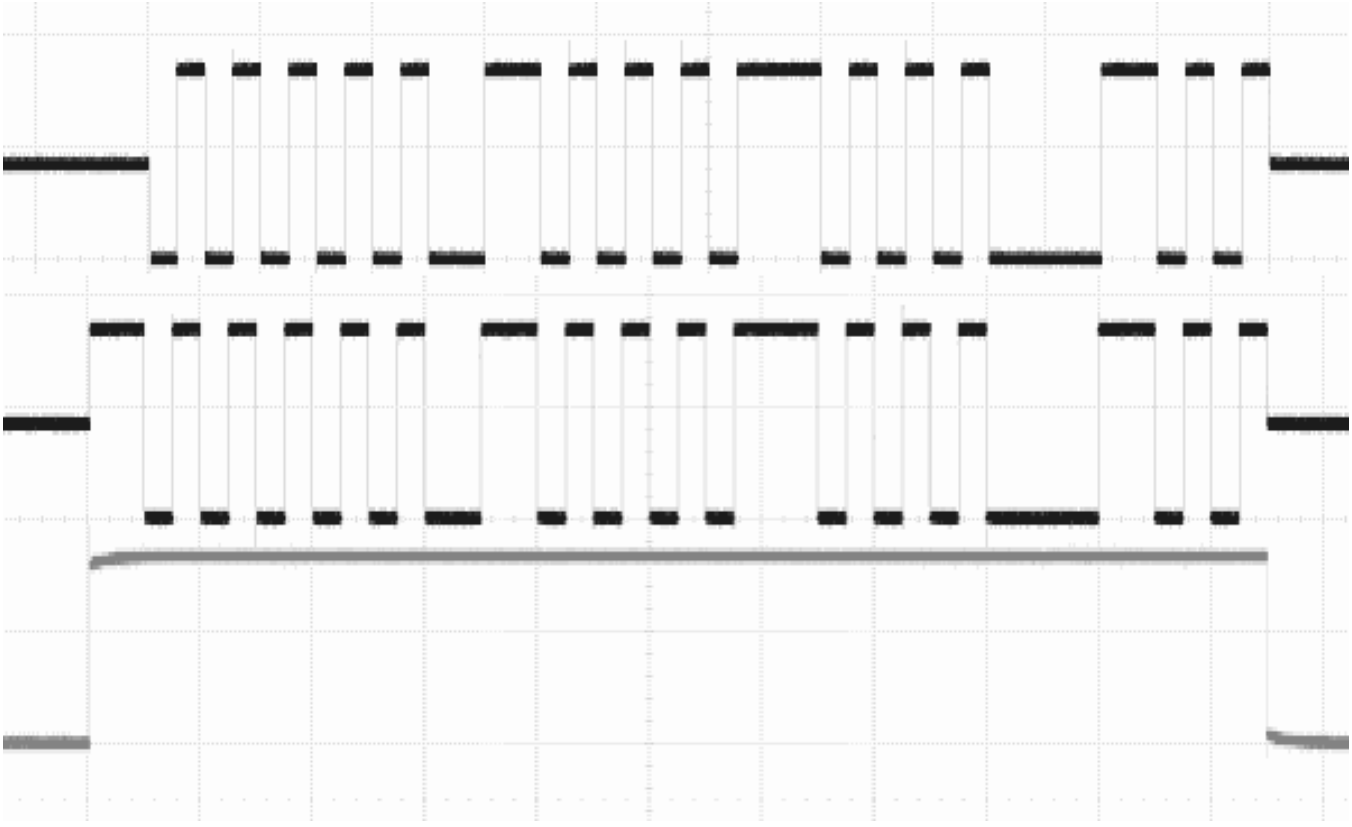
- When not transmitting, the input pin is floating and potentially drifting to zero, which results in missing the first startbit and thus data corruption. It is reasonable to use a pullup to define idle level.

  In figure below, to demonstrate idle level (i.e. periods when transmitter is not active), UART_Tx pin was pulled by a voltage divider to middle point between GND and VDD; and sequence of 0x55-0x56-0x57-0x58 was transmitted. Would idle level drift to ground, it would be equivalent of the receiver's decision level being at the red line - receiver would see a continuous 0 until the end of the first startbit, so it would see the second bit as startbit. Receiver would receive corrupted data, with Framing errors at first, third and fourth byte.



- The newer UART version (since 'F0/'F3) features the UART_DE signal to control transceivers (e.g. for RS485) during transmission. This signal is an identical copy of the internal signal which controls the GPIO pin's "directionality". In this version of UART, it is possible to generate an actively driven idle period before (and after) transmission using the USART_CR1.DEAT (.DEDT), up to almost 2-bit-long period.

  This is illustrated in the figure below, with the same setup as for the previous figure. The top waveform was taken with USART_CR1.DEAT = 0; the middle waveform with USART_CR1.DEAT = 0x1F while the bottom waveform is the UART_DE signal for the latter.



- UART_Tx pin in halfduplex mode is not driven during the IDLE frame (generated as consequence of setting UART_CR1.TE), exactly as the UART_DE signal.
- In case of pin placement pressure and if Rx-only UARTs are needed and the UART_Rx pin is unavailable (used for a different purpose), using halfduplex may be a neat trick to use the UART_Tx pin for Rx.
- Another neat trick to spare one pin is to use halfduplex mode in conjunction with the similarly naturally halfduplex RS485 bus.

Created: 9.September 2023

174. Anecdotes about missing VDDx

Most of the STM32 models have multiple VDD and VSS (GND) pins. Besides the basic "digital" VDD/VSS, there are also "specialized" VDDx (and sometimes VDDx/VSSx pairs), such as the analog VDDA/VSSA (and although not supply, VREF+/VREF- pins partially belong to this group, too); VBAT for battery-backup (mostly for RTC but also BKPSRAM where available); VDDUSB for, well, USB; VDDIO2 for a couple of GPIO pins; VDDDSI/VDDSD/VDDSDMMC/VDDPHYHS for the MIPI/DSI display interface, Sigma-Delta converter (unique to STM32F373), SD/MMC unit, and internal USB High-Speed PHY (unique to STM32F723 and its "value-line" counterpart), respectively.

In the vast majority of cases, these pins should not be left unconnected, even if the related functionality is not explicitly used. Generally, all VSS/ground pins should be connected to a common ground potential; most of the "specialized" VDDx pins, if unused, can be tied to the "basic digital" VDD. The datasheet is always the definitive guide how to treat these pins.

If used, some of the "special" voltage domains are behind an isolation switch, which has to be switched on by software. VBAT-powered domain is behind a switch which is controlled by a circuitry related to brown-out and power-on-reset; although the backup SRAM if present, is powered from an internal low-power regulator which has an additional dedicated switch.

In any case, the "special" voltages have to be present and of required level, on the given VDDx pins, should the related functionality be used. While this sounds self-evident, if the related VDDx is inadvertently absent, the actual symptoms may be quite surprising. Following are two anecdotes resulting from such omission.

User wynandsp reported an interesting problem, where the USB D+ pin upon enabling the internal 1k5 pullup to indicate Full-Speed Device, generated short cca 10µs pulses every cca 450µs.

It turned out that by omission, power on VDDUSB was missing, only a 100nF capacitor was connected to this pin. While one would expect an entirely non-functioning PHY in this case, or a low-voltage but steady output level resulting from internal leakages, the pulsed output was genuinely confusing.

During hardware shakedown, an STM32F407-based design with VBAT connected to a CR2032-type coin cell exhibited confusing symptoms after a power cycle, akin to incomplete reset. Additionally, time in RTC was lagging a couple of seconds after each power cycle.

Investigation revealed, that a HF-choke through which VDDA was connected to VDD, had been removed in a previous test, and by mistake it had not been soldered back in place before the next tests. During powerdown, voltage on VDDA did not follow VDD but was kept relatively high thanks to its decoupling capacitor, thus the brownout-poweron reset circuitry did not engage and the internal VBAT-domain switch remained improperly switched to the powered-down VDD domain. As a consequence, the whole digital part of the mcu, and parasitically also part of the external circuitry; were supplied from a rapidly dying coin cell, and the incomplete power cycle on VDDA subsequently resulted in improper power-on reset.

Created: 16.September 2023

175. Data register reads zero in 'H7 SPI/I2S, if read width is narrower than data width set for SPI/I2S

The SPI/I2S unit in the 'H7 family is a significantly more complex implementation than the SPI/(I2S) modules in other STM32 families. This complexity results in arguable benefits, but also in surprising behaviour in certain corner cases.

One of these surprises, as experienced by user Alefal and explained by user LCE is, that when I2S/SPI SPI2S_RXDR register is read out (either by processor or by DMA) using a narrower data width than the module is configured for (in SPI_CFG1.DSIZE for SPI mode, or in SPI_I2SCFGR.DATLEN for I2S mode), zeros are read out. In the mentioned case this occured, when the module was configured for 24-bit I2S frames but the DMA reading out the data was configured for 16-bit (halfword) transfers.

This behaviour is documented in the relevant RMs as an inconspicuous remark in the rather lengthy narrative related to the RxFIFO:

The FIFO data access less than the configured data size is forbidden. One complete data frame has to be always accessed at minimum.

Created: 23.September 2023

176. SPI/I2S set to PCM runs twice the expected sample rate

The SPI/I2S module in STM32 in I2S mode allows to select, by setting SPI_I2SCFGR.I2SSTD, one of several commonly used audio protocols: I2S, left-justified, right-justified, or PCM. While the former three are basically a variant to the same scheme - stereo samples framed by a left-right signal - PCM is a relatively simple mono protocol.

Reference manual provides formulas to calculate/set sampling rate, but those are intended for the I2S-like protocols. For PCM protocols, they result in twice the intended sampling rate - logically, the same number of data frames results in twice the samples in MONO than in STEREO.

