Osnove mikroprocesorske elektronike

Priprava 13: Igrica – prvič

Pri tej vaji prenehamo s programiranjem na sistemskem nivoju in se lotimo izdelave *aplikacije*. V našem primeru bo aplikacija kar igrica. Zgradbo in delovanje igre kot aplikacije lahko razumemo nekako takole:

1) aplikacija potrebuje nekakšne *podatkovne strukture*, kjer hrani vse potrebne informacije za svoje delovanje,
2) *algoritmi* obdelujejo te podatke na točno določen način
3) in rezultate prikazujejo na *uporabniškem vmesniku* (zaslonu),
4) pri čemer pa uporabnik komunicira z algoritmom aplikacije preko *vhodno-izhodnih naprav* (angl. input-output devices).

Vidite, da lahko aplikacijo v grobem razdelimo nekako na štiri sestavne dele. Namen te priprave je, da se spoznate s temi sestavnimi deli, da boste nato lahko sami sprogramirali igro do polne funkcionalnosti.

Poglejmo, kako smo v našem primeru organizirali te sestavne dele igre:

1) modul `objects.c` bo poskrbel za *definicijo* in *manipulacijo* vseh *objektov* (tj. podatkov), ki jih igra potrebuje za svoje delovanje,
2) modul `game.c` bo poskrbel za *implementacijo poteka in pravil* igre, pri čemer bomo algoritem igre implementirali v obliki *avtomata stanj* (angl. state machine),
3) modul `graphics.c` bo poskrbel za izris vseh objektov na uporabniški vmesnik,
4) za komunikacijo med uporabnikom in aplikacijo pa bodo poskrbeli *moduli sistemskega nivoja*: `led.c`, `kbd.c`, `joystick.c` ter seveda `led.c`.

V pripravi spodaj boste najprej popravili nekaj malenkosti obstoječega projekta, ki ste ga razvijali tekom preteklih vaj. Nato boste preučili program, ki vam bo demonstriral smisel in uporabo modulov `objects.c` in `graphics.c`. V drugem delu priprave pa boste preučili še avtomat stanj, s pomočjo katerega bomo implementirali "intro" zaslon in "game over" zaslon igre.

Priprava – popravki obstoječega projekta in dodatki

Tekom razvoja sistemskih modulov se nam je prikradlo nekaj napak, ki se pokažejo šele kasneje, ko se lotimo implementacije aplikacije. In nekaj teh napak boste odpravili sedaj.

1. ***S pomočjo orodja*** `STM Cube Project Copy` ***ustvarite nov projekt z imenom***

   `VAJA_13-game`

   na podlagi projekta, kjer ste rešili prejšnjo vajo. Projekt nato uvozite v "STM Cube IDE".

2. ***Modulu*** `kbd.c` ***dodajte funkcijo*** `KBD_flush()`***, ki poskrbi, da se izprazni medpomnilnik tipkovnice.***

   Tako dobimo možnost, da *"počistimo" vse odvečne pritiske tipk*, ki je izvedel uporabnik aplikacije. Vsebina funkcije je zelo preprosta: uporabi se BUF_ funkcijo za praznjenje medpomnilnika,

   ```
   void KBD_flush(void)
   {
   	BUF_flush(&kbd_buf_handle);
   }
   ```

   Ne pozabite v zglavni .h datoteki dodati tudi prototip te nove funkcije.

3. ***Modulu*** `timing_utils.c` ***dodajte funkcijo*** `TIMUT_get_stopwatch_elapsed_time()`***, s katero bomo lahko dobili informacijo o pretečenem času od postavitve časovnega zaznamka.***

   ```
   uint32_t TIMUT_get_stopwatch_elapsed_time(stopwatch_handle_t *stopwatch)
   {
   	TIMUT_stopwatch_update(stopwatch);

   	return stopwatch->elapsed_time;
   }
   ```

   Ne pozabite v zglavni .h datoteki dodati tudi prototip te nove funkcije.

