16. Pontos időzítés - Timerek

Írta: Treszkai László

Lektorálta: Proksa Gábor

Ebben a tananyagrészben az időzítő (*timer*) periféria programozásán keresztül tanulhatsz olyan alapelveket, amelyek hasznodra válnak majd egy villamosmérnök vagy szoftverfejlesztő karrier során.

BEVEZETÉS

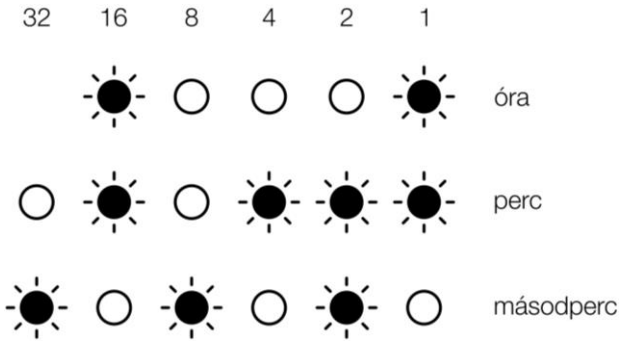
Egy digitális óra szoftverét fogjuk megírni közösen, két módon: először a timer használata nélkül, majd annak a segítségével. Sok helyen nem írtam le a teljes kódrészletet, a kérdéseket pedig helyenként megválaszolatlanul hagytam. Bár a tananyag mellékletében, ill. honlapján megtalálható a teljes, működő szoftver, javaslom, hogy a hiányzó részeket írd meg magad, a kérdéseken gondolkodj el – így tanulhatsz a legtöbbet. (Márpedig azért olvasod a tananyagot, igaz?) Egy szakácskönyvet forgathatunk azért, hogy jól kinéző receptet találjunk, de csak attól, hogy elolvassuk őket, még nem fogunk megtanulni főzni.

Első példának kezdjük a kőkorszakkal: van egy processzorunk, és ezzel szeretnénk egy digitális órát építeni, ami jól nézne ki a *Vissza a jövőbe* következő részében.

A TIMER PERIFÉRIA HASZNÁLATA NÉLKÜL HOGY NÉZNE KI EGY DIGITÁLIS ÓRA SZOFTVERE?

Mit csinál a legegyszerűbb digitális óra?
Az enyém most épp 17:23:42-t mutat. Hopp, már 17:23:43-at, mi történik? Úgy fogalmaznék, hogy miután bekapcsolt, folyamatosan kijelzi az időt másodperc pontossággal. Gyakorlatban ez azt jelenti, hogy egy számláló értékét növeli másodpercenként, ami éjfélkor 0-ról indul, és 86 400 másodperc elteltével átfordul 0-ba (ezt a jelenséget úgy nevezik, hogy a számláló „túlcsordul”). Ennek a számlálónak az értékét jelzi ki a karóra, általában „óra:perc:másodperc” formátumban.

Egy „bináris” óránál LED-ek jelzik kettes számrendszerben az órákat, perceket, másodperceket. Például az 1. ábrán az alsó sorban lévő LED-ek jelzik a másodperceket: a 32-es, 8-as és 2-es LED-ek világítanak, ami 32+8+2 = 42 értéknek felel meg. Le tudod olvasni, mennyi időt mutat ez a kijelző?



1. ábra *- Bináris óra példa állása*

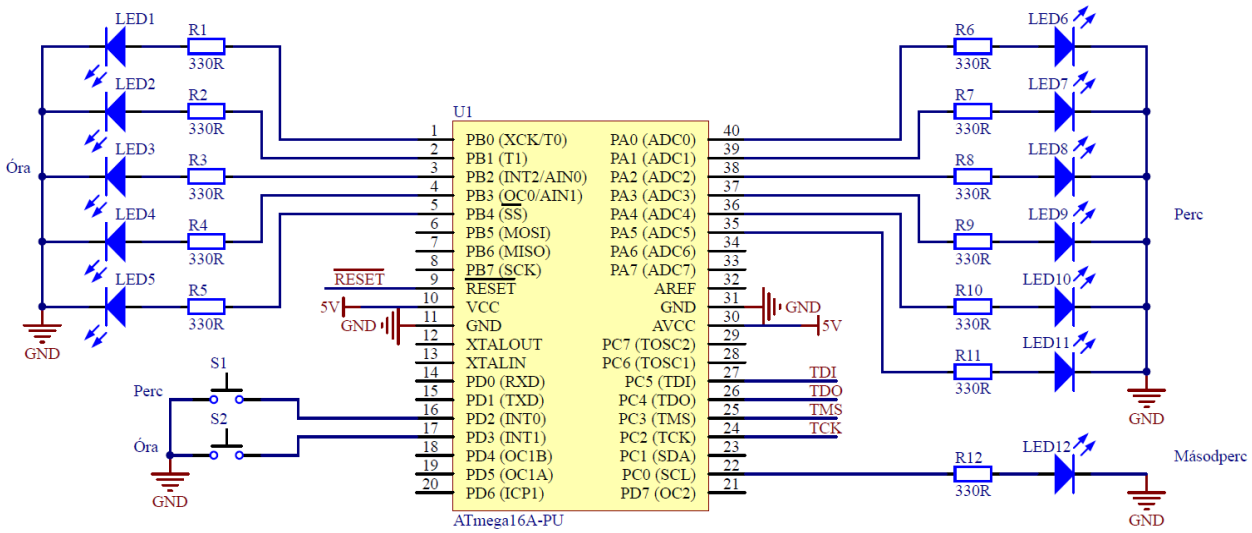
Honnan tudja egyáltalán az óra, hogy mennyi az idő?

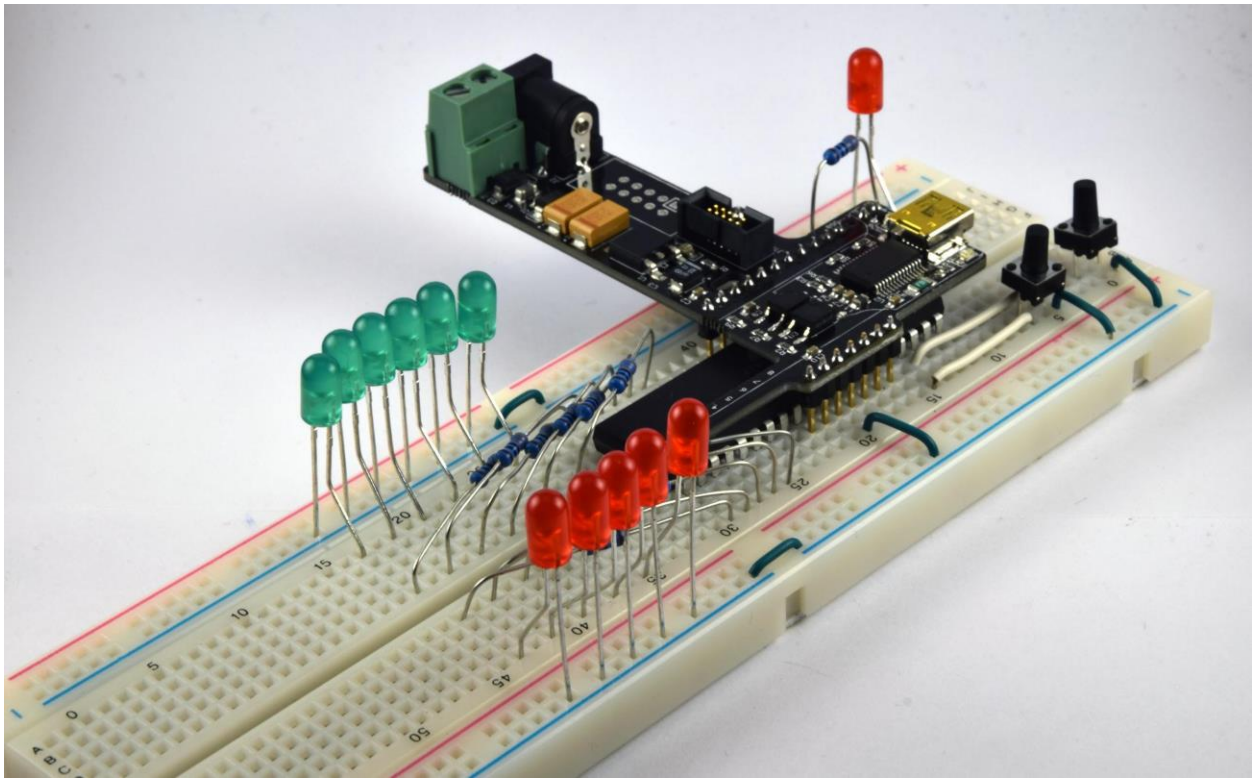
Sehonnan, bekapcsoláskor 0:00:00-ról indul, és nekünk kell beállítani. A példakódban ezt is a lehető legegyszerűbben oldjuk majd meg: egy gombbal a percszámlálót lehet növelni, egy másikkal pedig az órák értékét. Ez a megvalósítás hagy kívánnivalót maga után: érdemes elgondolkozni azon, hogyan legyen a *fejlesztés* helyett a *használat* minél egyszerűbb és kényelmesebb. Vizsgáld meg a körülötted lévő használati tárgyak kezelőfelületét!