As manual does not contain PCM-specific guidelines, users have to adjust the formulas accordingly. Similarly, users have to take this into consideration when using libraries such as Cube/HAL, which may be written without this PCM-specific detail in mind.

Issue spotted by user vincent.

Created: 30.September 2023

177.I2SxExt allows not only fullduplex SPI, but also dual-Tx/dual-Rx

SPI/I2S is a combined module in STM32, which extends the SPI functionality with I2S (and similar audio protocols) capability. At the end of the day, I2S is a similar shift-register-like protocol as SPI is. Not all incarnations of SPI module implement also the I2S functionality, but in most STM32 families around half of the SPI modules present in given model do.

The SPI functionality is implemented as full-duplex, i.e. there are two data pins, one for input and one for output; and they even can swap direction as required by master or slave role. However, quite inexplicably, the I2S functionality in the same module is implemented only as simplex[1].

This means, that there is only one data pin available in I2S mode, set to either input (for audio recording) or output (playback) as required. However, audio CODECs - i.e. chips which contain both an ADC and a DAC - usually have two data pins - one dedicated for input and one dedicated for output - while they share a single set of clock signals (SCK (data/bit clock, sometimes denoted as BCLK), WS (word-select clock, sometimes denoted as LRCK for left-right clock), and in some cases a high-frequency master clock MCLK). This means, that to use such chip, especially when recording and playing back simultaneously, two SPI/I2S units have to be employed, and their respective WS and SCK pins have to be mutually interconnected. If the STM32 is to play master role in this connection, it would need to generate the clocks in one of the SPI/I2S units (i.e. have it set as master), and the other unit needs to be set as slave. It does not matter, whether it's a master transmitter and slave receiver, or vice versa, master receiver and slave master (with the respective data pins connected to the CODEC's data pins as appropriate for their direction). If the CODEC or some third connected device generates the clocks, both SPI/I2S units in STM32 would need to be set as slaves.

As interconnecting pins externally means wasting a precious resource, ST worked around this problem in some STM32 families[2] by attaching a second, simplified, unit to the SPI/I2S units, called I2SxExt (where x matches the number of its respective SPI/I2S unit, e.g. I2S3Ext is attached to the SPI3/I2S3 unit). I2SxExt is the same design as SPI/I2S, except it's simplified by having its SPI and master (clock generation) capabilities removed. It has its SCK and WS pads internally attached to the SCK/WS pads of its respective SPI/I2S counterpart. It means, that I2SxEXT has only its data pad exposed through GPIO to a pin, so together with its SPI/I2S counterpart they provide the fullduplex I2S capability.

In short, **I2SxExt is functionally an identical unit to I2S/SPI (with functionality restricted to slave I2S)**, exactly in the same way as with two separate I2S/SPI with interconnected clocks, except that **it has its SCLK and WS signals connected to its respective SPIx/I2Sx unit internally**.

One consequence of this fact is, that - same with two separate I2S/SPI with interconnected clocks - the I2S/I2SExt couple does not necessarily need to be set up as fullduplex, i.e. with one unit set as input and other as output. Sometimes, the application requires to have two simultaneous outputs or two simultaneous inputs (i.e. four audio channels in the same direction). The I2SxExt+SPI/I2S combo is perfectly capable to handle this situation, too.

---

1. ... except in 'H7. However, the SPI/I2S unit in 'H7 is an overcomplicated design, very different from SPI/I2S in other STM32.

2. 'F3 and 'F4, although not in all models of these families, e.g. 'F410 has no I2SxExt. In the newer 'F7, 'L4, 'G4 families, where audio capability is expected, while simplex I2S is still retained in some SPI/I2S models, the primary method of interfacing audio codecs (including fullduplex) is through the SAI unit.

Created: 7.October 2023

178.I2SxExt_SD and SPIx_MISO are different AF in 'F4, but the same AF in 'F3

As discussed previously, the I2SxExt modules are stripped-down slave-I2S-only versions of the SPI/I2S module, nonetheless they are separate modules connected to their "master" SPIx/I2Sx counterpart only at the IO-signals level (internal to the chip, but external to modules themselves). In the 'F4 family, this is underpinned also by the fact, that the sole I2SxExt signal (I2SxExt_SD, i.e. data) which is brought out to an external pin, is at the same pin but separate Alternative Function (AF) than the corresponding SPIx/I2Sx module's MISO pin (which is unused in I2S mode).

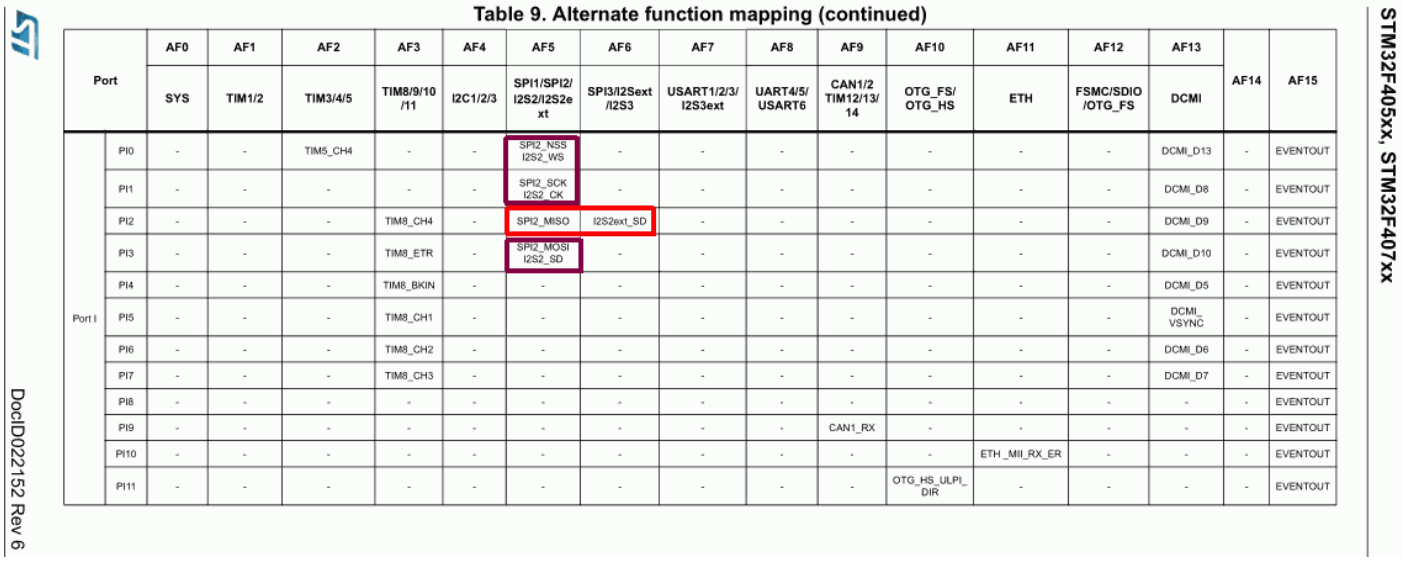
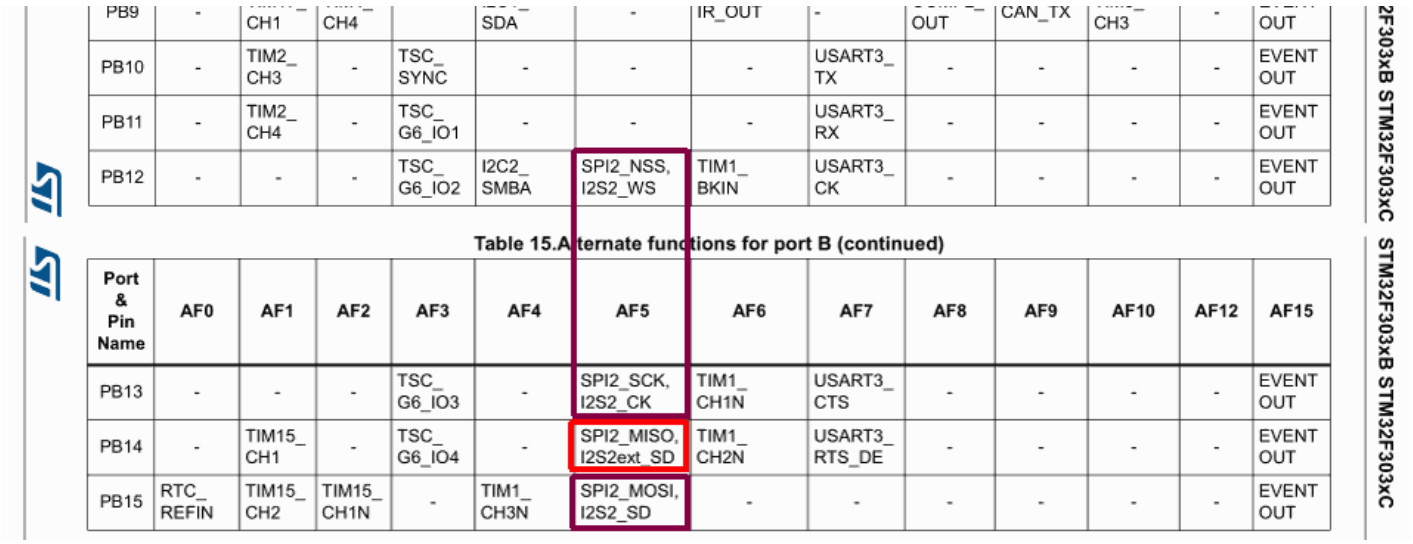


Table 9. Alternate function mapping (continued)

