Hes·so VALAIS WALLIS
School of Engineering

XF

Jean Nanchen

OCTOBER 23, 2020

HES-SO – 3ÈME ANNÉE

Table des matières

1. INTRODUCTION

Dans ce laboratoire, nous allons implémenter notre propre Exécution Framework (XF). Il sera implémenté sur deux plateforme, QT & PTR Embedded System. En premier lieu, nous utiliserons QT pour une implémentation rapide, puis il sera adapté pour le système embarqué.
Pour tester notre code, nous utiliserons les testbench fournis.
Une documentation des tests : « documentation.html »
Github : https://github.com/73jn/XF

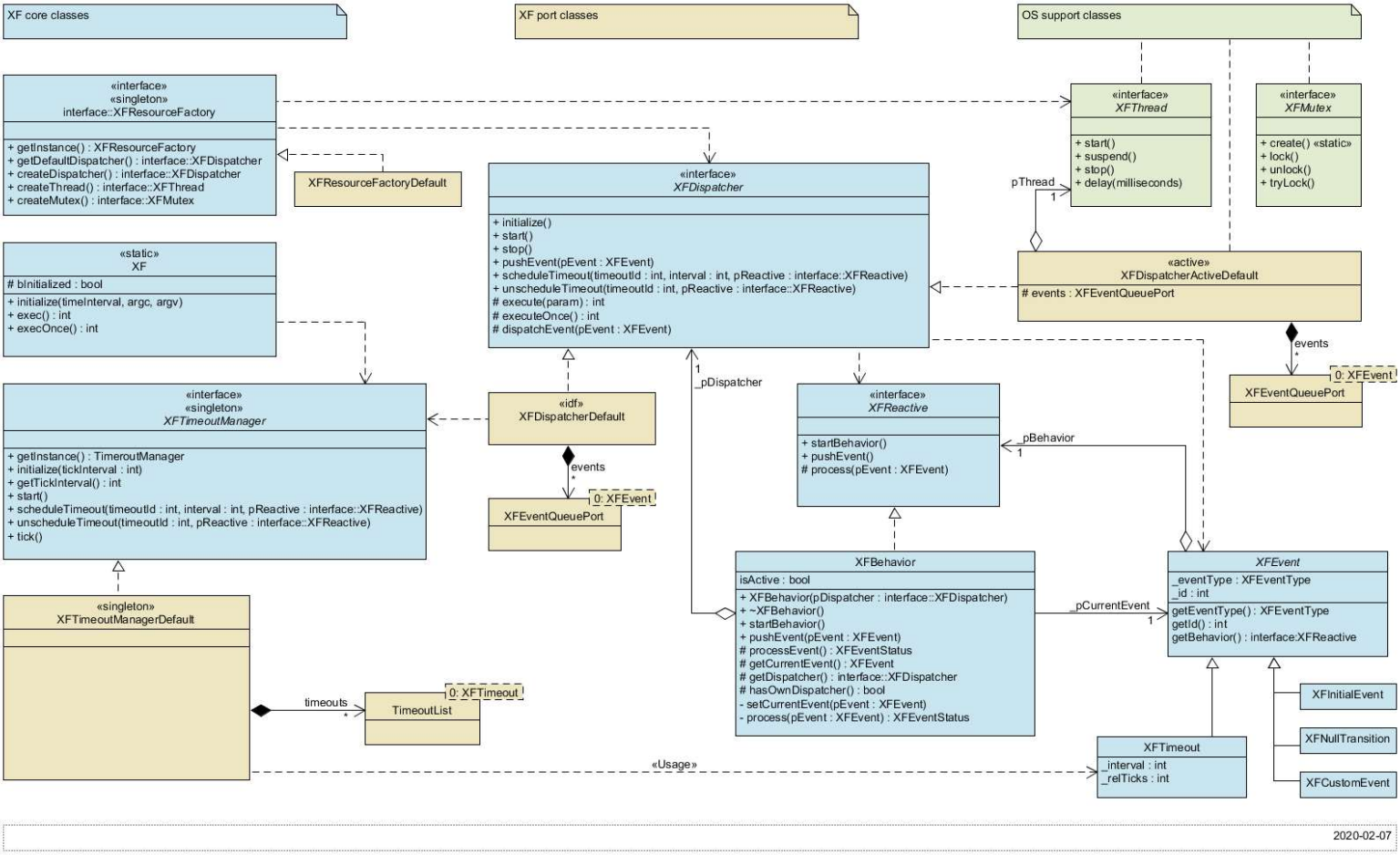
2. CAHIER DES CHARGES

Les objectifs à remplir sont :

- Algorithms pour le XFTimeoutManagerDefault pour qu'il se comporte de la même façon avec un ou n timouts.
- Implémenter les classes Core du XF
- Implémenter les classes Port du XF
- Tester et commenter le code du XF
- Tester sur QT et le système embarqué

