

Eindopdracht 1

Veel embedded applicaties beschikken over een command line interface (CLI). Deze interface is meestal niet bedoeld voor de eindgebruiker van het embedded product, maar kan bijvoorbeeld bij onderhoudswerkzaamheden worden gebruikt om:

- log gegevens op te vragen;
- bepaalde tests uit te voeren;
- bepaalde instellingen te wijzigen.

Als eerste eindopdracht voor EMS30 ga je zo'n CLI voor de CC3220S LaunchPad ontwerpen, implementeren en uitgebreid testen. Je laat hiermee zien dat je de leerdoelen uit week 1 tot en met 3 hebt behaald en dat je een betere C-programmeur bent geworden.

Je laat zien dat je hebt geleerd hoe je:

- de *interface* van een module in een `.h` bestand kunt *declareren*;
- de *implementatie* van een module in een `.c` bestand kunt *definiëren*;
- deze module kunt gebruiken en testen op een pc;
- deze module kunt gebruiken en testen op een CC3220S LaunchPad;
- een datastructuur kunt implementeren die dynamisch groeit en krimpt met behulp van **structs** en pointers;
- softwareontwikkeling kan doen volgens de zogenoemde git-flow;
- gelijktijdig meerdere nieuwe features voor een embedded applicatie kunt ontwikkelen;
- C-code systematisch kan testen met behulp van het eenvoudige test-framework Catch2;
- tests automatisch na elke commit kan laten uitvoeren, door gebruik te maken van Bitbucket pipelines;
- test-stubs kunt gebruiken om software die bedoeld is voor een embedded systeem, toch zo veel mogelijk op een pc te kunnen testen;
- fouten bij het gebruik van dynamische geheugenallocatie zoals geheugenlekken kunt opsporen met speciale tools zoals valgrind;
- weet dat je genoeg testcode hebt geschreven door gebruik te maken van test coverage metingen;
- gebruik kunt maken van verschillende *C coding standards*;
- gebruik kunt maken van verschillende *static code analysis* tools, waaronder Cppcheck, om de kwaliteit van je C-code te verbeteren.

De te ontwikkelen CLI kan gebruikt worden om de leds van de CC3220S LaunchPad mee aan te sturen. We beschouwen de 3 leds als een binair getal waarbij groen het meest significante bit en rood het minst significante bit voorstelt. De waarde zes (binair 110) wordt dus aangegeven door alleen de groene en gele led te laten branden. De CLI moet beschikken over de volgende specifieke commando's:

nul	zet de decimale waarde 0 op de leds
een	zet de decimale waarde 1 op de leds
twee	zet de decimale waarde 2 op de leds
drie	zet de decimale waarde 3 op de leds
vier	zet de decimale waarde 4 op de leds
vijf	zet de decimale waarde 5 op de leds
zes	zet de decimale waarde 6 op de leds
zeven	zet de decimale waarde 7 op de leds
schakelroodaan	zet de rode led aan
schakelrooduit	zet de rode led uit
schakelroodom	toggle de rode led
schakelgeelaan	zet de gele led aan
schakelgeeluit	zet de gele led uit
schakelgeelom	toggle de gele led
schakelgroenaan	zet de groene led aan
schakelgroenuit	zet de groene led uit
schakelgroenom	toggle de groene led

Verder moet de CLI ook beschikken over de volgende generieke commando's:

einde	sluit het programma af
af	sluit het programma af

De commando's moeten 'case insensitive' zijn. Dat wil zeggen dat de commando's een, EEN, Een en eeN allemaal hetzelfde zijn. De CLI moet na het intypen van een commando gevolgd door het indrukken van de `Enter`-toets het ingetypte commando uitvoeren. Als het ingetypte commando niet bestaat, dient een passende foutmelding te worden gegeven. De CLI moet gebruikt kunnen worden via de UART-interface van de CC3220S LaunchPad. We maken gebruik van het terminal programma Tera Term¹, zodat we ANSI/VT100 terminalcommando's²

¹ Zie <https://ttssh2.osdn.jp/index.html.en>.

