Dokumentacija SENSIRRIKA firmware-a

Fran Penić

28. 2. 2021

1 Scheduler

Kako bi se olakšalo pisanje programske podrške za SENSIRRIKA ugradbeni sustav razvijen je jednostavni scheduler. Scheduler omogućuje dodavanje taskova na listu i njihovo izvršavanje redosljedom određenim njihovim prioritetima. Također je moguće blokiranje izvršavanja taskova dok se ne zadovolje zadani uvjeti.

Za razliku od schedulera korištenog u FreeRTOS-u, taskovi se nakon izvršavanja brišu sa liste i ne ponavlja se njihovo izvršavanje.

Uz scheduler, dodana je i funkcionalnost za inicijalizaciju, čitanje i pisanje cirkularnih buffera. Bufferi također imaju mogućnost dodavanja taskova za izvršavanje kada dosegnu određenu razinu popunjenosti.

1.1 Sučelje scheduler-a

U ovom poglavlju su opisane strukture i funkcije koje se koriste pri pisanju firmware-a koji koristi scheduler. Sve strukture i funkcije definirane su u datotekama scheduler.h i scheduler.c .

1.1.1 Scheduler

```
typedef struct task task_s;
struct task {
    void (*function)(task_s *);
    void *args;
    task_s *next;
    volatile uint32_t *flag;
    uint32_t mask;
    uint8_t priority;
};
```

Struktura `task_s` sadrži sve informacije o tasku.
`function` je pointer na funkciju koja se izvršava kad task dođe na red. Funkcije koje poziva scheduler kao argument primaju pointer na task kojem pripadaju.
`args` može sadržavati proizvoljne argumente za funkciju.
`next` je pointer na sljedeći task u listi. Ovu varijablu koristi scheduler i pri inicijalizaciji ju je potrebno postaviti na `NULL`.
Varijable `flag` i `mask` služe za blokiranje taskova. Task će biti blokiran sve dok u varijabli na koju pokazuje `flag` nisu postavljeni svi bitovi definirani u varijabli `mask`. `(*flag & mask != mask)`
`priority` određuje prioritet izvršavanja taska. Veća vrijednost varijable označava veći prioritet. U slučaju da dva taska imaju isti prioritet prvi će biti izvršen onaj koji je prvi dodan na listu.

Nakon što se definira `task_s` struktura za pojedini task, moguće je taj task dodati na listu za izvođenje korištenjem funkcije `queue_task(task_s task)`.

Scheduler se pokreće zvanjem funkcije `run_scheduler()`. Ova funkcija sadrži beskonačnu petlju u kojoj izvršava taskove, a u slučaju da nema taskova koji su spremni za izvršavanje stavlja mikrokontroler u način rada niske potrošnje.

1.1.2 Bufferi

```
typedef struct buffer buffer_s;
struct buffer {
    uint32_t n_elem;
    uint32_t size_elem;
    uint8_t *buff;
    uint8_t *rd_ptr;
    uint8_t *wr_ptr;
    uint32_t n_curr;
    task_cond_t add_task_cond;
    task_s task_to_add;
};
```

Struktura `buffer_s` sadrži sve informacije o cirkularnom bufferu.
`n_elem` označava maksimalni broj elemenata u bufferu.
`size_elem` označava veličinu svakog elementa u bajtovima.
`buff`, `rd_ptr`, `wr_ptr` su interne varijable koje nije potrebno inicijalizirati. `buff` pokazuje na početak memorije dodijeljene bufferu. `rd_ptr` pokazuje na element nakon zadnjeg pročitanog, a `wr_ptr` na element nakon zadnjeg upisanog.
`n_curr` sadrži broj elemenata sadržanih u bufferu.
`add_task_cond` sadrži jedan ili više uvjeta koji moraju biti ispunjeni kako bi task na koji pokazuje `task_to_add` bio dodan na listu za izvođenje. Mogući uvjeti su definirani u sljedećem enum-u :

```
typedef enum {
    COND_NEVER = 0,
    COND_ALWAYS = 1<<0,
    COND_FULL = 1<<1,
    COND_HALF_FULL = 1<<2,
    COND_EMPTY = 1<<3,
    FLAG_COND = 1<<30,
    COND_NUMBER = 1<<31
}task_cond_t;
```

