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. In späteren Aufgaben werden wir die Effektivität Ihrer Implementierung mittels Fehlerinjektion testen.

Achten Sie deshalb auf einen sauberen Softwareentwurf um die einzelnen funktionalen Elemente zu modularisieren.

Die Vorgabe dieser Aufgabe befindet sich im Ordner `03_TMR` des Repositories `https://gitlab.cs.fau.de/ezs/vezs16-vorgabe.git`. Das Kompilieren und Starten erfolgt nach dem gewohnten Muster:`cd build`, `source ../ecosenv.sh`, `cmake ..`, `make run`

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

1 Grundlegende Übung

1.1 Vorbereitung

1. Fehlerhypothese: Bevor Sie anfangen die Filterimplementierung gegen transiente Fehler zu schützen, benötigen Sie eine Fehlerhypothese. *An welchen Stellen können Fehler auftreten, wenn man Speicherfehler/Registerfehler annimmt? Welche Folgen können diese Fehler beispielsweise haben?* Zählen Sie mindestens 3 Folgen auf.

Analysieren Sie die Filterimplementierung mit Hilfe der *Fehlerbaummethode* und halten Sie das Analyseergebnis schriftlich fest.

---

2. Grundstruktur: Erstellen Sie ein Taskgerüst (ohne konkrete Implementierung) für die periodische Ausführung (Periode 2 ms) Ihrer Signalverarbeitungskette. Der auslösende Alarm wurde schon von uns angelegt. Legen Sie für alle benötigten Schritte (*Eingabedaten erfassen*, *Filtern*, *Daten ausgeben*) eigene eCos-Fäden[1] an. *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?*

☞ cyg_flag_timed_wait

Konsultieren Sie bei Fragen bitte zunächst die eCos-Dokumentation[2] und erst wenn Sie nicht mehr weiter wissen die Rechnerübungsleiter.

1.2 Implementierung

3. Sensorwerte Fragen Sie die Messgröße vom redundant ausgelegte Sensorsystem ab. Wie bei analogen Sensoren üblich geben die Sensoren eine positive Ganzzahl vom Typ `uint16_t` zurück. Diese Zahl repräsentiert eine Spannung zwischen 0V und 15V. Berechnen Sie die anliegende Spannung in Volt und rechnen Sie mit dieser Zahler in Q-Notation weiter.

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

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 entsprechen an, falls notwendig.

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. Achten Sie darauf, dass Sie keine Werte aus der alten Runde verwenden.

☞ assert()

Normalerweise wäre die Ausnahmebehandlung für einen wiederherstellbaren Fehler eben diese Wiederherstellung. Da wir im aktuellen emulierten System aber mit an Sicherheit grenzender Wahrscheinlichkeit keine Hardwarefehler erfahren

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

werden, beenden Sie Ihr System auch im Falle eines wiederherstellbaren Fehlers, da dies vermutlich eine Verletzung des Replikationsdeterminsmuses bedeutet. Das heißt auch die Abwesenheit von Einstimmigkeit muss als Fehler gewertet werden. Was müsste *normalerweise* beim Erkennen eines Fehlers in *einem* Replikat getan werden?

Geben Sie den gefilterten Wert „aus“ indem Sie `ezs_setOutputA()` aufrufen.

1.3 Evaluation

6. 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 Verletzung und eliminieren Sie diese.

7. Codegröße *Welche neuen möglichen transienten Fehler sind hinzugekommen?*

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

---

8. SoR: Bestimmen Sie die durch Redundanz gesicherten Bereiche in Ihrem Programm (→ Sphere of Redundancy) und markieren Sie diese durch Kommentare in Ihrem Quellcode.

2 Erweiterte Aufgeben

9. Analyse Untersuchen Sie, wie in der Tafelübung vorgestellt, Ihr Kompilat. Welche Teile sind tatsächlich repliziert? ☞ objdump

---

Wo befinden sich noch Lücken in der Replikation? Wie können diese geschlossen werden?

---

10. Umsetzung Nutzen Sie die Datei `include/filter.hpp` um einen mithilfe der in der Tafelübung gezeigten Programmiermaßnahmen eine Codeduplizierung zu erreichen. Übernehmen Sie Ihre bisherige Änderungen in die Datei `src/app_ext.cpp`. Überschreiben Sie die Datei nicht: Sie ist schon für die Verwendung von C++ angepasst. Mit `make tmr_ext` und `make run_tmr_ext` können Sie die neue Implementierung kompilieren und testen.

☞ C++-Templates

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabezeit: 23.05.2016
- Fragen bitte an `i4ezs@lists.cs.fau.de`