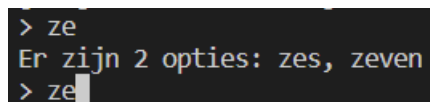
² Deze commando's kun je bijvoorbeeld gebruiken om de tekst in kleur weer te geven of om de pijltjestoetsen te verwerken, zie <https://www2.ccs.neu.edu/research/gpc/VonaUtils/vona/terminal/vtansi.htm>.

kunnen gebruiken. Elk printbaar karakter moet na ontvangst teruggestuurd (geëchood) worden over de UART-verbinding. Als de gebruiker op de `Backspace`-toets drukt dan moet het laatst ontvangen karakter genegeerd worden. Je kunt de karakterstring "`\x1b[D \x1b[D`" terugsturen om er voor te zorgen dat het karakter ook in het terminalprogramma wordt gewist.

Om deze CLI gebruiksvriendelijk te maken, *kunnen* de volgende extra features geïmplementeerd worden:

- **command completion**

Als de eerste letter of letters van een commando zijn ingetypt en de gebruiker drukt de `Tab`-toets³ in, dan wordt het commando automatisch aangevuld. Bijvoorbeeld: als de gebruiker `n` `Tab` intypt, dan verschijnt het commando `nul`. In het geval dat er meer mogelijkheden zijn om het commando automatisch aan te vullen gebeurt er niets (meer). Bijvoorbeeld: als de gebruiker `z` `Tab` intypt, dan verschijnt de tekst `ze`. Er zijn namelijk twee commando's die met een `z` beginnen, deze commando's hebben beide als tweede letter een `e`, maar de derde letter verschilt. Als de gebruiker dan *nogmaals* op de `Tab`-toets drukt, dan wordt een lijst met opties weergegeven. Bijvoorbeeld: als de gebruiker `z` `Tab` `Tab` intypt, dan verschijnt de uitvoer zoals weergegeven zoals in [figuur 1](#).



```
> ze
Er zijn 2 opties: zes, zeven
> ze
```

Figuur 1: Command completion na het intypen van `z` `Tab` `Tab`.

- **command history**

Het is gemakkelijk voor de gebruiker als een eerder ingevoerd commando eenvoudig herhaald kan worden. Dit kan geïmplementeerd worden door de uitgevoerde commando's in een LIFO-queue⁴ op te nemen. De gebruiker kan dan met de `↑`- en `↓`-toets door deze queue 'heenlopen'.

Niet printbare karakters die geen speciale functie hebben in de CLI moeten genegeerd worden.

E1.1 Implementeer de CLI die de bovenstaande commando's kan uitvoeren, *zonder* de hierboven beschreven extra features op de CC3220S LaunchPad. De commando's met

³ De `Tab`-toets wordt ook in veel andere CLI's, zoals bijvoorbeeld bash, voor command completion gebruikt.

⁴ Een LIFO-queue (Last In First Out) wordt ook wel een stack genoemd.

hun bijbehorende functie *moeten* in een dynamische boomstructuur worden opgeslagen. Zodat de programmeur op eenvoudige wijze commando's aan de CLI kan toevoegen. Op [pagina 5](#) worden nog enkele implementatietips gegeven. Ontwikkel en test deze software zoveel mogelijk op de pc, door, waar nodig, gebruik te maken van stubs. Ontwikkel de software zoveel mogelijk volgens de TDD-methode⁵. Maak daarbij gebruik van Catch2.

- E1.2** Controleer de ontwikkelde software op geheugenfouten. Maak daarbij gebruik van valgrind.
- E1.3** Zorg ervoor dat je voldoende testcode hebt geschreven om (bijna) alle ontwikkelde software te testen. Maak daarbij gebruik van test coverage metingen.
- E1.4** Verbeter de kwaliteit van de ontwikkelde software door gebruik te maken van static code analysis tools en coding standards. Maak daarbij op zijn minst gebruik van Cppcheck.
- E1.5** Schrijf een kort verslag waarin je de belangrijkste ontwerpbeslissingen en alle testresultaten (zowel op de pc als op de CC3220S LaunchPad) beschrijft.

Met bovenstaande opdrachten kun je maximaal 70 punten verdienen. De overige 30 punten kun je verdienen door de onderstaande opdrachten uit te voeren:

- E1.6** Implementeer de hierboven beschreven extra features van de CLI: command completion en command history op de CC3220S LaunchPad. Maak hierbij gebruik van de git-flow. Ontwikkel en test deze features zoveel mogelijk op de pc volgens de TDD-methode met behulp van Catch2. Op [pagina 6](#) wordt nog een implementatietip voor de command history feature gegeven.
- E1.7** Controleer de toegevoegde code op geheugenfouten. Maak daarbij gebruik van valgrind.

