Uvod u razvoj programa za Contiki OS

Seminarski rad, Bežične Mreže Osjetila, Ak. God. 2019/2019
Mentor: doc. dr. sc. Damir Arbula
Dražen Milković, Erik Smoljan, Sandi Šegota

Contiki OS je software otvorenog koda namjenjen za korištenje na Internet of Things platformama - za povezivanje malenih, jeftinih i ne energetski zahtjevnih mikrokontrolera sa internetom. Contiki pruža niskoenergetsku internetsku komunikaciju, putem IPv6 i IPv4, korištenjem 6lowpan, RPL i CoAP protokola.

Aplikacije za Contiki su pisane u C programskom jeziku. U ovom tekstu će biti ukratko opisani osnovna struktura i način pisanja programa za Contiki operacijski sustav. Cilj ovog teksta nije objašnjavanje načina izrade programa korištenjem specifičnog protokola, već objašnjenje osnovnih ideja. Za specifičnije primjere preporuča se pregled izvora u literaturi i primjera isporučenih unutar ContikiOS repozitorija. Kao prilog ovom tekstu daju se uputstva za postavljanje razvojnog okruženja, izradu slike za TI CC2650 mikrokontroler i postavljanje slike na isti mikrokontroler, te primjer programskog koda koji implementira Ping-Pong protokol, uz generirani firmware i sve potrebne datoteke za generiranje firmwarea(project-conf.h, Makefile).

1. Minimalni primjer - Hello World

U ovom dijelu dati ćemo primjer koji na konzoli ispisuje “Hello World!”, te zatim pali i gasi led diodu do zaustavljanja programa.

```
/*---------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------*/

PROCESS_THREAD(hello_world_process, ev, data) {
    PROCESS_BEGIN();
    printf("Hello World!\n");
    while(1){
        led_toggle(LEDS_RED)
    }
    PROCESS_END();
}
```

Početak programa sadržava definiciju procesa koju pokrećemo programom. U ovom slučaju definiramo process hello_world_process, sa nazivom “Hello world process”. Naredbom `AUTOSTART_PROCESSES(&hello_world_process);` pokrećemo definirane procese.

Što će se događati u procesu definiramo u PROCESS_THREAD funkciji. Ovo možemo shvatiti kao analog int main() funkciji jednog procesa. `PROCESS_THREAD(hello_world_process, ev, data)` definira kojem procesu pripada ovaj thread - ime procesa je prvi argument. Drugi i treći argument su varijable koje služe reagiranju na događaje (event) koje poziva operativni sustav.

Prva naredba u ovoj funkciji je `PROCESS_BEGIN();`. Ova naredba ne mora nužno biti prva - moguće je deklarirati varijable prije nje, ali sva komunikacija, ispis i kontrola događaja/procesa se mora događati nakon poziva `PROCESS_BEGIN();`. Slično tomu `PROCESS_END();` je posljednja naredba u procesu i ona označava kraj procesa, te omogučava oslobađanje resursa od strane operativnog sustava. Između ove dvije funkcije izvodimo sve ostale funkcije koje želimo da naš program obavlja.
U tom dijelu imamo primjer ispisa na standardni izlaz. Ovo se izvodi printf() funkcijom iz standardne C knjižnice. Korisna napomena vezana uz ispis - u većini firmwarea, unutar project_conf.h nalazi se ovakva struktura:

```
#define DEBUG 0
#if DEBUG
#define PRINTF(...) printf(__VA_ARGS__)
#else
#define PRINTF(...)
#endif
```

Promjenom vrijednosti DEBUG u 1 omogućavamo ispis debug informacija, što očito može pomoći kod debugiranja.

Slijedeći dio koda je while petlja. Unutar te petlje nalazi se naredba `leds_toggle(LEDS_RED).` Petlja je standardni dio C programskog jezika, i petlja definirana sa `while(1)` će se beskonačno izvoditi, ako program nije prekinut vanjskom intervencijom. Naredba `leds_toggle()` je dio Contiki sustava i definirana je u `dev/leds.h` header datoteci, zajedno sa `LEDS_RED` makrom koji definira pristup crvenoj LE diodi na mikrokontroleru (ako postoji). Funkcija `leds_toggle()` će mijenjati stanje LE dioda - iz ugašene u upaljene i obrnuto.
Ovime možemo zaključiti da će gornji primjer nakon pokretanja u konzoli ispisati “Hello World!”, te zatim kontinuirano paliti i gasiti crvenu LE diodu.