3. *DIAGRAMME DE CLASSE*

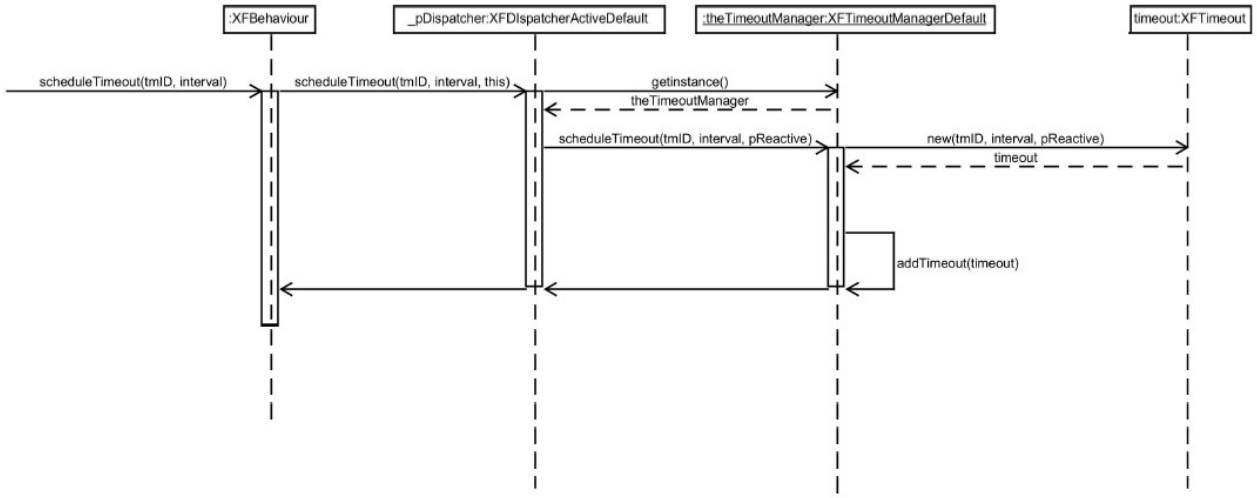
Dans ce diagramme de classe nous observons les relations entre les classes. En bleu se trouve le CORE et en brun les classe de PORT.
Le XFTimeoutManager s'occupe des timeouts, il ajoute, supprime, tick() la liste des timeouts.
Le dispatcher ajoute les events dans la liste des events et envoie dans le XFTimeoutManager les timeouts. Il réceptionne aussi les timeouts qui arrivent à échéance. Un thread exécute la fonction execute() qui traite le queue des events.
La classe behaviour est héritée par la state machine. Elle à accès au dispatcher donc elle peut lui envoyer des events, push des timeouts. Elle dispose d'une fonction start qui envoie un event start. C'est dans cette classe que l'on met à jour le pointeur du current event.



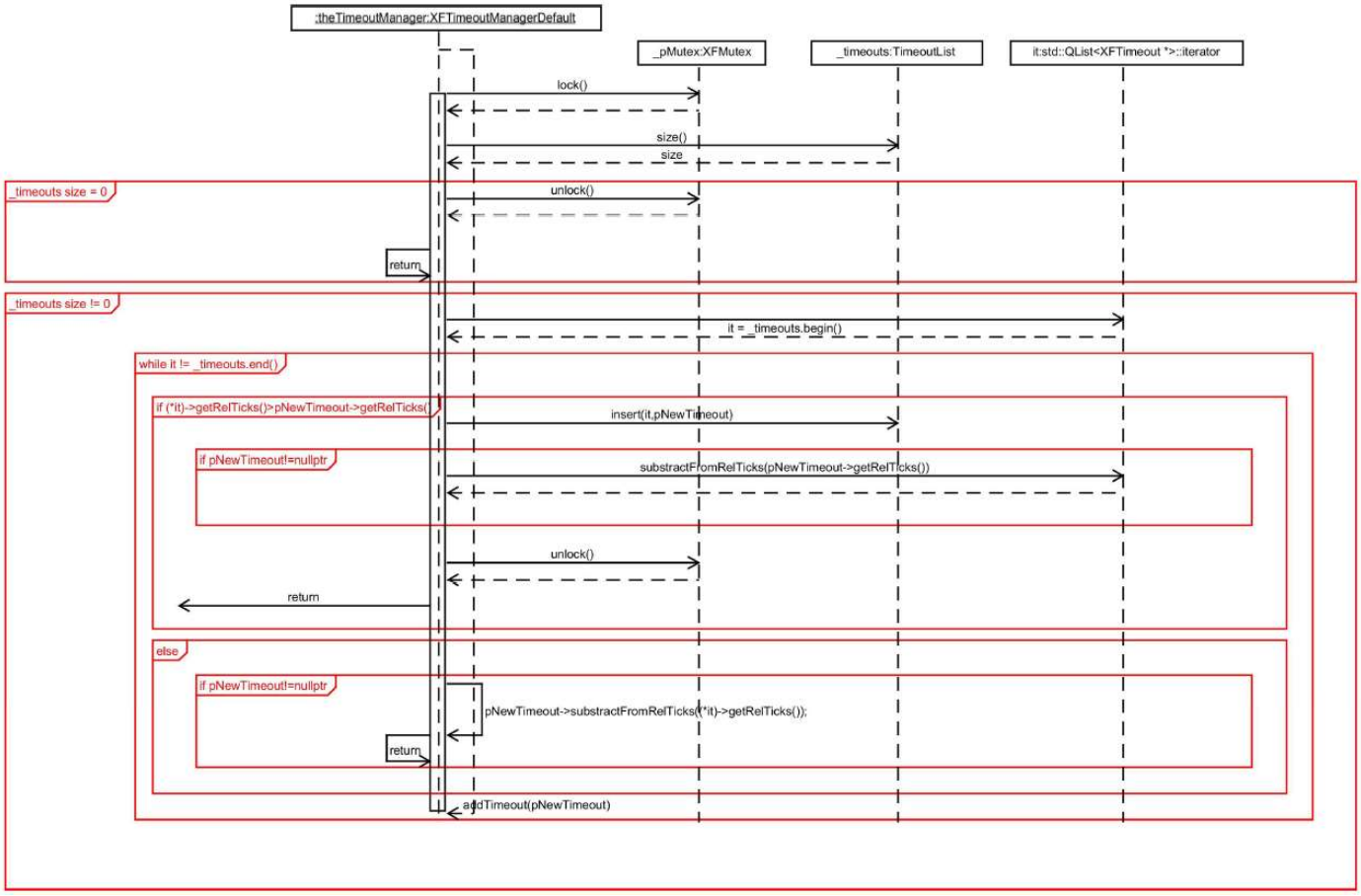
4. ANALYSE

DIAGRAMME DE SÉQUENCE

Sur ce diagramme de séquence, on peut observer le comportement du XF et le relations entre les classes lorsque l'on schedule un timeout. Le behaviour envoie au dispatcher qui va envoyer dans le timeoutManagerDefault. Le timeoutManagerDefault va créer un timeout et utiliser sa fonction addTimeout (décrite plus loin).