`COND_NEVER` znači da task nikada neće biti dodan na listu.
`COND_ALWAYS` znači da će task biti dodan za svaki element upisan u buffer.
`COND_FULL` znači da će task biti dodan kada je buffer pun.
`COND_HALF_FULL` znači da će task biti dodan kada buffer dosegne pola kapaciteta (zaokruženo na sljedeći cijeli broj).
`FLAG_COND` je zastavica koja se koristi interno kako bi se izbjeglo višestruko dodavanje taskova pri korištenju `COND_HALF_FULL` ili `COND_NUMBER`.
Kada je postavljen `COND_NUMBER`, task će biti dodan kada buffer sadrži broj elemenata jednak prvih 30 LSB-a varijable `add_task_cond`. Uvjet se tada zadaje na sljedeći način : `COND_NUMBER|n_elem`.

Inicijalizacija buffera se obavlja sljedećom funkcijom:

```
buffer_status_t init_buffer(buffer_s *buffer,
                    uint32_t n_elem, uint32_t size_elem,
                    task_s *task_to_add,
                    task_cond_t add_task_cond);
```

Parametri funkcije odgovaraju istoimenim parametrima `buffer_s` strukture.

Čitanje iz buffera se obavlja sljedećom funkcijom:

```
uint32_t read_from_buffer(buffer_s *buffer,
            void *data,
            uint32_t elem_to_read);
```

Parametar `buffer` je pointer na buffer iz kojega se čita.
`data` je **pointer na pointer** koji nakon izvršavanja pokazuje na prvi pročitani element.
`elem_to_read` određuje broj elemenata koji želimo pročitati. U slučaju da je `elem_to_read` veći od broja trenutno dostupnih elemenata, funkcija će pročitati samo dostupne elemente. U slučaju kada je dio dostupnih elemenata na kraju, a dio na početku buffera (wrap-around), funkcija će pročitati samo one elemente koji se nalaze na kraju buffera. Kada je `elem_to_read` jednak 0, biti će pročitani svi dostupni elementi.

Pisanje u buffer se obavlja sljedećom funkcijom:

```
buffer_status_t write_to_buffer(buffer_s *buffer,
                    void *data_ptr,
                    uint8_t rev);
```

Parametar `buffer` je pointer na buffer u koji se upisuju podaci.
`data_ptr` je pokazivač na početak elementa koji se upisuje u buffer. U buffer se uvijek upisuje jedan po jedan element. U slučaju kada je `data_ptr = NULL`, funkcija će promijeniti `n_elem` i `wr_ptr` bez upisivanja podataka. Ova funkcionalnost se može koristiti ako su podaci direktno upisani u buffer bez korištenja funkcije.
Parametar `rev` omogućuje promjenu redosljeda bajtova podataka koji se upisuju u buffer. Trenutno je podržana jedino promjena redosljeda bajtova za 16-bitne podatke (Little endian u Big endian i obratno).

1.2 Primjer korištenja schedulera

U ovom poglavlju je prikazan jednostavan primjer korištenjem schedulera.

