UNIVERZITET U SARAJEVU
ELEKTROTEHNIČKI FAKULTET
ODSJEK ZA AUTOMATIKU I ELEKTRONIKU

RT-Thread RTOS

SEMINARSKI RAD

- RAZVOJ SOFTVERA ZA UGRADBENE SISTEME -

Student:
Vedad Halimić

Mentor:
Red. prof. dr Samim Konjicija.

Sarajevo,
juli 2021.

Sažetak

Jednostavne aplikacije razvijane za ugradbene sisteme sadrže dva osnovna nivoa - hadrdver i aplikaciju. Usložnjavanjem aplikacije raste potreba za više razina sistema, pri čemu se tada između aplikacije i hardvera nalaze elementi kao što su *middleware* i operativni sistem. Prilikom razvoja sistema sa manje nivoa, programer mora voditi računa o načinu izvršavanja programa i resursima. U slučaju korištenja operativnog sistema, isti preuzima takve zadaće na sebe i omogućava efikasno izvršavanje aplikacije implementacijom konkurentnog izvršavanja zadataka. Neki od najpoznatijih *real-time* operativnih sistema su: FreeRTOS, Zephyr, RT-Thread, Azure RTOS, QuantumLeaps RTOS i Mbed RTOS.

U ovom radu će biti predstavljen RT-Thread *real-time* operativni sistem, zajedno sa njegovim nativnim objektima koji omogućavaju konkurentno izvršavanje zadataka (engl. *tasks*). Fokus će biti na dijelovima kernela operativnog sistema kao što su upravljanje nitima i objektima tipa tajmer, semafor, skup događaja, *mailbox* i drugi. Bit' će date definicije funkcija koje realiziraju funkcionalnost objekata, kao i odgovarajući primjeri koji pokazuju način korištenja tih objekata u aplikaciji. Emulacija RT-Thread operativnog sistema je izvršena u programu RT-Thread studio.

Sadržaj

Popis slika

Poglavlje 1

Uvod

RT-Thread je real-time operativni sistem otvorenog koda pogodan za ugradbene sisteme. Karakteriše ga visoka pouzdanost, skalabilnost i modularnost, kao i niska razina zauzeća resursa. Koristi se u senzorskim čvorovima, čipovima za bežično povezivanje i mnogim drugim aplikacijama sa ograničenim resursima. Kada su u pitanju aplikacije koje zahtijevaju visoke performanse, RT-Thread se koristi u hardveru za povezivanje mreža (engl. gateway), IPC-u (engl. *inter-process communication*) i pametnim zvučnicima. Ovaj operativni sistem sadrži sve ključne komponente vezane za Internet of Things, kao što su mrežni protokoli, sistemi datoteka (engl. file systems) i niska potrošnja energije. Podržani su svi najpoznatiji alati za kompajliranje kao što su GCC, Keil, IAR itd. kao i standardna sučelja kao što su POSIX, CMSIS, okruženje za C++ aplikacije, Micropython i Javascript - omogućavajući programerima razvoj širokog spektra aplikacija. RT-Thread također nudi odličnu komercijalnu podršku za sve najzastupljenije CPU arhitekture kao što su ARM Cortex-M/R/A, RISC-V, MIPS, X86, Xtensa itd. Lista svih uređaja koji podržavaju RT-Thread operativni sistem je data u prilogu D.

Kernel ovog operativnog sistema zahtijeva samo 1.2 KB RAM-a i 3 KB flash memorije. Komponente za upravljanje potrošnjom energije podržavaju razne režime rada za štednju energije. S obzirom na to da se radi o softveru otvorenog koda, treba spomenuti i doprinos zajednice programera koji su razvili približno 200 paketa za različite namjene: IoT, periferija, sistem, programski jezici, razni alati, multimedija, sigurnost itd.

RT-Thread je kompatibilan sa POSIX standardima sučelja, a sadrži podršku kako za 32-bitne, tako i za 64-bitne procesore kao što su ARM Cortex-A i RISC-V arhitekture, s tim da može raditi na jedno-jezgrenim i na simetričnim multi-jezgrenim sistemima. Arhitektura RT-Thread operativnog sistema je data na slici (1.1).

Kernel sadrži preemptivni višezadaćni real-time raspoređivač i komponente kao što su semafor, mutex, mailbox, red poruka (message queue), signale, događaje, menadžment memorije, tajmera i prekida.

Komponente predstavljaju softversku jedinicu na RT-Thread kernel sloju, kao što su komandna linija, drajveri za uređaje, mrežni framework, virtualni file system, stack TCP/IP mrežnog protokola, POSIX standardni sloj itd. Softverske komponente se nalaze u RT-Thread/components direktoriju i opisane su SConscript datotekom i dodane u RT-Thread sistem. Kada se otvori neka od softverskih komponenti u sistemskoj konfiguraciji, ona se kompajlira i povezuje sa finalnim RT-Thread softverom.

Paketi predstavljaju middleware pokrenut na RT-Thread IoT platformi operativnog sistema. Svaki paket sadrži vlastiti opis, izvorni kod ili bibliotečne datoteke. Može se raditi o zvaničnim RT-Thread paketima ili o paketima razvijenim od strane zajednice. Softverske pakete karakteriše modularnost i ponovna upotrebljivost.

Poređenje *RT-Threada* sa drugim *real-time* operativnim sistemima je dato u [1]. Pokazalo se da *RT-Thread* ima najmanje kašnjenje prilikom promjene konteksa sa zadatka najnižeg na zadatak najvišeg prioriteta korištenjem *semaphore* objekta na ARM Cortex-M4 arhitekturi u odnosu na neke druge operativne sisteme (FreeRTOS, Keil RTX itd.). *RT-Thread* je u različitim scenarijima datim u spomenutom radu pokazao izuzetno dobre rezultate, a u većini slučajeva postignuti rezultati su bolji od rezultata najraširenijeg *real-time* operativnog sistema u ugradbenim sistemima - FreeRTOS-a.

Seminarski rad o RT-Thread *real-time* operativnom sistemu se sastoji od tri poglavlja: **Uvod**, **RT-Thread Kernel** i **Zaključak**. U sklopu seminarskog rada se nalaze i četiri priloga, koji nose sljedeće nazive: **Pokretanje RT-Thread emulacije**, **Definicije RT-Thread objekata**, **RT-Thread shell komande** i **RT-Thread podržani hardver**. Informacije vezane za prestavljeni operativni sistem su preuzete iz zvanične RT-Thread *online* dokmentacije dostupne u [2].

Poglavlje **Uvod** sadrži osnovne napomene o RT-Thread operativnom sistemu i pregled poglavlja seminarskog rada.

Poglavlje **RT-Thread Kernel** predstavlja centralno poglavlje ovog završnog rada. U njemu su predstavljeni *nativni* tipovi objekata u analiziranom operativnom sistemu. Potpoglavlja se bave osnovama kernela operativnog sistema, upravljanjem nitima, upravljanjem tajmer objektom, te sinhronizacijom i komunikacijom između niti.

U fokusu rada su nativni RT-Thread objekti kao što su: nit (engl. *thread*), tajmer (engl. *timer*), semafor (engl. *semaphore*), objekat međusobno isključivog pristupa (engl. *mutex*), skup događaja (engl. *event set*), poštanski sandučić (engl. *mailbox*), red poruka (engl. *message queue*) i signal.

Za svaki od navedenih objekata su dati načini kreiranja kako statičke, tako i dinamičke varijante odgovarajućeg objekta, kao i funkcije koje realiziraju njihovu funkcionalnost. Aplikacije koje koriste različite prezentirane meanizme operativnog sistema su preuzeti iz zvanične RT-Thread dokumentacije dostupne na ovom linku i po potrebi prilagođene za potrebe izrade ovog rada. Primjeri aplikacija sadrže programski kod i ispis sa terminala prilikom izvršavanja.

Poglavlje **Zaključak** sadrži završne komentare vezane za realizirani seminarski rad.

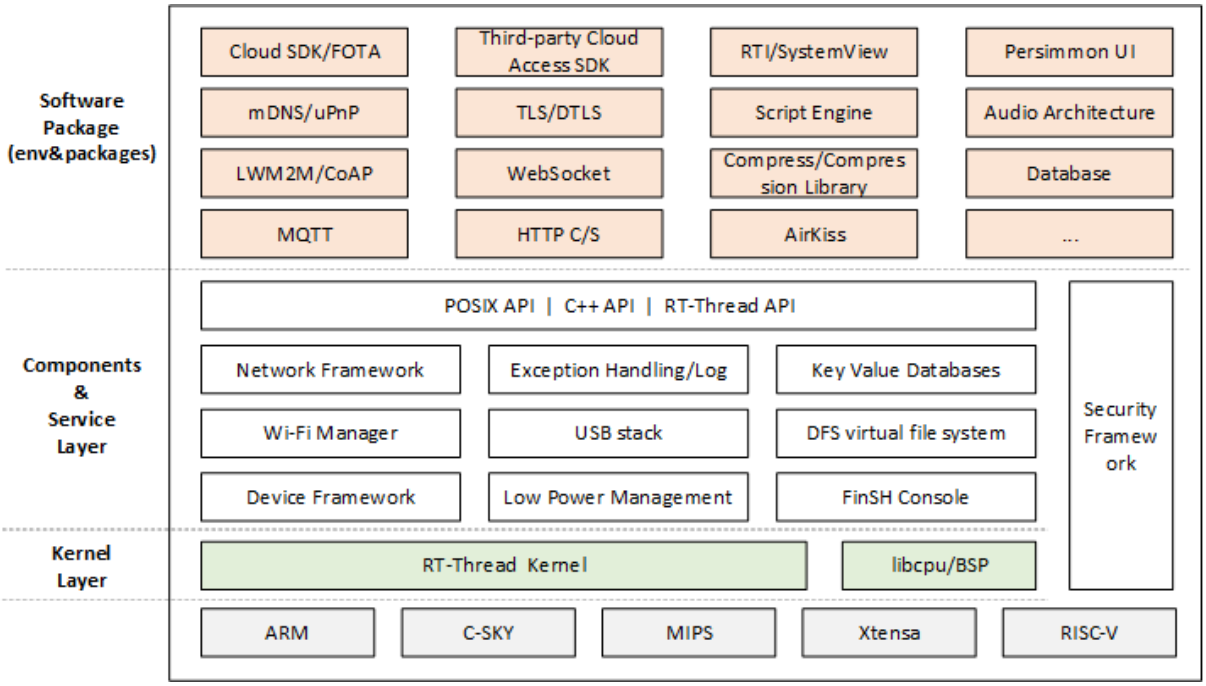
Dodatno, seminarski rad sadrži četiri priloga. S obzirom na to da se emulacija RT-Thread operativnog sistema izvršava u programu *RT-Thread Studio*, prilog **Pokretanje RT-Thread emulacije** je uključen u ovaj rad, kako bi čitaoci mogli samostalno testirati prezentirane funkcionalnosti i primjere aplikacija.

Nativni RT-Thread objekti su definirani kao strukture u programskom jeziku C. U prilogu **Definicije RT-Thread objekata** su date definicije tih struktura, kao i pokazivača na iste.

Prilikom emulacije RT-Thread operativnog sistema, odnosno procesa debagiranja u programu *RT-Thread studio*, moguće je putem konzole unositi komande koje su predstavljene u prilogu **RT-Thread shell komande**.

Kada su u pitanju *real-time* operativni sistemi, važna stavka je lista uređaja koji mogu podržati njihov rad. RT-Thread se može pohvaliti visokim nivoom kompatibilnosti sa različitim arhitekturama i razvojnim sistemima. Njihova lista je data u prilogu **RT-Thread podržani hardver**.

Popis *literature* korištene za izradu ovog rada, realiziranog u sklopu predmeta *Razvoj softvera za ugradbene sisteme*, je dat na kraju ovog seminarskog rada.



Slika 1.1: Arhitektura RT-Thread RTOS-a

Poglavlje 2

RT-Thread Kernel

2.1 Osnove kernela OS

Kernel je najosnovniji i najvažniji dio operativnog sistema. Nalazi se na vrhu hardverskog sloja. Sadrži kernel biblioteku i real-time kernel implementaciju. Kernel biblioteka predstavlja mali skup implementiranih funkcija u programskom jeziku C koje omogućavaju neovisan rad kernela. Real-time implementacija sadrži: upravljanje objektima, upravljanje nitima i raspoređivačem, komunikaciju između niti, upravljanje sistemskim satom i memorijom. RT-Thread karakterizira objekto-orijentirani dizajn.

Upravljanje nitima

Najosnovniji objekat sa kojim radi raspoređivač RT-Thread operativnog sistema je nit. Algoritam raspoređivanja niti je potpuno **preemptivni višenitni algoritam baziran na prioritetima**. Sistem podržava 256 nivoa prioriteta za niti, pri čemu 0 (*nula*) predstavlja najviši prioritet, dok su vrijednosti najnižih prioriteta rezervirane za niti koje su po svojoj prirodi uvijek u stanju mirovanja (engl. *idle*). Ukoliko pojedine niti imaju istu vrijednost prioriteta, važan parametar prilikom deklaracije predstavlja *timeslice* parametar, koji nosi informaciju o tome koliko iznosi dozvoljeno vrijeme njihovog izvršavanja, prije nego što procesor prepuste drugoj niti.

Upravljanje satom

U RT-Thread operativnom sistemu postoje dvije vrste tajmera, to su jednokratni (engl. *one-shot*) koji se aktivira samo jednom nakon dodijeljenog *timeout* vremena i periodični (engl. *periodic trigger*) tajmer, koji se aktivira periodično, sve dok ga korisnik ne zaustavi.

Sinhronizacija između niti

Da bi se postigla sinhronizacija između niti, RT-Thread koristi objekte kao što su semafori, mutexi i događaji. Mehanizam sinhronizacije omogućava nitima čekanje na osnovu prioriteta ili pristup resursima na osnovu zauzeća *mutex* ili *semaphore* objekta. Kada niti za svoje izvršavanje čekaju na pojavu više različitih događaja, pogodno je koristiti objekat tipa *event*.

Komunikacija između niti

Komunikacioni mehanizmi podržani od strane RT-Thread operativnog sistema su redovi poruka, mailbox itd. Dužina poruke u mailboxu je fiksne dužine od 4B, što omogućava veću

efikasnost u odnosu na red poruka, koji može primati poruke promjenljive dužine i smještati ih u vlastiti memorijski prostor. Akciju slanja je moguće izvršavati u sigurnom načinu rada, unutar prekidne rutine. Komunikacijski mehanizam omogućava nitima čekanje poruke na osnovu prioriteta ili FIFO (engl. *first-in first-out*) metodu.

Upravljanje memorijom

RT-Thread omogućava upravljanje kako statičkom, tako i dinamičkom memorijom. Kada je dostupan memorijski prostor u statičkoj memoriji, vrijeme alocirano za određeni memorijski blok će biti konstantno. Kada nema dostupnog prostora, sistem će tada zatražiti suspendiranje ili blokiranje niti memorijskog bloka. Kada ostale niti oslobode memorijski prostor, sistem će u aktivno stanje staviti nit koja je ranije suspendirana.

Upravljanje ulaznim i izlaznim uređajima

RT-Thread može koristiti periferne uređaje kao što su: PIN, I2C (engl. *Inter-Integrated Circuit*), SPI (engl. *Serial Peripheral Interface*), USB (engl. *Universal Serial Bus*) i UART (engl. *Universal Asynchronous Receiver-Transmitter*). Postoji realizacija podsistema za upravljanje ovim uređajima kojim se pristupa putem njihovog imena pomoću jedinstvenog API sučelja. U zavisnosti od karakteristika ugradbenog sistema, različiti događaji se mogu povezati sa različitim perifernim uređajima.

U nastavku će fokus biti na upravljanjem objektima kao što su niti, tajmeri, semafori, mutexi, događaji, signali, redu poruka (engl. *message queue*) i *mailboxu*.

2.1.1 Startanje RT-Thread OS

Postupak pokretanja RT-Thread operativnog sistema se može podijeliti na sljedeće korake: sistem se startuje pokretanjem *Startup* datoteke, zatim se poziva funkcija `rtthread_startup()` i na kraju se počinje izvršavati korisnički definiran program u funkciji `main()`. Detaljan proces pokretanja sistema je prikazan na slici (2.1).

Funkcija `$$Sub$$main()` poziva funkciju `rttread_startup()` koja inicijalizira hardver povezan sa sistemom, objekte kernela sistema (tajmeri, raspoređivači i signali), kreira glavnu nit, inicijalizira razne module u glavnoj niti, inicijalizira niti tajmera i *idle* nit, te pokreće raspoređivač.

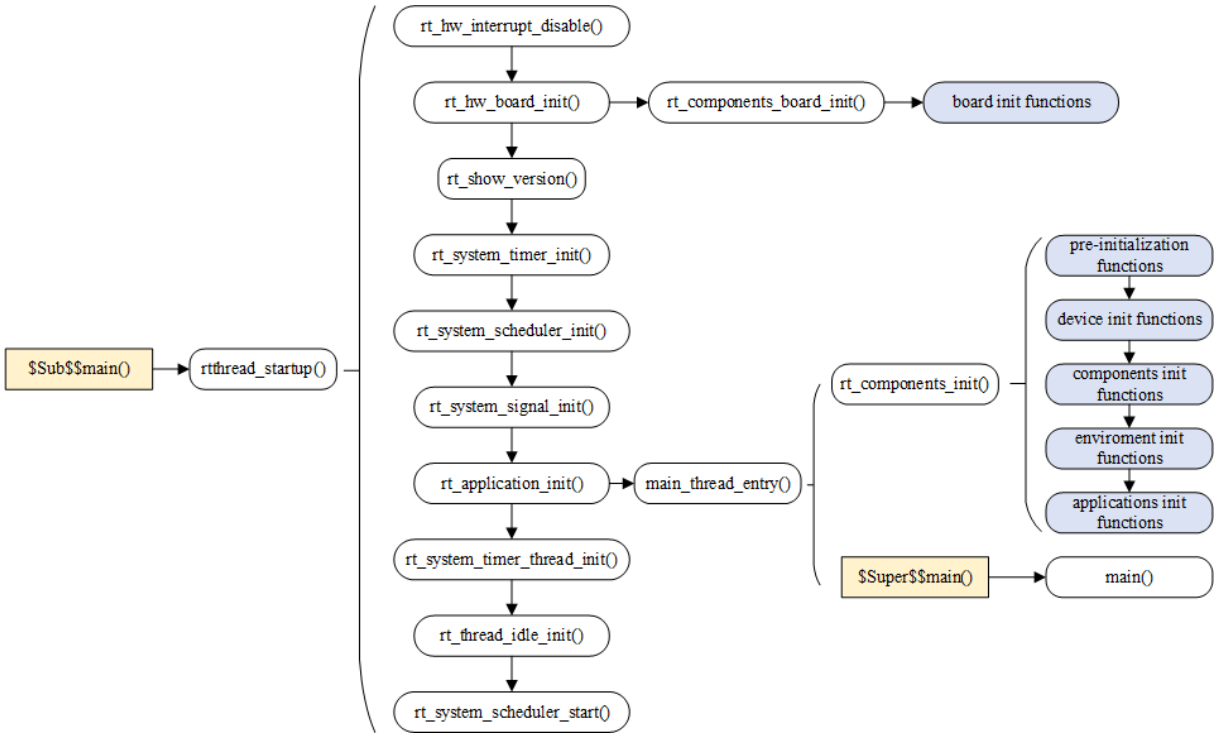
2.1.2 Statički i dinamički objekti

RT-Thread razlikuje statičke i dinamičke objekte. Pojedinačni objekti tipa nit, semafor, mutex itd. mogu imati ili jedan ili drugi oblik. Statički objekti su inicijalizirani u programu nakon pokretanja sistema, dok se dinamički objekti inicijaliziraju ručno. Treba napomenuti da svaki nativni tip objekta u RT-Thread operativnom sistemu ima sljedeći format:

```
rt_<ime_objekta>
```

Pseudo-pokazivači/reference, odnosno *object handleri* na odgovarajuće tipove objekata u RT-Thread *real-time* operativnom sistemu imaju sljedeći format:

```
rt_<ime_objekta>_t
```



Slika 2.1: Postupak startanja RT-Thread RTOS-a

Kada imamo *handler* nekog objekta, javnim atributima tog objekta se različito pristupa u zavisnosti od toga da li se radi o statičkim ili dinamičkim objektima. U slučaju dinamičkih objekata, operator za pristup je −>, dok je u slučaju statičkih operator pristupa tačka (`.`).