⁵ Bij de zogenoemde "Test Driven Design"-methode wordt eerst een falende test geschreven. Vervolgens wordt de code geschreven om de test te laten slagen. Hierna wordt de code indien nodig vereenvoudigd en beter leesbaar en onderhoudbaar gemaakt (dit wordt in het Engels 'code refactoring' genoemd) en weer getest. Hierna wordt weer een falende test geschreven en wordt de beschreven cyclus herhaald.

- E1.8** Zorg ervoor dat je voldoende testcode hebt geschreven om (bijna) alle toegevoegde code te testen. Maak daarbij gebruik van test coverage metingen.
- E1.9** Verbeter de kwaliteit van de toegevoegde software door gebruik te maken van static code analysis tools en coding standards. Maak daarbij op zijn minst gebruik van Cppcheck.
- E1.10** Schrijf een kort verslag waarin je de belangrijkste ontwerpbeslissingen en alle testresultaten (zowel op de pc als op de CC3220S LaunchPad) van de toegevoegde features beschrijft.

Implementatietips

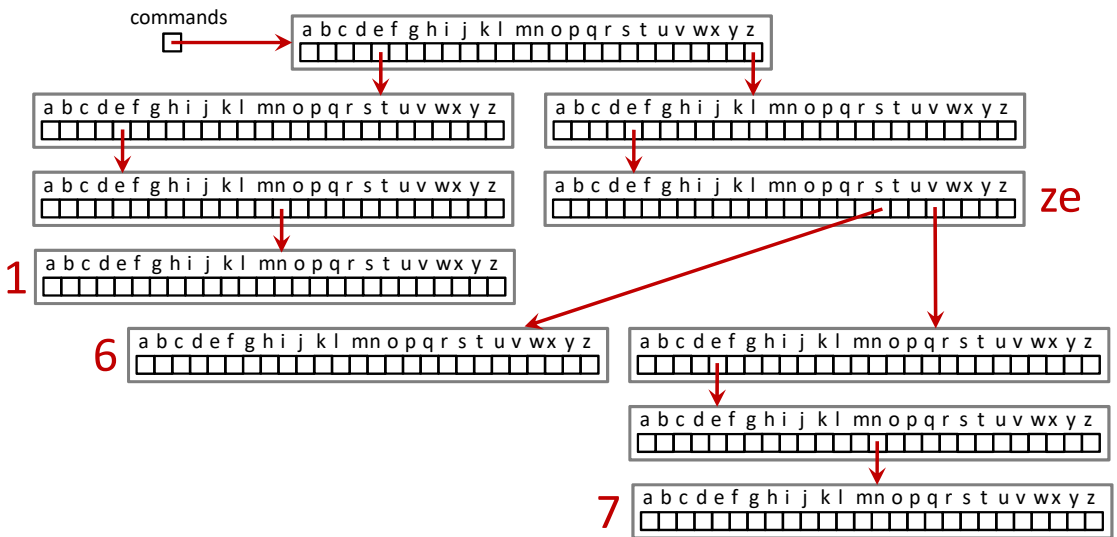
Karakters die binnenkomen via een UART-verbinding kun je meteen inlezen in je programma. Als je een programma uitvoert op de pc in een CLI zoals de terminal in WSL, dan worden ingetypte karakters pas doorgestuurd naar het programma als je op de `Enter`-toets drukt. Dit wordt de *cooked mode* genoemd. Voor deze applicatie is dat niet handig. Je kunt ervoor zorgen dat karakters wel meteen worden doorgegeven aan het programma, zodra ze ingetypt worden, door de zogenaamde *raw mode* te gebruiken. Hoe je dit doet is afhankelijk van het platform en het type terminal dat je gebruikt en kun je vinden op https://bitbucket.org/HR_ELEKTRO/ems30/wiki/RawTerminal.md. Ingetypte karakters kunnen vervolgens meteen worden ingelezen met behulp van de functie `getchar`.

Als datastructuur om de commando's in op te slaan moet je gebruik maken van een met dynamische geheugenallocatie aangemaakte zogenoemde 'boom' (Engels: *tree*). Een tree bestaat uit een zogenoemde *root* met vertakkingen (pointers) naar volgende *nodes*. In dit geval bevat elke node 26 vertakkingen (pointers): één voor elke letter uit het alfabet. Je kunt dan ingetypte commando's eenvoudig vinden door per letter de betreffende vertakking (pointer) te volgen, zie [figuur 2](#). Alle 'lege' pointers in [figuur 2](#) bevatten de waarde `NULL`.