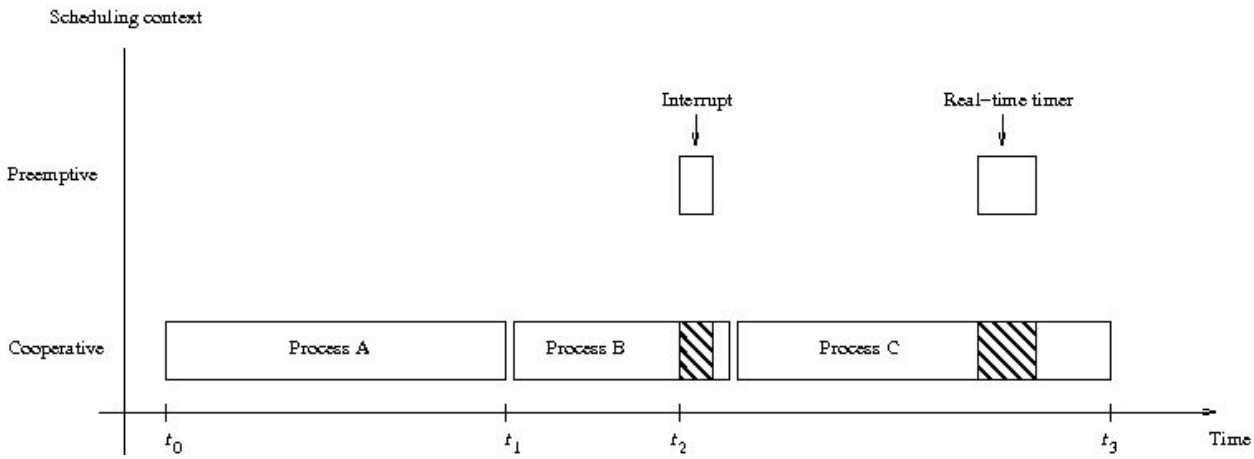
Kako bismo pokrenuli program, potrebno ga je sastaviti - a za to je potrebno napisati Makefile. Makefile za ovaj kod će izgledati:

```
CONTIKI_PROJECT = hello-world
all: $(CONTIKI_PROJECT)
DEFINES+=PROJECT_CONF_H=\"project-conf.h\"
CONTIKI = ../..
include $(CONTIKI)/Makefile.include
```

U prvoj liniji definiramo koju aplikaciju kompajliramo. Naredba all: će kompajlirati sve definirane aplikacije. Sa `DEFINES+=PROJECT_CONF_H=\"project-conf.h\"` definiramo uključivanje datoteke project-conf.h. Slijedeći dio je definiranje CONTIKI vrijednosti - a to je putanja do korijena contiki sustava. Naredbom `include $(CONTIKI)/Makefile.include` uključujemo glavni Contiki Makefile koji nam omogućava kompilaciju sustava za ContikiOS.

2. Procesi

Svi Contiki programi su procesi - gdje je proces definiran kao kod koji se regularno izvodi od strane Contiki sustava. Procesi se u pravilu pokreću kada se sustav podigne ili kada je učitan modul koji sadrži proces.
Procesi mogu biti u jednom od dva konteksta Cooperative(npr. Ranije opisan “Hello World” proces) ili Preemptive (npr timeri, prekidi...). Kooperativni kod se izvršava sekvencijalno sa drugim procesima, dok preemptivni proces može prekidati kooperativne u bilo kojem trenutku, te kooperatrivni neće nastaviti sa izvršavanjem do kraja izvršavanja preemptivnog procesa - ilustrirano na slici 1.



Slika 1. Preemptivni i kooperativni procesi.

Procesi se sastoje od kontrolnog bloka i procesnog threada. Procesni Thread je kod procesa i pohranjen je u ROMu. Kontrolni blok je pohranjen u RAMu i sadržava runtime informacije o procesu.