2.1.3 Struktura upravljanja kernel objektima

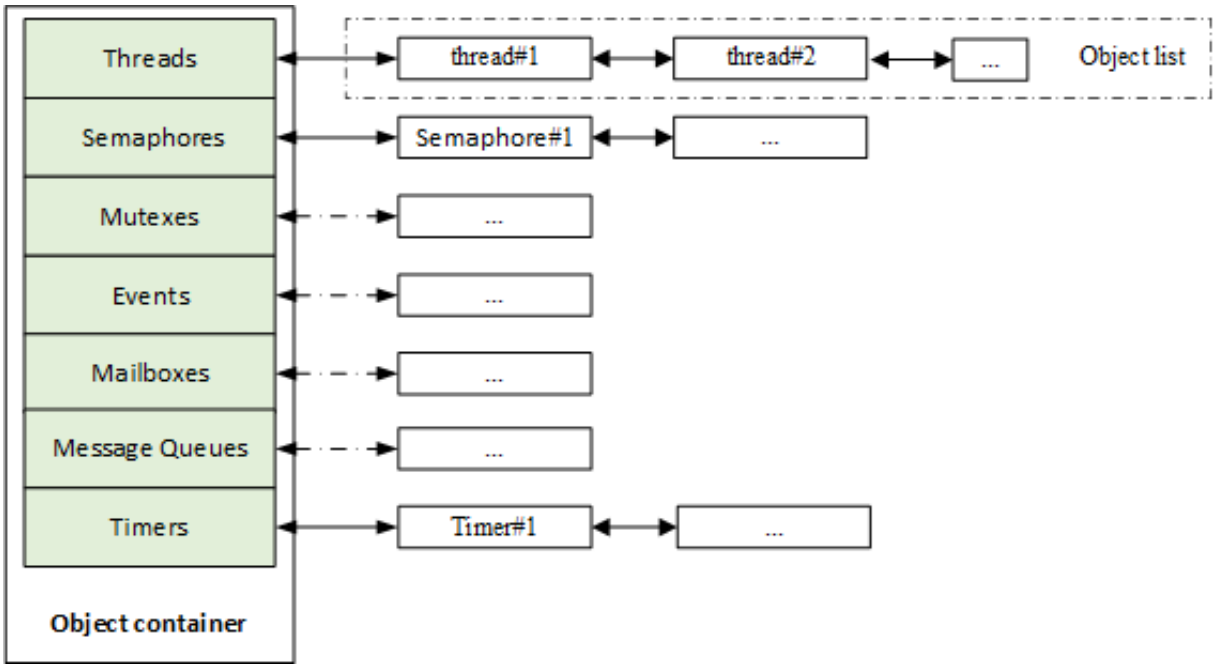
Kernel operativnog sistema koristi *kontejner objekata* (engl. *object container*), koji sadrži infromacije o svakom tipu kernel objekta. Svaki tip sadrži pokazivač na povezanu listu (engl. *linked list*) svih kreiranih objekata tog tipa. Opisana struktura je prikazana na slici (2.2).

Odlika objektno-orijentiranog pristupa je nasljeđivanje. U RT-Thread RTOS-u postoji jedan nativni *objekat najvišeg ranga* iz kojeg se nasljeđuju svi ostali objekti. Objekat najvišeg ranga sadrži osobine kao što su: ime, tip, *flag* i *list*. Definicija objekta najvišeg ranga je data u prilogu B.1. Objekti nižeg ranga, kao što su: niti, tajmeri, uređaji, semafori itd. se nasljeđuju iz navedene strukture i pored navedenih imaju dodatne osobine specifične za svaki objekat (npr. tajmer ima definirano vrijeme nakon kojeg se aktivira, semafor ima brojač itd.) Sve to je prikazano na slici (2.3). Prednosti ovakvog dizajna sistema su skalabilnost, olakšano dodavanje novih kategorija objekata, pojednostavljenje rada sa objektima različitog tipa i povećanje pouzdanosti sistema.

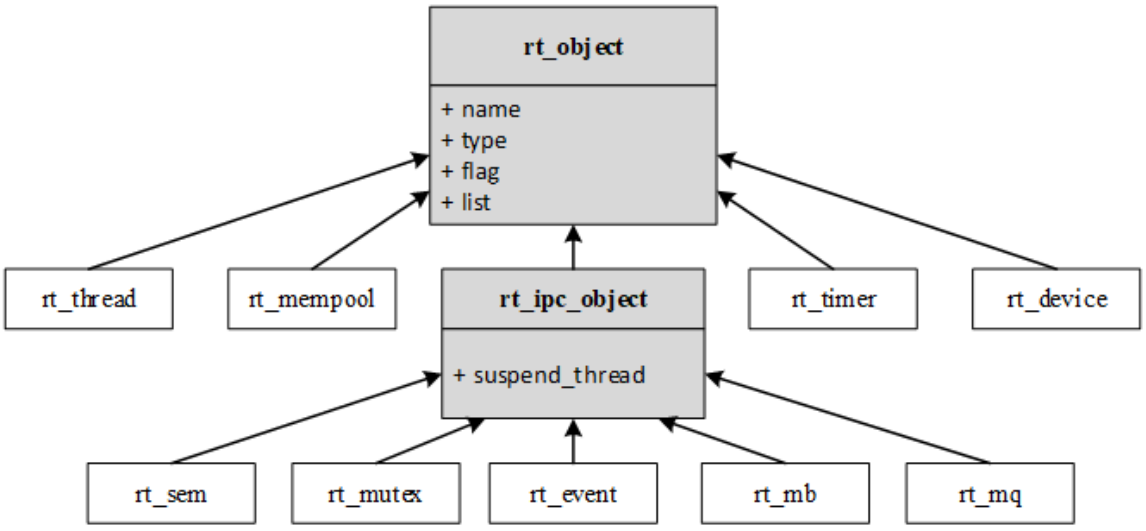
2.1.4 Primjer konfiguracije kernela

Važna karakteristika RT-Thread operativnog sistema je visok nivo prilagodljivosti prema potrebama aplikacije, odnosno korisnika. Podešavanje se vrši izmjenom datoteke `rtconfig.h`, koja se nalazi u projektnom folderu. Izmjenu je najlakše vršiti unutar *RT-Thread Studio* okruženja, detaljnije opisanog u prilogu A.

Ne umanjujući značaj ostalih opcija za podešavanje, u nastavku će biti prikazane samo neke



Slika 2.2: Kontejner objekata

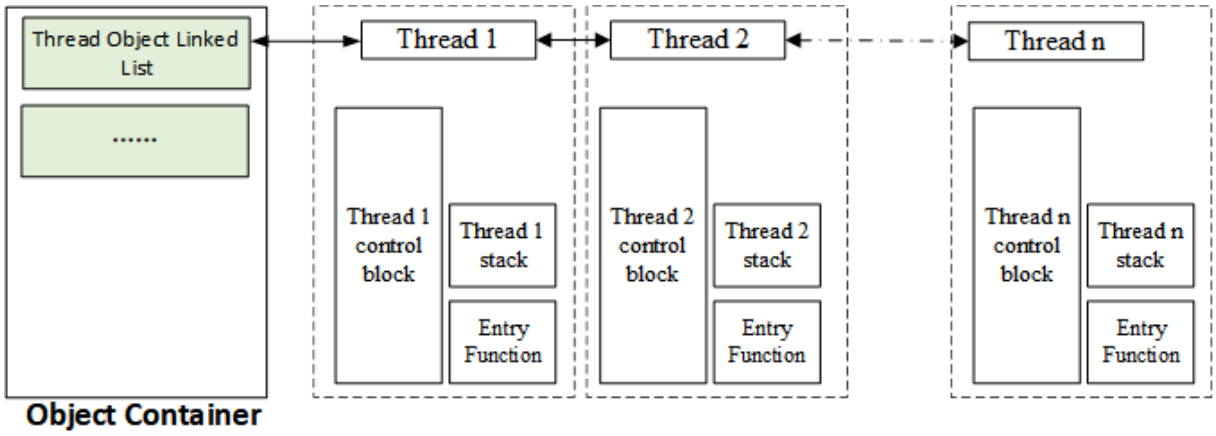


Slika 2.3: Objekat najvišeg ranga i nasljeđivanje osobina

od linija koda koje služe za konfiguraciju kernela operativnog sistema. Detaljnije informacije o konfiguraciji kernela je moguće pronaći u zvaničnoj *RT-Thread* dokumentaciji na linku. Posebno važna stavka je definiranje otkucaja sata pomoću makroa `RT_TICK_PER_SECOND`.

Program 2.1: rtconfig.h

```
#ifndef RT_CONFIG_H__
#define RT_CONFIG_H__

...

/* Dio za kernel */

/* Definiranje broja nivoa prioriteta za sistemske niti. */
#define RT_THREAD_PRIORITY_MAX 32

/* Definiranje otkucaja sata. Npr. za vrijednost 100, to znaci 100
   otkucaja u sekundi, odnosno takt je 10ms. */
#define RT_TICK_PER_SECOND 100

/* Definiranje ovog makroa omogucava koristenje hook funkcija. */
#define RT_USING_HOOK

/* Definiranje ovog makroa omogucava proces debagiranja.  */
#define RT_DEBUG

...

/* Dio za komunikaciju i sinhronizaciju izmedju niti */

/* Omogucava koristenje semafora, mutexa, dogadjaja, mailboxa, reda
   poruka i signala */
#define RT_USING_SEMAPHORE
#define RT_USING_MUTEX
#define RT_USING_EVENT
#define RT_USING_MAILBOX
#define RT_USING_MESSAGEQUEUE
#define RT_USING_SIGNALS

...

/* Omogucava koristenje konzole u nacinu rada debagiranja */
#define RT_USING_CONSOLE

...

/* Informacija o razvojnom sistemu. */
#define SOC_VEXPRESS_A9

...

#endif
```



Slika 2.4: Povezana lista niti u kontejneru objekata

2.2 Upravljanje nitima

Niti (engl. *threads*) su nosioci zadataka u aplikacijama koje se izvršavaju unutar *real-time* operativnih sistema. U sistemu postoje dvije vrste niti - to su korisnički definirane niti i sistemske niti. Sistemske niti kreira kernel operativnog sistema, a korisnički definirane niti kreira aplikacija. Sve niti se nalaze unutar kontejnera objekata u povezanoj listi objekta *thread*, kao što se može vidjeti na slici 2.4. Svaka nit sadrži vlastiti kontrolni blok, *stack*, ulaznu funkciju itd. Definicija objekta *thread* je data u prilogu B.2.

2.2.1 Atributi objekta thread

Struktura `rt_thread` ima nekoliko važnih atributa, a to su:

- vlastiti *stack*,
- stanje niti,
- prioriet,
- vremenski okvir za izvršavanje,
- ulazna funkcija i
- kod greške.

Stack

Svaka nit mora imati vlastiti ***stack*** da bi prilikom promjene konteksta u isti mogla sačuvati stanje svojih varijabli - ukoliko procesor treba dodijeliti resurse nekoj drugoj niti. Također, stack služi i da se prilikom ponovne aktivacije određene niti, iz *stacka* pročitaju vrijednosti korištenih varijabli. Prilikom kreiranja niti, moguće je definirati veličinu njenog *stacka*. Informaciju o veličini *stacka* svih trenutno aktivnih niti u sistemu je moguće dobiti pozivanjem komande `list_threads`. Puna lista komandi za *RT-Thread Shell* data u prilogu C.

Stanje niti

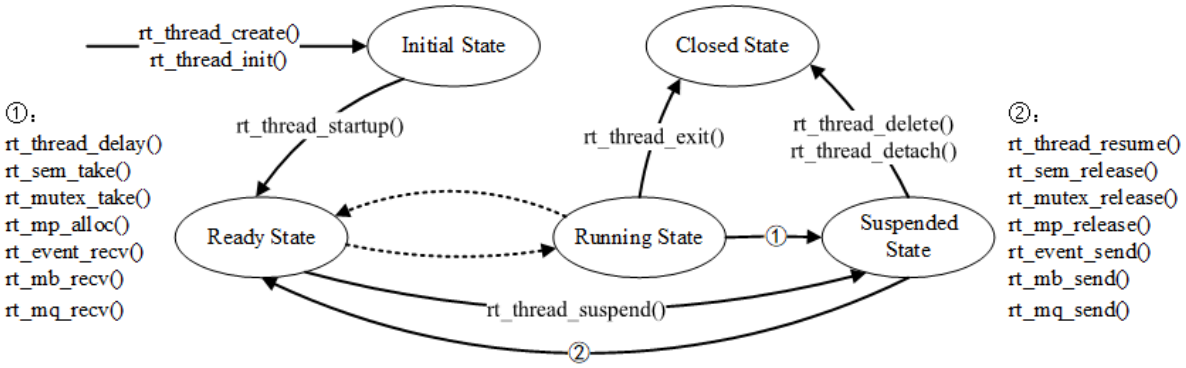
Kada je jedna nit u aktivnom stanju, samo ona ima pravo da koristi procesor, sve dok ne završi svoj rad ili sve dok se ne desi promjena konteksta, odnosno dok se procesor ne dodijeli nekoj drugoj niti. Niti u RT-Threadu imaju pet različith stanja i operativni sistem automatski određuje u kojem će se stanju neka nit nalaziti. **Stanja** su pobrojana u nastavku, a unutar zagrada, pored imena stanja, je data i odgovarajuća makro definicija.

- Inicijalno stanje (`RT_THREAD_INIT`) - Nit se nalazi u inicijalnom stanju u trenutku kada je upravo kreirana i ne nalazi se u stanju izvršavanja. U ovom stanju, nit ne učestvuje u raspoređivanju.
- *Ready* stanje (`RT_THREAD_READY`) - U ovom stanju se nit nalazi u redu čekanja za izvršenje po prioritetima. Kada nit, koja je trenutno u stanju izvršavanja, završi svoj rad, operativni sistem pokreće onu nit koja ima najveći prioritet, a koja se nalazi u *ready* stanju.
- Stanje izvršavanja (`RT_THREAD_RUNNING`) - (engl. *running state*) - Kod niti se izvršava. U slučaju jedno-jezgrenog (engl. *single-core*) sistema, samo jedna nit može biti u ovom stanju. Iako se pomoću promjene konteksta stvara privid izvršavanja više niti istovremeno, paralelno izvršavanje više niti bez prekidanja rada bilo koje od njih se može postići jedino u više-jezgrenim (engl. *multi-core*) sistemima.
- Blokirajuće stanje (`RT_THREAD_SUSPEND`) - (engl. *blocking state* ili *suspended state*) - Niti koje se nalaze u ovom stanju ne učestvuju u raspoređivanju, a u ovo stanje mogu doći u situacijama kada je resurs potreban za njihovo izvršavanje nedostupan ili zato što je korištena funkcija odgađanja (engl. *delay*).
- Terminirano stanje (`RT_THREAD_CLOSE`) - (engl. *closed state*) - Raspoređivač u svom radu ne uzima u obzir niti koje se nalaze u ovom stanju, a u ovo stanje se dolazi onda kada nit prestane sa izvršavanjem.

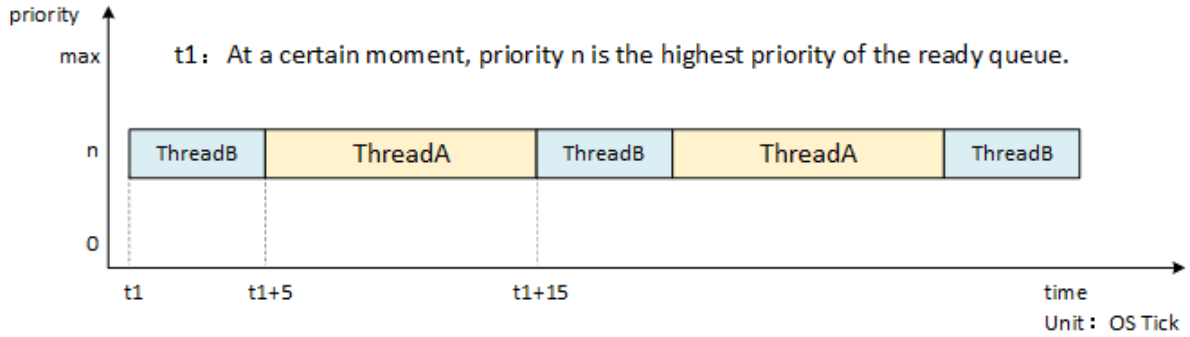
Promjena stanja Brigu o nitima vodi opeativni sistem i za promjenu stanja koristi sistemske pozive. Nit ulazi u inicijalno stanje pozivanjem funkcije `rt_thread_create()` ili `rt_thread_init()`; da bi raspoređivač operativnog sistema uzeo u razmatranje određenu nit, ona se mora nalaziti u *ready* stanju, u koje se iz inicijalnog stanja dolazi pozivanjem funkcije `rt_thread_startup()`. Nit najvišeg prioriteta ulazi u stanje izvršavanja i zadržava se u tom stanju sve dok se ne desi sistemski poziv `rt_thread_exit()` koje nit dovodi u terminirajuće stanje ili dok se ne desi sistemski poziv koji će nit dovesti u suspendirano stanje (primjeri takvih sistemskih poziva su: `rt_thread_delay()`, `rt_sem_take()`, `rt_mutex_take`). U suspendirano stanje se može doći i u slučaju kada nit ne dobije pristup odgovarajućim resursima. Prelazak u *ready* stanje iz suspendiranog stanja je moguć u slučaju da protekne određeno vrijeme bez pribavljanja traženih resursa ili ako ostale niti oslobode resurse. Korištenjem sistemskih poziva `rt_thread_delete()` i `rt_thread_detach()`, nit se iz suspendiranog prebacuje u terminirajuće stanje. Opisani konačni automat ima strukturu kao na slici (2.5).

Prioritet niti

Svaka nit mora imati definiranu vrijednost **prioriteta**. Što je prioritet niti veći, veća je šansa da će su raspoređivač uvrstiti u raspored izvršavanja niti. RT-Thread podržava maksima-



Slika 2.5: Moguća stanja niti predstavljena konačnim automatom



Slika 2.6: Naizmjenično izvršavanje niti istog prioriteta, ali različitog trajanja izvršavanja

lan broj od 256 prioriteta (od 0 do 255), pri čemu nula predstavlja najviši prioritet. U nekim sistemima sa ograničenim resursima je moguće odabrati samo sistemske konfiguracije koje podržavaju 8 ili 32 nivoa prioriteta. Najniži prioriteti se dodjeljuju *idle* nitima i nisu u upotrebi od strane korisnika. Ukoliko se u *ready* stanju pojavi nit koja ima viši prioritet od niti koja se trenutno izvršava, nit višeg prioriteta prelazi u stanje izvršavanja.

Vremenski okvir

Parametar koji se odnosi na **vremenski okvir** u kojem se određena nit izvršava je izuzetno važan u slučaju konkurentnog izvršavanja više niti **istog** prioriteta. Tada se procesor dodjeljuje nitima istog prioriteta na određeno vrijeme, definirano parametrom *timeslice*. Primjerice, neka postoje dvije niti, A i B, istog prioriteta, pri čemu su *timeslice* parametri za ove niti 10 i 5, respektivno. Dimenzionalnost vremena je *OS Tick*, koji zavisi od postavki u konfiguracijskoj datoteci `rtconfig.h`. Sistem će naizmjenično dodjeljivati procesor nitima A i B na definirano vrijeme, ukoliko nema niti višeg prioriteta u *ready* stanju. Nit A će se izvršavati duplo duže od niti B, što se može vidjeti na slici (2.6).

Ulazna funkcija

Ulazna funkcija (engl. *entry function*) je glavni dio svake niti, odnosno svakog `rt_thread` objekta. Generalno, funkcija može imati dva načina rada, a to su beskonačna petlja (engl. *infinite loop*) ili sekvencijalno izvršavanje (engl. *sequential execution*)/izvršavanje sa konačnim brojem ciklusa (engl. *finite-cycle mode*).

U sistemima koji zahtijevaju rad u realnom vremenu, niti su uglavnom pasivne u smislu da

čekaju vanjske događaje da bi izvršili obradu zahtjeva. Ne smije se dozvoliti da se nit nalazi u beskonačnoj petlji, jer se u protivnom niti nižeg prioriteta neće nikada izvršiti. Rješenje koje omogućava oslobađanje procesora i dodjelu istog nekoj drugoj niti podrazumijeva korištenje *delay* funkcije ili ručno suspendiranje niti. Svrha definiranja niti koja radi u režimu beskonačne petlje je omogućavanje da se nit izvršava više puta i da nikada ne bude obrisana, za razliku od *finite-cycle* načina rada.

