Opdrachten week 1 les 2 – Dynamisch geheugenallocatie

In de vorige les heb je een FIFO-buffer geïmplementeerd waarin een aantal getallen opgeslagen kan worden. Dit buffer wordt in een embedded IoT-applicatie gebruikt om meetdata op te slaan als de wifi-verbinding tijdelijk verbroken is. De code voor de buffer is eerst op de pc ontwikkeld en getest voordat deze code in de embedded applicatie is gebruikt. Doordat de code voor de buffer in een aparte module (een `.h` en een `.c` bestand) is opgenomen is het eenvoudig om deze code in verschillende applicaties te gebruiken. Zolang de interface van de module (in `buffer.h`) niet veranderd wordt, is het mogelijk om de implementatie van de module (in `buffer.c`) te wijzigen zonder dat de applicaties aangepast hoeven te worden.

In de buffer die je de vorige les hebt geïmplementeerd kan een maximaal aantal getallen (8) opgeslagen worden. Als je meer getallen in de buffer wilt opslaan, dan moet je een (hopelijk kleine) aanpassing maken in `buffer.c` en het programma opnieuw compileren. We zeggen dat het maximaal aantal getallen dat in de buffer opgeslagen kan worden *statisch* (vast) is. Als we te weinig ruimte voor de buffer reserveren, dan kan er data verloren gaan. Als we echter heel veel ruimte voor de buffer reserveren, hebben we veel RAM-geheugen nodig. Op de pc zal dit geen probleem zijn, maar op het embedded systeem zal de hoeveelheid RAM beperkt zijn.

Het zou natuurlijk veel mooier zijn als de buffer automatisch zou kunnen groeien en krimpen afhankelijk van de situatie. Dit kan gerealiseerd worden door gebruik te maken van *dynamische geheugenallocatie*. In de programmeertaal C kun je dynamisch geheugen alloceren met behulp van de standaardfunctie `malloc`[1]. Dynamisch gealloceerd geheugen kun je weer vrijgeven (voor hergebruik) met behulp van de standaardfunctie `free`[2].

Je leert deze les hoe je:

- dynamisch geheugen kunt alloceren en weer vrij kunt geven;
- een FIFO-buffer kunt implementeren die dynamisch groeit en krimpt met behulp van een *singly linked list*;
- een FIFO-buffer als een *user defined type* kunt definiëren waardoor je meerdere buffers in een programma kunt gebruiken.

Je kunt een dynamische *singly linked list* aanmaken door elementen van het hieronder gegeven type `buffer_element` dynamisch aan te maken en via de `next` pointers aan elkaar te rijgen.

[1] Zie: https://en.cppreference.com/w/c/memory/malloc.

[2] Zie: https://en.cppreference.com/w/c/memory/free.

```
typedef struct buffer_element_tag {
    int value;
    struct buffer_element_tag *next;
} buffer_element;
```

Als je dan een pointer naar het eerste element van de lijst genaamd `first` en een pointer naar het laatste element van de lijst genaamd `last` bijhoudt, dan kun je eenvoudig een FIFO-buffer implementeren. Bij een `buffer_put`-operatie moet een nieuw `buffer_element` aangemaakt worden met `malloc` en dit element moet achter het laatste element in de lijst geplaatst worden. De pointer `last` moet daarbij gebruikt en aangepast worden. Bij een `buffer_get`-operatie moet het eerste `buffer_element` losgemaakt worden uit de lijst en worden vrijgegeven met `free`. De pointer `first` moet daarbij gebruikt en aangepast worden. Bedenk dat er speciale acties nodig zijn als een element toegevoegd moet worden aan een lege lijst en ook als het allerlaatste element uit een lijst wordt verwijderd. De buffer is vol als `malloc` geen geheugen meer kan reserveren. De functie `malloc` geeft dan de waarde `NULL` terug.

1.2.1 Kopieer het directory `buffer_8` naar een nieuwe directory `buffer_dyn`. Verwijder het directory `build` uit het nieuwe directory `buffer_dyn`. Pas de testcode in het bestand `test.c` aan zodat getest wordt dat er een onbepaald aantal getallen in de buffer opgeslagen kan worden. Het geheugen moet dynamisch worden aangemaakt als een getal in de buffer wordt geplaatst en ook weer netjes vrijgegeven worden als een getal uit de buffer wordt verwijderd. De buffer moet werken als een FIFO-buffer (First In First Out).

1.2.2 Pas nu de code in het bestand `buffer.c` aan zodat er een onbepaald aantal getallen in de buffer opgeslagen kunnen worden. De buffer is pas vol als er geen geheugen meer beschikbaar is. Maak daarbij gebruik van een singly linked list zoals hierboven besproken. Het is niet de bedoeling dat het bestand `buffer.h` wordt aangepast, de interface van de buffer hoeft namelijk niet gewijzigd te worden. Door de interface niet te wijzigen zorg je ervoor dat de applicatiecode in `demo.c` en in het project `temperatuur_logger` niet gewijzigd hoeft te worden.

