04: Timer and Counter, UART

Microcontrollers

Stefan Huber

`www.sthu.org`

Dept. for Information Technologies and Digitalisation
FH Salzburg

Winter 2024

Timer and Counter

Counters and timers

Counters are a basic element of every microcontroller:

- Can count edges of an external pin
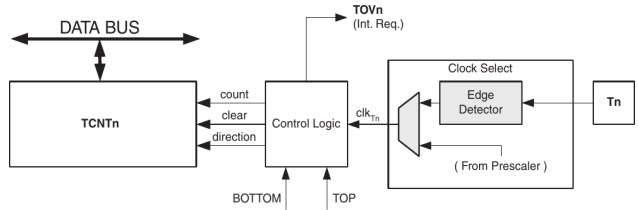- Becomes a timer when counting clock ticks.[1]

Figure: [ATmega32, p. 70, fig. 28]

Interrupts

A counter/timer provides different interrupt sources, e.g.,

- overflow
- match against a compare register

---

[1] Actually, our definitions of time units – like a second – have always been based on periodic events, like the earth's motion around the sun. Since 1967 we define a second such that a certain radiation frequency of the caesium-133 atom is 9 192 631 770 Hz. Einstein had to leave the grounds of absolute time in this theory of relativity. He would only define relative simultaneity of remote events to an observer when light rays arrive simultaneously.

Timer applications

Timers have various applications:

- ▶ Measuring time or annotating events with time stamps.
- ▶ Waveform generation, in particular pulse-width generation (PWM) for digital-analog conversion (DAC).
- ▶ Implementing periodic tasks like for
    - ▶ time-discrete signal processing (closed-loop control algorithms, reading sensors, setting actors)
    - ▶ reoccuring jobs (logging, updating displays)
    - ▶ providing a system clock
    - ▶ preemptive multi-tasking scheduling

Counter resolution

A counter with a resolution of n bits can have values within $[0, 2^n - 1]$. Incrementing $2^n - 1$ causes an overflow to 0.

ATmega32 timer/counters

- ▶ TCNT0 has 8-bit resolution
- ▶ TCNT1 has 16-bit resolution
- ▶ TCNT2 has 8-bit resolution

Counters with high resolution

When the counter resolution exceeds the word size then we need more than one instruction to read out the counter register.

- ▶ TCNT1 has a resolution of 16 bit, but ATmega32 has 8-bit registers.

Race condition

Consider a counter tick during the two 8-bit reads. Example: 0x08ff turns 0x0900:

- ▶ If we read the low byte first then we may obtain 0x09ff.
- ▶ If we read the high byte first then we may obtain 0x0800.

Solution of the ATmega32:[2]

- ▶ There is a temporary 8-bit high byte register for 16-bit register access.
- ▶ The low byte must be read first; the high byte is simultaneously put into the temporary register. Then, when reading the high byte the temporary buffer is read. Vice versa for a write.
- ▶ The C compiler takes care for that.

---

[2] See [ATmega32, p. 111 and p. 89.] for details. The temporary register is shared among 16-bit register accesses, i.e., for TCNT1, ICR1 and OCR1A/B (only write). ▶ ◀ ⨤ ▶ ⨤|⨤ ⟳ ⊙ ⟳

System clock The timer progresses with the system clock.

Prescaler The timer still uses the system clock, but a prescaler is applied to scale up the clock period in order to change the effective time range.

Ext. pulse Pulses on an external pin are counted. The pin is sampled, so the pulse must be at least a system clock long. This mode is also known as pulse accumulator.

Ext. crystal An external crystal is connected to two pins. The ATmega32' oscillator is optimized to a 32.768 kHz quartz. The timer is independent of the CPU clock. Therefore this mode is also called asynchronous mode and can be used to implement a real-time clock (RTC).
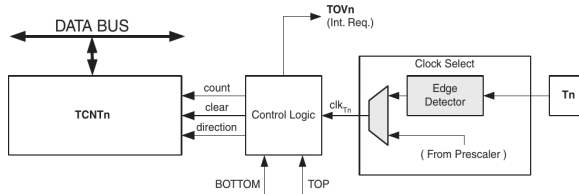

Figure: [ATmega32, p. 70, fig. 28]

Prescaler

A prescaler scales down the frequency of the system clock by powers of two.