Port		AF0 SYS	AF1 TIM1/2	AF2 TIM3/4/5	AF3 TIM8/9/10/11	AF4 I2C1/2/3	AF5 SPI1/SPI2/I2S2/I2S2ext	AF6 SPI3/I2Sext/I2S3	AF7 USART1/2/3/I2S3ext	AF8 UART4/5/USART6	AF9 CAN1/2 TIM12/13/14	AF10 OTG_FS/OTG_HS	AF11 ETH	AF12 FSMC/SDIO/OTG_FS	AF13 DCMI	AF14	AF15
Port I	PI0	-	-	TIM5_CH4	-	-	SPI2_NSS I2S2_WS	-	-	-	-	-	-	-	DCMI_D13	-	EVENTOUT
	PI1	-	-	-	-	-	SPI2_SCK I2S2_CK	-	-	-	-	-	-	-	DCMI_D8	-	EVENTOUT
	PI2	-	-	-	TIM8_CH4	-	SPI2_MISO	I2S2ext_SD	-	-	-	-	-	-	DCMI_D9	-	EVENTOUT
	PI3	-	-	-	TIM8_ETR	-	SPI2_MOSI I2S2_SD	-	-	-	-	-	-	-	DCMI_D10	-	EVENTOUT
	PI4	-	-	-	TIM8_BKIN	-	-	-	-	-	-	-	-	-	DCMI_D5	-	EVENTOUT
	PI5	-	-	-	TIM8_CH1	-	-	-	-	-	-	-	-	-	DCMI_VSYNC	-	EVENTOUT
	PI6	-	-	-	TIM8_CH2	-	-	-	-	-	-	-	-	-	DCMI_D6	-	EVENTOUT
	PI7	-	-	-	TIM8_CH3	-	-	-	-	-	-	-	-	-	DCMI_D7	-	EVENTOUT
	PI8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PI9	-	-	-	-	-	-	-	-	-	CAN1_RX	-	-	-	-	-	EVENTOUT
	PI10	-	-	-	-	-	-	-	-	-	-	-	ETH_MII_RX_ER	-	-	-	EVENTOUT
	PI11	-	-	-	-	-	-	-	-	-	-	OTG_HS_ULPI_DIR	-	-	-	-	EVENTOUT

Maybe this has proven to be confusing to the users, because ST in the newer 'F3 family ST merged I2SxExt_SD with the corresponding SPIx_MISO to the same AF:



PB9	-	CH1	CH4		SDA	-	IR_OUT	-	OUT	CAN_TX	CH3	-	OUT
PB10	-	TIM2_CH3	-	TSC_SYNC	-	-	-	USART3_TX	-	-	-	-	EVENT OUT
PB11	-	TIM2_CH4	-	TSC_G6_IO1	-	-	-	USART3_RX	-	-	-	-	EVENT OUT
PB12	-	-	-	TSC_G6_IO2	I2C2_SMBA	SPI2_NSS, I2S2_WS	TIM1_BKIN	USART3_CK	-	-	-	-	EVENT OUT

Table 15.Alternate functions for port B (continued)

Port & Pin Name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF12	AF15
PB13	-	-	-	TSC_G6_IO3	-	SPI2_SCK, I2S2_CK	TIM1_CH1N	USART3_CTS	-	-	-	-	EVENT OUT
PB14	-	TIM15_CH1	-	TSC_G6_IO4	-	SPI2_MISO, I2S2ext_SD	TIM1_CH2N	USART3_RTS_DE	-	-	-	-	EVENT OUT
PB15	RTC_REFIN	TIM15_CH2	TIM15_CH1N	-	TIM1_CH3N	SPI2_MOSI, I2S2_SD	-	-	-	-	-	-	EVENT OUT

That I2SxExt is still a separate module, is witnessed by it occupying its own separate address space.

Created: 14.October 2023

179. I2S clock numbering in 'F446 and 'F413 is confusing

In several STM32, there is a choice of clocks available for I2S, including a dedicated input from external pin through the GPIO matrix, and a dedicated PLL unit. There is a multiplexer in RCC to choose between them, controlled by I2SxSRC fields in RCC_DCKCFGR register. The "x" in name of the field (and thus naming of respective clock signal as **"I2Sx clock"**) is bound to the **APBx bus**, on which the SPI/I2S unit(s) fed by this clock reside.

In 'F446 and 'F413/'F423 it so happens, that SPI1/I2S1 is on APB2, and SPI2/I2S2 is on APB1 (there are more SPI/I2S units in either of these STM32, though). Thus, I2S1 is fed by "I2S2 clock" and I2S2 is fed by "I2S1 clock".

This can get very confusing...

Created: 21.October 2023

181. On the Commutation event (COM) in some timers

Timers, which **have channels with complementary output** - TIM1, TIM8, TIM20 called collectively Advanced Timers, and TIM15, TIM16, TIM17 - have an option to have the individual channels' **enable bits (CCxE, CCxNE) in TIMx_CCER** and the **output-compare-mode bitfield (OCxM) in TIMx_CCMRx, preloaded**[1]. This option is by default off and is switched on by setting TIMx_CR2.CCPC bit.

However, unlike other registers in timers which are preloaded, the "intermediate" values for these bits/bitfields (i.e. those directly written and being read back by the processor or any other busmaster) are **not loaded into the "working" registers upon Update event**. Rather, there is a dedicated event for loading these bits/bitfields, called **Commutation event** (COM). Remarkably, this event is there only for this sole purpose of loading preloaded enable bits and mode bitfields, intended to change them simultaneously at the same time for every channel of a timer.

Commutation event can be generated in two ways: in any case it can be generated "manually", by setting TIMx_EGR.COMG; but if TIMx_CR2.CCUS is set, commutation event is generated also upon rising edge of the TRGI signal.

---

1. This is not the only feature not directly related to complementary channels, shared by this group of timers - for example, they all also have the repetition counter.

Created: 4.November 2023

182. TIM channel can't be enabled in TIMx_CCER, if TIMx_CR2.CCPC is set by mistake

In timers which have TIMx_CR2.CCPC bit, setting this bit causes TIMx_CCER.CCxE/CCxNE and TIMx_CCMRx.OCxM to be preloaded. These bits are then loaded to the "working" registers only upon the Commutation event, which is quite unlikely to be generated in most usual timer setups.

If TIMx_CR2.CCPC bit is set inadvertently, by mistake, subsequent changes of TIMx_CCER.CCxE/CCxNE and TIMx_CCMRx.OCxM are ignored (pending no Commutation event). The outward effect of this is, that while changes to TIMx_CCER.CCxE/CCxNE and TIMx_CCMRx.OCxM can be read back, they take no effect, so for example, enabling PWM on a TIM channel has no effect on the pin's state. This can be very confusing.

How could possibly TIMx_CR2.CCPC be set, without user realizing it?

One possibility is, that user calls the LL_TIM_CC_EnablePreload() function, as its name may lead to incorrect conclusion that it enables preload of the Capture-Compare registers (TIMx_CCRx). (In fact this is done by a quite similarly named LL_TIM_OC_EnablePreload() function.)

Another such possibility is to call the LL_TIM_OC_Init() function with not fully initialized LL_TIM_OC_InitTypeDef struct. If the rarely used OCIdleState or OCNIdleState fields are not initialized, they may end up corrupting content of TIMx_CR2 register, including the TIMx_CR2.CCPC bit.

Created: 11.November 2023

183. After enabling clock in RCC, delay may be needed before accessing some peripherals

The notion of mcu peripherals having individually controlled clocks is a relatively novel concept in mcus, which appeared roughly together with advent of 32-bit mcus. It was enabled by the increasing number of transistors per chip, so this feature was not seen as an unnecessary complexity anymore. At the same time, it was necessitated with the huge number of peripherals on the new mcus, vast majority of which are not used in any particular application - while increasing consumption, would their clock be left running.

However, for users migrating from 8-bit mcus this was a surprising change. So much so, that it became the topic of the very first gotcha...

While enabling a peripheral in STM32 is as easy as set the respective enable bit in one of the enable registers in RCC, in some STM32 it may come as a surprise, that accessing a register in the peripheral immediately after it has been enabled, may go wrong: writes may get ignored by the peripheral and reads may return unexpected values. This is due to the SoC nature of these mcus, where signals may need to pass synchronizers between various clock domains, and this may take some time.

ST realized this problem quite soon (e.g. in the STM32F407 errata, *Delay after an RCC peripheral clock enabling* erratum appeared in 2012, one year after the 'F407 was introduced) and documented it in errata in older STM32 families and directly in RM in newer STM32 families.

Unfortunately, one of the offered workarounds - and that one which appears to be the most foolproof, easy to implement, and is even used in Cube - i.e. performing a dummy read of target peripheral's register after enabling clock and before any "real" access - is experimentally proven to be wrong in at least STM32F407/'F429.

The fact, that the problem couldn't be reproduced on 'F411 nor 'L476, and slightly different overall timing of related events on these models indicates, that the bus fabric attached to the processor has subtle yet significant differences between individual models/subfamilies, which are entirely undocumented (the processor itself is a component sourced by ARM, indicates the same version/revision number, so it's most likely identical in all four discussed models).

Solution to the root problem is compounded by the fact, that the required delay is very short, a few system clock cycles; and that in C or any other higher-level language there's little guarantee for code ordering (beyond what *volatile* can offer) yet alone execution timing (which in the 32-bitters is matter of uncertainty for multiple reasons anyway). Most probably the vast majority of users has never encountered the problem, even without adding a delay in their C code (or using the library macro which has the delay incorporated).

One easy and effective way to avoid having problems stemming from this issue is in the traditional controller-oriented mcu development. In that, the particular application is seen as central, thus all resources needed for that application are controlled/enabled at once, at the beginning of the program (this also makes reading of such program easier, as it's immediately clean which and how resources are enabled/set up). So, here, after enabling all clocks, an operation not requiring access to just-enabled peripherals is performed - e.g. setting up the various clocks, clearing/initializing variables in RAM, etc. - and that provides enough time so that all peripherals are enabled safely by the time they are going to be accessed for the first time. This is also more effective as far as code size and execution time is concerned.

