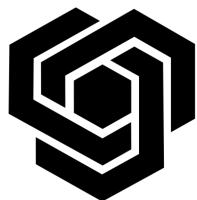


Université technique de Sofia

Faculté francophone

Génie informatique et télécommunications



Projet du fin d'études

Étudiant : Radostina Chipanova

Numéro de faculté : 211311015

Directeur scientifique : Prof. Svetozar Andreev

**Sujet : Systèmes industriels sur ordinateur
avec « real time Linux »**

	Contenu
1. Introduction.....	2
2. Motifs.....	3
2.1. Solutions actuelles – DOS et Windows.....	3
2.2. Solution Linux.....	4
3. Systèmes temps réel.....	5
3.1. Description et particularités.....	5
3.2. Classification par le degré d'importance de l'exécution précise dans le temps.....	6
3.3. Classification par la structure du fonctionnement.....	7
3.4. Classification des systèmes par ses architectures.....	9
4. Systèmes d'exploitation temps réel.....	10
4.1. Préemption.....	10
4.2. Inversion et héritage de la priorité.....	11
4.3. Conception.....	13
4.4. Noyau temps réel.....	15
5. Linux comme un système d'exploitation temps réel.....	16
5.1. Priorité d'un processus.....	16
5.2. PREEMPT-RT.....	18
5.3. Xenomai.....	2
6. Outils.....	26
6.1. Qt.....	26
6.2. GDB serveur.....	29
7. Conception du logiciel « Ladd ».....	31
7.1. Application.....	31
7.2. Structure principale.....	32
7.3. Fonctionnement principal.....	34
7.3.a Configurer le logiciel (set-up).....	34
7.3.b Communication entre l'ordinateur et le dispositif.....	35
8. Configuration du projet « Ladd ».....	36
8.1. La configuration comme structure des données.....	36
8.2. Désérialisation de la configuration.....	40
8.2.a Le fichier config.xml.....	40
8.2.b Chargement du fichier – interface utilisateur.....	42
8.2.c Désérialisation.....	47
8.3. Application de la configuration	54
9. Logique ladder.....	56
9.1. Conception.....	56
9.2. Structure Ladder dans le projet « Ladd ».....	59
9.2.a Structure générale.....	59
9.2.b Liste des entières – ladderSequence.....	61
9.3. Désérialisation du ladder.....	63
9.4. Fonctionnement du ladder.....	68
10. Communication avec des dispositif.....	72
10.1. Implémentation générale.....	72
10.2. Implémentation parallèle.....	78
10.3. Implémentation en série.....	81
11. Mode d'emploi.....	84
12. Conclusion.....	88
13. Bibliographie.....	89

1. Introduction

Les deux objectifs principaux de ce projet de fin d'études sont:

- De faire une recherche si le système d'exploitation Linux peut satisfaire les besoins fonctionnelles des systèmes d'automatisation dans l'industrie
- La réalisation un prototype du logiciel du contrôle d'automatisation sur un ordinateur avec Linux

La structure principale des systèmes d'automatisation envisagées consiste d'un ordinateur personnelle, une machine de production et un dispositif, qui fait la connexion entre l'ordinateur et le matériel des machines.

L'ordinateur s'occupe de l'élaboration des commandes d'actions, qui sont transmises vers la machine. Dans l'ordinateur se trouve la logique du système. La machine de sa part s'occupe de l'exécution physique des commandes reçues. Dans le prototype réalisé on utilise la logique ladder pour prédefinir la réaction du système aux événements variés. Le logiciel élaboré, nommé « Ladd », peut interpréter un document textuel décrivant la logique ladder et il peut suivre cette logique en réagissant selon elle aux événements dans le système.

Pendant le travail sur le projet on ne dispose pas d'une machine réelle, mais se ne pose aucun problème sur le développement du système, parce que pour l'ordinateur les événements et les réactions du système sont exprimés en signaux échangés avec le dispositif de connexion. Dans les applications ciblées de contrôle d'automatisation l'ordinateur n'a pas aucune contact direct avec les machines de production et quoi qu'il soit d'autres matériel externes que le dispositif.

Dans le document courant on analyse premièrement les motifs pour choisir Linux comme système d'exploitation pour une application industrielle de contrôle d'automatisation (cette analyse se trouve dans le chapitre 2.Motifs). Puis on présente les caractéristiques d'une application temps réel (ch. 3.Systèmes temps réels). Tous les systèmes d'automatisation sont des systèmes temps réels et c'est le facteur le plus important, ce qui concerne le développement du logiciel et le choix d'un système d'exploitation. On analyse aussi les caractéristiques d'un systèmes d'exploitation temps réel pour pouvoir évaluer si Linux est une solution convenable et effective pour les buts de l'application (ch. 4.Systèmes d'exploitation temps réel). Envisageant que en effet Linux n'est pas conçu d'être un système d'exploitation temps réel on a fait une recherche des façons de l'adapter aux exigences des applications industrielles (ch. 5. Linux comme système d'exploitation temps réel) . Puis on présente le serveur gbd, utilisé pour déboguer le prototype, et le projet Qt – un ensemble des bibliothèques c++, qui on utilise pour réaliser l'application du contrôle (ch. 6.Outils). On analyse la structure et le fonctionnement principal de l'application développée (ch. 7.Conception du logiciel « Ladd », ch. 8.Configuration du projet « Ladd » , ch. 9.Logique ladder, ch. 10.Communication avec des dispositif).

Ce projet est réalisé avec l'assistance d'une firme nommée Semis. Elle s'occupe à l'automatisation des machines industrielles. Ses programmes sont écrit à C++ sur des ordinateurs avec un système d'exploitation DOS. Dans cette firme on veut moderniser le logiciel du contrôle et ce prototype - « Ladd », sert comme la première pas de la transition de DOS vers Linux (Ubuntu). L'assistance de la firme au projet consiste en fournissant les dispositifs avec lesquels le logiciel communique en utilisant les interfaces de communication – LPT et RS232. Le projet « Ladd » est open source et son développement à ma part n'envisage pas des gains commerciaux .

2. Motifs

2.1. Solution actuelle

Les solutions actuelles utilisées par la firme Semis sont avec des systèmes d'exploitation DOS et Windows. En effet ni Linux, ni Windows, ni même si DOS sont conçus comme des systèmes d'exploitation temps réel. Cependant on réussit de réaliser des applications industrielles effectives et fiables avec ils. La plupart des systèmes d'exploitation temps réel ne sont pas une solution préférée, parce que ils sont payés, mal-connus ou avec une performance moyenne mauvaise.

Solution DOS

Le système d'exploitation DOS ne porte pas l'étiquette «RTOS» (real time OS), mais en pratique il répond aux exigences d'être tel. Avec son simplicité ce système est fiable et flexible ce qui concerne le déterminisme d'exécution des tâches (quand une interruption se produit on réagit immédiatement à lui) et la réinitialisation fréquence des compteurs d'ordinateur (timer). En plus c'est important que DOS est un système gratuit.

Les désavantages de DOS sont nombreux. Le développement d'un interface d'utilisateur – même le plus simple UI, est un défi énorme pour le développeur du logiciel. L'usage de DOS n'est pas du tout populaire actuellement et pour cette raison en recherchant pour information et conseillers sur l'internet on ne trouve pas beaucoup des ressources. En plus l'utilisation d'interface USB est presque impossible et pas fiable, quand on réussit à l'utiliser. La réalisation d'un réseau aussi n'est pas une tâche triviale - même si l'utilisation d'internet est sous question. C'est aussi un grand problème qu'on ne peut pas utiliser des nombreux outils logiciels développer pour les OS populaires comme TeamViewer, Wireshark, QtCreator et beaucoup d'autres.

Étant donné les difficultés en DOS d'utiliser des interfaces USB et Ethernet, on doit réaliser les systèmes en utilisant des interfaces LPT et RS232 (parallèle et en série). Se sont des interfaces vieilles et c'est de plus en plus difficile de trouver un ordinateur avec des ports parallèles et séries.

L'avantage des solutions DOS sont que la logique du logiciel est située seulement dans l'ordinateur et quand il y a un problème avec le matériel ce n'est pas difficile de le résoudre en remplaçant le matériel.

Solution Windows

Windows n'est pas une bonne solution pour un système temps réel ce qui concerne l'ordonnancement des tâches au niveau de processeur et la fréquence des compteurs. Ici il n'a pas des problèmes de DOS, qui est un système dépassé, mais Windows de sa part n'offre pas des outils nécessaires de satisfaire les besoins fondamentaux d'un système temps réel. En fait il y a une solution logicielle nommée Match, qui fonction sous Windows et satisfait parfaitement les exigences temps réel. Mais ce projet n'est pas open source et on ne peut pas voir comment ça se fait.

La solution Windows est réalisée en utilisant des microcontrôleurs (PIC). Toute la logique qui contrôle le travail de la machine est exportée dans ce pièce externe. Le microcontrôleur n'a aucun difficulté dans l'exécution des tâches en temps réel avec des fréquences très grandes. Les ordres arrivants du ordinateur sont très limitées et concernent des actions comme début d'exécution, nombre de répétition, vitesse moyenne etc. - des directions, qui n'exigent une exactitude dans le temps.

Le problème de cette solution est que la logique est située dans un microcontrôleur et quand il y a

un problème avec le matériel de ce contrôleur, l'utilisateur doit acheter le système de nouveau. Alors le logiciel n'est pas un *.exe qui peut être simplement installer sur un autre ordinateur. On doit acheter un contrôleur programmé de nouveau.

2.2. Solution Linux

Linux aussi n'est pas un système d'exploitation temps réel. En effet il y a une bonne explication pourquoi les OS populaires ne sont pas temps réel. Un RTOS (real time OS) est un OS lent. Ça a un aire paradoxal, mais est un conséquence absolument logique de l'architecture des RTOS. En donnant des préférences à un tâche, on doit diminuer le temps de processeur des autres tâches. Alors tous les autres tâches sont exécutées plus lentement.

De toute façon la société énorme des programmistes, qui s'occupent au développement du Linux, a fait attention sur la demande aux fonctionnalités temps réel. Nombreux solutions sont élaborées pour satisfaire cette demande et chaque solution est open source, présentée et expliquée clairement comment de l'implémenter. En effet c'est une caractéristique essentielle pour la culture des développeurs qui utilisent Linux – le partage d'information. Naturellement la culture open source n'est pas réservée seulement pour Linux, mais elle fait partie de ses fondements.

Les solution qui on a analysées pendant le développement du prototype sont le projet Xenomai et un patch préempte du noyau Linux.

L'utilisation de Linux pour le développement des systèmes industrielles est une solution effective et bonne marché:

- On peut utiliser tous les outils développés pour un OS moderne – par exemple TeamViewer
- On peut utiliser USB et Ethernet, qui sont des interfaces de communication beaucoup plus populaires (drivers, ordinateurs avec des ports convenables), vites et fiables que RS232 et LPT.
- On peut utiliser Internet et construire sans problème un réseau des ordinateurs.
- On peut obtenir information des sources nombreux dans l'internet sur chaque aspect du développement du logiciel
- On peut réaliser facilement des interfaces utilisateur
- On peut réaliser un système temps réel sans utiliser du matériel spécialisé (dans la plupart des cas).

Ce sont les motifs pour développer un système temps réel sur un ordinateur avec Linux.

3. Systèmes temps réel

3.1. Description et particularités

En informatique, le terme « systèmes temps réel » (Real Time Computing - RTC) décrit les systèmes matériels et logiciels, qui doivent observer des contraintes temporelles exactes d'exécution des tâches. Par exemple la réponse du logiciel aux événements du système (alarmes, mouvement des moteurs, stop urgent etc.) doit être exécuter dans le temps avec une précision des millisecondes ou parfois même si des microsecondes sont d'une importance significative.

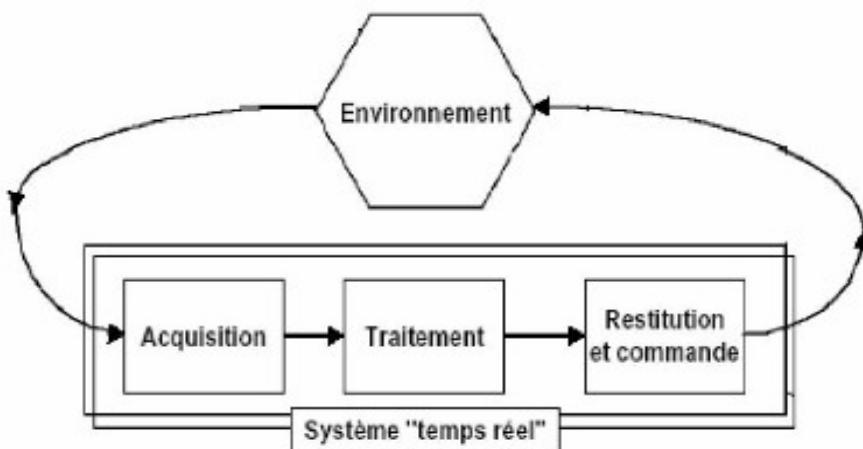


Figure 3.1.1 – schéma du fonctionnement du système temps-réel en accord avec les événements de l'environnement

On parle d'un système temps réel lorsque ce système est capable de contrôler un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé – Figure 3.1.1. Ça veut dire recevant des données, les traiter, et retourner des résultats suffisamment rapidement pour pouvoir affecter l'environnement à ce moment-là.

Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par le fait que le respect des contraintes temporelles est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés.

Le terme «en temps réel» est également utilisé dans l'industrie pour signifier des systèmes de contrôle de processus «sans retard significatif».

« Même la bonne réponse est fausse si elle arrive trop tard » - Figure 3.1.2

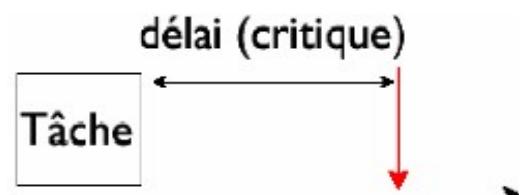


Figure 3.1.2 Illustration du délai critique dans le temps

Un système temps réel pilote un processus comportant des contraintes de temps aléatoires et variées. Ce système doit être déterministe puisqu'il doit savoir avec précision l'instant de début et de fin d'un traitement. Dans un système temps réel, les contraintes temporelles portent essentiellement

sur les dates de début et de fin d'exécution des tâches.

Une tâche temps réel est associée à des contraintes de temps et de ressources.

Selon les contraintes et les caractéristiques des tâches le logiciel en temps réel peut utiliser un ou plusieurs des éléments suivants:

- Langages de programmation synchrones .

La programmation synchrone (aussi « programmation réactive synchrone ») est un paradigme de programmation informatique réalisé par des langages de programmation synchrones. Les premiers langages de programmation synchrones ont été inventés en France dans les années 1980: Esterel, Lustre et Signal. Depuis, de nombreux autres langages synchrones ont émergé.

Pour prendre un exemple concret, la déclaration en Esterel "toutes les 60 secondes minutes emit" indique que le signal "minute" est exactement synchrone avec le 60-ème apparition du signal "second".

- Un système d'exploitation temps réel

En anglais RTOS pour « real-time operating system » est un système d'exploitation multitâche destiné aux applications temps réel. Un RTOS n'a pas nécessairement pour but d'être haut performant et rapide. Un RTOS fournit des services et des primitives qui, si elles sont utilisées correctement, peuvent garantir les délais souhaités. Un RTOS utilise des ordonneurs spécialisés afin de fournir aux développeurs des systèmes temps réel les outils et les primitives nécessaires pour produire un comportement temps réel souhaité dans le système final.

- Des réseaux en temps réel.

Le développement de systèmes temps réel nécessite donc que chacun des éléments du système soit lui-même temps réel, c'est-à-dire permettre de prendre en compte des contraintes temporelles et la priorité de chacune des tâches.

3.2. Classification des systèmes par ses architectures

Les systèmes temps réel peuvent être classés selon leur couplage avec des éléments matériels avec lesquels ils interagissent. Ainsi, l'application concurrente et le système d'exploitation qui lui est associé peuvent se trouver :

- soit directement dans le procédé contrôlé : c'est ce que l'on appelle des systèmes embarqués (embedded systems). Le procédé est souvent très spécialisé et fortement dépendant du calculateur. Les exemples de systèmes embarqués sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager, missile. C'est le domaine des systèmes spécifiques intégrant des logiciels sécurisés optimisés en encombrement et en temps de réponse.
- soit le calculateur est détaché du procédé : c'est souvent le cas lorsque le procédé ne peut être physiquement couplé avec le système ou dans le cas général des contrôle/commandes

de processus industriels. Dans ce cas, les applications utilisent généralement des calculateurs industriels munis de systèmes d'exploitation standards ou des automates programmables industriels comme dans les chaînes de montage industrielles par exemple .

3.3. Classification des systèmes du temps réel par le degré d'importance de l'exécution précise dans le des tâches :

- Le temps réel souple – Figure 3.3.1

Le non respect d'une échéance d'une fonction temps réel à contrainte souple n'est pas catastrophique et la qualité de la réponse, bien que dégradée, reste acceptable. Un système temps réel souple ne comporte que des fonctions ayant des contraintes temps réel souples. Il peut être validé en raisonnant en termes statistiques sur le temps de réponse moyen.

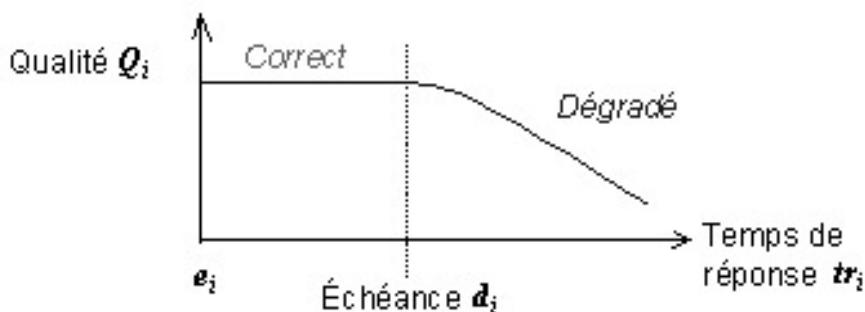


Figure 3.3.1 Illustration du fonctionnement d'un système temps-souple dans le temps

C'est le cas par exemple de fonctions de communication ou de traitement multimedia pour lesquels la qualité générale des documents n'est pas affectée de manière significative par le retard ou le manque d'informations d'une image ou d'un son.

Les composants utilisés pour réaliser les systèmes temps réel souples (systèmes d'exploitation, réseaux de communication, etc...) n'ont pas nécessairement des temps de réponse déterministes.

- Le temps réel dur – Figure 3.3.2

Le non respect d'une échéance d'une fonction ayant une contrainte temps réel dure est considéré comme une défaillance pouvant entraîner des conséquences catastrophiques. Un système temps réel dur comporte des fonctions de ce type et éventuellement des fonctions à contraintes souples.

L'objectif principal dans la réalisation d'un tel système est de satisfaire strictement toutes les contraintes de temps dures. Les techniques d'ordonnancement et de validation associées sont basées sur les échéances.

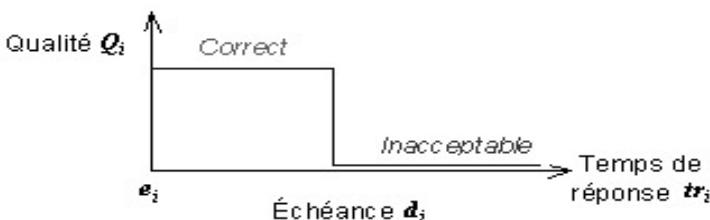


Figure 3.3.2 Illustration du fonctionnement d'un système temps-dure dans le temps

Les applications caractéristiques du temps réel dur se trouvent en aéronautique, dans le domaine

automobile, dans la robotique, la commande de processus industriels, ...

Les éléments utilisés pour réaliser ces systèmes doivent tous posséder un comportement déterministe. Il est nécessaire de choisir des composants logiciels et matériels et des principes de réalisation souvent spécifiques et permettant de garantir un comportement global.

- Le temps réel ferme – Figure 3.3.3

Certaines fonctions ayant des contraintes temps réel souples doivent respecter un temps de réponse moyen tout en restant à l'intérieur d'une fenêtre temporelle stricte. De telles fonctions dites "temps réel ferme" sont prises en compte dans les méthodes d'analyse et de validation temporelle des systèmes temps réel.

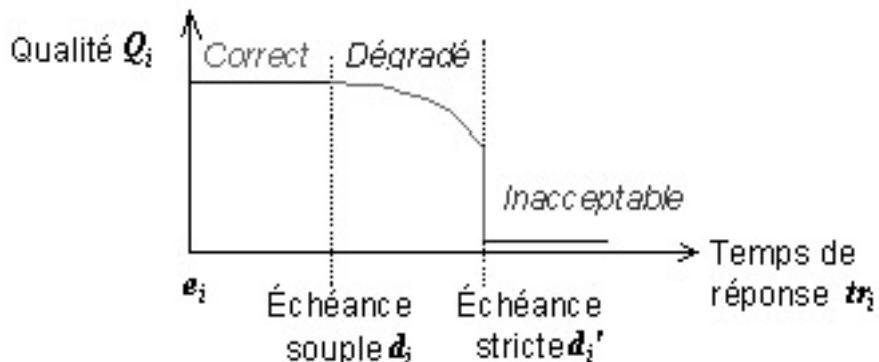


Figure 3.3.3 Illustration du fonctionnement d'un système temps-souple dans le temps

C'est le cas par exemple d'une fonction d'échantillonnage périodique devant répondre en moyenne en une période tout en étant capable de conserver au plus les données de quelques périodes.

Dans le cas d'une charge importante, celle-ci doit obligatoirement fournir ses résultats avant l'écoulement de ces quelques périodes.

- Le temps réel incrémental – Figure 3.3.4

La qualité de la réponse des fonctions à contraintes de temps incrémentales s'accroît avec le temps de calcul. A l'échéance du traitement, la réponse est acceptable si cette qualité dépasse un certain seuil.

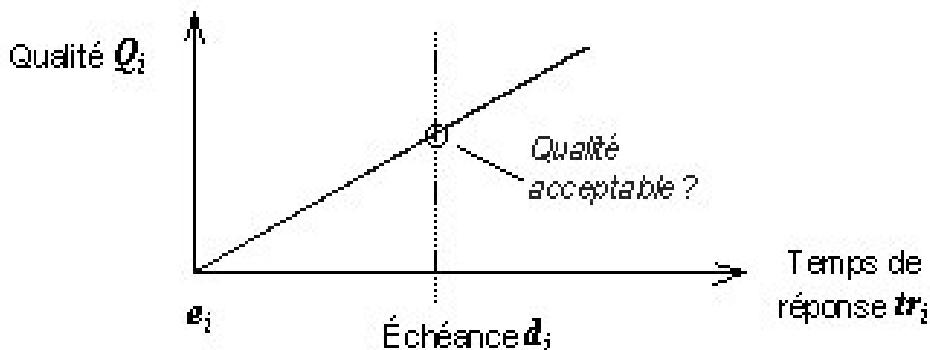


Figure 3.3.4 Illustration du fonctionnement d'un système temps- incremental dans le temps

La recherche de matériels et logiciels ayant les meilleures performances possibles en restant

compatible avec un coût maximal de la solution est quasiment la règle dans ce type d'application. Ils sont utilisés à leurs limites pour diminuer le temps au bout duquel la qualité est atteinte ou bien pour obtenir la meilleure qualité à temps de réponse constant. Ces fonctions se trouvent par exemple en traitement d'images (reconnaissance d'objets), dans les applications multimédia (compression vidéo) ou en calcul scientifique.

Des fonctions à contraintes incrémentales sont prises en compte par les méthodes d'analyse de performances. Des fonctions appelées IRIS ("Increased Reward with Increased Service") dans la littérature américaine peuvent être séparées en deux parties : une partie obligatoire devant se terminer et une partie incrémentale à résultat imprécis.

3.4. Classification par la structure de fonctionnement du système – elle peut être en boucle ouverte ou fermée.

Celui concerne la façon dans laquelle le système est utilisable – si on doit avoir un opérateur, qui donne des commandes à exécution, ou il s'agit d'un système autonome, qui prennent automatiquement des décisions comment réagir aux événements courants.

Structure en boucle ouverte – Figure 3.4.1



Figure 3.4.1 Schéma de la structure en boucle ouverte

Structure en boucle fermé – Figure 3.4.2

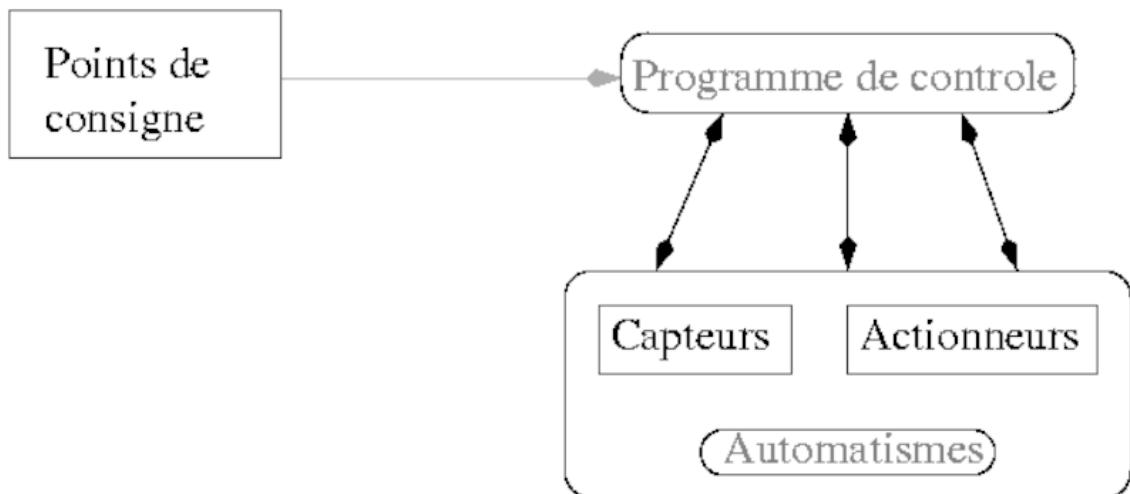


Figure 3.4.2 Schéma de la structure en boucle fermé

4.Systèmes d'exploitation temps réel (RTOS)

Ce qui concerne les systèmes d'exploitation, le terme « temps réel » signifie déterminisme - exactitude et précision d'exécution des tâches dans le temps. Cependant « temps réel » ne signifie pas exécution rapide des tâches - un RTOS n'a pas nécessairement pour but d'être performant et rapide, il seulement fournit des services et des primitives qui, si elles sont utilisées correctement, peuvent garantir les délais souhaités exactes.

Dans l'informatique presque chaque avantage d'un domaine est obtenu par un sacrifice d'efficacité dans un autre domaine. Ce soit le cas du déterminisme des systèmes d'exploitation temps réel et pour cela cette systèmes ne sont pas bien connue entre la plupart des utilisateurs contemporains.

4.1. Préemption

Le système d'exploitation temps réel est un système préemptif. En informatique, la préemption est la capacité d'un système d'exploitation multitâche à exécuter ou arrêter une tâche planifiée en cours.

Dans un système d'exploitation multitâche préemptif, les processus ne sont pas autorisés à prendre un temps non-défini pour s'exécuter dans le processeur. Une quantité de temps définie est attribuée à chaque processus ; si la tâche n'est pas accomplie avant la limite fixée, le processus est renvoyé dans la pile pour laisser place au processus suivant dans la file d'attente, qui est alors exécuté par le processeur. Ce droit de préemption peut tout aussi bien survenir avec des interruptions matérielles.

L'ordonnanceur distribue le temps du processeur entre les différents processus. Dans un système préemptif, à l'inverse d'un système collaboratif, l'ordonnanceur peut interrompre à tout moment une tâche en cours d'exécution pour permettre à une autre tâche de s'exécuter.

Certaines tâches peuvent être affectées d'une priorité ; une tâche pouvant être spécifiée comme « préemptible » ou « non préemptible ». Une tâche préemptible peut être suspendue (mise à l'état « ready ») au profit d'une tâche de priorité plus élevée ou d'une interruption. Une tâche non préemptible ne peut être suspendue qu'au profit d'une interruption. Le temps qui lui est accordé est plus long, et l'attente dans la file d'attente plus courte.

Au fur et à mesure de l'évolution des systèmes d'exploitation, les concepteurs ont quitté la logique binaire « préemptible / non préemptible » au profit de systèmes plus fins de priorités multiples. Le principe est conservé, mais les priorités des programmes sont échelonnées. Dans les système RTOS même si les tâches du noyau du système doivent être préemptibles

Pendant la préemption, l'état du processus (drapeaux, registres et pointeur d'instruction) est sauvé dans la mémoire. Il doit être rechargé dans le processeur pour que le code soit exécuté de nouveau : c'est la commutation de contexte. Les différents états des tâches sont décrits dans la Fig. 4.1.1

Un système d'exploitation préemptif conserve en permanence la haute main sur les tâches exécutées par le processeur, contrairement à un système d'exploitation non préemptif, ou collaboratif, dans lequel c'est le processus en cours d'exécution qui prend la main et est seul juge du moment où il la rend. L'avantage le plus évident d'un système préemptif est qu'il peut en permanence décider d'interrompre un processus, principalement si celui-ci échoue et provoque l'instabilité du système.

