Opdrachten kwartaal 3 week 1 – Modules in C

Bij het programmeren van embedded systemen wordt meestal de programmeertaal C gebruikt[1]. Het is echter niet handig om alle software voor het embedded systeem meteen op het embedded systeem te testen. Grote delen van de software kunnen op een pc ontwikkeld en getest worden. Dit heeft als voordeel dat je deze delen kan testen zonder dat het embedded systeem, waar de software uiteindelijk voor bedoeld is, beschikbaar is. Bij het ontwikkelen en testen van software op de pc is het bovendien eenvoudiger om software-engineering technieken zoals *test-driven development* (TDD)[2] en *continuous integration* (CI)[3] te gebruiken. We komen later in deze cursus nog op deze technieken terug.

Het is handig om de delen van een embedded applicatie die op de pc ontwikkeld en getest worden in zogenoemde softwaremodules op te nemen zodat ze eenvoudig geïntegreerd kunnen worden in de embedded applicatie. In tegenstelling tot veel andere programmeertalen, zoals bijvoorbeeld Python[4], heeft de programmeertaal C geen directe support voor modulair programmeren[5]. Toch kun je in C, door je programma op te splitsen in losse `.c` en `.h` bestanden, op een modulaire manier software ontwikkelen.

Als voorbeeld gebruiken we bij deze opdracht een eenvoudig buffer dat we uiteindelijk in een embedded toepassing willen gebruiken. De uiteindelijke toepassing maakt gebruik van een CC3220S processor van Texas Instruments. Deze software zal in een van de volgende weken getest worden op een CC3220S LaunchPad. Deze week gaan we de buffer eerst op een pc ontwikkelen en testen.

Om software op de pc te kunnen ontwikkelen en testen maken we gebruik van de veelgebruikte open-source C compiler die is opgenomen in gcc (GNU compiler collection)[6] en de bijbehorende debugger gdb (GNU debugger)[7]. Deze software is via een *command-line interface* (CLI) te gebruiken. De meeste studenten zullen echter de voorkeur geven aan een

[1] Bron: https://bytebucket.org/HR_ELEKTRO/ems31/wiki/Documenten/Aspencore_2023_Embedded_Markets_Study.pdf#page=9.

[2] Zie eventueel: https://en.wikipedia.org/wiki/Test-driven_development.

[3] Zie eventueel: https://en.wikipedia.org/wiki/Continuous_integration.

[4] Zie eventueel: https://docs.python.org/3/tutorial/modules.html.

[5] Bron: https://en.wikipedia.org/wiki/Modular_programming.

[6] Zie eventueel: https://gcc.gnu.org/.

[7] Zie eventueel: https://www.gnu.org/software/gdb/.

integrated development environment (IDE) met een *graphical user interface* (GUI). Wij gaan daarom gebruik maken van Visual Studio Code[8].

Sommige tools die we gaan gebruiken zijn alleen beschikbaar onder Linux. Als je Windows gebruikt, dan ga je gebruik maken van Windows Subsystem for Linux (WSL)[9] om deze tools te kunnen gebruiken.

Een installatiehandleiding om deze programma's op je eigen machine te installeren kun je vinden op: https://bitbucket.org/HR_ELEKTRO/ems31/wiki/Software_WSL.md.

3.1.1 Voer de hierboven genoemde installatiehandleiding uit en zorg ervoor dat je Visual Studio Code kunt gebruiken om C-code onder WSL te schrijven en te debuggen.

Je leert deze les 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 m.b.v. professionele tools zoals Windows Subsystem for Linux (WSL), Visual Studio Code, make en CMake;
- snel en efficiënt kunt werken met WSL door een aantal eenvoudige Linux-commando's te gebruiken zoals `cd`, `wget`, `unzip`, `rm`, `mkdir` en `cp`.

In deze les implementeren we een eenvoudige buffer. Een buffer wordt vaak gebruikt in embedded toepassingen, bijvoorbeeld voor het implementeren van een digitaal filter. Er zijn natuurlijk al veel bestaande implementaties die je zou kunnen (her)gebruiken, maar in deze les gaan we juist zelf een buffer implementeren om te leren hoe je in C modulair kan ontwikkelen. We beginnen met een eenvoudige implementatie van een buffer dat slechts één element kan bevatten en breiden dit buffer in deze en de volgende lessen uit tot een volwaardige implementatie.

Als we een softwaremodule zowel in een pc-programma als in een embedded programma willen gebruiken, dan is het logisch om deze code op te nemen in een apart bestand. Daarbij splitsen we de module in twee bestanden: een `.h` en een `.c` bestand. In het zogenoemde *header bestand* (het `.h` bestand) declareren[10] we de functies en de typedefinities die in de module zijn opgenomen. We noemen dit ook wel de *interface* van de module omdat de

[8] Zie: https://code.visualstudio.com/.

[9] Zie: https://learn.microsoft.com/en-us/windows/wsl/about.

[10] Dat betekent dat alleen het prototype (de eerste regel) van de functies in het `.h` bestand staan.

module gebruikt kan worden via de functies die in het `.h` bestand gedeclareerd zijn. Het is daarbij niet nodig om te weten hoe deze functies geïmplementeerd zijn.

