CODE-REVIEW IOT-SOFTWARE_

Gebaseerd op “The Power of 10”

Versie: 1
Klas: INTOFT21-P3/11
Docent: Chris van Uffelen
Course: IoT-Software

Mike Ackerschott - 1652342

11 april 2025

HAN_UNIVERSITY
OF APPLIED SCIENCES

INHOUDSOPGAVE

INLEIDING

In dit document wordt mijn uitwerking van het beroepsproduct (opgesteld in hoofdstuk 7 van de reader van IoT-software) beoordeeld aan de hand van de code-stijlvoorschriften opgesteld door Gerard Holzmann in het document: “The power of 10: Rules for Developing Safety-Critical Code”.
Dit document geeft met behulp van 10 regels aan hoe je er voor kan zorgen dat C code functioneert zoals verwacht en hoe snel gemaakte fouten voorkomen kunnen worden.

1 REGELS

Regel 1: Beperk je code tot het gebruiken van simpele controle-flows

Deze regel benadrukt dat de “flow” van het programma simpel en logisch moet zijn. Het gebruiken van statements zoals *goto*, *setjmp* of *longjmp* maakt je code gevoelig voor recursie of onverwacht gedrag dat moeilijk is om te debuggen. Verder maakt het gebruik van dit soort statements de code ook minder leesbaar.
De gemaakte C code in dit beroepsproduct maakt hier ook geen gebruik van en ook recursie wordt niet toegepast in mijn code. Deze regel wordt dus goed aangehouden

Regel 2: Geef alle loops een vaste limiet in hoeveelheid iteraties

Deze regel benadrukt het limiteren van het aantal iteraties wat mogelijk is in een for of while loop. Hierdoor is er in het programma nooit de mogelijkheid dat een loop voor altijd (of langer dan verwacht) doorgaat waardoor het programma vastloopt. Deze regel aanhouden voorkomt dus onverwachtse recursie.
Mijn code voldoet hier niet aan. Het gebruiken van de buffers in mijn code gaat aan de hand van for loops gebaseerd op het aantal waarden in de circulaire buffer of een meegegeven size. Deze waarden zijn dus variabel en kunnen gedurende het programma waardes verkrijgen dat de snelheid van het programma beïnvloed, aangezien er geen upper limit aanwezig is in het programma. Echter is dit ook by-design. Als een van de eisen is dat het programma een gemiddelde over de gehele circulaire buffer moet kunnen berekenen, moet de upper-limit variabel zijn om deze werking te kunnen ondersteunen. Hierdoor zorgt het niet aanhouden van deze regel in dit beroepsproduct niet tot een lagere kwaliteit van code.

Regel 3: Gebruik geen dynamic memory allocation na initialisatie

Deze regel benadrukt het gebruik van malloc() en free() om na initialisatie van het programma nog varaiebelen te initialiseren gedurende runtime van het programma. Hierdoor is het mogelijk dat toegewezen RAM-geheugen voor een gemallocde variabele niet meer wordt teruggegeven nadat de variabele niet meer nodig is, waardoor je de mogelijkheid hebt om uit RAM-geheugen te raken (ook wel een memory-leak genoemd).

Mijn code voldoet hier ook niet aan. Voor het resizen van de circulaire buffer wordt gebruik gemaakt van malloc om een nieuwe array aan te maken, waarna de data wordt overgezet naar de nieuwe array en de oude array wordt opgeruimd aan de hand van free(). Ook dit is by-design. Een mogelijke oplossing is het initialiseren van een circulaire buffer met een groot aantal indexes die er maar een aantal gebruikt gebaseerd op de geconfigureerde size. Dit zorgt er wel voor dat het RAM-geheugen van het programma aanzienlijk verhoogt wordt terwijl dit nooit altijd gebruikt wordt. Ook wordt de success van de malloc call gecheckt en gaan er visueel ledjes branden wanneer dit faalt, zodat het voor een gebruiker gelijk duidelijk is dat het programma zich nu onverwachts kan gedragen (zoals voorgeschreven in de opdracht). Hierdoor denk ik dat in dit geval het gebruiken van malloc en free geen negatieve impact zal hebben op het programma en de kwaliteit ervan.