1.2.3 Je wilt natuurlijk ook testen of jouw implementatie goed werkt als het geheugen vol is? Dit is op de pc niet zo gemakkelijk omdat er zoveel RAM-geheugen beschikbaar is. Bovendien wordt op de pc zogenoemd virtueel geheugen gebruikt als het RAM-vol

is. Een deel van de inhoud van het RAM wordt dan verplaatst naar het achtergrondgeheugen (hard disk) als het RAM vol is. Dit wordt dan later weer teruggezet als dit nodig is, waarbij ruimte wordt gemaakt door de inhoud van een deel van het RAM dat momenteel niet nodig is naar het achtergrondgeheugen te kopiëren. Om te testen of je code goed werkt als het geheugen vol is, kun je zelf een functie `test_malloc` definiëren die bijvoorbeeld de eerste 100 keer gewoon `malloc` aanroept maar de 101ste keer `NULL` teruggeeft. Implementeer de functie `test_malloc` en pas de code in het bestand `buffer.c` aan zodat deze functie wordt gebruikt in plaats van `malloc`. Test of de dynamische buffer nog steeds correct werkt, ook als het geheugen vol is.

1.2.4 Je wilt natuurlijk ook testen dat het dynamisch gealloceerd geheugen ook weer correct wordt vrijgegeven. Dit is nog niet zo eenvoudig. Je zou bijvoorbeeld zelf een functie `test_malloc` en `test_free` kunnen definiëren die bijhouden hoe vaak ze worden aangeroepen en vervolgens `malloc` respectievelijk `free` aanroepen. Je kunt dan checken of als de buffer weer leeg is, `test_free` net zo vaak is aangeroepen als `test_malloc`. Implementeer de functies `test_malloc` en `test_free` en pas de code in het bestand `buffer.c` aan zodat deze functies worden gebruikt in plaats van `malloc` en `free`. Test of al het dynamisch gealloceerd geheugen ook weer correct wordt vrijgegeven

De FIFO-buffer die je als module hebt geïmplementeerd heeft één belangrijke tekortkoming. Je kunt per applicatie slechts één buffer gebruiken. Het zou natuurlijk veel mooier zijn als je dynamisch zoveel buffers zou kunnen aanmaken als je nodig hebt. Zo'n herbruikbaar FIFO-buffer waarvan je er zoveel kunt aanmaken als je wilt, noemen we een *user defined type* (UDT). Om dit mogelijk te maken moet de interface van de buffer aangepast worden.

Een bruikbare interface is gegeven in `buffer.h`:

```
#ifndef _HR_BroJZ_buffer_
#define _HR_BroJZ_buffer_

#include <stdbool.h>

// interface for buffers with int's

typedef struct buffer_tag *buffer;

// create a new buffer
// returns NULL on failure
```

```
buffer new_buffer(void);

// delete a buffer b
void delete_buffer(buffer b);

// put value i in buffer b if buffer b is not full
// returns true on success or false otherways
bool buffer_put(buffer b, int i);

// get value from buffer b and writes it to *p if buffer b not empty
// returns true on success or false otherways
bool buffer_get(buffer b, int *p);

// returns true when buffer b is full or false otherways
bool buffer_is_full(buffer b);

// returns true when buffer b is empty or false otherways
bool buffer_is_empty(buffer b);

#endif
```

Merk op dat een buffer geïdentificeerd wordt met behulp van een parameter van het type `buffer`. Dit type is een pointer naar een **`struct`** `buffer_tag` maar dit type is niet gedefinieerd in de file `buffer.h`. De gebruiker van de UDT buffer hoeft dus niet te weten hoe een buffer opgeslagen en geïmplementeerd wordt.

1.2.5 Kopieer het directory `buffer_dyn` naar een nieuwe directory `buffers_dyn`. Pas de testcode in het bestand `test.c` aan zodat getest wordt dat er een onbepaald aantal FIFO-buffers aangemaakt en weer verwijderd kan worden. Het benodigde geheugen moet dynamisch worden aangemaakt als een buffer wordt aangemaakt en weer worden vrijgegeven als een buffer wordt verwijderd. In elk FIFO-buffer moet een onbepaald aantal getallen opgeslagen kunnen worden. Het geheugen moet dynamisch worden aangemaakt als een getal in de buffer wordt geplaatst en ook weer netjes vrijgegeven worden als een getal uit de buffer wordt verwijderd. Let er op dat het geheugen ook correct wordt vrijgegeven als een buffer dat nog getallen bevat wordt verwijderd.

1.2.6 Pas nu de code in het bestand `buffer.c` aan zodat er een onbepaald aantal FIFO-buffers aangemaakt en weer verwijderd kan worden. In elk FIFO-buffer moet een

onbepaald aantal getallen opgeslagen kunnen worden. Het type `struct buffer_tag` zou bijvoorbeeld als volgt gedefinieerd kunnen worden:

```
struct buffer_tag {
    buffer_element *first;
    buffer_element *last;
    bool full;
};
```

Er kan geen buffer meer aangemaakt worden en geen element meer aan een buffer worden toegevoegd als er geen geheugen meer beschikbaar is.

1.2.7 In het bestand `demo.c` is een demoprogramma gegeven waarin twee FIFO-buffers gebruikt worden. Voer dit programma uit met jouw implementatie van de UDT buffer op de pc. In het bestand `demo_CC3220S.c` is een versie van dit programma gegeven dat onder TI-RTOS kan draaien. Voer dit programma uit met jouw implementatie van de UDT buffer op de CC3220S LaunchPad.