Het bestand `buffer.h` heeft de volgende inhoud:

```
#ifndef _HR_BroJZ_buffer_
#define _HR_BroJZ_buffer_

#include <stdbool.h>

// interface for a buffer with int's

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

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

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

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

#endif
```

Door middel van de preprocessor directives **`#ifndef`** enz. worden compilatiefouten voorkomen als de programmeur die de module gebruikt het bestand `buffer.h` per ongeluk meerdere malen geïnclude heeft. De eerste keer dat het bestand geïnclude wordt, wordt het symbool `_HR_BroJZ_buffer_` gedefinieerd. Als het bestand daarna opnieuw geïnclude wordt, wordt in de **`#ifndef`** 'gezien' dat het symbool `_HR_BroJZ_buffer_` al bestaat en wordt pas bij de **`#endif`** weer verder gegaan met vertalen.

Zoals je hebt gezien bevat het bestand `buffer.h` slechts de *declaraties* van de functies waarmee de buffer gebruikt kan worden. Deze functiedeclaraties worden ook wel *prototypes* genoemd[11]. Het (gewenste) gedrag van elke functie is beschreven in het, aan het prototype voorafgaande, commentaar. De prototypes worden voorafgegaan door het keyword **`extern`**.

[11] Zie eventueel: https://en.cppreference.com/w/c/language/function_declaration.

Dit is een zogenoemde *storage-class specifier*[12] die aangeeft dat de betreffende functie ook aan te roepen is van buiten (oftewel 'extern') het bestand waarin de functie gedefinieerd is. Als bij het prototype van een functie geen storage-class specifier wordt opgegeven dan is die per default **extern**. De meeste programmeurs nemen niet de moeite om expliciet aan te geven dat de prototypes **extern** zijn.

Het type `bool` wordt gebruikt in de prototypes, vandaar dat het bestand `stdbool.h` geïnclude moet worden.

Het bestand `buffer.c` bevat onder andere de volgende code:

```
#include "buffer.h"
// implementation for a buffer with ints

// shared variables within this file
static int buffer;
static bool full = false;
```

De definities van de door de functies gedeelde variabelen worden voorafgegaan door het keyword **static**. Dit is een zogenoemde *storage-class specifier*[13] die aangeeft dat de betreffende variabele alleen te gebruiken is in het bestand waarin de variabele gedefinieerd is. De functies die gedeclareerd zijn in `buffer.h` worden gedefinieerd in `buffer.c` en kunnen dus gebruik maken van de gedeelde variabelen `buffer` en `full`. Maar vanuit andere bestanden kan geen gebruik gemaakt worden van deze variabelen.

3.1.2 Bestudeer de C-code in de bestanden `buffer.h` en `buffer.c` en zorg ervoor dat je deze code begrijpt. Vraag eventueel extra uitleg aan de docent als je iets niet begrijpt.

Er is ook een programma genaamd `demo.c` gegeven waarin gebruik gemaakt wordt van de buffer module. Dit programma bevat onder andere de volgende code:

```
#include "buffer.h"

void *producer(void *arg)
{
//   ...
    if (!buffer_is_full())
```

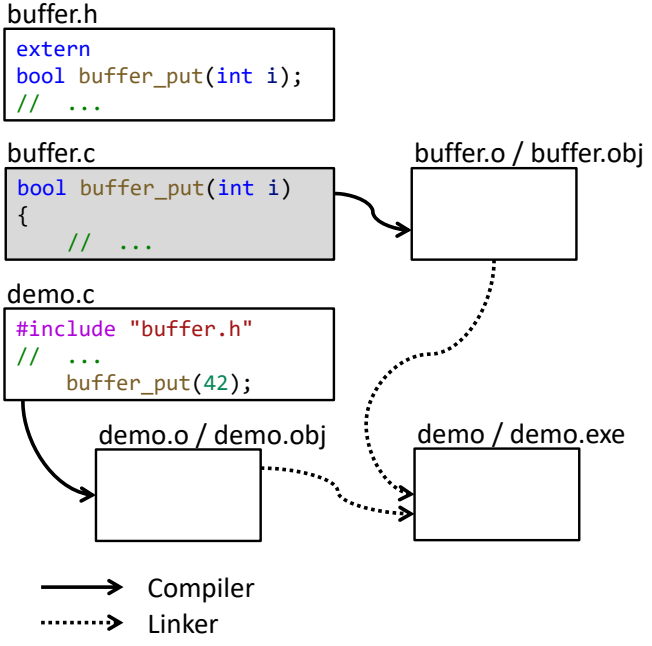
[12] Zie eventueel: https://en.cppreference.com/w/c/language/storage_duration.

[13] Zie eventueel: https://en.cppreference.com/w/c/language/storage_duration.

```
{
    buffer_put(i);
}
```

In `demo.c` wordt het bestand `buffer.h` geïnclude zodat de functies, die in `buffer.c` gedefinieerd zijn, aangeroepen kunnen worden.