2.1. Kontrolni blok

Kontrolni blok sadrži informacije o:

- Nazivu procesa
- Statusu procesa
- Pokazivaču na Process Thread
- Slijedećem procesu u liniji za izvršavanje

Kontrolnom bloku se ne pristupa iz user spacea, nego se koristi od strane kernela. Kontrolni blok je struktura koja se definira i kreaira kroz `PROCESS` makro i poziva se u kodu na sljedeći način:

```
PROCESS(hello_world_process, "Hello world process");
```

Izgled strukture kontrolnog bloka je:

```
struct process {
   struct process *next;
   const char *name;
   int (* thread)(struct pt *,
                  process_event_t,
              process_data_t);
   struct pt pt;
   unsigned char state, needspoll;
 };
```

2.2. Procesni thread

Kao što je rečeno, procesni thread sadrži programski kod. Definira se makrom `PROCESS_THREAD` i sadrži kod koji se izvršava kao i posebne naredbe vezane uz ponašanje procesa. Te naredbe su:

```
PROCESS_BEGIN(); // početak protothreada
PROCESS_END(); // kraj protothreada
PROCESS_EXIT(); // izlazak iz procesa
PROCESS_WAIT_EVENT(); // čekaj na bilo koji događaj
PROCESS_WAIT_EVENT_UNTIL(); // čekanje na događaj sa uvjetom
```

```
PROCESS_YIELD(); // ekvivalentno PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL(); // Čekaj na uvjet, ne prepušta nužno izvršavanje
PROCESS_PAUSE(); // Privremeno zaustavi proces
```

2.2.1. Protothread

Protothread je način strukturiranja koda koji dopušta izvršavanje slično izvršavanju sa threadovima - ali sa manje overheada što je bitno za sustave sa malo memorijskih resursa, poput IoT moteova. Protothread je definiran kao C funkcija, koja počinje makroima `PT_BEGIN()` i `PT_END()`, te može izvršavati posebne naredbe poput `PROCESS_YIELD()` i ostalih spomenutih ranije.

2.3. Događaji

Uz procese, bitno je u kontekstu ContikiOS spomenuti i događaje. U ContikiOS procesi se pokreću kada prime događaj. Procesnom threadu se, uz naziv, daju i dva argumenta - `ev` i `data`. Od ta dva argumenta ev sadrži tip događaja - tj. identifikator događaja , a data ikakve podatke koji mogu biti poslani uz događaj. Identifikatori događaja događaja su definirani:

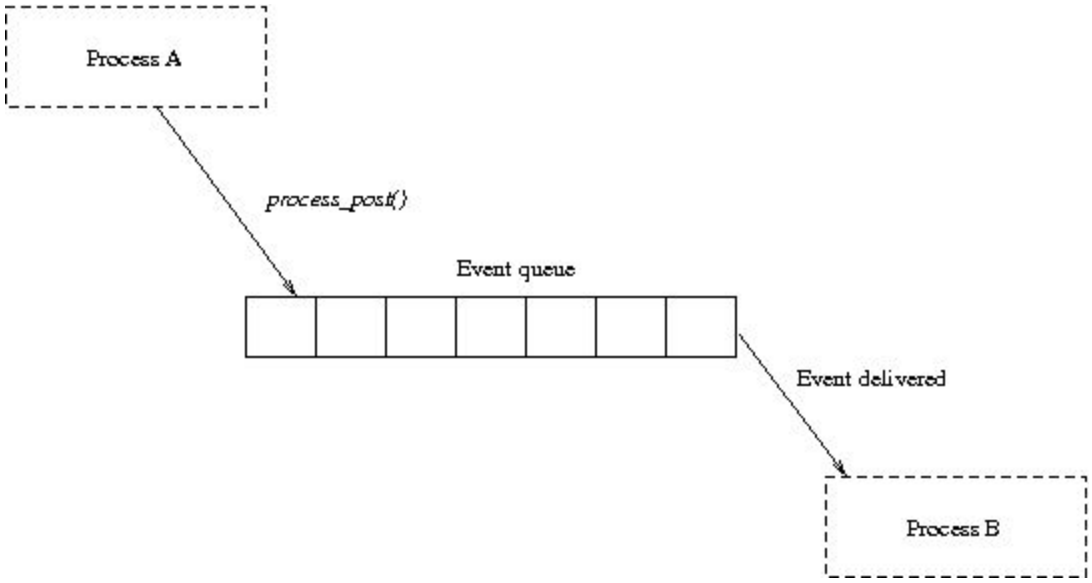
```
#define PROCESS_EVENT_NONE              128
#define PROCESS_EVENT_INIT              129
#define PROCESS_EVENT_POLL              130
#define PROCESS_EVENT_EXIT              131
#define PROCESS_EVENT_CONTINUE          133
#define PROCESS_EVENT_MSG               134
#define PROCESS_EVENT_EXITED            135
#define PROCESS_EVENT_TIMER             136
```

