CG1112 Engineering Principles and Practices
Tutorial 3
Timers, USART, Revision

Question 1

Sections 19 to 22 of the Atmega328P Data Sheet are about the timers, which are devices on the 328P that trigger interrupts at fixed intervals.

Consult these sections to answer the following:

a. Give two examples of uses for the timers.

- Configure to trigger once per second then write a soft real-time clock. E.g. at https://bitbucket.org/ctank/smarttimer/src/master/
- Used in operating systems to switch between tasks in a multitasking environment, to simulate multiple processes running on a single CPU.

b. Timers 0 and 2 are 8 bit timers. What does the term "8-bit timer" mean in terms of interrupt triggering intervals?

- The main counter TCNT0 / TCNT2 only counts from 0 to 255. Based on table 19-10 in the datasheet, the largest timer resolution is CLK_{IO} / 1024. On the Arduino CLK_{IO} is equal to the system clock which is 16 MHz, and thus the largest resolution is 16000000/1024 = 15,625 cycles between incremenets of TCNT0 or TCNT1. This works out to 0.000064 seconds between increments. The largest interval for Timer 0 and Timer 2 is therefore 0.000064 x 255 = 0.01632 seconds.

c. Timer 1 is a 16 bit timer. List one advantage and one disadvantage of a 16 bit timer over an 8-bit timer.

Advantage:

Timer 1 counter TCNT1 is 16 bits long, having a maximum value of 65535. The largest resolution is again 0.000064 seconds (Table 20-7), giving us a maximum interval of 65535 x 0.000064 = 4.1924 seconds.

It is thus possible to set intervals of over a second.

Disadvantage:

The Timer 1 registers TCNT1, OCR1A and OCR1B are all twice as large as those in Timers 0 and 2, requiring more programming to set them up (since the 328P can only set the registers up 8 bits at a time), and occupying more space on the chip.

Question 2

We are given the following contents of TCCR1A and TCCR1B, and OCR1A holds the value of 247. Given that TCNT1 is initially 0, that we are operating Timer 1 in CTC mode, and that all interrupt flags in TIMSK1 and SREG are properly set:

```
TCCR1A = 0b01010000;
TCCR1B = 0b00001100;
```

i) What is the interval between which TIMER1_COMPA_vect is triggered?

The prescalar is 0b100 which is 256. At 16MHz, the period is 256/16000000 = 16 us. The period therefore is (247 + 1) * 16 us = 3.968 ms.

ii) Sketch the waveform on OC1A.

There is a match approx. once per 4 ms, and the COM1A0 and COM1A1 bits are 0b01, which is a "toggle on match". Hence we get a square wave of period of about 8 ms.

Question 3

We want to trigger TIMER0_COMPA_vect on Timer 0 once every 650 us (microseconds) in CTC Mode. Complete the following table. Round up any cases where V is not an integer (e.g. 34.3 -> 35, 38.8 -> 39, etc). Assume that the Atmega328P is being clocked at 20 MHz and that CLKIO is identical to the Atmega's frequency.

Note: If the V value cannot be loaded into OCR0A, write "N/A" in the Actual Period column.

Note: To get the period we take P / 20000000, where P is the prescalar value.

Prescalar	Period in us	V (no rounding)	V (rounding)	Actual period
CLKIO/1	0.05	N/A	N/A	N/A
CLKIO/8	0.4	N/A	N/A	N/A
CLKIO/64	3.2	203.125	204	652.8 us
CLKIO/256	12.8	50.78125	51	652.8 us
CLKIO/1024	51.2	12.6953125	13	665.6

Based on the table above comment on which value of V gives the best accuracy. Why do you think this is so?

Prescalar of 64 and 256 give the same accuracy while prescalar of 1024 gives very poor accuracvy.

In general the smaller a prescalar is, the more accurate the timing would be. Of course this will also depend on the rounding error.

Question 4

In this question we explore the UART link-layer protocol:

a. Show how odd and even parity bits works, and how they can be used to detect errors.
A "parity bit" is an extra bit that is set/cleared so that the total number of "1" bits sent is either odd or even.