Le diagramme de séquence ci-dessous explique comment un timeout est ajouté dans la liste _timeouts grâce à la fonction addTimeouts(..).



On voit sur ce deuxième diagramme de séquence que lorsque l'on ajoute un timeout, on verrouille le mutex (ce qui permet de désactiver les interruptions). Ensuite, on demande la taille de la list des timeouts. Si elle est égale à 0, on met le timeout dans la liste, on déverrouille le mutex et on effectue un return. Si la taille n'est pas égale à 0 (il y a déjà des timeouts dans notre liste), un iterator est créé pour parcourir la liste, et nous effectuons l'algorithme de tri.

Sur l'image ci-dessous nous retrouvons l'explication de notre algorithme. Le remaining tick correspond au nombre de tick qu'il reste avant le timeout. Le timeoutTicks correspond au nombre de tick que l'on doit attendre pour lancer le timeout. Il faut ajouter notre nouveau timeout entre timeout ayant 25 de timoutTicks et celui ayant 53 de timeoutTick (car 30 est entre ces deux nombres). Quand nous ajoutons un timeout de 30 dans notre queue, il faut soustraire le remaining tick de notre nouveau timeout a la somme des remaining tick des éléments dans la queue (ceux avant l'endroit inséré). Cela permet d'ordrer la queue. Il faut bien sur rafraichir le remaining tick des timeouts après l'endroit de l'insertion.

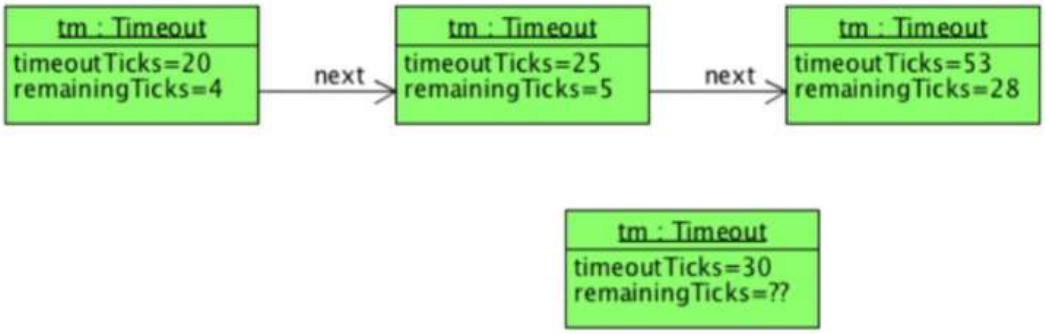


Figure 3 - Arrivée d'un nouveau timeout

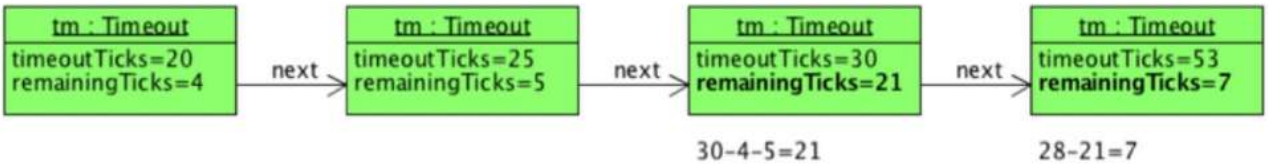


Figure 2 - Nouvelle queue avec les valeurs recalculées

Il est décrit, sur ce troisième diagramme de séquence, comment le tick() opère. Premièrement le mutex est verrouillé. S'il n'y a pas de timeout dans la liste, nous effectuons un return. Si la liste des timeouts n'est pas vide, nous créons un iterator pour parcourir la liste (pointe sur le début de la liste). Nous soustrayons une constante au premier élément de la liste. Ensuite nous regardons si cette valeur est négative ou égale à zéro. Si c'est le cas, cela veut dire qu'il faut envoyer le timeout dans la liste d'event, grâce à la fonction returnTimeout décrite plus bas. Il est ensuite effacé de la liste avec erase(..).