Het type van elke node in de tree zou dan als volgt gedefinieerd kunnen zijn:

```
typedef void (*command_function)(void);

typedef struct command_node_tag {
    command_function function;
    struct command_node_tag *next[26];
} command_node;
```



Figuur 2: Een mogelijke datastructuur om de commando's in op te slaan.

Als de `command_node` een geldig commando vertegenwoordigd, dan:

- verwijst de functiepointer `function` naar de functie die aangeroepen moet worden als het betreffende commando uitgevoerd moet worden.
- verwijzen de 26 pointers uit de array `next` naar commando's met hetzelfde begin als het betreffende commando.

Als de `command_node` geen geldig commando vertegenwoordigd, dan is de pointer `function` gelijk aan `NULL`.

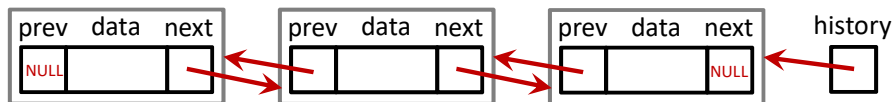
Als de CLI alleen de commando's een, zes en zeven zou bevatten, dan zou de tree opgebouwd zijn zoals gegeven in [figuur 2](#). De `command_node` gemarkeerd met **1** in [figuur 2](#) bevat dan de functiepointer naar de functie die moet worden uitgevoerd als het commando een is ingetypt. De `command_nodes` gemarkeerd met **6** en **7** bevatten respectievelijk de functiepointer voor de commando's zes en zeven.

Het implementeren van command completion is met behulp van deze tree ook relatief eenvoudig. Als de gebruiker bijvoorbeeld `z` `Tab` intypt, dan kan, door de z-pointer vanuit de root te volgen, eenvoudig worden bepaald dat het volgende karakter een e moet zijn. Het commando kan echter niet verder afgemaakt worden, omdat er hierna twee mogelijkheden zijn. Als de gebruiker vervolgens nogmaals op `Tab` drukt, kunnen de verschillende opties

(zes en zeven) gevonden worden door de s- en v-pointer vanuit de met **ze** gemarkeerde node te volgen.

De hier beschreven implementatiemethode met behulp van een tree maakt de implementatie relatief eenvoudig, maar kost wel veel geheugen. Je kunt ook voor een implementatie kiezen die minder geheugenruimte kost, bijvoorbeeld een alfabetisch geordende binaire boom⁶ van commando's.

De command history feature kan relatief eenvoudig gerealiseerd worden met behulp van een zogenoemde *doubly linked list*. Elk element (vaak *node* genoemd) uit zo'n lijst bevat een pointer genaamd *next* naar het volgende en ook een pointer genaamd *prev* naar het vorige element, zie **figuur 3**. De pointer data wijst naar het betreffende ingevoerde commando.



Figuur 3: Een mogelijke datastructuur om de command history in op te slaan.

Het type van elke node in de doubly linked list zou dan als volgt gedefinieerd kunnen zijn:

```
typedef struct history_node_tag {
    char *command;
    struct history_node_tag* prev;
    struct history_node_tag* next;
} history_node;
```

De pointer *history* wijst naar het laatst ingevoerde commando. Met behulp van de *prev*- en *next*-pointers kan relatief eenvoudig door de eerder ingevoerde commando's 'heengelopen' worden.

In **figuur 4** is te zien hoe de CLI op een pc zou kunnen werken.

⁶ Zie eventueel: https://en.wikipedia.org/wiki/Binary_tree.

```
> hallo
Onbekend commando: "hallo"
> nul
leds: groen = 0, geel = 0, rood = 0
> ze
Er zijn 2 opties: zes, zeven
> zes
leds: groen = 1, geel = 1, rood = 0
> schakelroodaan
leds: groen = 1, geel = 1, rood = 1
> schakelgroenuit
leds: groen = 0, geel = 1, rood = 1
> schakelgeelom
leds: groen = 0, geel = 0, rood = 1
> schakelgeelom
leds: groen = 0, geel = 1, rood = 1
> █
```

Figuur 4: Een mogelijke implementatie met behulp van stubs van de CLI op een pc.