Odd parity: Parity bit is set so that the total number of 1 bits is odd.
Even parity: Parity bit is set so that the total number of 1 bits is even.

b. Explain why error detection with parity bits is not reliable, and what engineers can do to overcome/improve this lack of reliability.

Parity checking can only detect an odd number of parity errors. E.g. in even parity with the following bit pattern:

0b00101100

The parity here would be 1:

0b00101100-1

However if we received instead:

0b00100110-1

Notice that the parity is still correct: We still have an even number of 1's. Engineers need to use higher layers of error checking like checksums and cyclic redundancy checks (CRCs) to detect errors.

c. Sketch the frames for the following pieces of 8-bit data in 8N1, 8E1, 7O1 and 7E1 frame formats. Where you are using 7-bit frames, discard the most significant (left most) bit.

0b10110001
0b01010011

St – Start bit: High to low transition
S – Stop bit: A high bit

8N1: St-10110001-S
8E1: St-10110001-0-S
7O1: St-0110001-0-S
7E1: St-0110001-1-S

8N1: St-01010011-S
8E1: St-01010011-0-S
7O1: St-1010011-1-S
7E1: St-1010011-0-S

Question 5

We extend the idea of parity so that we set parities not just within a byte, but across bytes, effectively producing a parity byte. The table below shows this idea, using odd parity (0 is considered to have an even number of '1' bits):

Data bits ->	b7	b6	b5	b4	b3	b2	b1	b0	Parity (within byte)
Byte 1	1	0	1	1	0	1	0	0	**1**
Byte 2	0	1	0	0	0	1	0	1	**0**
Byte 3	0	1	1	1	0	0	0	0	**0**
Parity (across bytes)	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**

This scheme can be used to correct single bit errors. Show how.

Note below that the row parity for byte 1 is wrong (there's an even number of bits in an odd-parity system). The number of "1" bits in column b4 is also wrong. This tells us that bit 4 of byte 1 (the highlighted square) is wrong and should be flipped to a 1

Data bits ->	b7	b6	b5	b4	b3	b2	b1	b0	Parity (within byte)
Byte 1	1	0	1	0	0	1	0	0	**1**
Byte 2	0	1	0	0	0	1	0	1	**0**
Byte 3	0	1	1	1	0	0	0	0	**0**
Parity (across bytes)	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**

You can actually detect more than one bit error, as long as there is no other error in the same column and row.

Question 3

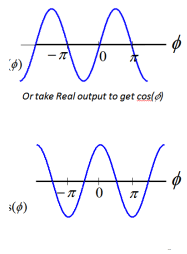
How long does it take to transmit all 128 bytes in a buffer at 9600 bps? What implications does this have on how fast your program can write to your buffers?

128 bytes = 1024 bits. It would take 0.11 seconds to send over 9600 bps. If data is written too quickly into the buffers, either the new data is lost or data that hasn't been sent yet is overwritten.

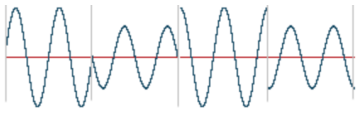
SAMPLE MIDTERM QUESTIONS

1. We have a communication system with a baud rate of 9600, communicated as a sine wave that can be transmitted at four voltage levels (1.25v, 2.5v, 3.75v, 5v) and four phase shift angles (0, 90 degrees, 180 degrees, 270 degrees). What is the maximum bit rate (in bits per second or bps) achievable on this system? The diagram below shows examples of signals at a 0 degree and 90 degree phase shift, and one example of some data being transmitted:

0 degrees and 90 degree phase shift examples:



Example data transmission



a. 9600 bps
b. 2400 bps
c. **+38400 bps**
d. 19200 bps
e. 4800 bps

Commented [CT1]: Each "symbol" can be one of four voltage levels and one of four phase shifts, giving 16 combinations. This can represent 4 bits. Thus bit rate = 4 x 9600 = 38400.