However, this approach goes against the grain of the now modern "driver" approach, where all manipulation of resources (including clocks, GPIO) are performed individually, "locally"; in the code, and at the time, where/when given peripheral is initialized. In that case, delay has to be provided in each case of enabling the peripheral's clock.

Created: 18.November 2023

184. DMAMUX clock - how to enable

In newer STM32, DMA requests (triggers) are selected using a separate unit, DMAMUX. As most other peripherals, DMAMUX needs to have clock enabled in order to allow its registers to be accessed.

However, two different schemes are employed in SMT32 families to enable DMAMUX clock:

1. In 'G4, 'L4+, 'L5 families, there is a dedicated DMAMUX enable bit, DMAMUX1EN[1].
2. In 'G0/'C0 and in 'H7[2], there is no separate DMAMUX bit. DMAMUX is enabled together by enabling DMA unit. In case where there are two DMA units (e.g. 'G0Bx), enabling either of the DMA units enables also DMAMUX.

The lesser-used low-power-clock bits in RCC are arranged similarly; and so are the reset bits, too: in STM32 families where DMAMUX is enabled together with DMA, it is also reset together with DMA; in families with separate DMAMUX clock enable bit there is also a separate DMAMUX reset bit in RCC.

These differences may lead to unpleasant surprises, when migrating between STM32 families.

---

1. With the exception of 'H7 (see footnote 2), there is always only one DMAMUX unit. Even if there are two DMA units in some models, the single DMAMUX unit serves both. Confusingly, ST uses an inconsistent naming of that single DMAMUX unit, calling it DMAMUX1 at some places and just DMAMUX at others, even if referring to that one particular instance.

2. In 'H7, there are three DMA units served by two DMAMUX units (the fourth MDMA does not have requests routed through a DMAMUX). The reason lies in the complex bus system - DMAMUX1 serves DMA1 and DMA2 in the "not the highest speed, mostly peripherals" D2 domain, and DMAMUX2 serves BDMA in the relatively separate low-power D4 domain. (Again, ST is not very meticulous in adding the respective number to "DMAMUX" even when referring to particular one of the two units, so documentation has to be read with care.)
DMAMUX2 is enabled together with BDMA. The 'H7 manual (RM0433 Rev 8) does not document how to enable DMAMUX1 clock; most likely it's enabled together with either of DMA1 or DMA2, similarly to 'G0Bx.

Created: 25.November 2023

185. What's all this kernel clock stuff, anyhow?

By now it's a well-known fact, that individual peripherals in 32-bit mcus such as STM32 have individually controlled clocks. Most STM32 peripherals - especially in older STM32 families - have only a single clock, which is equal to the clock of APB bus[1], to which given peripheral is connected.

However, some peripherals have **two clock domains**. One domain contains mostly the registers, through which the peripheral is controlled from the rest of the mcu (mostly processor, but also DMAs), and this domain is clocked from the **APB clock**. The second domain is the "execution" portion of the peripheral, performing the core functionality of it (e.g. in SPI the shift register and its associated clock and control circuitry) - this portion in documentation of newer STM32 peripherals is called *kernel*, and thus its clock is called **kernel clock**.

There are two options for the kernel clock

1. it can be **synchronous with the APB clock**, either being derived from it, or being derived from a common source clock. Example for the former is the ADC clock in some of its settings in most STM32. The most notable example for the latter is the Timer clock, which is derived from the same AHB clock as APB clock, but in some cases can be divided by a smaller factor thus run faster than the APB clock.
2. or it can be **asynchronous to APB clock**, either from an internal source (e.g. LSI in case of IWDG/RTC/LPTIM, LSE in case of RTC/LPUART, a different PLL than the "main" e.g. for I2S/SAI/USB), or from an external source through a dedicated pin (e.g. I2S_EXTCLK/SAI_EXTCLK for I2S/SAI, externally clocked LPTIM, TX/RX clocks for ETH, ULPI clock for OTG_HS used with external PHY).

While separate kernel clocks allow greater flexibility in choosing optimal operational frequencies for various portions of the application (e.g. the mcu's system clock may not be necessarily tied to I2S/digital audio clock), it also imposes requirements on the peripheral's designers. Transferring signals and data between the two domains may require synchronization in order to present a coherent set of signals to the other side. Configuration registers are relatively "easy": usually there is one "enable" bit, and configuration registers are required to be all set before "enable" bit is set, so that gives time for these signals to propagate to kernel side before "enable" bit starts the kernel side's activity[2] [3]. Reading out single bits (status flags) is also relatively simple, as they are mostly mutually independent, so they can be simply transferred from kernel to APB side. However, reading out numerical values may get tricky and requires precise synchronization with APB clock, otherwise the read out value may get incoherent. Even more tricky is a readout of multi-word values, where either a lock/release mechanism has to be employed (such as in RTC_SSR/TR/DR with BYPSHAD=0), or the user may be required to perform a read-twice-until-identical-values procedure (e.g. in reading out the LPTIM_CNT register).

From programmer's point of view, synchronous kernel clocks are usually relatively "harmless", i.e. they usually have no or very little requirements on the programmer and have small to negligible impact on the resulting application. For example, triggering ADC clocked from synchronous APB/n clock from a timer results in precise synchronization between timer and ADC, whereas in case of asynchronously clocked ADC there would be an uncertainty (jitter) in the trigger caused by resynchronization.

Asynchronous kernel clocks are much more tricky. They may impose requirements on minimum APB clock (as is in case of ETH and USB; but also RTC requires APB be at least 7x the RTC clock). There may be timing requirements on individual setup steps (e.g. in newer I2C, switching off and on the PE enable bit requires a 3-APB delay). Reading or consecutive writing registers in the kernel clock domain may impose automatic delays (e.g. in LTDC accessing LCD-domain registers will stall the APB bus for 6-7 APB clocks plus 5 LCD clocks). Readout may require the read-twice-until-identical-values procedure. As data transfer between clock domains is more complex, errors can be easily made in the hardware's implementation, such as the RTC inclomplete-lock erratum or ETH's *Successive write operations to the same register might not be fully taken into account* erratum.

As the number of transistors in STM32 increases, so does also the complexity of peripherals. In newer STM32, more and more peripherals have separate APB and kernel clocks, so more attention has to be given to effects resulting from this separation, especially if asynchronous clock is selected to be used for the kernel.

---

1. ... or AHB bus (e.g. in case of ETH or OTG_HS). For brevity, we will use "APB" throughout this chapter, even if in some cases it's AHB.

2. Usually, writing such peripheral while "enable" bit is set, is not prevented by hardware, and results in undefined behaviour of the peripheral. It's the responsibility of user to read carefully the documentation and act accordingly.

3. Nomenclature across peripherals is not unified, and sometimes "enable" bit does not start the kernel's functionality, but gates the kernel clock altogether, for example in LPTIM. In this case, writing to registers of kernel clock domain is not possible until such "enable" is set.

Created: 2.December 2023

186. Some kernel clocks are not selected/enabled in RCC

In many STM32 dual-clock peripherals, the kernel clock is enabled together with APB clock by a single bit in respective RCC_APBxENRx register. Also, if kernel clock can be selected from multiple choices, this selection is often done in a specialized RCC register (ST confusingly uses various names for this register in different families, including RCC_CFGRx, RCC_DCKCFGRx, RCC_CCIPRx), and the default selection is usually the APB clock. This allows to ignore the existence of separate kernel clock for most users, who don't need a different/asynchronous kernel clock for the peripherals, and then the handling of a given peripheral is identical to models where the same peripheral has single clock.

However, this is not always the case, and there are exceptions to this "rule".

In description of peripherals, the fact that kernel clock source is selected in RCC, is sometimes confusingly described as clock being selected/generated "at product level". This is consequence of manuals being compiled from separately written chapters (in the same manner as the chip itself is compiled from separately designed IP blocks), with little attention paid to adjust the chapters to given STM32 model and to unify terminology/signal names.

Here are some examples of maybe surprising kernel clock enable/selection methods:

- ADC is perhaps the most varying peripheral across STM32 families, and there are also various treatments of its kernel clock. Usually, it's selected in ADC itself (ADCx_CFGRx.CKMODE), in multi-ADC models in the common ADC control register (ADC_CCR.CKMODE); although in some cases there's also a secondary selection within RCC (if asynchronous clock is chosen). Kernel clock does not have a conspicuous separate enable bit, but it's probably gated by bits which are described to power the ADC on (ADON, ADVREFEN).
- ETH has dedicated input pins for clocks (ETH_MII_TX_CLK, ETH_MII_RX_CLK/ETH_RMII_REF_CLK) so it would appear that there's no need for selection, but the RMII/MII switch in SYSCFG impacts the treatment of the clock for the respective interface mode (in RMII mode, both Rx and Tx clocks are generated from the same single input), so it can't be omitted when using RMII instead of MII. Slightly surprisingly, ETH has not only peripheral (in this case AHB rather than APB) clock enable bit in RCC_AHBxENR, but in the same RCC_AHBxENR also dedicated enable bits for both kernel clocks (ETHMACTXEN/ETHMACRXEN), and even a third clock enable, ETHMACPTPEN, should the PTP feature of ETH be used.
- OTG_HS if used with external PHY, uses pin OTG_HS_ULPI_CK as kernel clock. It also has a separate enable bit, RCC_AHB1ENR.OTGHSULPIEN.
- in 'F723, the built-in USB HS PHY has a dedicated clock+power control unit, USBPHYC, so that's where the OTG HS kernel clock is set up (from a dedicated PLL) and enabled. However, the register portion of USBPHYC unit has its own APB enable bit in RCC_APB2ENR.
- in the combined SPI/I2S unit, the registers portion and kernel of SPI is clocked from APB clock, but I2S kernel has a separate clock. This in 'F0 is derived from SYSCLK, in 'F3 it can be selected between SYSCLK and I2S_CKIN pin, and in 'F4/'F7 can be selected between I2S_CKIN pin and a dedicated PLL. Source selection is controlled by RCC_CFGR.I2SSRC bit, while one of SPIx_I2SCFGR.I2SMOD/I2SE bits (or AND of both) serves as I2S kernel clock enable.
- RTC is in the VBAT (backup) power domain, and its kernel clock is selected (from LSE, LSI and HSE/32) and enabled in RCC_BDCR. Once selected, this selection can be changed only after a backup-domain reset. In some STM32 models, RTC has also a separate register-domain (APB) clock enable bit.
- some peripherals (e.g. SAI) function as clock-slave, i.e. the clock from external master serves as kernel clock. It appears, that some of these peripherals' implementation still requires "internal" kernel clock to be present and of a reasonably high frequency, as it is used in the synchronizer. This functionality is poorly documented.

