

Opdrachten week 1 les 1 – Modules in C

Bij het programmeren van embedded systemen wordt meestal de programmeertaal C gebruikt¹. 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 uitgebreid 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 Python⁴, heeft de programmeertaal C geen directe support voor modulair programmeren⁵. 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 in een embedded toepassing willen gebruiken. De toepassing maakt gebruik van een CC3220S processor van Texas Instruments. Deze software wordt getest op een CC3220S LaunchPad. We gaan ervan uit dat je beschikt over dit ontwikkelbord dat je ook al bij EMS20 hebt gebruikt. Om software te compileren voor dit bord gebruiken we Code Composer Studio, waar je ook bij EMS20 al mee hebt gewerkt.

Om software op de pc te kunnen 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 *integrated development environment*

¹ Bron: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf#page=32.

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/>.

(IDE) met een *graphical user interface* (GUI). Wij adviseren in dat geval om Visual Studio Code⁸ te gebruiken.

Sommige tools die we gaan gebruiken zijn alleen beschikbaar onder Linux. Als je Windows gebruikt, dan kun je gebruik maken van Windows Subsystem for Linux (WSL)⁹

Een installatiehandleiding om deze programma's op je eigen machine te installeren kun je vinden op: https://bitbucket.org/HR_ELEKTRO/ems30/wiki/Software_WSL.md. Als je liever een andere ontwikkelomgeving gebruikt om C code op de pc te kunnen ontwikkelen en testen, zoals bijvoorbeeld Eclipse IDE for C/C++ Developers¹⁰ of CLion¹¹, dan is dat ook prima.

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;
- deze module kunt gebruiken en testen op een CC3220S LaunchPad.

In deze les implementeren we een eenvoudig 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. 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 we de functies en de typedefinities die in de module zijn opgenomen. We noemen dit ook wel de *interface* van de module omdat de 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_
```

⁸ Zie: <https://code.visualstudio.com/>.

⁹ Zie: <https://learn.microsoft.com/en-us/windows/wsl/about>.

¹⁰ Zie: <https://www.eclipse.org/downloads/>.

¹¹ Zie: <https://www.jetbrains.com/clion/>.

```
#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 ziet bevat het bestand `buffer.h` slechts de *declaraties* van de functies waarmee de buffer gebruikt kan worden. Deze functiedeclaraties worden ook wel *prototypes* genoemd¹². Het (gewenste) gedrag van elke functie is beschreven in het, aan het prototype voorafgaande, commentaar. De prototypes worden voorafgegaan door het keyword `extern`. Dit is een zogenaemde *storage-class specifier*¹³ 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.

¹² Zie eventueel: https://en.cppreference.com/w/c/language/function_declaration.

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

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 gedeelde variabelen worden voorafgegaan door het keyword `static`. Dit is een zogenoemde *storage-class specifier*¹⁴ 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.

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

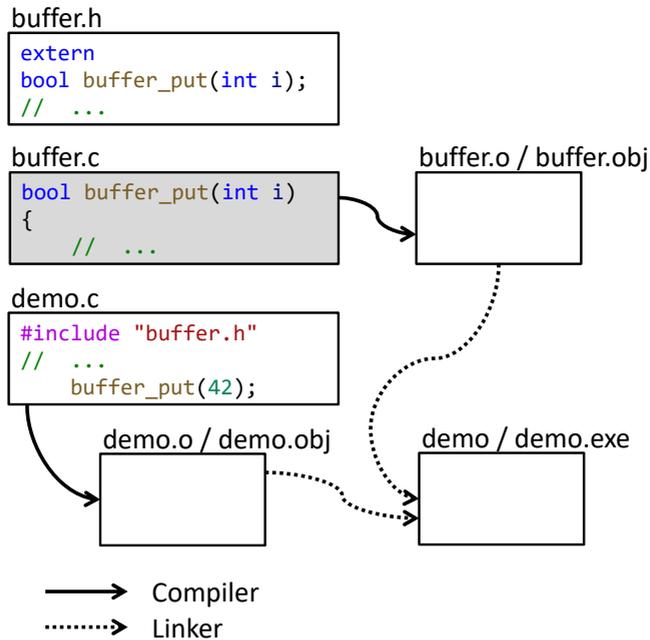
```
#include "buffer.h"

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

¹⁴ Zie eventueel: https://en.cppreference.com/w/c/language/storage_duration.

operating systeem meestal geen extentie. 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.

1.1.1 Download het bestand `buffer_1.zip` en kopieer dit naar een geschikte locatie (vermijd spaties in het pad), bijvoorbeeld `\\wsl.localhost\EMS30\home\ems\Opdrachten` in WSL onder Windows of `/home/student/ems30` in Linux. Voer vervolgens onderstaande commando's uit in een terminal¹⁵ in WSL m.b.v. Visual Studio Code. Hiermee installeer je eerst het unzip commando¹⁶, vervolgens pak je het zip bestand uit en compileer je de bestanden `buffer.c` en `demo.c`. Tot slot link je de objectbestanden samen tot één executable.