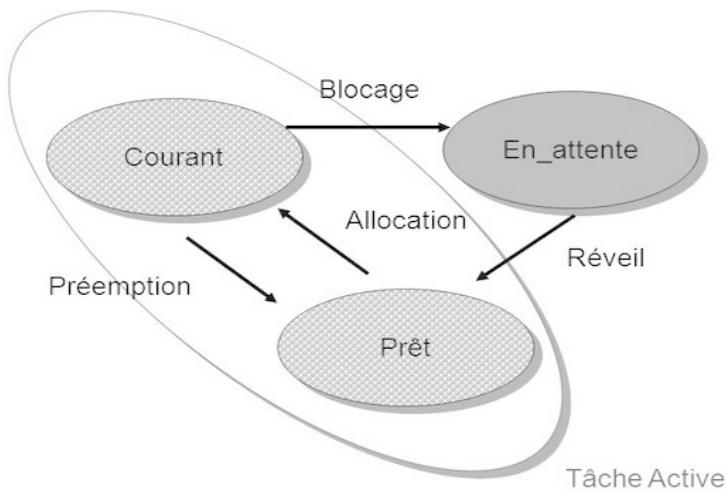


Figure 4.1.1 – Les différentes états d'une tâche et les transitions entre elles

Un ordonnanceur préemptif présente l'avantage d'une meilleure réactivité du système et de son évolution, mais l'inconvénient vient des situations de compétition (lorsque le processus d'exécution accède à la même ressource avant qu'un autre processus (préempté) ait terminé son utilisation), dans ce cas il se produit une inversion de priorité.

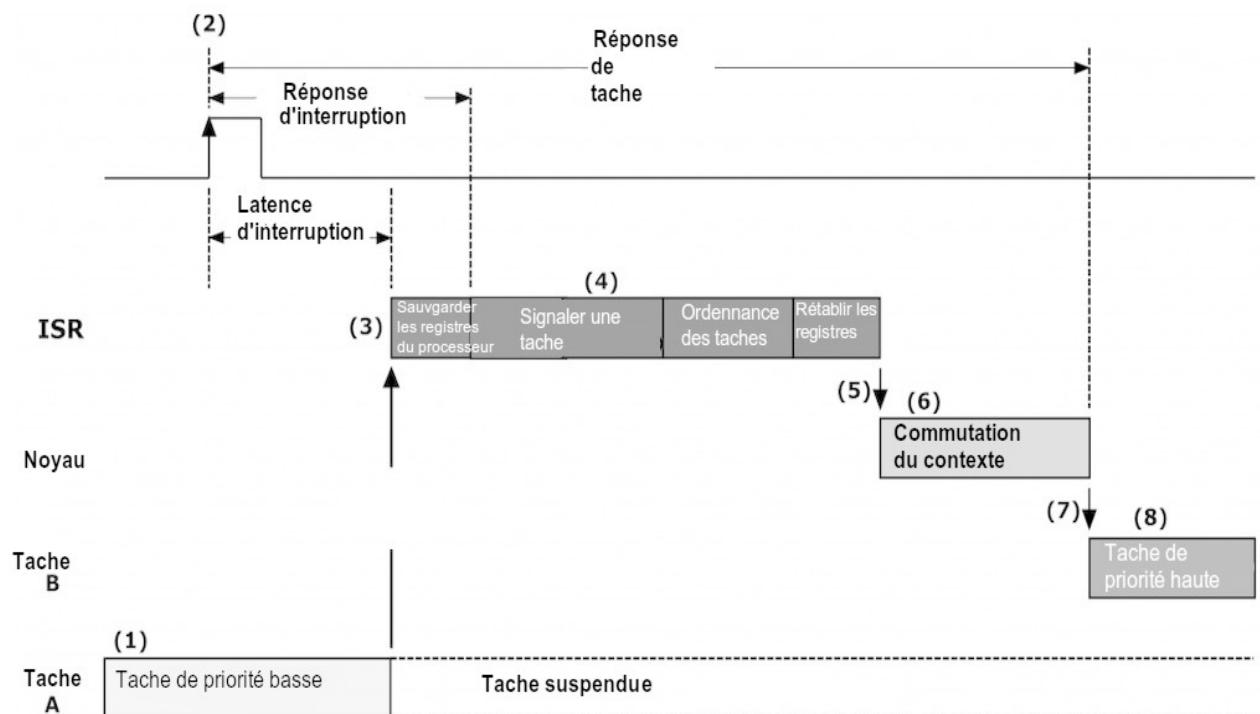


Figure 4.1.2 – Schéma de fonction d'un ordonnanceur préemptif

4.2 Inversion et héritage de priorité

L'inversion de priorité est un phénomène qui peut se produire en programmation concurrente. Il s'agit d'une situation dans laquelle un processus de haute priorité ne peut pas avoir accès au processeur car il est utilisé par un processus de plus faible priorité.

La non gestion de l'inversion de priorité peut avoir des effets désastreux. En effet, comme la non gestion de l'inversion de priorité implique qu'une tâche de haute priorité peut ne pas s'exécuter, il est possible qu'une réaction à des situations d'urgence ne soit pas prise en compte – Fig. 4.2.1.

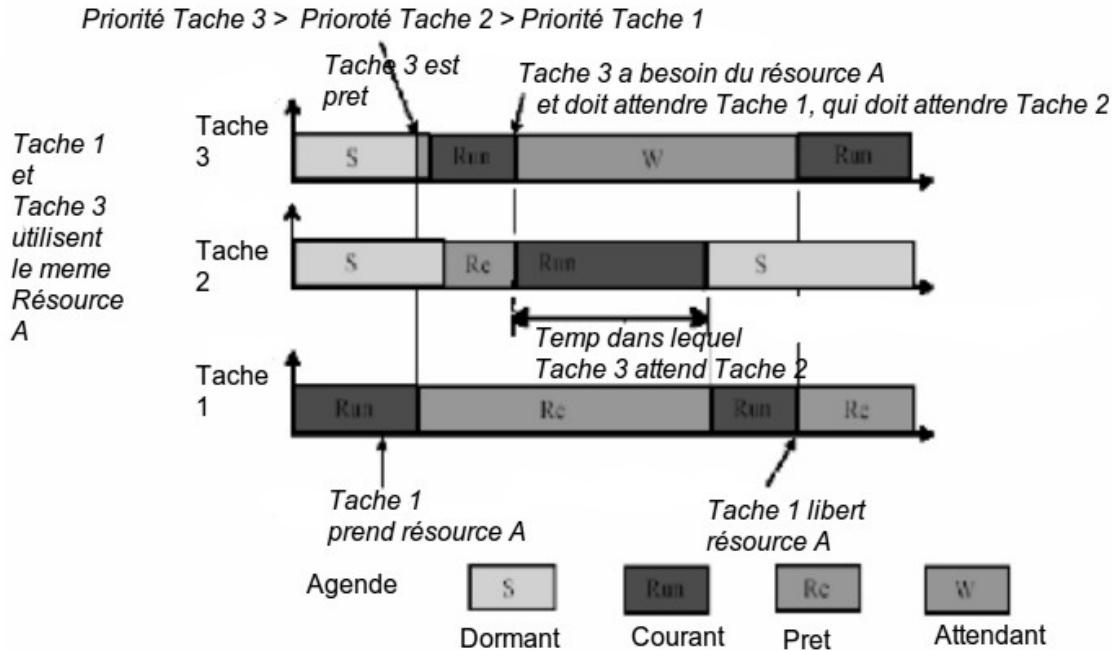


Figure 4.2.1 Schéma de l'inversion priorités

Il n'existe pas de solution simple permettant d'éviter toutes les inversions de priorité. Il est néanmoins possible de prendre des mesures pour limiter ces risques. En particulier, il est possible de

- n'autoriser l'accès à des sections critiques qu'à des threads de même priorité ;
- utiliser des sémaphores adaptés, par exemple des sémaphores à héritage de priorité ou des sémaphores à priorité plafond – Fig. 4.2.2.

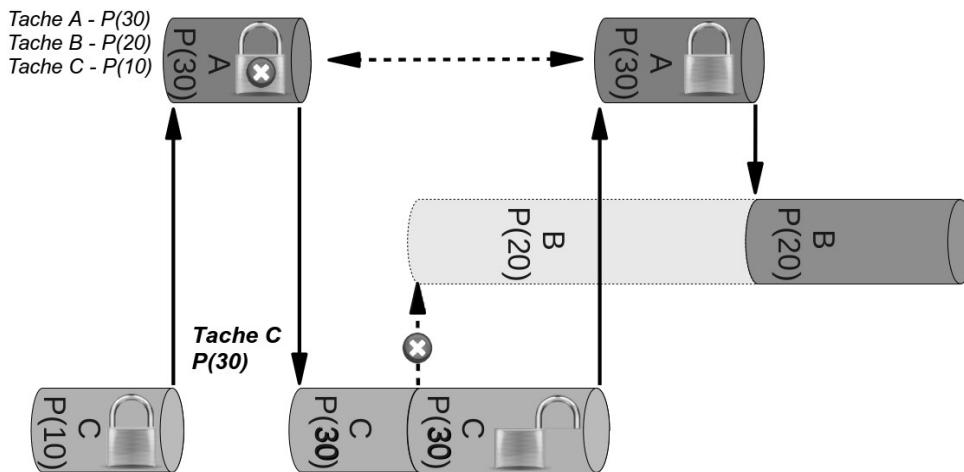


Figure 4.2.2 Schéma de l'héritage de priorité

4.3 Conception

Un RTOS facilite la création d'un système temps réel, mais ne garantit pas que le résultat final respecte les contraintes temps réel, ce qui exige le développement correct du logiciel. Par exemple on doit faire attention que les tâches, qui on doit exécuter en boucle pour interruption du processeur, ne prennent plus du temps que le période de l'interruption elle même.

Un RTOS utilise des ordonnanceurs spécialisées afin de fournir aux développeurs des systèmes temps réel les outils et les primitives nécessaires pour produire un comportement temps réel souhaité dans le système final.

Deux types de conceptions des RTOS existent :

A) Événementielle (ordonnancement par priorité) : l'ordonnanceur ne change de tâche que lorsqu'un événement de plus haute priorité a besoin de service – Fig. 4.3.1

Se sont des systèmes, qui comptent sur le principe de préemption selon la priorité des tâches pour assurer son fonctionnement déterministe.

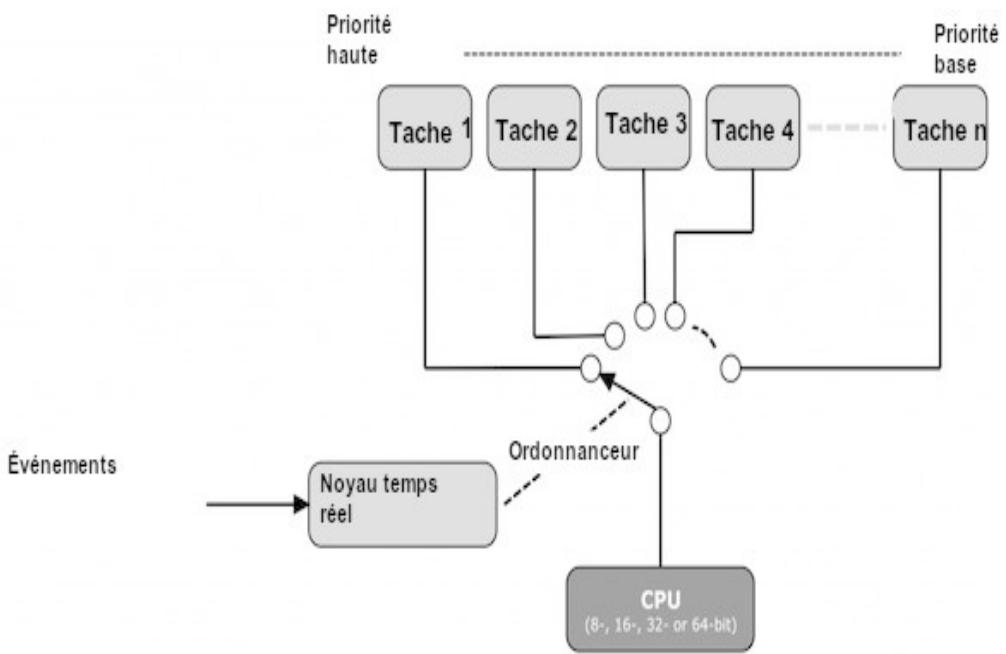


Figure 4.3.1 Schéma de fonctionnement d'ordonnanceur par priorité

Chaque tâche possède une priorité, plus la tâche est importante plus sa priorité doit être élevée.

Priorité statique :

la priorité de chaque tâche ne change pas durant l'exécution. Chaque tâche reçoit une priorité fixe lors de sa création. Toutes les tâches du système et leurs contraintes temporelles sont connues à la compilation.

Priorité dynamique :

la priorité de chaque tâche peut changer durant l'exécution, chaque tâche peut changer sa priorité durant l'exécution. Cette caractéristique des noyaux temps-réel permet d'éviter l'inversion de priorité. Expresso permet ce changement de priorité.

B) Par partage des tâches : L'ordonnanceur change de tâche aux interruptions de l'horloge, et lors des événements. Se méthode est appelé Round Robin

Une petite unité de temps, appelé « time quantum » ou « time slice », est définie. La ready queue est gérée comme une file circulaire. Le ordonnanceur parcourt cette file et alloue le processeur à chacun des processus pour une intervalle de temps de l'ordre d'un quantum au maximum.

La performance de round-robin dépend fortement du choix du quantum de base. Si le quantum est de 4 ms et qu'il faut 1 ms pour changer de processus, on perd donc $1/(4+1) = 20\%$ du temps en changement (exemple de quantum trop court par rapport au temps de changement).

Si le quantum est de 4 ms et que le processus met 2 ms à s'exécuter, on perd $(2+1)/(4+1) = 60\%$ du temps (exemple de quantum trop long par rapport au temps d'exécution). En général le quantum de temps est défini en fonction du comportement statistique des processus. L'idée est de définir un quantum de temps qui fait que les processus vont à 80 % finir leur utilisation du processeur avant la fin du quantum de temps. Ainsi il n'y a que peu de perte d'efficacité.

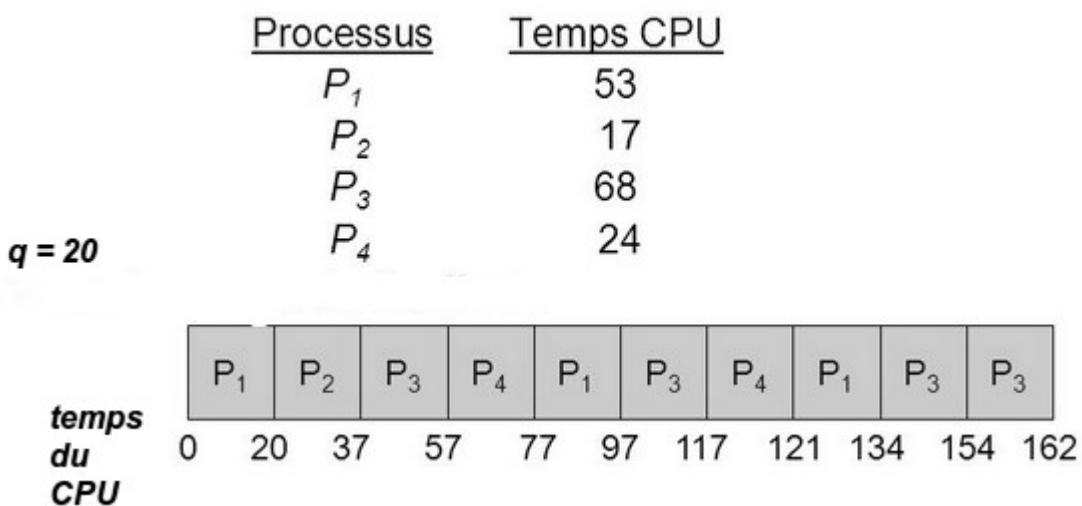


Figure 4.3.2 Exemple du fonctionnement des RTOS par partage des tâches

La conception par partage de tâche change de tâche plus souvent que c'est strictement nécessaire mais donne un caractère plus doux, plus déterministe au multitâche, donnant l'illusion à un processus ou à un utilisateur qu'il est le seul utilisateur de la machine.

Les premières conceptions de processeur avaient besoin de beaucoup de cycles pour changer de tâche, durant lesquels le processeur ne pouvait rien faire d'utile. Ainsi, les premiers RTOS essayaient de limiter le gaspillage de temps CPU en évitant au maximum les permutations de contexte.

Les plus récents processeurs utilisent largement moins de temps pour permuter de contexte. Le cas extrême est le Barrel processeur qui commute d'une tâche à l'autre en zéro cycle. Les plus récents RTOS implémentent invariablement l'ordonnancement par partage de tâche avec un ordonnancement par priorité.

4.4 Noyau temps réel

Les noyaux temps réel sont fonctionnellement spécialisés. Ce sont des noyaux généralement assez légers qui ont pour fonction de base stricte de garantir les temps d'exécution des tâches.

Les noyaux temps réel peuvent adopter en théorie n'importe quelle architecture. Ils fournissent souvent deux interfaces séparées, l'une spécialisée dans le temps réel et l'autre - générique. Les applications temps réel font alors appel à la partie temps réel du noyau.

Une des architectures souvent retenue est un noyau hybride qui s'appuie sur la combinaison d'un micro-noyau temps réel spécialisé, allouant du temps d'exécution à un noyau de système d'exploitation non spécialisé – Fig. 4.4.1. Le système d'exploitation non spécialisé fonctionne en tant que service du micro-noyau temps réel. Cette solution permet d'assurer le fonctionnement temps réel des applications, tout en maintenant la compatibilité avec des environnements préexistants.

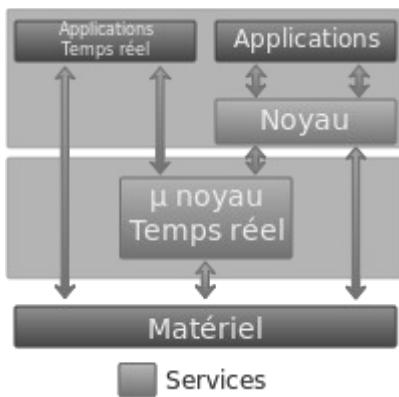


Figure 4.4.1 Schéma d'un noyau temps réel hybride

Par exemple, on peut avoir un micro-noyau temps réel allouant des ressources à un noyau non temps réel tel que Linux (RTLinux, RTAI, Xenomai) ou Windows (RTX). L'environnement GNU (resp. Windows) peut alors être exécuté à l'identique sur le noyau pour lequel il a été conçu, alors que les applications temps réel peuvent faire directement appel au micro-noyau temps réel pour garantir leurs délais d'exécutions.

VxWorks est un noyau propriétaire temps réel très implanté dans l'industrie bien que les systèmes à base de noyau Linux se déplient énormément et aient un succès grandissant via RTAI et Xenomai (RTLinux étant breveté).

5. Linux comme système d'exploitation temps réel

Par définition Linux n'est pas un système d'exploitation temps réel. C'est bien connues que les système temps réels ont une mauvaise influence sur le performance, l'efficacité, la vie de la batterie, et pour ces raison on doit faire beaucoup des compromis pour assurant le déterminisme du système.

De toute façon il y a beaucoup des solutions pour faire le système Linux exécuter une application temps réel. Dans cette chapitre on prestera seulement trois façons :

- Changement de la priorité du processus
- Installation du noyau avec des patches temps réel
- Utilisation de Xenomai – un sous noyau entre le matériel et le noyau Linux

5.1 Priorité d'un processus

Pour assurer que l'application temps réel soit favorisée par le noyau concernant le temps du processeur alloué, on doit configurer un priorité maximal du processus de l'application.

Chaque processus se voit affecter une priorité qui correspond à un numéro. Lorsqu'une commande est lancée, le processus a une priorité maximale. Plus le processus occupe de temps d'exécution pour le processeur, plus son numéro de priorité baisse, et moins le processus occupe de temps d'exécution pour le processeur, plus son numéro de priorité augmente. Ainsi, plusieurs processus peuvent être exécutés en même temps. La commande nice permet de diminuer la priorité du processus (pour les commandes longues et peu urgentes, par exemple). Le paramètre spécifié après l'option -n est un nombre compris entre 0 et 20 qui indique le facteur de diminution :

```
sudo nice -n 20 find / -type f -name "install*" -print  
> liste 2> /dev/null &
```

L'administrateur système (compte root) peut également augmenter la priorité avec un nombre négatif :

```
sudo nice -n -20 find / -type f -name "install*" -print  
> liste 2> /dev/null &
```

et la commande renice permet de changer le facteur de priorité en cours d'exécution de la commande, en spécifiant le nouveau facteur de priorité et le numéro de processus :

```
renice 10 733
```

Le résultat informe alors le super utilisateur du changement :

```
733: old priority 19, new priority 10
```

On peut quand même changer la priorité par l'interface UI du programme System Manager – Fig. 5.1.1. On doit sélectionner un processus et pousser le bouton droit de la souris. Puis un choisir « ChangeProperty » → « VeryHigh » - Fig. 5.2.2

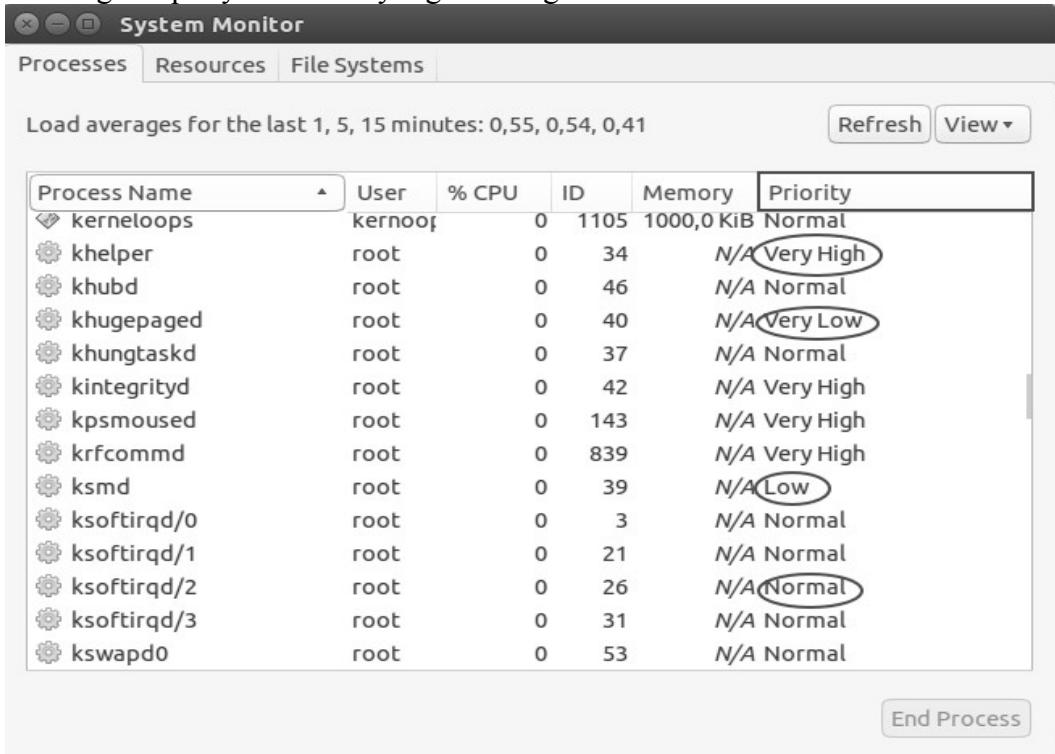


Figure 5.1.1 Photo instantanée des priorités différentes des processus

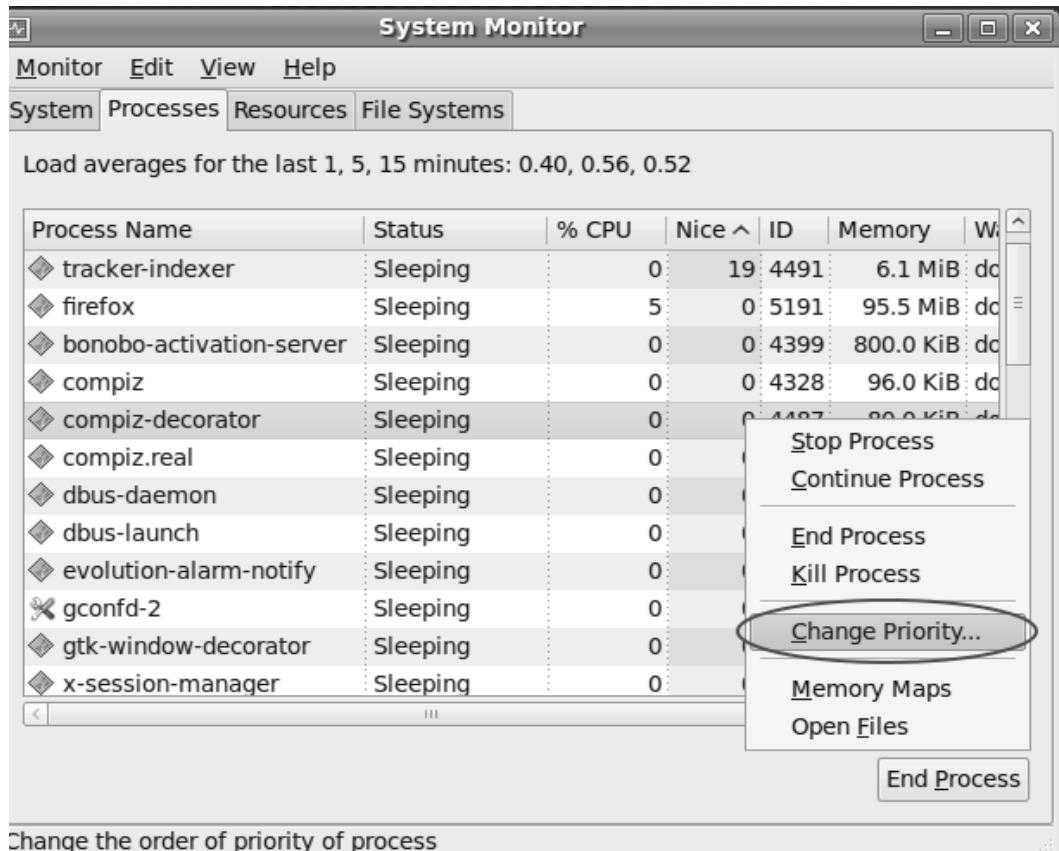


Figure 5.1.2 Photo instantanée du changement des priorités

5.2 Patches temps réel PREEMPT-RT

Linux-rt (où RT signifie « Real Time » → « temps réel ») est une branche du noyau Linux initiée dans le but de répondre aux contraintes d'un système temps réel.

L'application du patch officiel PREEMPT-RT sur le noyau Linux standard lui donne des fonctionnalités temps réel.

Un tel noyau est par exemple fourni en option par la distribution Ubuntu.

Il agit en rendant préemptible la majeure partie du code du noyau, et en particulier les sections critiques, les gestionnaires d'interruptions – Fig 5.2.1. Il modifie par ailleurs certains mécanismes pour réduire les temps de latence induits par le fonctionnement du système.

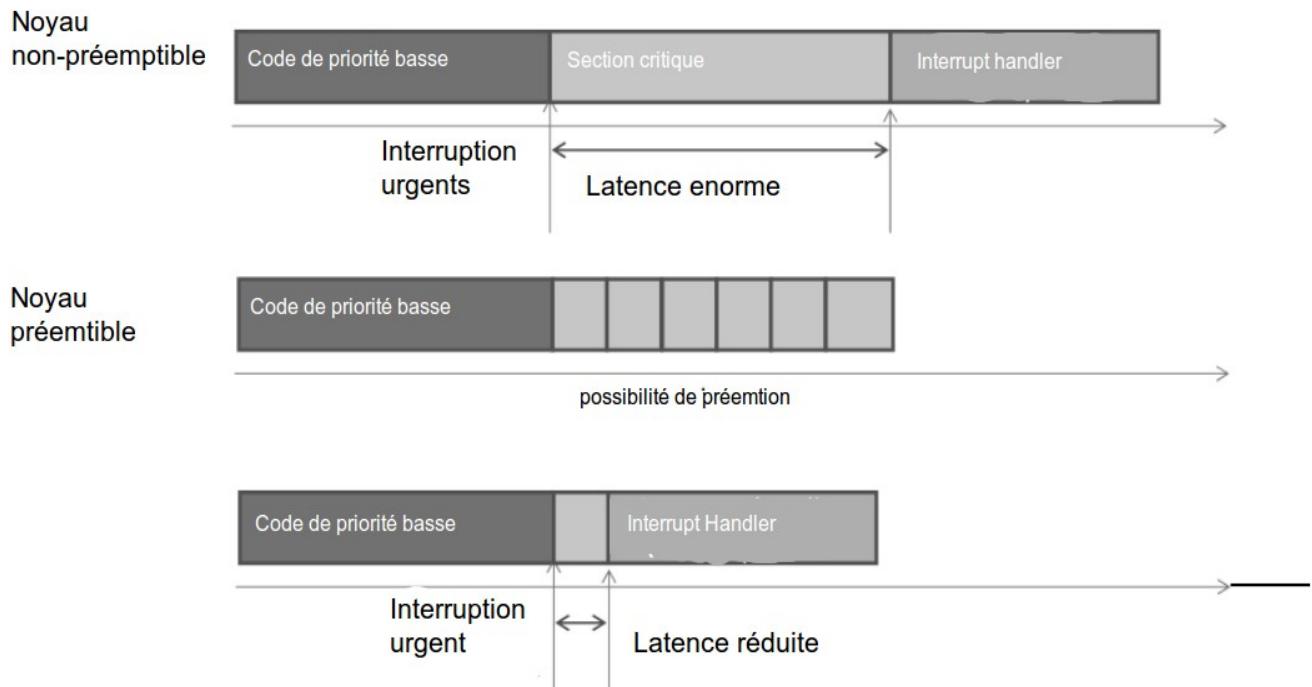


Figure 5.2.1 Comparaison entre noyau préemptible et pas préemptible

Ce patch met aussi en place un mécanisme de protection contre le problème connu sous le nom d'"inversion de priorité", par l'utilisation de sémaphore à héritage de priorité.