Beide `.c` bestanden kunnen nu los van elkaar gecompileerd worden tot zogenoemde objectbestanden. Een objectbestand bevat de machinecode van het betreffende `.c`-bestand en heeft afhankelijk van de gebruikte compiler de extensie `.o` of `.obj`. Vervolgens kunnen de objectbestanden met een zogenoemde *linker* gecombineerd worden tot een uitvoerbaar bestand, een zogenoemde executable. Een uitvoerbaar bestand heeft op het Windows operating systeem de extensie `.exe` en op het Linux operating systeem meestal geen extensie. Zie figuur 1. De linker voegt trouwens ook nog objectcode toe uit de standaard C-library zoals bijvoorbeeld de objectcode van `printf` maar dit is niet weergegeven in figuur 1.



Figuur 1: Het compileren en linken van de module buffer.

Het bestand `buffer.c` is in figuur 1 grijs gekleurd om aan te geven dat dit bestand niet nodig is om de module buffer te kunnen gebruiken. De files `buffer.h` en `buffer.o` zijn voldoende om de module buffer te kunnen gebruiken. Dit geeft ons de mogelijkheid om een module aan andere programmeurs beschikbaar te stellen zonder de C-code van onze functies vrij te

geven (closed source). In de praktijk wordt het `.o` dan meestal met andere `.o` bestanden opgenomen in een zogenoemd static *library* bestand (met de extensie `.a` of `.lib`). Zo'n library kan dan via de linker worden gekoppeld aan de rest van de code.

3.1.3 Start Visual Studio Code en druk op F1 om het commando venster te openen. Type "WSL" en kies voor "WSL: Connect to WSL in New Window". Open een terminal[14] window in Visual Studio Code door op ctrl + ‘ te drukken. Installeer de programma's `wget` en `unzip` met het volgende commando[15]:

```
sudo pacman -Sy wget unzip
```

Ga naar het directory `Opdrachten` door in het terminal window in te typen:

```
cd Opdrachten
```

Download het bestand `buffer_1.zip`, pak het uit en verwijder het weer met de volgende commando's:

```
wget https://bitbucket.org/HR_ELEKTRO/ems31/raw/master/Program-↩
 ↪ mas/buffer_1.zip
unzip buffer_1.zip
rm buffer_1.zip
```

Voer vervolgens onderstaande commando's uit om naar het directory `buffer_1` te gaan en daar de bestanden `buffer.c` en `demo.c` te compileren. Tot slot link je de objectbestanden samen tot één executable.

```
cd buffer_1
gcc -std=c18 -Wall -Wextra -pedantic-errors -g3 -O0 -c buffer.c
gcc -std=c18 -Wall -Wextra -pedantic-errors -g3 -O0 -c demo.c
gcc buffer.o demo.o -o demo
```

Met het eerste commando ga je naar het directory `buffer_1`. Met de volgende twee commando's worden `buffer.c` en `demo.c` gecompileerd naar respectievelijk `buffer.o`

[14] In dit terminalwindow van Visual Studio Code draait een zogenoemde bash shell waarin je Linux-commando's kunt uitvoeren. Mocht je geen ervaring met Linux hebben; in bijlage A kun je een lijstje met standaard commando's terugvinden die je in een Linux-omgeving kunt gebruiken. De bash shell heeft vele mogelijkheden en is niet alleen een command interpreter maar ook een programmeertaal. Zie eventueel `https://www.gnu.org/software/bash/manual/bash.pdf`.

[15] Het "[sudo] password for ems" is : `ems`.

en `demo.o`. Met de optie `-std=c18` geef je aan dat je de C18-standaard[16] wilt gebruiken. Met de opties `-Wall -Wextra` zorg je ervoor dat de compiler zoveel mogelijk warnings geeft bij potentiële fouten in je code. De optie `-pedantic-errors` zorgt ervoor dat de compiler errors geeft als je code niet aan de C18-standaard voldoet. Met de optie `-g3` zal de compiler uitgebreide debug-informatie in de objectbestanden opnemen die gebruikt kan worden door de debugger gdb. De optie `-O0` zorgt ervoor dat de code niet geoptimaliseerd wordt. Tijdens het debuggen is dat ook niet gewenst. Je wilt tijdens het debuggen het programma zoals je het hebt geschreven kunnen volgen en dat kan niet als de compiler bepaalde code of variabelen weg-geoptimaliseerd heeft. De optie `-c` geeft aan dat de `.c` alleen gecompileerd (en nog niet gelinkt) moet worden. Het laatste commando zorgt ervoor dat de twee `.o` bestanden gelinkt worden tot één executable. Met de optie `-o demo` bepaal je de naam van deze executable.