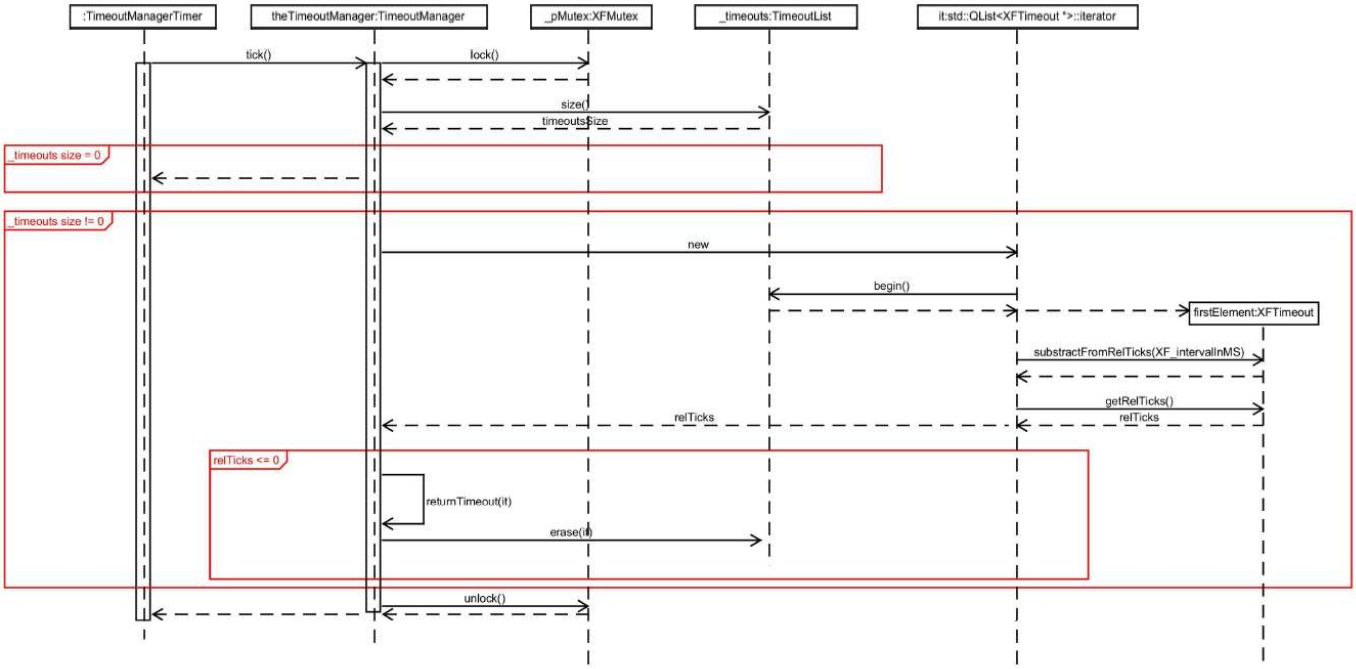


Figure 4 - tick()

Ci-dessous se trouve le diagramme de séquence de la fonction returnTimeouts() utilisée pas la fonction tick(). On voit que l'on envoie le timeout dans la liste d'event.

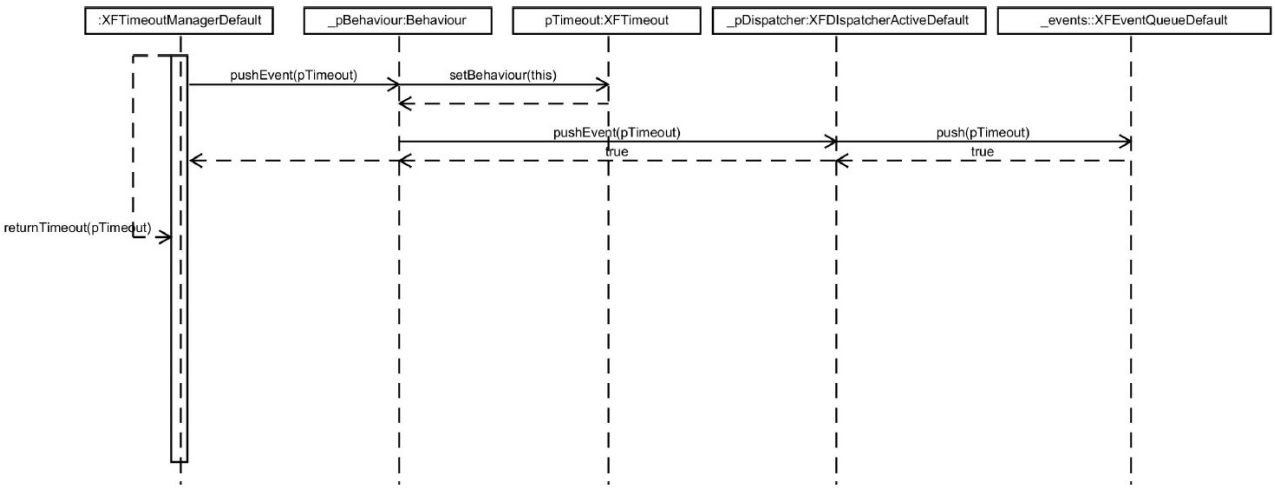


Figure 5 - returnTimout()

5. TESTS

TEST 01

Dans ce test, nous envoyons « Say Hello » chaque 1000ms et « Echo » toutes les 500ms.

Voici le résultat attendu :