Par rapport à des extensions concurrentes du noyau Linux tels que Xenomai ou RTAI, il ne fait que modifier le fonctionnement du noyau standard sans ajouter un second noyau ou une couche de virtualisation temps réel, ce qui simplifie et allège le système résultant.

Il n'ajoute aucune interface de programmation spécifique, utilisant l'API POSIX standard, et ne requiert par là même aucune modification d'une application existante. Un programme prévu pour fonctionner sur un noyau Linux conventionnel fonctionnera donc naturellement sur linux-rt et en tirera immédiatement certains bénéfices (temps de latence réduits) sans aucune re-compilation.

Le patch préempte a pour effet de donner au noyau Linux un comportement temps réel dur, mais en limitant au maximum le nombre de modifications apportées. Une partie des fonctionnalités ajoutées ont depuis été introduites directement dans le noyau – l'évolution de ses relations en détail – par version de noyau, est présentée dans le tableau au-dessous – Tableau 1.

Tableau 1. Fonctionnalités temps-réel dans les versions différentes du noyau Linux

Architecture	x86	x86/64	powerpc	arm	mips	68knommu
Fonctionnalité						
Ordonnanceur déterministe	●	●	●	●	●	●
Multitâche préemptif	●	●	●	●	●	●
PI Mutexes (priorité héritable)	●	●	●	●	●	● ³
Timer Haute Résolution(précise autant que la matériel permet)	●	● ¹	● ¹	● ¹	● ¹	●
Read-Copy Update (RCU) préemptible	● ²					
IRQ Fils	● ⁴	● ^{3,4,5}				
Raw Spinlock Annotation	● ⁶					
IRQ Fils forcés	● ⁷					
R/W Sémaphore Cleanup	● ⁷					
Soutien complet de la Préemption	● ¹	● ³				

● Disponible dans mainline Linux

●¹ Disponible quand les patches temps-réel préempte sont installées

- 1) Depuis noyau 2.6.24 dans mainline Linux
- 2) Depuis noyau 2.6.25 dans mainline Linux
- 3) Patches temps-réel préempte 2.6.24.7-rt15
- 4) Depuis noyau 2.6.30 dans mainline Linux
- 5) Pas encore adapté pour le code d'interruption générique
- 6) Depuis noyau 2.6.33 dans mainline Linux
- 7) Depuis noyau 2.6.39 dans mainline Linux

On peut voir que une grande parties des fonctionnalités ajoutées ont été introduites dans le noyau, mais ça ne se passe à la cause de la productivité d'ordinateur. En plus la latence maximale du noyau générique et toujours très grande -. On peut comparer le performance en utilisant un teste nommé cyclictest, qui sert à analyser le performance des système d'exploitation temps réel– Fig. 5.2.2, 5.2.3

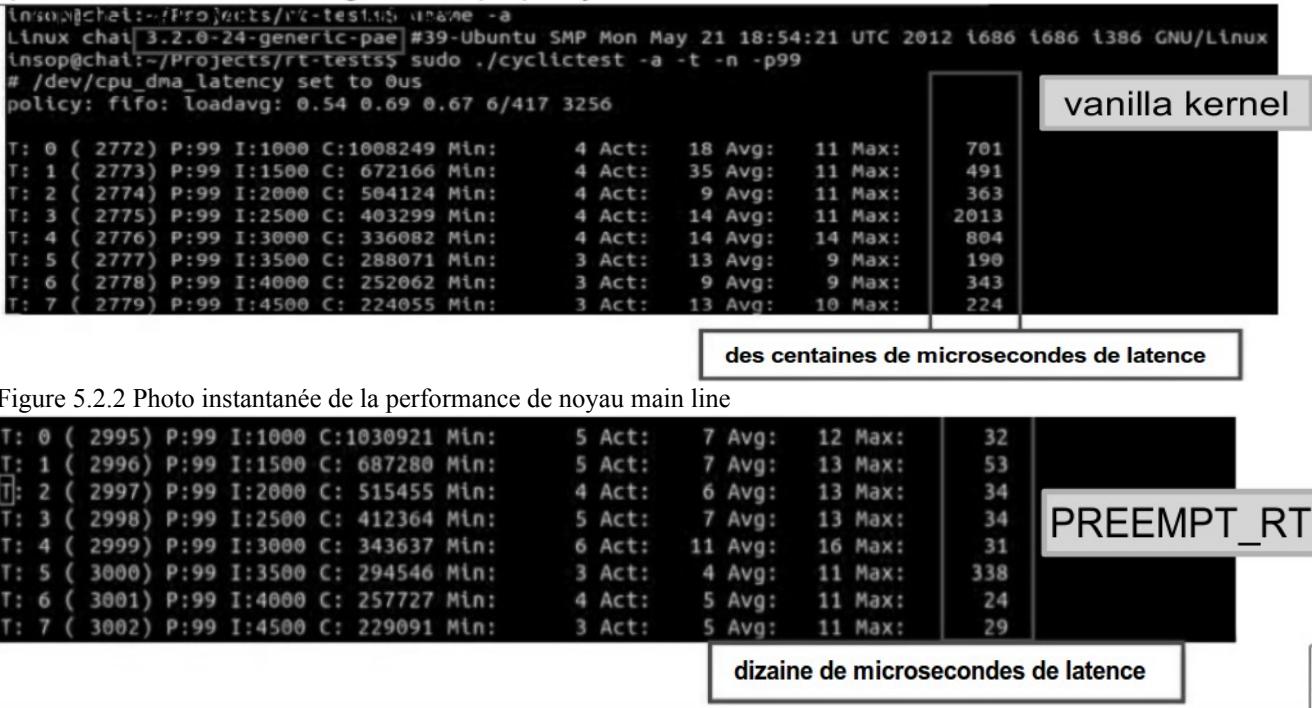


Figure 5.2.2 Photo instantanée de la performance de noyau main line



Figure 5.2.2 Photo instantanée de la performance de noyau préempte

Le principe du patch est d'ajouter systématiquement des occasions d'appel à l'ordonnanceur et donc de minimiser le temps entre la réception d'un événement et l'appel à la fonction schedule(). Les résultats des tests montrent que le patch préempte est effective, mais pas suffisamment pour répondre aux besoins d'un système temps réel dur.

Installation

Pour ajouter les fonctionnalités temps réel à un système Linux, il faut appliquer un patch aux sources du noyau. Cette manipulation n'est pas destinée à introduire les notions temps réel dans le noyau, mais simplement à ajouter un module. Une fois compilé, le noyau est plus ou moins identique à la version non patchée, à part que il est prêt à donner un support temps réel.

Pour réaliser l'instalation on a besoin de l'ensemble des outils de compilation d'installés, c'est à dire les traditionnels gcc, make. On a besoin d'avoir un accès root et vous devez avoir déjà une bonne connaissance de Linux et des commandes de base.

Il convient au préalable de récupérer le noyau Linux vanilla ainsi que le patch PREEMPT-RT en corrélation avec la version du noyau vanilla. Pour savoir quelle version de noyau Linux vanilla il faut choisir en fonction de la version de PREEMPT-RT, le patch PREEMPT-RT possède un nom générique du style patch-version_linux-rtversion_preempt-rt.bz2, par exemple patch-2.6.33.9-rt31.bz2. Sur l'exemple, il s'agit de la version PREEMPT-RT 31 en corrélation avec la version 2.6.33.9 du noyau Linux vanilla.

Voici sont des actions pas à pas pour installer un patch préempte.

Cas général

- 1 – Téléchargement du nouveau patch Linux-Preempt-RT

```
# cd ~/tmp/
# wget http://www.kernel.org/pub/linux/kernel/projects/rt/patch-2.6.33.9-rt31.bz2
```
- 2 – Téléchargement du noyau Linux 2.6.33.9

```
# wget http://www.kernel.org/pub/linux/kernel/v2.6/longterm/v2.6.33/linux-2.6.33.9.tar.bz2
```
- 3 – Décompression des sources de Linux et du patch

```
# tar xjf linux-2.6.33.9.tar.bz2
# bunzip2 patch-2.6.33.9-rt31.bz2
```
- 4 – Application du patch Preempt-RT

```
# cd linux-2.6.33.9
# patch -p1 < ../patch-2.6.33.9-rt31
```
- 5 – Compilation du noyau

```
# cp ./config-linux-2.6.33-rt ./.config
# make ARCH=arm menuconfig
# make ARCH=arm CROSS_COMPILE=/cross-arm-linux/usr/bin/arm-linux- uImage -j16
```
- 6 – Installation du noyau compilé

```
# cp arch/arm/boot/uImage /media/boot/
# umount /media/boot/
```
- 7 – Test du nouveau noyau

```
# uname -a
```

Ubuntu

- Installation du logiciel supplémentaire

```
sudo apt-get install kernel-package fakeroot build-essential libncurses5-dev
```
- Téléchargement du noyau vanilla et RT patch

```
mkdir -p ~/tmp/linux-rt
cd ~/tmp/linux-rt
wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.tar.bz2
wget http://www.kernel.org/pub/linux/kernel/projects/rt/3.4/patch-3.4-rt7.patch.bz2
tar xjvf linux-3.4.tar.bz2
cd linux-3.4
patch -p1 <<(bunzip2 -c ../patch-3.4-rt7.patch.bz2)
```
- Configuration du noyau

```
cp /boot/config-$(uname -r) .config && make oldconfig → Ici on doit choisir "full preemption" (option 5)
```
- Compilation du noyau

```
sed -rie 's/echo "+"/#echo "+"/' scripts/setlocalversion
make-kpkg clean
CONCURRENCY_LEVEL=$(getconf _NPROCESSORS_ONLN) fakeroot make-kpkg --initrd --revision=0
kernel_image kernel_headers
```
- Installation du noyau

```
sudo dpkg -i ./linux-{headers,image}-3.4.0-rt7_0_*.deb
```

5.3 Xenomai

Le noyau temps réel auxiliaire. Les promoteurs de cette technologie considèrent que le noyau Linux ne sera jamais véritablement temps réel et ajoute donc à ce noyau un autre noyau disposant d'un véritable ordonnanceur temps réel à priorités fixes. Ce noyau auxiliaire traite directement les tâches temps réel et délègue les autres tâches au noyau Linux, considéré comme la tâche de fond de plus faible priorité – Fig. 5.3.1. Cette technique permet de mettre en place des systèmes temps réel «durs».

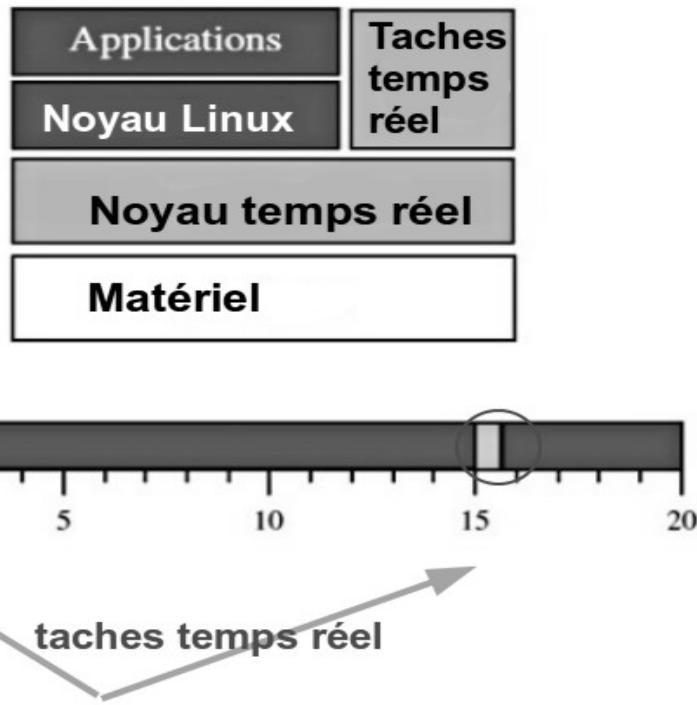


Figure 5.3.1 Schéma de fonctionnement du noyau Xenomai

Au sein d'Adeos, chaque noyau est représenté par une entité appelée « domaine » qui surveille les événements circulant sur le I-pipe – Fig. 5.3.3. Adeos (Adaptive Domain Environment for Operating System) est une couche de virtualisation de ressources matérielles permettant de faire cohabiter plusieurs noyaux sur une même machine physique – Fig. 5.3.2 . Adeos existe sous la forme d'un patch à appliquer au noyau Linux et permettant de le faire cohabiter avec un RTOS comme Xenomai.

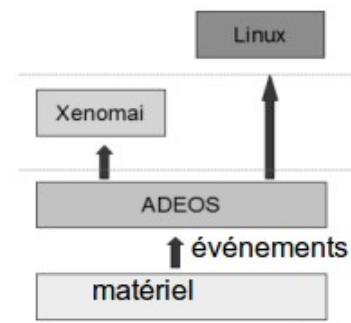


Figure 5.3.2 Schéma du fonctionnement de ADEOS

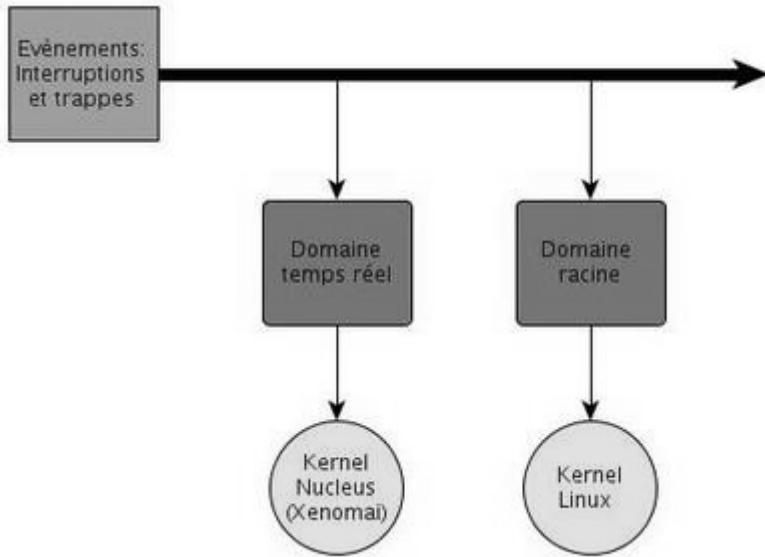


Figure 5.3.3 Schéma du fonctionnement du I-Pipe

Xenomai 2 donne l'opportunité d'utiliser un débogueur classique comme GDB. On peut dire que, du point de vue de l'utilisateur, Xenomai rend obsolète la notion de « double programmation » - la programmation séparé des applications temps réel. Cependant, la programmation temps réel dans l'espace du noyau reste toujours possible.

Chaque domaine du pipeline peut avoir besoin d'être notifié de ce qu'il se passe sur d'autres domaines. Pour cela, Adeos met en place un système de notification inter-domaines. Ces notifications sont utilisées, par exemple, pour avertir de chaque appel système ou de chaque événement déclenché par le kernel Linux lui-même. Cela comprend donc les syscall, le réordonnancement, la création ou la destruction de processus, les différentes fautes d'exécution, etc. Plus généralement, ce sont des événements logiciels.

Un domaine doit pouvoir différer le traitement d'une interruption venant du I-pipe afin de ne pas interrompre son noyau. Le domaine est dit bloqué.

Le blocage du domaine est réalisé par la fonction `ipipe_stall_pipeline_from()` qui vient positionner à 1 le masque d'interruption logiciel (bit 0 du champ `status`).

Cela permet de ne plus avoir besoin de bloquer matériellement les interruptions arrivant sur le système. Ainsi le domaine racine peut être bloqué sans empêcher l'arrivée d'interruptions sur le domaine temps réel. Cependant, pour ne pas perturber l'exécution du noyau temps réel, les interruptions matérielles sont tout de même désactivées pendant le traitement par celui-ci.

Le patch Adeos ré-implémente les fonctions du noyau Linux qui servent normalement à désactiver ou réactiver les interruptions pour venir bloquer ou débloquer le domaine racine.

Lorsque le domaine racine est bloqué, les interruptions ne progressent plus sur le pipeline et sont stoppées à son niveau. Chaque interruption reçue est sauvegardée dans le i-log.

Par exemple, le noyau Linux bloque son domaine pendant le traitement d'une première interruption

(contexte d'interruption). Une seconde interruption peut arriver sur le processeur car les interruptions ne sont pas désactivées sur le processeur. Cette interruption sera mise dans le i-log lorsqu'elle aura atteint le domaine racine.

XENOMAI permet de créer de tâches temps réel dans l'espace utilisateur et non plus uniquement dans l'espace noyau comme auparavant avec RTLinux ou RTAI (autre technologies des noyaux doubles). Xenomai permis une exécution simple et bien performante des tâches temps-réel dans l'espace utilisateur. La cohérence des priorités entre l'espace temps-réel Xenomai et la classe temps-réel Linux est conservée, mais il y a une possibilité de migration transparente des tâches entre ces espaces. Au-delà de ces aspect, c'est le niveau d'intégration du sous-système temps-réel avec le cœur standard Linux qui a été fortement augmenté. Xenomai permet ainsi d'exécuter un programme temps-réel dans un processus utilisateur.

L'un des objectifs de Xenomai est de permettre la migration aisée d'applications issues d'un environnement RTOS traditionnel vers un système GNU/Linux. L'architecture de Xenomai repose ainsi sur un nano-kernel proposant une API générique « neutre » utilisant les similarités sémantiques entre les différentes API des RTOS les plus répandus (fonctions de création et gestion de thread, sémaphores, etc.). Xenomai peut donc émuler des interfaces de programmation propriétaires en factorisant de manière optimale leurs aspects communs fondamentaux, en particulier dans le cas de RTOS spécifique (ou maison) très répandus dans l'industrie – Fig. 5.3.4 .

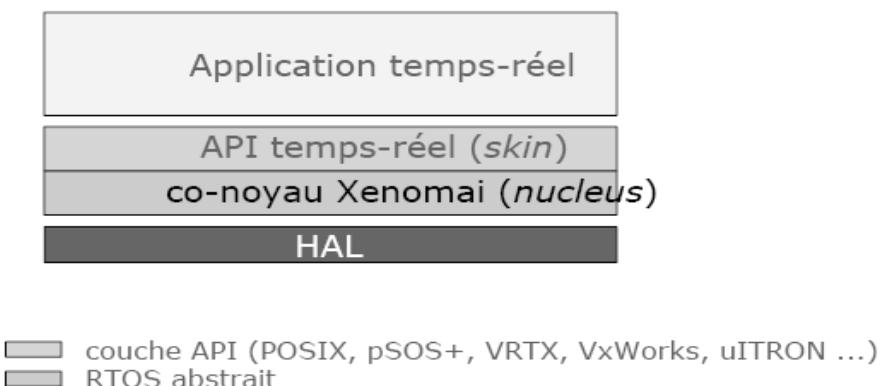


Figure 5.3.4 Structure du système utilisant la solution Xenomai

Xenomai définit sa propre interface (skin) native exploitant au mieux les capacités du système temps-réel qu'il introduit, dont les principales classes de service sont:

- Gestion de tâches
- Objets de synchronisation
- Messages et communication
- Entrées/sorties
- Allocation mémoire

- Gestion du temps et des alarmes
- Interruptions

Xenomai fonctionne sur les architectures suivantes :

- ARM ;
- Blackfin (DSP d'Analog Devices) ;
- i386 ;
- ia64 (Intel Itanium) ;
- Power PC ;
- x86_64 (Intel : Pentium 4, Pentium D, Pentium Extreme Edition, Celeron D, Xeon, Pentium Dual-Core ;
- AMD : Athlon 64, Athlon 64 FX, Athlon 64 X2, Turion 64, Turion 64 X2, Opteron, Sempron).

À l'aide des tests de performance on détermine la latence maximale en microsecondes pour des deux solutions présentées.

Configuration	Avg	Max	Min
XENOMAI	43	58	2
PREEMPT	88	415	27

Figure 5.3.5 Comparaison des fonctionnalités de Xenomai et le noyau préemptif

Avec l'apparition de solutions comme Xenomai 2, le développement d'applications temps réel durs sous Linux est grandement facilité. L'utilisation de l'espace utilisateur permet à la fois de respecter les contraintes des licences (GPL/LGPL) mais aussi de disposer des outils classiques de la programmation sous Linux.

De toute façon pour les cas où on peut faire des compromis avec le déterminisme la solution de patch préemptif est un choix convenable et simple.

6.Outil

6.1 Qt

Qt est:

- une API orientée objet et développée en C++ par Qt Development Frameworks, filiale de Digia. Qt offre des composants d'interface graphique (widgets), d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc. ;
- par certains aspects, elle ressemble à un framework lorsqu'on l'utilise pour concevoir des interfaces graphiques ou que l'on conçoit l'architecture de son application en utilisant les mécanismes des **signaux et slots** par exemple.

Qt permet la portabilité des applications qui n'utilisent que ses composants par simple précompilation du code source. Les environnements supportés sont les Unix (dont GNU/Linux) qui utilisent le système graphique X Window System ou Wayland, Windows, Mac OS X et également Tizen – Fig. 6.1.1. Le fait d'être une bibliothèque logicielle multiplateforme attire un grand nombre de personnes qui ont donc l'occasion de diffuser leurs programmes sur les principaux OS existants. Qt supporte des bindings avec plus d'une dizaine de langages autres que le C++, comme Java, Python, Ruby, Ada, C#, Pascal, Perl, Common Lisp, etc.

Qt est notamment connu pour être le framework sur lequel repose l'environnement graphique KDE, l'un des environnements de bureau par défaut de plusieurs distributions GNU/Linux.

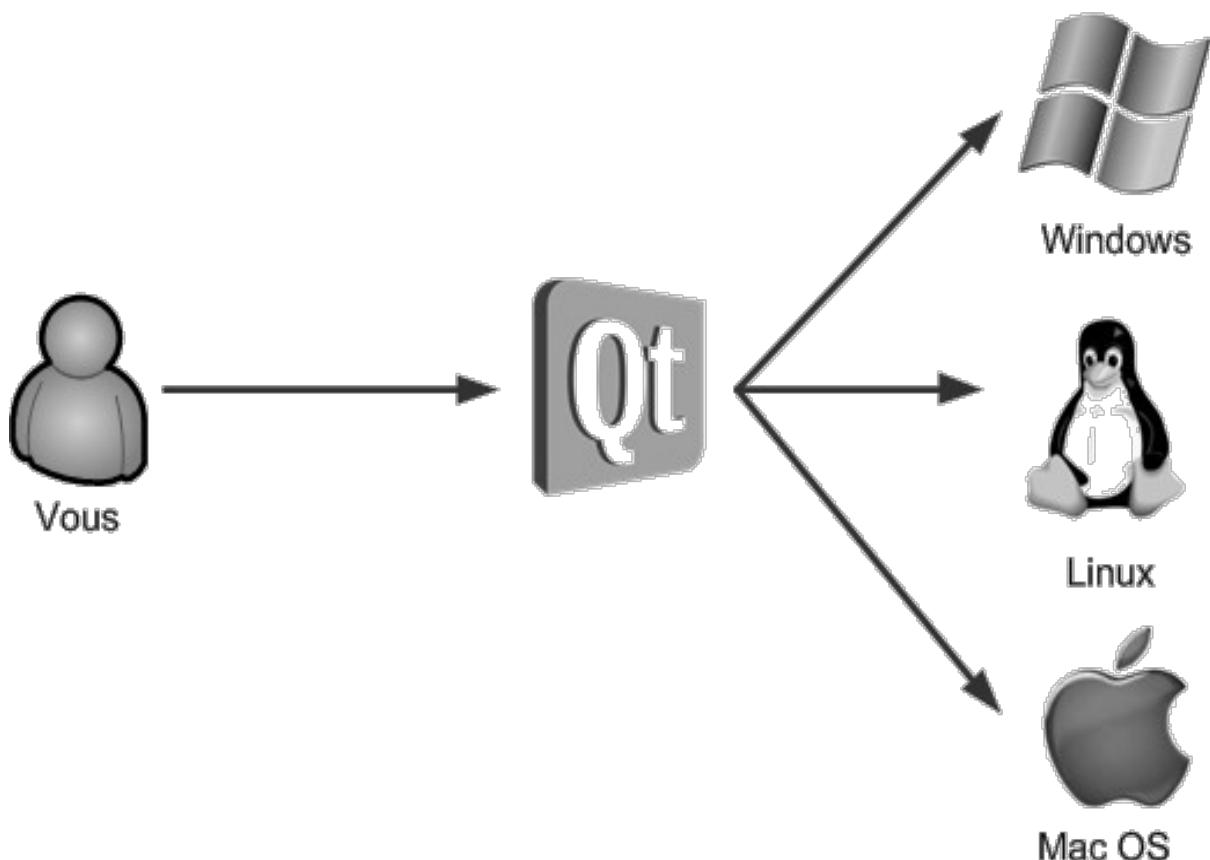


Figure 6.1.1 Visualisation de la multi compatibilité du QT

Structure générale

L'API Qt est constituée de classes aux noms préfixés par Q et dont chaque mot commence par une majuscule (ex: QLineEdit), c'est la typographie CamelCase. Ces classes ont souvent pour attributs des types énumérés déclarés dans l'espace de nommage Qt19. Mis à part une architecture en pur objet, certaines fonctionnalités basiques sont implémentées par des macros (chaîne de caractères à traduire avec tr, affichage sur la sortie standard avec qDebug...).

Les conventions de nommage des méthodes sont assez semblables à celles de Java : le lowerCamelCase est utilisé, c'est-à-dire que tous les mots sauf le premier prennent une majuscule (ex: indicatorFollowsStyle()), les modificateurs sont précédés par set, en revanche les accesseurs prennent simplement le nom de l'attribut (ex : text()) ou commencent par is dans le cas des booléens (ex : isChecked()).

Arborescence des objets

Les objets Qt (ceux héritant de QObject) peuvent s'organiser d'eux-mêmes sous forme d'arbre. Ainsi, lorsqu'une classe est instanciée, on peut lui définir un objet parent. Cette organisation des objets sous forme d'arbre facilite la gestion de la mémoire car avant qu'un objet parent ne soit détruit, Qt appelle récursivement le destructeur de tous les enfants.

Cette notion d'arbre des objets permet également de débugger plus facilement, via l'appel de méthodes comme QObject::dumpObjectTree() et Object::dumpObjectInfo() .

Compilateur de meta-objets

Le moc21 (pour Meta Object Compiler) est un préprocesseur qui, appliqué avant compilation du code source d'un programme Qt, génère des meta-informations relatives aux classes utilisées dans le programme. Ces meta-informations sont ensuite utilisées par Qt pour fournir des fonctions non disponibles en C++, comme les signaux et slots et l'introspection.

L'utilisation d'un tel outil additionnel démarque les programmes Qt du langage C++ standard. Ce fonctionnement est vu par Qt Development Frameworks comme un compromis nécessaire pour fournir l'introspection et les mécanismes de signaux. À la sortie de Qt 1.x, les implementations des templates par les compilateurs C++ n'étaient pas suffisamment homogènes22.

Signaux et slots

Les signaux et slots sont une implémentation du patron de conception observateur. L'idée est de connecter des objets entre eux via des signaux qui sont émis et reçus par des slots. Du point de vue du développeur, les signaux sont représentés comme de simples méthodes de la classe émettrice, dont il n'y a pas d'implémentation. Ces « méthodes » sont par la suite appelées, en faisant précéder « emit », qui désigne l'émission du signal. Pour sa part, le slot connecté à un signal est une méthode de la classe réceptrice, qui doit avoir la même signature (autrement dit les mêmes paramètres que le signal auquel il est connecté), mais à la différence des signaux, il doit être implémenté par le développeur. Le code de cette implementation représente les actions à réaliser à la réception du signal.

C'est le MOC qui se charge de générer le code C++ nécessaire pour connecter les signaux et les slots.

Concepteur d'interface

Qt Designer, le concepteur d'interface.

Qt Designer est un logiciel qui permet de créer des interfaces graphiques Qt dans un environnement

convivial. L'utilisateur, par glisser-déposer, place les composants d'interface graphique et y règle leurs propriétés facilement. Les fichiers d'interface graphique sont formatés en XML et portent l'extension .ui

Lors de la compilation, un fichier d'interface graphique est converti en classe C++ par l'utilitaire uic. Il y a plusieurs manières pour le développeur d'employer cette classe :

- l'instancier directement et connecter les signaux et slots
- l'agréger au sein d'une autre classe
- l'hériter pour en faire une classe mère et ayant accès ainsi à tous les éléments constitutifs de l'interface créée
- la générer à la volée avec la classe QUiLoader qui se charge d'interpréter le fichier XML .ui et retourner une instance de classe QWidget

qmake

Qt se voulant un environnement de développement portable et ayant le MOC comme étape intermédiaire avant la phase de compilation/édition de liens, il a été nécessaire de concevoir un moteur de production spécifique. C'est ainsi qu'est conçu le programme qmake.

