AUFGABE 3: TRIPLE MODULAR REDUNDANCY

In dieser Aufgabe werden Sie Ihren in Aufgabe 2 implementierten Filteralgorithmus **dreifach redundant** ausführen, um Hardwarefehler zu maskieren und erste Erfahrungen mit TMR zu sammeln.

Die Vorgabe befindet sich im Ordner `03_TMR` des Vorgaben-Repositories:

```
git@gitlab.cs.fau.de:ezs/vezs20-vorgabe.git
```

Starten Sie die Anwendung mit `make run` im Build-Verzeichnis, nachdem Sie mittels

```
source ./ecosenv.sh && mkdir build && cd build && cmake ..
```

dieses erstellt haben.

Ziel der Aufgabe ist es ein redundantes Filtersystem zu entwerfen. Dafür stellen wir Ihnen drei virtuelle Sensoren zu Verfügung, die (redundant) jeweils dieselbe Messgröße erfassen.

1 Aufgabenstellung

Vermerken Sie Ihre Antworten zu den Fragen der einzelnen Aufgaben an den vorgesehenen Stellen in der vorgegebenen `answers.md`*. Bitte erstellen Sie, um die Abgabe durch Mergerequests zu vereinfachen,* ***pro Aufgabe einen eigenen Branch****. Um einen konsistenten Zustand zu gewährleisten, benutzen sie dazu bitte folgenden Befehl:*
`git fetch git@gitlab.cs.fau.de:ezs/vezs20-vorgabe.git aufgabe3 && git checkout -b aufgabe3 FETCH_HEAD`

1.1 Konzeption

Aufgabe 1 *Fehlerhypothese*

Bevor Sie anfangen die Filterimplementierung gegen transiente Fehler zu schützen, benötigen Sie eine Fehlerhypothese. Gehen Sie für diese Aufgabe von einer trivialen, *nicht redundanten* Signalverarbeitungskette aus: Sensorik → Verarbeitung → Aktorik. *An welchen Stellen können Fehler auftreten, wenn man Speicherfehler/Registerfehler annimmt? Welche Folgen können diese Fehler beispielsweise haben? Welche Arten an Auswirkungen kann der Defekt verursachen?* Zählen Sie drei mögliche Folgen für das System auf.

Antwort:

Aufgabe 2 *Grundstruktur*

Wir haben für Sie in der Vorgabe bereits für jeden Teil der Verarbeitungskette (*Eingabedaten erfassen*, *Filtern*, *Ausgabedaten bestimmen*) Einstiegspunkte für die dazugehörigen eCos-Fäden angelegt. Erstellen Sie nun diese Fäden[1], um mit Hilfe des vorgegebenen Alarms eine periodische Ausführung Ihrer Signalverarbeitungskette zu ermöglichen. *Worauf müssen Sie bei der Implementierung der Rangfolge zwischen den einzelnen Fäden achten, wenn man annimmt, dass Fehler bei der Ausführung der einzelnen Fäden auftreten können?* Konsultieren Sie bei Fragen bitte zunächst die eCos-Dokumentation[2] bevor Sie Fragen in der Rechnerübung stellen.

☞ cyg_flag_timed_wait()

Antwort:

1.2 Implementierung

Implementieren Sie die Signalverarbeitung mit einer Periode von 10 ms. Der auslösende Alarm wurde schon von uns angelegt.

Aufgabe 3 *Sensorwerte*

Fragen Sie die Messgröße des redundant ausgelegten Sensorsystems ab. Wie bei analogen Sensoren üblich geben die Sensoren eine positive Ganzzahl vom Typ `uint16_t` zurück. Diese Zahl repräsentiert eine Spannung zwischen 0V (`0U`) und 15V (`UINT16_MAX`). Berechnen Sie die anliegende Spannung in Volt und rechnen Sie mit dieser Zahl in Q-Notation weiter. Da hier die Sensorik redundant ausgelegt ist, müssen die Messwerte durch einen geeigneten Vorverarbeitungsschritt

☞ ezs_getValueSensorA()
☞ ezs_getValueSensorB()
☞ ezs_getValueSensorC()

[1] `http://ecos.sourceware.org/docs-latest/ref/kernel-thread-create.html`
[2] `http://ecos.sourceware.org/docs-latest/ref/ecos-ref.html`

zusammengeführt werden. *Warum?* Implementieren Sie eine geeignete, anwendungsfallspezifische Vorverarbeitung.

Antwort:

Aufgabe 4 *Replizierte Filterung*

Filtern Sie die abgefragten Sensorwerte nach den in Vorlesung und Übung vorgestellten Prinzipien **dreifach redundant**. Achten Sie besonders auf den Replikdeterminismus und eine saubere Kapselung der einzelnen Replikate. Nutzen Sie wie in Teilaufgabe 2 vorgesehen einen **eigenen Aktivitätsträger** für jedes Replikat. Kopieren Sie für die Filterung selbst Ihre Filterimplementierung aus Aufgabenblatt 2 an die vorgesehenen Stellen in den Dateien `src/filter.c` und `include/filter.h`. Behalten Sie unsere Schnittstellen bei und passen Sie Ihren Code entsprechend an, falls notwendig.