Program 2.2: Beskonacna_petlja

```
1 void thread_entry(void* paramenter)
2 {
3     while (1)
4     {
5     /* waiting for an event to occur */
6
7     /* Serve and process events */
8     }
9 }
```

Niti sa sekvencijalnim izvršavanjem se neće izvršavati vječno i mogu se opisati kao *jednokratne* niti. Nakon što se izvrše, brišu se automatski od strane sistema.

Program 2.3: Sekvencijalno_izvrsavanje

```
1 static void thread_entry(void* paraemter)
2 {
3     /* Processing Transaction #1 */
4         ...
5     /* Processing Transaction #2 */
6         ...
7     /* Processing Transaction #3 */
8 }
```

Kod greške

Svaka nit sadrži varijablu u kojoj se smješta **kod greške** u slučaju pojavljivanja iste. Kodovi greške mogu imati vrijednosti date u nastavku.

```
#define RT_EOK              0 /* No error   */
#define RT_ERROR            1 /* Regular error      */
#define RT_ETIMEOUT         2 /* Timeout error      */
#define RT_EFULL            3 /* Resource is full   */
#define RT_EEMPTY           4 /* No resource        */
#define RT_ENOMEM           5 /* No memory */
#define RT_ENOSYS           6 /* System does not support    */
#define RT_EBUSY            7 /* System busy        */
#define RT_EIO              8 /* IO error   */
#define RT_EINTR            9 /* Interrupt system call      */
#define RT_EINVAL          10 /* Invalid Parameter */
```

2.2.2 Main i idle niti

Sistem pri pokretanju kreira **glavnu nit** (engl. *main thread*). Njena ulazna funkcija je `main_thread_entry()`. Korisnička aplikacija se pokreće unutar glavne niti. Proces je prikazan u nastavku.

```
$$Sub$$main() → rtthread_startup() → rt_application_init() →
                main_thread_entry() → main()
```

***Idle* nit** kreira operativni sistem i dodjeljuje joj najniži mogući prioritet. Ova vrsta sistemske niti se uvijek nalazi u *ready* stanju i izvršava se svaki put kada ne postoji niti jedna druga nit spremna za izvršavanje. Implementacija ove niti se može shvatiti kao beskonačna petlja. *Idle* nit je moguće povezati sa *hook* funkcijom.

2.2.3 Rad sa nitima

U nastavku će biti prikazane razne nativne RT-Thread funkcije, koje omogućavaju upravljanje nitima. Bit' će prikazane definicije funkcija sa opisom ulaznih argumenata i izlaznih parametara tih funkcija.

Kreiranje niti

Nit dinamičkog tipa se može **kreirati** pomoću funkcije `rt_thread_create`, čija su definicija i opis argumenata datu u nastavku.

```
rt_thread_t rt_thread_create(const char* name,
                             void (*entry)(void* param),
                             void* param,
                             rt_uint32_t stack_size,
                             rt_uint8_t priority,
                             rt_uint32_t tick);
```

- Argumenti:
  - name - ime niti
  - entry - funkcija koju nit izvršava
  - param - parametar ulazne (engl. *entry*) funkcije
  - stack_size - veličina *stacka* niti izražena u bajtima
  - priority - nivo prioriteta niti
  - tick - *timeslice* dodijeljen niti, izražava se u *OS Tick* jedinici
- Moguće povratne vrijednosti:
  - thread - nit uspješno kreirana, funkcija vraća *thread handle*
  - `RT_NULL` - nit nije kreirana

U slučaju da nit kreirana pomoću gore navedene funkcije više nije potrebna ili je došlo do neke greške, moguće ju je **obrisati** pomoću funkcije:

```
rt_err_t rt_thread_delete(rt_thread_t thread);
```

Funkcija prima *thread handle* na nit kao argument, a vraća `RT_EOK` ako je brisanje uspješno, odnosno `-RT_ERROR` u onda kada brisanje nije uspješno. Nit se nakon brisanja uklanja iz povezane liste niti i oslobađa se prostor koji je ova nit zauzela.

Statička nit se **inicijalizira** korištenjem funkcije `rt_thread_init`, čija su definicija i opis argumenata dati u nastavku. Kreiranje statičke niti podrazumijeva da su upravljački blok (engl. *control block*) i *stack* niti globalne varijable, alocirane i definirane prilikom kompajliranja.

```
rt_err_t rt_thread_init(struct rt_thread* thread,
                        const char* name,
                        void (*entry)(void* param),
                        void* param,
                        void* stack_start,
                        rt_uint32_t stack_size,
                        rt_uint8_t priority,
                        rt_uint32_t tick);
```

- Argumenti:
  - thread - *thread handle*, odnosno pokazivač na adresu upravljačkog bloka niti
  - name - ime niti
  - entry - funkcija koju nit izvršava
  - param - parametar ulazne (engl. *entry*) funkcije
  - stack_start - početna adresa za *stack* niti
  - stack_size - veličina *stacka* niti izražena u bajtima
  - priority - nivo prioriteta niti
  - tick - *timeslice* dodijeljen niti, izražava se u *OS Tick* jedinici
- Moguće povratne vrijednosti:
  - `RT_EOK` - nit uspješno kreirana
  - `-RT_ERROR` - nit nije kreirana

Niti kreirane pomoću funkcije `rt_thread_init` se ne mogu brisati, već se samo mogu **odvojiti** (engl. *detach*) od reda niti (engl. *thread queue*) i kernelovog menadžera objekata pomoću funkcije:

```
rt_err_t rt_thread_detach (rt_thread_t thread);
```

koja prima *thread handle* inicijaliziranog isključivo pomoću `rt_thread_init` funkcije, a vraća `RT_EOK` ako je brisanje uspješno, u suprotnom vraća `-RT_ERROR`.

S obzirom na način raspoređivanja niti prema prioritetima od strane raspoređivača, prepoznaje se potreba za eventualnom promjenom prioriteta u toku izvršavanja aplikacije. To je moguće postići korištenjem `rt_thread_control()` funkcije, o kojoj se više informacija može pronaći u zvaničnoj *RT-Thread* dokumentaciji na linku.

Pokretanje niti

Nit koja je kreirana/inicijalizirana se nalazi u inicijalnom stanju i raspoređivač je u svom radu ne uzima u obzir za izvršavanje. Da bi se to omogućilo, potrebno je prevesti nit u *ready* stanje **startenjem** iste pomoću funkcije:

```
rt_err_t rt_thread_startup(rt_thread_t thread);
```

koja vraća `RT_EOK` ukoliko je nit uspješno, odnosno `-RT_ERROR` identifikator ukoliko start nije bio uspješan. U slučaju uspješnog pokretanja, nit se nalazi u redu za čekanje, na odgovarajućoj poziciji shodno svom prioritetu. Posmatrana nit odmah započinje izvršavanje, ukoliko ima veći prioritet od niti koja se trenutno izvršava.

Pauziranje izvršavanja

U nekim aplikacijama je potrebno odgoditi izvršavanje niti koja je trenutno aktivna za neki period vremena. **Pauziranje** se može pozvati jednom od sljedeće tri navedene funkcije:

```
rt_err_t rt_thread_sleep(rt_tick_t tick);

rt_err_t rt_thread_delay(rt_tick_t tick);

rt_err_t rt_thread_mdelay(rt_int32_t ms);
```

Ove tri funkcije imaju isti efekat, koji prevodi nit iz stanja aktivnog izvršavanja u suspendirano stanje. Nakon nekog vremena, nit iz koje se pozvao ovaj tip funkcije će ući u *ready* stanje. Parametar prve dvije funkcije ima dimenzionalnost *1 OS Tick*, dok se kod funkcije `rt_thread_mdelay` vrijeme zadaje u milisekundama.

Dobavljanje trenutno aktivne niti

U toku izvršavanja programa, isti kod se može izvršavati od strane različitih niti. Za vrijeme izvršavanja je pomoću funkcije

```
rt_thread_t rt_thread_self(void);
```

moguće dobaviti *thread handle* niti koja se trenutno izvršava. Ukoliko raspoređivač nije počeo sa radom, ova funkcija vraća `RT_NULL`.

Idle hook

Ranije je spomenuta *idle* nit, koja se izvršava onda kada se ne izvršavaju korisnički definirane niti. Moguće je definirati korisničku *idle hook* funkciju, koja će se izvršavati u sklopu *idle* niti, onda kada raspoređivač nema drugih *niti* na raspolaganju za izvršavanje. Sučelje za postavljanje i brisanje *idle hook* funkcije se sastoji od funkcija `rt_thread_idle_sethook` i `rt_thread_idle_delhook`, respektivno.

```
// postavljanje hook funkcije
rt_err_t rt_thread_idle_sethook(void (*hook)(void));

// brisanje hook funkcije
rt_err_t rt_thread_idle_delhook(void (*hook)(void));
```

Obje funkcije vraćaju `RT_EOK` ukoliko je izvršavanje uspješno, odnosno `-RT_EFULL` ili `-RT_ENOSYS` ukoliko željena operacija nije uspješna. Treba napomenuti da je *idle* nit uvijek u *ready* stanju i unutar korisnički definirane funkcije za izvršavanje te niti nije poželjno koristiti funkcije koje će tu nit dovesti su suspendirano stanje.

Scheduler hook

Promjena konteksta je najčešći događaj u radu operativnog sistema. Moguće je definirati *scheduler hook* funkciju ukoliko npr. želimo znati koje niti su učestvovale u promjeni konteksta. Ako definiramo sljedeću funkciju:

```
void hook(struct rt_thread* from,
          struct rt_thread* to);
```

čiji su parametri *from* - pokazivač na upravljački blok niti koja je upravo završila sa radom i *to* pokazivač na upravljački blok niti koja će započeti svoje izvršavanje nakon što se `hook()` funkcija izvrši. Da bi se ova funkcija zaista izvršavala pri promjeni konteksta, potrebno je u kodu iskoristiti funkciju `rt_scheduler_sethook`.

```
void rt_scheduler_sethook (void (*hook)(struct rt_thread
   * from, struct rt_thread* to));
```

2.2.4 Primjeri rada sa nitima

U ovoj podsekciji će biti dati primjeri kodova vezanih za rad sa nitima. Bit' će predstavljen primjer kreiranja i izvršavanja niti, primjer sa konkurentnim izvršavanjem niti istog prioriteta i primjer korištenja *scheduler hook* funkcije.

Kreiranje niti

Kreirane su dvije niti, pri čemu je *nit 1* statička (kreirana sa funkcijom `rt_thread_create`), a *nit 2* dinamička nit (inicijalizirana sa funkcijom `rt_thread_init`). Druga nit ima veći prioritet (nivo prioriteta iznosi 24) i izvršava se sve dok se petlja definirana u njenoj ulaznoj funkciji ne izvrši do kraja. Nakon toga nit 2 terminira, a prva nit (čiji nivo prioriteta iznosi 25) počinje svoje izvršavanje i ispisuje brojeve svakih $0.5s$. Periodično ispisivanje je postignuto korištenjem funkcije `rt_thread_mdelay`. Programski kod ovog primjera i ispis su prikazani u nastavku.

Program 2.4: Thread Basics

```
#include <rtthread.h>

#define THREAD_PRIORITY         25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_thread_t tid1 = RT_NULL;

/* Entry Function for Thread 1 */
static void thread1_entry(void *parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* Thread 1 runs with low priority and prints the count
           value all the time */
        rt_kprintf("thread1 count: %d\n", count ++);
        rt_thread_mdelay(500);
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
/* Entry for Thread 2 */
static void thread2_entry(void *param)
{
    rt_uint32_t count = 0;

    /* Thread 2 has a higher priority to preempt thread 1 and get
       executed */
    for (count = 0; count < 10 ; count++)
    {
        /* Thread 2 prints count value */
        rt_kprintf("thread2 count: %d\n", count);
    }
    rt_kprintf("thread2 exit\n");
    /* Thread 2 will also be automatically detached from the system
       after it finishes running. */
}

/* Thread Sample */
int main(void)
{
    /* Creat thread 1, Name is thread, Entry is thread1_entry */
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

    /* Start this thread if you get the thread control block */
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* Creat thread 2, Name is thread2, Entry is thread2_entry */
    rt_thread_init(&thread2,
```

```
55                      "thread2",
56                      thread2_entry,
57                      RT_NULL,
58                      &thread2_stack[0],
59                      sizeof(thread2_stack),
60                      THREAD_PRIORITY - 1, THREAD_TIMESLICE);
61      rt_thread_startup(&thread2);
62
63      return 0;
64  }
```

```
// ispis

msh >thread2 count: 0
thread2 count: 1
thread2 count: 2
thread2 count: 3
thread2 count: 4
thread2 count: 5
thread2 count: 6
thread2 count: 7
thread2 count: 8
thread2 count: 9
thread2 exit
thread1 count: 0
thread1 count: 1
thread1 count: 2
thread1 count: 3
...
```

Konkurentno izvršavanje niti istog prioriteta

Kreirane su dvije niti dinamičkog tipa, koje dijele istu ulaznu funkciju, ali različite ulazne argumente koji služe za razlikovanje niti prilikom ispisa. Prioritet obje niti je isti. Međutim, period vremena za izvršavanje dodijeljen svakoj niti se razlikuje tako da jedna nit ima *timeslice* od 3 *OS Tick*-a, dok isti parametar kod druge niti iznosi 1 *OS Tick*. S obzirom na podešavanja u `rtconfig.h` datoteci, *1 OS Tick* iznosi 10*ms*, što predstavlja period takt impulsa procesora. Programski kod je dat u nastavku.

Program 2.5: Time-Slice Round Robin

```
#include <rtthread.h>

#define THREAD_STACK_SIZE   1024
#define THREAD_PRIORITY     15
#define THREAD_TIMESLICE    3

/* Thread Entry */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;
    rt_uint32_t count = 0;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        if(0 == (count % 25))
        {
            rt_kprintf("\n thread %d is running, thread %d count = %
               d ", value , value , count);

            if(count> 500)
               return;
        }
         count++;
     }
}

//int timeslice_sample(void)
int main(void)
{
    rt_thread_t tid = RT_NULL;
    /* Create Thread 1 */
    tid = rt_thread_create("thread1", thread_entry, (void*)1,
       THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid != RT_NULL)
        rt_thread_startup(tid);


    /* Create Thread 2 */
    tid = rt_thread_create("thread2", thread_entry, (void*)2,
       THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE-2);
    if (tid != RT_NULL)
        rt_thread_startup(tid);
    return 0;
}
```

Izvršavanje obje niti se odvija naizmjenično, a svaka nit se izvršava određeni period vremena. Da bi se ukazalo na različito vrijeme izvršavanja, vrši se ispis varijable *count* svaki put kada je njena vrijednost djeljiva sa definiranim brojem (vrijednost se uvećava prolaskom kroz petlju). Ispis je dat u nastavku. Može se primijetiti da *nit 1* zaista ima veći *timeslice*, s obzirom na to da je 375 najveći broj ispisan u prvom izvršavanju, dok je u slučaju *niti 2* to 150. Može se također primijetiti da je izvršavanje prve niti prekinuto prilikom ispisa, zatim se određeno vrijeme izvršava druga nit, te se nakon sljedeće promjene konteksta ispis iz prve niti nastavlja tamo gdje je zaustavljen. *Entry* funkcije ovih niti su pisane tako da terminiraju nakon što varijabla *count* pređe određenu definiranu vrijednost. Kako *nit 1* ima veći *timeslice* od *niti 2*, to

će ona prije stići do te vrijednosti, terminirati i više se neće pokretati, te će se bez promjene konteksta do kraja izvršiti druga nit.

```
// ispis

 thread 1 is running, thread 1 count = 0
 thread 1 is running, thread 1 count = 25
 thread 1 is running, thread 1 count = 50
 thread 1 is running, thread 1 count = 75
 thread 1 is running, thread 1 count = 100
 thread 1 is running, thread 1 count = 125
 thread 1 is running, thread 1 count = 150
 thread 1 is running, thread 1 count = 175
 thread 1 is running, thread 1 count = 200
 thread 1 is running, thread 1 count = 225
 thread 1 is running, thread 1 count = 250
 thread 1 is running, thread 1 count = 275
 thread 1 is running, thread 1 count = 300
 thread 1 is running, thread 1 count = 325
 thread 1 is running, thread 1 count = 350
 thread 1 is running, thread 1 count = 375
 thread 1 is
 thread 2 is running, thread 2 count = 0
 thread 2 is running, thread 2 count = 25
 thread 2 is running, thread 2 count = 50
 thread 2 is running, thread 2 count = 75
 thread 2 is running, thread 2 count = 100
 thread 2 is running, thread 2 count = 125
 thread 2 is running, thread 2 count = 150 running, thread 2 count =
     175
 thread 1 is running, thread 1 count = 425
 thread 1 is running, thread 1 count = 450
 thread 1 is running, thread 1 count = 475
 thread 1 is running, thread 1 count = 500
 thread 1 is running, thread 1 count = 525
 thread 1 is running, thread 1 count = 525
 thread 2 is running, thread 2 count = 200
 thread 2 is running, thread 2 count = 225
 thread 2 is running, thread 2 count = 250
 thread 2 is running, thread 2 count = 275
 thread 2 is running, thread 2 count = 300
 thread 2 is running, thread 2 count = 325
 thread 2 is running, thread 2 count = 350
 thread 2 is running, thread 2 count = 375
 thread 2 is running, thread 2 count = 400
 thread 2 is running, thread 2 count = 425
 thread 2 is running, thread 2 count = 450
 thread 2 is running, thread 2 count = 475
 thread 2 is running, thread 2 count = 500
 thread 2 is running, thread 2 count = 525
```

Scheduler hook funkcija

Kreirane su dvije niti, koje samo ispisuju informaciju o tome koja je nit trenutno aktivna. Funkcija `hook_of_scheduler` je pomoću funkcije `rt_scheduler_sethook` vezana za raspoređivač tako da vrši korisnički definiranu operaciju prilikom promjene konteksta. Može se reći da su joj argumenti pokazivači na dvije niti, jednu koja je suspendirana i drugu, koja

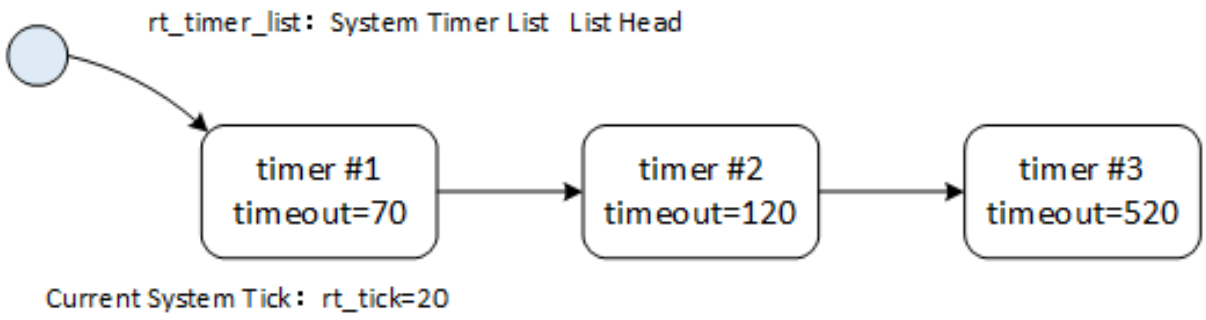
treba započeti/nastaviti svoje izvršavanje. Radi kratkoće ispisa, prikazano je samo nekoliko promjena konteksta korištenjem funkcije *strcmp* za detekciju samo korisnički definiranih niti prilikom promjene konteksta i terminacijom niti nakon što interni brojač prestigne određenu vrijednost. Programski kod i ispis su prikazani u nastavku.