Created: 9.December 2023

STM32 gotchas

187. Where's that clock enable? It's tied to some other peripheral.

In STM32, most peripherals need to have clock enabled in some of the RCC_AxBxENRx registers. However, sometimes it's not easy to spot the bit which enables some particular peripheral.

One of the reasons for this is, that sometimes a clock enable bit is common for several peripherals. This sometimes leads also to confusingly named enable bit.

Here are some examples:

- DMAMUX in some models is enabled together with DMA (or one of the DMAs) it serves
- a very common combo is SYSCFG with the "minor analog" modules, such as
  - RCC_AxBxENRx.SYSCFGEN enabling SYSCFG + COMP + VREFBUF in 'L4 and 'G0
  - RCC_AxBxENRx.SYSCFGEN enabling SYSCFG + COMP + VREFBUF + OPAMP in 'G4 (note that 'L4 does have a separate RCC_APB1ENR1.OPAMPEN bit)
  - RCC_AxBxENRx.SYSCFGCOMPEN enabling SYSCFG + COMP in 'F0
- RCC_AxBxENRx.DCMIEN in 'L4+ enables either DCMI or PSSI, depending on the enable bit within the respective module (PSSI_CR.ENABLE bit and the DCMI_CR.ENABLE bit must not be set to 1 at the same time)
- in 'F4 (and possibly other models with dual bxCAN), CAN1 and CAN2 share the 512 byte (filter) memory, so in effect CAN2 can't work without CAN1 clock being enabled, even if CAN1 is not used
- in 'G4, RCC_AxBxENRx.ADC12EN enables both ADC1 + ADC2, and RCC_AxBxENRx.ADC345EN enables ADC3 + ADC4 + ADC5. While ADC1/ADC2 and ADC3/ADC4 form pairs tied together with a common portion of hardware (enabling to operate in dual mode), ADC5 is independent, so it's surprising that it is enabled together with ADC3/ADC4.
- In 'F4 (and probably also 'F2 and 'F7), there's a triple ADC (ADC1/ADC2/ADC3) with a common portion, which allows to use combined modes. The common portion is used also to set up the kernel clock, which is common to all 3 ADCs. The common portion also has bits to enable the temperature sensor/VBAT/VREFINT inputs, although they are connected only to ADC1. While all 3 ADCs do have separate enable bits, the common portion does not have a separate enable bit, and enabling either of the ADCs enables also the common portion (btw. this detail is not documented by ST)[1,2].

---

1. Most peripherals in RCC have, besides an enable bit, also a reset bit, usually in the same position in RCC_AxBxRSTRx as is the respective enable bit in RCC_AxBxENRx. ADC1/ADC2/ADC3 in 'F4 is an exception in this, too, as they have only a single common reset bit, RCC_AxBxRSTRx.ADCRST, positioned where RCC_AxBxENRx.ADC1EN is.

2. As an interesting although useless detail, enabling either ADCx clock enables all 3 ADC registers for read, but only the enabled ADCx registers for write (see also footnote 1 here)

Created: 16.December 2023

188. ... and then there's also no clock enable at all...

Peripherals in STM32 have a clock enable bit, which has to be set before given peripherals can be used in the program. This is so important that it's gotcha number 1.

However, this does not pertain to some of the peripherals, which have clock unconditionally enabled all the time:

- RCC itself - obviously, otherwise bits in it could not be set
- quite obviously, too, peripherals which are part of the processor, i.e. NVIC and the SBC block of registers
- DBGMCU (which is somewhere at the boundary between DBG in processor and the non-processor items) - there are exceptions in 'F0/'L0/'G0
- EXTI (for whatever reason) - there may or may not (if it turns out to be only ST's documentation error) be an exception to this in 'F410/'F412/'F413
- APB interface of RTC - there are several exceptions
- IWDG
- memories (SRAM, FLASH/FLASH programming interface) - these in some STM32 models/families don't have enable bits at all and are unconditionally enabled all the time, in some models/famiels they have enable bits which may or may not be set by default, and in some models/families they have only low-power enable bits.

Created: 23.December 2023

189. ... or the clock is enabled by default...

In various STM32 models/families, there are peripherals which *do* have a clock enable bit, but it is **set by default**.

As most enable bits are clear after reset, bits which are enabled by default may cause confusing behaviour if an unaware user clears the whole RCC_AxBxENR register containing such bit, e.g. when trying to restore reset state, typically when transitioning from bootloader to application.

Examples of such bits include:

- **CCMDATARAMEN**, enable bit for the CCM RAM in 'F4
- **DTCMRAMEN**, enable bit for the DTCM RAM in 'F7
- **FLASHEN**, enable bit for the FLASH registers in 'L4/'L5/'G0/'G4 - this bit can't be cleared unless FLASH itself is powered down by setting FLASH_ACR.RUN_PD. Confusingly, the same bit in 'L1 is called **FLITFEN**; and in 'L0 it's called **MIFEN**.
- **RTCAPBEN**, enable bit for interface between RTC and APB (bus matrix) - this bit is present in all STM32 models, and it is also enabled only in some STM32 models by default

There is also a confusing exception (sort of): in the 'F1, 'F0 and 'F3 families, there are FLITFEN and SRAMEN enable bits for the FLASH interface and SRAM; and indeed, they are set by default. However, clearing these bits wouldn't switch off the FLASH interface and SRAM during Run mode, as they are relevant only during Sleep mode[1].

---

1. Contrary to other STM32 families, the 'F1/'F0/'F3 families do not have dedicated low-power clock enable registers.

Created: 30.December 2023

191. Datecode in chip marking is not unique

STM32 chips marking contains a datecode which identifies the year and week when given chip was manufactured. Contrary to the 4-digit YYWW format used by many chip manufacturers, ST on STM32 uses a 3-digit format, YWW, where ony the last digit of the year is used.

STM32 are now around for more than a decade - and given ST's 10-year commitment it is likely that some STM32 models will see more than 20 years of being manufactured. And that together with single-digit year means, that some datecodes will inevitably repeat. This makes identification of the exact manufacturing date a challenge.

This of course is not a problem with the "usual" (and undoubtedly preferred) business model, where a relatively large batch of chips is bought directly from ST or through some of the primary bulk distributors and immediately put into production. Problems arise when "old new stock" is used e.g. as it happened not uncommonly during the "Great Parts Shortage" of early 2020s; or when older equipment containing STM32s is to be maintained or refurbished/updated/reprogrammed, which will hopefully occur at increasing rate in the future, as that's how real ecology works.

Examples of cases when the manufacturing date may be relevant include:

- STM32F303xB/C erratum *Imprecise VREFINT calibration values*, which pertains to datecodes less than 628, meaning chips manufactured before week 28 in 2016.
- In 'F105/107, according to AN2606, bootloader on devices whose date code is lower than 937 is in version 1.0, which has severe flaws; in this case this is 2009 week 37.

Recently, ST started to add QR-codes to the marking, and this - together with other parts of the marking - supposedly allows to identify exactly the manufacturing time. However, structure of this marks and their exact meaning is not documented publicly, so such identification is possible only through requests through ST.

Created: 13.January 2024

192. What's up with these -P and -Q suffixed STM32? They have different pinout than the non-suffixed ones.

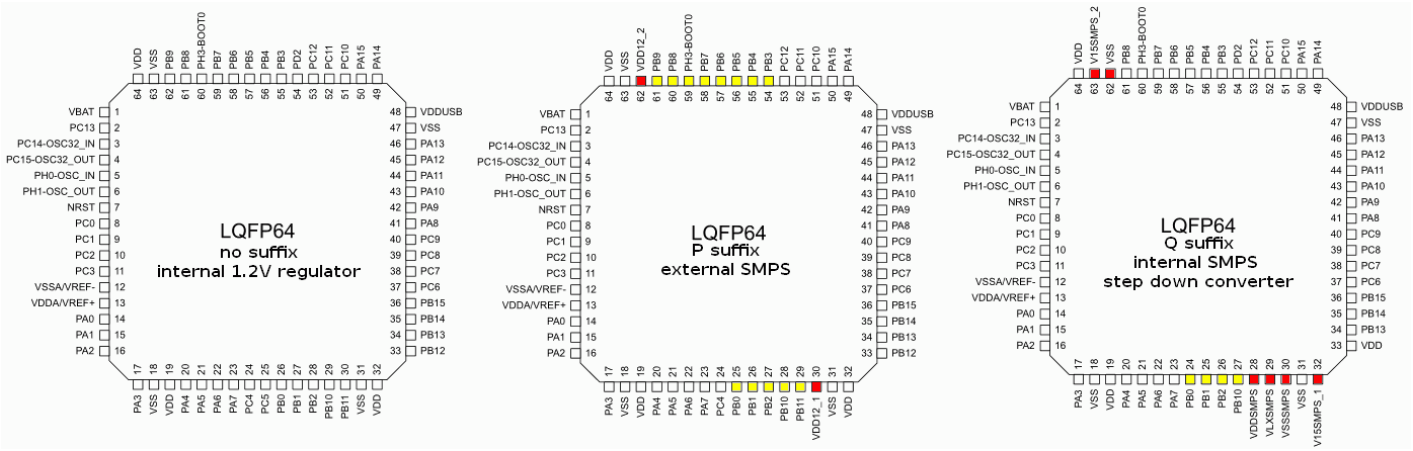
While STM32 VDD is generally supplied from a 1.8V-3.3V source, only a small portion of the chip actually works at this voltage (mainly GPIO). Most of the chip's logic - i.e. processor, bus fabric, peripherals - runs at a significantly smaller voltage (sometimes marked as V_{CORE}), 1.2V and below. This both decreases power consumption through leakages, and allows to use smaller CMOS transistors geometry. This voltage is generated by an on-board linear power supply regulator (LDO).