3.1.4 Bestudeer de code in het bestand `demo.c`. Je kunt dit bestand openen in de editor van Visual Studio Code door in het terminal window het volgende commando uit te voeren:

```
code demo.c
```

Er worden in dit programma twee *pthreads*[17] aangemaakt: `producer` en `consumer`. De `producer` plaatst de integers `0` tot en met `9` één voor één in de buffer met een tussenpoos van 1000 ms. De `consumer` leest elke 500 ms alle integers uit de buffer en print ze op het scherm. Omdat de buffer gebruikt wordt door twee threads moet deze gemeenschappelijke resource beschermd worden met een mutex[18]. Het gebruik van de mutex zorgt ervoor dat de twee threads niet gelijktijdig aanpassingen in de buffer kunnen maken. Elke thread *locked* de mutex alvorens de buffer te gebruiken en *unlocked* de mutex weer, als hij klaar is met het gebruik van de buffer, zodat een andere thread de buffer kan gebruiken. Vraag eventueel extra uitleg aan de docent als je iets niet begrijpt.

[16] Ook wel C17 genoemd, zie eventueel https://en.wikipedia.org/wiki/C17_(C_standard_revision).

[17] Als je bent vergeten wat pthreads zijn en hoe je die gebruikt, bekijk dan deze kennisclips en lab 1 van week 4 van EMS20 opnieuw.

[18] Zie: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/.

Voer nu het programma `demo` nu uit door in een terminal het volgende commando te gebruiken:

```
./demo
```

De punt (`.`) in het commando geeft aan dat het programma `demo` in het huidige directory staat.

Doordat de code van de buffer in een aparte `.h` en `.c` file is geplaatst los van de applicatie `demo.c` kan deze module ook eenvoudig in een andere applicatie gebruikt worden. Het programma `demo.c` laat de werking van de buffer zien, maar kan natuurlijk niet als een degelijke test van de buffer worden gezien. Om de code van de buffer goed te testen is het programma `test.c` geschreven.

Dit programma maakt gebruik van zogenoemde preprocessor macro's[19] om tests op eenvoudige manier te coderen. Het bestand `test.c` bevat onder andere de volgende code:

```
    TEST(buffer_put(test_value), "value %d can not be written into ↩
 ↪ the buffer", test_value)
```

Deze code voert de expressie `buffer_put(test_value)` uit. Als dit *geen* **`true`** oplevert, dan wordt de tekst `"value %d can not be written into the buffer"` afgedrukt. Waarbij voor `%d` de waarde van de variabele `test_value` wordt ingevuld.

De macro `TEST` is als volgt gedefinieerd (je hoeft deze definitie niet te begrijpen, als je maar snapt hoe je de macro kunt gebruiken):

```
#define TEST(condition, ...) \
    tests++;\
    if (!(condition))\
    {\
        fails++;\
        printf("Error: ");\
        printf(__VA_ARGS__);\
        printf("\n");\
    }
```

Als deze macro wordt gebruikt, dan wordt het als eerste meegegeven argument ingevuld voor `condition` en worden alle overige argumenten ingevuld voor `__VA_ARGS__`.

[19] Zie: https://en.cppreference.com/w/c/preprocessor/replace.

Het volgende gebruik van de macro:

```
    TEST(buffer_put(test_value), "value %d can not be written into ↩
↪ the buffer", test_value)
```

wordt tijdens de eerste fase van het compilatieproces, de zogenoemde preprocessing fase, omgezet naar de volgende C-code:

```
    tests++;\
    if (!(buffer_put(test_value)))\
    {\
        fails++;\
        printf("Error: ");\
        printf("value %d can not be written into the buffer", ↩
↪ test_value);\
        printf("\n");\
    }
```

De \-en aan het einde van elke regel zorgen ervoor dat de 'enter' aan het einde van elke regel wordt overgeslagen zodat de **#define** uit één regel bestaat (want dat moet nu eenmaal). De variabelen `tests` en `fails` zijn twee globale, **static** variabelen die respectievelijk het aantal uitgevoerde en het aantal gefaalde tests bijhouden. Aan het einde van `main` wordt de macro `PRINT_TEST_REPORT` gebruikt, die ervoor zorgt dat het aantal uitgevoerde tests en de resultaten daarvan op het scherm worden afgedrukt.

3.1.5 Bestudeer het programma `test.c` en zorg ervoor dat je begrijpt welke tests uitgevoerd worden en hoe dit gecodeerd is met behulp van de macro `TEST`. Vraag eventueel extra uitleg aan de docent als je iets niet begrijpt. Compileer de bestanden `buffer.c` en `test.c` en link de objectbestanden samen tot één executable genaamd `test` en voer deze uit. Als het goed is, dan is de uitvoer als volgt:

```
15 tests performed: 15 succeded, 0 failed.
```

De gegeven implementatie van de buffer in `buffer.c` is niet erg bruikbaar in de praktijk; er is maar plaats voor één integer in de buffer.

3.1.6 Pas in Visual Studio Code het programma `demo.c` aan zodat de producer elke 300 ms een getal produceert (in plaats van elke 1000 ms). Sla de wijziging op door op ctrl + s

te drukken. Compileer het bestand `demo.c` en link het met `buffer.o` tot de executable `demo`[20].

```
gcc -std=c18 -Wall -Wextra -pedantic-errors -g3 -O0 -c demo.c
gcc buffer.o demo.o -o demo
```

Voer dit programma vervolgens uit:

```
./demo
```

Je zult zien dat niet alle getallen aankomen bij de consumer[21].

