FREERTOS
JOURNAL FOR ØV. 2

E4ISD2

Janus Bo Andersen (JA67494)

Marts 2020

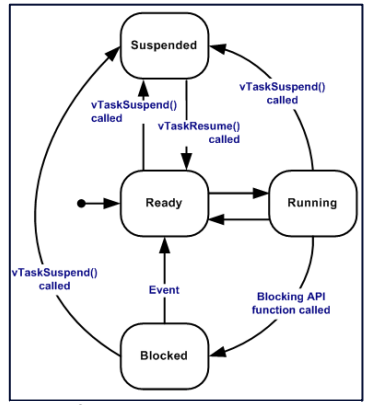
FREERTOS
OVERBLIK

Punkt	Detaljer

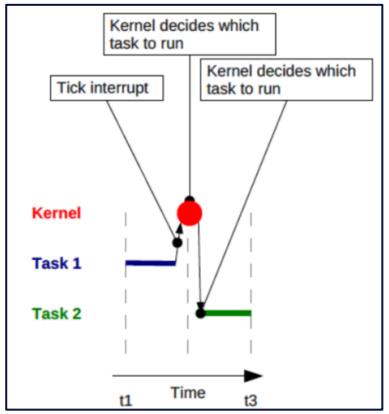
Hvad er det?	• Letvægts-realtidsoperativsystem til embeddede devices, giver mulighed for at designe til både hård realtid[1] og til et multi-tasking-paradigme[2] (se fig. 1).
	• FreeRTOS består af en mikrokerne:
	  • Kernen giver basale abstraktioner som 'tasks', 'states' (se fig. 2) og 'priorities'.
	  • Kernen laver 'scheduling' af tasks, som er klar til at køre (se fig. 3). Dette gøres ud fra en scheduling policy og task priorities. Som standard benyttes pre-emptive scheduling[3].
	  • Kernen stiller funktioner/mekanismer til afkobling af / kommunikation mellem / synkronisering af processer; navnlig 'queues' og 'semaphores', samt 'mutexes' til beskyttelse af delte ressourcer.
	  • Kernen implementerer forskellige metoder til dynamisk memoryhåndtering.
	  • Kernen indeholder ikke netværks-stack, drivers eller andet high-level.

Anvendelser af et realtidssystem	Oplagt at designe til hård realtid[1], hvor:
	• Sikkerhedsmæssige, regulatoriske eller UX-hensyn stiller krav til hurtig reaktion på events, inden for fastlagte tidsgrænser.
	  • Automotive, fx ABS, airbags, GPS,
	  • Medicinsk udstyr, fx pacemakers, kardiovaskulær overvågning, guidet defibrillator,
	  • Kontrolsystemer i industrien, fx fysiske/kemiske/biologiske procesanlæg,
	  • Våbensystemer, fx electronic warfare, osv.
	
	Oplagt at bruge multi-tasking-paradigmet[2] til systemer med:
	• Behov for flere processer / opdeling af programarkitektur i flere samtidige funktioner.
	• Behov for høj grad af responsiveness, *samtidig* med der udføres forskellige andre opgaver, såsom:
	  • sensoraftastning og beregning, seriel-kommunikation, opdatering af displays, osv.
	• Beregningstunge men ikke tids-kritiske opgaver, som med fordel kan flyttes i baggrunden.

Figur 1. Multitasking giver oplevelsen af at alle tasks kører samtidig, mens dog kun én afvikles ad gangen.



Figur 2. En task kan have følgende states og transitioner.



Figur 3. Ved hvert tick vælger scheduler en task med "Ready" og sætte denne til at afvikle.

1. Garanterede deadlines, dvs. garanteret reaktionstid eller frekvens, fx i ms eller antal ticks. Kan være 'strict timing', 'flexible timing' eller 'deadline-only timing'.
2. Flere samtidige processer med hver sin stack frem for procedural/sekventiel afvikling af en proces først efter en anden er kørt til slut.
3. Typen *pre-emptive* scheduling betyder at scheduler fordeler processortid mellem tasks på baggrund af bl.a. de definerede task-prioriteter. Hvis time-slicing er slået til, skiftes også mellem tasks med lige prioritet. Scheduler swapper tasks 'in' og 'out', og sørger for at kontekst til en task gemmes/genoprettes efter state er skiftet (se fig 2.). *Pre-emptive* er i modsætning til *cooperative*, som kræver at en task giver processoren tilbage (=yield'er) og til *time-sharing*, hvor tiden fordeles ligeligt mellem alle tasks.