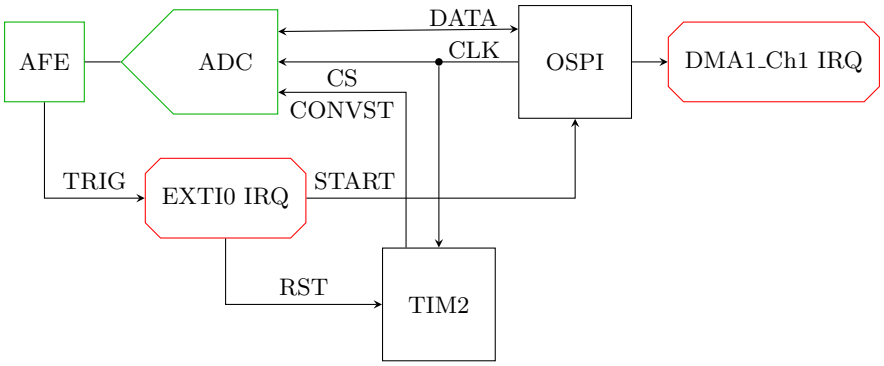
```
#include "scheduler.h"

void ex_function(task_s *task);

task_s ex_task;

uint32_t ex_flag;

buffer_s ex_buffer;


int main(void){
    uint32_t a = 0;

    //Inicijalizacije task_s strukture
    ex_task = (task_s){
        .function = &ex_function,
        .args = (void *)&ex_buffer,
        .next = NULL,
        .flag = NULL,
        .mask = 0,
        .priority = 5
    }

    //Inicijalizacija buffera
    init_buffer(&ex_buffer, 2, 4, &ex_task, COND_ALWAYS);

    //U buffer se upisuje vrijednost 0
    //Kako je add_task_cond = COND_ALWAYS,
    //ex_task se dodaje na listu
    write_to_buffer(&ex_buffer, &a, 0);

    //Pokretanje schedulera
    run_scheduler();
}

void ex_function(task_s *task){
    uint32_t *i;

    //citanje vrijednosti iz ex_buffer buffera
    read_from_buffer((buffer_s *)task->args, &i, 1);

    //ispis procitane vrijednosti
    //Napomena: ovo ne funkcionira na pravom ugradbenom
        sustavu
    printf("%u\n", *i);

    //Vrijednost se uvecava za jedan
    *i += 1;

    //Uvecana vrijednost se upisuje u isti buffer
    //Time se ex_task ponovno stavlja na listu
    write_to_buffer((buffer_s *)task->args, i, 0);
}
```

2 Akvizicija i obrada

U ovom poglavlju objašnjeni su taskovi, bufferi i prekidne rutine koje čine firmware SENSIRRIKA ugradbenog sustava.

2.1 Akvizicija

Ugradbeni sustav za komunikaciju sa ADC-om koristi QuadSPI protokol, koji podržava OctoSPI periferna jedinica mikrokontrolera. Međutim, zbog nedostataka te periferne jedinice, za komunikaciju se također koristi timer TIM2.
Blok shema sustava za akviziciju prikazana je na slici 1.



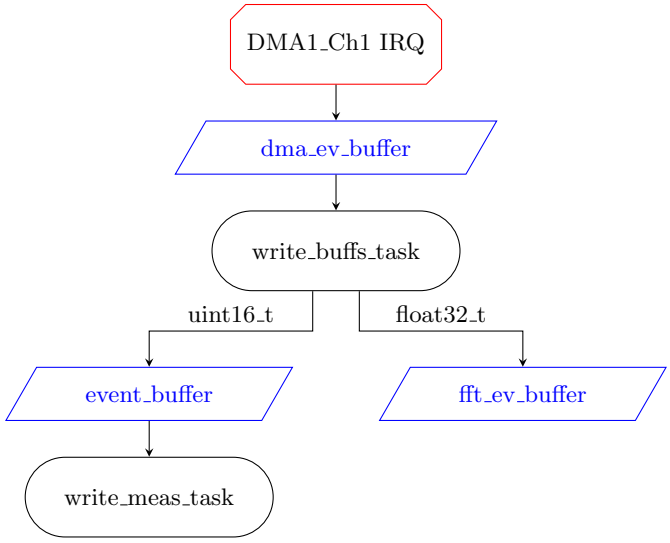
Slika 1: Blok shema akvizicijskog sustava

Proces akvizicije počinje kada mikrokontroler sa analognog front-enda (AFE) dobije TRIGGER signal, što aktivira EXTI0 prekid. U slučaju da akvizicija nije u tijeku, prekid resetira TIM2 i pokreće OSPI jedinicu za komunikaciju. U slučaju da je akvizicija već u tijeku, samo se pamti redni broj uzorka koji se trenutno očitava radi kasnijeg spremanja.

Korišteni ADC počinje slati podatke na prvi brid CLK impulsa nakon promjene CS signala u 0, ali OSPI sklop zahtijeva barem dva CLK impulsa prije početka primanja podataka. Iz tog razloga se CS signalom upravlja TIM2 timerom, a OSPI je konfiguriran na način da očitava 32-bitne brojeve (4 "prazna" ciklusa pa 4 ciklusa podataka) od kojih DMA sprema samo zadnjih 16 bita. Kako izlazi timera imaju znatno kašnjenje, OSPI započinje očitavanje svakog eventa sa 5 "dummy" ciklusa koji služe da se to kašnjenje kompenzira.