Ce dernier prend en entrée un fichier (avec l'extension .pro) décrivant le projet (liste des fichiers sources, dépendances, paramètres passés au compilateur, etc.) et génère un fichier de projet spécifique à la plateforme. Ainsi, sous les systèmes UNIX qmake produit un Makefile qui contient la liste des commandes à exécuter pour génération d'un exécutable, à l'exception des étapes spécifiques à Qt (génération des classes C++ lors de la conception d'interface graphique avec Qt Designer, génération du code C++ pour lier les signaux et les slots, ajout d'un fichier au projet, etc.).

Le fichier de projet est fait pour être très facilement éditable par un développeur. Il consiste en une série d'affectations de variables. En voici un exemple pour un petit projet:

```
TARGET = monAppli
SOURCES = main.cpp mainwindow.cpp
HEADERS = mainwindow.h
FORMS = mainwindow.ui
QT += sql
```

Ces déclarations demandent que l'exécutable soit nommé monAppli, donne la liste des fichiers sources, en-têtes et fichiers d'interface graphique. La dernière ligne déclare que le projet requiert le module SQL de Qt.

QtCreator

Qt Creator est un environnement de développement intégré multiplate-forme faisant partie du framework Qt. Il est donc orienté pour la programmation en C++.

Il intégré directement dans l'interface un débogueur, un outil de création d'interfaces graphiques, des outils pour la publication de code sur Git et Mercurial ainsi que la documentation Qt. L'éditeur de texte intégré permet l'autocomplétion ainsi que la coloration syntaxique. Qt Creator utilise sous Linux le compilateur gcc. Il peut utiliser MinGW ou le compilateur de Visual Studio sous Windows.

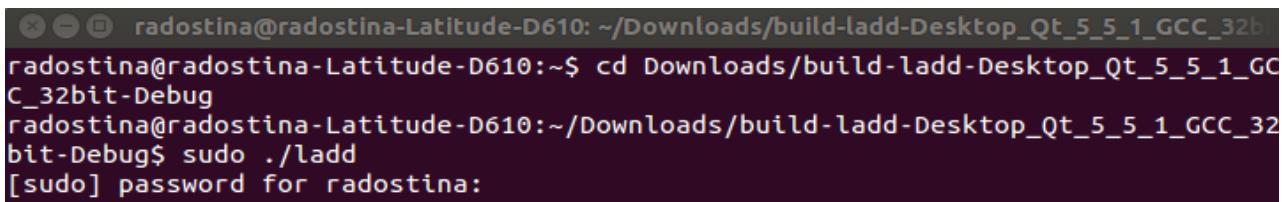
De plus en plus de développeurs utilisent Qt, y compris parmi de grandes entreprises. On peut notamment citer : Google, Adobe System, Skype ou encore la NASA. Le site de Digia recense les entreprises utilisant Qt et les applications basées sur Qt.

6.2. Serveur gdb

Dans Linux il existe la conception d'un utilisateur racine – root, qui possède toutes les permissions sur le système, aussi bien en mode mono qu'en mode multi-utilisateur. Ainsi, un tel utilisateur équivaut à un utilisateur suprême, doté de fonctions supérieures et d'accès privilégiés.

De toute façon s'est pas une bonne idée s'utiliser l'ordinateur comme un utilisateur root, parce que une erreur simple typographique peut entraîner des dommages graves. Au lieu de ça on utilisera la commande sudo (abréviation de substitute user do « faire en se substituant à l'utilisateur ») - une commande informatique utilisée principalement dans Linux pour obtenir les droites d'accès égaux à celles du utilisateur root. Pour utiliser la commande sudo on a besoin de savoir le mot de passe de l'utilisateur root.

Pour pouvoir accéder les ports série et parallèle on doit avoir des droites d'accès root. On utilise la commande sudo en démarrant le logiciel – Fig. 6.2.1 .



```
radostina@radostina-Latitude-D610: ~/Downloads/build-ladd-Desktop_Qt_5_5_1_GCC_32bit-Debug
radostina@radostina-Latitude-D610:~$ cd Downloads/build-ladd-Desktop_Qt_5_5_1_GCC_32bit-Debug
radostina@radostina-Latitude-D610:~/Downloads/build-ladd-Desktop_Qt_5_5_1_GCC_32bit-Debug$ sudo ./ladd
[sudo] password for radostina:
```

Figure 6.2.1 Photo instantanée de démarage du processus du programme avec le mot sudo

Dans cette façon on peut utiliser le logiciel, mais on ne peut pas le déboguer. Quand on essaie de déboguer le programme par le QtCreator une erreur se produit, parce que le Creator n'a pas accès sur les ports alors les programmes démarrées par lui n'ont pas ces droites d'accès aussi – Fig. 6.2.2 .

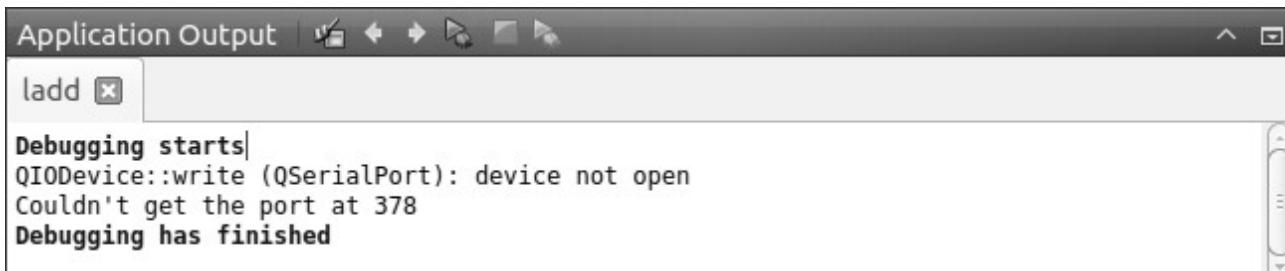


Figure 6.2.2 Photo instantanée d'erreur qui se produit en déboguant dans le QtCreator

On peut donner au QtCreator les droites root, pour que on le logiciel démarrée via lui on les mêmes droites, mais c'est une idée très mauvaise. Le QtCreator est un programme très lourd et extensif et constitue une menace pour la sécurité de l'ordinateur si il est lancé avec les droites root.

Pour pouvoir déboguer on utilise une combinaison des commandes sudo et gdbserver.

Le gdbserver est une commande Unix permettant de déboguer à distance (debug croisé) avec le GNU Debugger (le débogueur standard du projet GNU – gdb utiliser aussi par QtCreator).

Il ne requiert que la présence de l'exécutable sur la cible, les fichiers sources restent du côté de la machine utilisé par le développeur pour déboguer (avec tout de même une copie du binaire).

Dans le cas de ce projet (« Ladd ») on utilise la même machine pour exécuter et déboguer le logiciel. Dans la Fig. 6.2.3 on a présenté comment de créer le processus qui va être exécuter sur le serveur

```
radostina@radostina-Latitude-D610:~/Downloads/build-ladd-Desktop_Qt_5_5_1_GCC_32bit-Debug$ sudo gdbserver host:2346 ladd
Process ladd created; pid = 2541
Listening on port 2346
```

Figure 6.2.3 Photo instantanée de la création du processus « ladd » sur le gdb serveur

Puis on utilise le QtCreator, mais pas pour démarrer le logiciel. On attache le QtCreator au serveur exécutant le logiciel. Le chemin pour faire ça est Debug → Start Debugging → Attach to running debug server. Une fenêtre apparaît pour configurer les données du serveur et le programme exécuté.

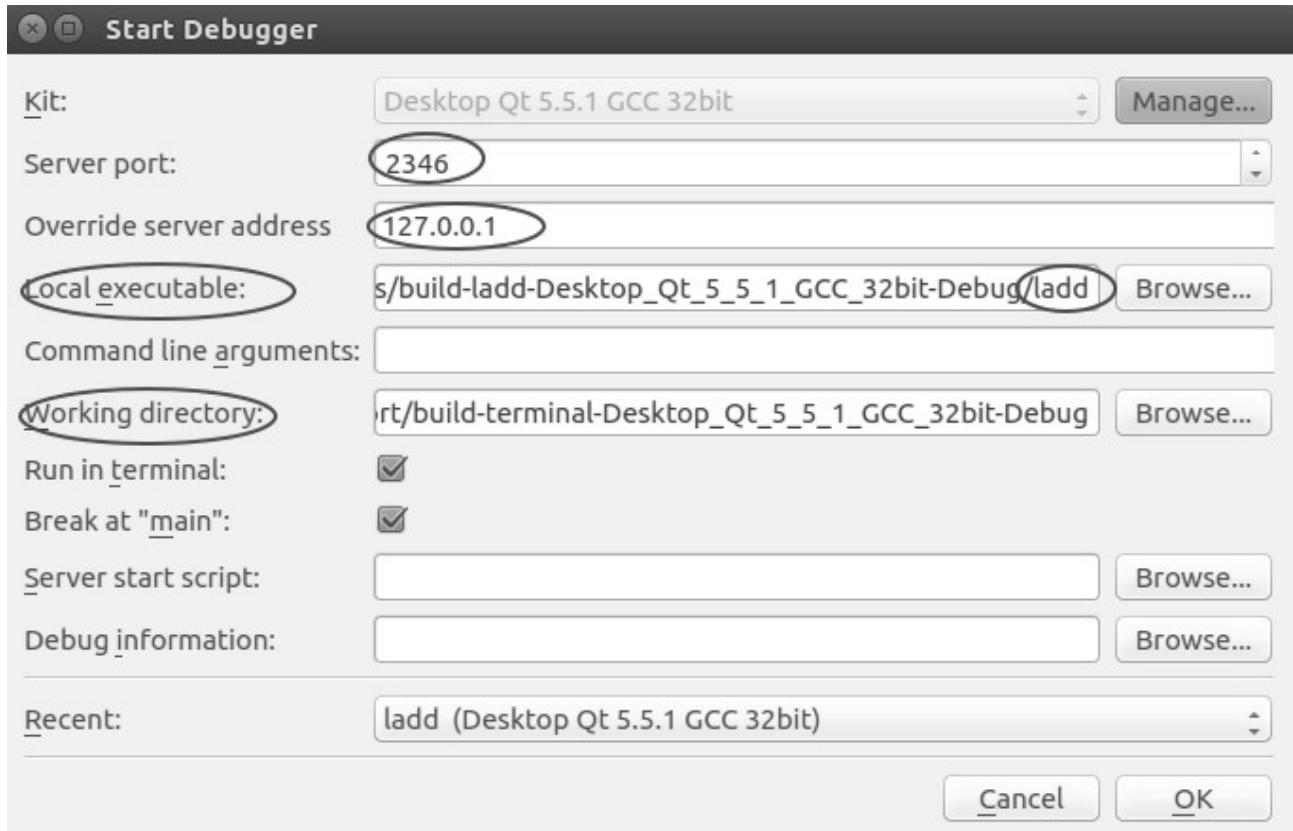


Figure 6.2.4 Photo instantanée du démarrage du débogueur connecté au gdbserver par le QtCreator

Étant donné que le serveur se trouve sur la même machine on va utiliser l'adresse localhost en faisant appel au serveur – 127.0.0.1 . Le numéro du port et le même – 2346 que dans la commande gdbserver. On donne aussi le chemin pour l'exécutable locale.

7. Conception du logiciel « Ladd »

7.1. Application

L'objectif du projet « Ladd » est d'aider de développer un logiciel de contrôle des processus temps réels dans l'industrie. Le logiciel doit communiquer avec un dispositif d'entrées et sorties, qui est lié au matériel de machine. Suivant une logique décrite dans le fichier *.LAD on calcule les réactions correspondantes aux données d'entrées initiales. Les réactions sont représentées en données d'être écrites sur les sorties du dispositif.

Dans ce manire on a dj raliser plusieurs machine d'automatisation des processus en utilisant le systme d'exploitation DOS et le langage C. Par exemple une machine de production d'alimentation pour les astronautes – Fig. 7.1.1.. Et une machine pour laboration d'une couverture sur les pices de type stent - Fig. 7.1.2 . Le stent est un dispositif, le plus souvent mtallique, maill et tubulaire, gliss dans une cavit naturelle humaine (ou animale) pour la maintenir ouverte. Il, s'agit donc, le plus souvent, d'un support artriel. Il est essentiellement utilis dans des artres au cours d'une angioplastie, dans le cas de la maladie coronarienne par exemple.



Figure 7.1.1 Image d'une soupe pour consommation par des astronautes



Figure 7.1.2. Image des différents types du stent

Le projet « Ladd » de sa part est réalisé utilisant le système d'exploitation Linux, le langage C++ et une architecture avec des classes et objet. On introduit aussi un fichier XML de configuration pour pouvoir configurer le logiciel à communiquer avec des dispositifs différents sans changement du code source – le nombre des bits peut varier de 0 à 8 et les valeurs booléennes initiales des bits d'entrée peuvent être soit 0 ou 1.

7.2. Structure principal

Dans la méthode main() du logiciel « Ladd » on initialise le MainController - c'est l'objet principal du logiciel. Dans ce contrôleur se trouve la logique essentielle du fonctionnement. Le MainController contient un référence aux deux objets principaux – au page du réglage (**SettingsPage**) et au contrôleur de communication (**CommunicationController**). Dans la Fig. 7.2.1 on a présenté l'architecture principale des classes, ses composants et les relations entre elles.

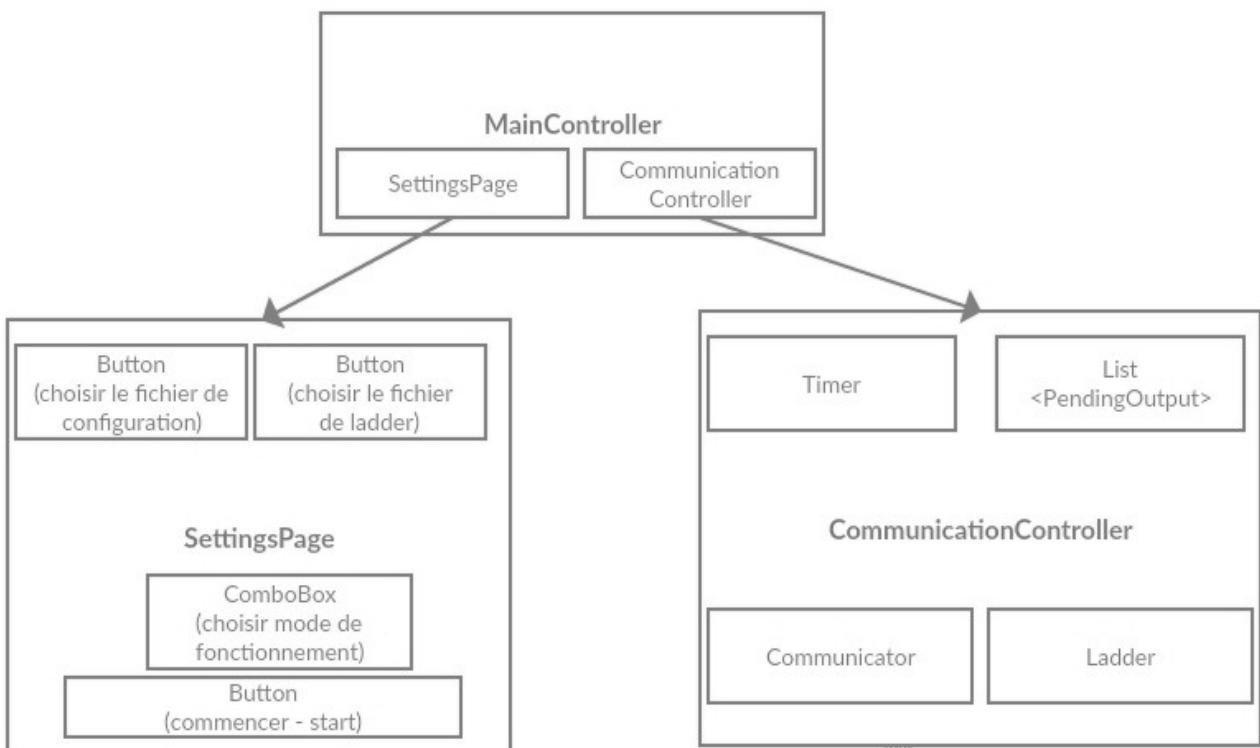


Figure 7.2.1 Architecture principale des classes, ses composants et les relations entre elles

Le **SettingsPage** est utilisé pour:

- choisir le mode de communication entre l'ordinateur et le dispositif – en série ou parallèle
- charger les fichiers XML de la configuration
- charger le fichier *.lad, qui décrit la logique pour ménager les sorties du dispositif en fonction des entrées du dispositifs (la logique **ladder**).
- Commencer la communication entre l'ordinateur et le dispositif

Ce page contient trois bouton et une boîte combo. Deux des bouton servent à sélectionner des fichiers de configuration et ladder, un des bouton sert à initialiser la communication (start) et la boîte combo sert à choisir entre mode de communication en série et parallèle.

Le **CommunicationController** est utilisé pour:

- initialiser la communication entre l'ordinateur et le dispositif
- connecter le **Communicator** et le **Ladder** (le Communicator sert à communiquer avec le dispositif et le Ladder sert à traiter les données lire des entrées de dispositif)
- garder les données destinées aux sorties de dispositif, qui sont en attente (**PendingOutputs**)
- Initialiser un compteur et traiter ses interruptions pour lire et écrire périodiquement sur le dispositif

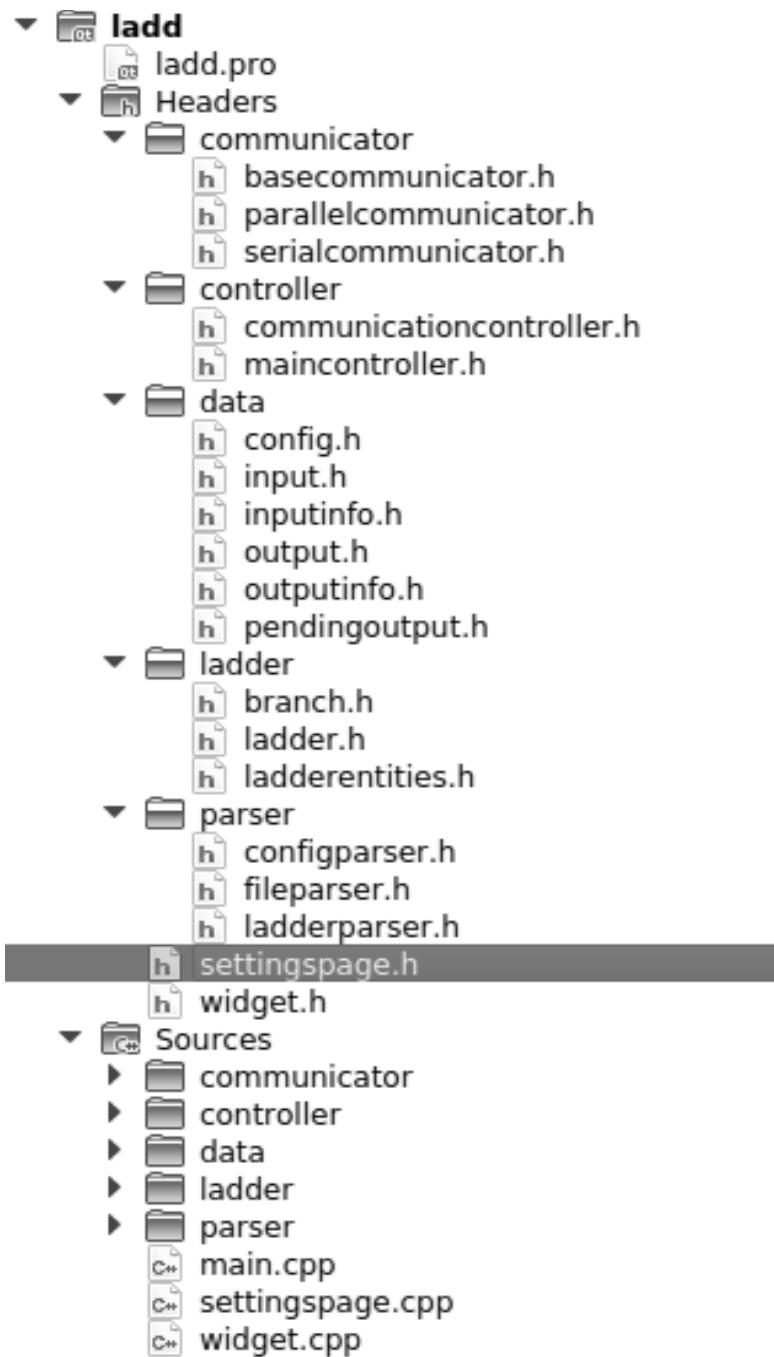


Figure 7.2.1 Structure hiérarchique des classes

Dans la figure 7.2.1 on a présenté la structure hiérarchique des classes du projet. On va les analyser en détail dans les sections suivantes. Il y a six groupes principales :

- « communicator » - les classes responsables pour la communication entre l'ordinateur et le dispositif
- « controller » - les classes ménageant le fonctionnement du projet
- « data » - les classe représentants des structure des données, avec lesquelles on travail dans le projet
- « ladder » - les classe contenant et responsable à exécuter la logique de ladder
- « parser » - les classes qui servent à dé-sérialiser les fichier du ladder et de la configuration
- UI

7.3. Fonctionnement principal

7.3.A Configurer le logiciel (set-up)

Avant de commencer la communication entre l'ordinateur et le dispositif, on doit charger deux fichiers séparées pour configurer le logiciel. Les fichiers de configuration décrivent des paramètres spécifiques pour le dispositif. Le fichiers de ladder décrivent la logique du ménagement des entrées et sorties du système.

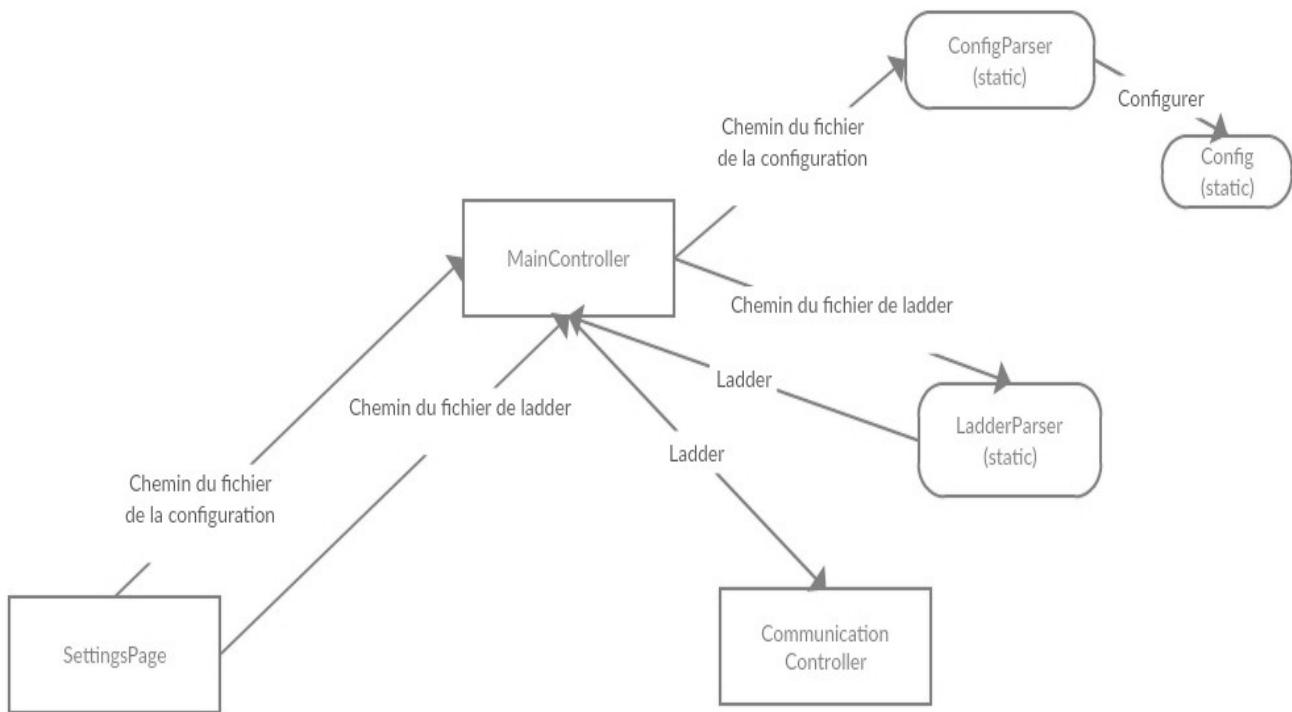


Figure 7.3.A.1 Schéma du processus de set-up du logiciel

- Configurer le Config (la configuration du système) – Fig. 7.3.A.1
 - Dans le SettingsPage on choisir le fichier de ladder
 - Le chemin du fichier soit transmis vers le MainController
 - Le MainController fait appel au objet statique ConfigParser en le passant le chemin du fichier de configuration
 - Le ConfigParser ouvre et dé-sérialise le fichier en un objet statique Config
 - L'objet Config est public et statique – ça veut dire – accessible dans tous les classes du logiciel
- Configurer le Ladder – Fig. 7.3.A.1
 - Dans le SettingsPage on choisir le fichier de ladder
 - Le chemin du fichier soit transmis vers le MainController
 - Le MainController fait appel au objet statique LadderParser en le passant le chemin de fichier de ladder
 - Le LadderParser ouvre et dé-sérialise le fichier en un objet Ladder
 - Le LadderParser passe l'objet Ladder au MainController
 - Le MainController passe l'objet Ladder au CommunicationController

7.3.B Communication entre l'ordinateur et le dispositif

La communication entre l'ordinateur est commencée par l'interface utilisateur – le SettingsPage. Le page fait appel au MainController et lui dit de démarrer la communication. Le MainController à sa part dit au CommunicationController de commencer la communication.



Figure 7.3.B.1 Schéma du démarrage de la communication

Dans la figure 7.3.B.1 on a présenté le processus de démarrage de la communication. Le CommunicationController commande au Communicator d'établir une connexion avec le dispositif. Dans la figure 7.3.B.2 on a présenté le processus de communication avec les dispositifs. Le contrôleur initialise un compteur pour pouvoir faire des appels périodiques au Communicator et lui demande de lire des données d'entrées (Inputs) du dispositif. Ces données sont transmises vers le contrôleur et il les passe au Ladder. L'objet Ladder traite les données et en suivant la logique décrite dans le fichier de ladder il calcul les données de sortie (Outputs) correspondantes d'écrire sur le dispositif. Les données de sortie sont transmises vers le contrôleur et il les passent au Communicateur. Le communicateur les écrit sur le dispositif.

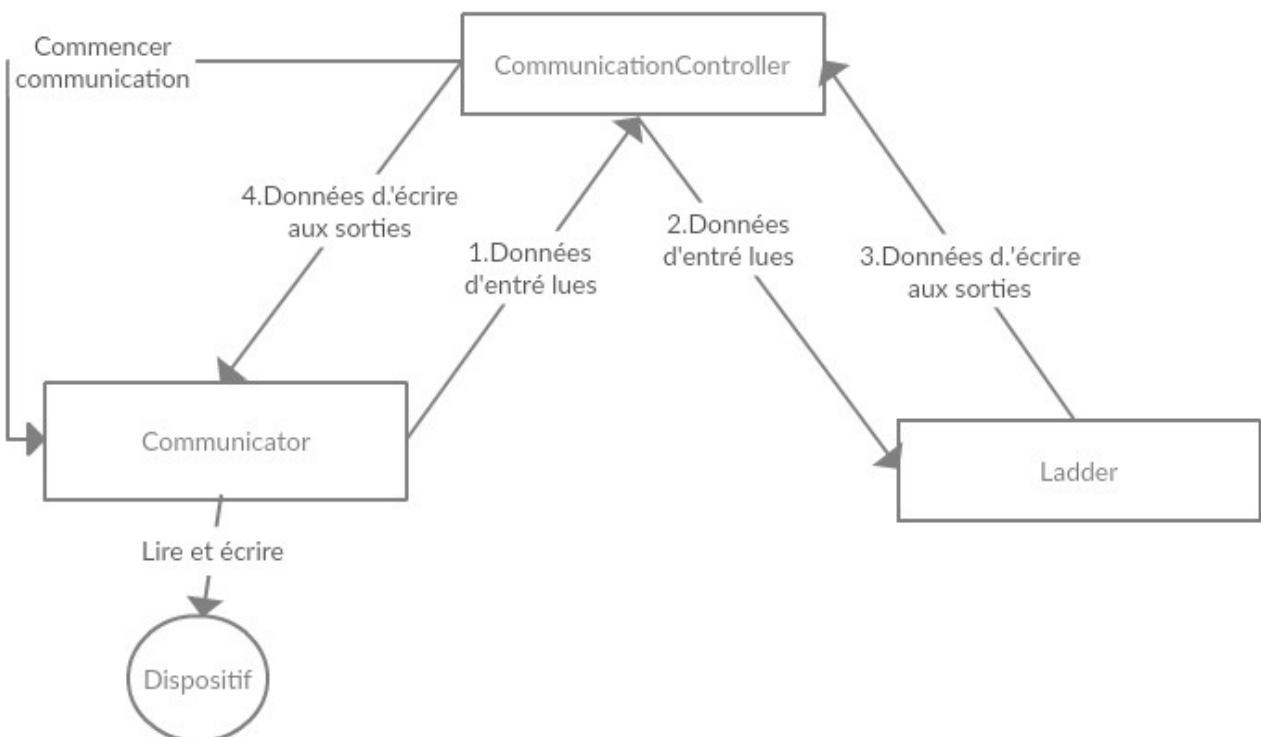


Figure 7.3.B.2 Schéma de la communication avec le dispositifs

8. Configuration du projet « Ladd »

8.1. La Configuration comme structure des données

La configuration du système est contenue dans un objet de la classe Config. L'objet Config est un **singleton** - Fig. 8.1.1.