Je gaat er later in deze opdracht voor zorgen dat er meerder getallen in de buffer opgeslagen kunnen worden. Maar we gaan nu eerst kijken hoe we de C-bestanden op een eenvoudigere manier kunnen compileren en linken door gebruik te maken van een zogenoemde *'build tool'*[22].

Het is bij grote projecten die uit vele `.c`- en `.h`-files bestaan niet eenvoudig om te onthouden welke files allemaal gewijzigd zijn en dus opnieuw gecompileerd moeten worden. Er is een handige utility genaamd `make` beschikbaar die dit voor je kan doen. In een zogenoemd 'make-file' geef je de afhankelijkheden tussen de verschillende bestanden aan en de commando's die uitgevoerd moeten worden als een bepaald bestand gewijzigd is. Je hoeft dan alleen nog maar het commando `make` in te typen om het project opnieuw te bouwen. Er is een eenvoudige makefile beschikbaar die zowel `demo` als `test` genereert: `makefile`.[23]

3.1.7 Pas in Visual Studio Code het programma `demo.c` aan zodat de producer elke 200 ms een getal produceert (in plaats van elke 300 ms). Gebruik het commando `make` om de executable `demo` te bouwen.

```
make demo
```

Je ziet dat nu alleen het bestand `demo.c` opnieuw gecompileerd wordt en gelinkt wordt met `buffer.o` tot de executable `demo`.

[20] In het terminalwindow kun je met de ↑-toets commando's die je al eerder hebt gebruikt terughalen.

[21] Als het getal 9 niet aankomt bij de consumer, dan moet je het programma stoppen door op ctrl + c te drukken.

[22] Zie eventueel https://en.wikipedia.org/wiki/Build_automation.

[23] Een goede introductie in het gebruik van `make` kun je vinden op: https://www.gnu.org/software/make/manual/html_node/Introduction.html.

Voer dit programma vervolgens uit:

```
./demo
```

Je zult zien dat er nu nog minder getallen aankomen bij de consumer[24]. Voer de volgend commando's uit om de executable `test` aan te maken en uit te voeren en om tot slot alle door `make` aangemaakte bestanden weer te verwijderen:

```
make test
./test
make clean
```

We gaan niet verder in op het gebruik van makefiles omdat we in het vervolg CMake[25] gebruiken om de build te automatiseren. CMake maakt dan de benodigde makefiles voor ons aan. Maar het is dus wel belangrijk om te weten dat makefiles bestaan en dat je ze kunt gebruiken om je projecten te bouwen.

3.1.8 Pas in Visual Studio Code het programma `demo.c` aan zodat de producer weer elke 1000 ms een getal produceert. Voer in de terminal (in het directory `buffer_1`) het volgende commando uit (let op, het commando eindigt met een spatie en een punt[26]):

```
code -r .
```

Met dit commando open je Visual Studio Code in het huidige directory (`buffer_1`). Als het goed is, wordt CMake automatisch gestart en wordt er een directory `build` aangemaakt.

Voer nu in de terminal (ctrl + ‘) de volgende commando's uit:

```
cd build
make
./demo
```

Met het eerste commando ga je naar het directory `build`. Als het directory `build` niet automatisch is aangemaakt, dan geeft dit commando een foutmelding. In dat geval kun je de volgende commando's uitvoeren:

[24] Als het getal 9 niet aankomt bij de consumer, dan moet je het programma stoppen door op ctrl + c te drukken.

[25] Zie https://cmake.org/.

[26] De punt is een afkorting voor de huidige directory.

```
mkdir build
cd build
cmake ..
make
./demo
```

Met het commando `cmake ..` voer je CMake uit om de build te configureren. De twee punten (`..`) in het commando geven aan dat CMake het bestand `CMakeLists.txt` in het bovenliggende directory moet gebruiken om de build te configureren. Met het commando `make` bouw je het project met behulp van de door CMake aangemaakte makefile en met het laatste commando voer je de executable `demo` uit.

Bestudeer de inhoud van het bestand `CMakeLists.txt` waarmee je CMake verteld hoe het project `buffer` gebouwd moet worden. De meeste CMake commando's spreken voor zich. Indien nodig kun je extra informatie vinden in de online documentatie van CMake[27]:

- `cmake_minimum_required`;
- `project`;
- `add_executable`;
- `target_compile_options`;

Vraag eventueel extra uitleg aan de docent als je iets niet begrijpt.

De CMake-plugin in Visual Studio Code heeft een aantal handige commando's die je kunt gebruiken om je projecten te bouwen, uit te voeren en te debuggen. Veel van deze commando's zijn toegankelijk via de statusbalk onderin het scherm, zie figuur 2.

Figuur 2: De statusbalk in Visual Studio Code met de CMake-plugin.