Nakon što je očitana polovina uzoraka (njih 512) javlja se DMA1_Ch1 prekid, u kojemu se prvih 256 uzoraka sprema u `dma_ev_buffer`. Isti prekid se javlja i nakon što su očitani svi uzorci, kada se resetira OSPI jedinica i, u slučaju da je došlo do ponovne aktivacije triggera za vrijeme trajanja akvizicije, u `dma_ev_buffer` sprema dodatnih 256 uzoraka. Osim samih uzoraka, u buffer se sprema i timestamp, koji označava vrijeme od paljenja sustava u stotinkama sekunde.

Svako upisivanje podataka u `dma_ev_buffer` na listu stavlja write_buffs_task. Ovaj task upisuje podatke iz `dma_ev_buffer` u `event_buffer`, koji služi za pisanje snimaka na SD karticu i `fft_ev_buffer`, koji služi za daljnju obradu. Podaci se prije upisivanja u `fft_ev_buffer` pretvaraju u float32_t format, koji se koristi u svim sljedećim koracima obrade.

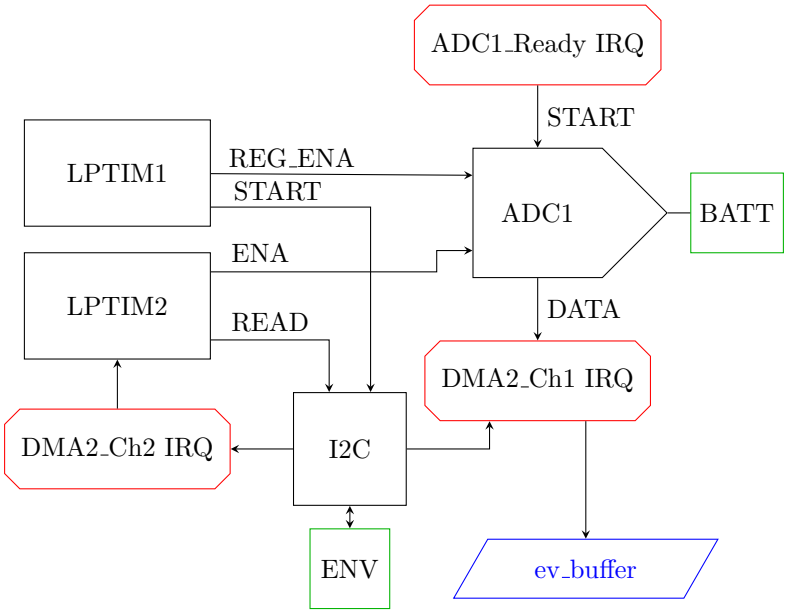


Slika 2: Spremanje uzoraka u buffere za spremanje i obradu

Sustav također ima mogućnost mjerenja temperature, vlage i napona baterije. Napon baterije se mjeri pomoću ADC-a ugrađenog u mikrokontroler, a temperatura i vlaga pomoću Sensirion SHT31 senzora s kojim mikrokontroler komunicira pomoću I2C-a. Kako bi se uštedila energija, ADC se nakon svakog mjerenja postavlja u "deep sleep" način rada, a kako bi se uštedjelo procesorsko vrijeme za I2C komunikaciju se koristi DMA.

Mjerenja se pokreću svakih 20 minuta, kada LPTIM1 timer pokrene paljenje regulatora ADC-a i slanje komande senzoru. Nakon završetka slanja komande, javlja se DMA2_CH2 prekid koji pokreće LPTIM2 timer, koji služi da osigura vrijeme potrebno za paljenje regulatora ADC-a i mjerenje temperature i tlaka. Nakon što LPTIM2 završi sa odbrojavanjem, pali se ADC i pokreće čitanje vrijednosti sa senzora. Mjerenje napona baterije pokreće ADC1_Ready prekid, koji se javlja kada se ADC upali.