A counter is a cascade of frequency-halving flip-flops.

- The clock is c_0. The period of c_{i+1} is double of the period of c_i.
- A prescaler is simply a counter again. The prescaler is fed by the system clock and the clock of the actual timer/counter is fed by one of the c_i.

This prescaler, as a kind of "pre-counter", can be reset.

c_3	c_2	c_1	c_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Prescaler

A prescaler extends the timer range:

▶ An 8-bit timer at a clock of 4 MHz overflows in $256 \cdot \frac{1}{4}$ µs $= 64$ µs.

▶ Using a prescaler of 1024, the timer overflows in 65.536 ms.

But a prescaler makes granularity coarser:

▶ At a clock of 4 MHz the granularity is 250 ns with no precaler.

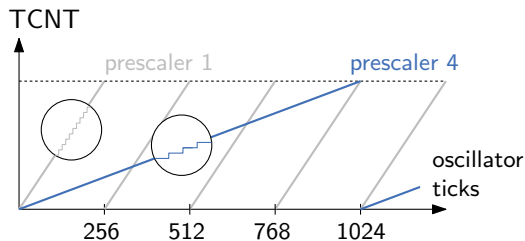▶ With prescaler 1024 the granularity is 256 µs.

Figure: 8-bit timer example. Prescaler 4 extends time range and makes granularity coarser.

General rule

Choose the smallest prescaler such that required time range is met. If required granularity is not met then reduce prescaler and count overflows in software.

Example: Periodic timer interrupt

We would like to have a periodic timer interrupt, say every 1024 µs.

- ▶ We use the interrupt upon timer overflow, hence the timer needs to overflow every 1024 µs.
- ▶ Let us use timer 2, which is an 8-bit timer, so it overflows after 256 timer ticks. Hence, a single timer tick needs to be 4 µs long. We have a clock rate of 8 MHz, so 0.125 µs per CPU clock. Hence, we need a prescaler of $^{4\,µs}/_{0.125\,µs} = 32$.

```
ISR (TIMER2_OVF_vect) {
    /* TCNT2 raised an overflow interrupt every 1024us when F_CPU is 8 MHz. */
}

void initialize() {
    /* Timer/counter control register for timer/counter 2. (p125)
     * Set clock select to internal clock with prescaler 32 (p127). */
    TCCR2 = 0x03;
    /* Set TOIE2 bit (timer overflow interupt enable for timer/counter 2) (p82) */
    TIMSK |= (1 << TOIE2);
    /* Set global interrupt enable bit. */
    sei();
}
```

Input capture

Input capture allows to timestamp events.

- ▶ At an event the timer register is copied to a input capture register (ICR) and an interrupt is raised.
- ▶ ATmega32 can use two kind of events:
  - ▶ The input capture pin changes its level
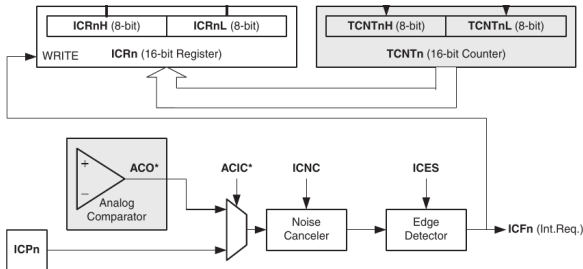 - ▶ The analog comparator changes its output

Figure: Input capture unit of the ATmega32 16-bit timer 1. [ATmega32, p. 93]

Input capture accuracy

Of course we want the time stamp to be as *accurate* as possible:

- ▶ We want a fine prescaler.
- ▶ But the accuracy also depends on the digital input sampling
    - ▶ Metastability is avoided by synchronizer latches, which add a signal delay.[3].
    - ▶ If noise cancellation is enabled then the signal is delayed further.[4]

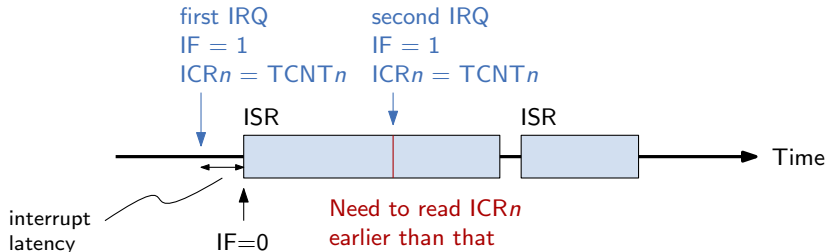By compensating the above delay, we can measure time at the precision of the system clock.