☞ `perform_processing_step()`

Aufgabe 5 *Ausgangsmaskierung*

Führen Sie nun die Ergebnisse der Filterung in einem Ausgangsvoter zusammen und vergleichen Sie die Ergebnisse der Filterung. Beenden Sie die Ausführung Ihrer Anwendung falls Sie einen Zustand feststellen, von dem sich das System nicht mehr erholen kann. Stellen Sie andernfalls den korrekten Wert wieder her. Achten Sie darauf, dass Sie keine Werte aus der alten Runde verwenden. Zeigen Sie zur einfachen Nachverfolgung durch geeignete Ausgaben an, aus welcher Konsenzmenge sich der endgültige Wert ergibt.

☞ `perform_voting()`

☞ **`printf`**

Um welche Art von Test handelt es sich bei Ihrem Voter? Welche Maßnahmen sollten *normalerweise* beim Erkennen eines Fehlers in *einem* Replikat eingeleitet werden? (Eine Implementierung für Fehlerfälle ist in dieser Teilaufaufgabe noch nicht notwendig.)

Antwort:

Wir haben die Weitergabe des Wertes an die Aktorik mittels des Funktionsstumpfs `ezs_setOutputA()` simuliert. Passen Sie dies ggf. abhängig von Ihrem Voter-Ergebnis an.

Aufgabe 6 *Datenfehler*

Implementieren Sie eine anwendungsspezifische Wiederherstellung für den Fall eines Datenfehlers (kaputter Eingabewert oder Bitkipper an einer intern an der Berechnung beteiligten Variable) im Voter.

Testen Sie diese Implementierung, indem Sie (ausgelöst durch Setzen des Macros `INJECT_DATAFAULT`), in *einem* Replikat Ihrer Wahl einen *transienten*, d.h. u.A. *einmaligen*, Datenfehler am Eingabewert der Filterung produzieren. *Erläutern Sie, auch anhand der Ausgaben Ihres Programmes in diesem Fehlerfall, wie Ihr Code mit diesem Fehler umgeht und einen konsistenten Zustand wiederherstellt.*

Antwort:

Hinweis: Deaktivieren Sie `INJECT_DATAFAULT` nun wieder.

Aufgabe 7 *Zeitüberschreitung*

Testen Sie nun, wie Ihre Implementierung mit Zeitüberschreitungen umgeht, indem Sie (ausgelöst durch Setzen des Macros `INJECT_LOOP`), in *einem* Replikat Ihrer Wahl einen *permanenten* Ausfall der Filterung durch eine Endlosschleife produzieren. *Überprüfen Sie, auch hier wieder gestützt durch Ausgaben Ihres Programms, dass Ihre Implementierung hier globalen Fortschritt garantiert, d.h. durch die korrekten Ergebnisse der beiden intakten Replikate den Aktor zuverlässig ansteuert.*

Wie müsste eine Fehlerwiederherstellung aussehen, die auch mit sich durch Fehlerausbreitung gegebenenfalls als ***permanente Laufzeitüberschreitung manifestierenden, transienten Defekten*** *umgehen könnte?*

Aufgrund der deutlich größeren Implementierungskomplexität ist eine Implementierung hier *nicht* erforderlich, eine prinzipielle Umschreibung des Vorgehens ist ausreichend!

Antwort:

Hinweis: Deaktivieren Sie `INJECT_LOOP` nun wieder.

1.3 Evaluation

Aufgabe 8 *Replikdeterminismus*

Führen Sie das System für eine Minute ohne Fehler aus. Falls Sie Fehler entdecken, spricht dies vermutlich für eine Verletzung des Replikdeterminismus. Finden Sie etwaige solche Verstöße und eliminieren Sie diese.

Aufgabe 9 *TMR Analyse (I)*

Welche neuen möglichen Angriffsflächen für transienten Fehler sind durch die Implementierung von Triple Modular Redundancy hinzugekommen?

Antwort:

Aufgabe 10 *TMR Analyse (II)*

Wie kann die tatsächliche Effektivität Ihrer TMR-Maßnahmen überprüft werden?

Antwort:

Aufgabe 11 *Sphere of Replication*

Bestimmen Sie die durch Redundanz gesicherten Bereiche in Ihrem Programm (→ Sphere of Replication) und markieren Sie diese durch Kommentare in Ihrem

Quellcode. Nutzen Sie hierfür die folgenden Kommentare:
Beginn des Redundanzbereiches: `//< BEGIN_SOR >`
Ende des Redundanzbereiches: `//< END_SOR >`
Sie können diese Kommentare — so nötig — mehrfach verwenden.

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 09.12.2020
- Fragen bitte an `i4ezs@lists.cs.fau.de`