4. ***Zakomentirajte registracijo funkcije za strojno pospešeno risanje polnih pravokotnikov na LCD zaslon z*** `uGUI`.

   Implementacija strojno pospešene funkcije je napačna in jo bomo popravili kasneje. Zaenkrat pa jo lahko odstranimo. Zakomentirajte kodo v modulu `lcd.c`, ki je označena na sliki spodaj.

```
// Inicializacija uGUI knjižnice za delo z našim LCD zaslonom.
void LCD_uGUI_init(void)
{
    // Inicializacija uGUI knjižnice: registracija funkcije za izris enega piksla na zaslon,
    // specifikacija resolucije zaslona.
    UG_Init(&gui, UserPixelSetFunction, ILI9341_GetParam(LCD_WIDTH), ILI9341_GetParam(LCD_HEIGHT));

    // Nastavitev "default" fontov in barv za besedilo in ozadje.
    UG_FontSelect(&FONT_8X12);
    UG_SetForecolor(C_WHITE);
    UG_SetBackcolor(C_BLACK);

    // Registracija funkcij za izris pravokotnika.
    UG_DriverRegister(DRIVER_FILL_FRAME, (void *)_HW_FillFrame_);
    UG_DriverEnable(DRIVER_FILL_FRAME);
}
```

5. ***Odstranite demonstracijo avtomatskega skeniranja stanja tipkovnice znotraj*** periodic_services.c.

   Potrebno je odstraniti demo funkcijo, ki se odziva na zaznane pritiske tipk (glejte spodaj). Če bi to funkcijo pustili v sklopu periodičnih rutin, bi nam ta demo funkcija "požrla" vso informacijo o pritisnjenih tipkah in ta informacija sploh ne bi prišla do aplikacije.

```
void PSERV_run_services_Callback(void)
{
    // here we execute all the low-priority periodic services
    KBD_scan();

    // respond to the buttons being pressed -->
    // --> after the wake-up toggle the appropriate LEDs
    KBD_demo_toggle_LEDs_if_buttons_pressed();

}
```

6. ***V funckijo za inicializacijo "joysticka"*** `JOY_init()` ***vnesite izmerjene kalibracijske parametre.***

   Ker se kalibracijski parametri "joyticka" s časom spreminjajo razmeroma počasi, se jih lahko enkrat izmeri in nato zabeleži za nadaljno uporabo.

   Torej, s pomočjo terminala odčitajte kalibracijske parametre vašega "joysticka" (primer spodaj),

```
\n
******** CALIBRATION IN PROGRESS ********\n
--- Joystick status ---\n
X_RAW = 1924\n
Y_RAW = 1900\n
X_RAW_MIN = 820\n
X_RAW_MAX = 3101\n
Y_RAW_MIN = 739\n
Y_RAW_MAX = 3100\n
X_RAW_RANGE = 2281\n
Y_RAW_RANGE = 2361\n
X = 48\n
Y = 49\n
\n
```

   ter jih nato uporabite pri funkciji za inicializacijo `JOY_init()` (primer spodaj).

```
// Smiselno nastavimo začetne ekstremne vrednosti pozicije "joysticka".
joystick.position_raw_min[X] = 820;
joystick.position_raw_min[Y] = 739;

joystick.position_raw_max[X] = 3101;
joystick.position_raw_max[Y] = 3100;

joystick.position_raw_range[X] = joystick.position_raw_max[X] - joystick.position_raw_min[X];
joystick.position_raw_range[Y] = joystick.position_raw_max[Y] - joystick.position_raw_min[Y];
```

Priprava – preučitev uporabe modulov `objects.c` in `graphics.c`

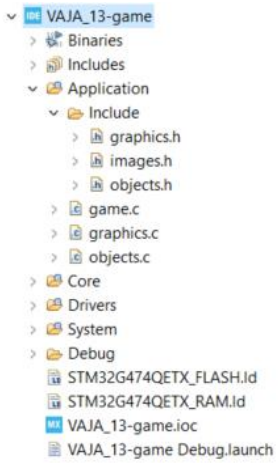
Sedaj boste s pomočjo demonstracijskega programa preučili smisel in uporabo modulov za manipulacijo podatkovnih struktur aplikacije ter izris objektov na zaslon.