Program 2.6: Scheduler Hook

```
#include <rtthread.h>

#define THREAD_STACK_SIZE   1024
#define THREAD_PRIORITY     20
#define THREAD_TIMESLICE    10

/* Counter for each thread */
volatile rt_uint32_t count[2];

/* Threads 1, 2 share an entry, but the entry parameters are
   different */
static void thread_entry(void* parameter)
{
    rt_uint32_t value;
    rt_uint32_t count = 0;

    value = (rt_uint32_t)parameter;
    while (1)
    {
        rt_kprintf("thread␣%d␣is␣running\n", value);
        rt_thread_mdelay(1000); // Delay for a while

        /*
         * Next line of code shortens the console output,
         * so we can see only a few of the context changes.
         */
        if (count++>3)
        {
            return;
        }
    }
}

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

static void hook_of_scheduler(struct rt_thread* from, struct
   rt_thread* to)
{
    /*
     * We will print output to a console,
     * only when one of the threads in context change
     * is user defined thread.
     */
    rt_int8_t log1 = rt_strcmp(from->name, "thread1");
    rt_int8_t log2 = rt_strcmp(from->name, "thread2");
    rt_int8_t log3 = rt_strcmp(to->name, "thread1");
    rt_int8_t log4 = rt_strcmp(to->name, "thread2");

    if (!log1 || !log2 || !log3 || !log4)
    {
```

```
        rt_kprintf("from:␣%s␣-->␣␣to:␣%s␣\n", from->name , to->name);
    }
}

int main(void)
{
    /* Set the scheduler hook */
    rt_scheduler_sethook(hook_of_scheduler);

    /* Create Thread 1 */
    tid1 = rt_thread_create("thread1",
                            thread_entry, (void*)1,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    /* Create Thread 2 */
    tid2 = rt_thread_create("thread2",
                            thread_entry, (void*)2,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY,THREAD_TIMESLICE - 5);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    return 0;
}
```

```
// ispis

msh />from: tshell -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle0
from: tidle0 -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle0
from: tidle0 -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle0
from: tidle0 -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle0
from: tidle0 -->  to: thread1
thread 1 is running
from: thread1 -->  to: thread2
thread 2 is running
from: thread2 -->  to: tidle0
from: tidle0 -->  to: thread1
from: thread1 -->  to: thread2
from: thread2 -->  to: tidle0
```



Slika 2.7: Povezana lista objekata tipa *timer*

2.3 Timer i clock

RT-Thread nudi opciju rada sa *tajmerima* za npr. periodično obavljanje određenih zadataka ili obavljanja zadatka nakon nekog vremenskog perioda. Razlikuju se **hardverski** i **softverski** tajmeri. U slučaju korištenja hardverskog tajmera, podešava se fizički *timer* modul na sistemu koji ima preciznost reda nanosekundi. Softverski tajmer predstavlja sučelje operativnog sistema. Nakon isteka definiranog vremena, priroda *timeout* funkcije kod hardverskog tajmera je funkcija prekidne rutine, dok se kod softverskog tajmera pokreće odgovarajuća nit. Vrijeme definirano u softverskom tajmeru može biti samo cjelobrojni umnožak jedinice *OS Tick*. Npr. ukoliko je period takt impulsa 10*ms* (*1 OS Tick*), onda se za *timeout* tajmera mogu podesiti samo vremena 20, 30, 40*ms* itd. ali ne i npr. 15*ms*. Definicija *timer* objekta je data u prilogu B.3.

Razlikuju se dva *timer*-mehanizma: jednokratni i periodični. Jednokratni (engl. *one-shot*) tajmer generira događaj samo jednom nakon isteka definiranog vremena i onda se automatski zaustavlja, dok se periodični tajmer automatski obnavlja i generira događaj periodično sve dok ga korisnik ne zaustavi ručno. Ukoliko korisnik želi koristiti softverske tajmere u svojoj aplikaciji, potrebno je da u konfiguracijsku datoteku doda direktivu `#define RT_USING_TIMER_SOFT`, dok su hardverski tajmeri po *defaultu* uključeni.

2.3.1 Mehanizam rada

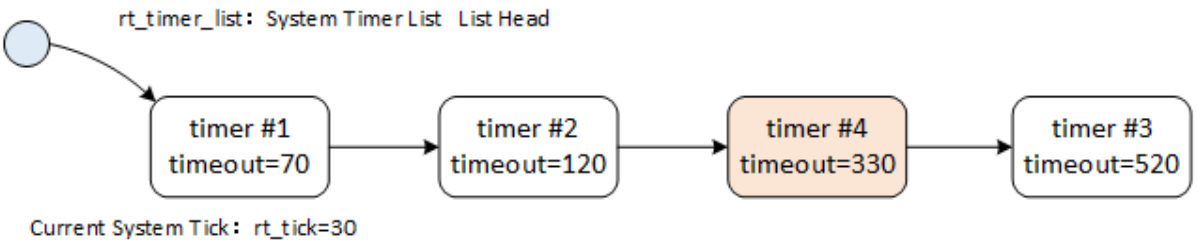
RT-Thread operativni sistem nudi informaciju o trenutnoj vrijednosti broja takt impulsa koji su protekli od početka izvršavanja aplikacije. Tu informaciju je moguće dobiti korištenjem funkcije `rt_tick_get`.

```
rt_tick_t rt_tick_get(void);
```

Povratnu vrijednost možemo sačuvati u neku varijablu, npr. neka se ta varijabla zove `rt_tick`. Ukoliko se npr. u trenutku *rt_tick=20* kreiraju tri tajmera: *timer#1*, *timer#2* i *timer#3* čija su *timeout* vremena 50, 100 i 500 takt impulsa. Formira se povezana lista, pri čemu se *timeout* vremena modificiraju tako da se na njih doda trenutna vrijednost sata, kao što je prikazano na slici (2.7).

Nakon 50 otkucaja (odnosno kada *rt_tick* bude imao vrijednost 70) izvršit će se *timeout* funkcija prvog tajmera i taj tajmer će se deaktivirati. Isto će se desiti i sa ostala dva tajmera, onda kada *rt_tick* imao vrijednost 120, odnosno 520.

Neka se sada razmatra isti slučaj, s tim da se u trenutku *rt_tick*=30 kreira novi tajmer - *timer#4*, čiji je *timeout* jednak 300. S obzirom na njegovu vrijednost *timeout* vremena i trenutne



Slika 2.8: Primjer *ubacivanja* novog *timer* objekta u povezanu listu

vrijednosti sata, *timer#4* će se aktivirati u trenutku kada *rt_tick* bude imao vrijednost 330 i bit će *ubačen* u povezanu listu između drugog i trećeg tajmera, kako je prikazano na slici (2.8). Algoritam koji omogućava efikasno dodavanje tajmera u povezanu listu se naziva *Timer Skip List Algorithm*. Treba napomenuti da se vremena aktivacije ostalih tajmera **neće** promijeniti u ovom slučaju.

2.3.2 Rad sa tajmerima

Kreiranje/inicijalizacija timera

Objekti tipa `rt_timer`, kao i niti, mogu biti statički i dinamički. Funkcije za kreiranje/inicijalizaciju tajmera su slične onima koje služe za definiranje niti u aplikaciji. **Dinamičko kreiranje tajmera** se postiže korištenjem funkcije `rt_timer_create`.

```
rt_timer_t rt_timer_create(const char* name,
                           void (*timeout)(void* param),
                           void* param,
                           rt_tick_t time,
                           rt_uint8_t flag);
```

- Argumenti:
  - name - ime tajmera
  - void (timeout) (void parameter) - pokazivač na funkciju koja se izvršava kada kreirani tajmer kreira događaj
  - param - ulazni parametar *timer* ulazne funkcije
  - time - vrijeme za koje će tajmer generirati događaj
  - flag - argument koji definira prirodu tajmera (jednokratni ili periodički, hardverski i softverski)
- Moguće povratne vrijednosti:
  - Timer Handle - tajmer uspješno kreiran
  - `RT_NULL` - kreiranje neuspješno

Što se tiče argumenta *flag*, mogući identifikatori su:

- `RT_TIMER_FLAG_ONE_SHOT` - jednokratni tajmer
- `RT_TIMER_FLAG_PERIODIC` - periodični tajmer
- `RT_TIMER_FLAG_HARD_TIMER` - hardverski tajmer
- `RT_TIMER_FLAG_SOFT_TIMER` - softverski tajmer

Kada određeni dinamički tajmer više nije potreban u aplikaciji, moguće ga je **obrisati** sa funkcijom `rt_timer_delete`, koja prima *timer handle* kao argument i vraća `RT_EOK` kao znak da je brisanje uspješno.

```
rt_err_t rt_timer_delete(rt_timer_t timer);
```

Kreiranje statičkog tajmera je moguće izvršiti pomoću funkcije `rt_timer_init`.

```
void rt_timer_init(rt_timer_t timer,
                   const char* name,
                   void (*timeout)(void* param),
                   void* param,
                   rt_tick_t time, rt_uint8_t flag);
```

- Argumenti:
  - timer - *timer handle*
  - name - ime tajmera
  - void (timeout) (void parameter) - pokazivač na funkciju koja se izvršava kada kreirani tajmer kreira događaj
  - param - ulazni parametar *timer* ulazne funkcije
  - time - vrijeme za koje će tajmer generirati događaj
  - flag - argument koji definira prirodu tajmera (jednokratni ili periodički, hardverski i softverski)

Kada statički tajmer više nije potreban, može se koristiti funkcija `rt_timer_detach` koja za uspješnu *detach* operaciju vraća `RT_EOK` identifikator.

```
rt_err_t rt_timer_detach(rt_timer_t timer);
```

Neki od parametara već kreiranih *timer* objekata se mogu promijeniti i tokom izvršavanja koda, ukoliko je to potrebno, korištenjem `rt_timer_control()` funkcije. Više informacija se može pronaći u zvaničnoj *RT-Thread* dokumentaciji na linku.

Startanje i stopiranje tajmera

Kada se tajmer kreira/inicijalizira, raspoređivač operativnog sistema ga ne uzima u obzir sve dok se ne **startuje** pomoću funkcije:

```
rt_err_t rt_timer_start(rt_timer_t timer);
```

koja prevodi stanje tajmera u aktivno (`RT_TIMER_FLAG_ACTIVATED`) i dodaje ga u povezanu listu tajmera u kontejneru objekata operativnog sistema. Funkcija kao argument prima *timer handle*. Ukoliko u nekom trenutku želimo **zaustaviti** tajmer i izbaciti ga iz povezane liste tajmera, pozivamo funkciju:

```
rt_err_t rt_timer_stop(rt_timer_t timer);
```

koja vraća `RT_EOK` ako je zaustavljanje uspješno, odnosno `-RT_ERROR` u suprotnom slučaju.

2.3.3 Primjer tajmer aplikacije

Kreirana su dva objekta tipa `rt_timer`, od kojih je jedan periodički, a drugi *one-shot* timer. Također, jedan od tajmera je statičkog, a drugi dinamičkog tipa. *Timeout* funkcija periodičkog tajmera ispisuje tekst na konzolu svakih 10 *OS Tickova*, dok je *timeout* jednokratnog tajmera jednak broju 50. Za potrebe predstavljenog primjera je kreirana funkcija `void print_tick(char* string)` koja prima pokazivač na prvi znak u nizu znakova, ispisuje taj tekst i trenutnu vrijednost sistemskog brojača takt impulsa. Programski kod je predstavljen u nastavku.

Program 2.7: Timer Sample

```
1  #include <rtthread.h>
2
3  /* Timer Control Block */
4  static rt_timer_t timer1;
5  static struct rt_timer timer2;
6  static int cnt = 0;
7
8  void print_tick(char* string)
9  {
10     rt_kprintf("%s - rt_tick = %d\n", string, rt_tick_get());
11 }
12
13 /* Timer 1 Timeout Function */
14 static void timeout1(void *parameter)
15 {
16     print_tick("periodic timer");
17     rt_kprintf("periodic timer is timeout %d\n", cnt);
18
19     /* On the 10th time, stops perodic timer */
20     if (cnt++>= 5)
21     {
22         rt_timer_stop(timer1);
23         rt_kprintf("periodic timer was stopped! \n");
```

```
24      }
25  }
26
27  /* Timer 2 Timeout Function */
28  static void timeout2(void *parameter)
29  {
30      print_tick("one-shot␣timer");
31      rt_kprintf("one␣shot␣timer␣is␣timeout\n");
32  }
33
34  int main(void)
35  {
36      print_tick("Application␣started.");
37      /* Create Timer 1 Periodic Timer */
38      timer1 = rt_timer_create("timer1", timeout1,
39                               RT_NULL, 10,
40                               RT_TIMER_FLAG_PERIODIC);
41
42      print_tick("Timer␣1␣created.");
43
44      /* Start Timer 1*/
45      if (timer1 != RT_NULL) rt_timer_start(timer1);
46
47      print_tick("Timer␣1␣started.");
48
49      /* Create Timer 2 One Shot Timer */
50      rt_timer_init(&timer2, "timer2", timeout2,
51                    RT_NULL, 50,
52                    RT_TIMER_FLAG_ONE_SHOT);
53
54      print_tick("Timer␣2␣initialized.");
55
56      rt_timer_start(&timer2);
57
58      print_tick("Timer␣2␣started.");
59  }
```

Funkcija `print_tick` služi za demonstraciju rada aplikacije. Poziva se u dijelovima koda koji su značajni sa stanovišta izvršavanja aplikacije i kao argument joj se šalje konstantni niz znakova koji opisuje trenutak od značaja. Npr. poziv `print_tick("Timer 2 started.");` se u kodu nalazi nakon startanja tajmera broj 2. Rad periodičkog tajmera broj 1 se završava nakon što brojač dostigne određenu vrijednost.

Može se vidjeti da su oba tajmera kreirana i startana u trenutku `rt_tick = 105`. Kako je definirani *timeout* jednokratnog tajmera (*Timer 2*) jednak broju 50, to će *Timer 2* generirati događaj u trenutku $rt\_tick = 105 + 50 = 155$, što se može vidjeti u ispisu datom u nastavku. Isto tako, *Timer 1* ima periodički *timeout* od 10 *OS Tickova*, što znači da će generirati događaj u trenucima $rt\_tick = 105 + n \cdot 10$, pri čemu je n prirodan broj, sve dok se *Timer 1* ne stopira.

```
// ispis

Application started. - rt_tick = 105
Timer 1 created. - rt_tick = 105
Timer 1 started. - rt_tick = 105
Timer 2 initialized. - rt_tick = 105
Timer 2 started. - rt_tick = 105
msh />periodic timer - rt_tick = 115
periodic timer is timeout 0
periodic timer - rt_tick = 125
periodic timer is timeout 1
periodic timer - rt_tick = 135
periodic timer is timeout 2
periodic timer - rt_tick = 145
periodic timer is timeout 3
one-shot timer - rt_tick = 155
one shot timer is timeout
periodic timer - rt_tick = 155
periodic timer is timeout 4
periodic timer - rt_tick = 165
periodic timer is timeout 5
periodic timer was stopped!
```

2.4 Sinhronizacija između niti

Ukoliko je pristup nekoj memorijskoj lokaciji nije dozvoljen samo jednoj niti, onda se može desiti da više niti istovremeno pristupa nekom resursu. Do problema sa konzistencijom podataka može doći ukoliko neka od niti pokuša npr. pročitati vrijednost sa određene lokacije, a druga nit čiji je zadatak da upisuje vrijednost na tu lokaciju, akciju pisanja nije u potpunosti završila. Pristup dijeljenim resursima mora biti isključiv tj. ako jedna nit pristupa nekoj varijabli, druge niti moraju čekati svoj red da bi dobile pristup. Dio koda kojim pristupa više niti se naziva kritična sekcija (engl. *critical section*). Sinhronizacija između niti podrazumijeva izvršavanje koda tačno određenim redoslijedom, pri čemu upravo te niti kontrolišu izvršavanje. Mehanizmi za sinhronizaciju su *semafori*, *mutexi* i *događaji* (engl. *events*).

2.4.1 Semaphore

Semafor (engl. *semaphore*) je kernel objekat koji služi za sinhronizaciju između niti. Zauzimanjem i oslobađanjem semafora, nit može postići sinhronizaciju ili međusobnu isključivost. Svaki `rt_semaphore` objekat ima vrijednost semafora i red niti koje čekaju na zauzeće. Definicija *semaphore* objekta je data u prilogu B.4.

Kreiranje/inicijalizacija semafora

Objekti tipa `rt_semaphore` mogu biti statički i dinamički. **Dinamička alokacija semafora** je moguća korištenjem funkcije `rt_sem_create`.

```
rt_sem_t rt_sem_create(const char *name,
                       rt_uint32_t value,
                       rt_uint8_t flag);
```

- Argumenti:
  - name - ime semafora
  - value - inicijalna vrijednost semafora
  - flag - zastavica
- Moguće povratne vrijednosti:
  - *semaphore control block pointer* - funkcija vraća pokazivač na kontrolni blok semafora ukoliko je kreiranje semafora uspješno
  - `RT_NULL` - kreiranje semafora neuspješno

pri čemu parametar *flag* može imati vrijednost `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`. U FIFO (engl. *first-in first-out*) režimu rada, red za čekanje na naredno zauzeće semafora se popunjava tako što prioritet ima nit koja je ranije poslala zahtjev za zauzeće. Postoji i prioritetni način raspoređivanja, pri čemu prioritet na naredno zauzeće semafora ima nit sa najvišim prioritetom (odnosno najmanjom vrijednosti *priority* atributa). Dinamičke semafore je moguće **obrisati** korištenjem funkcije `rt_sem_delete`, koja vraća `RT_EOK` identifikator u slučaju uspješnog brisanja.

```
rt_err_t rt_sem_delete(rt_sem_t sem);
```

Obekat tipa `rt_semaphore` statičkog tipa se **inicijalizira** pomoću funkcije `rt_sem_init`, koja ima sljedeću definiciju i opise parametara.

```
rt_err_t rt_sem_init(rt_sem_t sem,
                     const char *name,
                     rt_uint32_t value,
                     rt_uint8_t flag)
```

- Argumenti:
  - sem - *handle* `rt_semaphore` objekta
  - name - ime semafora
  - value - početna vrijednost semafora
  - flag - zastavica
- Moguće povratne vrijednosti:
  - RT_EOK - inicijalizacija uspješna

Moguće vrijednosti za *flag* argument su identične kao u slučaju dinamičkog semafora tj. moguće je birati *FIFO* ili *prioritetni* način rada. Za **odvajanje** semafor objekta od kernel upravitelja objekata koristi se funkcija `rt_sem_detach`, koja također vraća `RT_EOK` u slučaju uspješne operacije.

```
rt_err_t rt_sem_detach(rt_sem_t sem);
```

Zauzimanje i oslobađanje semafora

Prethodno opisane funkcije kreiranja/inicijalizacije su karakteristične za sve objekte u RT-Thread operativnom sistemu. U nastavku će biti prikazano sučelje specifično za objekat tipa semafor. Kada je vrijednost semafora veća od 0, nit može zauzeti semafor i vrijednost semafora će se umanjiti za 1. Kada nit oslobodi semafor, vrijednost se povećava za 1. Ako je vrijednost semafora jednaka nuli, nit koja je zatražila zauzeće će, u zavisnosti od postavljenog argumenta:

- terminirati,
- čekati određeno vrijeme da se semafor oslobodi, pa nakon toga terminirati ili
- čekati zauvijek da se semafor oslobodi.

Zauzimanje (engl. *obtaining*) **semafora** se postiže sljedećom funkcijom:

```
rt_err_t rt_sem_take (rt_sem_t sem,
                      rt_int32_t time);
```

koja prima *handle* semafor objekta (argument *sem*) i vrijeme čekanja (argument *time* izražen u jedinici *OS Tick*). Ukoliko je potrebno da nit čeka sve dok se semafor ne oslobodi, potrebno je staviti identifikator `RT_WAITING_FOREVER`. U zavisnosti od povratne vrijednosti funkcije, mogu se razlikovati slučajevi kada je zauzimanje semafora uspješno (`RT_EOK`), neuspješno nakon čekanja definiranog vremena (`-RT_ETIMEOUT`) ili su se dogodile neke druge greške (`-RT_ERROR`).

Zauzimanje semafora bez čekanja je moguće realizirati sljedećom funkcijom:

```
rt_err_t rt_sem_trytake(rt_sem_t sem);
```

koja ima isti efekat kao poziv: `rt_sem_take(sem, 0)`. Ovdje izraz *bez čekanja* znači da nit koja poziva ovu funkciju neće uopšte čekati da semafor bude dostupan za zauzimanje, ukoliko je njegova vrijednost u tom trenutku nula; umjesto toga, nit će terminirati. Povratne vrijednosti funkcije za instantno zauzimanje semafora su identične povratnim vrijednostima funkcije `rt_sem_take`, sa izuzetkom `-RT_ERROR` identifikatora.