Regel 4: Functies moeten niet langer zijn dan wat past op een pagina (max. 60 regels)

Deze regel benadrukt het maken van code en functies waarvan de inhoud overzichtelijk is en makkelijk te begrijpen is. Wanneer een functie meer dan 60 regels omvat, gebeurt er vaak logica wat overzichtelijker zou zijn als het opgedeeld was in een subset van functies.
In mijn code is er geen functie die langer is dan 60 regels, maar de functie handleRequest() komt dichtbij. Deze functie is inclusief comments 60 regels lang en dus een grensgevalletje. De comments geven gelukkig wel verduidelijking in de logica die uit wordt gevoerd in deze functie, maar het zou beter zijn geweest om de logica die de code-comments beschrijven op te splitsen in meerdere functies, zoals:

- readBuffer() – leest de volledige HTTP- request
- getContentLength() – parsed de content length van de HTTP-request
- getBody() – parsed de body van de HTTP-request als deze bestaat
- checkAllowedEndpoints() – checkt of de endpoint bestaat, zo niet retourneert een 404
- handleCallback() – checkt of de gebruikte methode op de endpoint toegestaan is. Zo niet retourneert een 404. Zo wel roept het de handler functie aan.

De huidige implementatie voldoet dus wel, maar de bovengenoemde implementatie had de codekwaliteit en leesbaarheid aanzienlijk verbeterd.

Regel 5: Elke functie moet minimaal twee assert-calls beschikken

Deze regel benadrukt dat het programma moet checken op condities (aan de hand van de C macro assert) die niet mogelijk zijn in een “real life scenario”, zodat het programma hierop correct kan reageren door bijvoorbeeld een error-code te retourneren.
Mijn code gebruikt alleen geen asserts. Hierdoor voldoe ik niet aan deze regel. Wel zijn er functies in mijn code die, zoals de regel voorschrijft, checkt of de code werkt zoals verwacht aan de hand van booleaanse expressies. Voorbeelden zijn:

- resize_buffer() – return false wanneer de size gelijk is aan 0 of wanneer de malloc faalt

- finalize_aggregate – return -1 voor alle opgevraagde data wanneer er nog geen meting is geweest. Returnt 0 voor sample variance en standaard deviatie wanneer er 1 meting is geweest.

Persoonlijk denk ik niet dat de codekwaliteit aanzienlijk wordt verbeterd als elke functie in mijn code gebruikt gaat maken van minimaal twee asserts. Complexere logica wordt al afgehandeld met correcte return statements, zodat het programma hierop kan reageren en wanneer het systeem faalt, door bijvoorbeeld een te grote circulaire buffer, geeft het systeem dit visueel aan met behulp van de ledjes.

Regel 6: Declareer alle data-objecten met een zo'n klein mogelijke scope

Deze regel benadrukt het principe van data-hiding. Als een variabele maar in één functie nodig is, is het slim om hem ook pas hier aan te maken. Hierdoor is het onmogelijk dat andere functies of klasses het object op een ongewenste manier kunnen manipuleren en is de kans op onverwachts gedrag klein.
In mijn code gaat dit redelijk goed. In arduino_server.ino worden globale variabelen gebruikt die nodig zijn voor timers. Verder worden de benodigde objecten voor de ethernet-connectie en de circulaire buffers globaal gedeclareerd, maar dit is nodig omdat de data en connectie altijd aanwezig moet zijn om het programma correct te laten draaien, dus in principe is dit de kleinst mogelijke scope.
Verder bevat cserver.h nog een aantal globale variabelen die data bijhouden en de status van het systeem bijhouden. Deze variabelen worden ook op verschillende plekken in het systeem gebruikt en heeft dus nut om globaal te zijn.

Regel 7: Elke aanroepende functie moet de returnwaarde van niet-void functies controleren, en elke aangeroepen functie moet de geldigheid van alle door de aanroeper aangeleverde parameters controleren

Deze regel benadrukt het principe van het valideren van input en output van de geschreven functies. In mijn code hebben de meeste functies hebben al enige validatie, maar er zijn verbeteringen nodig om volledig te voldoen aan deze regel:

- in init_cserver wordt de boolean returnwaarde van de init_buffer() calls niet gecheckt
- in resize_buffer wordt de parameter buffer niet gecheckt of het gelijk is aan NULL
- in welford_online.c wordt voor beide functies niet gecheckt of de meegegeven aggregate gelijk is aan NULL
- in buffermock.c wordt voor alle functies niet gecheckt of de meegegeven aggregate gelijk is aan NULL

Deze checks wel uitvoeren zal ervoor zorgen dat het systeem minder snel onverwacht gedrag zal vertonen.