7. ***V projektno mapo razpakirajte arhiv*** `Application.zip` (mapa "predloga" na eFE).

   Poskrbite, da se datoteka `Application.zip` ne bo nahajala v projektni mapi.

   Poskrbite, da se nova mapa `Application` ter podmapa `Include` dodata projektu, podobno kot ste to naredili pri 4. laboratorijski vaji za mapo `System`.

   Vidite, da sedaj res prehajamo na aplikacijski nivo. Pri tem skrbimo za urejenost in ločitev datotek.



8. ***Vključite oba modula v glavni program*** `main.c` ***in pripravite vse potrebno, da se demonstracijska funkcija lahko zažene***

   Dodati je potrebno

```
#include "objects.h"
#include "graphics.h"
```

   Pred glavno `while(1)` zanko pa pokličite

```
GFX_demo();
```

   Predlagam, da demonstracijske funkcije še ne zaženete takoj, pač pa se najprej lotite pregleda kode te funkcije. Tako si boste najprej s premišljanjem ustvarili predstavo, kako naj bi ta modul deloval. Ko boste pa nato zagnali demo funkcijo, pa boste lahko še preverili, ali je vaša predstava prava ter razčistili morebitne napačne predstave.

9. ***Preučite vsebino demo funkcije*** `GFX_demo()` ***ter preostalih modulov aplikacijskega nivoja:*** `objects.c`, `graphics.c` ***ter*** `images.h`.

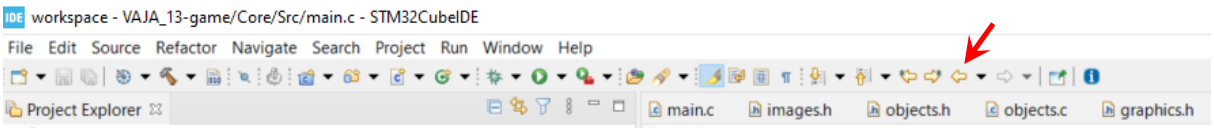
   Pridobiti si želite okvirno razumevanje, kaj se najaja v teh modulih ter kakšno vlogo igrajo ti moduli pri implementaciji demo funkcije. Razumevanje principa implementacije ter pomena izpostavljenih funkcij bo ključno, ko se boste sami lotili programiranja igre.

   Nasvet: kodo lahko zelo enostavno raziskujete v globino tako, da držite tipko CTRL in nato kliknete na funkcijo ali spremenljivko, ki vas zanima. "CTRL + klik" vas bo usmeril na mesto, kjer so te stvari definirane. Poglejte primer spodaj.

```
// --------- Application demo -----

GFX_demo();
while (1);
```

   Če se pa želite vrniti nazaj na mesto, kjer se kliknili na hiper-link, pa uporabite tole puščico:



10. ***Ko si boste razjasnili, kaj naj bi demo funkcija storila, pa kočno zaženite program in preučite izris na zaslonu.***

    Poskrbite, da vam bo jasno, katera koda poskrbi za izris posameznih objektov na zaslonu.

    Mimogrede: opazili boste, da izrisana slika nekoliko utripa. Do tega pojava pride zaradi kombinacije periodičnega osveževanja LCD zaslona ter krmiljenja LED osvetlitve zaslona s pravokotnim PWM signalom. V naslednjem koraku boste to težavo preprosto odpravili z dvigom frekvence PWM signala.

