The ultimate 16550 UART character printing routine

Jesús Arias

November 4, 2021

1 Introduction

The classic 16550 UART from National Semiconductor was an usual peripheral in PCs before being replaced by USB and it is still in use in many microcontrollers, in particular in those made by NXP. Probably its popularity was the deciding factor for including such an aging peripheral in modern microcontrollers. We must remark that this UART was in fact an enhancement of the old 8250, another simpler UART made by Intel for its 8080 CPU, differing only in the addition of 16-byte FIFOs for both the transmitter and the receiver. The receiver FIFO proved to be very useful in reducing the interrupt overhead in old MS-DOS computers connected to "high-speed" (56Kbit/s) modems, one of the early cases where a hardware hack allowed to hide the crappy details of the software to the user: The real problem was the very long interrupt latency of the MS-DOS drivers, but instead of improving the software the serial port was replaced with a new UART with FIFOs and this allowed a reliable communication with the modem.

So, the receiver FIFO is really very useful and any modern software will enable it, but what's about the transmitter FIFO? It is also enabled along the receiver FIFO but it is rarely used at all due to the stupid meaning of its flags. Instead of a FIFO-full flag to check before transmitting what we really got is a FIFO-empty flag (THRE bit in LSR register). Therefore the simple character printing routine, whose code is listed next, waits until the TX FIFO is completely empty before sending the next character instead of queuing it if there is still a free space in the FIFO.

```
void U0putch(int d)
{
    while(!(U0LSR&0x20));
    U0THR=d;
}
```

In this code the function parameter is really 8-bit wide in spite of using a 32-bit variable because the compiler generates a more compact code if variables fit the native width of the CPU registers (we are compiling this code for a 32-bit ARM cortex-M core). The UART registers are all 8-bit wide and only the 8 least significant bits of variables are actually read or written. The "while" loop waits until the THRE flag (bit 5 of LSR) is one, and then writes the character into the THR register. By calling this routine the transmitter FIFO never holds more than a single character.

What I'm pretending here is to make a smarter use of the transmitter FIFO in spite of its flags. The solution isn't yet perfect, because there isn't a reliable way to know if the FIFO is full, but it can still make use of the FIFO under many practical situations.

2 The transmitter flags

The Line Status Register (LSR) contains two bits related to the transmitter. These are:

- Bit 5. Transmitter Holding Register Empty (THRE). This bit is one if the TX FIFO is empty.
- Bit 6. Transmitter Empty (TEMT). This bit is one if the TX FIFO is empty and the output shift register is also empty. This means that all pending data was completely transmitted.

I want to notice that the shift register of the transmitter provides one additional byte of storage and, therefore, the UART can hold up to 17 bytes awaiting to be transmitted, one in the shift register and 16 in the FIFO.

From our software point of view it is useful to summarize the meaning of these flags in the following table:

TEMT LSR[6]	THRE LSR[5]	max # of bytes to write	
		FIFO OFF	FIFO ON
1	1	2	17
0	1	1	16
0	0	0	0 to 15
1	0	Never happens	

The ugly detail here is the uncertainty about the filling of the FIFO. If THRE is zero the FIFO can be still not full and it could store more additional bytes to transmit, but there is no way to know if this is true. What we really know is that we can transmit up to 16 bytes without waiting if THRE is one or 17 bytes if TEMT is also one.

3 The routine

Here is the code:

```
typedef signed int   s32;
void U0putch(int d)
{
    s32 nf;
    //static u8 nfifo;
    #define nfifo U0SCR
    if ((((s32)U0LSR)<<(31-6))<0) nf=16;
    else {
        nf=nfifo-1;
        if (nf<0) {
            while ((((s32)U0LSR)<<(31-5))>=0);
            nf=15;
        }
    }
    U0THR=d;
    nfifo=nf;
}
```

The main idea behind this code is to keep track of how many characters have been sent to the UART since the last time the flags were set, because we know we can send up to 16 characters when THRE is set, or one character more if TEMT is also set. The static variable “nfifo” holds how many more characters can be sent before waiting again for THRE. In this example this variable is in fact the scratch register of the UART (SCR), a register without any usage by the hardware that can store one byte. In this way no RAM is needed for the “nfifo” variable.

There are also some less clear optimizations in the source code. For instance, the static “nfifo” variable is copied to the local variable “nf” because local variables are usually mapped to registers instead of memory and the resulting code is better than using the static variable directly. This is specially true for the UART registers because they are declared as “volatile”, meaning they have to be read every time they appear in an expression. Also, the flags of U0LSR are sifted from their positions to bit 31 in order to test them as sign bits. This saves the loading of a mask value to a register prior to an AND/TST instruction (as is the case of the simple routine).

The resulting code after compilation with the “-Os” option (optimize code for size) is listed next. The target CPU is an ARM cortex-M (16-bit Thumb-2 code).

```
0000036e <U0putch>:
 36e:    4a07          ldr    r2, [pc, #28]    ; (38c)
 370:    2310          mov    r3, #16
 372:    6951          ldr    r1, [r2, #20]    ; (U0LSR)
 374:    0649          lsl    r1, r1, #25
 376:    d406          bmi    386
 378:    69d3          ldr    r3, [r2, #28]    ; (U0SCR)
 37a:    3b01          sub    r3, #1
 37c:    d503          bpl    386
 37e:    6953          ldr    r3, [r2, #20]    ; (U0LSR)
 380:    069b          lsl    r3, r3, #26
 382:    d5fc          bpl    37e
 384:    230f          mov    r3, #15
 386:    6010          str    r0, [r2, #0]     ; (U0THR)
 388:    61d3          str    r3, [r2, #28]    ; (U0SCR)
 38a:    4770          bx     lr
 38c:    40008000      .word  0x40008000       ; UART #0 Base address
```