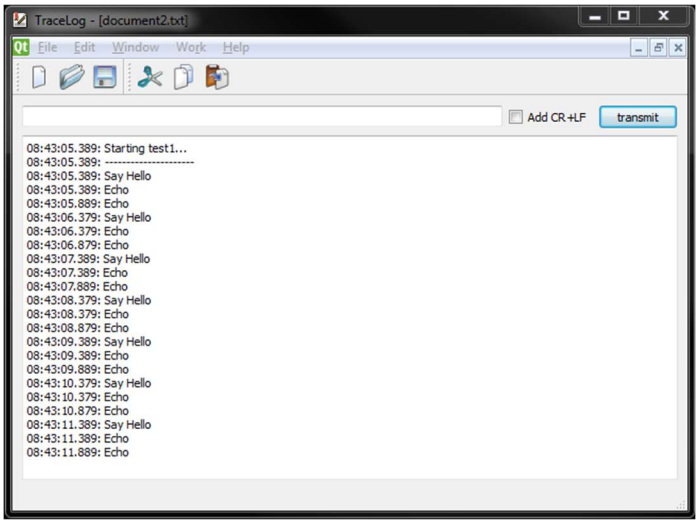


Figure 7 - Test 01 résultat attendu

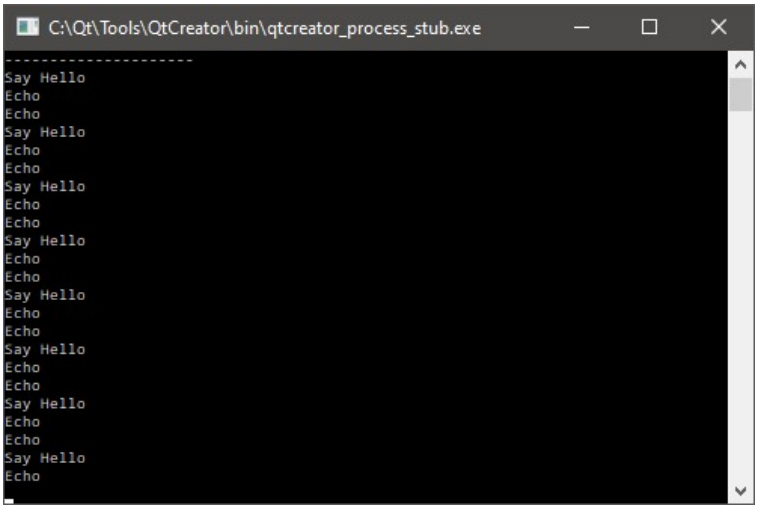


Figure 6 - Test 01 QT résultat obtenu

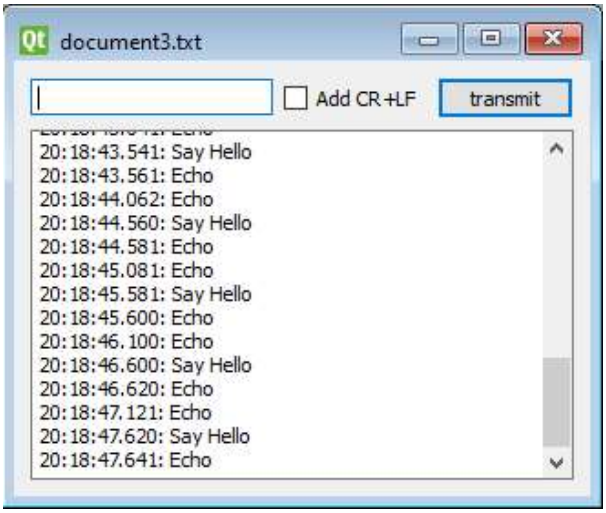


Figure 8 - Test 01 STM32 résultat obtenu

TEST 02

Dans ce test, nous créons deux SM qui lance chacune un compteur. Une fois le compteur décrémenté, nous envoyons un signal terminate aux deux SM et nous appelons le destructeur.

Voici le résultat attendu :