11. ***Odpravite nezaželjeno utripanje slike na LCD zaslonu tako, da dvignete frekvenco PWM signala za krmiljenje osvetlitve LCD zaslona*** s 100 Hz na 1 kHz.

    To boste storili tako, da boste zgolj spremenili vrednost delilnika ure časovnika `TIM4` ("prescaler"). Ne smete pa spreminjati modula časovnika (tj. "counter period"); ta mora ostati enak 99.

Priprava – preučitev avtomata stanj za implementacijo igre

Kadar je potrebno sprogramirati napravo, da *izvaja nekakšen zapleten postopek*, ki ima svoja *pravila*, se ta problem lahko zelo elegantno reši s pomočjo avtomata stanj.

Avtomat stanj je v splošnem *sistem*, ki na podlagi *vhodnih podatkov* in *trenutnega stanja sistema* prehaja v *nova stanja* in pri tem tvori ustrezne *izhodne podatke*. Prehodi med stanji avtomata so določeni s *pravili* (tj. logiko) avtomata. Tak opis avtomata stanj je precej abstrakten in se zato sprva zdi, da je uporaba takega koncepta v programiranju zgolj nepotrebna komplikacija. Izkaže pa se ravno nasprotno – v marsikaterem primeru lahko z uporabo koncepta avtomata stanj zelo elegantno in enostavno rešimo precej zapletene probleme!

Pomislite: s tem, ko avtomat stanj prehaja v nova stanja, si je avtomat stanj *na nek način* pravzaprav *sposoben zapomniti, kaj vse pomembnega se je že zgodilo* – sicer se ne bi nahajal v stanju, v katerem se nahaja sedaj! In če se sistem "zaveda", kaj vse se je že zgodilo, je potem zelo enostavno poskrbeti, da ta sistem *pravilno "ukrepa" v trenutni situaciji* – da torej tvori ustrezen odziv! In ko tak sistem implementiramo, ga lahko *implementiramo po delih* tako, da tvorimo ustrezen odziv za vsako situacijo – stanje posebej. To pa pravzaprav pomeni, da smo *kompleksen problem razstavili* na več manjših podproblemov, ki pa so običajno razmeroma enostavni za implementacijo. Pomeni pa tudi, da mora avtomat stanj v vsakem trenutku-situaciji-stanju običajno opraviti le neko kratko nalogo, kar pa povzroči, da se avtomati stanj lahko izvajajo postopoma in hitro. To pa dalje pomeni, da lahko izvajamo več avtomatov stanj enega za drugim in tako enostavno ustvarimo *učinek hkratnega izvajanja večih opravil*!

Prav zaradi zgoraj navedenih možnosti so avtomati stanj zelo močno orodje pri *realizaciji procesov*, kjer je *postopek poteka* procesa *točno določen in znan*, kot na primer implementacija komunikacije po standardnem protokolu, krmiljenje parkirišče zapornice, implementacija uporabniškega vmesnika, algoritem, ki upravlja kavomat itd. V primeru naše laboratorijske vaje pa bomo idejo avtomata stanj uporabili pri programiranju igre.