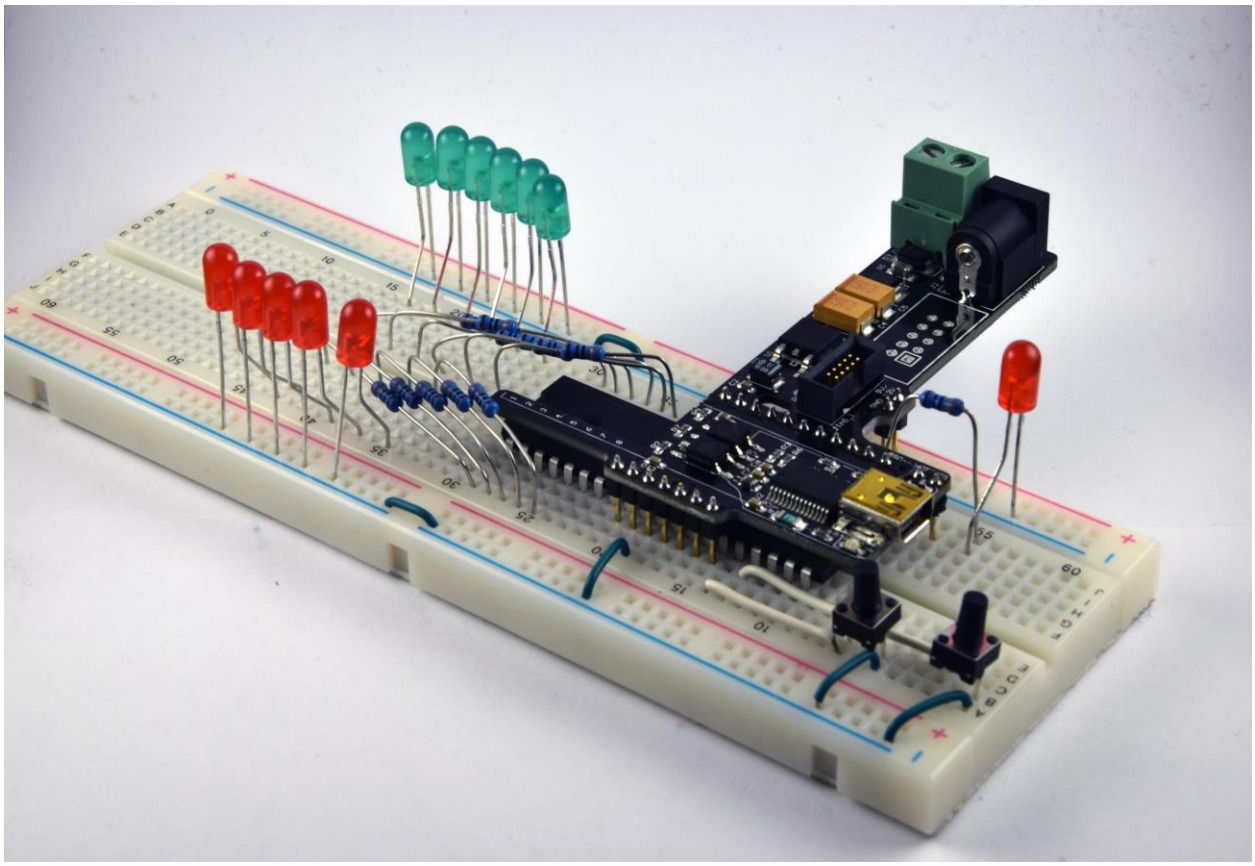
Hogyan épüljön fel az eszközünk?

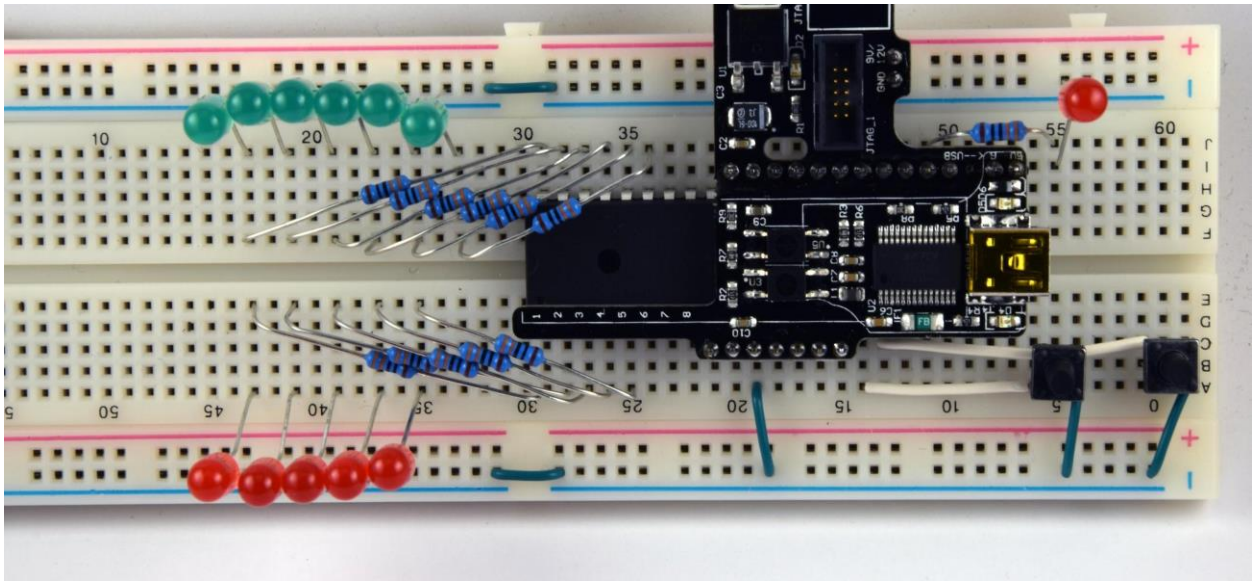
A fényforrásként választott LED-eket egy-egy sorba kapcsolt ellenálláson keresztül köthetjük a mikrokontroller lába és a föld – a negatív táp – közé. Így ha azt akarjuk, hogy világítson a LED, akkor a kimenetet magas állapotba kell állítanunk.

A bemeneteket alapállapotban a mikrokontroller belső felhúzóellenállása húzza fel tápfeszültségre, és a nyomógombok húzzák le a földre. Így a szoftverből a nyomógomb benyomását úgy érzékelhetjük, hogy a bemeneten alacsony értéket olvasunk.