2. There is only one UART port on the ATMega328P, and one way around this limitation is to use a "soft serial" solution – software that simulates a UART connection. In the following code we use the Arduino timers to simulate a serial connection. The global variable *byteToSend* contain the byte to be sent, and *byteReceived* contains the byte received from the sender. We assume that this code is used to send a single byte between two 16 MHz Arduino UNOs. We further assume that all C statements take negligible time to execute.

```
// Global variable containing the byte to send. This byte is set elsewhere
// in the code
unsigned char byteToSend;
ISR(TIMER1_COMPA_vect) {
	static int bitsSent = 0;

	bitsSent = (bitsSent + 1) % 9;

	if(bitsSent) {
		if(byteToSend & 0x1)
			digitalWrite(3, HIGH);
		else
			digitalWrite(3, LOW);

		byteToSend = byteToSend >> 1;
	}
	else {
			// Pull line high
			digitalWrite(3, HIGH);

			// Turn off Timer 1
			TCCR1B = 0x0;
		}
}

char byteReceived;

ISR(TIMER2_COMPA_vect) {
	static int bitsReceived = 0;
	static char bitToWrite = 0x80;

	bitsReceived = (bitsReceived + 1) % 9;

	if(bitsReceived) {
		if(digitalRead(4) == 0)
			byteReceived &= ~bitToWrite;
		else
			byteReceived |= bitToWrite;

		bitToWrite = bitToWrite >> 1;
	}
	else
		TCCR2B = 0x0; // Turn off Timer 2.
}
```

```
// Send a byte
void sendByte(unsigned char byte) {

       byteToSend = byte;

       // Pull line low
       digitalWrite(3, LOW);
       delayMicroseconds(500); // Wait 0.5 ms

       /*
          Code to set up and start Timer 1 with a resolution of
          4 microseconds and OCR1A = 249 omitted for brevity
       */
}

// Receive a byte. Called by code expecting a byte
// to come in. Assume that there is some suitable mechanism
// that prevents byteReceived from being read until all
// bits are read in.
void receiveByte() {

       while(digitalRead(4));
       delay(1); // Wait 1 ms

       /*
          Code to set up and start Timer 2 with a resolution of
          4 microseconds and OCR2A = 249 omitted for brevity
       */
}
```

We make the following statements about the code. We assume that all C statements take negligible time to execute.

i. The bits sent are sampled at the end of each bit period.
ii. The bit rate of this soft serial solution is 1000 bps.
iii. The bits sent are received in reverse order.
iv. Pin 3 is used for TX and 4 for RX lines.
v. The GND line between two Arduinos do not need to be connected.

Which states is/are true?

a. i., iii., and iv. only are true.
b. ii., iii. and v. only are true.
c. i., iv. and v. only are true.
d. i., ii. and iii. only are true.
e. +ii., iii and iv. only are true.

Commented [CT2]: False. The sender starts sending 0.5 ms after pulling line 3 low and sends 1 bit every ms, and the receiver starts sample 1 ms later, and then 1 ms thereafter. So it samples in the middle of the bit.

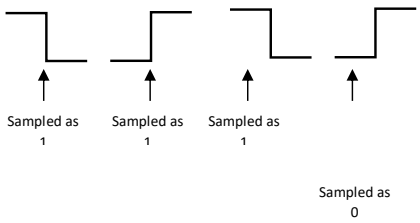
Commented [CT3]: True because the timers are set to trigger the ISRs every ms.

Commented [CT4]: True, receive is setting bits from left to right but send is sending bits from right to left

Commented [CT5]: True, sender is toggling on line 3 and receiver is reading on line 4.

Commented [CT6]: False they need to have a common ground.

3. We send the bit pattern 0b1011000101 over a 9600 bps connection. If the receiving side was set at 4800 bps, what would it receive? (Ignore the presence of start, stop and parity bits. Assume that the receiver always samples in the middle of a bit). When the receiver samples, it takes the value at the very instance of sampling, as shown here (the vertical arrow indicates receiver's sampling point):