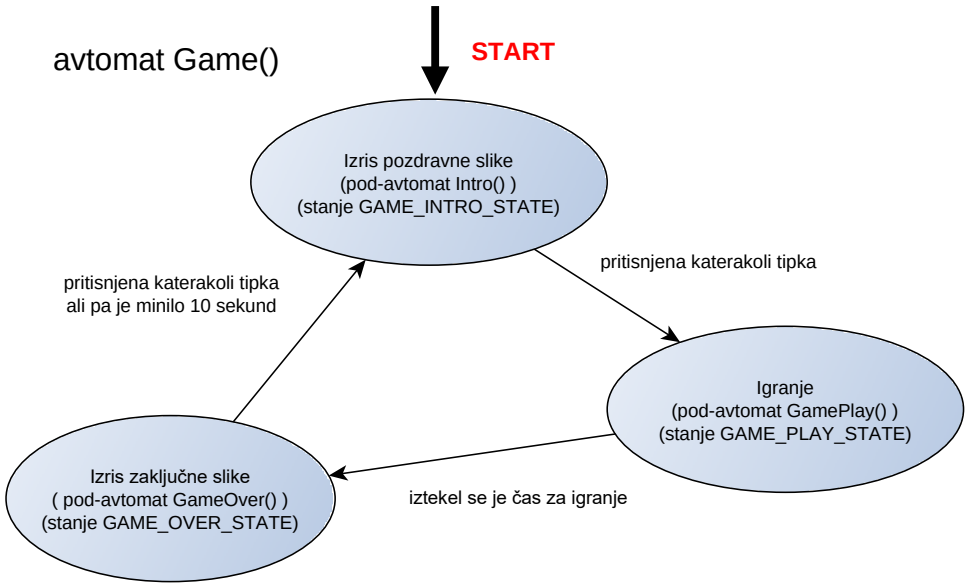
Potek naše igre lahko v grobem opišemo takole:

1) ob zagonu igre se bo prikazal *pozdravni zaslon* igre (angl. splash screen) in prižgale se bodo vse LEDice. Izpisal se bo napis "Press any key to continue". Ob pritisku na katerokoli tipko se bodo LEDice ugasnile in pričelo se bo

2) *igranje igre*. Ko se bo čas za igranje iztekel, se bo

3) igra končala in prikazala se bo *zaključna "game over" slika*. Počakamo 3 sekunde, da igralec dojame situacijo. Nato se pa ob pritisku katerekoli tipke oziroma po 10-tih sekundah vrnemo nazaj na začetek igre v pozdravni zaslon.

Zgornji potek igre smo opisali z besedami. Zelo jasno pa ga lahko prikažemo s pomočjo sheme, ki ji pravimo *diagram poteka* (angl. flow chart). Poglejte osnovni diagram poteka za našo igro na sliki spodaj.



Sedaj pa s pomočjo diagrama poteka razmišljamo in opazujemo dalje:

a) avtomat stanj se *nikoli ne konča* oziroma zaključi, saj je nekako cikličen. To je smiselno, saj si želimo, da je igra vedno na voljo.

b) Diagram namiguje, da bomo avtomat za implementacijo igre *implementirali v obliki funkcije* `Game()`.

c) V treh stanjih igra *izvaja določena opravila*: bodisi izris slik bodisi "igranje igre".

d) Posamezna stanja očitno predstavljajo nekakšna *pod-opravila* igre.

e) **Ideja:** kaj če bi tudi ta pod-opravila implementirali v obliki avtomata stanj?

f) Tako dobimo nekakšne *ugnezdene avtomate* in problem elegantno razbijamo dalje na vse manjše in enostavnejše naloge! Tudi te bomo implementirali kot funkcije: `Intro()`, `GamePlay()` ter GameOver().

g) Ti trije *pod-avtomati* pa se od *nad-avtomata* `Game()` očitno morajo razlikovati v tem, da se v neki točki *zaključijo* (npr. izteče se čas za igranje).