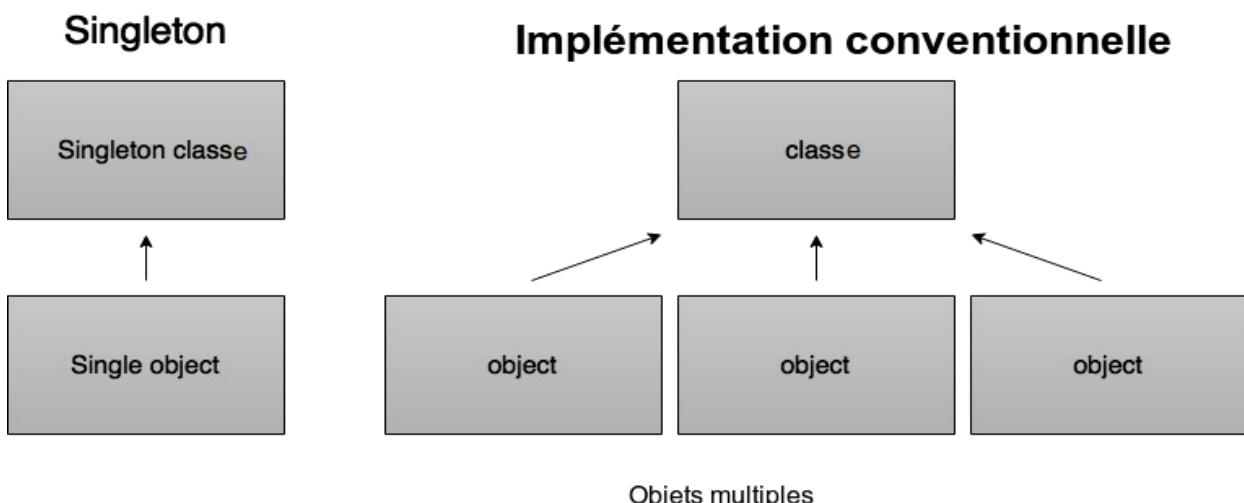


Figure 8.1.1 Comparaison entre le modèle singleton et le modèle conventionnel

Fig

En génie logiciel, le singleton est un patron de conception (design pattern) dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement). Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance seulement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Dans beaucoup des langages du type objet, il faudra veiller à ce que le constructeur de la classe soit privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

```
35 |     private:  
36 |         Config();  
37 |     private:  
38 |         static Config s_instance;  
39 |     
```

Figure 8.1.2 Photo instantanée du code source concernant le constructeur privé et le membre statique caractères pour un classe singleton

Dans cette capture d'écran (screenshot) de l'entête config.h on voit que le constructeur de la classe Config est privé et la classe possède un membre statique de soit même pour pouvoir donner aux autres classes un référence ver le seul objet du type Config - Fig 8.1.2 . Ce objet est initialisé dans le config.cpp

```
1 #include "config.h"
2
3 Config Config::s_instance = Config();
```

Figure 8.1.3 Photo instantanée du code source concernant le membre statique caractères pour un classe singleton

On initialise l'objet directement dans le cpp – Fig 8.1.3, pas dans une fonction supplémentaire init(), afin que on soit sûr que l'objet est toujours initialisé et accessible pour tous les autre partie du logiciel.

```
22 public:
23     static Config &getInstance();
24 }
```

Figure 8.1.4 Photo instantanée du code source concernant le getter statique du membre statique Config

```
24 Config &Config::getInstance()
25 {
26     return s_instance;
27 }
```

Figure 8.1.5 Photo instantanée du code source concernant le getter statique du membre statique Config

La fonction getInstance() n'a pas une modificateur const à la fin de son déclaration (static Config getInstance() const;) - Fig 8.1.4 et Fig 8.1.5. Ça veut dire que on peut modifier l'objet, dont la référence est transmise par cette fonction. Alors l'objet statique Config est modifiable au dehors de la classe Config par la fonction getter. C'est une pratique pas bonne d'avoir un getter susceptible à écriture, mais dans le cas du singleton, j'ai considéré que c'est une solution adéquat à la situation, parce que a cette façon on a la possibilité de modifier la configuration du système facilement et par tout dans le logiciel.

Dans la classe Config on définit le type énuméré CommunicationType. En programmation informatique, un type énuméré est un type de données qui consiste en un ensemble de constantes appelées énumérateurs. Lorsque on crée un type énuméré on définit ainsi une énumération – Fig.8.1.6 . Lorsqu'un identificateur tel qu'une variable est déclaré comme étant de type énuméré, cette variable peut recevoir n'importe quel énumérateur (lié à ce type énuméré) comme valeur. Le nom est généralement choisi pour décrire collectivement les énumérateurs d'ensemble. Les énumérateurs sont équivalents à des entiers. Le premier énumérateur vaut zéro, tandis que tous les suivants correspondent à leur précédent incrémenté avec un. Il est possible de choisir explicitement les valeurs des constantes d'énumération (ou de certaines d'entre elles ; dans ce cas, les autres suivent la règle donnée précédemment → + 1)

```
16 enum CommunicationType
17 {
18     Serial,
19     Parallel,
20     Invalid
21 };
```

Figure 8.1.6 Photo instantanée du code source concernant le type énuméré CommunicationType

Les valeurs des énumérateurs sont Serial, Parallel et Invalid. Serial et Parallel signifient respectivement le type de communication par le port en série (COM) et par le port parallèle(LPT). La valeur Invalid, signifie que aucune mode de communication soit sélectionnée par la boîte combo dans le SettingsPage alors le système a aucune type de communication configuré. Le type de communication n'est pas un membre du objet Config, parce que dans le future on prévoit d'avoir les deux mode de communication fonctionnant simultanément. De toute façon on utilise le type énuméré dans une autre partie du logiciel.

L'objet Config contient d'information sur la configuration des **entrées** et **sorties** du dispositif et sur la valeur en milliseconde de l'intervalle entre deux interruptions des différents **compteurs**.

```

47     QMap<int, int>      m_parallelTimerIdToMsecs;
48     QMap<int, InputInfo> m_parallelInputBitNumberToInputInfo;
49     QMap<int, int>      m_parallelOutputIdToBitNumber;
50
51     QMap<int, int>      m_serialTimerIdToMsecs;
52     QMap<int, InputInfo> m_serialInputBitNumberToInputInfo;
53     QMap<int, int>      m_serialOutputIdToBitNumber;
54

```

Figure 8.1.7 Photo instantanée du code source concernant les membres de la classe Config

Les membres de la classe sont divisés en deux groupes principaux, décrivant respectivement la configuration de la communication par le port en série et par le port parallèle – Fig. 8.1.7. Les deux groupes sont absolument identiques et chaque une d'elles comporte trois QMap remplis avec des données de la configuration:

QMap est un dictionnaire du type d'arbre bicolore – Fig. 8.1.8. Les arbres bicolores, offrent la meilleure garantie sur le temps d'insertion, de suppression et de recherche dans les cas défavorables. Ceci leur permet non seulement d'être utilisables dans des applications en temps réel, mais aussi de servir comme fondement d'autres structures de données à temps d'exécution garanti dans les cas défavorables, par exemple en géométrie algorithmique. Leur principal intérêt réside dans la complexité logarithmique des opérations suivantes : l'insertion, la recherche et la suppression. Ils sont cependant assez complexes à mettre en œuvre, car les opérations d'insertion et de suppression font appel à de nombreuses études de cas.

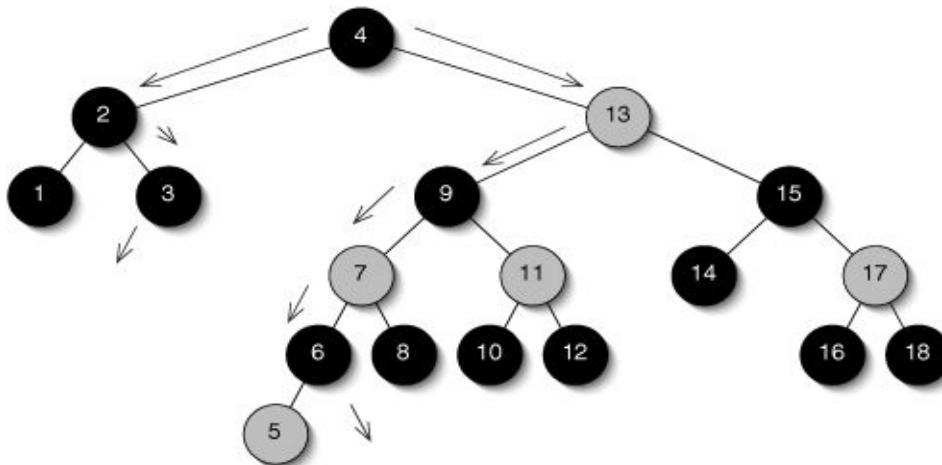


Figure 8.1.8 Schéma d'une arbre binaire rouge-noire

La recherche dans un arbre binaire d'un nœud ayant une clé particulière est un procédé récursif. On commence par examiner la racine. Si sa clé est la clé recherchée, l'algorithme se termine et renvoie la racine. Si elle est strictement inférieure, alors elle est dans le sous-arbre gauche, sur lequel on effectue alors récursivement la recherche. De même si la clé recherchée est strictement supérieure à la clé de la racine, la recherche continue dans le sous-arbre droit. Si on atteint une feuille dont la clé n'est pas celle recherchée, on sait alors que la clé recherchée n'appartient à aucun nœud, elle ne figure donc pas dans l'arbre de recherche.

Un arbre bicolore est un arbre binaire de recherche dans lequel chaque nœud a un attribut supplémentaire : sa couleur, qui est soit **rouge** soit **noire**. En plus des restrictions imposées aux arbres binaires de recherche, on ajoute les règles suivantes :

1. Un nœud est soit rouge soit noir ;
2. La racine est noire ;
3. Le parent d'un nœud rouge est noir ;
4. Le chemin de chaque feuille à la racine contient le même nombre de nœuds noirs.

Les trois QMap qui décrivent la configuration sont :

- QMap des « entiers, entiers » - contenant d'information sur les sorties du dispositif `m_parallelOutputIdToBitNumber`

Chaque sortie de dispositif est définie dans la configuration par un numéro d'identification et un nombre du bit. Avec le numéro d'identification on décrit la sortie dans la logique ladder. Le nombre du bit correspondant sert à savoir quelle bit de dispositif exactement de mettre en haut ou en bas, quand le numéro d'identification de cette sortie est présente entre les résultats de la logique du Ladder . Les clés d'arbre bicolore sont les numéros d'identification et les valeurs correspondantes – les numéros du bit.

- QMap des « entiers, InputInfo » contenant d'information sur les entrées du dispositif `m_parallelInputBitNumberToInputInfo`
La classe InputInfo décrit la configuration d'une entrée. Les entrées sont définies par trois caractéristiques – Fig. 8.1.9 .

```

21  private:
22      int id;
23      int bitNumber; // LSB
24      bool initialUnsetValue;
```

Figure 8.1.9 Photo instantanée du code source concernant les membres privés de la classe InputInfo

- id - numéro d'identification, qui représente l'entrée dans la logique ladder
- bitNumber – définit quel bit du dispositif précisément correspond au numéro d'identification
- initialUnsetValue – en fonction du matériel on peut avoir une entrée qui n'est pas active, mais donne la valeur lue est haute – 1. C'est une décision du fabricant du dispositif et le logiciel doit pouvoir s'adapter aux différents configurations.

Les clés d'arbre bicolore sont les numéros d'identification et les valeurs correspondantes – les objets InputInfo.

- QMap des « entiers, entiers » décrivant les compteurs `m_parallelTimerIdToMsecs`

Chaque compteur est défini dans la configuration par un numéro d'identification et un nombre de millisecondes. Avec le numéro d'identification on décrit le compteur dans la logique ladder. Le nombre de millisecondes correspondant signifie la grandeur de l'intervalle temporelle, que l'on doit attendre avant d'envoyer les résultats du ladder vers le dispositif. Les clés d'arbre bicolore sont les numéros d'identification et les valeurs correspondantes – les nombres de millisecondes.

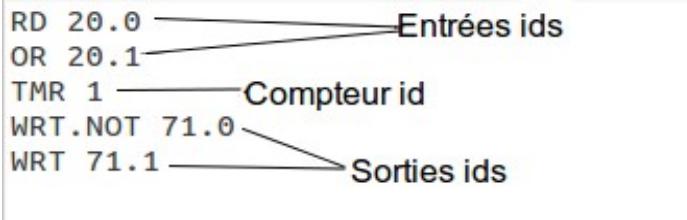


Figure 8.1.6 Représentation de l'utilisation des numéros d'identification des sorties, entrées et compteur dans une partie d'un ladder

8.2. Désérialisation de la configuration

8.2.A Le fichier config.xml

Le fichier de la configuration est composé en format XML.

Un fichier XML est un fichier texte possédant une structure particulière. XML est une notation, c'est à dire une manière d'écrire les informations. On utilise des balises pour délimiter les informations. XML permet d'imbriquer les balises. C'est à dire qu'une balise peut contenir des informations, mais aussi d'autres balises. XML est très bien adapté à la représentation d'informations hiérarchiques.

Les avantages du XML sont multiples :

- **Lisibilité** : il est facile pour un humain de lire un fichier XML car le code est structuré et facile à comprendre. En principe, il est même possible de dire qu'aucune connaissance spécifique n'est nécessaire pour comprendre les données comprises à l'intérieur d'un document XML.
- **Disponibilité** : ce langage est libre et un fichier XML peut être créé à partir d'un simple logiciel de traitement de texte (un simple bloc-note suffit).
- **Interopérabilité** : Quelques soit le système d'exploitation ou les autres technologies, il n'y a pas de problème particulier pour lire ce langage.
- **Extensibilité** : De nouvelles balises peuvent être ajoutées à volonté.
- Plusieurs parseurs XML différents doivent en principe (s'ils sont bien codés) produire le même résultat.
- Tous les navigateurs internet récents intègrent un parseur XML, pour lire les documents de ce langage informatique.

Le XML impose des règles de syntaxe très spécifiques :

- Les balises sont sensibles au majuscules et minuscules (case sensitive)

- Toute balise ouverte doit impérativement être fermée.
- Les balises doivent être correctement imbriquées.
Le XML étant très préoccupé par la structure des données, des balises mal imbriquées sont des fautes graves de sens. Ainsi l'écriture suivante est incorrecte car les balises ne sont pas bien imbriquées :<parent><enfant></parent></enfant> L'écriture correcte avec une bonne imbrication des éléments est :<parent><enfant></enfant></parent>
- Tout document XML doit comporter une racine.
En fait, la première paire de balises d'un document XML sera considéré comme la balise de racine (root).
- Les valeurs des attributs doivent toujours être mises entre des guillemets.
Le XML peut avoir (comme le Html) des attributs avec des valeurs. En XML, les valeurs des attributs doivent obligatoirement être entre des guillemets, au contraire du Html où leur absence n'a plus beaucoup d'importance.

Le fichier de la configuration est en format xml, parce que c'est un format convenable et très souvent utilisé pour des configuration en principe. Le langage est bien lisible pour les hommes et encore pour les ordinateurs. Dans la configuration on décrit les conditions initiales du dispositif, les caractéristiques du matériel et les valeurs principales des compteurs de délais.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  |<config>
3  |  |<parport>
4  |  |<serial>
5  |</config>
6  |
7  |
8  |
9  |
10 |
11 |
12 |
13 |

```

Figure 8.2.A.1 Photo instantanée du fichier config.xml avec les deux composants principal – parport et serial

La balise primaire <?xml version="1.0" encoding="UTF-8"?> donne d'information sur l'encodage du fichier – UTF8, et la version du langage xml - 1.0 .

Les balises principales – le racine, avec le nom « config » définissent l'espace dans lequel on décrit la configuration.

Ainsi suivent les deux parties principal de la configuration – la partie décrivant le dispositif connecté sur le port parallèle et cette responsable du dispositif connecté sur le port en série, respectivement embarqué par les balise <parport> et <serial> - Fig.8.2.A.1

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  |<config>
3  |  |<parport>
4  |  |<inputs>
5  |  |<outputs>
6  |  |<timers>
7  |</parport>
8  |<serial>
9  |  |<inputs>
10 |  |<outputs>
11 |  |<timers>
12 |</serial>
13 |</config>
14 |

```

Figure 8.2.A.2 Photo instantanée du fichier config.xml avec les composants principal du parport et serial – inputs, outputs et timers

Tous les deux domaines – parport et serial comportent trois composants – intputs (partie décrivant les conditions initiales des entrées), outputs(partie décrivant les conditions initiales des sorties), timers(partie décrivant les conditions initiales des compteurs du délais) – Fig. 8.2.A.2. Les trois composants sont structurées dans la même façon dans la partie parport et dans la partie serial, pour cette raison on fera l’analyse simultanée des deux. Les structure xml des inputs, outputs et timers correspondent toute à fait aux structures de la classe Config.

```

4 [ ] <inputs>
5   | <input>
6   |   | <id>10.0</id>
7   |   | <bitnumb>3</bitnumb>
8   |   | <!--LSB!-->
9   |   | <initialValue>1</initialValue>
10  |   | </input>
```

Figure 8.2.A.3 Photo instantanée du fichier config.xml avec les composants du input

```

36 [ ] <outputs>
37 [ ] <output>
38   | <id>70.0</id>
39   | <bitnumb>0</bitnumb>
40   | <!--LSB!-->
41   | </output>
```

Figure 8.2.A.4 Photo instantanée du fichier config.xml avec les composants du output

```

63 [ ] <timers>
64 [ ] <timer>
65   | <id>1</id>
66   | <msecs>6000</msecs>
67   | </timer>
```

Figure 8.2.A.5 Photo instantanée du fichier config.xml avec les composants du timer

8.2.B Chargement du fichier – interface utilisateur

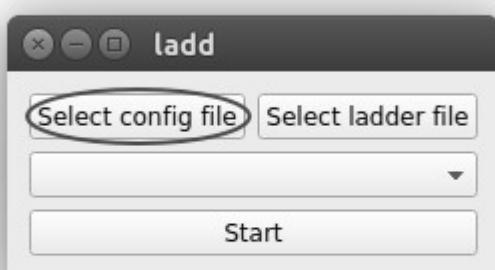


Figure 8.2.B.1 Photo instantanée de interface utilisateur

Quand on fait démarrer le programme, le fenêtre de la Fig. 8.2.B.1 apparaît sur l’écran. La classe logicielle de cet objet s’appelle SettingsPage - page de configuration. Là on a la possibilité de choisir le fichier de configuration, ladder, le mode de communication et de démarrer la communication avec le dispositif.

```
11 class SettingsPage : public QWidget  
12 {  
13     Q_OBJECT  
14 }
```

Figure 8.2.B.2 Photo instantanée du code source implémentant

La classe SettingsPage hérite la classe QWidget .

La classe QWidget est la classe de base de tous les objets d'interface utilisateur.

Le widget est l'atome de l'interface utilisateur : il reçoit des événements de souris, de clavier et d'autres depuis le système de fenêtrage, et peint une représentation de lui-même à l'écran.

Un widget qui n'est pas embarqué dans un widget parent est appelé une fenêtre. Habituellement, les fenêtres ont un cadre et une barre de titre, bien qu'il soit aussi possible de créer une fenêtre sans de telles décos en utilisant les drapeaux de fenêtre adaptés. Dans Qt, QMainWindow et les différentes sous-classes de QDialog, sont les types de fenêtres les plus communs.

Tout constructeur de widgets accepte un ou deux arguments standards :

- QWidget *parent = 0 est le parent du nouveau widget. Si c'est 0 (la valeur par défaut), le nouveau widget sera une fenêtre. Si ce n'est pas le cas, il sera un enfant de parent et limité par la géométrie de parent (à moins que vous ne spécifiez Qt::Window comme drapeau de fenêtre) ;
- Qt::WindowFlags f = 0 (si disponible) définit les drapeaux de fenêtre ; la valeur par défaut est adaptée à quasiment tous les widgets mais, pour obtenir par exemple, une fenêtre sans système de cadre, vous devrez utiliser des drapeaux spéciaux.

QWidget détient un bon nombre de fonctions membres, mais certaines d'entre elles ont peu de fonctionnalités directes ; par exemple, QWidget a une propriété de police mais ne l'utilise jamais lui-même. Il existe beaucoup de sous-classes qui fournissent une réelle fonctionnalité, telle que QLabel, QPushButton, QListWidget et QtabWidget.

La classe SettingsPage implémente aussi la macro Q_OBJECT – Fig. 8.2.B.2

La macro Q_OBJECT est obligatoire pour tout objet qui implémente des signaux, des slots ou des propriétés. La macro Q_OBJECT doit apparaître dans la section privée de la définition d'une classe qui déclare ses propres signaux et slots et/ou qui utilise d'autres services fournis par le système de métas-objets de Qt.

Chaque classe qui contient la macro Q_OBJECT possédera également un métas-objet. Un métas-objet contient des informations sur une classe qui hérite de QObject, par exemple le nom de la classe, le nom de la super-classe, les propriétés, les signaux et les slots. Les informations du métas-objet sont nécessaires au mécanisme de connexion signal/slot et au système de propriétés.

Cette macro requiert que la classe soit un dérivé de QObject.

```
9 SettingsPage::SettingsPage(QWidget* parent)  
10 : QWidget( parent )
```

Figure 8.2.B.3 Photo instantanée du code source de constructeur de la classe SettingsPage

Dans le cas de SettingsPage on utilise le constructeur « QWidget *parent = 0 » - Fig. 8.2.B.3

L'objet contient une référence au bouton – `m_loadConfigButton`.

Le widget QPushButton fournit un bouton de commande.

Le bouton, ou bouton de commande, est peut-être le widget le plus communément utilisé dans toute interface graphique : appuyer sur un bouton pour ordonner à l'ordinateur d'effectuer une action ou pour répondre à une question. Les boutons typiques sont OK, Appliquer, Annuler, Fermer, Oui, Non et Aide.

Un bouton de commande est de forme rectangulaire et affiche en général un texte décrivant son action.

Les boutons affichent un libellé textuel et optionnellement une petite icône. Ils peuvent être définis par l'utilisation des constructeurs et modifiés par la suite par l'utilisation de `setText()` et `setIcon()`. Si le bouton est désactivé, l'apparence du texte et de l'icône sera manipulée en conformité avec le style de d'interface graphique, de manière à donner une apparence « désactivée » au bouton.

Un bouton émet le signal `clicked()` quand il est activé par la souris, la barre d'espace ou par un raccourci de clavier. Connectez ce signal pour effectuer l'action du bouton. Les boutons fournissent aussi des signaux moins communément utilisés, par exemple, `pressed()` et `released()`.

```
16     m_loadConfigButton = new QPushButton();
17     m_loadConfigButton->setText("Select config file");
18     connect(m_loadConfigButton, SIGNAL(clicked()), this, SLOT(loadNewConfigurationFile()));
```

Figure 8.2.B.4 Photo instantanée du code source de l'initialisation du bouton load

On initialise le bouton dans le constructeur de SettingsPage – Fig.8.2.B.4. Le mémoire est alloué dynamiquement et pour cela on doit faire attention sur la destruction du objet QPushButton. En principe quand on crée un objet sur le tas (heap) on doit le détruire manuellement dans le destructeur du SettingsPage. Du tout le système de Qt offre une organisation très bien pour le ménagement des objets, qui existent sur le tas. Quand on fait passer un objet parent du type – QObject, au constructeur d'un objet enfant – l'objet parent s'en occupe du détruire ses enfants dans son propre destructeur.

Lorsque on utilise un layout, il n'y a besoin de faire passer un parent au constructeur de widgets enfants. Le layout impose automatiquement un parent aux widgets (en utilisant `QWidget :: setParent ()`), de telle façon que les widgets, appartenant au layout sont des enfants du widget dont le layout est alloué.

Widgets dans un layout sont des enfants du widget sur lequel le layout est installé, pas du layout lui-même. Widgets peuvent avoir seulement d'autres widgets comme parents - pas des layouts.

On peut avoir des layouts imbriqués en utilisant `addLayout ()`. Le layout imbriqué devient un enfant au layout, dans lequel il est inséré. Tous ça est utilisé dans le logiciel pour assurer la destruction du objet `m_loadConfigButton`. En plus en utilisant des layouts on ménage la position du widget dans l'espace du fenêtre principal – le SettingPage.

```
34     QBoxLayout* const buttonsLayout = new QBoxLayout();
35     buttonsLayout->addWidget(m_loadConfigButton, 1, Qt::AlignBottom);
```

Figure 8.2.B.5 Photo instantanée du code source de l'initialisation du layout des boutons

```
37     QVBoxLayout* const mainLayout = new QVBoxLayout(this);
38     mainLayout->addLayout(buttonsLayout);
```

Figure 8.2.B.6 Photo instantanée du code source de l'addition du layout des bouton dans le layout général

QHBox- et QVBoxLayout sont deux variations de layout (mise en page) qui diffèrent dans l'orientation de ses widgets dans l'espace. La classe de QHBoxLayout aligne les widgets horizontalement, la classe QVBoxLayout – verticalement.

Tout d'abord, on crée les widgets que l'on veut intégrer au layout. Puis on crée l'objet QHBoxLayout - buttonsLayout, et on ajoute les widgets dans le layout – Fig. 8.2.B.5. À la fin on crée l'objet QVBoxLayout en faisant passer le pointure « this » au constructeur – Fig. 8.2.B.6, ce qui assure que le mainLayout est installé sur fenêtre dont « this » fait référence - SettingsPage - tout le même comme appelant la méthode QWidget::setLayout(). Quand le , les widgets (du layout) sont réassignés au «this» comme parent.

Quand le bouton m_loadConfigButton est poussé il émit un signal « clicked() » et fait appel au slot, avec lequel ce signal est connecté – loadNewConfigurationFile().

```
31     private slots:
32         void loadNewConfigurationFile();
33
34     private:
35         void loadNewFile(FileType fileType);
36
```

Figure 8.2.B.7 Photo instantanée du code source des fonctions du chargement des fichiers nouveaux

```
65
66     void SettingsPage::loadNewConfigurationFile()
67     {
68         loadNewFile(SettingsPage::FileType::FileConfig);
69     }
70
```

Figure 8.2.B.8 Photo instantanée du code source des fonctions du chargement des fichiers nouveaux

La fonction loadNewConfigurationFile() de sa part fait appel à la fonction loadNewFile(SettingsPage::FileType fileType) – Fig. 8.2.B.8 . L'idée est de éviter la répétition de code en réutilisant une fonction pour le chargement du fichier de la configuration et de ladder. Alors on fait passer comme argument un paramètre du type FileType – un énumérateur qui spécifie le type de fichier, qui est exigé d'être chargé – 8.2.B.9 .

```
15     public:
16     enum FileType
17     {
18         FileConfig,
19         FileLadder
20     };
21
```

Figure 8.2.B.9 Photo instantanée du code source du type énuméré FileType

Les deux types énumérés désignent le fichier de la configuration et le fichier du ladder, mais en utilisant la technique des types énumérés on peut facilement ajouter quelconques nombres des types supplémentaires, quand on en a besoin.

Dans la fonction loadNewFile() on fait appel à la fonction statique de la classe QFileDialog QStringList QFileDialog::getOpenFileNames (QWidget * parent = 0, const QString & caption = QString(), const QString & dir = QString(), const QString & filter = QString(), QString * selectedFilter = 0, Options options = 0) [static].

Cette fonction crée une boîte de dialogue des fichiers modale avec le widget parent donné. Si le parent est différent de 0, la boîte de dialogue sera affichée centrée sur le widget parent.

Le répertoire de travail de la boîte de dialogue est défini par le paramètre dir. Si dir inclut un nom de fichier, le fichier sera sélectionné. Seuls les fichiers correspondant au filtre filter donné sont affichés. Le filtre sélectionné est défini par selectedFilter. Les paramètres dir, selectedFilter et filter peuvent être des chaînes vides.

Le titre de la boîte de dialogue est défini par caption. Si caption n'est pas spécifié, un titre par défaut sera utilisé.

La classe QFileDialog fournit une boîte de dialogue qui permet aux utilisateurs de sélectionner des fichiers ou des répertoires.

La classe QFileDialog permet à l'utilisateur de naviguer à travers le système de fichier afin de sélectionner un ou plusieurs fichiers ou répertoires.

La manière la plus simple de créer une QFileDialog est d'utiliser les fonctions statiques. Ces fonctions statiques appelleront si possible la boîte de dialogue native – Fig. 8.2.B.10.

```
43 void SettingsPage::loadNewFile(SettingsPage::FileType fileType)
44 {
45     const QString filePath = QFileDialog::getOpenFileName(this,
46                                         "Open",
47                                         "/home",
48                                         fileType == FileConfig
49                                         ? tr("XmlFiles (*.xml)")
50                                         : tr("TextFiles (*.LAD|"));
51     emit fileLoaded(filePath, fileType);
52 }
```

Figure 8.2.B.10 Photo instantanée du code source de la fonction loadNewFile()

La boîte de dialogue affiche initialement le contenu du répertoire /home et affiche les fichiers correspondant aux masques donnés par la chaîne « XmlFiles (*.xml) », parce que le paramètre fileType est d'une valeur FileConfig. Le parent de la boîte de dialogue est défini à this et le titre de la fenêtre à « Open » - Fig.8.2.B.11.