Oslobađanje semafora je moguće postići sljedećom funkcijom:

```
rt_err_t rt_sem_release(rt_sem_t sem);
```

koja prima *handle* objekat semafora, a vraća `RT_EOK` u slučaju uspješnog oslobađanja semafora, odnosno povećanja njegove vrijednosti za 1. Ukoliko već neka nit čeka u redu za zauzeće, ona će sljedeća zauzeti semafor.

Primjer korištenja semafora

U nastavku je prikazan primjer aplikacije u kojoj se koriste objekti tipa `rt_semaphore`. Kreirane su dvije niti i jedan dinamički semafor. Jedna nit je zadužena za oslobađanje, a druga za zauzimanje semafora.

Program 2.8: Semaphore Sample

```
#include <rtthread.h>

#define THREAD_PRIORITY         25
#define THREAD_TIMESLICE        5

/* pointer to semaphore */
static rt_sem_t dynamic_sem = RT_NULL;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
static void rt_thread1_entry(void *parameter)
{
    static rt_uint8_t count = 0;

    while(1)
    {
        if(count <= 100)
        {
            count++;
        }
        else
            return;

        /* count release semaphore every 10 counts */
         if(0 == (count % 10))
        {
            rt_kprintf("t1 release a dynamic semaphore.\n");
            rt_sem_release(dynamic_sem);
        }
    }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread2_entry(void *parameter)
{
    static rt_err_t result;
    static rt_uint8_t number = 0;
    while(1)
    {
        /* permanently wait for the semaphore; once obtain the
           semaphore, perform the number self-add operation */
        result = rt_sem_take(dynamic_sem, RT_WAITING_FOREVER);
        if (result != RT_EOK)
        {
            rt_kprintf("t2 take a dynamic semaphore, failed.\n");
            rt_sem_delete(dynamic_sem);
            return;
        }
        else
        {
            number++;
            rt_kprintf("t2 take a dynamic semaphore. number = %d\n"
                ,number);
        }
```

```
56      }
57  }
58
59  /* initialization of the semaphore sample */
60  int main(void)
61  {
62      /* create a dynamic semaphore with an initial value of 0 */
63      dynamic_sem = rt_sem_create("dsem", 0, RT_IPC_FLAG_FIFO);
64      if (dynamic_sem == RT_NULL)
65      {
66          rt_kprintf("create␣dynamic␣semaphore␣failed.\n");
67          return -1;
68      }
69      else
70      {
71          rt_kprintf("create␣done.␣dynamic␣semaphore␣value␣=␣0.\n");
72      }
73
74      rt_thread_init(&thread1,
75                     "thread1",
76                     rt_thread1_entry,
77                     RT_NULL,
78                     &thread1_stack[0],
79                     sizeof(thread1_stack),
80                     THREAD_PRIORITY, THREAD_TIMESLICE);
81      rt_thread_startup(&thread1);
82
83      rt_thread_init(&thread2,
84                     "thread2",
85                     rt_thread2_entry,
86                     RT_NULL,
87                     &thread2_stack[0],
88                     sizeof(thread2_stack),
89                     THREAD_PRIORITY-1, THREAD_TIMESLICE);
90      rt_thread_startup(&thread2);
91
92      return 0;
93  }
```

Thread 1 oslobađa semafor kada je vrijednost lokalne varijable brojača djeljiva sa deset. Kada *Thread 2* zauzme semafor, povećava vrijednost svog brojača za 1. Ispis je prikazan u nastavku.

```
// ispis

create done. dynamic semaphore value = 0.
msh >t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 1
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 2
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 3
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 4
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 5
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 6
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 7
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 8
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 9
t1 release a dynamic semaphore.
t2 take a dynamic semaphore. number = 10
```

2.4.2 Mutex

Međusobno isključivi (engl. ***mut**ually **ex**clusive*) pristup dijeljenim resursima se u *RT-Thread* operativnom sistemu postiže korištenjem *mutex* objekata. Ako se za *semaphore* objekat može reći da je poopštenje *mutexa*, onda se *mutex* može opisati kao *semaphore* čija je vrijednost jednaka 1. To znači da samo jedna nit može istovremeno imati pristup određenom resursu. Oslobađanje semafora može izvršiti bilo koja nit, dok *mutex* može osloboditi samo ona nit koja ga je zauzela.

Mutex može biti *zaključan* ili *otključan*, odnosno može imati samo dva stanja. Za nit koja zaključa *mutex* se kaže da ima vlasništvo nad mutexom, te da ga otključavanjem gubi. Dok jedna nit zauzima mutex, druge niti koje pokušavaju dobiti pristup istom ga neće dobiti sve dok se mutex ne otključa. Definicija *mutex* objekta je data u prilogu B.5.

Nasljeđivanje prioriteta

Ranije je naglašeno da je algoritam raspoređivanja niti: preemptivni višenitni algoritam baziran na prioritetima. U slučaju korištenja *mutexa*, mogu se javiti problemi vezani za *inverziju prioriteta*. Npr. neka imamo dvije niti **A** i **C** različitih prioriteta i neka je nit **C** zaključala mutex, ukoliko nit **A** dođe u *ready* stanje, prema algoritmu raspoređivanja - nit **A** će se izvršavati. Problem nastaje ukoliko nit A pokuša zaključati isti mutex koji je zaključala nit C, odnosno ukoliko želi pristupiti zajedničkom resursu.

Ovaj problem se rješava pomoću algoritma nasljeđivanja prioriteta *engl. priority inheritance*. U RT-Thread operativnom sistemu se u ovakvim slučajevima, kada npr. nit A pokuša zaključati mutex koji je zaključala nit C, **mijenja** prioritet niti **C** tako da ima isti prioritet kao nit **A**, sve dok se mutex ne otključa/oslobodi. U nastavku programa sve niti imaju svoje inicijalne vrijednosti prioriteta. *Mutex* objekat sadrži atribut `original_priority` koji čuva informaciju o inicijalnoj vrijednosti prioriteta niti ukoliko dođe do *nasljeđivanja prioriteta*.

Kreiranje/inicijalizacija mutexa

Da bi se **kreirao** *mutex* dinamičkog tipa, koristi se funkcija `rt_mutex_create` za koju su definicija i opis argumenata dati u nastavku.

```
rt_mutex_t rt_mutex_create (const char* name,
                            rt_uint8_t flag);
```

- Argumenti:
  - name - ime mutexa
  - flag - zastavica koja određuje način rada *mutexa*
- Moguće povratne vrijednosti:
  - *mutex handle* - mutex uspješno kreiran
  - `RT_NULL` - kreiranje neuspješno

Argument *flag* može imati vrijednosti `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`. Definiranje ovog parametra je važno zato što isti određuje način rada mutexa u slučaju kada više niti čeka pristup na resurse. Prioritetni način rada podrazumijeva da će pristup mutexu dobiti ona nit koja u redu čekanja ima najviši prioritet, dok će u *FIFO* načinu rada, ukoliko više niti pokušava dobiti pristup mutexu, prednost dobiti ona koja je ranije poslala zahtjev. Za **brisanje dinamički kreiranog mutexa**, koristi se funkcija `rt_mutex_delete` koja prima *mutex handle* a vraća `RT_EOK` u slučaju uspješnog brisanja.

```
rt_err_t rt_mutex_delete (rt_mutex_t mutex);
```

Inicijalizacija *mutexa* statičkog tipa je moguća korištenjem funkcije `rt_mutex_init`.

```
rt_err_t rt_mutex_init (rt_mutex_t mutex,
                        const char* name,
                        rt_uint8_t flag);
```

- Argumenti:
  - mutex - *mutex handle*, pokazuje na memorijski blok mutex objekta
  - name - ime mutexa
  - flag - zastavica koja određuje način rada *mutexa*
- Moguće povratne vrijednosti:
  - `RT_NULL` - kreiranje neuspješno

Razdvajanje pokazivača na mutex i samog mutex objekta je moguće izvršiti pomoću funkcije `rt_mutex_detach` koja je data u nastavku, a koja prima i vraća iste argumente kao ranije spomenuta `rt_mutex_delete` funkcija.

```
rt_err_t rt_mutex_detach (rt_mutex_t mutex);
```

Zaključavanje i otključavanje mutexa

Mutex može biti zauzet/zaključan od strane samo jedne niti istovremeno. Ako je mutex slobodan, pokušaj njegovog zaključavanja od strane niti će biti uspješan. Ukoliko zaključani mutex kontrolira druga nit, nit koja pokušava preuzeti kontrolu će suspendirati svoj rad i čekati sve dok se mutex ne otključa ili dok ne istekne specificirani period vremena. Treba napomenuti da mutex može otključati samo nit koja ga je ranije zaključala. Ukoliko dođe do ranije spomenute promjene prioriteta niti u slučaju *nasljeđivanja prioriteta*, prilikom oslobađanja mutexa prioritet niti se vraća na staru vrijednost. Mutex se **zaključava** pomoću funkcije `rt_mutex_take`, a **otključava** pomoću funkcije `rt_mutex_release`. Definicije i opisi argumenata su dati u nastavku.

```
// zaklucavanje mutexa
rt_err_t rt_mutex_take (rt_mutex_t mutex,
                        rt_int32_t time);
```

- Argumenti (zaključavanje *mutexa*):
  - mutex - *handle* mutex objekta
  - time - vrijeme čekanja na oslobađanje mutexa u slučaju da je već zauzet
- Moguće povratne vrijednosti:
  - `RT_EOK` - uspješno zaključan mutex
  - `-RT_ETIMEOUT` - isteklo vrijeme čekanja
  - `-RT_ERROR` - neuspješno zaključavanje

```
// otkljucavanje mutexa
rt_err_t rt_mutex_release (rt_mutex_t mutex);
```

- Argumenti (otključavanje *mutexa*):
  - mutex - *handle* mutex objekta
- Moguće povratne vrijednosti:
  - `RT_EOK` - uspješno otključavanje mutexa

Primjer mutex aplikacije

U nastavku je dat primjer korištenja objekata tipa *mutex*. Aplikacija sadrži dvije niti sa različitim prioritetima. Kreirana je i varijabla `number` koja se modificira u obje kreirane niti radi prikaza funkcionalnosti *mutexa*. Zadatak obje niti je da zaključa mutex (ako je slobodan) i poveća vrijednost varijable za 1 (u slučaju niti 1), odnosno udvostruči vrijednost varijable (u slučaju niti 2). Programski kod je prikazan u nastavku.

Program 2.9: Mutex sample

```
#include <rtthread.h>

#define THREAD_PRIORITY         8
#define THREAD_TIMESLICE        5

/* Pointer to the mutex */
static rt_mutex_t dynamic_mutex = RT_NULL;
static rt_uint8_t number = 0;
rt_uint8_t max = 8;

void print_priorities (void);

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
static void rt_thread_entry1(void *parameter)
{
      while(1)
      {
          rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
          rt_thread_mdelay(10);
          rt_kprintf("Thread 1 obtains the mutex. //+1 \n");

          print_priorities();

          number++;
          rt_kprintf("Number value: %d \n", number);

          rt_mutex_release(dynamic_mutex);
          rt_kprintf("Thread 1 releases the mutex. \n");

          print_priorities();

          if(number>=max)
              {
              rt_kprintf("* Thread 1 returns. *\n");
              return;
              }
       }
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
static void rt_thread_entry2(void *parameter)
{
      while(1)
      {
```

```
49
50              rt_mutex_take(dynamic_mutex, RT_WAITING_FOREVER);
51              rt_thread_mdelay(10);
52              rt_kprintf("Thread␣2␣obtains␣the␣mutex.␣//*2␣\n");
53
54              number*=2;
55              rt_kprintf("Number␣value:␣%d␣\n", number);
56
57              rt_mutex_release(dynamic_mutex);
58              rt_kprintf("Thread␣2␣releases␣the␣mutex.␣\n");
59
60              if(number>=max)
61                  {
62                  rt_kprintf("*␣Thread␣2␣returns.␣*\n");
63                  return;
64                  }
65          }
66  }
67
68  void print_priorities (void)
69      {
70          rt_kprintf("Thread␣1␣priority:␣%d␣\n", thread1.
               current_priority);
71          rt_kprintf("Thread␣2␣priority:␣%d␣\n", thread2.
               current_priority);
72      }
73
74  /* Initialization of the mutex sample */
75  int main(void)
76  {
77
78      /* Initial value */
79      rt_kprintf("Number␣value:␣%d␣\n", number);
80
81      /* Create a dynamic mutex */
82      dynamic_mutex = rt_mutex_create("dmutex", RT_IPC_FLAG_PRIO);
83      if (dynamic_mutex == RT_NULL)
84      {
85          rt_kprintf("create␣dynamic␣mutex␣failed.\n");
86          return -1;
87      }
88
89      rt_thread_init(&thread1,
90                     "thread1",
91                     rt_thread_entry1,
92                     RT_NULL,
93                     &thread1_stack[0],
94                     sizeof(thread1_stack),
95                     THREAD_PRIORITY, THREAD_TIMESLICE);
96      rt_thread_startup(&thread1);
97
98      rt_thread_init(&thread2,
99                     "thread2",
100                    rt_thread_entry2,
101                    RT_NULL,
102                    &thread2_stack[0],
103                    sizeof(thread2_stack),
104                    THREAD_PRIORITY-1, THREAD_TIMESLICE);
```

```
105     rt_thread_startup(&thread2);
106     return 0;
107 }
```

Obje niti pristupaju dijeljenoj varijabli, ali ne istovremeno - nit 1 vrijednost varijable uvećava za jedan, dok nit 2 udvostručava trenutnu vrijednost. Dok je mutex zaključan u jednoj niti, druga nit se nalazi u *ready* stanju i spremna je preuzeti izvršavanje nakon otključavanja mutexa. Radi lakšeg praćenja izvršavanja, ispisuje se odgovarajući tekst nakon važnih dijelova programa, kao što su zaključavanje i otključavanje mutexa.

Thread2 je nit koja ima veći prioritet u odnosu na *Thread1*, stoga je važno primijetiti da se prilikom zaključavanja mutexa od strane niti 1 i pokušaja niti 2 da zauzme mutex, dešava nasljeđivanje prioriteta, pri kojem sada obje niti imaju isti nivo prioriteta, sve dok se *mutex* ne otključa. Na kraju nema nasljeđivanja prioriteta, zato što je *Thread2* završio sa radom i raspoređivač ga ne uzima u obzir prilikom dodjele *CPU* resursa. Ispis je prikazan u nastavku.

```
// ispis

Number value: 0
Thread 1 obtains the mutex. //+1
Thread 1 priority: 7
Thread 2 priority: 7
Number value: 1
Thread 1 releases the mutex.
Thread 1 priority: 8
Thread 2 priority: 7
Thread 2 obtains the mutex. //*2
Number value: 2
Thread 2 releases the mutex.
Thread 1 obtains the mutex. //+1
Thread 1 priority: 7
Thread 2 priority: 7
Number value: 3
Thread 1 releases the mutex.
Thread 1 priority: 8
Thread 2 priority: 7
Thread 2 obtains the mutex. //*2
Number value: 6
Thread 2 releases the mutex.
Thread 1 obtains the mutex. //+1
Thread 1 priority: 7
Thread 2 priority: 7
Number value: 7
Thread 1 releases the mutex.
Thread 1 priority: 8
Thread 2 priority: 7
Thread 2 obtains the mutex. //*2
Number value: 14
Thread 2 releases the mutex.
* Thread 2 returns. *
Thread 1 obtains the mutex. //+1
Thread 1 priority: 8
Thread 2 priority: 7
Number value: 15
Thread 1 releases the mutex.
Thread 1 priority: 8
Thread 2 priority: 7
* Thread 1 returns. *
```

2.4.3 Events

Objekat tipa događaja (engl. *events*) u RT-Thread operativnom sistemu podrazumijeva skup (engl. *set*) događaja. Skup događaja se može upotrijebiti za tipove sinhronizacije niti kao što su *one-to-many* i *many-to-many*.

Veza između niti i događaja može biti takva da bilo koji od odgovarajućih događaja može *probuditi* nit ili način rada može podrazumijevati da samo određeni, istovremeno aktivni događaji, pokreću izvršavanje posmatrane niti. Kolekcija višestrukih događaja se može predstaviti 32-bitnom cjelobrojnom vrijednošću bez predznaka.

Logička funkcija **ILI** (engl. *OR*) u smislu objekta tipa *event* predstavlja *neovisnu sinhronizaciju*. Da bismo ilustrirali ovaj tip sinhronizacije, navest' ćemo primjer korištenja autobusnog vozila. Neka osoba želi doći na odredište vozeći se autobusom i neka postoje 3 autobusa koja voze na to odredište. Prvi autobus koji dođe na stanicu na kojoj se osoba nalazi će biti dovoljan da ta osoba stigne na odredište, bez da osoba čeka dolazak ostala dva. Ova funkcionalnost se postiže pomoću identifikatora `RT_EVENT_FLAG_OR`.

Logička funkcija **I** (engl. *OR*) u smislu objekta tipa *event* predstavlja *asocijativnu sinhronizaciju*. U primjeru korištenja autobusnog vozila, neka je cilj jedne osobe, koja se nalazi na autobusnoj stanici, da otputuje na određenu destinaciju sa još jednom osobom. Osoba će ostvariti svoj cilj ako su ispunjena dva uslova: da osoba sa kojom putuje i autobus koji vozi na željeno odredište dođu na stanicu. Ova funkcionalnost se postiže pomoću identifikatora `RT_EVENT_FLAG_AND`.

Definicija *event set* objekta je data u prilogu B.6. Skup događaja (engl. *event set*) u RT-Thread operativnom sistemu ima sljedeće karakteristike:

1. Događaji su vezani samo za niti i neovisni su jedni od drugih. Svaka nit može imati 32 *event flaga*, koji predstavljaju bite u 32-bitnom neoznačenom (engl. *unsigned*) broju, kao što je prikazano na slici (2.9).

2. Događaji se mogu koristiti za sinhronizaciju, ali ne i za prenos podataka između niti.

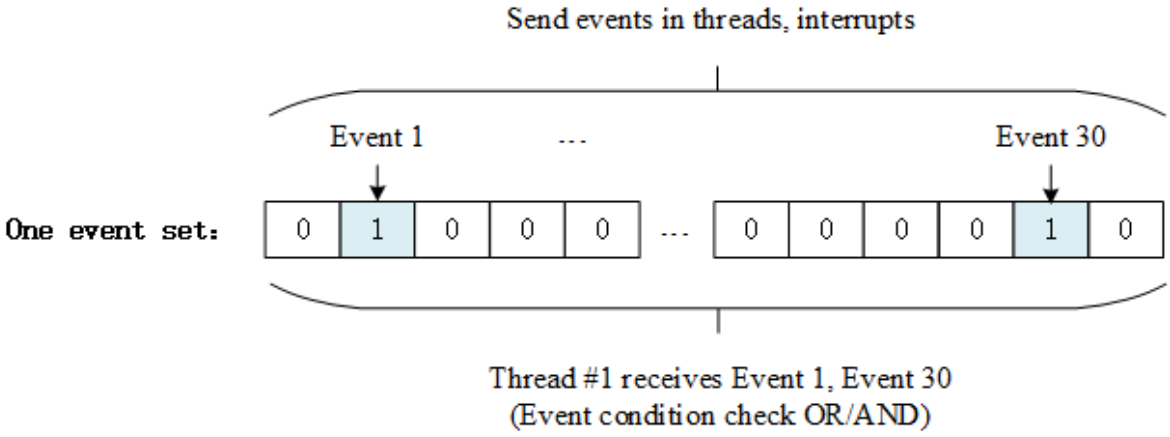
3. Događaji se ne smiještaju u redove tj. višestruko slanje istog događaja će se registrirati samo jednom od strane niti (kao da je događaj poslan samo jednom). Nakon registriranja *flaga* odgovarajućeg događaja, isti se može anulirati pomoću identifikatora `RT_EVENT_FLAG_CLEAR`.