***2. ábra** - Kapcsolási rajz az eszközről*

Még azt kell eldönteni, hogy milyen LED-et és mekkora értékű ellenállást válasszunk. A két kérdés összefügg, mert az ellenállás értékének növelésével a LED árama csökken, így annak a fényereje is: adott típusú LED fényereje a rajta keresztülfolyó árammal arányosan nő.

Általában szeretnénk, hogy egy fényforrás minél fényesebben világítson, de mind a LED, mind a mikrokontroller kimeneteinek a maximális árama limitált. Az előbbit a LED adatlapjából olvashatjuk ki, de a LED-ek többsége 20 mA-t biztosan kibír. A kimenetek áramkorlátját a mikrokontroller adatlapjában a 27.2. „DC Characteristics” táblázat alatti megjegyzések írják le: az A port kimeneteit összesen 100 mA-rel, a B és C portok kimeneteit *összesen* 100 mA-rel lehet terhelni PDIP tokozás esetén.

A B porton 5, a C porton 1 LED van; ha ezen a 6 LED-en ugyanakkora áram folyik keresztül, az legfeljebb 16 mA-t jelent kimenetenként. Az A port kimeneteinél is ilyen számokat kapunk eredményül.

Az ellenállás méretezéséhez szükségünk van az áramára és a feszültségére. A rajta eső feszültség a LED-en eső feszültségtől függ, ami pedig a LED színétől és a rajta keresztülfolyó áramtól. A mellékelthez hasonló piros és zöld LED-eken 16 mA-es áramnál körülbelül 2 V feszültség esik, úgyhogy ha a tápfeszültségünk 5 V, akkor a sorba kötött ellenálláson 3 V mérhető. (A pontos feszültség–áram-karakterisztika függ a LED típusától, úgyhogy az adott LED adatlapjában egy grafikon írja le az áramerősséget a feszültség függvényében.)

Az ellenállás értékét végül az Ohm-törvény alapján számolhatjuk ki: $R = U / I = 3\ V / 16\ mA \approx 0.19\ k\Omega = 190\ \Omega$. Minél *nagyobb* az ellenállás értéke, a LED-en annál *kevesebb* áram folyik át, úgyhogy *legalább* ekkora kell legyen az ellenállásunk értéke. (Az ellenállások toleranciája miatt két tizedesjegynél többel nincs értelme számolni.)

Hogyan épüljön fel a programunk?

A kezdeti inicializálás után egy végtelen ciklusban végzünk el három feladatot: beolvassuk a nyomógombok állását, frissítjük a kijelzőt, és várakozunk egy másodpercet a következő frissítésig. Valahogy így:

```c
int main(void)
{
	uint32_t time = 0;

	IoInit();

	while (1)
	{
		time = SetTime(time);

		UpdateDisplay(time);
		WaitSecond();

		time++;
		if (time >= 86400)
		{
			time = 0;
		}
	}
	return 0;
}
```

Az `IoInit()` függvény a használt I/O portokat inicializálja, amiről az adatlap 12.2.1. szakasza ad leírást. A DDRx regiszterek bitjei leírják, hogy a megfelelő pint ki- vagy bemenetként használjuk-e (ahol *x* a port azonosítója, azaz A, B, C vagy D). Például ha DDRB-nek a 0. és 2. bitje 1-es (azaz a regiszter értéke 0000 0101_2 = 4 + 1 = 5), akkor PB0 és PB2 kimenet lesz, a többi PBn pin pedig bemenet. Ha egy pint bemenetként használunk, akkor a PORTx regiszter megfelelő bitjének 1-be állításával kapcsolhatjuk be a belső felhúzóellenállást. Ezek alapján meg tudod írni az `IoInit()` függvényt is egyedül. Bízom benned, sikerülni fog! :)

A `SetTime()` függvény beolvassa a nyomógombok állását, azok alapján hozzáad 60-at vagy 3600-at a `time` változóhoz, és visszatér `time` új értékével. A PD2 bemenet állapotát a PIND regiszter 2. bitje tartalmazza. (Ahol a bitek számozása természetesen 0-tól kezdődik, kezdve a legalacsonyabb helyiértékű bittel.) Emlékezz vissza, hogy ha a nyomógomb be van nyomva, akkor a bemeneten alacsony értéket olvasunk.

Az `UpdateDisplay()` függvénnyel csak digitális kimeneteket kezelünk, a korábbi tananyagrészekben megszerzett tudásunkkal. Az egyetlen kihívás az, hogy a másodperceket tartalmazó változóból hogyan számoljuk ki az órák, percek, másodpercek értékét. Ehhez jó ugrókő az, ha egy tízes számrendszerbeli számból próbáljuk kiszámolni, hogy az egyes helyiértékei milyen számjegyet takarnak. Például ha a függvényünk bemenete 527, akkor a következők lesznek igazak:

- 52<u>7</u> % 10 = <u>7</u>
- <u>52</u>7 / 10 = <u>52</u>
- 5<u>2</u> % 10 = <u>2</u>
- <u>52</u> / 10 = <u>5</u>

Ezek alapján írd meg az `UpdateDisplay()` függvényt! Annyi különbség lesz a fenti példához képest, hogy míg egy tízes számrendszerbeli szám összes számjegye 10-féle értéket vehet fel, addig a karóránkon a másodperc- és percmutatók 60-féle értéket, az óramutató 24-féle értéket mutathatnak. A kimenetek kezelése nagyon egyszerű lesz, ha az órákat, perceket, másodperceket egy-egy port egymás utáni bitjei jeleznek ki. Például ha a `minutes` változó a percek értékét tartalmazza, akkor a

```
PORTA = minutes;
```

utasítás egyszerre beállítja az összes perckimenetet.

A `WaitSecond()` függvény célja, hogy várjon egy másodpercet, mialatt azzal tölti el az időt, hogy számol, körülbelül „egy, megérett a meggy, kettő, csipkebokor vessző” stílusban. Mondjuk az közelebb áll az igazsághoz, hogy egy lokális (helyi) változó értékét növeli.

```
#define SECOND_COUNTER_MAX 1000000

void WaitSecond(void)
{
	volatile uint32_t i;

/* Ennek a for ciklusnak nincsen semmilyen hatása a WaitSecond() függvényen kívül, és ezzel a fordító is tisztában van. Azzal viszont nincs tisztában, hogy a futása alatt telik az idő, pedig mi pont ezt használjuk ki. Ha i nem volatile változó lenne, akkor a fordító optimalizációkor kitörölhetné az egész for ciklust. */
	for (i = 0; i < SECOND_COUNTER_MAX; i++)
	{
/* Szándékosan üres ciklusmag. A tétlen várakozást az i indexváltozó egyesével növelésével érjük el. */
	}
}
```

A tananyag honlapján van a példakódnak egy olyan változata, ami még kiegészítésre vár. Töltsd ki a „TODO” megjegyzéssel jelölt hiányzó részeket, töltsd fel a mikrokontrollerre a programot, javítsd ki az esetleges hibákat, és gyönyörködj a menő kütyüdben!

Változások nyomon követése a szoftverünkbe

Amikor szoftverfejlesztés közben folyamatosan módosítgatunk a szoftveren, előfordulhat, hogy valamelyik változtatás nem válik be, és ilyenkor szeretnénk egy korábbi verzióhoz visszatérni. Egy kézenfekvő megoldás, ha nagyobb változtatások előtt készítünk egy másolatot a forráskódról – ez azonban hamar fejetlenséghez vezet, és így könnyű elveszíteni, melyik verzió melyik után következett, vagy mi volt benne a módosítás. Ezen segítenek a verziókövető szoftverek, amik egy adatbázisban tárolnak minden változtatást a mi megjegyzéseinkkel, és ezek így később könnyen visszakereshetőek. Én a honlapon található példakód fejlesztésekor az ingyenes Git rendszert és TortoiseGit programot használtam; azzal megtekintheted a szerkesztési előzményeket.

Milyen feltételezésekkel éltünk ennek a programnak a megírásakor, miket kellett lemérnünk/kísérleteznünk, milyen problémákkal szembesültünk?