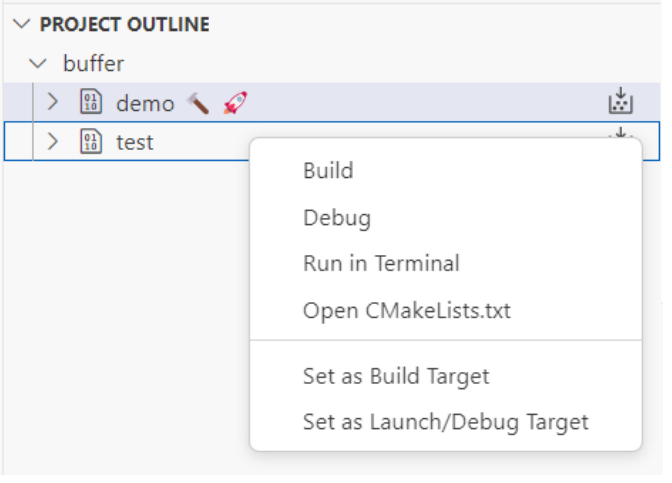
Door op het icoontje bij de 1 te klikken of op F7 te drukken kun je het project bouwen. Als er foutmeldingen of waarschuwingen zijn, dan worden deze getoond bij 2. Als je hierop klikt

[27] CMake is een zeer uitgebreide tool met heel veel opties en mogelijkheden. Wij gebruiken slechts een klein deel van de mogelijkheden van CMake.

of op ctrl + ⇧ + M drukt, dan worden de foutmeldingen en waarschuwingen getoond in het problems-window.

Als je op het icoontje bij 3 klikt of op ⇧ + F5 drukt, dan wordt de executable (in dit geval `demo`) uitgevoerd. Als je op het icoontje bij 4 klikt of op ctrl + F5 drukt, dan wordt de executable gedebugged. Je moet dan wel eerst zelf een breakpoint zetten in de code. Er wordt namelijk niet automatisch een breakpoint gezet op de eerste regel van de main-functie, zoals in Code Composer Studio.

Standaard wordt de eerst target (executable) uit de `CMakeLists.txt` uitgevoerd of gedebugged. Je kunt de andere target (de executable `test`) uitvoeren of debuggen door eerst op het CMake icoontje (in de rechterbalk) te klikken en vervolgens de "PROJECT OUTLINE" te openen en met de rechtermuisknop op de target te klikken die je wilt uitvoeren of debuggen. Zie figuur 3.



Figuur 3: De PROJECT OUTLINE in Visual Studio Code met de CMake-plugin.

In de voorgaande opdrachten heb je gezien dat implementatie van de buffer wel heel eenvoudig is; er is maar plaats voor één integer in de buffer. Je gaat nu het `buffer.c` uitbreiden zodat er meerdere getallen in de buffer opgeslagen kunnen worden. Om het testen eenvoudig te houden, ga je nu een buffer implementeren waar maximaal 8 getallen in opgeslagen kunnen worden.

3.1.9 Gebruik de volgende commando's in de Visual Studio Code WSL terminal om een kopie te maken van het directory `buffer_1`:

```
cd ~
cd Opdrachten
cp -r buffer_1 buffer_8
cd buffer_8
rm makefile
rm -r build
code -r .
```

Met het eerste commando ga je naar je home directory. Met het tweede commando ga je naar het directory `Opdrachten`. Met het volgende commando maak je een kopie van het directory `buffer_1` met de naam `buffer_8`. De `-r` in het commando staat voor 'recursive' en zorgt ervoor dat ook alle subdirectories en bestanden gekopieerd worden. Vervolgens wordt de makefile verwijderd omdat deze niet meer nodig we gebruiken immers CMake om de makefiles te genereren. Daarna wordt het directory `build` verwijderd omdat we een nieuwe directory `build` gaan aanmaken. Tot slot wordt Visual Studio Code geopend in het directory `buffer_8`. De `-r` in het commando staat voor 'reuse-window' en zorgt ervoor dat het directory in het huidige hoofdwindow van Visual Studio Code wordt geopend. Als het goed is wordt het CMake automatisch gestart en wordt het build directory aangemaakt. Je kunt nu de executables `demo` en `test` bouwen met behulp van de statusbalk onderin het scherm, zie figuur 2. Als je op de run of debug knop drukt, dan wordt boven in het scherm gevraagd welk executable uitgevoerd of gedebugged moet worden. Kies voor `demo`. Als je later de executable `test` wilt uitvoeren of debuggen, klik dan op het CMake icoontje, open vervolgens de "PROJECT OUTLINE" en klik met de rechtermuisknop op `test`.

Pas de code in het bestand `demo.h` aan zodat de producer elke 350 ms een getal produceert (in plaats van elke 1000 ms). Als je het programma `demo` nu uitvoert zie je dat er getallen verloren gaan[28].

Pas nu de code in het bestand `buffer.c` aan zodat er maximaal 8 getallen in de buffer opgeslagen kunnen worden. De buffer moet werken als een FIFO-buffer (First In First Out). Maak daarbij gebruik van de volgende array om de getallen in op te slaan:

```
static int buffer[8];
```

[28] Als het getal 9 niet aankomt bij de consumer, dan moet je het programma stoppen door op ctrl + c te drukken.