Being linear, around a third to a half of the energy consumed in Run mode is turned into heat in this regulator. This may be a concern especially in low-power devices.

To circumvent this problem, ST started to offer some of the newer STM32 models also in a variant, where the internal voltage is exposed to a pin, so the internal LDO can be replaced by a high-efficiency **external switched-mode power supply (SMPS)**. These variants are marked with **suffix P**.

In some even newer STM32, ST even started to include the SMPS onto the chip, so that only an external inductor and bulk capacitor needs to be connected to have a highly power efficient solution. Variants with **on-chip SMPS** are marked with **suffix Q**.

Unfortunately, these modifications mean, that some of the signal pins had to be given up to allow these new power-related pins. But what's even more annoying is, that ST decided also to shift some of the remaining signals to the neighbouring pins, making design of compatible boards harder. The following figure illustrates this problem on the STM32L552 in LQFP64 package, which is manufactured in all three variants, i.e. STM32L552RxTx, STM32L552RxTxP, STM32L552RxTxQ:



As the suffix is located relatively deeply in the parts' marking, it's easily overlooked and may lead to unpleasant surprises, especially if the user is not aware of the problem, and especially if the power supply chain is stressed and depleted, as it was the case in the last couple of years.

Created: 20.January 2024

193. My STM32 board does not work. Are all these pins properly connected?

When bringing up a new hardware design, sometimes it happens that a connection gets overlooked, or on a prototype, a bad solder joint or a short prevents some pin being physically connected to where it belongs. This may result in a situation where "nothing is working", i.e. the STM32 does not respond to a known-good programmer or debugger.

Here is a list of pins, which should be connected in some particular way for basic functionality of an STM32 design:

- **all VDD/VSS** pins must be connected to a stable power source providing voltage within specified range (usually 1.8V..3.3V) and properly decoupled.
- **VDDA/VSSA** pins must be connected to specified power supply too. Some STM32 models (probably all 'Fxx families) require VDDA to be within a relatively small difference from VDD; this is important especially during powerup, so the VDDA circuitry must be designed with this in mind (for what may happen if this is not maintained see e.g. here, here and here).
- **VCAP**, if present, must be connected to properly sized capacitors as given in datsheet; if there are several such pins, this applies to all of them.
- **PDR_ON** and **BYPASS_REG**, if present, must be connected as given in the datasheet.
- leaving **VBAT** unconnected may be asking for additional trouble, althought the VBAT brownout erratum requires software intervention (performing backup domain reset upon startup), even if the otherwise unused VBAT is connected to VDD as specified in DS.
- **BOOT0** if present; in newer STM32 this may not be needed if relevant option bits are set so that BOOT0 pin is ignored (if booting from System memory is required to use the built-in bootloader, BOOT1 pin, if present, must be appropriately connected, too).
- contrary to popular belief, it's not absolutely necessary to have any special circuit for NRST due to built-in pullup; although a decoupling capacitor is recommended by the DS. However, NRST must not be driven actively high by external circuit, otherwise internal reset sources won't work.

For each STM32 family/subfamily, ST does provide an appnote named *Getting started with STM32xxxx MCU hardware development* or similarly. These do contain most the above information and more, so it's essential to check them out before starting to make a new hardware design. Another good source of inspiration for hardware design are be the Nucleo/Disco/EVAL boards, for which ST provides schematics and BOM on their respective product pages.

Created: 27.January 2024

194. PLL does not output expected frequency or does not work at all, as physical ratio is not identical to the value written to registers

In every STM32, there is at least one PLL in the RCC module, which generates the high frequency clock (>16 MHz) for overall mcu operation from relatively low frequency (0.1MHz..16MHz) primary clock source (HSI, MSI, HSE).

In the older and lower-end STM32 ('F1, 'L1, 'F0, 'F3, 'L0), the PLL is controlled by a single parameter, which determines the PLL's multiplier. However, the value written to respective register field (usually RCC_CFGR.PLLMUL) is not equal to the ratio by which the PLL's input frequency is multiplied. In most STM32 with simple PLL, the physical ratio = (value + 1), i.e. value of 0b0000 results in x2 frequency, value of 0b0001 results in x3, etc; with the exception that both 0b1110 and 0b1111 result in x16 frequency.

The 'L0 and 'L1 families have a different form of simple PLL: its output frequeny ratios can be described only by a table (x3, x4, x6, x8, x12, x16, x24, x32, x48).

In the higher-end and newer STM32[1], the PLL has a pre-scaler[2], the multiplier itself, and usually 3 output dividers, potentially serving 3 clock branches with different frequencies. The prescaler's ratio is denoted M, the multiplier's N, and the output dividers' P, Q and R.

Here again, the physical divider/multiplier ratios are not necessarily equal to the values written to respective registers fields (in the dedicated PLL register, RCC_PLLCFGR). The details differ across STM32 families, so it's necessary to carefully read the description of these registers fields when migrating between families.

The N multiplier ratio usually does match the written value.

In 'F4 and 'F7, the P post-divider's ratio = 2 * (value + 1) (i.e. /2, /4, /6, /8); this divider's output can be used as system clock. Strangely, the Q (and R in the SAI PLL where present) ratios equal to the written values, but values of 0 and 1 are forbidden; the same applies to the M prescaler.

In 'G0, the M prescaler's, and all 3 post-scaler's ratio = value + 1.

In 'L476 (historically the first member of 'L4 family), the M pre-scaler's ratio = value + 1; R (which here supplies system clock) and Q divider's ratio = 2 * (value + 1), and the P divider is selectable only between 2 quite strange values, /7 and /17.

In all other 'L4 and also in 'G4 and 'L5, the same scheme is used, except that there's an extra field for the P-postscaler, following the "ratio equals value, but minimum value is 2" scheme. However, for (quite uncomprehensible) backward compatibility reasons, the original bit for P switching between the weird /7 and /17 ratios is maintained, and is valid when the new field is set to 0b0000.

---

1. except 'H7 and 'U5, which have a much more complex fractional-N PLL

2. in high-end families with multiple PLLs, there may be only one common pre-scaler; this fact in itself may be a source of mistake

Created: 03.February 2024

195.PLL output does not work, if its enable bit is not set

In higher-end and newer STM32 RCC, PLL units have usually 3 outputs, marked P, Q and R, which divide down output of the same oscillator (VCO) by 3 different ratios, before they are used for various purposes (system clock, 48MHz USB clock, audio/I2S clock, independent clock for other peripherals etc.) in the rest of given STM32.

In newer STM32 families ('G0,'G4,'L4,'L5), besides a register bit-field for the related divider, these outputs have also individual enable bits (PLLPEN, PLLQEN, PLLREN). If this enable bit is not set, given PLL output does not provide the expected clock, even if the rest of PLL is set up correctly.

While this sounds quite logical, sometimes it may be source of surprising malfunction. For example, this post reports, that Cube does not set the respective PLL output enable bit, if this PLL output is used as source for MCO output without being used as clock of any other module.

Created: 10.February 2024

196. Not all RTC are created equal

In various STM32 families/models, there are basically three versions of RTC, which were developed historically. **AN4759** provides an excellent overview of their capabilities, although it inexplicably omits RTCv1[1].

RTCv1, used only in the **'F1** family, is basically just a 32-bit counter[2], plus a simple alarm.

RTCv2 is used in the majority of "contemporary" STM32 (**'F0/'F2/'F3/'F4/'F7/'L0/'L1, most 'L4/'L4+/H7**). It features a **calendar**, but not a counter; two alarms plus a wakeup, input frequency error correction (**calibration**), synchronization, configurable output pin. Backup registers are now part of the RTC and there's a tamper input, which both clears the backup registers in hardware and provides an interrupt and a timestamp. Atomic calendar readout is supported by a hardware lock mechanism, which unfortunately has a small error propagated across all RTCv2 models (see here, here and here).

RTCv3 is the RTC version present in newest STM32 models (**'C0/'G0/'G4/'H5/'L5/'U5 and some newer of 'L4/'L4+/'H7**). It adds back the binary **counter** in addition to calendar (see caveats below) and separates the backup registers and tamper (which has now more features) to a **separate TAMP module**. (but they are still intimately interconnected, so this makes their usage somewhat more complicated). AN4759 contains a list of STM32 models/families which are equipped by RTCv3 - one maybe surprising element in it is, that while most 'L4 models have RTCv2 the lowest-end 'L41x and highest-end 'L4P5 have RTCv3.

However, not all incarnations of RTCv2 and RTCv3 are the same, across various STM32 families/models. In some models, some features are omitted - while there are more such subtle omissions (and they are well documented again in AN4759), notably, some RTCv2 don't have SSR register, and some RTCv3 don't have the simple counter.

While having a **calendar (called "BCD mode" by ST)** appears to be beneficial for many simple applications, which only need to keep and display/output local time; it is in fact a major hindrance for any more sophisticated application, where time zones and time differences have to be supported/calculated. In the latter case, the **simple counter (called "binary mode" by ST)** is a more convenient solution, as these operations are practically impossible to be performed in calendar mode, and conversion to epoch time format and vice versa is inevitable; the simple counter would provide epoch time so only one conversion would need to be performed.