Postoje asinkroni i sinkroni događaji.

2.3.1. Asinkroni događaji

Kada se asinkroni događaj pošalje funkcijom `process_post()` kernel ga postavlja u queue te će ga dostaviti događaju, pozivajući taj događaj po redu primanja. Funkcija `process_post()` provjeri ima li mjesta u queue - ako ima mjesta postavi događaj u red, a ako nema mjesta vrati pogrešku. Očito, asinkroni događaj nije nužno dostavljen u isto vrijeme kada je poslan. Asinkroni događaj može biti poslan jednom ili svim procesima. Ako je poslan svim procesima,

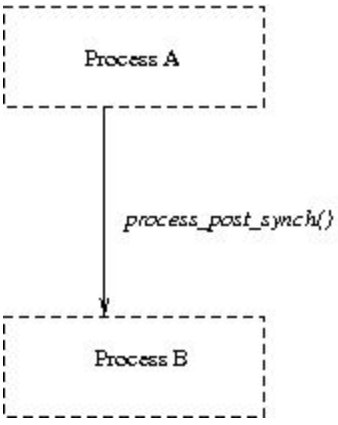
događaj se procesima dostavlja sekvencijalno. Ilustracija slanja asinkronog događaja je vidljiva na slici 2.



Slika 2. Ilustracija asinkronog događaja

2.3.2. Sinkroni događaji

Sinkroni događaji se dostavljaju čim su poslani. Sinkroni događaji se mogu dostavljati samo određenom procesu. Pošto se dostavljaju odmah prilikom poziva, sinkore evente se može smatrati ekvivalentnima pozivu funkcije. Proces kojem je poslan događaj je automatski pozvan, a proces koji je poslao događaj ostaje blokiran dok pozvani proces ne završi sa obradom događaja. Slanje sinkronog događaja je ilustrirano na slici 3.



Slika 3. Ilustracija sinkronog događaja

4. Problemi tijekom izrade projekta

4.1. Virtualna mašina

Za potrebe ovog projekta koristili smo virtualnu mašinu Instant Contiki koja je dostupna na službenoj stranici ContikiOS-a. Dolazi sa svim alatima potrebnim za kompilaciju Contiki-a na podržanim mikrokontrolerima, kao i Cooja simulator. Dolazi i sa softverom za postavljanje slika ali ne podržava modul koji je korišten na ovom projektu. Moguće je skinuti softver od Texas Instrumentsa no on je namijenjen 64-bitnim sustavima a virtualka je 32-bitna. Zato je postavljanje slike sustava rađeno van virtualke. Ideja je bila napraviti prijenos slike pomoću shared foldera koje nudi VMware Workstation player no ta funkcionalnost u virtualci nije radila. Ali bilo je moguće u virtualnoj mašini mountati USB na koji se onda stavljao program.

4.2. Softver za postavljanje slika

Za postavljanje slike moguće je koristiti *Uniflash*(dostupan za Linux, macOS i Windows) ili *SmartRF Flash Programmer 2*(dostupan za Windows). Prvo smo pokušali koristi Uniflash ali nije bio stabilan i pomoću njega nismo uspjeli staviti sliku na sustav. Pomoću *SmartRF Flash Programmer 2* uspjeli smo postaviti sliku koja se nalazila na USB-u, no kada smo sliku postavili na C volume računala program nije mogao otvoriti datoteku kao što vidimo na SLICIX.