Figure 8.2.B.11 Photo instantanée du code source du type énuméré FileType

Le signal public fileLoaded(const QString filePath, SettingsPage::FileType fileType) – Fig. 8.2.B.11 émit le chemin du fichier et son type (config ou ladder) vers le MainController . Ça se produit grâce au système des signaux et des slots du Qt.

```
28 signals:  
29     void fileLoaded(const QString filePath, SettingsPage::FileType fileType);  
30 |
```

Figure 8.2.B.11 Photo instantanée du code source du signal public fileLoaded()

Les frameworks plus anciens réalisent ce type de communication en utilisant des fonctions de rappel (callbacks). Ce sont en fait des pointeurs sur une fonction, donc si on veut qu'une fonction de traitement signalise et transmet d'information en cas de certains événements, on lui fait passer un pointeur sur une autre fonction (fonction de rappel). La fonction de traitement va alors appeler la fonction de rappel lorsque ce sera nécessaire. Les fonctions de rappel ont deux défauts fondamentaux : premièrement, elles n'ont pas de mécanisme de sûreté du typage - on ne peut jamais être certains que la fonction de traitement va appeler la fonction de rappel avec des arguments corrects. Deuxièmement, la fonction de rappel est fortement couplée à la fonction de traitement étant donné que cette dernière doit savoir quelle fonction de rappel d'appeler.

Le mécanisme des signaux et slots fournit un contrôle des types : la signature d'un signal doit correspondre à la signature du slot récepteur (en réalité, un slot peut avoir une signature plus courte que celle du signal qu'il reçoit car il peut ignorer les arguments en trop). Étant donné que les signatures doivent être compatibles, le compilateur peut nous aider à détecter les erreurs de correspondance de types. Les signaux et les slots sont faiblement couplés : une classe qui émet un signal ne sait pas (et ne se soucie pas de) quels slots vont recevoir ce signal. C'est le mécanisme signaux/slots qui va garantir que, si vous connectez un signal à un slot, ce slot sera appelé avec les paramètres du signal en temps voulu. Les signaux et les slots peuvent prendre n'importe quel nombre d'arguments de n'importe quel type. Ils assurent un contrôle complet des types.

Toutes les classes qui héritent de QObject ou d'une de ses sous-classes (par exemple, QWidget) peuvent contenir des signaux et des slots.

8.2.C. Déserialisation

```
12 m_settingsPage = new SettingsPage();  
13 connect(m_settingsPage,  
14     SIGNAL(fileLoaded(QString,SettingsPage::FileType)),  
15     this,  
16     SLOT(onFileLoaded(QString,SettingsPage::FileType)));  
17 |
```

Figure 8.2.C.1 Photo instantanée du code source de l'initialisation de SettingsPage

Dans le classe MainController on fait la connexion entre le signal de SettingsPage et le slot du contrôleur – Fig. 8.2.C.1. C'est très important de définir les variables du type personnalisé (custom variables) avec leur complet espace de noms, pour que le système métta de Qt peut les reconnaître (par exemple Settings::FileType, même si quand on a introduire l'espace de noms(namespace) avec la déclaration « using SettingsPage ; »). Dans le cas de SettingsPage on na pas introduire un espace de noms, parce que le projet et encore très petit et en n'a pas besoin, mais de toute façon c'est important d'en faire attention si le système reconnaît les variables. En plus, en faisant la connexion entre signal et slot, on ne doit pas séparer avec une nouvelle ligne les partie du espace de noms, même si ils sont énormément longs , parce que si on fait ça, quand on construit le programme en mode de release (release build), les signaux et les slots ne travailleront comme il faut.

```

44 void MainController::onFileLoaded(const QString filePath, SettingsPage::FileType fileType)
45 {
46     if(fileType == SettingsPage::FileConfig)
47     {
48         ConfigParser::parseConfig(filePath);
49     }

```

Figure 8.2.C.2 Photo instantanée du code source de la fonction onFileLoaded

La fonction parseConfig(const QString &filePath) est du type void, parce que l'objet Config est singleton et il n'y a pas besoin de lui sauvegarder dans un objet contrôleur ou quoi qu'il soit d'autre objet de ménagement. Dans la fonction parseConfig on accède l'objet Config en utilisant une référence inscriptible – Fig. 8.2.C.3 .

```

22 void ConfigParser::parseConfig(const QString &filePath)
23 {
24     Config& result = Config::getInstance();
25     QFile file(filePath);
26     if(!file.open(QIODevice::ReadOnly))
27     {
28         return;
29     }
30
31     QXmlStreamReader* const xmlReader = new QXmlStreamReader(&file);

```

Figure 8.2.C.3 Photo instantanée du code source de la fonction parseConfig

La classe QXmlStreamReader permet de lire rapidement des fichiers XML un peu à la manière de SAX(Simple API for XML). Comme SAX, on parcourt l'arbre XML et on ne peut le remonter pendant son parcours. QXmlStreamReader repose sur le principe d'une boucle dans laquelle on va parcourir le fichier à l'aide de la méthode readNext() et vérifier sur quel type de jeton(token) on est positionné.

Avant la version 4.3 de Qt on utilise le modèle de désérialisation des fichiers xml – DOM. Les méthodes SAX et DOM adoptent chacune une stratégie très différente pour analyser la syntaxe des documents XML, elles s'utilisent donc dans des contextes différents. DOM charge l'intégralité d'un document XML dans une structure de données, qui peut alors être manipulée puis reconvertie en XML. Cependant pour cela, il faut que la taille de la structure représentant le document XML ne soit pas supérieure (ou pas trop) à ce que peut contenir la mémoire vive. La méthode SAX apporte alors une alternative dans les cas de figure où les documents XML sont de taille très importante (on parle alors d'adaptation au volume de données).

De toute façon la configuration du logiciel « ladder » n'est pas d'une grandeur supérieure, mais en réalisant le projet on a décidé d'utiliser le QXmlStreamReader au lieu du QDomElement pour expérimenter avec la nouveauté introduite dans les versions dernières de Qt.

```

33 while(true){
34     tokenType = xmlReader->readNext();
35     if (tokenType == QXmlStreamReader::EndDocument
36         || tokenType == QXmlStreamReader::Invalid)
37     {
38         break;
39     }
40

```

Figure 8.2.C.4 Photo instantanée du code source du balayage de la liste des éléments du fichier xml

Dans cette boucle infinie on traverse le fichier entier. Quand il y a un erreur ou on atteint la fin du fichier on sorti la boucle. La fonction readNext() retourne une valeur du type QXmlStreamReader::TokenType – Fig.8.2.C.4 .

Nom	Valeur	Description
QXmlStreamReader::NoToken	0	On n'a rien lit encore
QXmlStreamReader::Invalid	1	Il s 'a produit une erreur
QXmlStreamReader::StartDocument	2	On peut accéder la version du XML en documentVersion(), et l'encodage spécifié dans le fichier xml dans documentEncoding().
QXmlStreamReader::EndDocument	3	On signale la fin du document
QXmlStreamReader::StartElement	4	On signal le début d'un élément avec le nom namespaceUri() and name(). Les éléments vides sont aussi signalé par le StartElement, suivis directement parEndElement. Attributs peuvent être accéder par la fonction attributes(), et les déclarations des espaces de noms par - namespaceDeclarations().
QXmlStreamReader::EndElement	5	On signale la fin du document, mais les valeurs de namespaceUri() et name() sont encore accessible pour l'utilisateur .
QXmlStreamReader::Characters	6	On signale que il a une chaîne des caractères accessibles par la fonction text().
QXmlStreamReader::Comment	7	On signale que il a une chaîne des caractères accessibles par la fonction text(), mais ils représentent un commentaire.

```

41 ▼    if (tokenType == QXmlStreamReader::StartElement)
42 {
43     const QString& tagName = xmlReader->name().toString();
44     Config::CommunicationType type = ConfigParser::tagToCommunicationType().value(tagName, Config::CommunicationType::Invalid);
45     ConfigParser::parseConfig(xmlReader, result, type);
46 }
```

Figure 8.2.C.5 Photo instantanée du code source du processus de différentiation de config parport au config serial

Étant donné que la description de la configuration pour les dispositifs parallèles et en série sont absolument les mêmes on utilise une fonction pour la déserialisation des deux type de configuration - Fig. 8.2.C.5. Vers la fonction privée de la classe ConfigParser on fait passer comme arguments un pointeur vers l'objet lecture du xml, la référence vers l'objet Config et le type du configuration. De cette fonction on obtient un arbre rouge noir, qui contient les types énumérés correspondant au nom des balises, qui entourent les deux configurations séparées (en série et parallèle) – respectivement <serial></serial> et <parport></parport> - Fig. 8.2.C.6 .

On utilise un objet QMap statique, afin d'éviter les créations multiples de la même structure. C'est bien possible qu'on n'utilisera cette fonction tellement sauvet, mais de toute façon cette un patron de conception (design pattern), avec lequel on peut expérimenter.

```

345     QMap<QString, Config::CommunicationType> ConfigParser::tagToCommunicationType()
346     {
347         static QMap<QString, Config::CommunicationType> map =
348             QMap<QString, Config::CommunicationType>(
349             {
350                 {s_parallelTag, Config::CommunicationType::Parallel},
351                 {s_serialTag, Config::CommunicationType::Serial}});
352
353         return map;
354     }

```

Figure 8.2.C.6 Photo instantanée du code source de l'arbre rouge-noir des types de configuration

Le setter de chaque variable dans la classe Config accepte deux paramètre – un correspondant au type de la valeur et un correspondant au type de la configuration (parallèle ou en série). En utilisant ces setters on réussit de dé-sérialiser les composantes deux types de configuration dans des fonctions communes – Fig. 8.2.C.7.

```

77     if (tagName == s_inputsTag)
78     {
79         config.setInputBitNumberToInputInfo(
80             ConfigParser::parseInputs(xmlReader), type);
81     }
82 }
```

Figure 8.2.C.7 Photo instantanée du code source d'appel à la fonction qui dé-sérialise les inputs

On peut voir clairement dans l'image suivant la structure du parseur. Il y a trois fonction pour la désérialisation des compteurs (parseTimers), des sorties (parseOutputs) et des entrées (parseInputs). Chaque d'elles utilise une fonction supplémentaire pour ses éléments imbriqués, respectivement – parseTimer, parseOutput et parseInput. Tous ses fonctions retournent les structures correspondantes à la structure de la classe Config – Fig. 8.2.C.8.

```

28 public:
29     static void parseConfig(const QString& filePath);
30 private:
31     static void parseConfig(QXmlStreamReader* const xmlReader, Config& config, Config::CommunicationType type);
32     static QMap<int, int> parseOutputs(QXmlStreamReader* const xmlReader);
33     static QMap<int, InputInfo> parseInputs(QXmlStreamReader* const xmlReader);
34     static QMap<int, int> parseTimers(QXmlStreamReader* const xmlReader);
35     static QPair<int, int> parseOutput(QXmlStreamReader* const xmlReader);
36     static QPair<int, InputInfo> parseInput(QXmlStreamReader* const xmlReader);
37     static QPair<int, int> parseTimer(QXmlStreamReader* const xmlReader);

```

Figure 8.2.C.8 Photo instantanée du code source des fonctions supplémentaires dans la classe ConfigParser

```

9 class Config
10 {
11
12 public:
13     void setTimerIdToMsecs(const QMap<int, int> &timerIdToMsecs, CommunicationType type);
14     void setInputBitNumberToInputInfo(const QMap<int, InputInfo> &inputBitNumberToInputInfo, CommunicationType type);
15     void setOutputIdToBitNumber(const QMap<int, int> &outputIdToBitNumber, CommunicationType type);
16

```

Figure 8.2.C.9 Photo instantanée du code source des setters dans la classe Config

Les fonctions de déserialisation sont structurées dans une même façon et pour cette raison on va analyser seulement une d'elles – celle qui s'occupe aux entrées.

- ParseInputs- Fig. 8.2.C.10

Vers la fonction privée on fait passer comme argument un pointeur vers l'objet lecture du xml. L'objet lecture garde toujours un référence vers l'endroit atteint dans le fichier et reprend la lecture dans ce point précise.

Pour savoir que la fin d'élément <inputs> est atteinte on vérifie que :

- le type du jeton (token type) est égal à QXmlStreamReader::EndElement
- le nom d'élément est « inputs » - ça signifie que l'objet lecture est sur le tag de fermeture </inputs>.

Quand ses deux conditions sont vraies ont interrompe la boucle.

On doit aussi faire attention si les éléments, que on attache au réponse sont valides. Pour cela on vérifie que la première partie du couple nommé input est différent de la valeur statique constante invalide s_invalidInt (égale à -1) et que l'objet InputInfo – la deuxième partie du couple et valide aussi. Dans les cas, où le couple n'est pas valide, on n'interrompe pas la déserialisation, parce que même si un élément aura une faute, les autres éléments peuvent contenir d'information utile.

```
186 do
187 {
188     if(tagName == s_inputTag
189         && tokenType == QXmlStreamReader::StartElement)
190     {
191         QPair<int,InputInfo> input= parseInput(xmlReader);
192
193         if(input.first != Config::s_invalidInt
194             && input.second.isValid())
195         {
196             result.insert(input.first, input.second);
197         }
198     }
199
200     tokenType = xmlReader->readNext();
201     tagName = xmlReader->name().toString();
202 }
203
204 while(tagName != s_inputsTag
205     || tokenType != QXmlStreamReader::EndElement);
206 }
```

Figure 8.2.C.10 Photo instantanée du code source de la logique de déserialisation des inputs

- parseInput

Dans la fonction imbriquée responsable à la déserialisation d'élément <input> on utilise la même logique pour découvrir la balise fermeture </input> et interrompe la boucle do-while.- Fig. 8.2.C.11

```
223 do
224 {   {...}
225     tokenType = xmlReader->readNext();
226     tagName = xmlReader->name().toString();
227 }
228
229 while(tagName != s_inputTag
230     || tokenType != QXmlStreamReader::EndElement);
231 }
```

Figure 8.2.C.11 Photo instantanée du code source de la logique de vérification de la fin des éléments input

Dans cette do-while boucle on vérifie le nom de balise imbriqué. Il peut être - <id>, <bitnumb> et <initialValue>. Avec des conditions if(statement) on traite les trois cases différents.

Le cas le plus intéressant est cet de la déserialisation du numéro d'identification. Par défaut le id comporte deux partie des nombres entiers séparées par un point.

Par exemple :

10.2 ; 31.4 ; 70.7 ; 80.0 ;

On doit faire une remarque, que le nombre dans la seconde partie ne peut pas dépasser la valeur de 7. Alors – 0, 1, 2, 3, 4, 5, 6, 7 sont les valeurs possibles. Cette condition provient du protocole de description de la logique ladder.

Pour les besoins du logiciel « ladd » on doit convertir ces valeur en un seule valeur entière. Ça se passe dans la fonction idToInt(const QString& branchString) – Fig. 8.2.C.12.

```

226
227 if(tagName == s_idTag
228     && tokenType == QXmlStreamReader::StartElement)
229 {
230     int id = idToInt(xmlReader->readElementText());
231     result.second.setId(id);
232 }
```

Figure 8.2.C.12 Photo instantanée du code source de la logique de la transformation des chaînes de caractères en ids entières

- **idToInt**

Cette fonction est statique et publique, appartenant à la classe de base FileParser, qui ConfigParser hérite. Elle est utilisée pour la déserialisation des numéros d'identification des entrées et sorties du dispositif, mais aussi pour les id-s des compteurs.

Les id-s des compteurs sont sous en forme différente. Leur structure comporte seulement un nombre entière – sans des points ou quoi qu'il soit autres caractères différents. Pour cette raison on fait une vérification des nombres des partie de la chaîne de caractères séparables par un point. La variable entière size comporte cette valeur – Fig. 8.2.C.13 .

```

20 QStringList idParts = branchString.split(FileParser::s_idSeparator);
21 const int size = idParts.size();
```

Figure 8.2.C.13 Photo instantanée du code source de la logique d'estimation de nombres des parties de id

La variable ok sert comme une flag indiquant si la conversion de chaîne de caractères en entière est réussie. Dans le cas d'une faute on retourne la valeur constante invalide – Fig. 8.2.C.14.

```

22 QString firstPart = idParts.takeAt(0);
23 bool ok = false;
24 result = firstPart.toInt(&ok);
25 if(!ok)
26 {
27     return Config::s_invalidInt;
28 }
```

Figure 8.2.C.14 Photo instantanée du code source de la vérification des données de id

Puis on vérifie si le nombre des parties est égale au nombre fixé des parties d'un numéro d'identification des compteurs – Fig.8.2.C.15. Si la condition est vraie on retour le résultat courant.

```
30 ▼ if(size == s_numberPartsTmrId)
31 {
32     return result;
33 }
34
```

Figure 8.2.C.15 Photo instantanée du code source de la vérification de nombres des parties de id

Si le nombre des parties est égal au nombre prédéfini pour un id de ladder on multiplie le résultat courant par huit et puis on ajoute la deuxième partie entière (secondInt). On multiplie par huit pour que le id en forme entière soit vraiment unique.

```
10.0 → 80
10.1 → 81
10.2 → 82
10.3 → 83
10.4 → 84
10.5 → 85
10.6 → 86
10.7 → 87

11.0 → 88
```

Il est possible d'avoir seulement huit différent valeur pour la deuxième partie du numéro d'identification et pour cela on doit multiplier la première partie par huit – Fig. 8.2.C.16.

```
35 ▼ if(size == s_numberPartsLaddId)
36 {
37     result *= 8;
38     QString secondPart = idParts.takeAt(0);
39     int secondInt = secondPart.toInt(&ok);
40     if(!ok)
41     {
42         return Config::s_invalidInt;
43     }
44
45     result += secondInt;
46
47     return result;
48 }
```

Figure 8.2.C.16 Photo instantanée du code source de la logique de transformation des id en entières

La valeur constante invalide est utilisée aussi par la fonction isValid de InputInfo, ou on vérifie si la valeur de id est égale à s_invalidInt – Fig. 8.2.C.17.

```
13 ▼ bool InputInfo::isValid() const
14 {
15     return [id != Config::s_invalidInt];
16 }
17
```

Figure 8.2.C.17 Photo instantanée du code source de la vérification des ids de la classe InputInfo

8.3 Application de la configuration dans la conversion numéro d'identification

Il existe deux applications de la configuration

- Conversion des numéros d'identification vers des numéros entières et l'inverse.
- Définition du période de temps, que on doit attendre avant de transmettre d'information vers les sorties de dispositif.

Dans cette chapitre on analyse seulement l'application de la configuration dans le domaine de conversion des numéros d'identification, parce que l'autre application est fortement liée avec la logique ladder et c'est plus convenable de l'analyser ensemble avec le ladder.

- char vers pair<int,bool>

La classe utilisée pour décrire les données d'une entrée sont présentées par un couple de valeur entière et valeur booléenne. La valeur entière représentant le numéro d'identification. La valeur booléenne, qui donne l'information si l'entrée est active ou pas. On peut appeler ce couple avec le mot clé Input. La classe qui exécute la logique de ladder travail avec ces couples d'information.

Pour convertir la valeur char, obtenue en lisant les entrées du dispositif, en valeur Input, on fait appel au objet statique Config et lui demande le QMap, qui contient la connexion entre un nombre de bit et un objet InputInfo. Ça se passe dans le BaseCommunicator – Fig. 8.3.2.

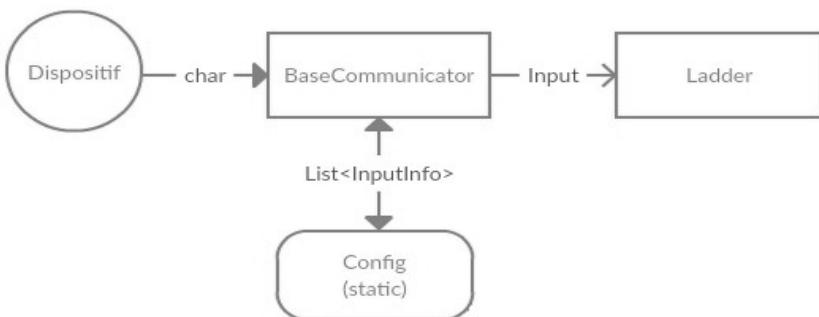


Figure 8.3.2 Schéma de l'utilisation de la configuration dans la transformation des données lues des dispositifs en structures Input

On traverse l'arbre bitnumToInputInfos et pour chaque paire de clé et valeur de cette arbre on vérifie si le bit correspondant au clé est en haut ou en bas. La fonction isBitSet exécute cette vérification, mais on va l'analyser dans une autre chapitre concernant la communication avec les dispositifs. Puis on crée le pair de Input et l'insert dans le QMap result, qui on va retourner à la fin de la fonction. Le clé de cette paire est le numéro d'identification de l'objet InputInfo et la valeur correspondante est le résultat de la vérification de l'inégalité entre la valeur défaut de l'état de l'entrée (en bas/en haut) et la valeur courante – Fig. 8.3.3 .

```
105     QMap<int, bool> BaseCommunicator::charToInputs(char x)
106 {
107     QMap<int, bool> result;
108     QMap<int, InputInfo> bitnumToInputInfos =
109         Config::getInstance().inputBitNumberToInputInfo(m_communicationType);
110
111     QMap<int, InputInfo>::iterator iter = bitnumToInputInfos.begin();
112     for(;iter != bitnumToInputInfos.end(); iter++)
113     {
114         bool value = isBitSet(x,iter.key());
115         InputInfo info = iter.value();
116         result.insert(info.getId(), (value != info.getInitialUnsetValue()));
117     }
118
119     return result;
120 }
```

Figure 8.3.3 Photo instantanée du code source la fonction transformante les données lues en structures Input

- char vers pair<int,Output>

Après le traitement des Inputs par un objet de la classe Ladder, qui contient la logique ladder, on fait passer un liste des valeurs de type Outputs vers des communicateurs, qui sont responsables à transmettre des données vers les dispositifs. La classe Output comporte une valeur entière – id, et une valeur booléenne – isActive – Fig.8.3.4 . La classe communicateur doit convertir ses données en une valeur de type char, parce que la communication avec des dispositifs est réalisée par l'échange des données de type char.

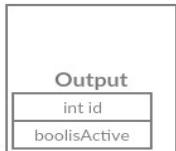


Figure 8.3.4 Schéma de la structure Output

Pour convertir la valeur Output en valeur char, on fait appel au objet statique Config et lui demande le QMap, qui contient la connexion entre un numéro d'identification et un nombre de bit. Ça se passe dans le BaseCommunicator, parce que tous les communicateurs, qui héritent cette classe, utilisent cette fonction de conversion – Fig. 8.3.5 .

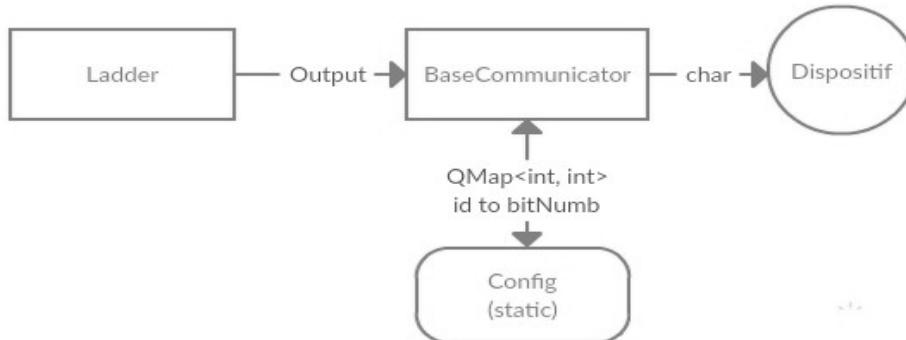


Figure 8.3.5 Schéma de la transformation des structures Output en données à écrire sur les dispositifs

QMap comporte des clés représentant les numéros d'identification des sorties – Fig. 8.3.6.

```

115 unsigned char BaseCommunicator::outputsToChar(QList<Output> outputs)
116 {
117     unsigned char x = 0x0; // TODO are all low if no other info available?
118
119     const QMap<int,int>& outputIdToBitNumber = Config::getInstance().outputIdToBitNumber(m_communicationType);
120     for( Output& output : outputs )
121     {
122         if(output.isActive())
123         {
124             setBitHigh(x,outputIdToBitNumber.value(output.id()));
125         }
126         else
127         {
128             setBitLow(x,outputIdToBitNumber.value(output.id()));
129         }
130     }
131
132     return x;
133 }

```

Figure 8.3.6 Photo instantanée du code source de la fonction transformante des objets Output en données à écrire

9.Ladder

9.1. Conception

Langage Ladder est un langage graphique très populaire auprès des automatistes pour programmer les automates programmables industriels. Il ressemble un peu aux schémas électriques – Fig. 9.1.1 , et est facilement compréhensible. L'idée initiale du Ladder est la représentation de fonction logique sous la forme de schémas électriques. Par exemple, pour réaliser un ET logique avec des interrupteurs, il suffit de les mettre en série. Pour réaliser un OU logique, il faut les mettre en parallèle.

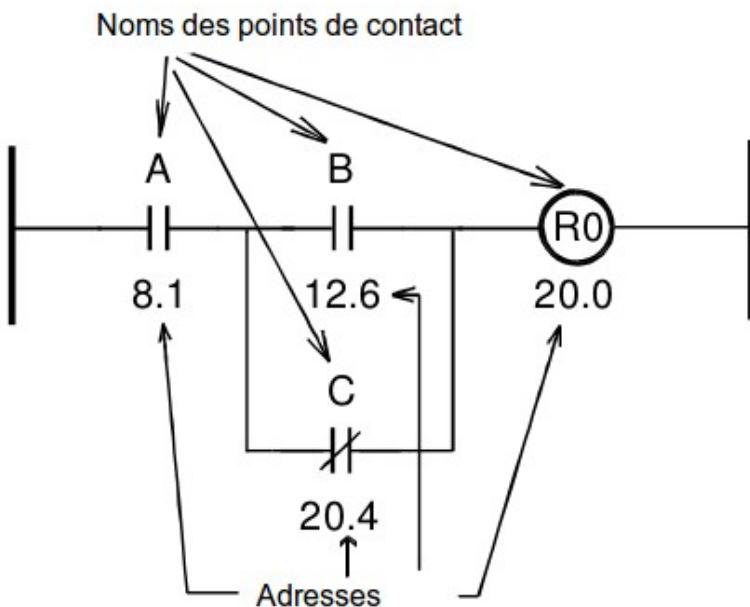


Figure 9.1.1 Schéma ladder

Cette langage premièrement purement graphique est développé en un langage déclaratif par la firme Fanuc - fabricant japonais de robots industriels. En effet c'est plutôt un protocole, qui décrit les fonctions logiques entre les entrées et sorties d'un dispositif. Pour utiliser le langage Ladder on doit fabriquer un interpréteur du protocole, qui dé-sérialise les fichiers textuels et réalise la logique décrite.

Un interpréteur du protocole ladder est réalisé dans le projet « Ladd ». Pour pouvoir analyser le projet on doit premièrement se faire connaître aux règles du langage Ladder.

Il existe 3 types d'éléments de langage :

- les entrées (ou contacts), qui permettent de lire la valeur d'une variable booléenne ;
- les sorties (ou bobines) qui permettent d'écrire la valeur d'une variable booléenne ;
- les blocs fonctionnels qui permettent de réaliser des fonctions avancées.

Les entrées et les sorties sont présentées par des numéros d'identification(adresses), composés de deux parties numériques entières, séparées par un point. La partie deuxième peut avoir une valeur de 0 à 7. Par exemple : 10.3 ; 32.4 ; 40.0.

Les blocs fonctionnels d'autre part sont des chaînes de caractères strictement définis et crucial pour la réalisation du interpréteur. Pour cela on doit les comprendre clairement.

