Univerzitet u Banjoj Luci
Elektrotehnički fakultet

SEMINARSKI RAD

PREDMET: MIKROPROCESORSKI SISTEMI 2

TEMA: Demonstrirati rad komunikacije između taskova korištenjem reda čekanja poruka (message queue), opisati API.

Mentor:
Velibor Škobić

Studenti:
Dejan Đekanović
Ljubomir Grublješić
Marinko Grublješić

Banja Luka, septembar 2017. godine

SADRŽAJ

1. Uvod

Operativni sistem za rad u realnom vremenu (Real-Time Operating System, RTOS) je operativni sistem namijenjen da služi u ugrađenim sistemima koji obrađuju podatke u trenutku kada se podaci pojave.

Osnovna karakteristika ovih operativnih sistema je nivo konzistencije koji se tiče vremena koje je potrebno da se prihvati i završi zadatak aplikacije. Glavni cilj dizajna ovih sistema nije visok protok, nego garancija da će se zadaci izvršavati u trenucima u kojima trebaju biti izvršeni.

FreeRTOS je jedna od vrsta operativnih sistema za rad u realnom vremenu. Veoma je široko rasprostranjen i besplatan je za korištenje. Portovan je na 35 mikrokontrolera.

FreeRTOS-a je biblioteka sa određenim skupom funkcionalnosti, među kojima je i mogućnost komunikacije između procesa upotrebom reda čekanja. Na sajtu FreeRTOS-a nalazi se kompletan API za korištenje kao i njihove biblioetke.

2. Zadaci (tasks)

Cilj zadatka je da se napiše API za komunikaciju između zadataka tako što se napravi sistem zasnovan na principu reda čekanja, tj. enlgleski message queue. Ujedno da se na kratkom primjeru demonstrira rad sa datim API-em.

Sama komunikacija zasnovan je na struturi ”AMessage” koja ima sledeći izgled:

```
struct AMessage {
    char taskID;
    char value;
};
```

Promjenljiva “taskID” služi da identifikuje zadatak kome je sadržaj poruke namijenjen, tako da svaki zadatak ima svoj ID. Promjenljiva “value” treba da prenese odgovarajuću sadržaj poruke.

Zadaci se kreiraju pozivom funkcije xTaskCreate, koja kao argument prima ime funkcije koju će zadatak izvršavati, opisno ime zadatka, veličina steka koja će se koristiti za taj zadatak, parametar/parametri koji će biti proslijeđen funkciji, prioritet kojim će se zadatak izvršavati i *handle* zadatka:

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char * const pcName,
                       unsigned short usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *pxCreatedTask);
```

Funkcija koju će zadatak izvršavati je funkcija čiji je povratni tip void, a argument tipa void*, drugim riječima:

```
void taskFunction(void *parameters);
```

Kada se naprave svi zadaci, pokrene se raspoređivač zadataka. Poziv funkcije koja pokreće raspoređivač zadataka je blokirajući, što znači da kod, koji se nađe poslije poziva te funkcije, neće biti izvršen. Raspoređivač zadataka se pokreće pozivom funkcije vTaskStartScheduler, koja ima sledeći potpis:

```
void vTaskStartScheduler(void);
```

Za demonstraciju smo napravili dvije promjenljive koje su povezane na periferije ploče. One su povezane na prekidače i LED diode.

Sada cemo objasniti kod koji smo napisali u primjeru.

```
volatile char* SWITCHes = (char*) XPAR_DIP_SWITCHES_BASEADDR;

volatile char* LEDs = (char*) XPAR_LEDS_BASEADDR;
```

Pokazivačima SWITHCes i LEDs dodijeljena je bazna adresa prekidača i LED dioda respektivno, tako da je moguće čitati, odnosno upisivati odgovarajuće vrijednosti prekidača, odnosno LED dioda. Sledeće tri linije koda deklarišu funkcije koje ce zadaci izvršavati.