Tražili smo rješenje tog problema na internetu ali nismo uspjeli naći odgovor, također smo probali pokrenuti *SmartRF Flash Programmer 2* kao administrator ali to nije riješilo problem. Kada bi se slika sustava stavila na drugi volume(npr D:) nije bilo problema sa postavljanjem slike

4.3. Timeri

Zbog početnog slabijeg razumijevanja rada Contiki sustava sa događajima imali smo problema sa kontrolom toka programa, prilikom korištenja makroa za čekanje na događaj i timera u kombinaciji. Kao primjer, korišteni makro WAIT_FOR_EVENT() je reagirao na ranije postavljeni timer u programskom kodu. Problem je riješen restrukturiranjem koda, nakon boljeg proučavanja načina rada procesa i događaja.

4.4. Unicast

Originalno smo naš zadatak željeli riješiti korištenjem Unicast metode i UDP protokolom. No, primijetili smo da dolazi do problema kod podizanje pločice i ne pronalaska servisa na drugoj pločici. Pretpostavljamo da je dio problema bio u činjenici da se servis nije startao zbog loše usklađenih događaja, ili da pločica koja je inicijator kriovremeno pokušava pronaći servis. Problem smo zaobišli na način da smo umjesto unicast algoritma koristili broadcast koji je, s obzirom da imamo samo dvije pločice, funkcionalno veoma sličan unicastu - ali izbjegava problem postavljanja servisa.

5. Programiranje ContikiOS za distribuirane algoritme

Sjetimo se da mote u distribuiranom sustavu ima neku mogućnost procesiranja, memoriju, alarm, mogućnost komunikacije i status. Nažalost ContikiOS pruža gotovo sve ove funkcije, ali ne omogućava postavljanje statusa motea direktno korištenjem ugrađenih funkcija. Stoga je ovo potrebno implementirati. Postoje dva načina za implementaciju statusa.

Prvi je korištenje statusa na razini jednog procesa, npr. pohranjivanjem tog statusa u varijabli. Zatim se, tokom izvršavanja tog procesa modificira njegovo ponašanje korištenjem funkcija kontrole toka (`if`, `else if`, `else`, `switch`...) u ovisnosti o vrijednosti varijable statusa. Ovo je primjereno za jednostavne aplikacije, kojima je potreban samo jedan proces - poput našeg ping-pong algoritma.
Drugi način je korištenje više procesa - gdje je svaki proces ekvivalentan jednom statusu. Tada status motea odgovara procesu koji se na njemu trenutno pokreće. Ovo je značajno kompleksniji način programiranja distribuiranih sustava na ContikiOS i dodatno ulazi u probleme sa manipulacijom procesa i događaja, ali je značajno primjereniji za kompleksnije sustave gdje je potrebno izvršavanje više procesa.

6. Ping pong algoritam kao primjer korištenja Contiki OS sustava

U ovom poglavlju, ukratko ćemo proći kroz bitnije dijelove algoritma koji smo napisali kao primjer razvoja distribuiranih algoritama na ContikiOS. Bitno je primjetiti da algoritam koristi 6LoWPAN za komunikaciju, i pisan je kao distribuirani algoritam (oba motea imaju jednaki firmware, ali je njihovo ponašanje određeno statusom). Cijeli kod je dan u prilogu rada, kao i u zip arhivi predanoj uz rad.

Program šalje poruku sa jedne na drugu pločicu. Kada ta pločica primi poruku, šalje poruku prvoj pločici. Prva pločica zatim ponovno odgovara i tako u nedogled.