Kreiranje/inicijalizacija događaja

Funkcije za **kreiranje** (dinamički tip) i **inicijalizaciju** (statički tip) skupa događaja (engl. *event set*) su date u nastavku sa svojim definicijama i opisom argumenata i povratnih vrijednosti. Za kreiranje dinamičkog skupa događaja se koristi funkcija `rt_event_create`, a za inicijalizaciju skupa događaja statičkog tipa se koristi funkcija `rt_event_init`.

```
rt_event_t rt_event_create (const char* name,
                            rt_uint8_t flag);
```

- Argumenti:
  - name - ime skupa događaja



Slika 2.9: Prikaz *flagova* u *event setu*

  - flag - zastavica za prioritetni `RT_IPC_FLAG_PRIO`, odnosno `RT_IPC_FLAG_FIFO` za FIFO (engl. *first-in first-out*) režim rada

- Moguće povratne vrijednosti:
  - `RT_NULL` - kreiranje neuspješno
  - event set handler - kreiranje uspješno

```
rt_err_t rt_event_init (rt_event_t event,
                        const char* name,
                        rt_uint8_t flag);
```

- Argumenti:
  - event - *event set object handle*
  - name - ime skupa događaja
  - flag - zastavica za prioritetni `RT_IPC_FLAG_PRIO`, odnosno `RT_IPC_FLAG_FIFO` za FIFO (engl. *first-in first-out*) režim rada

- Moguće povratne vrijednosti:
  - `RT_EOK` - kreiranje uspješno

Brisanje dinamičkog i statičkog tipa objekta *event set*, kada nam u programu više nisu potrebni, je moguće učiniti pomoću funkcija `rt_event_delete` i `rt_event_detach`, respektivno. Obje funkcije primaju *event set handle*, a vraćaju `RT_EOK` u slučaju uspješnog brisanja objekta tipa događaja.

```
//dinamicki tip
rt_err_t rt_event_delete(rt_event_t event);

//staticki tip
rt_err_t rt_event_detach(rt_event_t event);
```

Slanje i primanje događaja

Slanje događaja se postiže funkcijom `rt_event_send`.

```
rt_err_t rt_event_send (rt_event_t event,
                        rt_uint32_t set);
```

- Argumenti:
  - event - *handle* objekta tipa događaj
  - set - vrijednost *flaga* jednog ili više poslanih događaja
- Moguće povratne vrijednosti:
  - `RT_EOK` - slanje uspješno

Kernel koristi 32-bitnu cjelobrojnu vrijednost da bi identificirao *event set* objekat, pri čemu svaki bit te vrijednosti može predstavljati neki događaj. Kernel operativnog sistema će *probuditi*, odnosno postaviti nit u stanje izvršavanja u zavisnosti od parametra *option* koji može imati vrijednost *OR* (`RT_EVENT_FLAG_OR`) ili *AND* (`RT_EVENT_FLAG_AND`). Kada *option* ima vrijednost *OR*, nit će biti aktivirana kada bude aktivan bilo koji od specificiranih događaja. S druge strane, nit se neće aktivirati sve dok svi specificirani događaji ne budu poslani, onda kada *option* ima vrijednost *AND*. Funkcija za **primanje događaja** je `rt_event_recv`. Njena definicija i opis argumenata i povratnih vrijednosti su dati u nastavku.

```
rt_err_t rt_event_recv (rt_event_t event,
                        rt_uint32_t set,
                        rt_uint8_t option,
                        rt_int32_t timeout,
                        rt_uint32_t* recved);
```

- Argumenti:
  - event - *event set object handle*
  - set - određuje koji događaji će biti okidač za izvršavanje niti
  - option - opcije za registriranje događaja
  - timeout - vremenski period čekanja na prijem događaja
  - recved - pokazivač na primljeni događaj
- Moguće povratne vrijednosti:
  - `RT_EOK` - događaj uspješno primljen
  - `-RT_ETIMEOUT` - isteklo vrijeme za prijem
  - `-RT_ERROR` - greška

Za neovisnu sinhronizaciju (logičko *ILI*) je u parametar option potrebno postaviti identifikator `RT_EVENT_FLAG_OR`, dok je za asocijativnu sinhronizaciju (logičko *I*) potrebno postaviti `RT_EVENT_FLAG_AND`. Jedan od ova dva parametra su obavezna, dok se opcionalno, korištenjem C logičke funkcije "ILI" (oznaka vertikalne linije |) može omogućiti brisanje bita događaja sa identifikatorom `RT_EVENT_FLAG_CLEAR`.

Prilikom nailaska na funkciju `rt_event_recv` unutar niti, moguće su sljedeće situacije:

- Ukoliko se željeni događaj već dogodio, onda se nastavlja izvršavanje niti u kojoj se ovaj poziv nalazi i (opcionalno) briše bit zastavice u skupu događaja ukoliko je aktivan identifikator `RT_EVENT_FLAG_CLEAR`.
- Ukoliko se željeni događaj nije dogodio, onda se nit suspendira i čeka slanje odgovarajućeg događaja. Čekanje traje određeno vrijeme, specificirano parametrom *timeout*. Ukoliko to vrijeme istekne, funkcija vraća `-RT_ETIMEOUT`.

Primjer korištenja događaja

Kreirane su dvije niti različitih prioriteta. *Thread1* (viši prioritet) čeka na pojavu događaja i obrađuje ih. *Thread2* šalje događaje koje prva nit obrađuje.

Thread1 prva započinje izvršavanje, međutim nailazi na poziv funkcije za prijem događaja, te je uslov za nastavak izvršavanja ove niti pojava jednog od dva događaja: `EVENT_FLAG3` ili `EVENT_FLAG5` (logička funkcija *ili*).

Prva nit se suspendira i prepušta izvršavanje niti *Thread2* koja šalje događaj `EVENT_FLAG3`. Sadržaj *event* skupa je dat u (2.1). Kako je uslov za nastavak rada prve niti ispunjen, *Thread1* ispisuje vrijednost događaja, koji je u ovom slučaju jednak broju 8, što je rezultat operacije `#define EVENT_FLAG3 (1 << 3)`, odnosno pomijeranja bita ulijevo za tri pozicije. Bit registriranog događaja se anulira zbog *clear* identifikatora.

Bit	2^{31}	2^{30}	...	2^5	2^4	2^3	2^2	2^1	2^0
Set	0	0	...	0	0	1	0	0	0

(2.1)

Thread1 nailazi na novu prepreku. Sada je za nastavak izvršavanja potrebno da istovremeno budu aktivni događaji `EVENT_FLAG3` i `EVENT_FLAG5`. Ova nit se ponovo suspendira. *Thread2* šalje `EVENT_FLAG5`. Objekat tipa *event* sadrži atribut naziva *set* koji sada ima vrijednost 32 (rezultat operacije `#define EVENT_FLAG5 (1 << 5)`). Binarno gledajući, atribut *set* ima jedinicu samo na poziciji 2^5. Sadržaj *event* seta u ovom trenutku je dat u (2.2).

Bit	2^{31}	2^{30}	...	2^5	2^4	2^3	2^2	2^1	2^0
Set	0	0	...	1	0	0	0	0	0

(2.2)

Thread1 ipak ne nastavlja svoje izvršavanje zato što funkcija za primanje događaja očekuje da oba događaja (*flag3* i *flag5*) budu aktivna. Treba napomenuti da je *flag3* bio ranije aktivan, ali je anuliran nakon prvog poziva funkcije `rt_event_recv`. Da bi se nastavilo izvršavanje, atribut *set* treba imati vrijednost koja u binarnoj reprezentaciji ima logičke jedinice samo na pozicijama 2^5 i 2^3. Dakle, vrijednost varijable *set* mora biti $2^5 + 2^3 = 32 + 8 = 40$. Odnosno, *event set* mora imati sadržaj kao u (2.3).

Bit	2^{31}	2^{30}	...	2^5	2^4	2^3	2^2	2^1	2^0
Set	0	0	...	1	0	1	0	0	0

(2.3)

Nakon što *Thread2* konačno pošalje i *flag3*, *Thread1* prelazi u aktivno stanje i aplikacija završava svoj rad. Programski kod i ispis su prikazani u nastavku.

Program 2.10: Event set sample

```
#include <rtthread.h>

#define THREAD_PRIORITY      9
#define THREAD_TIMESLICE     5

/* Number 1 in hex                              00000001
 * (1 << 3) shift number 1 three times left  00001000 => 8
 * (1 << 5) shift number 1 five times left   00100000 => 32
 */

#define EVENT_FLAG3 (1 << 3)
#define EVENT_FLAG5 (1 << 5)

/* Event control block */
static struct rt_event event;

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;

/* Thread 1 entry function*/
static void thread1_recv_event(void *param)
{
    rt_uint32_t e;
    /* The first time the event is received, either event 3 or event
        5 can trigger thread 1, clearing the event flag after
       receiving */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                      RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
                      RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: OR recv event 0x%x\n", e);
    }

    rt_kprintf("thread1: delay 1s to prepare the second event\n");
    rt_thread_mdelay(1000);

    /* The second time the event is received, both event 3 and event
        5 can trigger thread 1, clearing the event flag after
       receiving */
    if (rt_event_recv(&event, (EVENT_FLAG3 | EVENT_FLAG5),
                      RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                      RT_WAITING_FOREVER, &e) == RT_EOK)
    {
        rt_kprintf("thread1: AND recv event 0x%x\n", e);
    }
    rt_kprintf("thread1 leave.\n");
}


ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
```

```
50
51  /* Thread 2 Entry */
52  static void thread2_send_event(void *param)
53  {
54      rt_kprintf("thread2: send event3\n");
55      rt_event_send(&event, EVENT_FLAG3);
56      rt_kprintf("Event set value: %d\n", event.set);
57      rt_thread_mdelay(200);
58
59      rt_kprintf("thread2: send event5\n");
60      rt_event_send(&event, EVENT_FLAG5);
61      rt_kprintf("Event set value: %d\n", event.set);
62      rt_thread_mdelay(200);
63
64      rt_kprintf("thread2: send event3\n");
65      rt_event_send(&event, EVENT_FLAG3);
66      rt_kprintf("Event set value: %d\n", event.set);
67      rt_kprintf("thread2 leave.\n");
68  }
69
70  int main(void)
71  {
72      rt_err_t result;
73
74      rt_kprintf("Starting program.\n");
75      rt_kprintf("EVENT_FLAG3 = %d ; EVENT_FLAG5 = %d\n", EVENT_FLAG3,
            EVENT_FLAG5);
76
77      /* Initialize event object */
78      result = rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
79      if (result != RT_EOK)
80      {
81          rt_kprintf("init event failed.\n");
82          return -1;
83      }
84
85      rt_thread_init(&thread1,
86                     "thread1",
87                     thread1_recv_event,
88                     RT_NULL,
89                     &thread1_stack[0],
90                     sizeof(thread1_stack),
91                     THREAD_PRIORITY - 1, THREAD_TIMESLICE);
92      rt_thread_startup(&thread1);
93
94      rt_thread_init(&thread2,
95                     "thread2",
96                     thread2_send_event,
97                     RT_NULL,
98                     &thread2_stack[0],
99                     sizeof(thread2_stack),
100                    THREAD_PRIORITY, THREAD_TIMESLICE);
101     rt_thread_startup(&thread2);
102
103     return 0;
104 }
```

```
// ispis

Starting program.
EVENT_FLAG3 = 8 ; EVENT_FLAG5 = 32
thread2: send event3
thread1: OR recv event 0x8
thread1: delay 1s to prepare the second event
Event set value: 0
thread2: send event5
Event set value: 32
thread2: send event3
Event set value: 40
thread2 leave.
thread1: AND recv event 0x28
thread1 leave.
```

2.5 Komunikacija između niti

U razvoju softvera za ugradbene sisteme gdje se ne koristi operativni sistem kao veza između korisničkog sučelja i hardvera, uglavnom se koriste globalne varijable za komunikaciju između različitih funkcija. Npr. jedna funkcija mijenja vrijednost neke globalne varijable uslijed određenih operacija, dok druga čita vrijednost te varijable i na osnovu toga izvršava određene akcije.

Proslijeđivanje informacija između različitih niti u RT-Thread operativnom sistemu se može realizirati pomoću mehanizama kao što su poštanski sandučić (engl. *mailbox*), red poruka (engl. *message queue*) i signali (engl. *signals*).

2.5.1 Mailbox

Poštanski sandučić (engl. *mailbox*) je tipični način za komunikaciju između niti u *real-time* operativnim sistemima. Npr. ako imamo dvije niti, pri čemu je zadatak prve niti da detektuje stanje nekog tastera, a zadatak druge niti je da mijenja stanje *LED* (engl. *light emitting diode*) indikatora na osnovu stanja tastera. Bez korištenja globalnih varijabli, informaciju o stanju tastera prva nit može poslati drugoj niti pomoću *mailbox* mehanizma. Definicija *mailbox* objekta je data u prilogu B.7.

Mehanizam rada mailboxa

Mailbox karakterizira mali *overhead* i visoka efikasnost. Svaka poruka u *mailboxu* ima veličinu od $4B$. Ukoliko nit pošalje poruku u mailbox, poruka će se kopirati u mailbox ukoliko isti nije pun. Ukoliko je mailbox pun, postavljanjem parametra *timeout* se postiže čekanje sve dok se ne oslobodi mjesto za poruku ili dok ne istekne definirano vrijeme.

Kada nit prima poruku iz *mailboxa*, poruka se kopira u prijemni *cache* u slučaju da mailbox nije prazan. Kada je mailbox prazan, nit može suspendirati svoje izvršavanje ili čekati određeni vremenski period da se poruka pojavi.

Kreiranje/inicijalizacija mailboxa

Dinamičko kreiranje objekta tipa mailbox, odnosno **inicijalizacija** mailboxa statičkog tipa se postiže korištenjem funkcija `rt_mb_create` i `rt_mb_init`, respektivno.

```
// dinamicki tip
rt_mailbox_t rt_mb_create (const char* name,
                           rt_size_t size,
                           rt_uint8_t flag);
```

- Argumenti:
  - name - ime *mailboxa*
  - size - kapacitet mailboxa
  - flag - *mailbox* zastavica, moguće vrijednosti ovog argumenta `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`
- Moguće povratne vrijednosti:
  - `RT_NULL` - kreiranje neuspješno
  - *mailbox object handle* - kreiranje uspješno

Na ovaj način se dinamički alocira *mailbox* u memoriji. Veličina koju zauzma kreirani objekat predstavlja umnožak fiksne vrijednosti veličine poruke od $4B$ i vrijednosti *size* argumenta.

```
// staticki tip
rt_err_t rt_mb_init(rt_mailbox_t mb,
                    const char* name,
                    void* msgpool,
                    rt_size_t size,
                    rt_uint8_t flag)
```

- Argumenti:
  - mb - *mailbox object handle*
  - name - ime *mailboxa*
  - msgpool - pokazivač na buffer u memoriji gdje se smještaju poruke
  - size - kapacitet *mailboxa*
  - flag - *mailbox* zastavica, moguće vrijednosti ovog argumenta `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`
- Moguće povratne vrijednosti:
  - `RT_EOK` - inicijalizacija uspješna

Prilikom inicijalizacije *mailbox* objekta, pojavljuje se argument *msgpool*. Ako je broj bita u bufferu na koji pokazuje *msgpool* jednak N, onda kapacitet mailboxa treba biti $N/4$, zato što jedna poruka ima fiksnu dužinu od $4B$.

Brisanje u slučaju dinamičke alokacije, odnosno **odvajanje** *mailbox object handlera* od *mailboxa* se može izvršiti korištenjem funkcija `rt_mb_delete` i `rt_mb_detach`, respektivno. One primaju *mailbox handler* kao argument i vraćaju identifikator `RT_EOK` kao signalizaciju da su željene radnje izvršene.

```
// dinamicki tip
rt_err_t rt_mb_delete (rt_mailbox_t mb);

// staticki tip
rt_err_t rt_mb_detach (rt_mailbox_t mb);
```

Slanje i primanje pošte

Nit može **slati** poštu (engl. *mail*), odnosno poruke drugim nitima koristeći funkciju `rt_mb_send`.

```
rt_err_t rt_mb_send (rt_mailbox_t mb,
                     rt_uint32_t value);
```

- Argumenti:
  - mb - *mailbox handle*
  - value - sadržaj poruke
- Moguće povratne vrijednosti:
  - `RT_EOK` - poruka uspješno poslana
  - `-RT_EFULL` - *mailbox* pun

Poslana poruka može biti bilo koji podatak formatiran na 32-bita, cjelobrojna vrijednost ili pokazivač na *buffer*. Kada je *mailbox* pun, nit će dobiti povratnu vrijednost `-RT_EFULL`.

Slanje poruka sa čekanjem je moguće postići korištenjem funkcije `rt_mb_send_wait`.

```
rt_err_t rt_mb_send_wait (rt_mailbox_t mb,
                          rt_uint32_t value,
                          rt_int32_t timeout);
```

Razlika u odnosu na `rt_mb_send` se odnosi na dodatni *timeout* parametar. Nit će, u slučaju da je *mailbox* popunjen, čekati određeni vremenski period - specificiran spomenutim parametrom, da se oslobodi mjesto u poštanskom sandučiću. Ukoliko se to ne desi, funkcija vraća grešku. Mogući identifikatori greške su `-RT_ETIMEOUT` i `-RT_ERROR`.

Primanje pošte unutar niti se postiže korištenjem funkcije `rt_mb_recv`, čija su definicija i opis argumenata i povratnih vrijednosti date u nastavku.

```
rt_err_t rt_mb_recv (rt_mailbox_t mb,
                     rt_uint32_t* value,
                     rt_int32_t timeout);
```

- Argumenti:
  - mb - *handle mailbox* objekta
  - value - specificirana lokacija u koju će se smjestiti primljena vrijednost
  - timeout - vremenski period za koji će nit čekati dok se poruka ne pojavi u *mailboxu*
- Moguće povratne vrijednosti:
  - `RT_EOK` - primanje poruke uspješno
  - `-RT_ETIMEOUT` - poruka nije primljena niti nakon isteka *timeouta*
  - `-RT_ERROR` - neuspješno primanje poruke

Primjer korištenja mailboxa

U okviru aplikacije su kreirane dvije niti koje imaju isti nivo prioriteta. Nit *Thread2* je zadužena za slanje poruka niti *Thread1*. Poruke su zapravo stringovi/nizovi znakova koji su deklarirani unutar niti koja šalje poruke, tako da nit *Thread1* koja prima poruke nema direktan pristup istim s obzirom na to da se ne radi o globalnim varijablama. Treba napomenuti da primjer dat u zvaničnoj *RT-Thread* dokumentaciji na ovom linku koristi navedene stringove kao *globalne varijable*, što nije uvijek preporučljiv pristup kada imamo mogućnost korištenja operativnog sistema u razvoju aplikacije.

Nakon primitka poruke, *Thread1* je ispisuje i nastavlja čekanje nove poruke. Nakon nekoliko slanja i primanja poruka, aplikacija završava svoj rad. Programski kod, kao i ispis su dati u nastavku.