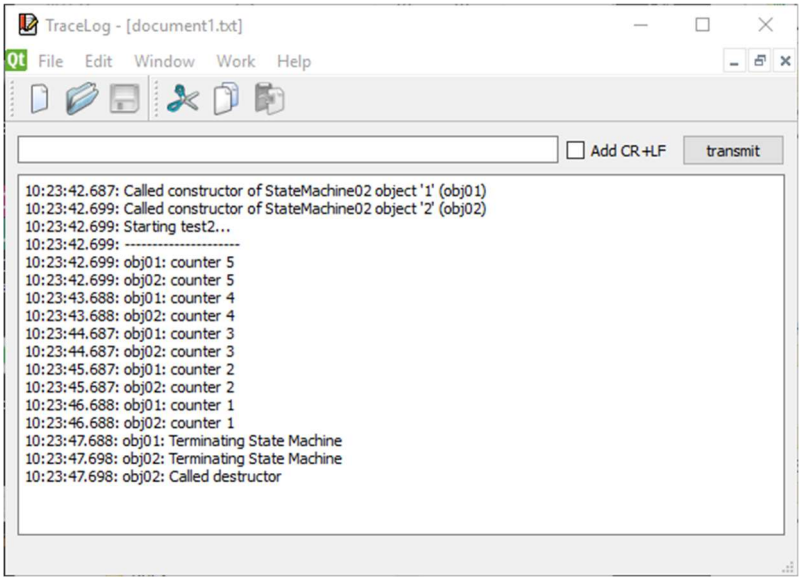


Figure 11 - Test 02 résultat attendu

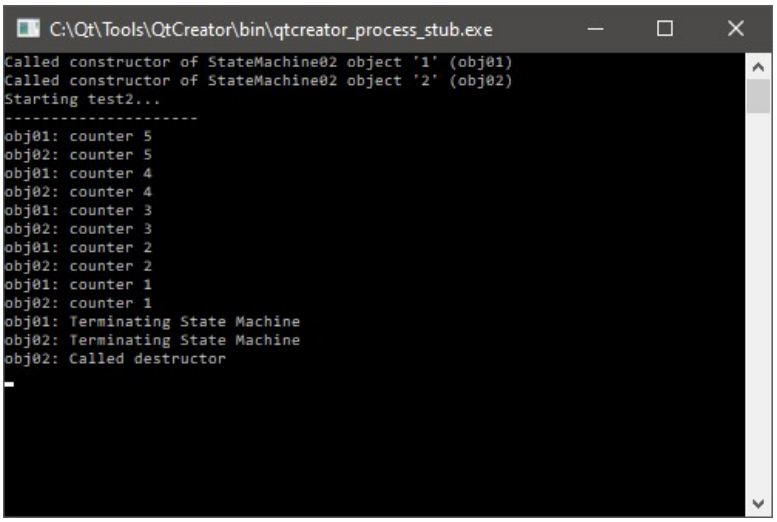


Figure 10 - Test 02 QT résultat obtenu

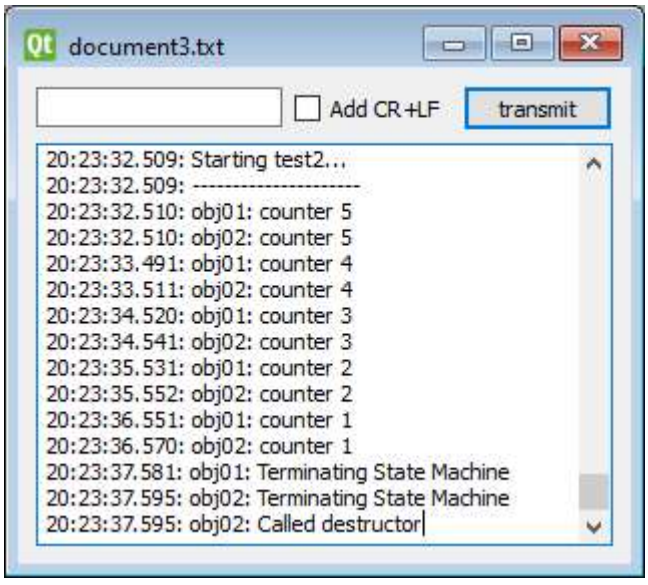


Figure 9 - Test 02 STM32 résultat obtenu

TEST 03

Dans ce test, nous passons de l'état wait à l'état wait restart en utilisant les timeouts.

Voici le résultat attendu :

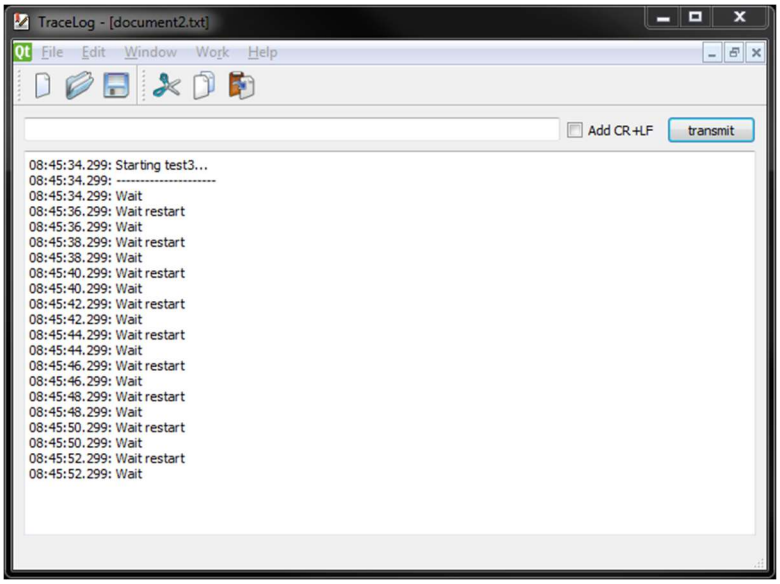


Figure 12 - Test 03 résultat attendu

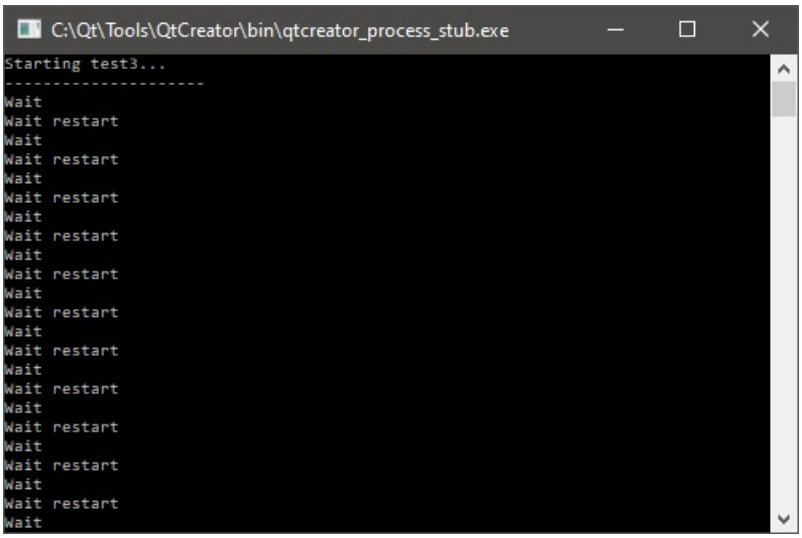


Figure 14 - Test 03 QT résultat obtenu

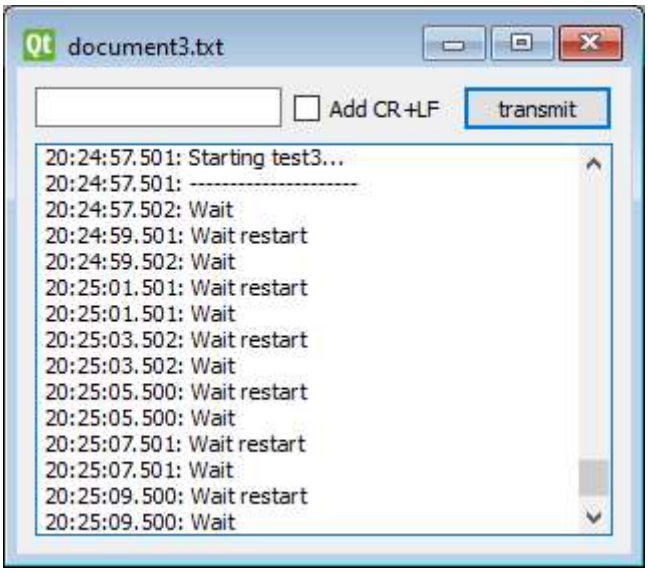


Figure 13 - Test 03 STM32 résultat obtenu

TEST 04 QT

Dans ce test, nous testons si les timeouts sont correctement annulés.