1. Tudnunk kell, mennyi ideig tart egy for ciklus végrehajtása. Ezt az órajel és a mikrokontroller ismeretében ki lehet számolni, de az macerás: a mikrokontroller adatlapjának a 30. fejezete leírja, melyik gépi utasítás hány órajelciklusig tart. Egyszerűbb, ha lemérjük, milyen gyorsan jár az óránk a `SECOND_COUNTER_MAX` makró egy tetszőleges értékével, és ez alapján korrigáljuk az értékét. Például ha azt látod, hogy az órád ötször gyorsabban jár a kelleténél, akkor a makró értékét csökkentsd az ötödére, majd végezd el ismét a mérést, hogy egy finomabb változtatást tehess rajta. Ennek a mérésnek az elvégzését rád hagyom: a példaprogramban az említett makró csak egy közelítő értéket tartalmaz, azzal az óránk nem fog a megfelelő ütemben járni. A fent leírt módon végezd el a `SECOND_COUNTER_MAX` makró kalibrálását! (Megjegyzés: A `WaitSecond()` függvényben használhatnánk az Atmel által nyújtott `_delay_ms()` makrót is, de a többi problémánk azzal is fennállna.)

Méréstechnikai kitekintés

Minél nagyobb értéket mérünk, annál pontosabb eredményt kapunk, ha ugyanakkora a hibánk abszolút értéke.

Például van egy ránézésre pontos, de ismeretlen óránk, aminek az ütemét szeretnénk lemérni egy pontos órával. Ezt megtehetjük úgy, hogy lemérjük a pontos óránkkal, hogy a mérendő óránkon mennyi idő alatt telik el egy perc. Ez a mérés egy fix hibát fog tartalmazni, a kézi órakezelés miatt mondjuk két tized másodpercet.

Ha egy eltelt perc helyett egy óra idejét mérjük, akkor annak a hibája szintén két tized másodperc lesz. De vegyük észre, hogy az utóbbi esetben a hiba a mért érték (0,2 / 3600) = 0,0055%-a, míg az előbbi esetben a (0,2 / 60) = 0,33%-a, ami napi 24 · 60 · 0,33% = 4,7 perc csúszást jelentene! Ennyi a különbség egy használhatatlan és egy éppen hogy használható óra között.

2. Feltételeztük, hogy a függvényeink végrehajtási ideje minden ciklusban ugyanannyi. Ez nem igaz, például ha van benne olyan `if-else` elágazás, aminek az `if` és az `else` ága nem ugyanolyan hosszú. Ha használnánk megszakításokat (amikről a tananyag egy későbbi részében lesz szó), akkor azok futási idejével csak nagyon nehezen tudnánk számolni.
3. Feltételeztük, hogy a fejlesztés során a `while` ciklus végrehajtási ideje állandó. Azonban ha változtatunk a programon vagy a fordító optimalizációs beállításain, akkor a `while` ciklus többé nem pontosan ugyanannyi időt vesz igénybe, ezért ekkor a `SECOND_COUNTER_MAX` állandót újra kell kalibrálni.
4. A `WaitSecond()` függvény futása alatt a mikrokontroller nem tud semmilyen más feladatot elvégezni. (Például a karóra gombjait vagy háttérvilágítását kezelni.) Ha ezzel mégis megpróbálkoznánk, az az óránk késését eredményezné.

Ezek közül az összes problémára megoldást nyújt a timer periféria használata.

MIRE VALÓ A TIMER PERIFÉRIA?

Egy mikrokontroller annyival több egy processzornál, hogy a művelet-végrehajtó magon kívül kiegészítő áramköröket, ún. perifériákat tartalmaz, a környezettel való kommunikáció megkönnyítése érdekében.

A timer/counter periféria fő funkciója, hogy négyszögjelek éleit számolja. Ezek általában egy belső órajel élei (amik fix ütemben érkeznek, így az idő múlását jól jelzik, ezért a *timer* megnevezés), de a periféria képes külső négyszögjelek éleinek számolására is, például egy mosógép vagy belsőégésű motor tengelyének fordulatainak a számolására (amik nem feltétlenül fix üteműek, ezért a *counter* megnevezés).

A timer leveszi a main függvény válláról az idő nyilvántartásának a terhét, így a szoftverfejlesztőnek kevesebb munkája van vele, mégis egy megbízhatóbb időforrást kap. Amíg a mikrokontroller magja utasítások végrehajtásával foglalkozik, a timer a háttérben folyamatosan növeli egy számláló értékét.

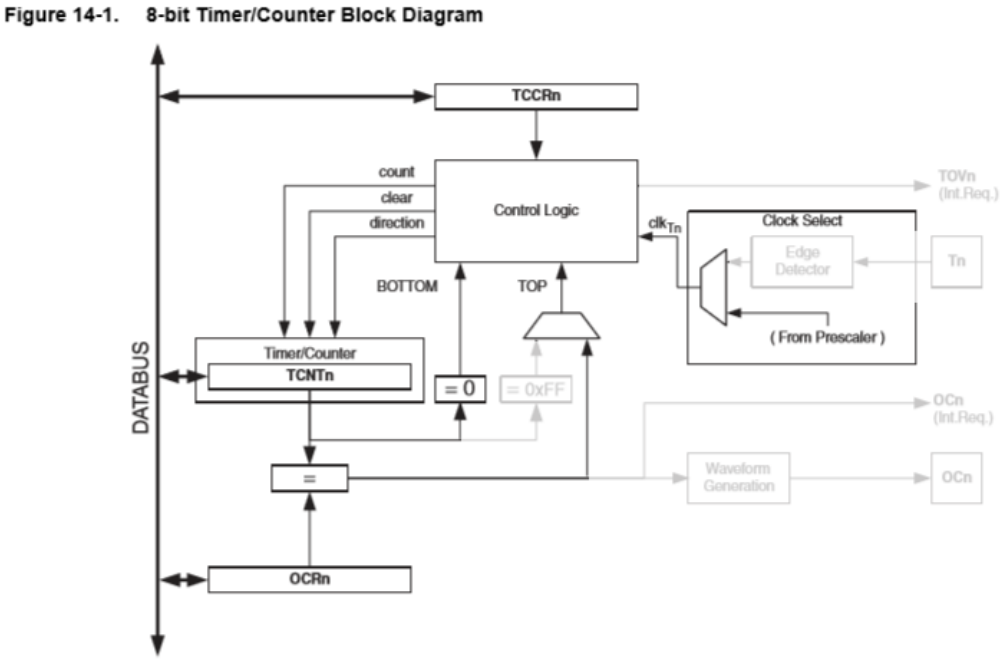
Az időzítő segítségével többek közt egyszerűen le tudjuk kérdezni, hogy mennyi idő telt el a mikrokontrollerünk indulása óta (ezt nevezik *uptime*-nak), képesek vagyunk milliszekundum, vagy mikroszekundum pontosságú késleltetésekre, vagy végrehajthatunk bizonyos feladatokat periodikusan: például analóg jelek beolvasására ezerszer egy másodpercben, vagy a fűtés ki-/bekapcsolására egyszer óránként.

Ez a periféria képes PWM-jelek generálására is – erről a következő részben lesz szó.

HOGYAN MŰKÖDIK A TIMER?

Az ATmega16A mikrokontrontrollerben három timer periféria van, melyek a Timer/Counter0, 1, 2 neveket viselik. Ezek közül a Timer/Counter0 a legegyszerűbb, aminek a számlálója 8 bit széles, és egy PWM jel generálására képes. A Timer/Counter1 16 bites, és két PWM jel előállítására, valamint egy külső esemény idejének rögzítésére képes. A Timer/Counter2 ismét csak 8 bites, de rendelkezik egy saját oszcillátorral, így képes a rendszer órajelétől független frekvenciával számolni.