```
sudo pacman -Sy unzip
unzip buffer_1.zip
cd buffer_1
gcc -std=c17 -Wall -Wextra -pedantic-errors -g3 -O0 -c buffer.c
gcc -std=c17 -Wall -Wextra -pedantic-errors -g3 -O0 -c demo.c
gcc buffer.o demo.o -o demo
```

Met het eerste commando installeer je het unzip commando. Met het tweede commando pak je het zip-bestand uit. Met het derde commando ga je naar het directory waar je de zip-file hebt uitgepakt. Met de volgende twee commando's worden `buffer.c` en `demo.c` gecompileerd naar respectievelijk `buffer.o` en `demo.o`. Met de optie `-std=c17` geef je aan dat je de C17-standaard 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 C17-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.

¹⁵ 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>.

¹⁶ Het “[sudo] password for ems” is : ems.

1.1.2 Bestudeer de code in het bestand `demo.c`. Er worden in dit programma twee *threads*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 mutex¹⁸. 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.

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

```
./demo
```

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's¹⁹ 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:

```
#define TEST(condition, ...) \
    tests++; \
    if (!(condition)) \
```

¹⁷ Als je bent vergeten wat *threads* 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/>.

¹⁹ Zie: <https://en.cppreference.com/w/c/preprocessor/replace>.

```
{\
    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__`.

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.

1.1.3 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`. Kun je nog meer tests bedenken om de buffer nog beter te testen? Voeg deze tests dan toe. Compileer de bestanden `buffer.c` en `test.c` en link de objectbestanden samen tot één executable en voer deze uit.

Als je geen extra tests hebt toegevoegd is de uitvoer als volgt:

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

De code voor de buffer die ontwikkeld is op een pc kan nu ook gebruikt worden op een embedded systeem. Als voorbeeld is een applicatie voor de CC3220S LaunchPad beschikbaar genaamd `temperatuur_logger`. Deze applicatie bevat onder andere de volgende twee pthreads:

- een producer die elke seconde de temperatuur (uitgedrukt in tiende graden Celsius) inleest en die wegschrijft in de buffer;
- een consumer die wacht op een binnenkomend UDP-pakket. Als dit pakket de C-string `"temp?"` bevat, dan wordt een temperatuur uit de buffer gelezen en wordt deze waarde (uitgedrukt in tiende graden Celsius), gecodeerd als een C-string, teruggestuurd via UDP naar de afzender. Als op het moment dat dit UDP-pakket binnenkomt geen waarde in de buffer aanwezig is dan wordt een halve seconde gewacht en wordt opnieuw gekeken of er al een waarde in de buffer aanwezig is. Dit wordt net zo lang herhaald totdat een waarde uit de buffer is gelezen. Als het ontvangen UDP-pakket niet de C-string `"temp?"` bevat, dan wordt de C-string `"?"`, teruggestuurd via UDP naar de afzender.

Om UDP-berichten te kunnen versturen van de CC3220S LaunchPad naar een pc is een wifi-router nodig waarmee beide systemen kunnen verbinden. De UDP-pakketten kunnen op de pc ontvangen en zichtbaar gemaakt worden met het programma `ontvang_temperaturen.py`.

1.1.4 Importeer voor Code Composer Studio 12.x met simplelink SDK 6.10 het project `temperatuur_logger_6_10.zip`²⁰ en vul de ontbrekende informatie in, in `main.c`:

- naam van de wifi-gateway;
- wachtwoord van de wifi-gateway;

Als je het project genaamd `tirtos_builds_cc32xx_release_ccs` nog niet in jouw workspace hebt geïmporteerd, dan moet je dit project importeren vanuit de directory `...\\ti\\simplelink_cc32xx_sdk\\...\\kernel\\tirtos\\builds\\cc32xx\\release\\ccs`. Pas nu de *Properties* van het project `temperatuur_logger` als volgt aan: ga naar 

²⁰ Als je nog gebruik maakt van Code Composer Studio 11.x met simplelink SDK 5.30 dan kun je het project `temperatuur_logger_5_30.zip` gebruiken.

Dependencies, verwijder het *unresolved* project en voeg nu het geïmporteerde project toe door op Add.. te drukken en `tirtos_builds_cc32xx_release_ccs` te selecteren.