Program 2.11: Mailbox sample

```
1  #include <rtthread.h>
2
3  #define THREAD_PRIORITY      10
4  #define THREAD_TIMESLICE     5
5
6  /* Mailbox control block */
7  static struct rt_mailbox mb;
8  /* Memory pool for mails storage */
9  static char mb_pool[128];
10
11 ALIGN(RT_ALIGN_SIZE)
12 static char thread1_stack[1024];
13 static struct rt_thread thread1;
14
15 /* Thread 1 entry */
16 static void thread1_entry(void *parameter)
17 {
18     char *str;
19
20     while (1)
21     {
```

```
22          rt_kprintf("thread1: try to recv a mail\n");
23
24          /* Receive mail from the mailbox */
25          if (rt_mb_recv(&mb, (rt_uint32_t *)&str, RT_WAITING_FOREVER)
                == RT_EOK)
26          {
27              rt_kprintf("thread1: get a mail from mailbox, the
                    content:%s\n", str);
28              //if (str == mb_str3)
29              if (rt_strcmp(str, "over")==0)
30                  break;
31
32              /* Delay 100ms */
33              rt_thread_mdelay(100);
34          }
35      }
36      /* Executing the mailbox object detachment */
37      rt_mb_detach(&mb);
38  }
39
40  ALIGN(RT_ALIGN_SIZE)
41  static char thread2_stack[1024];
42  static struct rt_thread thread2;
43
44  /* Thread 2 entry*/
45  static void thread2_entry(void *parameter)
46  {
47      rt_uint8_t count = 0;
48
49      static char mb_str1[] = "I'm a mail!";
50      static char mb_str2[] = "this is another mail!";
51      static char mb_str3[] = "over";
52
53      count = 0;
54      while (count < 10)
55      {
56          count ++;
57          //rt_kprintf("count & 0x1 = %d\n", count & 0x1);
58          if (count & 0x1)
59          {
60              /* Send the mb_str1 address to the mailbox */
61              rt_mb_send(&mb, (rt_uint32_t)&mb_str1);
62          }
63          else
64          {
65              /* Send the mb_str2 address to the mailbox */
66              rt_mb_send(&mb, (rt_uint32_t)&mb_str2);
67          }
68
69          /* Delay 200ms */
70          rt_thread_mdelay(200);
71      }
72
73      /* Send mail to inform thread 1 that thread 2 has finished
            running */
74      rt_mb_send(&mb, (rt_uint32_t)&mb_str3);
75  }
76
```

```
77  int main(void)
78  {
79      rt_err_t result;
80
81      /* Initialize a mailbox */
82      result = rt_mb_init(&mb,
83                          "mbt",                      /* Name is mbt
                                */
84                          &mb_pool[0],                /* The memory
                               pool used by the mailbox is mb_pool */
85                          sizeof(mb_pool) / 4,        /* The number of
                                messages in the mailbox because a
                               message occupies 4 bytes */
86                          RT_IPC_FLAG_FIFO);          /* Thread
                               waiting in FIFO approach */
87      if (result != RT_EOK)
88      {
89          rt_kprintf("init␣mailbox␣failed.\n");
90          return -1;
91      }
92
93      rt_thread_init(&thread1,
94                     "thread1",
95                     thread1_entry,
96                     RT_NULL,
97                     &thread1_stack[0],
98                     sizeof(thread1_stack),
99                     THREAD_PRIORITY, THREAD_TIMESLICE);
100     rt_thread_startup(&thread1);
101
102     rt_thread_init(&thread2,
103                    "thread2",
104                    thread2_entry,
105                    RT_NULL,
106                    &thread2_stack[0],
107                    sizeof(thread2_stack),
108                    THREAD_PRIORITY, THREAD_TIMESLICE);
109     rt_thread_startup(&thread2);
110     return 0;
111 }
```

```
// ispis

thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail from mailbox, the content:over
```

2.5.2 Message queue

Red poruka (engl. *message queue*) u *RT-Thread* operativnom sistemu je mehanizam za asinhronu komunikaciju između niti. Predstavlja ekstenziju objekta *mailbox*. Red poruka može primati poruke proizvoljne dužine od strane niti ili prekidnih rutina za razliku od *mailboxa* gdje su dužine poruka fiksne. Može se reći da je *mailbox* objekat efikasniji, a *message queue* fleksibilniji mehanizam komunikacije između niti.

Kada je red poruka prazan, niti koje pokušaju čitanje poruke mogu suspendirati svoje izvršavanje i ponovo započeti svoje izvršavanje onda kada se poruka pojavi u *message queue* objektu kako bi je pročitali i procesirali. Red poruka radi na principu *first-in first-out*, što podrazumijeva da će poruka koja ranije bude poslana, ranije biti i pročitana, osim u slučaju *hitnog* slanja, o kojem će više informacija biti dato u nastavku.

Message queue objekat se sastoji od nekoliko važnih elemenata. Poruke se smještaju u povezanu listu (engl. *linked list*) prilikom slanja. Povezana lista ima svoj početak (engl. *header*) i kraj (engl. *tail*). Postoji i povezana lista poruka u mirovanju (engl. *list of idle message boxes*).

Pri kreiranju objekta tipa *message queue*, lista poruka (koja ima *header* i *tail* pokazivače) je prazna. Kada nit pošalje poruku, jedna *prazna* kutija za poruku (engl. message box) se iz *idle* liste prebacuje u povezanu listu poruka. Informacija koju šalje nit se kopira u prazni *message box*.

Čitanje poruka se uvijek vrši sa *headera* liste, dok se svaka nova poruka smješta na *tail* liste, osim u slučaju hitnog slanja. Definicija *message queue* objekta je data u prilogu B.8.

Kreiranje/inicijalizacija reda poruka

Kreiranje objekta reda poruka dinamičkog tipa je moguće učiniti pomoću funkcije `rt_mq_create`.

```
rt_mq_t rt_mq_create(const char* name,
                     rt_size_t msg_size,
                     rt_size_t max_msgs,
                     rt_uint8_t flag);
```

- Argumenti:
  - name - ime reda poruka
  - msg_size - maksimalna vrijednost dužine poruke izražena u *B*
  - max_msgs - maksimalni broj poruka u redu poruka
  - flag - metod čekanja reda poruka, mogući identifikatori `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`
- Moguće povratne vrijednosti:
  - *message queue object handle* - kreiranje uspješno
  - `RT_NULL` - kreiranje neuspješno

Kada nam dinamički objekat reda poruka više nije potreban, moguće ga je obrisati pomoću funkcije `rt_mq_delete`, koja prima *message queue object handle* i vraća `RT_EOK` u slučaju uspješnog brisanja.

```
rt_err_t rt_mq_delete(rt_mq_t mq);
```

Inicijalizacija statičkog objekta reda poruka se postiže funkcijom `rt_mq_init`.

```
rt_err_t rt_mq_init(rt_mq_t mq,
                    const char* name,
                    void *msgpool,
                    rt_size_t msg_size,
                    rt_size_t pool_size,
                    rt_uint8_t flag);
```

- Argumenti:
  - mq - *message queue object handle*
  - name - ime reda poruka
  - msgpool - pokazivač na *buffer* koji smješta poruke u memoriji
  - msg_size - maksimalna vrijednost dužine poruke izražena u *B*
  - max_msgs - maksimalni broj poruka u redu poruka
  - flag - metod čekanja reda poruka, mogući identifikatori `RT_IPC_FLAG_FIFO` ili `RT_IPC_FLAG_PRIO`

- Moguće povratne vrijednosti:
  - `RT_EOK` - kreiranje uspješno

Odvajanje *message queue* objekta od kernel objekt menadžera se postiže funkcijom `rt_mq_detach`, čiji su ulazni i izlazni parametri isti kao u slučaju funkcije `rt_mq_delete`.

```
rt_err_t rt_mq_detach(rt_mq_t mq);
```

Slanje i primanje poruka

Prilikom ubacivanja poruke u *message queue*, uzima se jedan prazni *message* objekat iz *idle* liste i sadržaj poruke se kopira u njega, a zatim se taj objekat stavlja na *tail* povezane liste poruka. Funkcijsko sučelje za **slanje poruka** predstavlja funkcija `rt_mq_send`.

```
rt_err_t rt_mq_send (rt_mq_t mq,
                     void* buffer,
                     rt_size_t size);
```

- Argumenti:
  - mq - *handle* objekta reda poruka
  - buffer - sadržaj poruke
  - size - veličina poruke, koju treba definirati programer
- Moguće povratne vrijednosti:
  - `RT_EOK` - slanje uspješno
  - `-RT_EFULL` - red poruka je popunjen, odnosno ne postoji više niti jedan *idle message block*
  - `-RT_ERROR` - slanje neuspješno, u slučaju kada je dužina poslane poruke veća od maksimalne dužine poruke u *message queue* objektu

Prethodno navedena funkcija izvršava slanje poruke bez čekanja. To znači da su moguća samo tri ishoda: red poruka je prazan i poruka je uspješno poslana; red poruka je pun i vraćena je greška i veličina poruke je veća od maksimalno dozvoljene. U slučaju kada je red poruka popunjen do kraja, moguće je pokrenuti proces čekanja da se oslobodi jedno mjesto u redu. **Slanje poruke uz čekanje** se postiže funkcijom koja je data u nastavku, a koja ima iste argumente i povratne vrijednosti kao `rt_mq_send`, uz dodatak argumenta *timeout* koji predstavlja period čekanja u slučaju popunjenosti reda poruka. Za slanje poruke uz mogućnost čekanja kreirana je funkcija `rt_mq_send_wait`.

```
rt_err_t rt_mq_send_wait(rt_mq_t mq,
                         const void *buffer,
                         rt_size_t size,
                         rt_int32_t timeout);
```

Sve poruke koje se pošalju kroz *message queue* objekat se smještaju na kraj reda poruka. Time se postiže čitanje sa početka liste po *FIFO* principu. Izuzetak je korištenje **hitnog** (engl. *urgent*) slanja poruka. Poruka poslana pomoću funkcije date u nastavku se ne pozicionira na kraju liste poruka, već na početku - tako da se prilikom sljedećeg čitanja ona pročita prva. Funkcija `rt_mq_urgent` za hitno slanje poruka ima iste argumente i povratne vrijednosti kao `rt_mq_send`.

```
rt_err_t rt_mq_urgent (rt_mq_t mq,
                       void* buffer,
                       rt_size_t size);
```

Primanje poruka korištenjem *message queue* mehanizma u RT-Thread operativnom sistemu je moguće ostvariti samo onda kada red poruka nije prazan. U suptronom će doći do čekanja, kada će nit suspendirati svoje izvršavanje i u zavisnosti od toga da li je primila poruku nastaviti svoj rad ili vratiti grešku.

Prilikom primanja poruke, u niti koja prima poruku se mora definirati mjesto gdje će se ta poruka sačuvati, kao i *message queue* objekat preko kojeg se razmjena poruka vrši. Argument *timeout* određuje vrijeme nakon kojeg će nit vratiti grešku u slučaju da se poruka ne pojavi. Primanje poruke ostvaruje funkcija `rt_mq_recv` sa definicijom i opisom argumenata datim u nastavku.

```
rt_err_t rt_mq_recv (rt_mq_t mq,
                     void* buffer,
                     rt_size_t size,
                     rt_int32_t timeout);
```

- Argumenti:
  - mq - *handle* objekta reda poruka
  - buffer - sadržaj poruke
  - size - veličina poruke
  - timeout - vremenski period nakon koje će funkcija vratiti grešku ukoliko se poruka ne pojavi u redu poruka
- Moguće povratne vrijednosti:
  - `RT_EOK` - primanje poruke uspješno
  - `-RT_EFULL` - red poruka je prazan
  - `-RT_ERROR` - primanje poruke je neuspješno

Primjer korištenja reda poruka

Aplikacija data u nastavku se sastoji od dvije niti. *Thread2* je nit koja je zadužena za slanje poruka prema niti *Thread1*. Poruka u opštem slučaju može biti bilo šta, a ovdje se vrši slanje

karaktera, odnosno slova 'A', 'B', 'C' itd. Pored običnog slanja poruka, realizirano je i *hitno* slanje poruka, pri kojem urgentna poruka dolazi na *početak* reda poruka i dobiva najviši prioritet za čitanje. Programski kod je prikazan u nastavku.

Program 2.12: Message queue sample

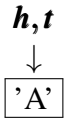
```
#include <rtthread.h>

/* Message queue control block */
static struct rt_messagequeue mq;
/* The memory pool used to place messages in the message queue */
static rt_uint8_t msg_pool[2048];

ALIGN(RT_ALIGN_SIZE)
static char thread1_stack[1024];
static struct rt_thread thread1;
/* Thread 1 entry function */
static void thread1_entry(void *parameter)
{
    char buf = 0;
    rt_uint8_t cnt = 0;

    while (1)
    {
        /* Receive messages from the message queue */
        if (rt_mq_recv(&mq, &buf, sizeof(buf), RT_WAITING_FOREVER)
            == RT_EOK)
        {
            rt_kprintf("thread1: recv msg from msg queue, the
                content:%c\n", buf);
            if (cnt == 5)
            {
                break;
            }
        }
        /* Delay 50ms */
        cnt++;
        rt_thread_mdelay(50);
    }
    rt_kprintf("thread1: detach mq \n");
    rt_mq_detach(&mq);
}

ALIGN(RT_ALIGN_SIZE)
static char thread2_stack[1024];
static struct rt_thread thread2;
/* Thread 2 entry */
static void thread2_entry(void *parameter)
{
    int result;
    char buf = 'A';
    rt_uint8_t cnt = 0;

    while (1)
    {
        if (cnt == 4)
        {
            /* Send emergency message to the message queue */
            result = rt_mq_urgent(&mq, &buf, 1);
```

```
52              if (result != RT_EOK)
53              {
54                  rt_kprintf("rt_mq_urgent␣ERR\n");
55              }
56              else
57              {
58                  rt_kprintf("thread2:␣send␣urgent␣message␣-␣%c\n",
                       buf);
59              }
60          }
61          else if (cnt>= 6)/* Exit after sending 20 messages */
62          {
63              rt_kprintf("message␣queue␣stop␣send,␣thread2␣quit\n");
64              break;
65          }
66          else
67          {
68              /* Send a message to the message queue */
69              result = rt_mq_send(&mq, &buf, 1);
70              if (result != RT_EOK)
71              {
72                  rt_kprintf("rt_mq_send␣ERR\n");
73              }
74
75              rt_kprintf("thread2:␣send␣message␣-␣%c\n", buf);
76          }
77          buf++;
78          cnt++;
79          /* Delay 5ms */
80          rt_thread_mdelay(5);
81      }
82  }
83
84  /* Initialization of the message queue example */
85  int main(void)
86  {
87      rt_err_t result;
88
89      /* Initialize the message queue */
90      result = rt_mq_init(&mq,
91                          "mqt",
92                          &msg_pool[0],          /* Memory pool points
                               to msg_pool */
93                          1,                     /* The size of each
                            message is 1 byte */
94                          sizeof(msg_pool),      /* The size of the
                            memory pool is the size of msg_pool */
95                          RT_IPC_FLAG_FIFO);     /* If there are
                            multiple threads waiting, assign
                            messages in first come first get mode.
                            */
96
97      if (result != RT_EOK)
98      {
99          rt_kprintf("init␣message␣queue␣failed.\n");
100         return -1;
101     }
102
```

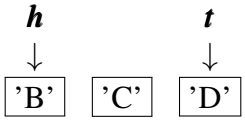
```
103     rt_thread_init(&thread1,
104                    "thread1",
105                    thread1_entry,
106                    RT_NULL,
107                    &thread1_stack[0],
108                    sizeof(thread1_stack), 25, 5);
109     rt_thread_startup(&thread1);
110
111     rt_thread_init(&thread2,
112                    "thread2",
113                    thread2_entry,
114                    RT_NULL,
115                    &thread2_stack[0],
116                    sizeof(thread2_stack), 25, 5);
117     rt_thread_startup(&thread2);
118
119     return 0;
120 }
```

Obje kreirane niti imaju isti nivo prioriteta i izvršavaju se naizmjenično, pri čemu svaka od njih ima *timeslice* od *5 OS Tick* jedinica vremena. *Thread2* šalje karaktere, a *Thread1* ih prima. Algoritam se može opisati na način prikazan u nastavku. Sa **h** je označen početak reda poruka *engl. header*, a sa **t** kraj reda poruka (engl. *tail*). Treba zapamtiti da se čitanje uvijek vrši sa *headera*.

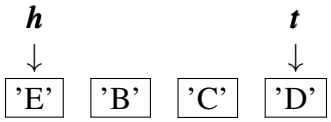
Thread2 šalje karakter `'A'` koji se smješta u red poruka.



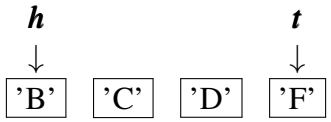
Thread1 čita poruku koja se nalazi na početku reda (ali i na kraju, s obzirom na to da se radi o samo jednom elementu) i ta poruka nestaje iz reda. *Thread2* redom šalje karaktere `'B'`, `'C'` i `'D'`, koji se pri svom dolasku smještaju na kraj, odnosno *tail* reda poruka. Poruke se **čitaju** sa *headera* po *first-in first-out* principu.



Thread2 šalje *hitnu* poruku koja sadrži karakter `'E'` i ta se poruka automatski smješta na *header* reda poruka i bit' će prva na redu za čitanje kada izvršavanje niti *Thread1* naiđe na poziv funkcije `rt_mq_recv`.



Thread1 čita karakter koji se trenutno nalazi na početku reda i izbacuje ga iz istog. Sljedeći karakter koji *Thread2* šalje je `'F'` i pošto se ne radi o urgentnom slanju, taj se element smješta na kraj reda.



Thread2 završava svoje slanje, a *Thread1* čita sve karaktere iz reda poruka. Čitanje se *uvijek* vrši sa *headera*. Ispis je prikazan u nastavku.

```
// ispis

thread2: send message - A
thread1: recv msg from msg queue, the content:A
thread2: send message - B
thread2: send message - C
thread2: send message - D
thread2: send urgent message - E
thread1: recv msg from msg queue, the content:E
thread2: send message - F
message queue stop send, thread2 quit
thread1: recv msg from msg queue, the content:B
thread1: recv msg from msg queue, the content:C
thread1: recv msg from msg queue, the content:D
thread1: recv msg from msg queue, the content:F
thread1: detach mq
```

2.5.3 Signals

Signali (engl. *signals*) u *RT-Thread* operativnom sistemu su, iz perspektive softvera - simulacija mehanizma prekida. U zvaničnoj dokumentaciji ovog operativnog sistema, spominje se još naziv *soft interrupt signal*. U principu, nit koja *prima signal* je slična procesoru koji prima zahtjev za prekidom.

POSIX standard pomoću tipa `sigset_t` definira skup signala (engl. *signal set*). Spomenuti tip može biti definiran na različite načine u različitim sistemima. U RT-Thread operativnom sistemu postoji nativni `rt_sigset_t` tip za mehanizam signala. Signali koje aplikacija može koristiti su SIGUSR1 i SIGUSR2.

Signali predstavljaju asinhroni tip komunikacije između niti. Koriste se za slanje obavještenja o abnormalnostima i hitnim slučajevima koji se mogu javiti u izvršavanju programa. Niti koje primaju signale mogu imati različite načine procesiranja za različite signale. Metode se mogu podijeliti u sljedeće tri kategorije:

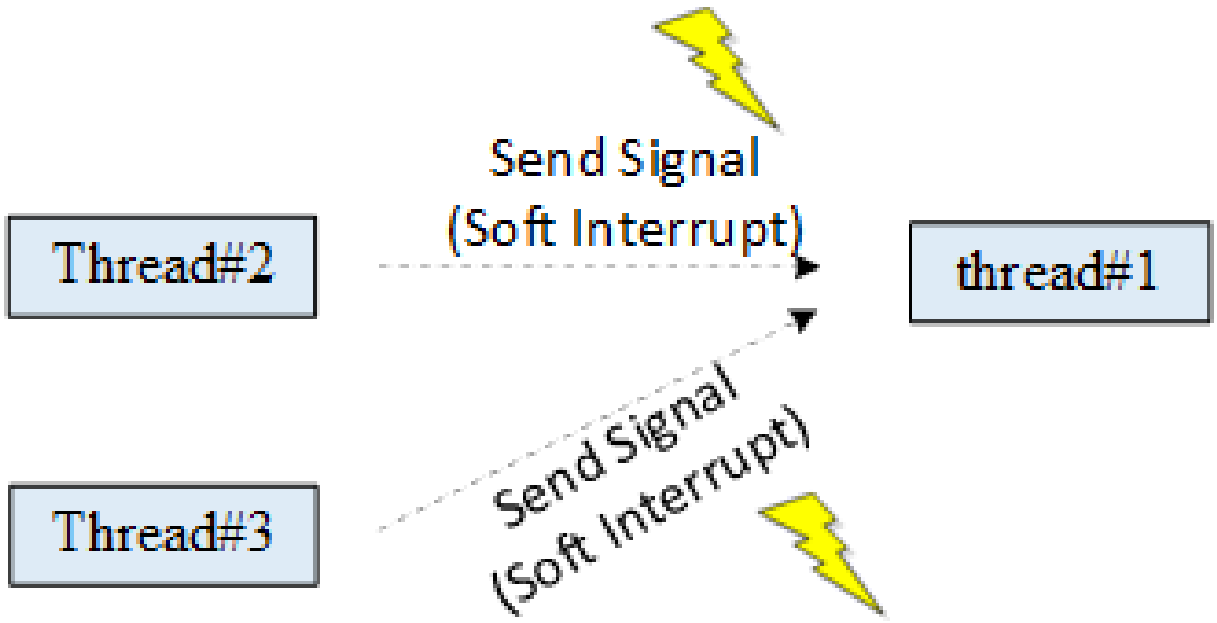
- Korištenje odgovarajuće funkcije za procesiranje signala.
- Potpuno ignoriranje signala, kao da se nije ni dogodio.
- Čuvanje *default* vrijednosti sistema za procesiranje signala.

Neka imamo situaciju kao na slici (2.10). Pretpostavimo da nit *thread#1* treba procesirati signal. Najprije je potrebno da *thread#1 instalira* signale i demaskira ih. U isto vrijeme se konfigurira način kako će se signali procesirati. Nakon toga druge niti mogu slati signale prema niti.