Regel 8: Het gebruik van de preprocessor moet gelimiteerd worden aan het includen van header files en simpele macro-definities

Deze regel benadrukt het houden van simpliciteit met het gebruik van de preprocessor en de complexiteit uit te werken in C code. Te veel gebruik van de preprocessor kan ervoor zorgen dat de code moeilijker te lezen en begrijpen is.
Mijn code gebruikt de preprocessor verder alleen voor het includen van header files. Verder gebruik ik ook geen #define, aangezien constante variabelen hiervoor beter zijn vanwege type safety. Wel maak ik gebruik van een wrapper (arduino_wrapper.cpp) zodat ik functionaliteit zoals millis(), Serial en EthernetClient kan gebruiken binnen mijn C code aan de hand van de extern C macro. Hierdoor heb ik twijfels over of ik me volledig aan deze regel houd, maar persoonlijk vond ik deze oplossing heel fijn werken.

Regel 9: Het gebruik van pointers moet gelimiteerd zijn. Functie pointers zijn niet toegestaan

Deze regel benadrukt hoe pointers moeilijk zijn om te beheren en hoe gevoelig C code kan zijn voor het per ongeluk verkeerd gebruiken van een pointer. Mijn code gebruikt hier en daar wel wat pointers. Onder andere voor het initialiseren en updaten van de circulaire buffers en de welford aggregates worden pointers gebruikt om de daadwerkelijke data op te slaan in de meegegeven buffer, zodat er geen kopie gemaakt wordt en de data na de functie-scope verloren gaat. Dit valt nog onder “gelimiteerd” naar mijn mening, aangezien dit basic practice is in veel Arduino libraries.
Ook gebruik ik functie pointers in cserver.c:268 om een handler functie te paren met een methode-endpoint call, wat volgens deze regel niet een goed idee is.

> *“Similarly, function pointers should be used only if there is a very strong justification for doing so because they can seriously restrict the types of automated checks that code checkers can perform” (Holzmann, 2006)*

Persoonlijk vind ik dit ook een goede reden voor het gebruiken van functiepointers. In principe maak ik gebruik van een hash-map voor het correct paren van een methode-endpoint call en een handle-functie. Ook worden de pointers verder niet aangepast, dus is er een vrij grote zekerheid dat de pointer naar de juiste functie wijst. Hierdoor ben ik het niet volledig eens met deze regel.

Regel 10: Compileer code met warning ingeschakeld en los alle warnings op alsof ze errors zijn

Gedurende het ontwikkelproces van dit beroepsproduct is in het begin geen rekening gehouden met warnings of statische code checkers. Pas halverwege kwam ik erachter dat dit een vereiste was en heb ik de compiler flag -Wall aangezet voor het arduino-compilatieproces. Dit heb ik gedaan door het

bestand “programmers.txt” aan te passen en hierbij de key `compiler.warning_flags.default=` te zetten naar `-Wall`. Dit bestand is te vinden in de directory:

```
C:\Users\YOUR_USERNAME\AppData\Local\Arduino15\packages\arduino\hardware\avr\1.8.6
```

De code compileert aan de hand van deze configuratie zonder warnings of errors.
Ook ben ik gaan kijken met cppcheck om bad practice en codesmells op te lossen. Aan de hand van het volgende commando heb ik dat gedaan:

```
$ cppcheck --enable=all --inconclusive --quiet --language=c *.c *.h *.ino *.cpp
```

De output van dit commando geeft in mijn oplossing enkel commentaar over code vanuit de token.c/.h en stream.c/.h bestanden. Verder geeft het ook aan dat de functies loop() en setup() binnen de arduino_server.ino nooit gebruikt worden. Dit is deel waar, aangezien ik ze nooit aanroep, maar dit gebeurt natuurlijk intern in de Arduino software.

2 CONCLUSIE

Aan de hand van de “Power of 10”-richtlijnen van Gerard Holzmann heb ik mijn C-code kritisch geëvalueerd op het gebied van veiligheid, leesbaarheid en onderhoudbaarheid. Over het algemeen voldoet het beroepsproduct aan een groot aantal van de gestelde regels. Waar dat niet het geval is, wordt duidelijk uitgelegd hoe dat tot stand is gekomen en wat de invloed hiervan is. Ook worden verbeterpunten duidelijk aangegeven en worden mogelijke oplossingen gegeven om alsnog te komen tot een goede oplossing.

3 BRONNEN

- Holzmann, G. (2006). The Power of 10: Rules for Developing Safety-Critical Code. In *ResearchGate*. Geraadpleegd op 8 april 2025, van https://www.researchgate.net/publication/220477862_The_Power_of_10_Rules_for_Developing_Safety-Critical_Code

OPEN UP
NEW HAN_UNIVERSITY OF APPLIED SCIENCES
HORIZONS.