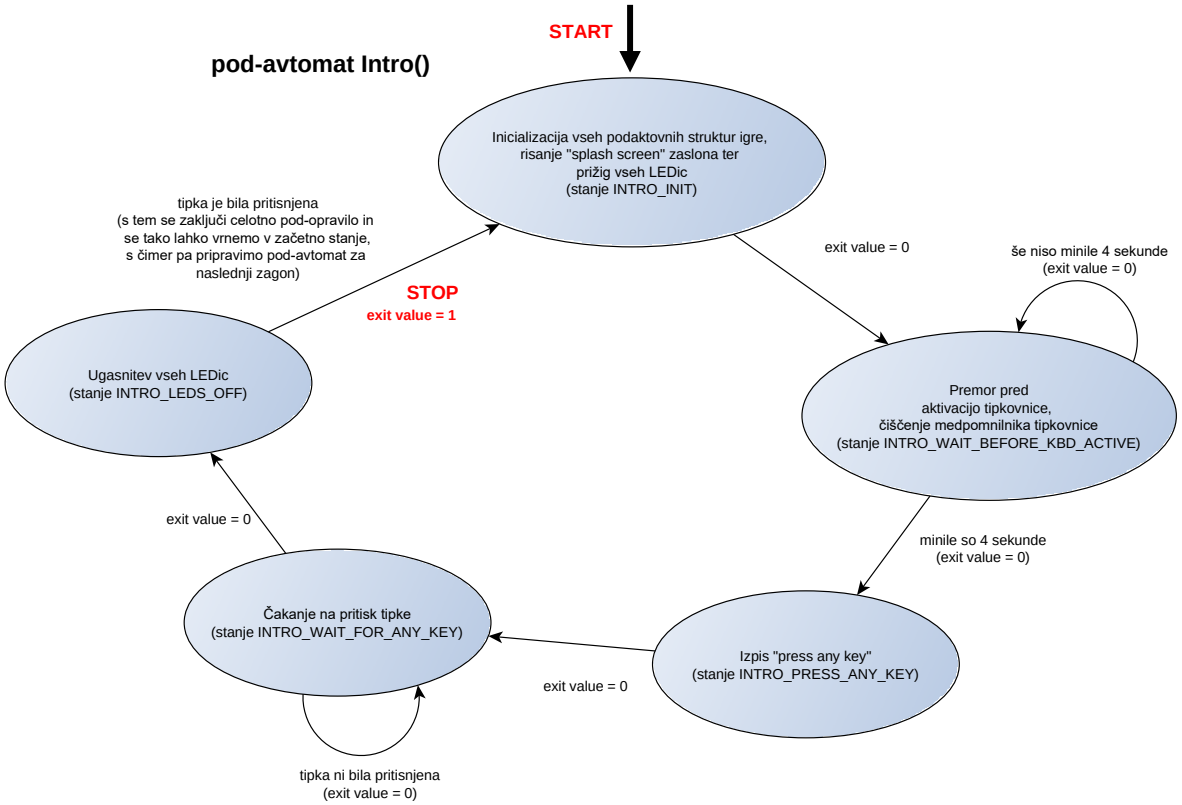
h) Ko se pod-avtomat zaključi, mora to nekako sporočiti nad-avtomatu `Game()`, da ta nato preide v novo stanje in prične z izvajanjem naslednjega pod-avtomata.

i) To pa pomeni, da moramo bomo pod-avtomate definirali kot *funkcije, ki vračajo vrednost* (angl. return exit value).

Premislite zgornja opažanja, saj vam bodo prišla prav pri programiranju avtomata stanj oziroma igrice.

12. Preučite diagrama poteka pod-avtomatov `Intro()` ***in*** `GameOver()`**.**

Ti dva avtomata boste namreč v sklopu vaje sprogramirali sami.



Opazite lahko, da je tudi za avtomat pravzaprav cikličen. ***Vendar se ne izvaja ciklično neprenehoma***. Ko pod-avtomat `Intro()` opravi vse svoje delo (konča delo v stanju `INTRO_PRESS_ANY_KEY`) , se ***izvajanje pod-avtomata zaključi***. Pri tem se zgodita dve pomembni stvari:

1) nad-avtomatu `Game()` se vrne "`exit value = 1`", s čimer pod-avtomat sporoča, da se je končal svoja opravila ter
2) zgodi se prehod nazaj v začetno stanje. S tem poskrbimo, da ko bomo naslednjič zagnali pod-avtomat `Intro()`, bomo pričeli izvajanje v pravem, začetnem stanju!

Preučite še pod-avtomat `GameOver()` spodaj.