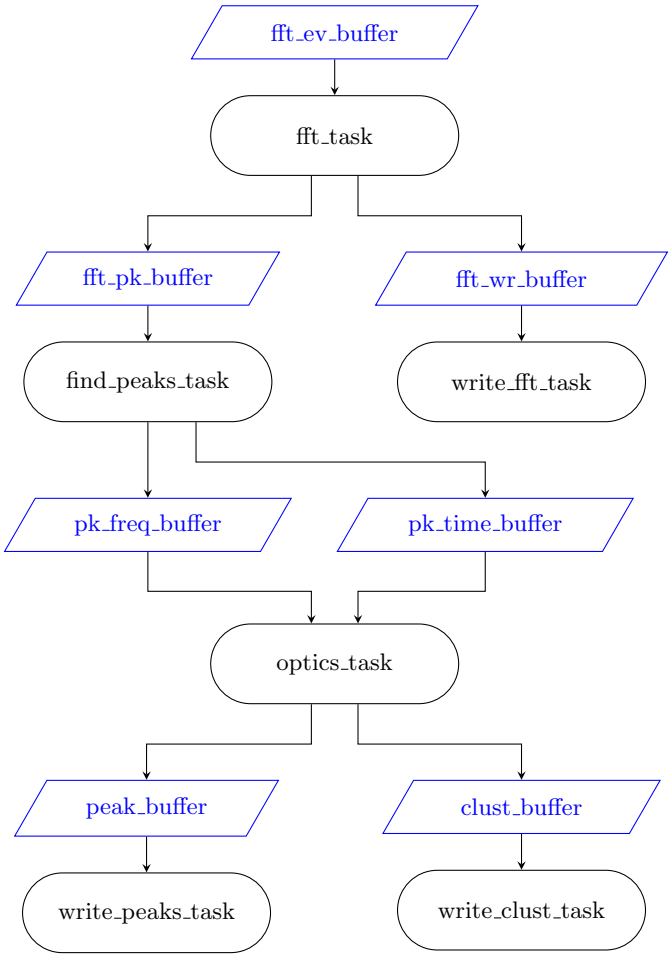
Prekid DMA2_Ch1 se javlja kada I2C završi čitanje vrijednosti sa senzora. U tom prekidu se vrijednosti tlaka, temperature i napona baterije zajedno sa timestamp-om spremaju u `ev_buffer`.



Slika 3: Očitavanje temperature, tlaka i napona baterije

2.2 Obrada

Obrada podataka se odvija u tri koraka, a svaki korak je implementiran u svom tasku. Dijagram toka obrade je prikazan na slici 4.



Slika 4: Obrada i spremanje clustera

Prvi korak obrade je fft_task koji računa magnitudni spektar snimljenih evenata i na njemu radi korekciju frekvencijske karakteristike piezo pretvornika i pojačala. Izračunati spektar se upisuje u `fft_wr_buffer` za pisanje na SD karticu i `fft_pk_buffer` za daljnju obradu.

Sljedeći korak obrade je find_peaks_task koji pronalazi peak-ove u spektru. Algoritam za pronalazak peakova se može konfigurirati pomoću tri parametra: MIN_PROM, MIN_WIDTH i WIDTH_MUL. Parametrom MIN_PROM se zadaje minimalna razlika između amplitude peak-a i lokalnih minimuma sa obe strane tog maksimuma. Parametrom MIN_WIDTH se zadaje minimalni broj točaka oko peak-a čije amplitude moraju biti veće od amplitude peak-a pomnožene sa parametrom WIDTH_MUL.

Zadnji korak obrade je optics_task koji radi clustering na dobivenim peakovima. Rezultat izvođenja optics_task-a su koordinate trapeza kojima su omeđeni peak-ovi koji spadaju u svaki pojedini cluster. Dobivene koordinate se spremaju na SD karticu zajedno sa oznakom clustera i brojem točaka u clusteru. Osim dobivenih koordinata, na SD karticu se spremaju i koordinate svakog pojedinog peak-a uz oznaku clustera kojemu taj peak pripada.

Funkcija koja implementira optics algoritam je preuzeta iz firmware-a koji je napisao Darjan Crnčić, uz minimalne izmjene koje omogućuju čitanje iz i pisanje u buffere.