QUEUES OG TASK-SYNKRONISERING
ØVELSE 2 UGE 7

Formål:

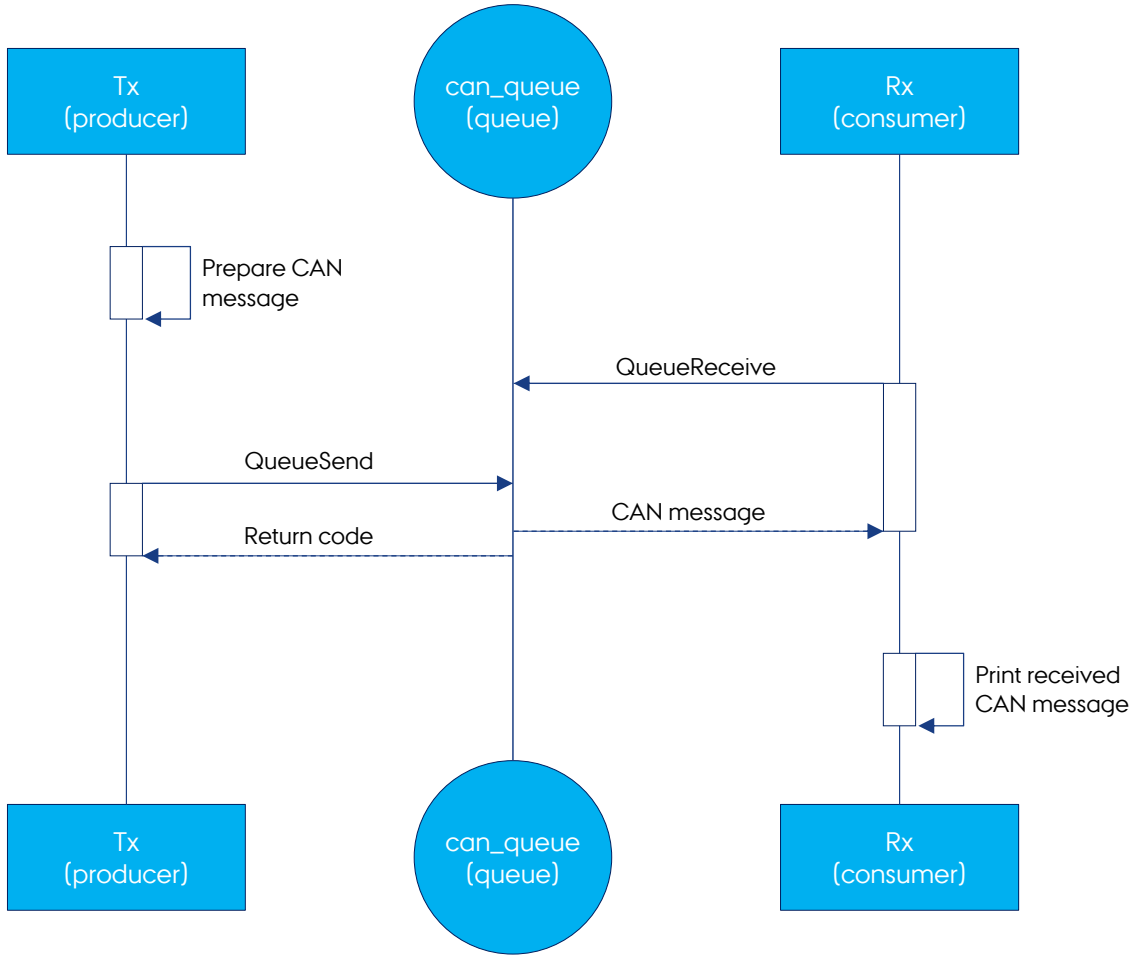
- Implementering af en kø mellem to tasks:
  - Udveksle CAN-beskeder
  - Task-synkronisering ved udveksling af beskeder

Opsætning:

- Ny datatype (CAN datagram)
- Kø med ny datatype
- Tx og Rx tasks i producer-consumer pattern

Metode (se figur):

- De to tasks kører asynkront af hinanden, men synkroniseres gennem queue
- Bemærk i illustration, at Rx kan forsøge at hente fra køen *før* der er data til rådighed
  - I så fald blokerer kaldet indtil der er data (eller indtil timeout er gået, hvad end sker først).
- Tx ville kunne blokere (el. gå timeout), hvis køen var fuld.