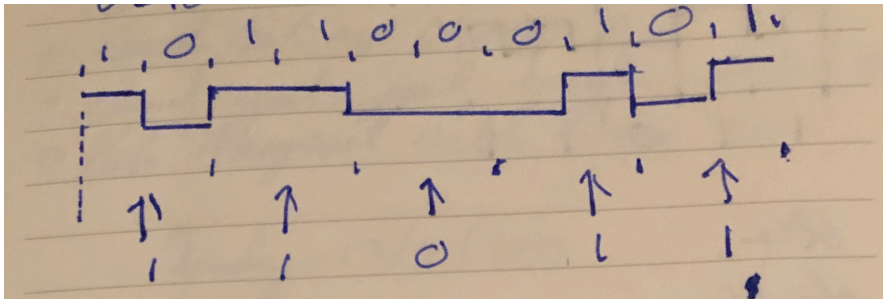


a. 0b11011
b. 0b0100111010
c. 0b00100
d. 0b0110101
e. 0b10110

Commented [CT7]: Main thing to understand is that bps indicates that each bit occupies some amount of time (9600 bps means each bit takes about 0.104 ms, while 4800 bps means each bit takes about 0.208 ms(. We then use a scaled drawing to represent this time. Here we use 1 cm = 0.104 ms in the upper tick marks, 2 cm = 0.208 ms in the lower tick marks. We then sketch the bit pattern using the upper tick marks. We sample using the halfway point of the lower tick marks and read off from there. See next page for sketch.

Explanation

The receiver is sampling at ½ the speed of the sender. Hence it is sampling only once every two bit positions relative to the sender. Diagram below shows what happens.



4. [Raspberry Pi] Which of the following statement(s) regarding the difference between Raspberry Pi and a typical PC is/are TRUE?

   i. Pi uses ARM processor while PC typically uses Intel processor.
   ii. Pi does not use RAM for runtime storage, unlike typical PC.
   iii. Pi does not have persistent storage, unlike typical PC.

   a. (i) only.
   b. (i) and (ii) only.
   c. (ii) and (iii) only.
   d. (i), (ii) and (iii).
   e. None of the above.

ANS:

Pi is a miniature PC, which the same main components: processor, RAM and secondary (persistent) storage. So, only statement (i) is correct.

Question 5-7 uses the following context:

Given an array A with N unsorted numbers and a target sum S, we want to find out whether there are two numbers in A that adds up to S. i.e. A[i] + A[j] == S. This is commonly known as the **pair-sum problem**.

5. If we solve the pair-sum problem by **exhaustively trying out all pairs of number**:

```
for (i = 0; i < N-1; i++){
    for (j = i+1; j < N; j++){
        if (A[i] + A[j] == S)  return (i, j);   //pseudo-code
```

The **worst-case** time complexity for this approach is:

   a. O(1)
   b. O($\log_2(N)$)
   c. O(N)
   d. O(N $\log_2(N)$)
   e. O(N^2)

ANS:

The if- statements get executed N-1, N-2, N-3….. 1 times ➔ O(N^2)

6. If we solve the pair-sum problem by:

```
A: bubble sort the array A,
B: for each A[i], binary search for (S-A[i])
```

What is the most accurate time complexity of this approach?

a. Phase A = $O(N^2)$; Phase B = $O(\log_2(N))$
b. Phase A = $O(N \log_2(N))$; Phase B = $O(\log_2(N))$
c. Phase A = $O(N^2)$; Phase B = $O(N \log_2(N))$
d. Phase A = $O(N \log_2(N))$; Phase B = $O(N \log_2(N))$
e. None of the above

ANS:

Bubble sort is $O(N^2)$

Each of the binary search is $O(\log_2(N))$, but we potentially need to search for ½ of the N elements ➔ $O(N \log_2(N))$.

So, (C) is the answer.

7. If we modify the solution in (3) by

```
A: bubble sort the array A,
B: set left = 0, right = N-1
   while (left < right) {
       if (A[left] + A[right] == S)  return (left, right);

      if (A[left] + A[right] > S)    right--;
      else if (A[left] + A[right] < S)    left++;
```

What is the most accurate time complexity of the **phase B**?

a. $O(1)$
b. $O(\log_2(N))$
c. $O(N)$
d. $O(N \log_2(N))$
e. $O(N^2)$

ANS:

Every iteration the left OR right will move one step. The code terminates when left and right meets ➔
Worst case the whole array is tested ➔ O(N)

8. [Git] Which of the following statements regarding the **staging** process of the Git is TRUE?