Voici le résultat attendu :

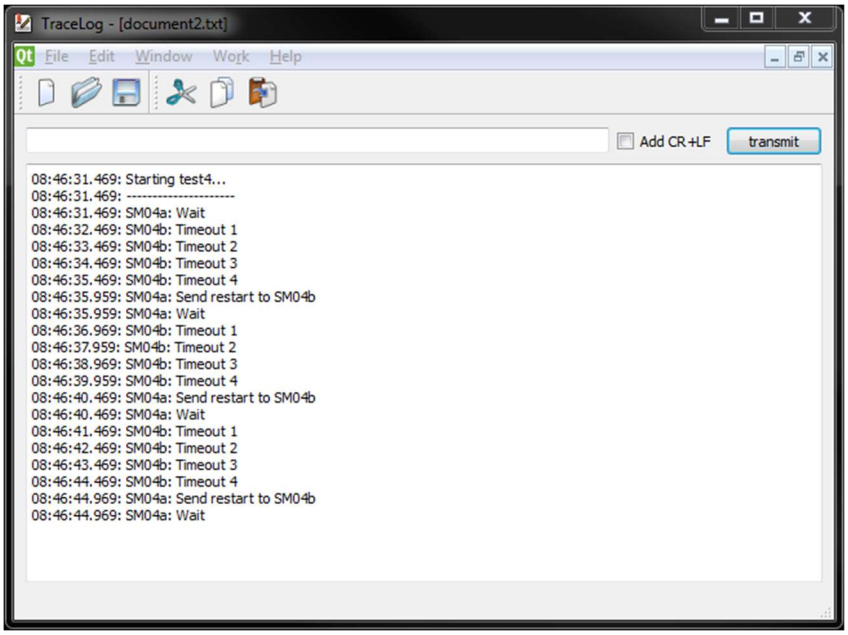


Figure 17 - Test 04 résultat attendu

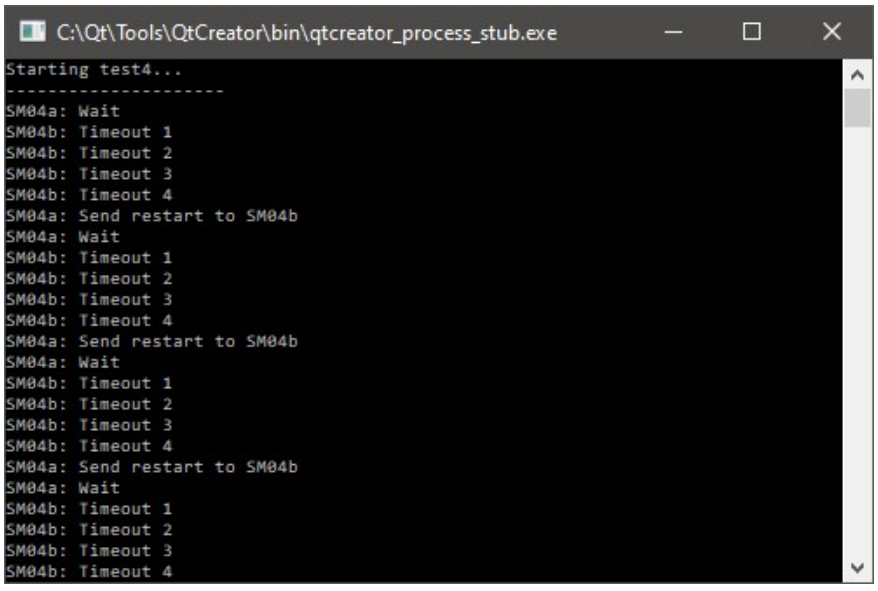


Figure 16 - Test 04 QT résultat obtenu

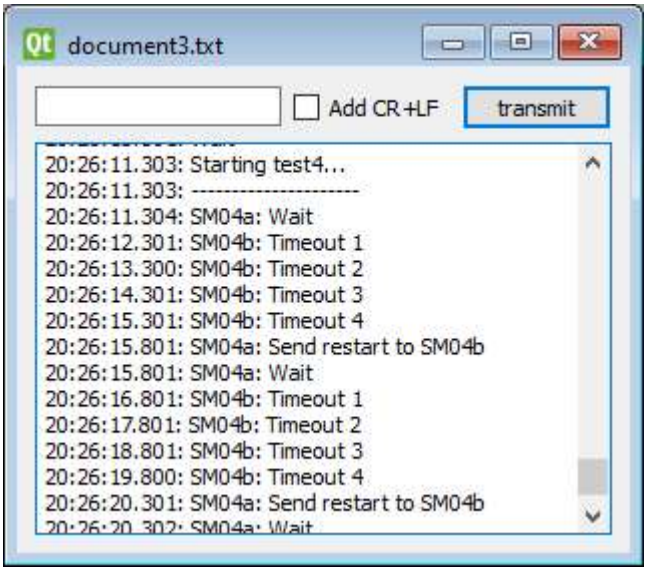


Figure 15 - Test 04 STM32 résultat obtenu

TEST 05 QT

Dans ce test, nous envoyons plusieurs timeouts en même temps.