CAN DATATYPE OG HANDLES

I denne kodestump erklæres grundelementerne til programmet

```c
/**
 * @file    E4ISD2_wk07_FreeRTOS_Queue_rx_tx.c
 * @author  Janus Bo Andersen (JA67494)
 * @date    March 2020 (E4ISD2 spring 2020)
 * @brief   Main function and tasks to implement tx-rx system
 *          that sends CAN datagrams via a queue in FreeRTOS.
 * @note    Inspiration from https://www.youtube.com/watch?v=yHfDO_jiIFw
 */

#include <stdio.h>
#include <stdlib.h>
#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MKL25Z4.h"
#include "fsl_debug_console.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

#define QUEUE_LEN 5         // Queue depth

/* The CAN datagram type */
typedef struct {
    unsigned int ID;        // holds 11/29 bit ID
    unsigned char ID_ext;   // set to 1 if ID is extended -29 bit, else 0
    unsigned char DLC;      // how many bytes of payload
    unsigned char data[8];  // the data payload
} can_tlg_t;

/* Create handle for the two tasks */
TaskHandle_t tx_hdl = NULL;
TaskHandle_t rx_hdl = NULL;

/* Create handle for the queue */
QueueHandle_t can_queue = NULL;
```

Headerfilerne FreeRTOS.h og task.h er nødvendige for at lave et RTOS-projekt.

Headerfilen queue.h er nødvendig for at bruge kø-datastrukturen og tilhørende metoder.

Definerer, at der skal kunne holdes 5 elementer i køen. Det tal er helt arbitrært i dette eksempel.

Definerer datastrukturen til CAN-beskeder.

Handles til producer og consumer tasks.

Handle til køen erklæres som global variabel, så begge tasks kan få fat i den.

PRODUCER TASK: TX

I denne kodestump defineres **producer task** – dvs. funktionen, som opretter og afsender CAN-beskeder gennem køen.

```
41 /* @brief This task transmits random CAN datagrams
42  * @param None
43  *  */
44 void tx(void * p) {
45
46     can_tlg_t message;  // datagram
47
48     BaseType_t rc;      // gets return code from xQueueSend
49
50     uint8_t r = 0;      // gets the random values
51     srand(0);           // seed to always start same place
52
53     while(1) {
54
55         /* fill the control fields */
56         message.ID = 14;
57         message.ID_ext = 0;
58         message.DLC = 7;        // 7 bytes of data to be sent
59
60         /* fill the data fields */
61         for (int i = 0; i < message.DLC; i++) {
62             r = rand() % UINT8_MAX; // draw random num
63             message.data[i] = r;    // put in buffer
64         }
65
66         /* copy CAN telegram to the queue */
67         /* wait for 500 ticks before timeout */
68         printf("\nSent telegram with ID %d to receiver task. ", message.ID );
69         rc = xQueueSend(can_queue, &message, 500);
70         puts( rc ? "Sent.\n" : "Failed.\n" );
71
72         vTaskDelay(200);
73     }
74
75     // Task must not return!
76     vTaskDelete(NULL);
77 }
```

Udfylder CAN-beskedens kontrolfelter, som spec'et i opgaven

Udfylder CAN-beskedens datafelter med værdier 0-255.

Sætter (sender) CAN-beskeden ind i køen (bagenden).

Tjekker om det gik godt (ingen fejl eller timeout).

Vent 200 ticks (0.2 sek @ 1kHz tick rate) før næste CAN-besked begyndes

CONSUMER TASK: RX

I denne kodestump defineres **consumer task** – dvs. funktionen, som opretter og modtager CAN-beskeder gennem køen og udskriver dem til konsollen.

```
79 /* @brief This task receives CAN datagrams and prints them
80  * @param None
81  *  */
82 void rx(void * p) {
83
84     can_tlg_t message;      // incoming message
85
86     while(1) {
87         /* Receive from the queue, wait max 500 ticks */
88         if ( xQueueReceive(can_queue, &message, 500) ) {
89             // received a message
90             printf("Received CAN telegram with %d bytes of data:\n", message.DLC);
91             printf("ID: %d\n", message.ID);
92
93             for (int i = 0; i < message.DLC; i++) {
94                 printf("Data [%d]: %d\n", i+1, message.data[i]);
95             }
96
97         } else {
98             // failed to receive, or timed out
99         }
100    }
101
102    // Task must not return!
103    vTaskDelete(NULL);
104 }
```

Henter data fra køen (eller blokerer), og tjekker samtidig returværdien.

Udskriver kontrolfelter som spec'et i opgaven.

Udskriver datafelter, afhængigt af datalængden i bytes (DLC).