ST effectively undermined this need for both counter and calendar in almost all but a handful of the most modern STM32. In 'F1, while there's understandably no calendar mode (reducing module size on chip manufactured by a relatively old and "large" 180nm technology), the associated "library" (SPL, Cube/HAL) functions use a weird day-incremental method, resulting in subtle but annoying idiosyncracies. There's no counter in RTCv2, and also in most real-world incarnations of RTCv3 counter is not supported. Only some models with RTCv3 do support both counter and calendar (again see AN4759 for this information, mostly the very newest higher-end models plus the 'WL family), and while they can be used simultaneously ("mixed mode"), this usage is restricted to synchronous prescalers ratio being powers of two, as the calendar is incremented when lowermost bits of the (now extended) SSR registers roll over to 0. However, the latter is only a minor nuissance, as the vast majority of applications use the 32.768kHz clock for RTC; as the associated extra logic (basically just the SSR extension and a selector of number of lowermost bits to increment calendar) appears to be minor compared to the rest of RTC, that's why the decision to omit it in some STM32 with RTCv3 is not quite comprehensible.

Also, quite inexplicably, ST included the relatively complex (thus presumably occupying relatively large portion of silicon) RTCv3 in the 'C0 family, which is supposed to be a low-cost "8-bit-killer", and which does not have the VBAT power domain, thus rendering RTC mostly unusable and mostly replaceable by a simple general-purpose or low-power timer.

---

1. AN4759 also does not use RTC**v**2/RTC**v**3 notation. It uses RTC2/RTC3, which in context of numbering peripherals in STM32 RMs is confusing, as it would indicate several instances of RTC within one STM32.

2. The 32-bit counter is split to two 16-bit registers, as it probably originates from some older 16-bit ST microcontrollers (similarly to some other STM32 peripherals). This results in inconvenience when reading it, as there's also no hardware mechanism for atomic readout.

Created: 17.February 2024

197. Problems with day rollover in 'F1 RTC, if SPL/Cube are used

In the primordial STM32F1xx family, RTC is implemented in a very simple manner. Basically, timekeeping consists just from a prescaler (which is to scale down the input clock to 1Hz) and a single 32-bit counter[1]. There is no calendar functionality, i.e. hardware does not count minutes-hours-days-etc. separately.

The logical and most straightforward way to tackle this situation is to use functions from the standard C library prototyped in <time.h> to calculate calendar data from the counter, and vice versa. However, ST chose a different and a slightly bizarre approach in their libraries (both the older SPL and the newer Cube/HAL): only time is kept in RTC's counter; date is maintained in one of the backup registers (which are in the same VBAT domain as RTC). When time is read out, that routine check, if time is below 24 hours, and if not, it subtracts 24 hours' worth of seconds from the RTC's counter and increments date in the backup register.

While this procedure may indeed maintain calendar for most of the applications, it also opens opportunity for various bugs. Users of ST libraries for RTC in 'F1 in the past years complained of:

- date not being incremented properly, when the mcu was powered off (with VBAT supplied by a backup battery)
- repeated 00:00:00 timestamp, when one-second periodic wakeup was used

These problems are certainly fixable and/or may have been fixed; but using the long proven methods mentioned above simply avoids this kind of problems.

---

1. The 32-bit counter is split to two 16-bit registers, as it probably originates from some older 16-bit ST microcontrollers (similarly to some other STM32 peripherals). This results in inconvenience when reading it, as there's also no hardware mechanism for atomic readout.

Created: 24.February 2024

198. RTC at midnight does not roll over, but goes to 24:00:00 and counts up

Observations in this article refer to RTCv2, although it's quite likely they apply also to RTCv3 in calendar mode.

If the RTC is set to 12-hour (AM/PM) mode (by setting RTC_CR.FMT = 1), and then the hour field in RTC_TR is set to a value above 12, the clock will run up to 23:59:59, but then upon the next second it will turn to 24:00:00 (rather than roll over to 00:00:00). The date is not incremented at this moment, either.

After that, RTC will continue to clock up to 39:59:59, and only after that it rolls over to 00:00:00. Date is incremented after that "late" rollover, too. After that, the RTC keeps running up to 12:59:59 PM, when it rolls over to 01:00:00 AM, as is normal in the 12-clock mode[1].

Of course, one would not set RTC to 12-hour mode and then set time to above 12 hours, so how could this happen?

One mechanism, how RTC_CR.FMT can get inadvertently set is through using uninitialized structs in calling Cube/HAL functions which write into RTC_CR. Another mechanism is the possible corruption of RTC registers due to VBAT brownout.

By the way, other RTC calendar fields behave similarly in that if they are inadvertently set beyond their normal range, they continue to be incremented until they roll over "naturally" due to number of bits. For example, after setting seconds to 61 they keep counting up to 79 after which the seconds "naturally" roll over to 00 [2]. Such incorrectly set calendar field can again occur due to incompletely set initialization structs when using Cube/HAL functions.

Note, that Cube/HAL functions, which read the calendar date/time, don't guarantee that the structs used will have all their fields filled in; so reading date/time into an uninitialized struct, modifying it, and writing it back may also result in writing unexpected values into some of the time/date register fields.

---

1. As an example, one of the many related threads reporting such behaviour here.

2. Don't forget, that the calendar fields don't count in binary, but in binary-coded-decimal (BCD). So, for this seconds example, the 4-bit wide RTC_TR.SU field counts from 0 to 9, and the 3-bit wide RTC_TR.ST field counts from 6 to 7 (normally, it would count from 0 to 5).
There's another, less obvious consequence of the RTC calendar fields being BCD - if such incorrectly set BCD field is used in a calculation which converts BCD to binary and adds the fields up to an "epoch"-style value, depending on how exactly that conversion is performed, the results may be wildly off with little clue to the underlying problem.

Created: 02.March 2024

199. RTC clock source can be set only once

Source of the clock which drives RTC's time and date counter[1] is selected between LSE, HSE/n and LSI in the RTCSEL field in RCC_BDCR register[2]. As with all other register fields which are in the backup-domain, before accessing it, it is necessary to disable backup-domain protection in PWR_CR.DBP.

However, RCC_BDCR.RTCSEL has an additional constraint: once it has been written to nonzero (zero being the "no RTC clock" default after backup-domain reset), it cannot be rewritten (until another backup-domain reset, either by grounding VBAT pin or through RCC_BDCR.BDRST).

There is an exception to this rule: if LSE is selected as RTC clock, in those newer STM32 families, which have Clock-Security System for LSE (LSECSS), after LSE failure is detected (LSECSSD = 1), RCC_BDCR.RTCSEL can be changed without resetting the whole backup domain.

This all is properly documented in the Reference Manual, but can be easily overlooked.

1.RTC is a typical dual-clock peripheral, having a low-frequency clock for the timekeeping functions and APB clock for register access. Consequences of this arrangement are both delays in accessing some RTC registers, and the shadow registers/lock mechanism in reading out the clock.

2.In 'L0 family, RTCSEL field is not in RCC_BDCR but in RCC_CSR register

Created: 09.March 2024

200. LSI is not a good clock to keep real time in RTC

STM32 are equipped by several oscillators, among others all STM32 contain a low-frequency low-power RC oscillator marked LSI. LSI is one of the possible input clocks to the RTC.

However, as is the case with RC oscillators, LSI is not precise. In STM32Fxxx, its frequency is given as nominally 40kHz, but with min/max range 30kHz..50kHz. That is 25% error, or 250'000ppm [1], and if an LSI running at some of the extremes would be used for timekeeping, the clock would be off by 15 minutes per hour.

LSI in STM32Lxxx and STM32Gxxx are better off - they have a nominal frequency of 32kHz and at ideal conditions (room temperature and nominal supply voltage) deviate at most 0.96kHz from it, i.e. 3% or 30'000ppm (in worst case across whole temperature and voltage range, the maximum deviation is 2.5kHz, 7.8%, 78'000ppm). While this is way better, it's still completely useless for real time keeping, as clock could be off around 2 minutes per hour.

While other RC oscillators (HSI, MSI) in STM32 do have some means of adjustment (albeit quite crude), there is no such option for LSI, so it can't be made "better" by calibrating/comparing it to a known precise clock source.

Also, LSI is in the VDD power domain, not the VBAT domain [2]; so when VDD is removed, it stops working, and the clock "freezes" even if VBAT is supplied.

So, why is LSI one of the RTC inputs? Well, in some applications, there is no need for precise timekeeping, but the application has to be very low power, spending most of the time in sleep, waking up from time to time for a burst of activity. For wakeup, a very low consumption clock source is needed, and also a very low power consumption counter driven by that clock and capable of waking up the processor in adjustable time periods. And the LSI+RTC combination is exactly this: LSI has an extremely low power consumption, and so is RTC (partially thanks to its asynchronous prescaler).

And is there any possiblity to maintain real time without having a 32.768kHz crystal or oscillator on LSE?

One option is to drive RTC from the HSE/n input, if a crystal or oscillator is present on HSE. Another option is to use some external precise clock source (e.g. 1Hz from GPS receivers) and feed it into a timer used to count real time (and calculating calendar e.g. using facilities of <time.h>). These solution don't work in VBAT mode, though.

---

1. To put this into perspective, LSE crystal oscillator can with moderate effort be made accurate to 20ppm, resulting in an error of 12 seconds per *week*.

2. Except in the rather new 'U5 family.

Created: 16.March 2024

201.STM32 with SMPS+LDO may be hard to recover, if inadvertently set to SMPS upon startup

This article has been contributed by M.Dudka.

Some STM32 (for example STM32H725 [1]) have a built-in Buck-mode (i.e. downconverting) Switched-Mode Power Supply (SMPS). It is not necessary to use it, the core can be supplied also using the built-in LDO (in RM0468, this is shown as variant 1. on *Figure 19. System supply configurations*), just like in other STM32.