---

[3] ATmega32, p. 51.
[4] ATmega32, p. 94.

Input capture race condition

Assume that the interrupt flag (IF) is set by an input capture IRQ.

▶ When the ISR is called then IF is cleared. If a second input capture IRQ happens *after that* then IF is set again: The ISR will be called a second time.

▶ If ICRn in the first ISR is read after the second IRQ then we read the updated ICRn in both ISRs! The original value is lost.

▶ Hence, read the ICRn as early as possible in the ISR. At least during the interrupt latency, however, we have to take this race condition into account.

▶ Nested interrupts (non-blocking ISRs) only increase complexity further.

UART

Communication interfaces

Serial communication has many applications:

- ▶ Peripherals
- ▶ Between microcontrollers
- ▶ With a PC, e.g., for debugging

Two major kinds of serial communication:

Asynchronous Each communication partner has its own clock and a synchronization mechanism is required.
Example: SCI (UART)

Synchronuous A common clock wire defines when the levels of the signal are valid. Synchronous communication enables higher bit rates, is simpler, requires an additional wire and a master clock.
Example: SPI

UART

This is the interface we know from the "standard RS-232 serial cables" with D-SUB9 plugs every PC and laptop used to have a long time ago. Now we use USB-serial converters...

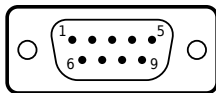
Figure: From Wikipedia. By Mayayu, under CC BY-SA license.

An Universal Asynchronous serial Receiver and Transmitter (UART) unit provides the Serial Communication interface (SCI):

▶ Asynchronous
▶ Two wires (RX, TX), full duplex

UART frame format configuration

- Low-level start bit, then 5 to 9 data bits, then no, even or odd parity bit, then 1 or 2 stop bits.
- The parity bit adds redundancy for error detection: For even parity the parity bit is the xor of all data bits. The odd parity is the even parity inverted.[5]

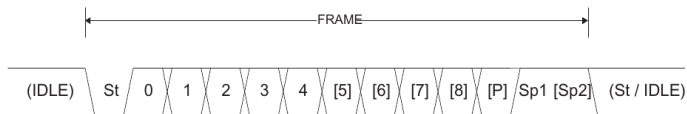Figure: See [ATmega32, p. 144, fig. 72].

A common short-hand notation for the frame format:

- Example: 8N1 means 8 data bits, no parity, 1 stop bit.
- In general: $D\{E|O|N\}S$, where E, O, N means even, odd or no parity bit, D is the number of data bits and S is the number of stop bits.

---

[5] The xor over the data bits including the parity bit is even (resp. odd) for even (resp. odd) parity.

Baud rate generation

Asynchronous communication: Sender and receiver must set the same (sufficiently similar) baud rate.[6]

The ATmega32 has a normal mode and a double-speed mode. In either case the UBRR register is used to set the resulting baud rate:

Operating Mode	Equation for Calculating Baud Rate[(1)]
Asynchronous Normal Mode (U2X = 0)	$BAUD = \dfrac{f_{OSC}}{16(\boldsymbol{UBRR} + 1)}$
Asynchronous Double Speed Mode (U2X = 1)	$BAUD = \dfrac{f_{OSC}}{8(\boldsymbol{UBRR} + 1)}$

Figure: See [ATmega32, p. 143, fig. 69].

UBRR is a 12-bit value.[7] It is split up into a high-byte UBRRH and a low-byte UBRRL.

---

[6] In our case a symbol is a bit, hence baud rate equals bit rate.
[7] See [ATmega32, p. 143].

To initialize the USART of the ATmega32, we have to[8]

- ▶ Set the UCSRB register to enable receiver and/or transmitter and interrupts.
- ▶ Set the UCSRC register to choose asynchronous mode and the frame format.
- ▶ UCSRA contains flags and the U2X bit to set double speed mode.

```c
void uart_init() {
    /* UBRR of 51 gives a baud rate of 8 MHz / (16*(51 + 1)) = 9615.4 Hz.
     * This is 0.16% off 9600 Hz, which is fine. */
    UBRRL = 51;
    UBRRH = 0;
    /* Asynchronous mode, 8N1 frame format (p164). */
    UCSRC = (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0);
    /* Enable receiver and transmitter (p161) */
    UCSRB = (1 << RXEN) | (1 << TXEN);
    /* For interrupt-based communication, enable interrupts in UCSRB. */
}
```

---

[8] ATmega32, p. 146.