Az adatlap 14.2. szakaszában láthatjuk a Timer/Counter0 periféria felépítését, amiből az alábbi kép mutatja be a számunkra fontos részeket:



***3. ábra** - A Timer/Counter0 belső felépítése*

A TCNTn* regiszter a számláló, amit a *timer control logic* vezérel, a TCCRn regiszterben megtalálható beállításoknak megfelelően. (Itt *n* a számláló sorszámát jelenti, azaz a Timer/Counter0 esetén TCNT0, TCCR0… a megfelelő regiszterek neve.) Ezek a beállítások a Timer0 esetében a következőket befolyásolják:

- bemeneti jel forrása külső jel legyen, vagy egy belső órajel az úgynevezett *prescaler*en keresztül (CS0 bitek);
- a számláló maximális értéke 0xFF legyen, vagy az OCR0 regiszter értéke (WGM0 bitek);
- egyéb, PWM-jelgenerálással kapcsolatos beállítások (COM0 bitek).

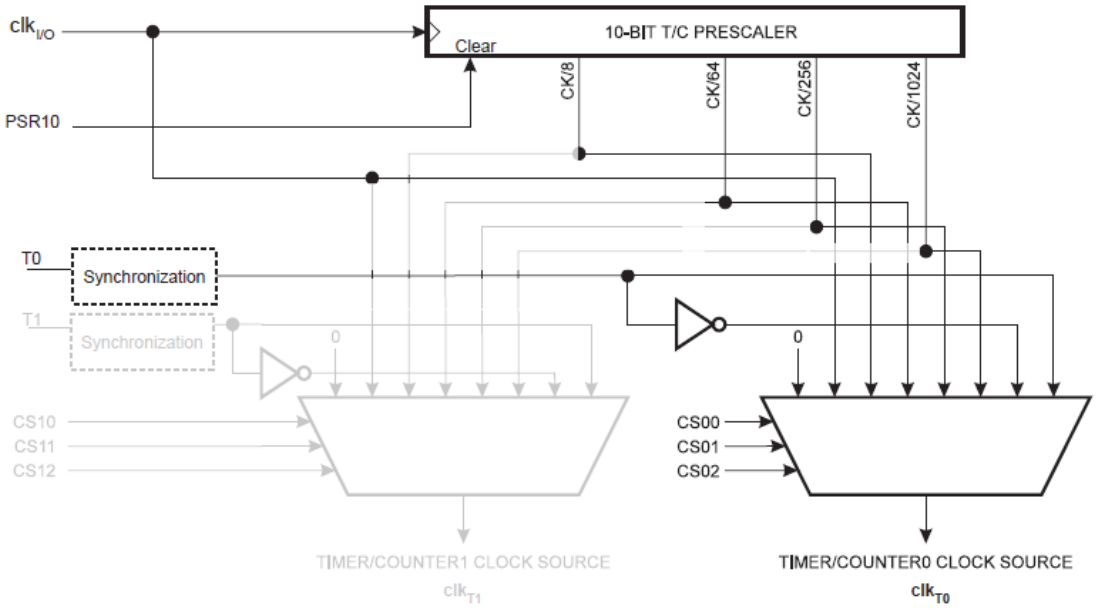
Azt, hogy a számláló milyen frekvenciával számol, az órajel forrása és a prescaler beállítása határozza meg, amiket a következő oldalakon írok le.

Mi az a prescaler?

Ha a mikrokontroller 16 MHz-es órajelet használ a maximális számolási teljesítményhez, és ha az órajel minden felfutó élének hatására eggyel növekszik a számláló értéke, akkor a 8 bites számláló 16 µs alatt éri el a maximális értékét, 255-öt. Bár egy ilyen gyors számlálóval is lehet nagyobb időket mérni, gyakran praktikus, ha a számláló lassabban jár. Ezt a prescaler – magyarul gyakran „előosztó” – oldja meg úgy, hogy leosztja a bemenetén lévő órajel frekvenciáját egy egész számmal, például a 16 MHz-es órajel frekvenciáját leosztja a nyolcadára, így a kimenetén 2 MHz-es órajel lesz. Ez ahhoz is szükséges, hogy nagyobb periódusidejű hardveres PWM-jeleket generáljunk, amit egy 8-bites számlálóval és gyors órajellel prescaler nélkül nem lehetne.

Az adatlap 14-9. ábráján jól láthatók a jelalakok prescaler használata esetén. (Ezzel szemben a 14-8. ábrán a prescaler nincs használatban.) A prescalert használni fogjuk majd a következő példaprogramban.

Figure 15-2. Prescaler for Timer/Counter0 and Timer/Counter1(1)



***4. ábra** - A prescaler belső felépítése*

MILYEN BEÁLLÍTÁSI LEHETŐSÉGEKKEL RENDELKEZIK A TIMER PERIFÉRIA?

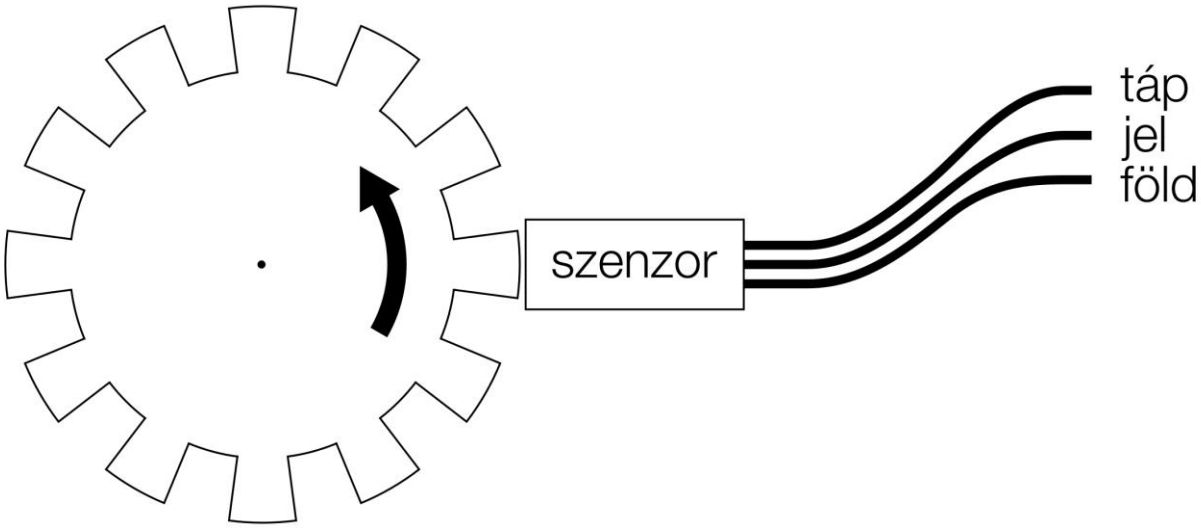
A két legfontosabb beállítási lehetőség közül az egyik az, hogyan működjön a számláló (angolul *„mode of operation”*).

Normal módban a számláló értéke 0-ról indul, és egyesével növeli a számláló értékét addig, amíg az el nem éri az adattípusának megfelelő maximális értéket (azaz 8 bites számláló esetén 255-öt, 16 bites számláló esetén 65535-öt). Ezután a következő órajelre a számláló túlcsordul, és 0-ról folytatja a számolást. Ezt a túlcsordulás eseményt a TOVn bit jelzi, ami a „timer overflow” kifejezésből kapta a nevét. Ezt a módot Timer/Counter0 esetében a TCCR0 regiszter WGM0[1:0] bitjeinek 0-ba állításával választhatjuk ki (lásd az adatlap 14-2. táblázatát). (Egy változó bitjeit szokás úgy jelölni, hogy a változó neve után szögletes zárójelben írjuk a bit sorszámát, ahol 0 a legkisebb helyiértékű bitet jelöli. A TCNT1[15:8] kifejezés a TCNT1 változó 15.-től 8.-ig helyiértékig terjedő bitjeit jelenti.)