After a new project is created in CubeMX, **SMPS-only is set as the initial power supply** (even if this is not the factory setting with which the chip starts up!)

If STM32H725 is connected to not use SMPS and the user forgets to change this setting in CubeMX (or deal with this situation in some other way), after the program starts to run, it switches off the LDO and the core loses its power. In such situation, it is difficult to connect to the chip using debugger (even "connect under reset" does not work).

One theoretical chance how to connect debugger is to use the short moment after the system is powered up (when the system supply is in the initial SMPS+LDO setting). Connection must be established until the problematic program starts to run and disconnects the core supply (which is a matter of milliseconds). Theoretically, reset (NRST pin) could be held externally active (low) during powerup and then debugger should be able to connect in one of the modes with externally controlled reset. I haven't tested this method, though.

Another way to solve this situation is to connect external power supply cca 1.2V to some of the VCAP pins, supplying temporarily the core, and then erase the offending program.

This is quite an unpleasant gotcha by CubeMX.

---

1. STM32H725 (and its crypto-enabled counterpart, STM32H735) in all packages does support SMPS. However, its marking does not follow the convention from other STM32 with SMPS in that it is not marked with Q suffix.

Created: 23.March 2024

Not confirmed by ST

Some issues brought up here have been discovered by myself or by other STM32 users, by experiments and experience, and have no ground in the public ST documentation.

This is nothing unusual: the STM32 are incredibly complex chips. It's far from being simple to describe the individual circuits within it in a concise and comprehensible way, especially in the zillions of ways how they can and do mutually interact. (And even if I like to publicly criticize the STM32 documentation, the truth is, that given the sheer amount and complexity of task, ST did and does a quite decent job in documenting the STM32 features. Of course, there is room for improvement, as always.)

Often, these issues involve tight timing, and so are hidden from most of the users (and sometimes ST itself) by using some of their "libraries" (Cube, SPL), or by simply not envisaging some not-that-usual (but still perfectly "legal") way of using some of the features.

Most if not all of these issues have been published at community.st.com. Sometimes ST acknowledges an issue, or sometimes they "discover" it on their own, and then they put it into the documentation, one way or another.

So, you should treat this class of issues as such, unconfirmed, or maybe just a misinterpretation of the official documentation.

And as always, I am open to corrections and comments, including (or foremost) by ST.

Created: 3.Oct 2020

STM32 gotchas

STM32 timeline and peripherals changes

We concentrate on the "general purpose" STM32, omitting 'WB/'WL (wireless).
We don't treat 'L4+ as a separate sub-family here, as it shares most characteristics with 'L4.

The original idea was to fill out the table with various modules' versions. However, this is a very hard task, and honestly, it should be ST doing it (link points to a lost "Idea" post in the STM32 forum, where several users requested this, and as most "Ideas", has been ignored by ST). Below the table there are links to gotcha articles which deal with variants of some of the peripherals. Also, a significant UART/I2C/SPI change happened at the point of introduction of 'F0/'F3.

STM32 Family	Introduced	Perk	Cortex	max clock	FLASH	GPIO	DMA	BKPSRAM	SPI	USB
F1	2007	First	M3	72 MHz	no ECC	APB	single-port ORed requests			devi (exc in 'F
L1	2009 NRND 2023	Lowlpower Midlend	M3	32 MHz	erased at 0x00 has EEPROM with ECC	AHB	single-port ORed requests			devi
F2	2010	Hilend M3	M3	120 MHz	no ECC, has OTP	AHB	2x dual-port channel select	4kB		OTG
F4	2011	Hilend M4	M4	180 MHz	no ECC, has OTP	AHB	2x dual-port channel select	4kB		OTG
F0	2012	Lowlend	M0	48 MHz	no ECC	AHB	single-port			devi
F3	2012	Analog/ Mixed	M4	72 MHz	no ECC	AHB	single-port ORed requests			devi
L0	2013	Lowlpower Lowlend	M0+	32 MHz	erased at 0x00 has ECC, has EEPROM	IOPORT	single-port			devi
F7	2014	Hilend M7	M7	216 MHz	no ECC, has OTP	AHB	2x dual-port channel select	4kB		OTG (HS
L4	2015	Lowlpower Hilend	M4	80 MHz	has ECC, has OTP	AHB	single-port			devi end) OTG high
H7	2016	Hilend M7(+M4)	M7(+M4)	480 MHz	has ECC	AHB4	4x(mixed) DMAMUX	4kB		OTG
G0	2018	Lowlend	M0+	64 MHz	has ECC, has OTP	IOPORT	single-port DMAMUX			devi only
G4	2019	Analog/ Mixed	M4	170 MHz	has ECC, has OTP	AHB	single-port DMAMUX			devi
L5	2019	Lowlpower Secure Midlend	M33	110 MHz	has ECC, has OTP,	AHB	2x single-port DMAMUX			devi
U5	2021	Lowlpower Secure Hilend	M33	160 MHz	has ECC, has OTP	AHB	GPDMA+LPDMA			OTG
C0	2023	Lowestlend	M0+	48 MHz	no ECC, has OTP	IOPORT	single-port (3ch only) DMAMUX			---
H5	2023	Secure Hilend	M33	250 MHz						OTG

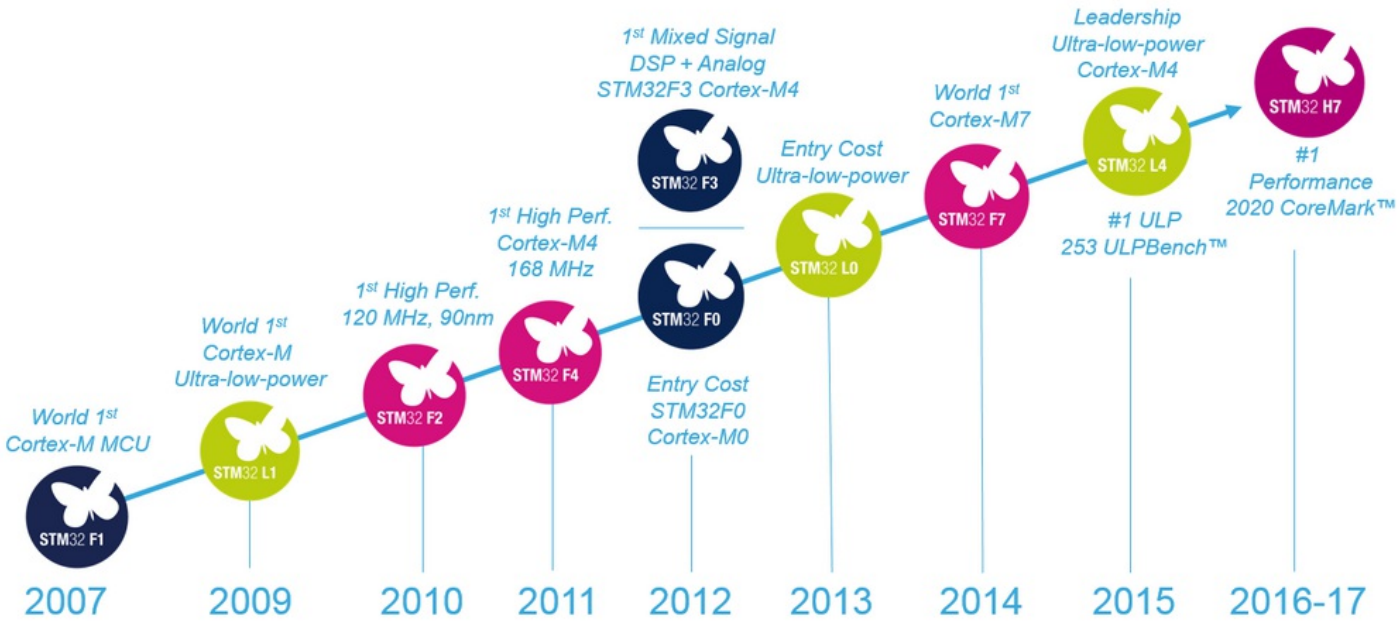
FLASH variants across families discussed here. **RTC** variant across families discussed here.

GPIO on AHB is faster, but GPIO on APB in 'F1 allows bit-banding.
IOPORT is a dedicated port on the Cortex-M0+ processor, allowing faster access from processor to GPIO at the cost of preventing DMA from accessing it.
In 'H7, GPIO is on a secondary bus-matrix, removed from processor and DMAs (except BDMA) through two busmatrices and respective bridges, making "manual toggling" of GPIO surprisingly slow.

The single-port **DMA** is described in AN2548, dual-port DMA in AN4031.
The DMA column also indicates, how requests from peripherals are steered to DMA: the simplest approach is that requests from several peripherals to an individual DMA channel are simply ORed together; in more

advanced models there is a request-selecting multiplexer controlled by bits in DMA registers; and in the newest models there is a standalone DMAMUX unit, effectively implementing a full matrix between requests from various peripherals and the set of available DMA channels (plus some additional features).
In 'H7, there's one advanced MDMA on the AXIM bus matrix, two dual-port DMA and one single-port DMA (as BDMA).
In 'U5, the DMAs are very different from rest of the STM32. As far as requests go, GPDMA and LPDMA both contain an embedded unit similar to DMAMUX. GPDMA is dual-port, LPDMA is single-port.

Folowing "10th anniversary" infographics from 2017 (i.e. 'G0/'G4/'L5/'U5/'C0/'H5 missing) linked from community.st.com:



Created: 3.Oct 2020
Modified 12.Sep 2021, 9.Apr 2022, 24.Apr 2022, 2.Sep 2022
'L5/'U5 added 17.Dec 2022
'L1 NRND and 'C0 added 12.Feb 2023
'H5 added 26.March 2023
link to ST's infographics fixed for new forum 24.June 2023
minor changes 10.February 2024