   a. If a file is already being tracked by Git, there is no need to stage the file to commit new change.
   b. To add a new file for Git to track, there is no need to stage the file to commit new change.
   c. We always need to stage **all modified files** before committing.
   d. Forgetting to stage a file will cause **merge conflict.**
   e. None of the above

ANS:

(a) to (c) are the opposite facts. (d) is simply made up 😉. Answer = (e)

9. Which of the following shows the correct sequence for interrupt processing?
(Note: PC = Program Counter, which keeps track of which instruction the CPU will execute next)

   a. Detect interrupt, get address of ISR, execute ISR, pop PC from stack, push PC to stack.
   b. Get address of ISR, detect interrupt, push PC to stack, get address of ISR, execute ISR, pop PC from stack.
   c. Push PC to stack, detect interrupt, get address of ISR, execute ISR, pop PC from stack.
   d. Detect interrupt, get address of ISR, push PC to stack, execute ISR, pop PC from stack.
   e. Detect interrupt, pop PC from stack, get address of ISR, execute ISR, push PC to stack.

Option d is correct: detect the interrupt, find the ISR for that interrupt, push PC to save it, execute the ISR, pop PC to restore execution to the previous code.

Option a. means that we lose the contents of PC, which prevents us from resuming execution of the interrupted code. Option b. doesn't make sense; how do we decide which ISR if we don't know which, if any interrupt was triggered? Option c is possible but if there's no interrupt then you need to pop PC. Option e will result in randomness being written to the PC when you pop before pushing it.

10. We have two buttons A and B connected to INT0 and INT1 respectively. EICRA is set so that INT0 and INT1 are triggered by the rising edge of the signal. EIMSK is set to enable both INT0 and INT1.

The following code is executed on an Atmega328P:

```
static volatile int turn = 0;
ISR(INT0_vect)
{
      turn=0;
}

ISR(INT1_vect)
{
   turn =1;
}

int main()
{
      while(1)
      {
            // flashRed and flashGreen flash the red and green LEDs
            // at a rate of 1 Hz, respectively.
            if(turn == 0)
                  flashRed();
            else
                  flashGreen();
      }
}
```

We press both A and B at exactly the same time, then release B, before releasing A. Which statement best describes what will happen?

a. The green LED will flash as long as B is held down, and the red LED will flash once B is released.
b. The red LED will flash while B is held down, and the green LED will flash once B is released.
c. The red LED will flash momentarily, then the green LED will flash.
d. The red LED will flash until button A is released, then the green LED will flash.
e. The green LED will flash momentarily, then the red LED will flash.

The answer is c. This is because INT0 and INT1 will be triggered at the same time, but INT0 has a higher priority than INT1, so flashRed is called. However once INT0_vect ISR is completed, control is handed to INT1_vect ISR, which will cause the green LED to flash.

Multiple Response Question Sample

Full marks will be given if ALL correct answers and NO wrong answers are selected. If only some correct answers are selected or any wrong answer is selected, you will be given partial credit.

MRQ1.

We have an Arduino with a 16 MHz clock. Timer 0 is configured for CTC mode, with OCR0A set to 249, and the TIMER0_COMPA_vect interrupt is enabled and captured. Which of the following are valid triggering intervals for TIMER0_COMPA_vect? Choose all that are correct. (Note: The counter starts from 0 and hence there are 250 steps from 0 to 249)

The valid prescalar values and their corresponding resolutions are shown below:

Prescaler	Resolution (16 MHz Clock)	Count *V* (for 1 ms or 1000 microseconds)
1	0.0625 microseconds	16000
8	0.5 microseconds	2000
64	4 microseconds	250
256	16 microseconds	62.5
1024	64 microseconds	15.63

Thus the valid intervals are (1 us = 1 microsecond):

(249 + 1) x 0.0625 = 15.625 us
(249 + 1) x 0.5 = 125 us
(249 + 1) x 4 = 1000 us = 1 ms
(249 + 1) x 16 = 4000 us = 4 ms
(249 + 1) x 64 = 16000 us = 16 ms

a. 20 us
b. +125 us
c. 2 ms
d. +4 ms
e. 20 ms

The only valid values are 125 us and 4 ms. Hence the answer is b. and d. Note that the answers presented may not be exhaustive.