Ingen handling defineret her, men det ville give mening at definere betydningen af, at ingen data er modtaget inden timeout...

- Er det en fejl, der skal rapporteres?
- Er det et tegn på at gå i dvale?
- Skal der bare forsøges igen?
- Osv...

MAIN: OPRETTELSE AF KØ, OPRETTELSE AF TASKS OG START AF SCHEDULER

I denne kodestump oprettes kø-objektet (der allokeres hukommelse).

De to tasks oprettes.

Scheduler startes.

```
120     PRINTF("E4ISD2 FreeRTOS exercise 2/wk7. Transmission via queues.\n");
121
122     /* Create the queue to hold 5 CAN datagrams */
123     can_queue = xQueueCreate(QUEUE_LEN, sizeof(can_tlg_t) );
124
125     if (!can_queue) {
126         printf("Error. Queue could not be created.\n");
127         return -1;  /* No point continuing */
128     }
129
130
131     /* Create the tx and rx tasks */
132     xTaskCreate(tx,                         // function pointer to tx task
133                 "TX",                       // name for debugging
134                 configMINIMAL_STACK_SIZE*2, // stack size
135                 NULL,                       // passing parameters to task
136                 tskIDLE_PRIORITY+1,         // task prio (higher than idle)
137                 &tx_hdl );                  // handle to grab the task by
138
139     xTaskCreate(rx, "RX", configMINIMAL_STACK_SIZE*2,
140                 NULL, tskIDLE_PRIORITY+1, &tx_hdl );
141
142     /* Run the scheduler */
143     vTaskStartScheduler();
144
```

Køen oprettes (allokering). Der allokeres X antal elementer, der hver har størrelse som CAN-besked-typen.

Bekræft, at allokering er OK.

Opretter de to tasks, hhv. Tx og Rx. Bemærk især:

- Stack size er 2 * minimum (2*90 words). Det er et sjus, og sikkert mere end rigeligt.
- Prioritet for begge tasks er ens, og er højere end idle-task (0).

Start scheduler

OUTPUT FRA DE TO TASKS

Output fra de to tasks er som spec'et i opgaven.

Hastigheden er således, at der udskrives en ny CAN-besked ca. hvert sekund. Dette er grundet høj latency ved skrivning til stdout/terminal via debug-interfacet.

```
Sent telegram with ID 14 to receiver task. Sent.

Received CAN telegram with 7 bytes of data:
ID: 14
Data [1]: 62
Data [2]: 8
Data [3]: 44
Data [4]: 132
Data [5]: 116
Data [6]: 26
Data [7]: 86

Sent telegram with ID 14 to receiver task. Sent.

Received CAN telegram with 7 bytes of data:
ID: 14
Data [1]: 230
Data [2]: 13
Data [3]: 173
Data [4]: 157
Data [5]: 231
Data [6]: 212
Data [7]: 210

Sent telegram with ID 14 to receiver task. Sent.

Received CAN telegram with 7 bytes of data:
ID: 14
Data [1]: 47
Data [2]: 59
Data [3]: 181
Data [4]: 66
Data [5]: 239
Data [6]: 125
```

Producer-task (Tx) beretter om afsendt CAN-besked.

Den ventende consumer-task (Rx), beretter om en modtaget CAN-besked, og udskriver datafelterne.

Pga. ens prioriteter får producer-task lov til at køre færdig før consumer-task swappes ind (ellers var denne linje brudt).

CAN-beskedens dataindhold ændres hver gang, men kontroldata forbliver uændret.

KONKLUSION

FreeRTOS kan nemt benyttes til at oprette flere tasks, afvikle dem som multi-tasking og lade dem kommunikere/synkronisere:

- Tasks er afkoblede og asynkrone, men synkroniserer eller kommunikerer vha. køer.
- Den indbyggede kø-abstraktion hænger tæt sammen med state-systemet, så en ventende task blokerer – indtil indsættelse af data i køen får den ventende task til at rykke til "Ready" state (eller indtil den går timeout).
- Tilsvarende, hvis køen er fuld, så vil den producerende task vente på ledig plads (eller gå timeout).
- Et timeout skal håndteres alt efter "semantik" og kontekst: Det skal vides, hvad timeoutet egentlig betyder.

Det anvendte designmønster (producer-consumer) er meget anvendt, og er ud over i FreeRTOS anvendeligt til:

- Afkobling af processer i flertrådet programmering.
- Afkobling af devices, fx med indskudt serielinterface, fx til co-design-projektet ☺

AARHUS
UNIVERSITET