A byte is transmitted by writing to the UDR register.[9]

```c
void uart_send(uint8_t data) {
    /* Busy wait until transmit buffer is empty, i.e., the Data Register Empty
     * (UDRE) flag is set so the next frame can be sent. */
    while (!(UCSRA & (1 << UDRE)));

    /* Writing to UDR (buffer) sends the data. */
    UDR = data;
}
```

Instead of busy wait (polling):

▶ There is also an interrupt that signals when UDRE is set (UDRIE flag).

---

9 ATmega32, p. 147.

A byte is received by reading from the UDR register.[10]

```
1 uint8_t uart_recv() {
2     /* Busy wait until we received data, i.e., the Receive Complete (RXC)
3      * flag is set. */
4     while (!(UCSRA & (1 << RXC)));
5
6     /* Read the received data from the buffer. */
7     return UDR;
8 }
```

Instead of busy wait (polling):
▶ There is also an interrupt that signals when RXC is set (RXCIE flag).

---

10 ATmega32, p. 150.

Concurrent bidirectional communication

The code listings for `uart_recv()` and `uart_send()` is blocking I/O on a byte level. Instead we want this:

```
// Should be non-blocking, i.e., we do not want to miss characters we receive
// at the (looong) time of sending.
serial_puts("hello world");
serial_printf("Pi is %f\n", 3.14);
```

How can we implement this?

Concurrent bidirectional communication

The code listings for `uart_recv()` and `uart_send()` is blocking I/O on a byte level. Instead we want this:

```
1 // Should be non-blocking, i.e., we do not want to miss characters we receive
2 // at the (looong) time of sending.
3 serial_puts("hello world");
4 serial_printf("Pi is %f\n", 3.14);
```

How can we implement this?

▶ We need concurrent handling of sending and receiving, hence we need interrupts.

▶ We need to decouple the ISR logic from the `serial_puts()` logic, i.e., have a communication buffer between those two. (Similar to inter-process communication.)
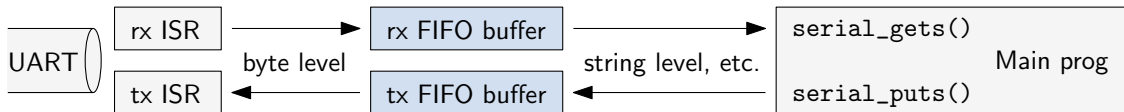
Circular buffer

The predominant data structure for this communication buffer is a circular buffer (aka. ring buffer).

- A fixed-size array is the storage.[11] The access semantics is FIFO (a queue).
- A write pointer and a read pointer each progress in the storage and wrap around when reaching the end. This allows for indefinite progress (in modulo arithmetic), i.e., the array forms a circle.
- The pointers must not overtake each other:
  - If both pointers point to the same index then the buffer is considered empty.
  - If the write pointer is one position behind the read pointer then the buffer is considered full.

---
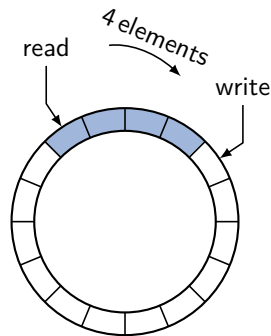
11 No dynamic memory allocation is necessary.

[ASCII]   *Coded Character Set - 7-Bit American National Standard Code for Information Interchange*. Standard ANSI X3.4. American National Standards Institute, 1986. URL: http://sliderule.mraiow.com/w/images/7/73/ASCII.pdf.

[ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash*. Atmel Corporation. Feb. 2011.

Prescaler: Changing the timer value

Assume a prescaler P greater than 1. Changing the timer value can be done in different ways:

On the fly We simply write to the timer register. This write is probably[12] not coinciding with a timer tick. Hence, the new timer value may last for anything between 1 to P system ticks instead of the expected P ticks, which can be problematic.