Open het programma `ontvang_temperaturen.py` in Thonny en vul het IP-adres van de CC3220S LaunchPad in. Zorg dat de pc verbonden is met hetzelfde LAN als de CC3220S LaunchPad en start het Python-programma op de pc. Compileer en start nu het project `temperatuur_logger` op de CC3220S LaunchPad. Wat gebeurt er als het Python-programma gestopt en weer opnieuw opgestart wordt? Gaat er data verloren?

Als je de implementatie van de buffer in `buffer.c` bekijkt, zul je zien dat deze implementatie wel heel eenvoudig is; er is maar plaats voor één integer in de buffer.

1.1.5 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`²¹.

```
gcc -std=c17 -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 consumer²².

In de embedded applicatie gaat er eveneens data verloren als de wifi-verbinding wordt onderbroken. Je gaat er later in deze opdracht voor zorgen dat er meerder getallen in de buffer opgeslagen kunnen worden.

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 ‘makefile’ 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

²¹ In het terminalwindow kun je met de `↑`-toets commando’s die je al eerder hebt gebruikt terughalen.

²² Als het getal 9 niet aankomt bij de consumer, dan moet je het programma stoppen door op `ctrl`+`c` te drukken.

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

1.1.6 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`. Voer dit programma vervolgens uit:

```
./demo
```

Je zult zien dat er nu nog minder getallen aankomen bij de consumer²⁴.

We gaan niet verder in op het gebruik van `makefiles` omdat we in het vervolg `CMake`²⁵ gebruiken om de `build` te automatiseren. Maar het is wel belangrijk om te weten dat `makefiles` bestaan en dat je ze kunt gebruiken om je projecten te bouwen.

1.1.7 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`) de volgende commando's uit:

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

Met het eerste commando start je een nieuwe Visual Studio Code instantie in het directory `buffer_1`. Met het tweede commando ga je naar het directory `build` en met het derde commando voer je `CMake` uit om de `build` te configureren. `CMake` gebruikt het bestand `CMakeLists.txt` om de `makefile` te genereren. Met het vierde commando

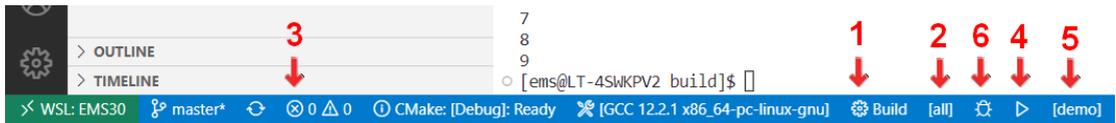
²³ Een goede introductie in het gebruik van `make` kun je vinden op: https://www.gnu.org/software/make/manual/html_node/Introduction.html.

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/>.

bouw je het project met behulp van de door CMake aangemaakte makefile en met het laatste commando voer je de executable demo uit.

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. Standaard worden alle targets (executables) gebouwd, maar je kunt ook een specifieke target selecteren door op `[all]` bij de 2 te klikken. Als er foutmeldingen of waarschuwingen zijn, dan worden deze getoond bij 3. Als je hierop klikt of op `Ctrl + Shift + M` drukt, dan worden de foutmeldingen en waarschuwingen getoond in het problems-window.

Als je op het icoontje bij 4 klikt of op `Shift + F5` drukt, dan wordt de executable genoemd bij 5 uitgevoerd. Door op 5 te klikken kun je een andere executable selecteren. Als je op het icoontje bij 6 klikt of op `Ctrl + F5` drukt, dan wordt de executable genoemd bij 5 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.

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.

1.1.8 Kopieer het directory `buffer_1` naar een nieuwe directory `buffer_8`. Verwijder het directory `build` uit het nieuwe directory `buffer_8`. Open dit directory in Visual Studio Code. Pas de testcode in het bestand `test.c` aan zodat getest wordt dat er meerdere getallen (maximaal 8) in de buffer opgeslagen kunnen worden. De buffer moet werken als een FIFO-buffer (First In First Out). Deze tests zullen (uiteeraard) falen, als je de huidige implementatie van `buffer.c` gebruikt.

1.1.9 Pas nu de code in het bestand `buffer.c` aan zodat er maximaal 8 getallen in de buffer opgeslagen kunnen worden. Maak daarbij gebruik van de volgende array om de getallen in op te slaan:

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

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 regel 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` en in het project `temperatuur_logger` niet gewijzigd hoeft te worden.

Als het je niet zelf kunt bedenken hoe je een FIFO-buffer kunt implementeren dan kun je gebruik maken van deze tips²⁶.

²⁶ Je kunt het FIFO-gedrag van de buffer implementeren met twee indexen `index_put` en `index_get`. Beide variabelen moeten geïnitieerd 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 de waarde van `index_put` met één verhoogd worden. Als er een getal uit de buffer gehaald wordt, dan moet 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 geïnitieerd 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.

A Veelgebruikte Linux-commando's

Unix/Linux Command Reference

FOSSwire.com

File Commands	System Info
ls - directory listing	date - show the current date and time
ls -al - formatted listing with hidden files	cal - show this month's calendar
cd dir - change directory to <i>dir</i>	uptime - show current uptime
cd - change to home	w - display who is online
pwd - show current directory	whoami - who you are logged in as
mkdir dir - create a directory <i>dir</i>	finger user - display information about <i>user</i>
rm file - delete <i>file</i>	uname -a - show kernel information
rm -r dir - delete directory <i>dir</i>	cat /proc/cpuinfo - cpu information
rm -f file - force remove <i>file</i>	cat /proc/meminfo - memory information
rm -rf dir - force remove directory <i>dir</i> *	man command - show the manual for <i>command</i>
cp file1 file2 - copy <i>file1</i> to <i>file2</i>	df - show disk usage
cp -r dir1 dir2 - copy <i>dir1</i> to <i>dir2</i> ; create <i>dir2</i> if it doesn't exist	du - show directory space usage
mv file1 file2 - rename or move <i>file1</i> to <i>file2</i> if <i>file2</i> is an existing directory, moves <i>file1</i> into directory <i>file2</i>	free - show memory and swap usage
ln -s file link - create symbolic link <i>link</i> to <i>file</i>	whereis app - show possible locations of <i>app</i>
touch file - create or update <i>file</i>	which app - show which <i>app</i> will be run by default
cat > file - places standard input into <i>file</i>	Compression
more file - output the contents of <i>file</i>	tar cf file.tar files - create a tar named <i>file.tar</i> containing <i>files</i>
head file - output the first 10 lines of <i>file</i>	tar xf file.tar - extract the files from <i>file.tar</i>
tail file - output the last 10 lines of <i>file</i>	tar czf file.tar.gz files - create a tar with Gzip compression
tail -f file - output the contents of <i>file</i> as it grows, starting with the last 10 lines	tar xzf file.tar.gz - extract a tar using Gzip
Process Management	tar cjf file.tar.bz2 - create a tar with Bzip2 compression
ps - display your currently active processes	tar xjf file.tar.bz2 - extract a tar using Bzip2
top - display all running processes	gzip file - compresses <i>file</i> and renames it to <i>file.gz</i>
kill pid - kill process id <i>pid</i>	gzip -d file.gz - decompresses <i>file.gz</i> back to <i>file</i>
killall proc - kill all processes named <i>proc</i> *	Network
bg - lists stopped or background jobs; resume a stopped job in the background	ping host - ping <i>host</i> and output results
fg - brings the most recent job to foreground	whois domain - get whois information for <i>domain</i>
fg n - brings job <i>n</i> to the foreground	dig domain - get DNS information for <i>domain</i>
File Permissions	dig -x host - reverse lookup <i>host</i>
chmod octal file - change the permissions of <i>file</i> to <i>octal</i> , which can be found separately for user, group, and world by adding:	wget file - download <i>file</i>
<ul style="list-style-type: none"> • 4 - read (r) • 2 - write (w) • 1 - execute (x) 	wget -c file - continue a stopped download
Examples:	Installation
chmod 777 - read, write, execute for all	Install from source:
chmod 755 - rwx for owner, rx for group and world	./configure
For more options, see man chmod .	make
SSH	make install
ssh user@host - connect to <i>host</i> as <i>user</i>	dpkg -i pkg.deb - install a package (Debian)
ssh -p port user@host - connect to <i>host</i> on port <i>port</i> as <i>user</i>	rpm -Uvh pkg.rpm - install a package (RPM)
ssh-copy-id user@host - add your key to <i>host</i> for <i>user</i> to enable a keyed or passwordless login	Shortcuts
Searching	Ctrl+C - halts the current command
grep pattern files - search for <i>pattern</i> in <i>files</i>	Ctrl+Z - stops the current command, resume with fg in the foreground or bg in the background
grep -r pattern dir - search recursively for <i>pattern</i> in <i>dir</i>	Ctrl+D - log out of current session, similar to exit
command grep pattern - search for <i>pattern</i> in the output of <i>command</i>	Ctrl+W - erases one word in the current line
locate file - find all instances of <i>file</i>	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.