- RD – Lire le signal de l'entrée suivant mentionnée
- RD.NOT - Lire le signal de l'entrée suivant mentionnée et inverser le – Fig.9.1.4
- WRT – Écrire le signal courant dans la sortie suivant mentionnée
- WRT.NOT - Écrire le signal courant inversé dans la sortie suivant mentionnée
- AND – Multiplication logique – Fig. 9.1.2
- AND.NOT – Multiplication logique en inversant le signal de l'entrée suivant mentionnée



Figure 9.1.2 Schéma ladder AND et AND.NOT

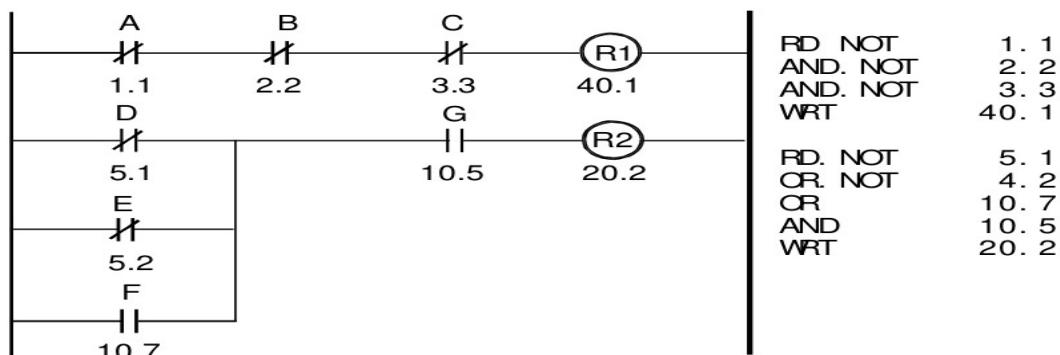


Figure 9.1.3 Schéma ladder OR et OR.NOT

- OR - Addition logique – Fig. 9.1.3
- OR.NOT Addition logique en inversant le signal de l'entrée suivant mentionnée

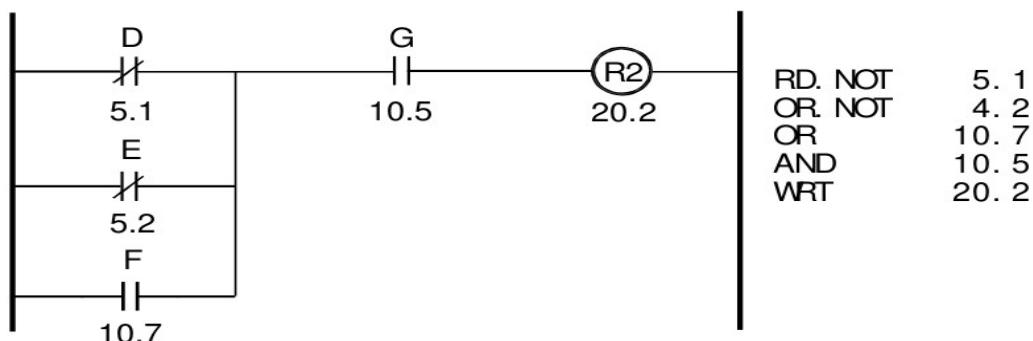


Figure 9.1.4 Schéma ladder RD.NOT

- RD.STK – on peut le regarder comme ouvrage des balises « (« - Fig.9.1.5
- RD.NOT.STK - ouvrage des balises « (« et inversion du signal de l'entrée suivant mentionnée
- AND.STK – fermeture des balises «) » et multiplication du signal résultant dans les balises avec le signal précédent
- OR.STK fermeture des balises «) » et addition du signal résultant dans les balises avec le signal précédent

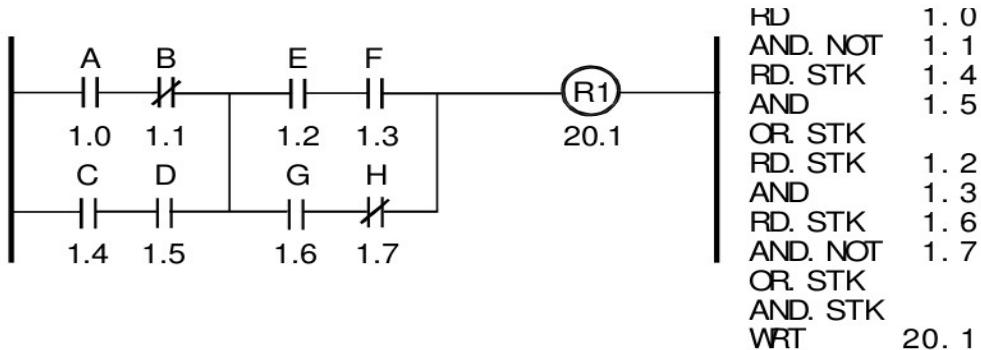


Figure 9.1.5 Schéma ladder RD.STK

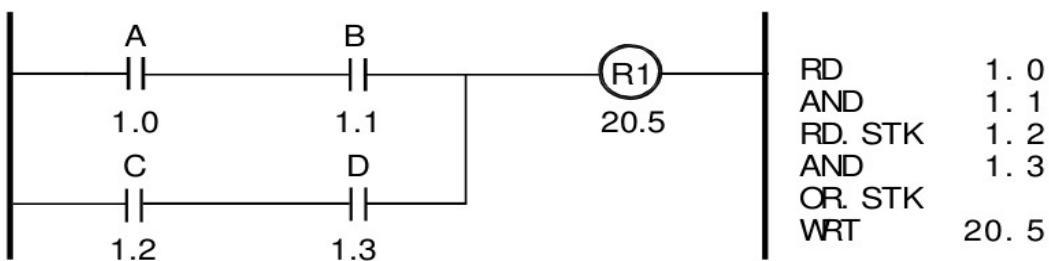


Figure 9.1.6 Schéma ladder OR.STK

- TMR – la valeur suivante ce mot est le numéro d'identification du compteur, qui sert pour réaliser un délai entre les signaux d'entrées et les signaux de sorties correspondant - Fig. 9.1.7

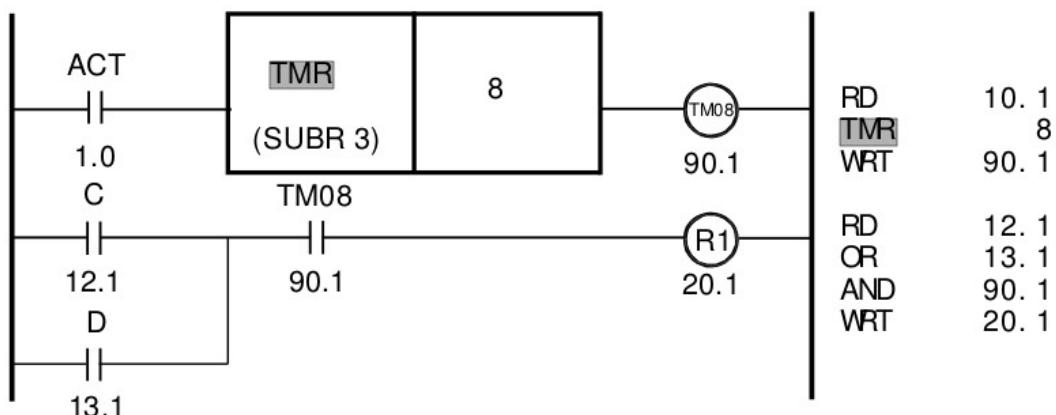


Figure 9.1.7 Schéma ladder TMR

9.2. Structure Ladder dans le projet « Ladd »

9.2.A. Structure générale

La logique ladder dans le projet « ladd » est configurée par la désérialisation d'un fichier d'un extension *.LAD . Les données sont stockées dans un objet de type Ladder. Il contient seulement une liste des objets de type Branch – Fig. 9.2.A.1 . Chaque Branch en effet contient la logique d'une schéma séparée (à voir les exemples précédentes). Cette logique de sa part est présentée par une Liste des entières dans la classe Branch.

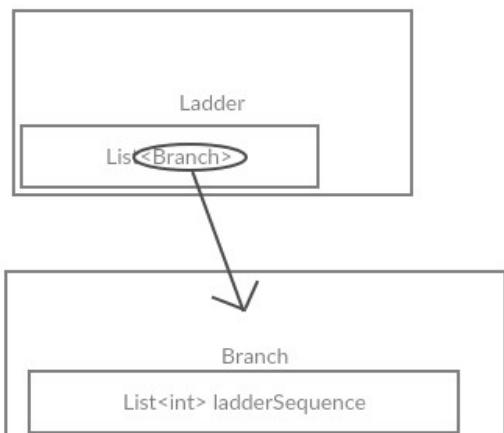


Figure 9.2.A.1 Schéma de la structure de la classe Ladder

Les classes Branch et Ladder ne servent pas seulement à garder d'information – ils traitent aussi les données lues sur les entrées du dispositif. La classe ladder détermine les données correspondant, qui on doit écrire sur les sorties du dispositif. C'est données sont mises dans les classes PendingOutput – Fig. 9.2.A.2.



Figure 9.2.A.2 Schéma du fonctionnement de la classe ladder

La classe PendingOutput hérite la classe Output avec son numéro d'identification entière et isActive booléenne. La nouveauté dans la classe PendingOutput sont des millisecondes notant combien des millisecondes on doit attendre avant la transmission des données vers le dispositif – Fig. 9.2.A.3 .

```
class PendingOutput : public Output
{
public:
    PendingOutput();
    bool isActive() const;
    int m_timeToWait; //msecs
    int timeToWait() const;
    void setTimeToWait(int timeToWait);
    void reduceTimeToWait(int timeWaited);
};
```

Figure 9.2.A.3 Photo instantanée du code source de l'entête de la classe PendingOutput

The diagram illustrates a ladder logic program structure. A vertical line on the left represents the ladder rung, with contacts at the top and coils at the bottom. A horizontal line on the right represents the ladder rail. A curved line labeled "Branch" connects two parallel sections of the ladder. The first section, labeled "Ladder", contains the following instructions:

- RD 20.0 ; NCRDY
- AND.NOT 10.0 ; SRV.KONT
- TMR 1
- WRT 50.0 ; SRVK.TM
- ;
- RD 10.1 ; ESP
- AND.NOT 50.0 ; SRVK.TM
- AND 61.7 ; SVRDY
- AND.NOT 61.1 ; PULT.OFF
- OR 60.4 ; SRV.INI
- WRT 20.0 ; NCRDY
- ;

The second section, also labeled "Ladder", contains the following instructions:

- RD 10.1 ; ESP
- WRT 61.0 ; SRV.RES
- ;
- RD 17.1 ; STOP.B
- OR 64.4 ; MEM.BTNSTOP
- AND.NOT 17.0 ; START.B
- WRT 64.4 ; MEM.BTNSTOP
- ;
- RD 17.0 ; START.B
- OR 61.2 ; START
- AND 60.1 ; MAN
- AND 61.4 ; HM.ENBL
- RD.STK 17.0 ; START.B
- OR 60.0 ; CYCL.ON
- AND 61.5 ; REF.OK
- ;AND 71.3 ; SP_ON.ALARM
- AND.NOT 60.1 ; AUTO
- OR.STK
- AND.NOT 64.4 ; MEM.BTNSTOP
- ;AND.NOT 60.6 ; ESC
- AND 20.0 ; NC.READY
- WRT 61.2 ; START
- ;
- RD 61.2 ; START
- WRT 24.0 ; START.L
- ;
- RD 60.1 ; MANUAL
- AND.NOT 63.2 ; JOG.L
- WRT 24.2 ; JOG.L
- ;

Figure 9.2.A.2 Exemple d'un fichier de ladder

9.2.B Liste des entières – ladderSequence

Le contenu du chaque branch est présenté dans la structure comme en séquence des entières. Ça veut dire que chaque opérande (numéro d'identification) et chaque opérateur (logique – addition, multiplication, ou action – écrire, lire) a une représentation en entière.

On a déjà analyser comment les numéros d'identification prennent une forme entière – la même fonction utilisée pour leur déserialisation dans la configuration idToInt est utilisée aussi ici. Les opérateurs logiques d'autre part sont présentées par des types énumérées – Fig. 9.2.B.1.

```
17 enum Operation
18 {
19     Read = INT_MIN,
20     ReadNot,
21     Write,
22     WriteNot,
23     And,
24     AndNot,
25     Or,
26     OrNot,
27     ReadStack,
28     ReadNotStack,
29     AndStack,
30     OrStack,
31     Timer,
32     Invalid
33 };
34
```

Figure 9.2.B.1 Photo instantanée du code source du type énuméré Operation

On choisi démarrer l'énumération de INT_MIN – la valeur entière minimale. Comme ça on ne peut pas avoir un énumérateur et un numéro d'identification avec les mêmes valeur, parce que les numéro d'identification sont toujours positifs.

L'ordre dans lequel on mis les entières et très important pour le fonctionnement du projet. L'ordre s'appelle notation polonaise inverse.

La notation polonaise inverse (NPI) (en anglais RPN pour Reverse Polish Notation), également connue sous le nom de notation post-fixée, permet d'écrire de les formules arithmétiques sans utiliser de parenthèses. Dérivée de la notation polonaise présentée en 1920 par le mathématicien polonais Jan Łukasiewicz, elle s'en différencie par l'ordre des termes, les opérandes y étant présentés avant les opérateurs et non l'inverse.

Par exemple, l'expression « $3 \times (4 + 7)$ » peut s'écrire en NPI sous la forme « $4\ 7 + 3 \times$ », ou encore sous la forme « $3\ 4\ 7 + \times$ ».

Évaluer une expression post-fixée est facile. Pour cela il suffit de lire l'expression de gauche à droite et d'appliquer chaque opérateur aux deux opérandes qui le précédent. Si l'opérateur n'est pas le dernier symbole on replace le résultat intermédiaire dans l'expression et on recommence avec l'opérateur suivant.

La réalisation d'une calculatrice NPI se fera à l'aide d'une pile.

L'algorithme est très simple. On commence par lire un par un les caractères de l'expression. Si le caractère lu est un opérande alors on l'empile. Si le caractère lu est un opérateur, alors on dépile les deux éléments se trouvant en haut de la pile, on calcule le résultat en appliquant l'opérateur sur les deux opérandes dépilerés et on empile le résultat. Une fois tous les caractères lus, la pile ne contient qu'un seul élément qui correspond au résultat final.

Voyons avec un exemple l'état de la pile après la lecture de chaque caractère de l'expression $((1+2)*4)+3$, ou $1\ 2\ +\ 4\ *\ 3\ +$ en NPI – Fig. 9.2.B.2 .

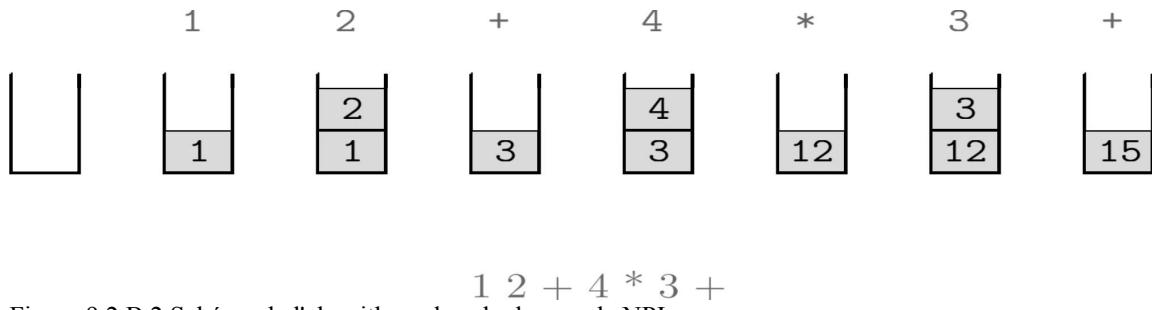


Figure 9.2.B.2 Schéma de l'algorithme de calcul avec le NPI

En pratique sur une calculatrice à NPI le calcul sera saisi en tant que :

« 1 », « entrée » ou « espace », « 2 », « + », « 4 », « × », « 3 », « + » ou « 3 », « entrée » ou « espace », « 4 », « entrée » ou « espace », « 1 », « entrée » ou « espace », « 2 », « + », « × », « + » (on observe que la première séquence nécessite moins de pressions de touches !)

L'expression est évaluée de la façon suivante (la pile est montrée après chaque opération. Elle est représentée dans le sens physique - le dernier élément de la pile en haut, bien que de nombreuses calculatrices placent l'élément le plus récent en bas pour des raisons d'ergonomie) :

	Entrée	Opération	Pile
Étape n° 1	1	Pousser l'opérande	1
Étape n° 2	2	Pousser l'opérande	2 1
Étape n° 3	+	Addition	3
Étape n° 4	4	Pousser l'opérande	4 3
Étape n° 5	×	Multiplication	12
Étape n° 6	3	Pousser l'opérande	3 12
Étape n° 7	+	Addition	15

Figure 9.2.B.3 Schéma de l'algorithme de calcul avec le NPI

Cette technique a plusieurs avantages :

- l'ordre des opérandes est préservé ;
- les calculs se font en lisant l'expression de gauche à droite ;
- les opérandes précèdent l'opérateur et l'expression qui décrit chaque opérande disparaît lorsque l'opération est évaluée.

Mais aussi des désavantages :

On ne peut exécuter un opérateur que s'il est de façon univoque binaire ou unaire, c'est-à-dire opère sur deux arguments ou un. Il faut donc différencier l'opérateur binaire de soustraction ($10 - 2$ devient $10 2 -$) de l'opérateur unaire de négation ($- 2$ devient 2 NEG). Plus généralement un opérateur doit prendre un nombre fixe d'arguments (il existe des opérateurs ternaires, quaternaires...) ou prendre un nombre fixe d'argument décrivant les autres arguments consommés par l'opérateur. Ainsi la fonction DROPN (HP48) consomme un premier argument dans la pile (un entier) qui lui donne le nombre des autres arguments à consommer (en l'occurrence le nombre d'éléments à retirer de la pile).

9.3. Désrialisation du ladder

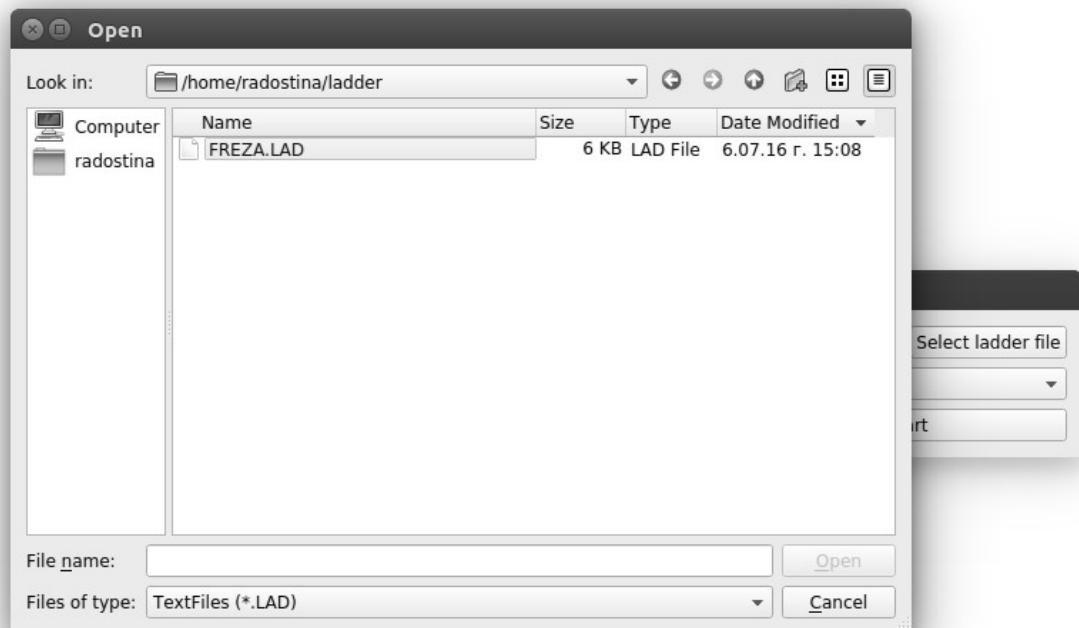


Figure 9.3.1 Interface utilisateur du chargement du fichier de ladder

On peut voir que en ouvrant le dialogue système pour choisir un fichier ladder dans le champ du texte « Files of type » on a une description « TextFiles (*.LAD) » - fichier textuelle avec une extension du type *LAD – Fig. 9.3.1 . C'est une description correspondant exactement au fichier que on cherche.

Quand on ouvre le dialogue dans le code source il y a une vérification du type de fichier exigé par le programme en ce moment là et les paramètre correspondant sont faites passés vers le dialogue.

Pour transmettre le chemin du fichier vers le MainController on utilise absolument la même technologie comme dans le cas de la désérialisation de la configuration – Fig.9.3.2 . Il y a une différence significative concernant la manière, dans laquelle on sauvegarde la structure dé-sérialisée. La configuration est un objet singleton et il n'y a pas de sens que le ConfigParser retourne une valeur de type Config – on simplement peut accéder l'objet statique Config et fait des ajustement sur ses paramètres selon les valeurs obtenues par le fichier config.xml.

Dans le cas d'objet du type Ladder on n'utilise pas la schéma du singleton, parce que dans le développement futur du projet on envisage que la communication en parallèle et en série peuvent exister simultanément. Alors on sauvegarde l'objet Ladder comme un membre du contrôleur de communication – CommunicationController

```
44 ▼ void MainController::onFileLoaded(const QString filePath, SettingsPage::FileType fileType)
45 {
46 ▼   if(fileType == SettingsPage::FileLadder)
47   {
48     Ladder ladder = LadderParser::parseLadder(filePath);
49     m_communicationController->setLadder(ladder);
50   }

```

Figure 9.3.2 Photo instantanée du code source appelant la fonction pour la désérialisation du fichier ladder

La classe, qui dé-sérialise le fichier .LAD comporte des fonctions exclusivement statiques – alors on n'a pas besoin d'un instance de la classe pour traiter le fichier.

```
33 ▼ Ladder LadderParser::parseLadder(const QString &filePath)
34 {
35   Ladder ladder;
36   QFile file(filePath);
37 ▼   if(!file.open(QIODevice::ReadOnly))
38   {
39     return ladder;
40   }
41
42
43 ▼   while(!file.atEnd())
44   {
45
46     Branch branch = readBranch(file);
47     if(!branch.isEmpty())
48     {
49       ladder.addBranch(branch);
50     }
51
52     // TODO description display (alarm, overload, etc)
53   }
54
55   return ladder;
56 }
57

```

Figure 9.3.3 Photo instantanée du code source de la logique désérialisant des branch séparés dans la fonction parseLadder

Dans la fonction statique parseLadder – on initialise un objet de type Ladder. Ce objet va être la résultat retournée même si dans le cas d'un erreur pendant la désérialisation.

On fait passer le chemin du fichier au constructeur d'un objet du type QFile.

La classe QFile fournit une interface pour la lecture et l'écriture des fichiers.

QFile représente un dispositif d'Entré/Sortie pour la lecture et l'écriture du texte, des fichiers binaires et des ressources. Un QFile peut être utilisé par lui-même ou, mais c'est plus commode

avec l'assistance d'un objet de type QTextStream ou QDataStream.

Le nom du fichier est généralement passé dans le constructeur, mais il peut être changé dans chaque moment en utilisant la fonction setFilename (). QFile attend le séparateur de fichier pour être '/' indépendamment du système d'exploitation. L'utilisation des autres séparateurs (par exemple, '\') ne sont pas pris en charge.

On peut vérifier l'existence d'un fichier en utilisant existe (), et de supprimer un fichier en utilisant remove (). Opérations plus complexes liées au système des fichiers sont fournis par QFileinfo et QDir.

Le fichier est ouvert avec open (), fermée avec close (), et le bouffer est rincée avec flush (). Les données sont généralement lues et écrites en utilisant QDataStream ou QTextStream, mais on peut également appeler les fonctions héritées de QIODevice - read (), readLine (), readAll (), écrire (). QFile hérite également getChar (), putChar () , et ungetChar () , qui fonctionne un caractère à la fois.

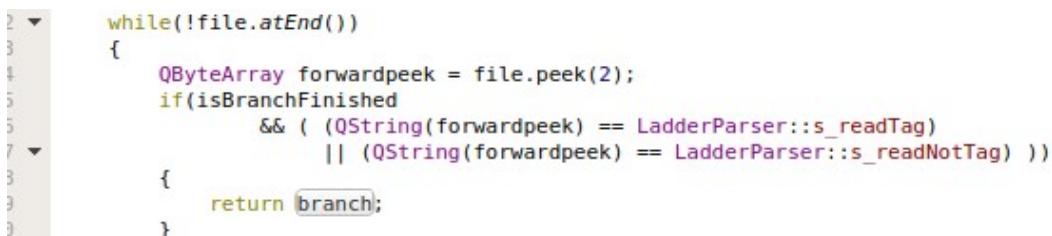
La taille du fichier est accessible par la fonction taille (). On peut obtenir la position actuelle du fichier en utilisant pos (), ou passer sur une nouvelle position de fichier en utilisant seek (). Quand on atteint la fin du fichier, atEnd () retourne vrai.

On voit dans le capture d'écran présenté, que on ne fait pas appel à la fonction close() , parce que quand l'objet QFile et détruit on fera automatiquement la fermeture du fichier associé avec l'objet.

Dans la boucle while() on fait la vérification si la fin de fichier est atteinte et quand c'est vrai on interrompe la boucle et retourne l'objet Ladder.

En regardant le code exécutable dans la boucle on peut remarquer que la responsabilité de séparer les différents Branch l'un d'autre pendant la désérialisation du fichier *.LAD appartient aux objets Branch elles mêmes – 9.3.3 . On lui les fait passer un objet QFile et excepte que quand ils auront obtenu l'information pour un seul Branch il lui retourneront – sans lisant tout le fichier jusqu'à la fin.

En effet on sépare deux branch l'un d'autre en utilisant l' opérateur RD (read – lire) et RD.NOT (read not – lire l'inverse) – Fig. 9.3.4.



```
2 while(!file.atEnd())
3 {
4     QByteArray forwardpeek = file.peek(2);
5     if(isBranchFinished
6         && ( (QString(forwardpeek) == LadderParser::s_readTag)
7             || (QString(forwardpeek) == LadderParser::s_readNotTag) )
8     {
9         return branch;
10    }
11 }
```

Figure 9.3.4 Photo instantanée du code source de la logique désérialisant des branch séparés dans la fonction readBranch

Seulement dans le début d'un branch on place ces opérateurs. Mais d'autre part dans le début de la désérialisation du Branch première on trouve aussi un opérateur RD ou RD.NOT . Pour pouvoir différencier entre le RD première et les autres RD à venir on ajoute encore un règle à vérifier. La valeur booléenne isBranchFinished doit être vraie – Fig. 9.3.5.

```

92     if(value == LadderEntities::Operation::Write
93         || value == LadderEntities::Operation::WriteNot
94         || value == LadderEntities::Operation::Timer)
95     {
96         isBranchFinished = true;

```

Figure 9.3.5 Photo instantanée du code source de la logique de signaliser que on a déjà tous les signaux nécessaires

Quand on trouve un opérateur WRT ou WRT.NOT ça veut dire que le Branch arrive à sa fin et la fois prochaine quand on trouve un opérateur RD – il appartient au Branch suivant – Fig. 9.3.5.

Alors en lisant un Branch on a besoin de savoir en avance si la chaîne de caractères sur la ligne suivante commence avec RD pour pouvoir interrompre la boucle et retourner l'objet Branche et le QFile. De toute façon la classe QFile elle même offre une fonction très convenable pour cette situation – forwardpeek().

```

71
72     QString line(file.readLine());
73     QString itemString = line.split(LadderParser::s_commentSymbol).takeFirst();
74
75     // ...

```

Figure 9.3.6 Photo instantanée du code source de la logique de séparation des données utiles aux commentaires

Après avoir vérifier que la fin du Branch n'est pas arrivée on lit une ligne entière du fichier. Puis on sépare l'information utilisable des commentaires. Le caractère, qui signalise un commentaire est « ; » - Fig. 9.3.6 . Tous les caractères après lui sont pas utiles pour le logiciel, mais servent au opérateur du programme ou le développeur. Alors on sépare la chaîne de caractère lire par le caractère « ; ». On prend la première des chaînes de caractères obtenues après la séparation.

```

79     QStringList items = itemString.split(QRegExp("\\"s"), QString::SkipEmptyParts);
80     QMap<QString, LadderEntities::Operation> operationsTagMap = LadderParser::operationsMap();
81
82     QStringList::iterator iter = items.begin();
83     while(iter != items.end())
84     {
85         int value = operationsTagMap.value(*iter, LadderEntities::s_invalidOperation);

```

Figure 9.3.7 Photo instantanée du code source de la transformation une chaîne de caractères en une listes des entières

Puis on sépare cette chaîne de caractère par le caractère « \s », qui signifie un espace blanc – Fig. 9.3.7 . Chaque opérande et opérateur est séparé aux autres par un espace blanc – alors le liste des chaînes de caractères obtenues dans QStringList items est un effet un liste des opérandes et opérateur séparés.

Pour les convertir en valeurs de type énuméré on utilise un QMap, contenant des chaînes de caractères de tous des opérateurs d'une part et d'autre part – ses valeurs énumérées correspondantes – Fig. 9.3.8. Dans les cas où le clé recherché ne se trouve pas dans le QMap operationsMap(), la valeur renournée est s_invalidOperation. On doit faire attention sur le fait que les opérateur RD et RD.STK ne font partie de cette QMap. Ils ne jouent aucune rôle dans exécution de la logique ladder, particulièrement quand elle est écrite sous forme de notation polonaise inverse. Pour cette raison on les ignore.

```

142     QMap<QString, LadderEntities::Operation> LadderParser::operationsMap()
143     {
144         QMap<QString, LadderEntities::Operation> result;
145         result.insert(s_writeTag, LadderEntities::Operation::Write );
146         result.insert(s_writeNotTag, LadderEntities::Operation::WriteNot );
147         result.insert(s_andTag, LadderEntities::Operation::And );
148         result.insert(s_andNotTag, LadderEntities::Operation::AndNot );
149         result.insert(s_orTag, LadderEntities::Operation::Or );
150         result.insert(s_orNotTag, LadderEntities::Operation::OrNot );
151         result.insert(s_readNotStackTag, LadderEntities::Operation::ReadNotStack );
152         result.insert(s_andStackTag, LadderEntities::Operation::AndStack );
153         result.insert(s_orStackTag, LadderEntities::Operation::OrStack );
154         result.insert(s_timerTag, LadderEntities::Operation::Timer );
155     return result;

```

Figure 9.3.8 Photo instantanée du code source de l'arbre rouge-noir des Operations

L' énumérateur du type Operation correspondant au opérateur est sauvegarder dans la variable entière value, mais seulement dans les cas où il est différent du s_invalidOperation. Dans les cas où l'opération est invalide ça signifie, qu'il ne s'agit pas d'un opérateur, mais d'un opérande – alors un numéro d'identification qu'on doit transformer en entière et ajouter au ladderSequence.

```
86 ►     if(value != LadderEntities::s_invalidOperation) {...}  
125 ▼     else  
126     {  
127         // operand Value  
128         value = FileParser::idToInt(*iter);  
129         if(value != Config::s_invalidInt)  
130         {  
131             branch.addLadderSequence(value);
```

Figure 9.3.9 Photo instantanée du code source de la transformation des caractères en un type énuméré

Dans les cas où il s'agit d'un opérateur logique (ET , OU etc.) - avant de l'enregistrer on doit premièrement lire le numéro d'identification qui suit l'opérateur. La notation polonaise inverse exige que l'opérande soit enregistrer avant du opérateur alors on doit premièrement enregistrer le numéro d'identification et après – l'opérateur logique – Fig. 9.3.10 .

```
99
100
101     ++iter;
102     if(iter != items.end())
103     {
104         QString operandString = *iter;
105         int operand = FileParser::idToInt(operandString);
106         if (operand == Config::s_invalidInt)
107         {
108             ++iter;
109             continue;
110         }
111         branch.addLadderSequence(operand);
112     }
113     else
114     {
115         continue;
116     }
117 }
118
119     if (!isBranchFinished)
120     {
121         branch.addLadderSequence(value);
122     }
123
124 }
```

Figure 9.3.10 Photo instantanée du code source de la logique ladder

Seulement dans les cas où il s'agit des opérateurs WRT, WRT.NOT et TMR – on ne respecte pas la notation polonaise inverse et sauvegarde premièrement l'opérateur et après ça – l'opérande – 8.3.11 . Les raisons pour cela restent dans les détails de l'implémentation du fonctionnement du ladder.

```
92         if(value == LadderEntities::Operation::Write  
93             || value == LadderEntities::Operation::WriteNot  
94             || value == LadderEntities::Operation::Timer)  
95     {  
96         isBranchFinished = true;  
97         branch.addLadderSequence(value);  
98     }
```

Figure 9.3.11 Photo instantanée du code source de la logique ladder.