With reset We stop the timer, set the timer register, reset the prescaler[13] and start the timer. The new timer value lasts for P ticks *after* it had been set to the new value.
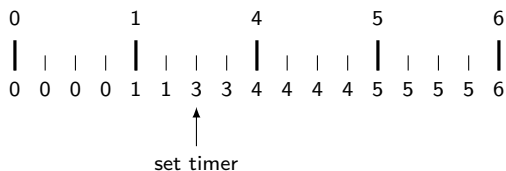
(a) Changing timer on the fly
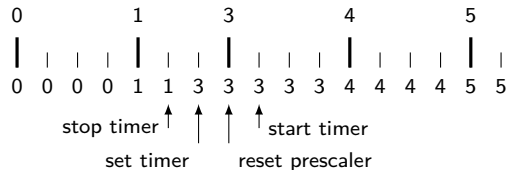
(b) Changing timer with prescaler reset

Figure: A timer with prescale $P = 4$ and the two approaches to change the timer value to 3.

---

12 With probability $1/P$ we hit a timer tick and with $1 - 1/P$ we miss it.

13 ATmega32, p. 84.

ASCII[14] is the predominant standard for encoding characters into bytes.

- It dates back to 1968, but its latest version is of 1986.
- It is a 7 bit code, but there are many 8-bit *extended ASCII* encodings (codepages).
- The mess with different codepages has come to an end thanks to the UTF-8 encoding, which contains 7-bit ASCII at its lower end.
- In C and many other programming languages, ASCII is used for string encoding. More modern languages tend to support UTF-8 for strings and even source code.

---

14 *Coded Character Set - 7-Bit American National Standard Code for Information Interchange.* Standard ANSI X3.4. American National Standards Institute, 1986. URL: http://sliderule.mraiow.com/w/images/7/73/ASCII.pdf.

This is an excerpt from RFC 20.

- See also linux man page ASCII(7).

Note that lower and upper case characters differ only in b6.

It also contains 32 control characters:

- For hardware, e.g., printers and terminals. For software, e.g., terminal emulators or TUIs.
- Typically we use escaping sequences with backslash to express them in source code.
- Examples: new line \n, carriage return \r, backspace \b, bell \a, form feed \f.

```
       B  \ b7 ----------->| 0   | 0   | 0   | 0   | 1   | 1   | 1   | 1   |
       I  \ b6 ---------->| 0   | 0   | 1   | 1   | 0   | 0   | 1   | 1   |
       T  \  b5 -------->| 0   | 1   | 0   | 1   | 0   | 1   | 0   | 1   |
        S               |-------------------------------------------------|
              COLUMN->| 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
|b4 |b3 |b2 |b1 | ROW |     |     |     |     |     |     |     |     |
+---+---+---+---+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 | 0 | 0 | 0 | 0   | NUL | DLE | SP  | 0   | @   | P   | `   | p   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 0 | 1 | 1   | SOH | DC1 | !   | 1   | A   | Q   | a   | q   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 0 | 2   | STX | DC2 | "   | 2   | B   | R   | b   | r   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 1 | 3   | ETX | DC3 | #   | 3   | C   | S   | c   | s   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 0 | 0 | 4   | EOT | DC4 | $   | 4   | D   | T   | d   | t   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 0 | 1 | 5   | ENQ | NAK | %   | 5   | E   | U   | e   | u   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 6   | ACK | SYN | &   | 6   | F   | V   | f   | v   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 1 | 7   | BEL | ETB | '   | 7   | G   | W   | g   | w   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0 | 0 | 0 | 8   | BS  | CAN | (   | 8   | H   | X   | h   | x   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0 | 0 | 1 | 9   | HT  | EM  | )   | 9   | I   | Y   | i   | y   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0 | 1 | 0 | 10  | LF  | SUB | *   | :   | J   | Z   | j   | z   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 0 | 1 | 1 | 11  | VT  | ESC | +   | ;   | K   | [   | k   | {   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 0 | 0 | 12  | FF  | FS  | ,   | <   | L   | \   | l   | |   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 0 | 1 | 13  | CR  | GS  | -   | =   | M   | ]   | m   | }   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 0 | 14  | SO  | RS  | .   | >   | N   | ^   | n   | ~   |
|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 1 | 1 | 1 | 15  | SI  | US  | /   | ?   | O   | _   | o   | DEL |
```