Er is hier gekozen voor 8 omdat het programma dan eenvoudig te testen is. Maar in praktijk heb je vaak een groter buffer nodig. Zorg er dus voor dat dit aantal eenvoudig aan te passen is door één **#define** in het `buffer.c` bestand te wijzigen. 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` niet gewijzigd hoeft te worden.

Als de buffer correct werkt en 8 getallen kan opslaan dan zullen er geen getallen meer verloren gaan als je het programma `demo` uitvoert.

Als het je niet zelf kunt bedenken hoe je een FIFO-buffer kunt implementeren dan kun je gebruik maken van deze tips[29].

3.1.10

Als je nu de applicatie `test` uitvoert, dan zul je zien dat alle tests nog steeds slagen.

Breid de testcode in het bestand `test.c` uit zodat ook getest wordt dat er meerdere getallen (maximaal 8) in de buffer opgeslagen kunnen worden.

Voeg, als voorbeeld, de testfunctie `test_FIFO_behavior` toe:

```
void test_FIFO_behavior(void)
{
    // write 1, 2 and 3 into the buffer
    TEST(buffer_put(1), "value 1 can not be written into the ↩
 ↪ buffer")
    TEST(!buffer_is_empty(), "buffer still empty after writing ↩
 ↪ into the buffer")
```

[29] Je kunt het FIFO-gedrag van de buffer implementeren met twee indexen `index_put` en `index_get`. Beide variabelen moeten geïnitialiseerd worden met `0`. De variabele `index_put` houdt de plaats bij waar het volgende getal in de buffer geplaatst moet worden. De variabele `index_get` houdt de plaats bij waar het volgende getal uit de buffer gehaald moet worden. Als er een getal in de buffer geplaatst wordt, dan moet daarna de waarde van `index_put` met één verhoogd worden. Als er een getal uit de buffer gehaald wordt, dan moet daarna de waarde van `index_get` met één verhoogd worden. Als de waarde van `index_put` met één verhoogd wordt, dan moet gecontroleerd worden of de waarde van `index_put` gelijk is aan het maximaal aantal elementen dat in de buffer past. Als dit zo is, dan moet de waarde van `index_put` weer op `0` gezet worden. Hetzelfde geldt voor de waarde van `index_get`. Om bij te houden of de buffer vol is, moet een variabele gebruikt worden die aangeeft hoeveel getallen er in de buffer zitten. Deze variabele moet geïnitialiseerd worden met `0` en moet bijgehouden worden in de functies `buffer_put` en `buffer_get`. Als er een getal in de buffer geplaatst wordt, dan moet de waarde van deze variabele met één verhoogd worden. Als er een getal uit de buffer gehaald wordt, dan moet de waarde van deze variabele met één verlaagd worden. Als de waarde van deze variabele gelijk is aan het maximaal aantal elementen dat in de buffer past, dan is de buffer vol. Als de waarde van deze variabele gelijk is aan `0`, dan is de buffer leeg.

```
    TEST(!buffer_is_full(), "buffer full after writing only ↩
 ↪ one int into the buffer")
    TEST(buffer_put(2), "value 2 can not be written into the ↩
 ↪ buffer")
    TEST(!buffer_is_full(), "buffer full after writing two ↩
 ↪ ints into the buffer")
    TEST(buffer_put(3), "value 3 can not be written into the ↩
 ↪ buffer")
    TEST(!buffer_is_full(), "buffer full after writing three ↩
 ↪ ints into the buffer")
    // test if values can be retrieved in FIFO order from the ↩
 ↪ buffer
    int retrieved_value;
    TEST(buffer_get(&retrieved_value), "retrieving 1 from the ↩
 ↪ buffer failed")
    TEST(retrieved_value == 1, "wrong value (%d) retrieved ↩
 ↪ from the buffer, expected 1", retrieved_value)
    TEST(buffer_get(&retrieved_value), "retrieving 2 from the ↩
 ↪ buffer failed")
    TEST(retrieved_value == 2, "wrong value (%d) retrieved ↩
 ↪ from the buffer, expected 2", retrieved_value)
    TEST(buffer_get(&retrieved_value), "retrieving 3 from the ↩
 ↪ buffer failed")
    TEST(retrieved_value == 3, "wrong value (%d) retrieved ↩
 ↪ from the buffer, expected 3", retrieved_value)
    TEST(buffer_is_empty(), "buffer not empty after writing ↩
 ↪ and retrieving three ints to and from the buffer")
}
```

Voeg ook de volgende regel toe aan de functie `main`:

```
    test_FIFO_behavior();
```

Voer het programma `test` uit en controleer of deze nieuwe tests slagen.

Bedenk zelf wat er nog getest moet worden om zeker te weten dat de FIFO-buffer met maximaal 8 integers correct werkt. Schrijf nu zelf meerdere testfuncties en voeg deze toe aan de functie `main`.

Je zou kunnen denken aan het testen van de volgende senario's:

- Er worden acht integers in de buffer geplaatst. Is de buffer nu vol? Als je de buffer nu uitleest, komen de integers er dan weer in FIFO volgorde uit? Is de buffer nu weer leeg?
- Er worden eerst drie integers in de buffer geplaatst en vervolgens worden er twee uitgehaald, komen deze integers er weer in FIFO volgorde uit? Plaats nog vijf integers in de buffer. Is de buffer nu nog steeds niet vol? Als je de buffer nu helemaal uitleest, komen de zes integers er dan weer in FIFO volgorde uit? Is de buffer nu weer leeg?