Clear timer on compare match (CTC) módban a számláló 0-ról indul, ami érték szintén egyesével nő. Amint a számláló értéke nagyobb lenne az OCRn (Output Compare Register) regiszterben beállított számnál, a számláló lenullázódik, és 0-ról folytatja a számolást. Ezzel egyidőben az OCFn (Output Compare Flag) bit értéke 1 lesz, amíg azt ki nem nullázza a programunk. (Lásd az adatlap 14-11. ábráját.)

A másik fontos beállítás az órajel forrása, amit a TCCR0 regiszter CS0[2:0] bitjeivel tudunk kiválasztani (lásd az adatlap 14-6. táblázatát). Lehetőségünk van a timer megállítására, a mikrokontroller fő órajelének használatára, a fő órajel prescaler által leosztott változatára, ill. a T0 pinen lévő jel számlálására. A T0 pinen érkező külső órajel segítségével a mikrokontrolleren kívülről érkező impulzusokat számolhatunk: például egy szenzor kimeneti jelét, ami a személyautók kerekén van egy fogaskerékhez közel. Ennek a szenzornak a kimenete egy digitális jel, azaz két értéket vehet fel: alapállapotban magas értéket jelez (azaz egy

tápfeszültséghez közeli értéket), de ha közel van hozzá egy fog, akkor alacsony értéket. Abból, hogy egységnyi idő alatt mennyi élváltást érzékel a mikrokontroller, kiszámolhatjuk a kerék sebességét.



***5. ábra** - Kerékszögsebesség mérése fogaskerék-érzékelő segítségével*

MILYEN LEHETŐSÉGEINK VANNAK, HA PONTOSAN AKARUNK IDŐZÍTENI VALAMIT?

Az adatlap 8. fejezete bemutatja, milyen órajelforrásokat lehet használni ezzel a mikrokontrollerrel, ezeket összegeztem az alábbi táblázatban.

	Frekvencia	Ár	Pontosság	Start-up time
Külső nagyfrekvenciás kristály	max. 16 MHz	30…500 Ft	10…100 ppm	16k órajel
Külső rezonátor	max. 8 MHz	100 Ft	0,1…0,5%	1k órajel
Külső alacsony frekvenciás kristály	32 kHz	30…100 Ft	10…100 ppm	1k / 32k órajel
Külső RC-oszcillátor	max. 12 MHz	5…20 Ft	5…10%	6-18 órajel
Belső RC-oszcillátor	1 / 2 / 4 / 8 MHz	0 Ft	1…3%	6 órajel
Külső órajelforrás	max. 16 MHz	?	?	6 órajel

Fontos tervezési szempontot jelentenek a költségeink, amik alatt nem csak az alkatrészek árát értem, hanem a tervezőmérnök idejét is. Mind az ár, mind az egyszerűség szempontjából a mikrokontroller belső RC-oszcillátora a nyerő, azonban az nagyságrendekkel kevésbé pontos, mint egy külső kristály.

Ha pontos és gyors időzítésre van szükségünk, akkor általában egy külső kristály a legjobb megoldás.

A külső órajelforrás (external clock) beállításra akkor van szükségünk, ha egy áramkörben több mikrokontroller van, és szükséges, hogy azok pontosan egyező frekvencián járjanak.

Milyen költségeink vannak?

Az alkatrészek ára nyilvánvaló, hogy miért fontos: ha 1000 darabot kell legyártani egy áramkörből, akkor egy 50 forinttal olcsóbb alkatrésszel 50 000 forintot takarítunk meg. A befektetett időnket viszont semmiképp sem szabad figyelmen kívül hagyni!

Az életben nagyon kevés az időnk, és akármivel is töltjük el, annál kevesebbet tudunk más, esetleg fontosabb vagy élvezetesebb dolgokkal foglalkozni.

A mérnökök órabére magas, és a munkáltatóiknak az az érdeke, hogy minél gyorsabban, hatékonyabban dolgozzanak.

Gyakran kell szoros határidővel dolgozni, hogy elérjünk egy fix dátumot (például a karácsonyi ajándéknak december 24-ére becsomagolva kész kell lennie), vagy hogy a konkurenciánál gyorsabbak legyünk (például a következő generációs mobiltelefonunk hamarabb jöjjön ki).

Minél hamarabb végzünk egy projekttel, annál hamarabb élvezhetjük az előnyeit (például a karácsonyi fények örömét, vagy az eladásból származó bevételeket).

Ezeket összeadva egy 50 000 forintos megtakarítás már nem éri meg minden esetben.

HOGYAN HASZNÁLJUK A TIMERT A DIGITÁLIS ÓRÁNKHOZ?

Mennyire kell, hogy pontos legyen az óránk?
Ezt abban szokták mérni, hogy adott idő alatt (pl. 100 másodperc alatt) mennyit késhet vagy siethet: pl. egy ±0,5% frekvenciatoleranciájú 10 MHz-es rezonátor 1 másodperc alatt 9 950 000 és 10 050 000 közötti impulzust ad le. Ennél pontosabb eszközöknél a toleranciát ppm-ben adják meg (*part per million*), ami 0,0001%-nak (a névleges frekvencia egymilliomodának) felel meg.

Én azt várom el az órámtól, hogy egy hónap alatt ne csússzon el az ideje többet, mint fél perc: ez naponta egy másodpercet jelent. Ez lefelé kerekítve $1/(24 \cdot 60 \cdot 60)=11$ ppm-et jelent.

Nem mindegy, hogy kicsit túl pontos, vagy kicsit túl pontatlan

Miért lefelé kerekítettem, amikor az eredmény 11,57 ppm volt?

Azért, mert az én követelményem az, hogy ennél pontosabb legyen – ha az órám 12 ppm-et késne, akkor az már nem teljesítené a követelményeimet. A tűrések irányára mindig oda kell figyelni: a tanárt nem

érdekli, ha korábban vagy ott az óráján, de ha késel, az annál inkább. Ha az autó féktávolsága fél méterrel rövidebb a szükségesnél, annak örülünk, de ha fél méterrel hosszabb…

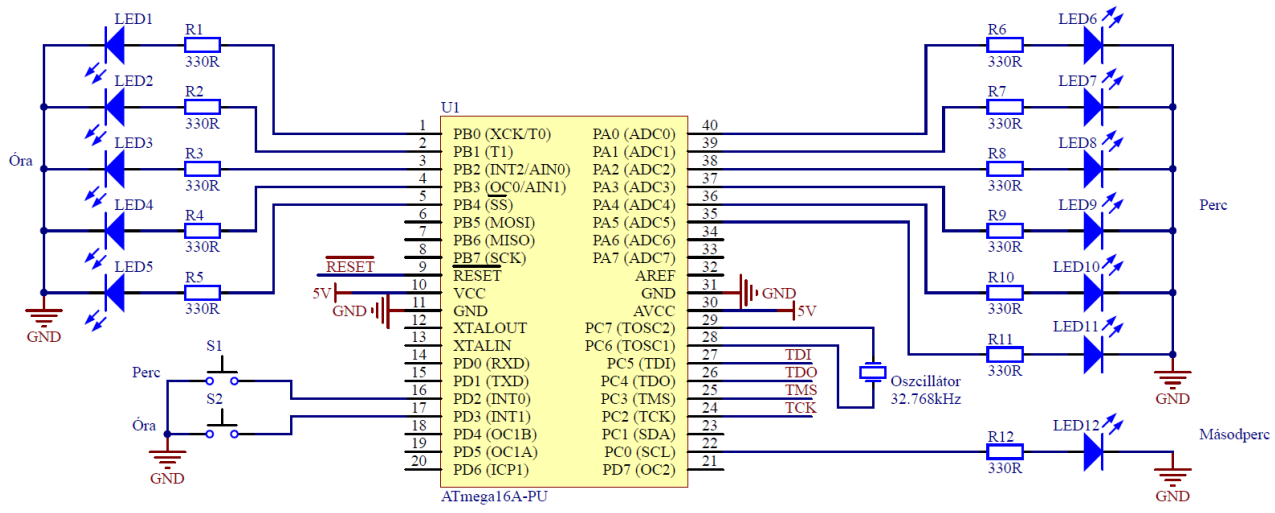
Ilyen pontosságú RC-oszcillátort lehetetlen készíteni, kerámia rezonátort pedig nehezen találnánk, ha egyáltalán létezik ilyen. Valójában ezzel a 11 ppm-mel elég magasra tettük a lécet: a tananyagrész írásakor az interneten könnyen beszerezhető legpontosabb kvarcok 10 ppm-esek. A kalibrált belső oszcillátorra csak 1%-os pontosságot garantál a gyártó, amivel akár naponta negyed órát is késhetne az óránk. Az nem lenne praktikus.