Prvo definiramo makro koje ćemo koristiti u programu(mogući statusi, vrijeme između slanja, port koji koristimo za slanje) kao i globalne varijable(npr. varijabla u kojoj pohranjujemo trenutni status algoritma, varijabla u koju ćemo pohranjivati podatke o našoj konekciji.)

```
#define UDP_PORT 1234
#define SEND_INTERVAL (2 * CLOCK_SECOND)
#define STATUS_IDLE 0
#define STATUS_SENDER 1
#define STATUS_RECEIVER 2

int status = STATUS_IDLE;
int message_counter = 0;
static struct simple_udp_connection broadcast_connection;
```

Zatim je potrebni kreirati kontroler procesa i započeti proces. Proces započinjemo automatski, prilikom pokretanja sustava.

```
PROCESS(bmo_ping_pong, "bmo ping pong");
AUTOSTART_PROCESSES(&bmo_ping_pong);
```

Naravno, potrebno je definirati i procesni thread, koji odgovara procesu koji smo započeli:

```
PROCESS_THREAD(bmo_ping_pong, ev, data) {
   ...
}
```

U tom threadu, nakon što započnemo proces naredbom PROCESS_BEGIN() startamo event timer (timer koji će nakon isteka poslati događaj procesu), te mu trajanje postavimo na na trajanje koje smo ranije definirali:

```
static struct etimer periodic_timer;
etimer_set(&periodic_timer, SEND_INTERVAL);
```

Zatim registriramo udp protokol na portu i zadamo ga konekciji. Koristimo ranije definiran port i konekciju.

```
simple_udp_register(&broadcast_connection, UDP_PORT, NULL, UDP_PORT,
receiver);
```

Pošto smo definirali sve potrebne varijable, možemo preći u kontinuiranu while petlju, strukturiranu na slijedeći način:

```
while (1){
      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
      etimer_reset(&periodic_timer);

      if (status == STATUS_IDLE)
      {
       ...
      }

      if (status == STATUS_SENDER)
      {
 ...
      }
      else if (status == STATUS_RECEIVER)
      {
 ...
      }
}
```

Prve dvije linije su zaustavljanje izvršavanja do specifičnog događaja - u ovom slučaju isteka ranije postavljenog timera, te resetiranje tog timera.

Status IDLE je početni status u komu se moteovi nalaze, te u njemu ne obavljamo nikakve radnje. Dolaskom poruke se automatski izlazi iz ovog statusa.

Najinteresantniji je status SENDER. U tom statusu gasimo LE diodu upaljenu kod primanja poruke. Paljenje i gašenje LE dioda, kao i definicije naziva pojedinih dioda su implementirane u knjižnici dev-leds.h. Zatim kreiramo linkove među lokalnim nodeovima koje Contiki može pronaći i pohranimo ih u adresu, te pošaljemo poruku na tu adresu. Na kraju promijenimo status u receiver.

```
if (status == STATUS_SENDER)
{
      leds_off(LEDS_RED);
      uip_create_linklocal_allnodes_mcast(&addr);
      simple_udp_sendto(&broadcast_connection, "Ping", 4, &addr);
      status = STATUS_RECEIVER;
}
```

Netko bi mogao primijetiti da se status RECEIVER u petlji doima beskorisnim, pošto samo mjenja status u SENDER. Ovo je istina. Razlog tomu je što je receiver implementiran u funkciji `receiver(...)` koja se automatski poziva kada se na portu zadane konekcije pojavi poruke (odnosno, procesu se šalje događaj tipa `PROCESS_EVENT_MSG`). Receiver je zadužen za ispis obavijesti o primljenoj poruci, inkrementiranju varijable koja sadrži broj primljenih poruka i paljenju LE diode kao signalu da je poruka primljena. Status se zatim mijenja u sender i poruka se šalje drugom mikrokontroleru.

```
static void
receiver(struct simple_udp_connection *c,
      const uip_ipaddr_t *sender_addr,
      uint16_t sender_port,
      const uip_ipaddr_t *receiver_addr,
      uint16_t receiver_port,
      const uint8_t *data,
      uint16_t datalen)
{

      printf("Data received on port %d from port %d with length %d\n",
            receiver_port, sender_port, datalen);
      status = STATUS_SENDER;
      message_counter = message_counter + 1;
      printf("Message count = %d\n", message_counter);
      leds_on(LEDS_RED);
}
```

Na kraju, program završavamo naredbom `PROCESS_END()`

7. Zaključak

U ovom tekstu je dan pregled načina programiranja aplikacija za ContikiOS operativni sustav. Objašnjeno je korištenje procesa i događaja u kontekstu programiranja aplikacija za Contiki sustav, i objašnjena je metoda primjene statusa u svrhu implementacije distribuiranih algoritama na ContikiOs operativnom sustavu.