Ako se signal pošalje dok je *thread#1* u suspendiranom stanju, njeno stanje će se promjeniti u *ready*, da bi se procesirao signal. Ukoliko je *thread#1* u stanju izvršavanja, kreirat će se novi *stack frame space* na trenutni *stack* niti za procesiranje konkretnog signala.

Instaliranje signala

Da bi nit mogla procesirati nadolazeće signale, signali moraju biti *instalirani* unutar niti. Instalacija se primarno koristi da bi se definirala relacija između signala koji će se procesirati



Slika 2.10: Mehanizam rada signala

i akcija koje će se izvršiti u slučaju pojave odgovarajućih signala. **Instalacija signala** se realizira pomoću funkcije `rt_signal_install`, čija su definicija i opis argumenata i povratnih vrijedosti date u nastavku.

```
rt_sighandler_t rt_signal_install(int signo,
                                  rt_sighandler_t[]
                                    handler);
```

- Argumenti:
  - signo - vrijednost signala, korisniku su dostupni SIGUSR1 and SIGUSR2
  - handler - pristup procesiranju vrijednosti signala
- Moguće povratne vrijednosti:
  - `SIG_ERR` - pogrešan signal
  - The handler value before the signal is installed. - akcija uspješna

Parametar *handler* određuje metodu procesiranja ovog signala i može imati sljedeće vrijednosti:

- Pokazivač na korisnički definiranu funkciju koja će procesirati dati signal, slično kao kod korištenja prekidne rutine.
- Identifikator `SIG_IGN`, koji podrazumijeva da se signal ignoriše i ne procesira, kao da se nije ni pojavio.
- Identifikator `SIG_DFL` za pozivanje `_signal_default_handler()` za procesiranje signala.

Blokiranje i deblokiranje signala

Ukoliko je signal blokiran, on neće biti dostavljen niti koja je već instalirala signal. **Blokiranje signala** je moguće postići koristeći funkciju `rt_signal_mask`, koja prima argument vrijednosti signala *signo*.

Unutar niti je moguće instalirati nekoliko signala. Korištenjem funkcije `rt_signal_unmask` je moguće **deblokirati signal**, odnosno dati mu značaj u odnosu na ostale signale. Slanjem deblokiranog signala će dovesti do prekida za nit.

```
void rt_signal_unmask(int signo);
```

Slanje i primanje signala

Slanje signala je pogodno u slučaju abnormalnosti u radu programa. Poziv funkcije `rt_thread_kill` šalje signal prema niti čiji je zadatak da procesira hitni signal.

```
int rt_thread_kill(rt_thread_t tid, int sig);
```

- Argumenti:
  - tid - nit koja prima signal
  - sig - vrijednost signala
- Moguće povratne vrijednosti:
  - `RT_EOK` - slanje uspješno
  - `-RT_EINVAL` - greška u parametrima

Funkcija `rt_signal_wait` realizira **čekanje signala**. Ako signal već nije poslan, nit koja poziva ovu funkciju suspendira svoje izvršavanje na vremenski period specificiran argumentom *timeout*. Ukoliko je signal poslan, pokazivač na signal se smješta u argument *si*. Spomenuta funkcija ima još jedan argument, *set*, koji definira koji se signal čeka za procesiranje. Funkcija ima tri moguće povratne vrijednosti: `RT_EOK` za uspješni dolazak signala, `-RT_ETIMEOUT` u slučaju isticanja vremena predviđenog za dolazak signala i `-RT_EINVAL` u slučaju da se nalazi greška u nekom od ulaznih parametara.

Primjer korištenja signala

Aplikacija sadrži samo jednu nit, *Thread1*. Pozivom funkcije `rt_signal_install` za instaliranje signala se definira signal koji se čeka (SIGUSR1), a kao drugi argument se šalje ime funkcije koja će procesirati spomenuti signal. Signal SIGUSR1 je blokiran, te ga je potrebno demaskirati koristeći poziv funkcije `rt_signal_unmask`. Nakon što se započne izvršavanje niti, signal će se instalirati, a zatim će *main* funkcija poslati spomenuti signal. Primanje signala od strane niti će podrazumijevati procesiranje u smislu ispisa teksta. Programski kod i ispis su dati u nastavku.

Program 2.13: Signal sample

```
#include <rtthread.h>

#define THREAD_PRIORITY         25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_thread_t tid1 = RT_NULL;

/* Signal process function for thread 1 signal handler */
void thread1_signal_handler(int sig)
{
    rt_kprintf("thread1 received signal %d\n", sig);
}

/* Entry function for thread 1 */
static void thread1_entry(void *parameter)
{
    int cnt = 0;

    /* Install signal */
    rt_signal_install(SIGUSR1, thread1_signal_handler);
    rt_signal_unmask(SIGUSR1);

    /* Run for 10 times */
    while (cnt < 10)
    {
        /* Thread 1 runs with low-priority and prints the count
           value all through*/
        rt_kprintf("thread1 count : %d\n", cnt);

        cnt++;
        rt_thread_mdelay(100);
    }
}

/* Initialization of the signal example */
int main(void)
{
    /* Create thread 1 */
    tid1 = rt_thread_create("thread1",
                            thread1_entry, RT_NULL,
                            THREAD_STACK_SIZE,
                            THREAD_PRIORITY, THREAD_TIMESLICE);

    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    rt_thread_mdelay(300);

    /* Send signal SIGUSR1 to thread 1 */
    rt_thread_kill(tid1, SIGUSR1);

    return 0;
}
```

```
// ispis

thread1 count : 0
thread1 count : 1
thread1 count : 2
thread1 received signal 30
thread1 count : 3
thread1 count : 4
thread1 count : 5
thread1 count : 6
thread1 count : 7
thread1 count : 8
thread1 count : 9
```

Zaključak

Porast kompleksnosti aplikacije povećava potrebu za postojanjem više razina sistema koji tu aplikaciju izvršava. Dok je najjednostavnijim aplikacijama dovoljno da postoje slojevi hardvera i aplikacije, sofisticirane i složene aplikacije u ugradbenim sistemima je izuzetno teško realizirati bez korištenja middlewarea i operativnog sistema. U ovom radu je predstavljen *RT-Thread real-time* operativni sistem, koji se ističe visokom pouzdanošću, modularnošću i jednostavnim sučeljem. Dokumentacija data od strane razvojnog tima je napisana koncizno te sadrži dosta primjera korištenja nativnih komponenti operativnog sistema. *RT-Thread* je podržan na velikom broju CPU arhitektura, kako na 32-bitnim, tako i na 64-bitnim procesorima. Rad na jedno-jezgrenim i više-jezgrenim sistemima je također omogućen.

U ovom radu su date osnove kernela operativnog sistema RT-Thread, koji karakterizira objektno-orijentirani dizajn. Kernel se može podijeliti na dijelove koji se tiču: upravljanja nitima, upravljanja satom, sinhronizacijom i komunikacijom između niti, te upravljanje ulaznim/izlaznim uređajima i memorijom. Svi nativni objekti ovog operativnog sistema imaju prefiks `rt_` i moguće ih je deklarisati u programu kao statičke ili dinamičke. U ovom radu je predstavljeno korištenje *RT-Thread* niti, tajmera, semafora, *mutexa*, skupa događaja, *mailboxa*, reda poruka i signala. Dati su načini deklarisanja spomenutih objekata, kao i funkcije koje realiziraju odgovarajuću funkcionalnost svakog od njih. Emulacija primjera programa koji koriste navedene objekte je izvršena u *RT-Thread studio* programskom paketu.

Jasno je da je moguće napraviti ma koliko složenu aplikaciju na sistemu sa manjim brojem nivoa, ukoliko je osoba koja razvija tu aplikaciju sposobna za to. Korištenjem operativnog sistema, koji omogućava automatsko konkurentno izvršavanje zadataka, se razvoj složenijih aplikacija za ugradbene sisteme znatno pojednostavljuje.

Prilozi

Prilog A

Pokretanje RT-Thread emulacije

Emulaciju RT-Thread *real-time* operativnog sistema i testiranje programa koji sadrže komponente kao što su niti, semafori, mutexi, tajmeri itd. je moguće izvršiti pomoću programa *RT-Thread Studio*, dostupnom na sljedećem linku.

S obzirom na to da se radi o *open-source* programu, instalira se jednostavno, kao bilo koji drugi program tog tipa. Kada se *RT-Thread Studio* prvi put pokrene, za emulaciju je potrebno kreirati odgovarajući projekat, prateći sljedeće korake:

1. File → New → RT-Thread Project.
2. Otvara se prozor u kome je potrebno izvršiti početna podešavanja.
   - U *Project name* date ime projektu.
   - U *Location* definirati mjesto na kojem će projekat biti spašen.
   - Odabrati opciju *Base On Board*
   - Board: QEMU-VEXPRESS-A9
   - Adapter: QEMU
   - Ostale opcije se mogu razlikovati u zavisnosti od verzije programa koji se koristi. U tom slučaju, najbolje je odabrati ponuđene postavke.
3. Kliknuti na Finish.
4. Kada se projekat kreira, u kartici *Project Explorer* je moguće pronaći `main.c` datoteku koja predstavlja glavni program koji se pokreće zajedno sa emulacijom.
5. Izmjenom `main.c` fajla, koji se nalazi u mapi *applications*, moguće je testirati komponente RT-Thread operativnog sistema.
6. Sada je za pokretanje emulacije potrebno iz alatne trake odabrati opciju *Debug launch* (Ctrl+F5). Projekat će se kompajlirati i emulacija će započeti sa radom.
7. Emulaciju ovdje zapravo predstavlja proces debagiranja, pa je tako moguće koristiti standardne opcije za analizu izvršavanja programa, kao što su: kreiranje *breakpointa*, resume, step over, step in itd.

Prilog B

Definicije RT-Thread objekata

U ovom prilogu su date definicije nativnih objekata u RT-Thread *real-time* operativnom sistemu predstavljenih u ovom radu. Kontrolni blok (engl. *control block*) svakog objekta predstavlja `struct` tip podataka u C-programskom jeziku. Također, data je i `typedef` deklaracija pokazivača na odgovarajući objekat.

B.1 Objekat najvišeg ranga

```
struct rt_object
{
    /* Kernel object name      */
    char        name[RT_NAME_MAX];

    /* Kernel object type      */
    rt_uint8_t  type;

    /* Parameters to the kernel object    */
    rt_uint8_t  flag;

    /* Kernel object management linked list */
    rt_list_t   list;
};
```

B.2 Thread

```
/* Thread Control Block */
struct rt_thread
{
    /* rt Object */
    char          name[RT_NAME_MAX];      /* Thread Name */
    rt_uint8_t    type;                   /* Object Type */
    rt_uint8_t    flags;                  /* Flag Position */

    rt_list_t     list;                   /* Object List */
    rt_list_t     tlist;                  /* Thread List */
```

```
    /* Stack Pointer and Entry pointer */
    void        *sp;                    /* Stack Pointer */
    void        *entry;           /* Entry Function Pointer */
    void        *parameter;             /* Parameter */
    void        *stack_addr;  /* Stack Address Pointer */
    rt_uint32_t stack_size;             /* Stack Size */

    /* Error Code */
    rt_err_t    error;            /* Thread Error Code */
    rt_uint8_t  stat;                   /* Thread State */

    /* Priority */
    rt_uint8_t  current_priority;    /* Current Priority */
    rt_uint8_t  init_priority;       /* Initial Priority */
    rt_uint32_t number_mask;

    ......

    /* Thread Initialization Count Value */
    rt_ubase_t  init_tick;

    /* Thread Remaining Count Value */
    rt_ubase_t  remaining_tick;

    /* Built-in Thread Timer */
    struct rt_timer thread_timer;

    /* Thread Exit Clear Function */
    void (*cleanup)(struct rt_thread *tid);

    rt_uint32_t user_data;   /* User Data */
};
```

B.3 Timer

```
struct rt_timer
{
    struct rt_object parent;

    /* Timer Linked List Node */
    rt_list_t row[RT_TIMER_SKIP_LIST_LEVEL];

    /* Timeout Function */
    void (*timeout_func)(void *parameter);

    /* Parameters of Timeout Function  */
    void       *parameter;

    /* Timer Initial Timeout Ticks */
    rt_tick_t init_tick;
```

```
    /* Number of ticks when the timer actually times out */
    rt_tick_t timeout_tick;
};
typedef struct rt_timer *rt_timer_t;
```

B.4 Semaphore

```
struct rt_semaphore
{
   /* Inherited from the ipc_object class */
   struct rt_ipc_object parent;

   /* Semaphore Value */
   rt_uint16_t value;
};
/* rt_sem_t is the type of pointer pointing to semaphore
   structure */
typedef struct rt_semaphore* rt_sem_t;
```

B.5 Mutex

```
struct rt_mutex
    {
        /* inherited from the ipc_object class */
        struct rt_ipc_object parent;

        /* mutex value */
        rt_uint16_t          value;

        /* hold the original priority of the thread */
        rt_uint8_t           original_priority;

        /* number of times holding the threads */
        rt_uint8_t           hold;

        /* thread that currently owns the mutex */
        struct rt_thread     *owner;
    };
    /* rt_mutext_t pointer type of the one poniter pointing to
        the mutex structure */
    typedef struct rt_mutex* rt_mutex_t;
```

B.6 Event set

```
struct rt_event
{
    struct rt_ipc_object parent;    /* Inherited from the
       ipc_object class */

    /* The set of events, each bit represents 1 event, the
       value of the bit can mark whether an event occurs */
    rt_uint32_t set;
};
/* rt_event_t is the pointer type poniting to the event
   structure */
typedef struct rt_event* rt_event_t;
```

B.7 Mailbox

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    /* the start address of the mailbox buffer */
    rt_uint32_t* msg_pool;

    /* the size of the mailbox buffer      */
    rt_uint16_t size;

     /* the number of messages in the mailbox      */
    rt_uint16_t entry;

    /* the entry and exit pointer of the mailbox buffer    */
    rt_uint16_t in_offset, out_offset;

     /* send the suspend and wait queue of the thread */
    rt_list_t suspend_sender_thread;
};
typedef struct rt_mailbox* rt_mailbox_t;
```

B.8 Message queue

```
struct rt_messagequeue
{
    struct rt_ipc_object parent;

    /* Pointer pointing to the buffer storing the messages */
    void* msg_pool;

    rt_uint16_t msg_size;    /* The length of each message */

    /* Maximum number of messages that can be stored */
```

```
    rt_uint16_t max_msgs;

    /* Number of messages already in the queue */
    rt_uint16_t entry;

    void* msg_queue_head;     /* Message linked list header  */
    void* msg_queue_tail;     /* Message linked list tail  */
    void* msg_queue_free;     /* Idle message linked list */
};
typedef struct rt_messagequeue* rt_mq_t;
```

Prilog C

RT-Thread shell komande

Unošenjem ključne riječi *help* u shell RT-Thread operativnog sistema, izlistava se sljedeća lista komandi.

```
RT-Thread Shell commands:

memcheck          - check memory data
memtrace          - dump memory trace information
memheaptrace      - dump memory trace information
memtrace_heap     - dump memory trace for heap
gic_dump          - show gic status
list_fd           - list file descriptor
clear             - clear the terminal screen
version           - show RT-Thread version information
list_thread       - list thread
list_sem          - list semaphore in system
list_event        - list event in system
list_mutex        - list mutex in system
list_mailbox      - list mail box in system
list_msgqueue     - list message queue in system
list_memheap      - list memory heap in system
list_mempool      - list memory pool in system
list_timer        - list timer in system
list_device       - list device in system
ls                - List information about the FILEs.
cp                - Copy SOURCE to DEST.
mv                - Rename SOURCE to DEST.
cat               - Concatenate FILE(s)
rm                - Remove(unlink) the FILE(s).
cd                - Change the shell working directory.
pwd               - Print the name of the current working directory.
mkdir             - Create the DIRECTORY.
mkfs              - format disk with file system
mount             - mount <device> <mountpoint> <fstype>
umount            - Unmount device from file system
df                - disk free
echo              - echo string to file
tail              - print the last N-lines data of the given file
exit              - return to RT-Thread shell mode.
help              - RT-Thread shell help.
ps                - List threads in the system.
free              - Show the memory usage in the system.
ifconfig          - list the information of all network interfaces
ping              - ping network host
dns               - list and set the information of dns
netstat           - list the information of TCP / IP
mtd_nand          - MTD nand device test function
date              - get date and time or set(local timezone)[y m d h min sec]
sf                - SPI Flash operate.
```

Prilog D

RT-Thread podržani hardver

Čipovi i razvojni sistemi koji podržavaju RT-Thread kao operativni sistem su dati u nastavku.

- at91sam9260
- at91sam9g45
- avr32uc3b0
- amebaz
- allwinner_tina
- asm9260t
- beaglebone
- bf533
- ck802
- dm365
- efm32
- es32f0654
- essemi
- frdm-k64f
- fh8620
- gd32303e-eval
- gd32450z-eval
- gd32e230k-start
- gd32vf103v-eval
- gkipc
- hifive1
- i.mx6sx
- i.mx6ul
- i.MX6 SoloX
- imxrt1064-nxp-evk
- imxrt1052-nxp-evk
- lm3s8962
- lm3s9b9x
- lm4f232
- lpc1114
- lpc176x
- lpc178x
- lpc2148
- lpc2478
- lpc408x
- lpc43xx
- lpc5410x
- lpc54114-lite
- lpc54608-LPCXpresso
- lpc55sxx
- lpc824
- ls1bdev
- ls1cdev
- k210
- microblaze
- m16c62p
- mb9bf500r
- mb9bf506r
- mb9bf568r
- mb9bf618s
- mini2440
- mini4020
- mm32l07x
- mm32l3xx
- nios_ii
- nrf51822
- nrf52832
- nuvoton_m05x
- nuvoton_m451
- nuvoton_m487
- nuvoton_nuc472
- nv32f100x
- psoc6-pioneerkit_modus
- pic32ethernet
- raspi2
- Raspberry Pi PICO
- Raspberry Pi3
- Raspberry Pi4
- rv32m1_vega
- rx
- sam7x
- samd21
- swm320-lq100
- synopsys
- stm32f072-st-nucleo
- stm32f091-st-nucleo
- stm32f401-st-nucleo
- stm32f407-st-discovery
- stm32f411-st-nucleo
- stm32f412-st-nucleo
- stm32f429-st-disco
- stm32f446-st-nucleo
- stm32f469-st-disco
- stm32f746-st-disco
- stm32f767-st-nucleo
- stm32f769-st-disco
- stm32g071-st-nucleo
- stm32g431-st-nucleo
- stm32h743-st-nucleo
- stm32l053-st-nucleo
- stm32l432-st-nucleo
- stm32l452-st-nucleo
- stm32l475-st-discovery
- stm32l476-st-nucleo
- stm32l4r5-st-nucleo
- stm32l4r9-st-eval
- stm32
- stm32f20x
- stm32f103-atk-nano
- stm32f103-atk-warshipv3
- stm32f103-dofly-M3S
- stm32f103-dofly-lyc8
- stm32f103-fire-arbitrary
- stm32f103-gizwits-gokitv21
- stm32f103-hw100k-ibox
- stm32f103-mini-system
- stm32f103-yf-ufun
- stm32f107-uc-eval
- stm32f405-smdz-breadfruit
- stm32f407-atk-explorer
- stm32f427-robomaster-a
- stm32f429-armfly-v6
- stm32f429-atk-apollo
- stm32f429-fire-challenger
- stm32f767-atk-apollo
- stm32f767-fire-challenger
- stm32f767-atk-apollo
- stm32h750-armfly-h7-tool
- stm32l475-atk-pandora
- stm32l496-ali-developer
- stm32f103-blue-pill
- stm32f410-st-nucleo
- stm32f411-weact-MiniF4
- stm32f413-st-nucleo
- stm32g070-st-nucleo
- stm32h743-st-nucleo
- stm32h747-st-discovery
- stm32l010-st-nucleo
- stm32l412-st-nucleo
- stm32l433-st-nucleo
- stm32l496-st-nucleo
- stm32mp157a-st-discovery
- stm32mp157a-st-ev1
- stm32wb55-st-nucleo
- stm32wl55-st-nucleo
- tm4c129x
- tms320f28379d
- w60x
- x86
- xplorer4330
- x1000
- zynq7000

Literatura

[1] Ungurean, I., “Timing comparison of the real-time operating systems for small microcontrollers”, Symmetry, Vol. 12, No. 4, 2020, str. 592.

[2] RT-Thread document center, RT-Thread IoT OS, 2021, dostupno na: https://www.rt-thread.io/document/site/