Prototípusok kalibrálása

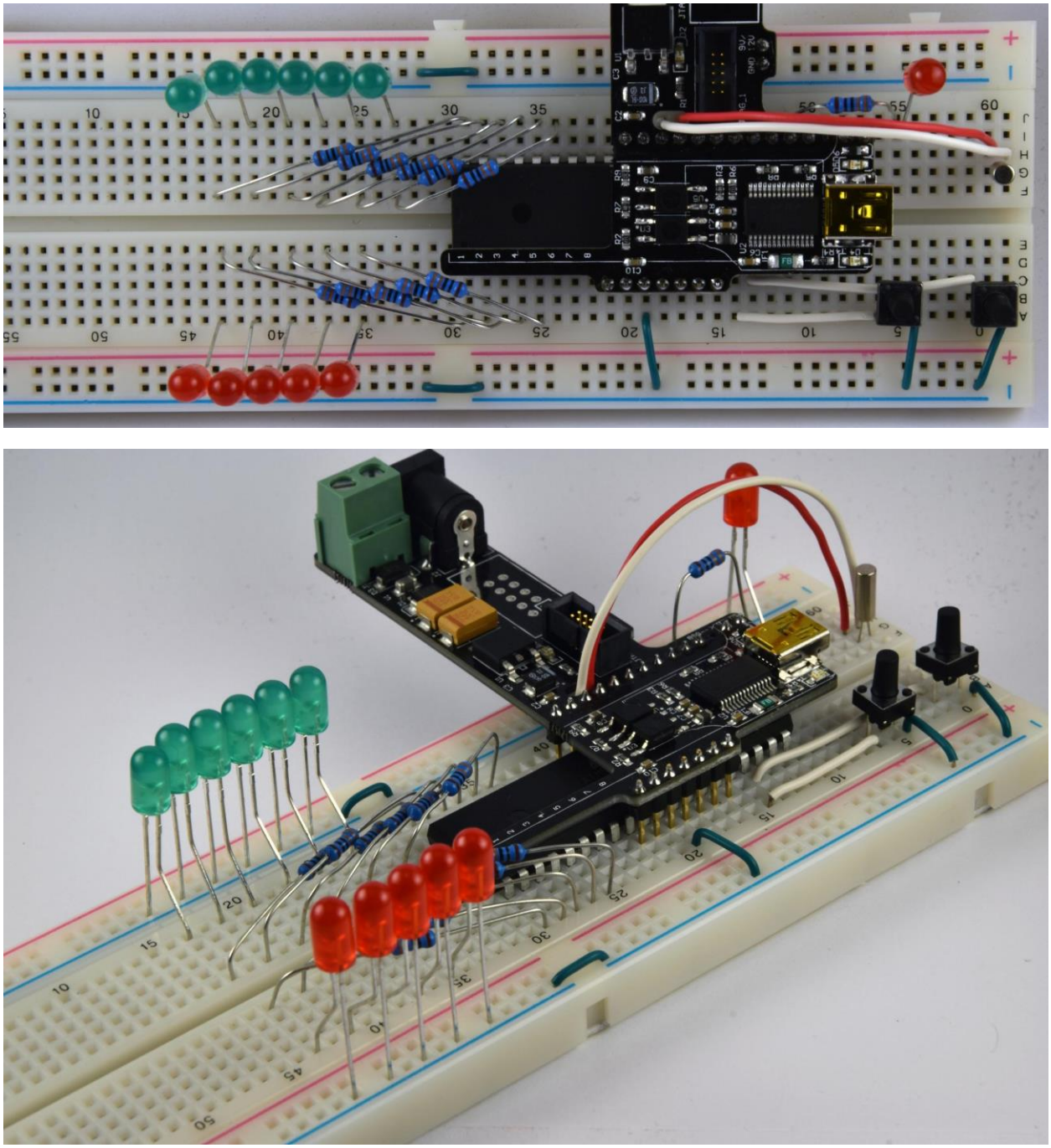
Igazából csak sorozatgyártott áramköröknél szükséges az órajelnek *pontosnak* lennie, nagyon alacsony példányszámnál elég, ha *stabil*. Ha csak egyetlen darabot kell gyártani, akkor azt kell figyelembe venni, hogy az oszcillátor frekvenciája mennyire változik különböző hőmérsékleteken, valamint az idő haladtával. A kezdeti hibát ugyanis ki lehet javítani a szoftverben, ezt nevezzük kalibrálásnak.

Tehát mindenképp kell egy kvarc az óránkba, hogy pontos legyen. (Nem véletlen a „kvarcóra” elnevezés.) Még így is két lehetőségünk van: a kvarcot vagy a mikrokontroller fő órajelének használjuk, vagy csak a Timer/Counter2 külső órajel bemenetének (ekkor lehetne a fő órajel a belső RC-oszcillátor). Az előbbi megoldás előnye, hogy a teljesítményfelvétel kisebb lesz, ha egy lassú kristályt használunk fő órajelnek a minimális 1 MHz-es RC-oszcillátor helyett. A második megoldás előnye, hogy a nagyobb működési frekvencia több műveletet tesz lehetővé (például bonyolultabb kijelzőkezelést), és nem kell a fuse-ok beállításain aggódnunk. A példaprogramhoz a második megoldást választottam, ezért a Timer/Counter2 perifériát használjuk majd.

Az ennek megfelelően módosított kapcsolási rajz a 6. ábrán látható. Bár külső kristály használata esetén általában kell tenni egy-egy kondenzátort a kristály két lába és a föld közé, az adatlap 8.9. fejezete alapján ez esetben nem szükséges.



***6. ábra** - Kapcsolási rajz az eszközről külső kristály használata esetén*



7. ábra *- A megépített kapcsolás*

Hogyan nézzen ki az új szoftver?

Az új szoftver koncepciója a következő: induláskor beállítjuk és elindítjuk a timer perifériát, majd egy végtelen ciklusban végzünk el három feladatot: beolvassuk a nyomógombok állását, frissítjük a kijelzőt, és várakozunk, amíg el nem telt egy másodperc. Valahogy így:

```
int main(void)
{
	uint32_t time_seconds = 0;

	IoInit();
	SetupTimer();

	while (1)
	{
		time_seconds = SetTime(time_seconds);

		UpdateDisplay(time_seconds);

		while (CheckIfSecondElapsed() == false)
		{
			// szándékosan üres ciklusmag
		}

		time_seconds++;
		if (time_seconds >= 86400)
		{
			time_seconds = 0;
		}
	}
	return 0;
}
```

Mi történik itt?

Kezdjük a `SetupTimer()` függvénnyel. Ez a függvény beállítja a Timer/Counter2 perifériát, a következő lépések segítségével:

1) Beállítja a timert, hogy bemenetnek a TOSC[1:2] pinek közötti 32 768 Hz-es oszcillátort használja, azzal, hogy bebillenti az AS2 bitet a ASSR regiszterben.
2) Beállítja a prescalert, hogy ossza le a fenti 32 768 Hz-es órajelet az 1024-edére. Így a számláló vezérlőlogikája egy 32 Hz-es órajelet fog számolni.
3) Beállítja a timert Clear Timer on Compare Match (CTC) módba. Ebben a módban a számláló 0 és OCR2 között számol, és minden túlcsorduláskor az OCF2 bit beáll 1-be (lásd az adatlap 17-11. ábráját).
4) Beállítja OCR2-t, azaz a számlálás felső határát 31-re, így a számláló 0 és 31 közötti értékeket fog felvenni, azaz 32 ütemenként fordul át. Ez a 32 Hz-es belső órajellel azt eredményezi, hogy a timer periféria másodpercenként egyszer 1-be állítja az OCF2 bitet. Ezt a flaget később a programunk fogja periodikusan visszaállítani 0-ba.
5) Elindítja a timert.