De buffer die je nu op de pc hebt ontwikkeld en getest, zal in een van de volgende weken gebruikt worden op een CC3220S LaunchPad. We gaan ervan uit dat je beschikt over dit ontwikkelbord dat je ook al bij EMS20 hebt gebruikt[30]. Om software te compileren voor dit bord gebruiken je dan Code Composer Studio (CCS), waar je ook bij EMS20 al mee hebt gewerkt. Zorg dat je deze software alvast hebt geïnstalleerd op je eigen pc voorafgaande aan de lessen van de volgende weken. Een installatiehandleiding voor CCS vind je op: https://bitbucket.org/HR_ELEKTRO/ems20/wiki/installatie_software.md. Je kunt het installatiebestand voor CCS versie 12.5 downloaden van https://dr-download.ti.com/software-development/ide-configuration-compiler-or-debugger/MD-J1VdearkvK/12.5.0/CCS12.5.0.00007_win64.zip. Daarnaast heb je ook de SimpleLink CC32XX SDK versie 6.10 nodig. Het installatiebestand voor deze SDK kun je downloaden van https://bitbucket.org/HR_ELEKTRO/ems31/downloads/simplelink_cc32xx_sdk_6_10_00_05.exe.

[30] Indien nodig kun je een CC3220S LaunchPad aanschaffen bij de winkel op Academie Plein.

A Veelgebruikte Linux-commando's

Unix/Linux Command Reference

FOSSwire.com

File Commands

ls – directory listing
ls -al – formatted listing with hidden files
cd *dir* - change directory to *dir*
cd – change to home
pwd – show current directory
mkdir *dir* – create a directory *dir*
rm *file* – delete *file*
rm -r *dir* – delete directory *dir*
rm -f *file* – force remove *file*
rm -rf *dir* – force remove directory *dir* *
cp *file1 file2* – copy *file1* to *file2*
cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist
mv *file1 file2* – rename or move *file1* to *file2* if *file2* is an existing directory, moves *file1* into directory *file2*
ln -s *file link* – create symbolic link *link* to *file*
touch *file* – create or update *file*
cat > *file* – places standard input into *file*
more *file* – output the contents of *file*
head *file* – output the first 10 lines of *file*
tail *file* – output the last 10 lines of *file*
tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

Process Management

ps – display your currently active processes
top – display all running processes
kill *pid* – kill process id *pid*
killall *proc* – kill all processes named *proc* *
bg – lists stopped or background jobs; resume a stopped job in the background
fg – brings the most recent job to foreground
fg *n* – brings job *n* to the foreground

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Examples:
chmod 777 – read, write, execute for all
chmod 755 – rwx for owner, rx for group and world
For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*
ssh -p *port user@host* – connect to *host* on port *port* as *user*
ssh-copy-id *user@host* – add your key to *host* for *user* to enable a keyed or passwordless login

Searching

grep *pattern files* – search for *pattern* in *files*
grep -r *pattern dir* – search recursively for *pattern* in *dir*
command* | grep *pattern – search for *pattern* in the output of *command*
locate *file* – find all instances of *file*

System Info

date – show the current date and time
cal – show this month's calendar
uptime – show current uptime
w – display who is online
whoami – who you are logged in as
finger *user* – display information about *user*
uname -a – show kernel information
cat /proc/cpuinfo – cpu information
cat /proc/meminfo – memory information
man *command* – show the manual for *command*
df – show disk usage
du – show directory space usage
free – show memory and swap usage
whereis *app* – show possible locations of *app*
which *app* – show which *app* will be run by default

Compression

tar cf *file.tar files* – create a tar named *file.tar* containing *files*
tar xf *file.tar* – extract the files from *file.tar*
tar czf *file.tar.gz files* – create a tar with Gzip compression
tar xzf *file.tar.gz* – extract a tar using Gzip
tar cjf *file.tar.bz2* – create a tar with Bzip2 compression
tar xjf *file.tar.bz2* – extract a tar using Bzip2
gzip *file* – compresses *file* and renames it to *file.gz*
gzip -d *file.gz* – decompresses *file.gz* back to *file*

Network

ping *host* – ping *host* and output results
whois *domain* – get whois information for *domain*
dig *domain* – get DNS information for *domain*
dig -x *host* – reverse lookup *host*
wget *file* – download *file*
wget -c *file* – continue a stopped download

Installation

Install from source:
./configure
make
make install
dpkg -i *pkg.deb* – install a package (Debian)
rpm -Uvh *pkg.rpm* – install a package (RPM)

Shortcuts

Ctrl+C – halts the current command
Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background
Ctrl+D – log out of current session, similar to **exit**
Ctrl+W – erases one word in the current line
Ctrl+U – erases the whole line
Ctrl+R – type to bring up a recent command
!! - repeats the last command
exit – log out of current session

* use with extreme caution.