```
static void task1Method(void *pvParameters);
static void task2Method(void *pvParameters);
static void task3Method(void *pvParameters);
```

Glavna (main) funkcija kreira tri zadatka, pokrene rasporedjivač zadataka i na kraju ode u beskonačnu for petlju. Iako je poziv vTaskStartScheduler() blokirajući i nikada ne bi trebalo da završi, ovako osiguravamo da, ako se ipak u slučaju neke greške desi da taj poziv odblokira, program ne završi. Dalje su definisane funkcije koje će zadaci izvršavati

```
xTaskCreate(task1Method, "task1", configMINIMAL_STACK_SIZE, NULL,
tskIDLE_PRIORITY, NULL);

xTaskCreate(task2Method, "task2", configMINIMAL_STACK_SIZE, NULL,
tskIDLE_PRIORITY, NULL);

xTaskCreate(task3Method, "task3", configMINIMAL_STACK_SIZE, NULL,
tskIDLE_PRIORITY, NULL);
```

Svaki zadatak ima svoju metodu, s tim da svi zajedno koriste jedan *queue*. Prvi zadatak detektuje promjenu na prekidaču i šalje (doda u *queue*) poruku drugom zadatku.
Njegova funkcija izgleda ovako:

```
static void task1Method(void *pvParameters) {
	struct AMessage xSendMessage;
	struct AMessage *pxSendMessage = &xSendMessage;
	char oldValue = *SWITCHes;
	while (1) {
		/* If switches changed */
		if (oldValue != *SWITCHes) {
			oldValue = *SWITCHes;
			/* Set message for second task */
			pxSendMessage->taskID = 2;
			pxSendMessage->value = oldValue;
			/* Send message to second task */
			xQueueSend(xQueue, (void* )&pxSendMessage, (TickType_t ) 0);
		}
	}
}
```

Drugi zadatak provjerava da li *queue* ima poruka za njega. Ukoliko prouka za njega postoji on prima tu poruku, zamijeni raspored bita sadržaja poruke (npr. 1001110 -> 0111001), a zatim pošalje tako izmijenjen sadržaj poruke trećem zadatku.

Njegova funkcija izgledao ovako:

```
static void task2Method(void *pvParameters) {
	struct AMessage xSendMessage;
	struct AMessage *pxSendMessage = &xSendMessage;
	struct AMessage *pxReceiveMessage;
	while (1) {
		/* Get message from queue */
		if (xQueuePeek(xQueue, &(pxReceiveMessage), (TickType_t)0)) {
			/* Check if message sent to second task */
			if (pxReceiveMessage->taskID == 2) {
				/* Set message for third task */
				pxSendMessage->taskID = 3;
				pxSendMessage->value = reverseBits(pxReceiveMessage->value);
				/* Remove message from queue */
				xQueueReceive(xQueue, NULL, (TickType_t ) 0);
				/* Send message to third task */
				xQueueSend(xQueue, (void* )&pxSendMessage, (TickType_t ) 0);
			}
		}
	}
}
```

Treći zadazak također provjerava da li postoje poruka za njega, i ukoliko postoje primi odgovarajuću poruku i ukloni je iz reda čekanja. Nakon što primi poruku postavi vrijednost LED dioda na vrijednost sadržaja poruke (value) koju je primio od drugog zadatka.

Njegova funkcija izgledao ovako:

```
static void task3Method(void *pvParameters) {
	struct AMessage *pxReceiveMessage;
	while (1) {
		/* Get message from queue */
		if (xQueuePeek(xQueue, &(pxReceiveMessage), (TickType_t ) 10)) {
			/* Check if message sent to third task */
			if (pxReceiveMessage->taskID == 3) {
				/* Set LEDs */
				*LEDs = pxReceiveMessage->value;
				/* Remove message from queue */
				xQueueReceive(xQueue, NULL, (TickType_t ) 10);
			}
		}
	}
}
```

3. Red čekanja (message queue)

Ukoliko se pojavi potreba da zadaci međusobno komuniciraju, odnosno da šalju poruke jedni drugima, poželjno je koristi red čekanja. Obično različiti zadaci obavljaju različite poslove, pa ukoliko rezultat koji je izračunao neki zadatak bude potreban nekom drugom zadatku, moguće ga je poslati upotrebom reda čekanja.

Red čekanja predstvlja strukturu podatak u kojoj se elementi dodaju na početak reda, a preuzimaju sa kraja reda, tzv. FIFO (*FirstInFirstOut*) struktura podataka.

Ako bi se za razmjenu podataka koristile globalne promjenljive moglo bi doći do situacije da različiti zadaci u isto vrijeme pristupaju određenom podatku, tj. može doći do greške u sistemu. Razmjena (dijeljenje) podataka može se ostvariti i uoptrebom semafora ali u tom slučaju dijeljeni resurs koji je "zaključan" od strane jednog zadatka nije dostupan drugom zadatku sve dok ga prvi zadazak ne "otključa", tako da će drugi zadatak čekati odgovarajuće vrijeme.

U nastavku su opisane korištene funkcija *API*-a za rad sa redom čekanja, ujedno su to i najčešće korištene funkcije *Queues API*-a.

Red čekanja se kreira pozivom funkcije xQueueCreate, koja ima sledeći potpis:

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength,
                           UBaseType_t uxItemSize );
```

Parametar uxQueueLength predstavlja veličinu, odnosno maksimalni broj elemenata reda, dok uxItemSize predstavlja veličinu jednog elementa reda. Povratna vrijednost predstavlja *handle* objekat za dati red.

Da bi se odgovarajući podaci poslali u red čekanja koristi se funkcija xQueueSend koja ima sledeći potpis:

```
BaseType_t xQueueSend(QueueHandle_t xQueue,
                      const void * pvItemToQueue,
                      TickType_t xTicksToWait);
```

Parametar xQueue predstavlja *handle* objekat za red čekanja u koji želimo da smjestimo odgovarajuću poruku. Drugi parametar pvItemToQueue predstavlja pokazivač na poruku koju želimo da dodamo u red čekanja, dok parametar xTicksToWait predstavlja maksimalno vrijeme koje će zadatak čekati ako je red pun. Funkcija vraća informaciju o tome da li je poruka uspješno dodana u red čekanja.

Da bi se poruke preuzele iz reda čekanja koriste se funkcije xQueuePeek i xQueueReceive. Razlika između ove dvije funkije je u tome što xQueuePeek prima poruku, ali je za razliku od xQueueReceive ne uklanja iz reda čekanja.

Funkcije xQueuePeek i xQueueReceive imaju sledeće potpise:

```
BaseType_t xQueuePeek(QueueHandle_t xQueue,
                      void *pvBuffer,
                      TickType_t xTicksToWait);
```

```
BaseType_t xQueueReceive(QueueHandle_t xQueue,
                         void *pvBuffer,
                         TickType_t xTicksToWait);
```

Parametar xQueue predstavlja *handle* objekat za red čekanja iz kojeg želimo da dobijemo odgovarajuću poruku. Drugi parametar pvBuffer predstavlja pokazivač na objekat u koji želimo da smjestimo primljenu poruku, dok parametar xTicksToWait predstavlja maksimalno vrijeme koje će zadatak čekati ako je red pun. Funkcija vraća informaciju o tome da li je poruka uspješno preuzeta iz reda čekanja.

Nakon završetka korištenja reda čekanja potrebno ga je ukloniti, odnosno osloboditi prostor koji je alociran za smještanje elemenata reda. Funkcija xQueueDelete ima zadatak da obriše odgovarajući red čekanja.

Data funkcija ima sledeći potpis:

```
void vQueueDelete(QueueHandle_t xQueue);
```

Parametar xQueue predstavlja *handle* objekat za red čekanja koji želimo da obrišemo.