8. Literatura

“Internet of Things IN 5 DAYS”; A.L. Coline, A. Vives, M. Zennaro, A. Bagula, E. Pietrosemoli; 2015

“Contiki Wiki” - https://github.com/contiki-os/contiki/wiki

Toolchain installation on linux” - https://github.com/contiki-ng/contiki-ng/wiki/Toolchain-installation-on-Linux

“Contiki Start” - http://www.contiki-os.org/start.html

Prilog 1. Programski kod

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include "contiki.h"
#include "sys/etimer.h"
#include "dev/leds.h"
#include "net/ip/uip.h"
#include "net/ipv6/uip-ds6.h"
#include "net/ip/uip-debug.h"
#include "simple-udp.h"
#include "servreg-hack.h"

//port used for sending/receiving messages
#define UDP_PORT 1234
```

```
//length of time between messages
#define SEND_INTERVAL (2 * CLOCK_SECOND)
//Statuses
#define STATUS_IDLE 0
#define STATUS_SENDER 1
#define STATUS_RECEIVER 2

//variable for storing current mote status
int status = STATUS_IDLE;
//variable for storing number of message received
int message_counter = 0;
//connection data
static struct simple_udp_connection broadcast_connection;

/*---------------------------------------------------------------------------
--*/
PROCESS(bmo_ping_pong, "bmo ping pong");
AUTOSTART_PROCESSES(&bmo_ping_pong);
/*---------------------------------------------------------------------------
--*/
/*
* Function that's called after "receive message" event
* Handles the message received
*/
static void
receiver(struct simple_udp_connection *c,
      const uip_ipaddr_t *sender_addr,
      uint16_t sender_port,
      const uip_ipaddr_t *receiver_addr,
      uint16_t receiver_port,
      const uint8_t *data,
      uint16_t datalen)
{

      printf("Data received on port %d from port %d with length %d\n",
            receiver_port, sender_port, datalen);
      //change mote status
      status = STATUS_SENDER;
      //increase received message counter
      message_counter = message_counter + 1;
      printf("Message count = %d\n", message_counter);
      printf("status changed to %d\n", status);
```

```
      //turn the LED on
      leds_on(LEDS_RED);
}

/*---------------------------------------------------------------------------
--*/
PROCESS_THREAD(bmo_ping_pong, ev, data)
{
      uip_ipaddr_t addr;
      PROCESS_BEGIN();

      printf("PING PONG protocol demonstration\n");
      //define timer
      static struct etimer periodic_timer;
      //set timer
      etimer_set(&periodic_timer, SEND_INTERVAL);
      //register UDP protocol service
      simple_udp_register(&broadcast_connection, UDP_PORT,
                          NULL, UDP_PORT,
                          receiver);
      while (1)
      {
      printf("STATUS = %d\n", status);
    //wait until timer expires
      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&periodic_timer));
      //reset the timer after it expires
      etimer_reset(&periodic_timer);

      if (status == STATUS_IDLE)
      {
            printf("STATUS = %d", status);
            printf("STATUS - IDLE\n");
      }

      if (status == STATUS_SENDER)
      {
            printf("STATUS - SENDER\n");
            leds_off(LEDS_RED);
            printf("Sending broadcast\n");
            printf("Creating local node list.\n");
      //find neighbours
            uip_create_linklocal_allnodes_mcast(&addr);
```

```
            printf("Creating local node list done.\n");
      printf("Sending message.\n");
            //send message using defined connection
      /*
      * connection to send data on
      * message data
      * message data size
      * address to send on
      */
      simple_udp_sendto(&broadcast_connection, "Ping", 4, &addr);
      //change status to receiver
            status = STATUS_RECEIVER;
            printf("Message sent, status set to:%d\n", status);
      }
      else if (status == STATUS_RECEIVER)
      {
            printf("STATUS LISTENER\n");
      printf("RECEIVER DONE. STATUS=%d\n", status);
            printf("message received, status is %d\n", status);
      //change status to sender
            status = STATUS_SENDER;
      }
      }

      PROCESS_END();
}
```