Az ATmega16A mikrokontrollerben a prescaler beállítása egyben el is indítja a timert (lásd az adatlap 17-6. táblázatát). Ennek megfelelően a 2-es és 5-ös pontokat összevonva a függvény implementációja így nézhet ki:

```
void SetupTimer( void )
{
	// TODO: 1. Timer input beállítása TOSC[1:2] pinekre
	// TODO: 2. Timer mód beállítása CTC módba
	// TODO: 3. OCR2 beállítása
	// TODO: 4. Prescaler beállítása 1024-edes osztásra
}
```

Érdemes megjegyezni, hogy a programot is ilyen sorrendben írom, ahogy te olvasod a részben: először a magas szintű koncepciót tisztázom a fejemben, és írom meg (jelenleg csak a main függvényt), azután az azt felépítő függvényeket (`SetupTimer`, `CheckIfSecondElapsed`) és adatstruktúrákat (ha van ilyen a programban), aztán legvégül átnyálazom az adatlapot, olyan részleteket kutatva, mint hogy a prescalert melyik regiszterrel kell beállítani. Ha bonyolultabb a program, vagy ha egy gondolatnál megakadok, akkor nem szégyellek papírt és ceruzát elővenni, hogy rajzokon vagy diagramokon tisztázzam a gondolataimat.

Az is nagyon sokat segít, ha elmagyarázzuk a program működését egy hozzáértő barátnak vagy munkatársnak: gyakran már a magyarázás közben észrevesszük, hogy a programunk hibás, vagy feleslegesen bonyolult. A legjobb programozók, akiket ismerek, rendszeresen kikérik egy kollégájuk véleményét*, és nem csak akkor, amikor nem működik a szoftverük. (Ez a tevékenység angolul „code review” néven ismert.) A kódunk közös átnézésére a legjobb alkalom közvetlenül a programunk lefordítása után van: ekkor már a fordító megtalálta a legtriviálisabb hibákat, de még nem töltöttünk el időt egy hibás kód felprogramozásával és hibakeresésével.

Hogyan segíthet a kutyánk a szoftverfejlesztésben?

Érdekes, hogy akinek magyarázzuk, nem kell feltétlenül hozzáértő szakembernek lennie, de még csak élő embernek sem. A „rubber duck debugging” módszer lényege, hogy a programunkat sorról sorra elmagyarázzuk egy nekünk kedves gumikacsának: azáltal, hogy szavakba öntjük, mit csinál a programunk, és mit akarjuk, hogy csináljon, a különbség sokkal szembetűnőbb lesz. A legjobb programozókra amúgy is őrült zseniként tekintenek, ezen a megítélésen már mit sem változtat, hogy egy gumiállattal beszélget az illető.

Na, vissza a programhoz! Szintén a timerrel foglalkozik a `CheckIfSecondElapsed()` függvény: ennek a függvénynek `true` értékkel kell visszatérnie, ha a timer elérte a beállított maximális értéket (azaz ha eltelt egy újabb másodperc), illetve `false` értékkel, ha még nem.

A timer periféria használatának az előnye, hogy ez a függvény nagyon egyszerű: a periféria attól függetlenül működik a háttérben, hogy a mikrokontroller milyen műveletet hajt végre. Abban a pillanatban, hogy eltelt egy egész másodperc, a hardver bebillenti az OCF2 flaget a TIFR regiszterben (Timer Interrupt Flag Register), és ez a flag addig ebben az állapotban marad, amíg a szoftver vissza nem állítja 0 értékbe. Ezt az adatlap 17-11. ábrája és az OCF2 bit leírása a 17.11.6. szakaszban árulta el. A TIFR

regiszternek az értéke furcsa módon változtatható: egy bit értékét úgy nullázhatjuk, ha 1-et írunk bele; ha 0-t írunk, az nem módosítja a bit értékét.

```
bool CheckIfSecondElapsed(void)
{
	bool second_elapsed_b;

	if (0 != (TIFR & (1 << OCF2)))
	{
		// Reseteljük az OCF2 bitet
		TIFR = (1 << OCF2);

		second_elapsed_b = true;
	}
	else
	{
		second_elapsed_b = false;
	}

	return second_elapsed_b;
}
```

Miért jó, hogy egy külön `second_elapsed_b` változót használtam, ahelyett hogy két helyen tértem volna vissza a függvényből két return utasítással?
Bár a fenti egyszerű függvény amúgy is érthető lett volna, általánosságban véve könnyebb egy olyan függvényt megérteni, debuggolni és karbantartani, amelyiknek csak egy visszatérési helye (angolul *„exit point”*) van. A saját projektjeid befejeztével figyeld meg, hogy szoftverfejlesztéskor nagyon sok idő megy el a hibák keresésével és kijavításával.

Mit jelent a „_b” a `second_elapsed_b` változó végén?
Mértéktelen fejfájástól kíméljük meg magunkat (és azokat, akik később meg szeretnék érteni a programunkat), ha a változók nevébe beleírjuk azoknak a mértékegységét. A kétértékű igaz-hamis (boolean) változókat sokan szokták „`_b`”-vel jelölni: ebből első pillantásra látszik, hogy a `second_elapsed_b` változó `true` vagy `false` értéket vehet fel, attól függően, hogy egy másodperc eltelt-e vagy sem. Érdekesség, hogy a NASA űrszondája, a Mars Climate Orbiter egy így megelőzhető hiba miatt veszett kárba. Ott az egyik szoftvermodul angolszász mértékegységben adta vissza a hajtómű által generált lendületet, szemben az előírt SI mértékegységgel. Így égett el egy 193 millió dolláros fejlesztés eredménye a Mars légkörében.

Az `UpdateDisplay()` függvény ugyanaz, mint az előző programban volt, ugyanis a függvényt úgy írtuk meg, hogy az független legyen az időzítés mikéntjétől. Másrészről így ha később a LED-ek helyett egy 7-szegmenses kijelzőt szeretnénk használni, vagy soros portra szöveges formátumban kiírni az időt, akkor csak kevés helyen kell majd változtatni a szoftverben. Ha egy másik projektben ugyanezt a funkciót szeretnénk használni, akkor könnyű lesz nagyobb programrészeket átemelni. (Feltéve, hogy azok jól vannak dokumentálva: mi emberek hajlamosak vagyunk a memóriánkat túlértékelni, és azt hinni, hogy a döntéseink okára helyesen fogunk emlékezni akár több év távlatából is. Ez gyakran nem így van, ezért

megkönnyítjük a jövőbeli énünk vagy kollégáink dolgát, ha a furfangosabb döntéseinknél leírjuk, mit miért csináltunk.)

A tananyag honlapján van a példakódnak egy olyan változata, ami még kiegészítésre vár. Töltsd ki a „TODO” megjegyzéssel jelölt hiányzó részeket, töltsd fel a mikrokontrollerre a programot, javítsd ki az esetleges hibákat, és gyönyörködj a még menőbb kütyüdben!

ÖSSZEFOGLALÁS

Ebben a tananyagrészben a következőkről tanulhattál:

- milyen nehézségeink vannak az időzítéssel a timer periféria nélkül,
- hogyan használjuk a timer perifériát,
- hogyan könnyítsük meg a szoftverfejlesztést,
- mik egy jó mérnök ismérvei: képes a problémák megfogalmazására és azok megoldására, és képes beismerni a saját hibáit.

A timer perifériát ebben a részben időzítő módban használtuk. A következőkben arról lesz szó, hogyan alkalmazhatod PWM-jelek generálására.