Voici le résultat attendu :

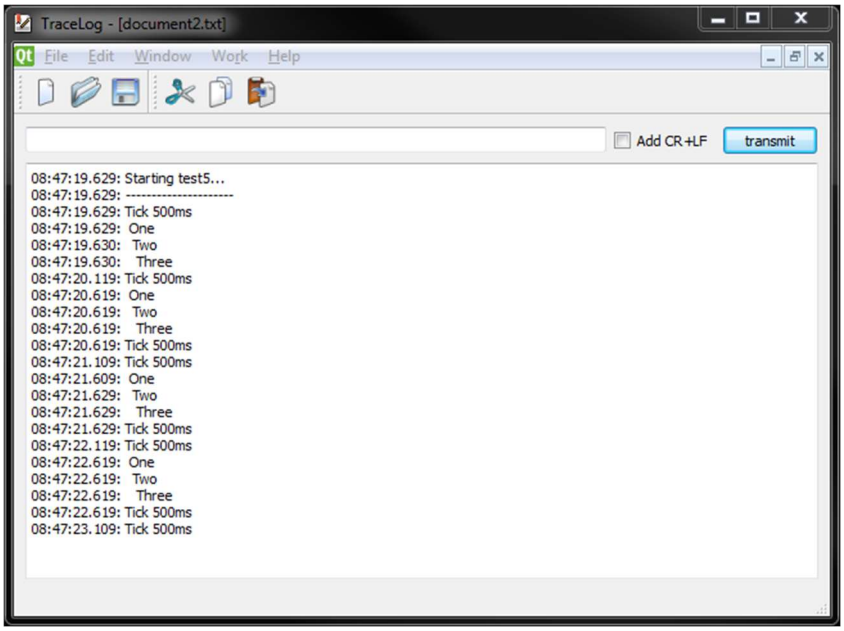


Figure 20 - Test 05 résultat attendu

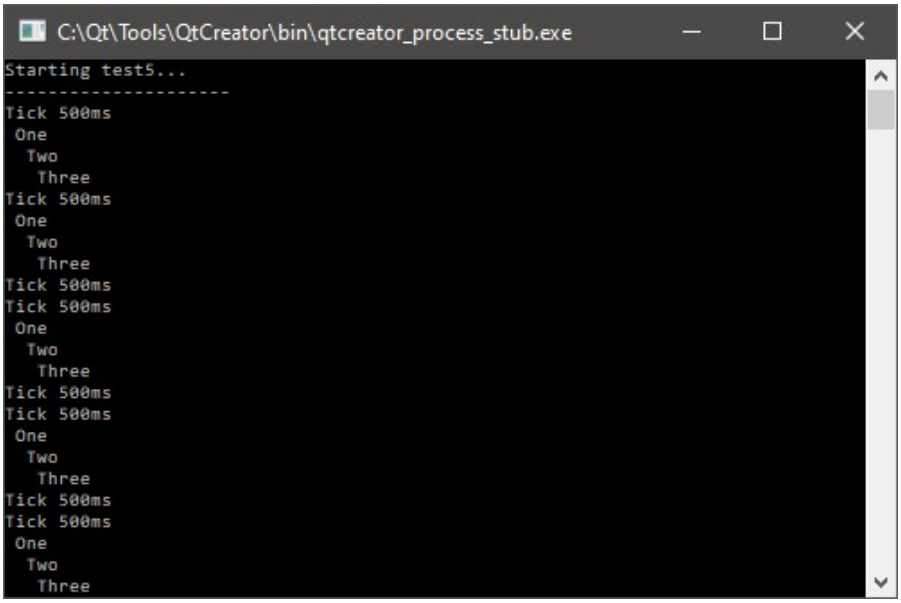


Figure 19 - Test 05 QT résultat obtenu

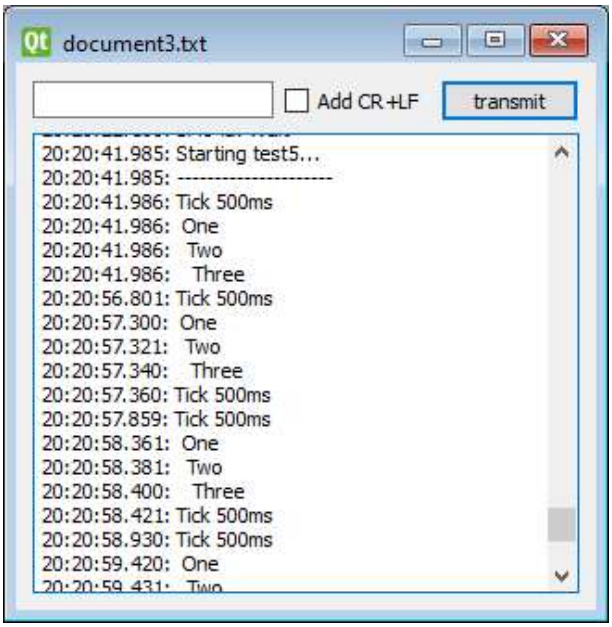


Figure 18 - Test 05 STM32 résultat obtenu

6. RÉSUMÉ DES TESTS

Test n°	Résultat	Remarque
01 QT	Fonctionnel	
02 QT	Fonctionnel	
03 QT	Fonctionnel	
04 QT	Fonctionnel	
05 QT	Fonctionnel	
01 STM32	Fonctionnel	
02 STM32	Fonctionnel	
03 STM32	Fonctionnel	
04 STM32	Fonctionnel	
05 STM32	Fonctionnel	

7. CONCLUSION

Durant ce projet, nous avons développé un architecture Exécution Framework. Nous l'avons d'abord testé sur QT et adapté pour un système embarqué. Le Framework fonctionne à 100% et un algorithme de queue intelligente pour les timeouts à été ajouté comme vu en cours.

8. ANNEXES

Github : github.com/73jn/XF