9.4. Fonctionnement

Quand on lit les données d'entrée du dispositif on fait appel au objet Ladder et sa fonction treatInputs – Fig. 9.4.1 . L'objet ladder du sa part traverse dans une boucle for() tous les objets Branch en les passant l'ensemble des données d'entrée. Chaque Branch traite les données primaire et retourne une liste des objets du type PendingOutput. À son tour l'objet Ladder retourne une liste cumulée de tous les autres.

```
13 ▼ QList<PendingOutput> Ladder::treatInputs(QMap<int, bool> &inputs)
14 {
15     QList<PendingOutput> result;
16     for(const Branch& branch : m_branches)
17     {
18         result.append(branch.treatInputs(inputs));
19     }
20
21     return result;
22 }
```

Figure 9.4.1 Photo instantanée du code source de la fonction treatInputs0

La logique du ladder est effectuée dans les objets Branch. La on initialise une liste des valeurs booléennes - passedValues. Cette liste sert comme une queue basée sur le principe FIFO – PEPS (premier entré premier sorti). À la fin du traitement des données d'entrées il faut que dans la liste passedValues il reste seulement une valeur booléenne – le résultat des opérations logiques sur les données primaire. Ce résultat doit être attaché à un ou plusieurs numéros d'identification – formant une structure de PendingOutput.

Pour arriver a cela, on traverse chaque entière de la séquence des entières (NPI) dé-sérialisées du fichier *LAD pour ce Branch. Pour chaque valeur de la séquence positive on cherche la valeur correspondant entre les clés de QMap passé comme argument de la fonction. Les valeurs positifs sont les valeur opérandes – les valeurs représentantes un numéro d'identification. Le QMap<int,bool> nommé inputs aussi contient des valeurs représentants un numéro d'identification – notamment les numéros d'identification des entrées dont les données sont lues. À chaque clé dans le QMap il y a une valeur booléenne correspondante. Cette valeur représente l'état dans lequel l'entrée se trouve – activée ou pas activée. Dans la section concernant la configuration on a déjà analyser profondément comment ce QMap des entrées et obtenue.

Quand on trouve une valeur correspondante à une entière positive de la séquence de Branch on l'enregistre dans la liste passedValues.

```
20 ▼ QList<PendingOutput> Branch::treatInputs(QMap<int, bool> &inputs) const
21 {
22     QList<PendingOutput> result;
23     QList<bool> passedValues;
24
25     PendingOutput finalOutput;
26
27     QList<int>::const_iterator iter;
28     for(iter = m_ladderSequence.constBegin();
29         iter != m_ladderSequence.end();
30         iter++)
31     {
32         if(*iter >= 0)
33         {
34             if(!inputs.contains(*iter))
35             {
36                 return result;
37             }
38
39             passedValues.append(inputs.value(*iter, s_defaultInputValue));
40             continue;
41         }
42     }
43 }
```

Figure 9.4.2 Photo instantanée du code source de la fonction treatInputs0

Ce qui concerne les opérations logiques on utilise une instruction switch() case : - Fig. 9.4.3 . Après vérifiant que la valeur de l'entièrre actuelle est moins que zéro passe l'entièrre à une instruction switch. Les cas possibles sont les opérations :

```

44     switch (*iter)
45 {
46     case LadderEntities::Operation::Write :
47     case LadderEntities::Operation::WriteNot: [....]
48     case LadderEntities::Operation::ReadNotStack: [....]
49     case LadderEntities::Operation::Timer: [....]
50     case LadderEntities::Operation::And:
51     case LadderEntities::Operation::AndStack:
52     case LadderEntities::Operation::AndNot: [....]
53     case LadderEntities::Operation::Or:
54     case LadderEntities::Operation::OrNot:
55     case LadderEntities::Operation::OrStack: [....]
56     default:
57         break;
58 }

```

Figure 9.4.3 Photo instantanée du code source de la fonction treatInputs0 – le switch des opérations différentes

- cas RD.NOT.STK - Fig. 9.4.4

Dans ce cas on veut invertir la valeur dernière de la liste, parce que l'opérateur se réfère à lui.

Premièrement on vérifie la quantité des éléments dans la liste. Puis on soustrait 1 du nombre des éléments et en utilisant l'opérateur [] on accède l'élément dernière de la liste. La même est valable pour la fonction last() mais avec la différence que last() est une fonction constante – alors on ne peut pas alterner la valeur d'élément accédé par elle.

```

71     case LadderEntities::Operation::ReadNotStack:
72 {
73     int size = passedValues.size();
74     int lastValuePos = size-1;
75     passedValues[lastValuePos] = !passedValues.last();
76     break;
77 }

```

Figure 9.4.4 Photo instantanée du code source de cas Read Not Stack

- cas WRT, WRT.NOT – Fig. 9.4.5

On fait la vérification que dans la liste passedValues il reste seulement une valeur. On prend cette valeur et la mit comme isActive du PendingOutput résultant – finalOutput. C'est possible que dans une branche on as plusieurs des PendingOutputs résultants. De toute façon chaque de ces PendingOutputs doit contenir la variable des millisecondes à attendre avant d'écrire les données sur le dispositif. Alors on réutilise la variable finalOutput en lui changeant le numéro d'identification et la valeur booléenne isActive mais gardant la valeur timeToWait. Une fois ajouter à la liste des PendingOutput result, on n'a pas besoin de préserver la variable finalOutput dans sa forme courante et on la change pour le PendingOutput suivant sans perdant d'information.

```

46
47     case LadderEntities::Operation::Write :
48     case LadderEntities::Operation::WriteNot:
49     {
50         bool outputValue;
51         if(passedValues.size() != 1)
52         {
53             return result;
54         }
55
56         outputValue = passedValues.first();
57         if(*iter == LadderEntities::Operation::WriteNot)
58         {
59             outputValue = !outputValue;
60         }
61
62         finalOutput.setIsActive(outputValue);
63         iter++;
64         if(iter != m_ladderSequence.end())
65         {
66             finalOutput.setId(*iter);
67         }
68
69         result.append(finalOutput);
70         break;
71     }

```

Figure 9.4.5 Photo instantanée du code source de cas Write

- cas AND, AND.STK, AND.NOT, OR, OR.NOT, OR.STK – Fig. 9.4.6

La logique dans les cas AND et OR est presque la même. On prend les deux valeurs dernières de la liste passedValues. On fait une inversion de la valeur dernière s'il y en a besoin (AndNot et OrNot). Puis on exécute l'opération logique – addition ou multiplication, entre les deux valeur et ajoute le résultat à la liste passedValues.

```

96     case LadderEntities::Operation::And:
97     case LadderEntities::Operation::AndStack:
98     case LadderEntities::Operation::AndNot:
99     {
100        if(passedValues.size() < 2)
101        {
102            return result;
103        }
104
105        bool lastValue = (*iter == LadderEntities::Operation::AndNot)
106        ? ! passedValues.takeLast()
107        : passedValues.takeLast();
108        bool beforeLastValue = passedValues.takeLast();
109
110        bool resultValue = (lastValue && beforeLastValue);
111        passedValues.append(resultValue);
112        break;
113    }

```

Figure 9.4.6 Photo instantanée du code source de cas Read Not Stack

- cas TMR – Fig. 9.4.7

Dans ce cas on prend la valeur suivante de la séquence m_ladderSequence. Elle doit représenter le numéro d'identification de timer, qui détermine la quantité de millisecondes, qui on doit attendre avant d'écrire les données sur les sorties du ce Branch. Alors on fait appel à la configuration globale statique pour obtenir la valeur des millisecondes correspondante au numéro d'identification. On met cette valeur comme temps d'attendre de la variable finalOutput, qui on va retourner comme résultat de la fonction.

```

78     case LadderEntities::Operation::Timer:
79     {
80         iter++;
81
82         if(iter != m_ladderSequence.end())
83         {
84             // TODO communication type
85             int milisecsToWait = Config::getInstance().timerIntervalForId(*iter, Config::Parallel);
86             if(milisecsToWait == LadderEntities::s_invalidTime)
87             {
88                 continue;
89             }
90             finalOutput.setTimeToWait(milisecsToWait);
91         }
92         break;
93     }
94
95 
```

Figure 9.4.7 Photo instantanée du code source de cas Timer

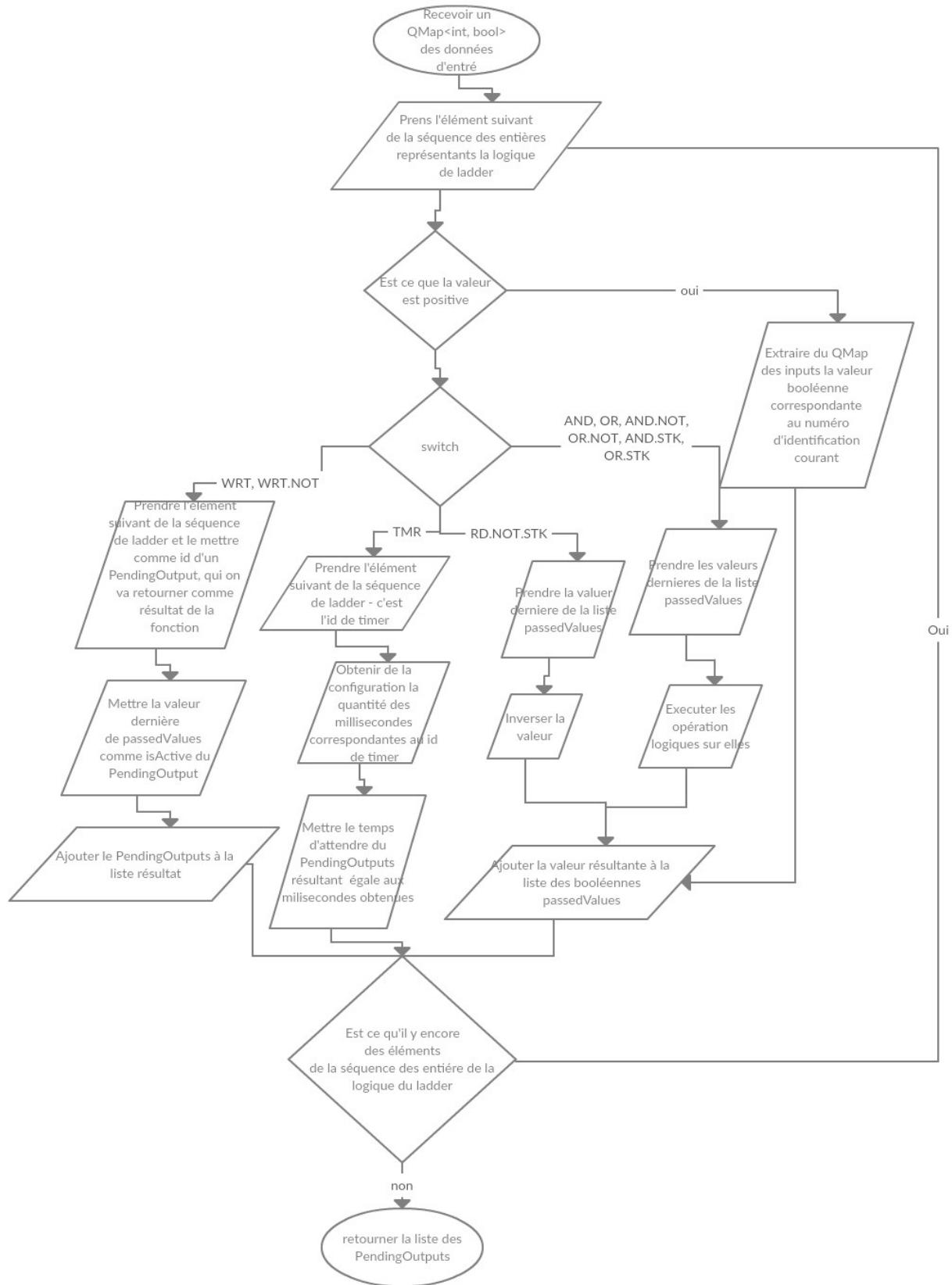


Figure 9.4.4 Schéma du fonctionnement de ladder

10. Communication avec le dispositif

10.1. Implémentation générale

La communication entre le dispositif et le PC dans le logiciel est un processus périphérique, qui se produit chaque 50 millisecondes. C'est une fréquence relativement bas et ne pose aucune problème ni au système d'exploitation ni au développeur du logiciel.

Le logiciel « Ladd » est actuellement implémenté pour effectuer communication par une interface parallèle ou une interface en série. De tout façon les interfaces USB et Ethernet sont aussi envisagées à implémenter dans le développement futur du logiciel. Alors l'architecture doit être conçue bien flexible ce qui concerne les interfaces différentes de communication.

Etan donné qu'il y a des dispositifs variés, avec lesquels on communique par des interface différentes, l'implémentation de la classe communicateur et fondé sur une classe de base - BaseCommunicator avec des fonctions virtuelles pures, quelles chaque classe héritante doit implémenter à sa part. Dans le contrôleur de communication (CommunicationController) on sauvegarde une référence ver un objet de type BaseCommunicator et l'utilise pour réaliser le processus de communication. Comme ça on peut facilement changer le dispositif et l'interface de communication sans même une touche sur le code source du contrôleur.

On doit aussi faire attention sur le fait que la communication avec les dispositif est structurée dans une manière asynchrone avec l'aide des signaux et des slots. La classe Base communicateur offre les deux possibilités – synchrone et asynchrone, pour lire les entrées des dispositifs, mais dans le contrôleur de communication on utilise la manière asynchrone, parce que la communication avec le port série est asynchrone.

On peut différencier trois étapes de la communication – sélection d'interface de communication, début de communication, boucle de communication.

- Sélection d'interface de communication – Fig. 10.1.1

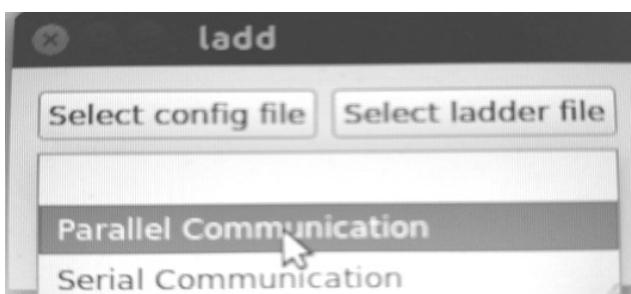


Figure 10.1.1 Photo instantanée d'interface utilisateur à changer les modes de communication

Pour sélectionner l'interface de communication on utilise la classe de Qt QComboBox.

Le widget QComboBox est combinaison d'un bouton et une liste pop-up. QComboBox fournit le moyen de présenter une liste d'options de la manière qui prend le minimum d'espace d'écran. Un QComboBox est un widget de sélection qui affiche l'élément couronnement sélectionné, mais qui peut faire apparaître une liste d'éléments sélectionnables.

Il y a deux signaux émis si l'élément courant d'un QComboBox est changé - currentIndexChanged() et activé (). currentIndexChanged () est toujours émise n'importe si le changement a été fait par le logiciel ou par une interaction de l'utilisateur, tandis que activé () est émis seulement quand le changement est du d'une interaction de l'utilisateur.

Dans le logiciel « Ladd » on utilise la fonction currentIndexChanged(). Et en initialisant le QComboBox on ajoute trois éléments – un élément vide, un avec le texte « Parallel Communication » et un - « Serial Communication ». Selon l'ordre dans lequel ils sont ajouté, l'élément parallèle et associé au index 1 et l'élément série au index 2 – Fig.10.1.2 .

```

30     m_communicationTypeSelector = new QComboBox();
31     m_communicationTypeSelector->addItem("");
32     m_communicationTypeSelector->addItem("Parallel Communication");
33     m_communicationTypeSelector->addItem("Serial Communication");
34     connect(m_communicationTypeSelector, SIGNAL(currentIndexChanged(int)), this, SLOT(onCommunicationTypeChanged(int)));
35

```

Figure 10.1.2 Photo instantanée du code source de la configuration de ComboBox

Selon l'index d'élément sélectionné on établir si c'est l'interface parallèle ou en série. Puis on émit le signal communicationChanged() avec l' énumérateur correspondant au type de la communication – Fig. 10.1.3 .

```

123 void SettingsPage::onCommunicationTypeChanged(int index)
124 {
125     if(index == 1)
126     {
127         emit communicationChanged(Config::CommunicationType::Parallel);
128     }
129     else if(index == 2)
130     {
131         emit communicationChanged(Config::CommunicationType::Serial);
132     }
133 }
134

```

Figure 10.1.3 Photo instantanée du code source du traitement du signal de changement des index

On connecte le signal de UI à la slot du CommunicationController dans le constructeur du MainController – Fig. 10.1.3.

```

16     connect(m_settingsPage,
17             SIGNAL(communicationChanged(Config::CommunicationType)),
18             m_communicationController,
19             SLOT(setupCommunication(Config::CommunicationType)));

```

Figure 10.1.3 Photo instantanée du code source de connexion de signal communicationChanged au MainController

Dans le contrôleur de communication on initialise le communicateur correspondant au type de communication sélectionnée – Fig. 10.1.4 . Avant de faire l'initialisation on arrête la communication précédente. Dans cette situation c'est très importante d'effacer l'objet vers lequel le pointeur m_baseCommunicator fait référence. Ça se passe dans la fonction stopCommunication(). Si on omit cet effacement il y aura une fuite de mémoire, ce qui est une faute grave. On efface aussi l'objet QTimer, quel on va analyser dans le point suivant.

```

156 void CommunicationController::setupCommunication(Config::CommunicationType type)
157 {
158     stopCommunication();
159
160     if(type == Config::CommunicationType::Parallel)
161     {
162         m_baseCommunicator = new ParallelCommunicator();
163     }
164     else if(type == Config::CommunicationType::Serial)
165     {
166         m_baseCommunicator = new SerialCommunicator();
167     }
168 }

```

Figure 10.1.4 Photo instantanée du code source de configuratin de ComboBox

- Début de communication

Le signal clicked() bouton « Start Communication » de UI est lié avec la fonction de contrôleur de communication - startCommunication() - Fig. 10.1.5 .

```

23
24 void CommunicationController::startCommunication()
25 {
26     if(m_baseCommunicator == nullptr)
27     {
28         return;
29     }
30
31     m_baseCommunicator->initiateCommunication();
32     connect(m_baseCommunicator, SIGNAL(inputsRead(QMap<int,bool>)), this, SLOT(onInputsRead(QMap<int,bool>)));
33
34     if(m_timer != nullptr)
35     {
36         stopAndDeleteTimer();
37     }
38
39     m_timer = new QTimer();
40     m_timer->setSingleShot(false);
41     setupTimerInterval();
42     connect(m_timer, SIGNAL(timeout()), this, SLOT(onInterrupt()));
43     m_timer->start();
44 }
```

Figure 10.1.5 Photo instantanée du code source de la fonction startCommunication dans la classe MainController

Pour pouvoir démarrer la communication on doit déjà avoir choisir le type de communication (serie ou parallèle) et avoir initialisé le communicateur m_baseCommunicator. On initialise la communication. C'est un processus différent pour chaque communicateur et on va l'analyser séparément. On connecte le signal de communicateur inputsRead() avec le slot du contrôleur onInputsRead(). On utilise signal et slot, parce que la communication entre le logiciel et les dispositifs est asynchrone.

Puis on initialise le compteur m_timer. Le compteur a pour but de ménager la fréquence de la communication entre l'ordinateur et les dispositif. Quand on veut que chaque 50 millisecondes le logiciel lis les entrées du dispositif et écrit sur ses sorties on configure le délais du compteur sur 50 millisecondes et connecte son signal timeout() avec un slots du contrôleur onInterrupt(). Dans la fonction onInterrupt() on vas démarrer une session de communication (lire – traiter par ladder – écrire) avec le dispositif chaque 50 millisecondes.

La classe QTimer fournit les compteurs (répétitifs ou single-shot (une seul exécution)) avec des interruptions à la fin de délais (timeout). La classe QTimer fournit une interface de programmation de haut niveau pour les compteurs pérennant en compte les détails spécifiques du système d'exécution ce qui concerne le travail avec le matériel d'ordinateur (dans ce cas - ce qui concerne le travail avec les compteurs).

Pour utiliser la classe on doit créer un objet QTimer, connecter son signal timeout() à un ou plusieurs slots, et appeler start(). Alors le compteur émettra un signal timeout() à des intervalles constants, mais pour être sûr que le compteur n'est pas single – shot, on mit cette valeur booléenne explicitement comme faux (setSingleShot(false)). La grandeur de ces intervalles est estimée par la variable timerInterval, qui on peut changer à l'aide de la fonction de QTimer setInterval().

- Boucle de communication – Fig. 10.1.6

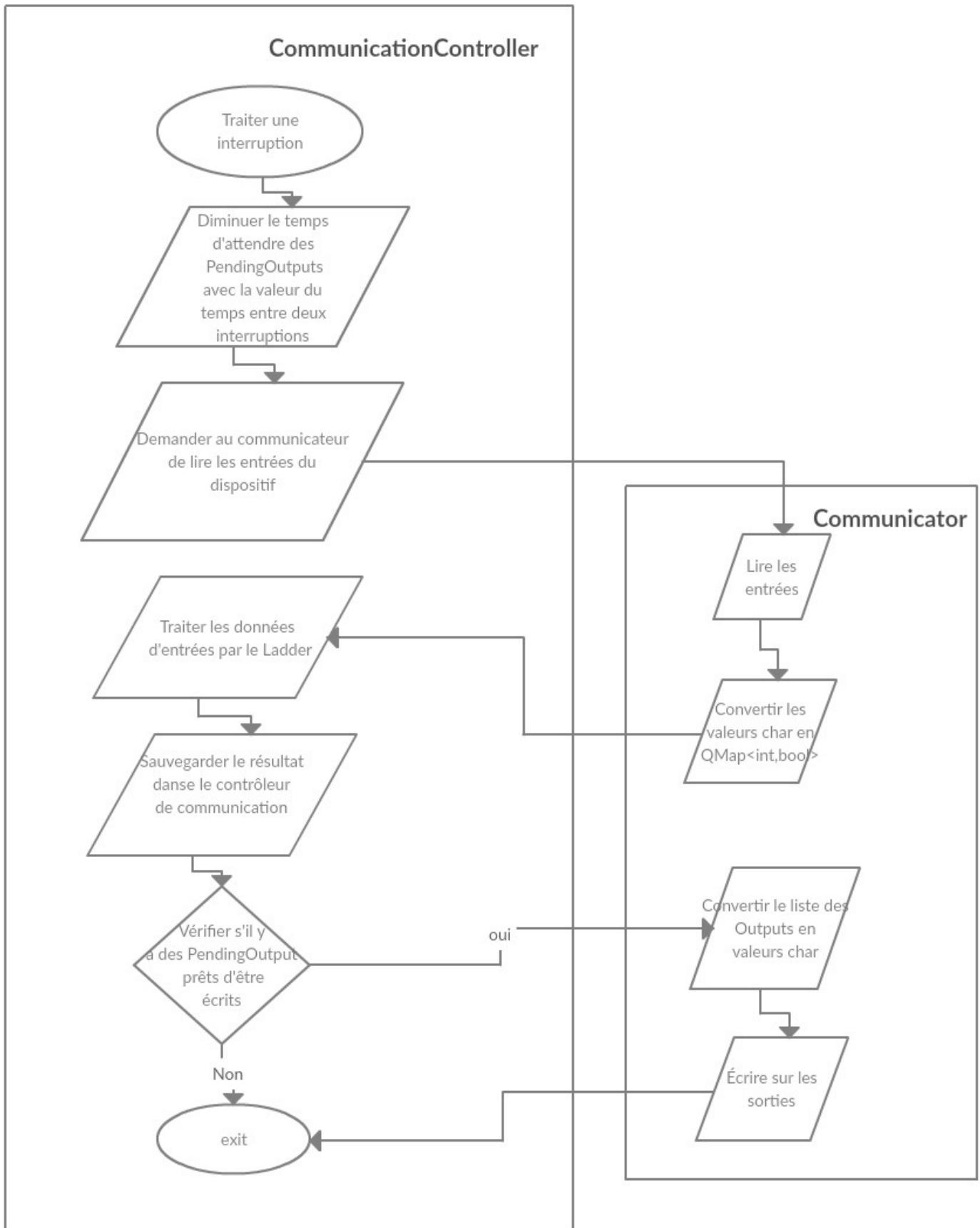


Figure 10.1.6 Schéma de communication avec les dispositifs

```

62 ▼ void CommunicationController::onInterrupt()
63 {
64     reduceTimePendingOutputs();
65     readInputs();
66 }

```

Figure 10.1.7 Photo instantanée du slot privé onInterrupt dans le CommunicationController

Chaque 50 millisecondes on commence une session de communication dans le slot onInterrupt() - Fig. 10.1.7. Premièrement on réduit le temps des m_pendingOutputs et puis on fait une demande au communicateur de lire les données d'entrées.

```

118 ▼ void CommunicationController::reduceTimePendingOutputs()
119 {
120     QMutableListIterator<PendingOutput> iter(m_pendingOutputs);
121     while (iter.hasNext())
122     {
123         PendingOutput& value = iter.next();
124         value.reduceTimeToWait(s_interruptInterval);
125     }
126 }

```

Figure 10.1.8 Photo instantanée de la fonction privée reduceTimePendingOutputs() dans le CommunicationController

Dans la fonction reduceTimePendingOutputs() on traverse la liste avec des données attendant d'être écrites sur les sorties du dispositif – Fig. 10.1.8. Chaque objet du type PendingOutput a une variable qui signifie le temps d'attendre et au début de chaque interruption nouvelle on soustrait 50 millisecondes de ce temps, parce que l'objet a déjà attendu les 50 millisecondes, qui sont passé depuis l'interruption précédente.

```

45
46 ▼ void CommunicationController::readInputs()
47 {
48     m_baseCommunicator->readInputs();
49 }

```

Figure 10.1.9 Photo instantanée de la fonction privée readInputs() dans le CommunicationController

Dans la fonction readInputs() on fait appel au communicateur actuel de lire les entrées du dispositif – Fig. 10.1.9 . On peut voir que le résultat retourné par cette fonction n'est pas traité par la logique du contrôleur, parce que on utilise une structure asynchrone pour obtenir les résultats de la lecture. Alors ils seront obtenues dans le slot onInputsRead() connecté au signal du communicateur inputsRead().

```

82 ▼ void CommunicationController::onInputsRead(QMap<int, bool> inputs)
83 {
84     QList<PendingOutput> outputs = ladder().treatInputs(inputs);
85     m_pendingOutputs.append(outputs);
86     sendPendingOutputs();
87 }
88

```

Figure 10.1.10 Photo instantanée du slot public onInputsRead() dans le CommunicationController

Dans la fonction onInputsRead() - Fig. 10.1.10, on passe les données d'entrées lues comme argument au objet ladder et obtient les objets de type PendingOutputs comme résultat. On ajoute ses objets à la liste des PendingOutputs, qui est membre de la classe CommunicationController – m_pendingOutputs. Puis on fait une liste des objets PendingOutputs qui ne doivent attendre plus de temps et on les passe au communicateur pour qu'il les écrit sur les sorties du dispositif.

```

97 ▼ void CommunicationController::sendPendingOutputs()
98 {
99     QList<Output> outputsToSend;
100    QMutableListIterator<PendingOutput> iter(m_pendingOutputs);
101    while (iter.hasNext())
102    {
103        PendingOutput& value = iter.next();
104
105        if (!value.isPending())
106        {
107            outputsToSend.append(value);
108            iter.remove();
109        }
110    }
111
112    m_baseCommunicator->writeOutputs(outputsToSend);
113}
114

```

Figure 10.1.11 Photo instantanée de la fonction privée sendPendingOutputs() dans le CommunicationController

Dans la classe BaseCommunicator on a trois fonctions statiques auxiliaires, qui servent à ménager les opérations binaires logiques avec les données reçues via les ports parallèles et séries.

Alors premièrement on reçoit une variable de type unsigned char (8 bits). Chaque bit peut représenter une entrée ou une sortie du dispositif. Selon la configuration on s'intéresse seulement à certaines bits. Alors pour chacun bit important on demande s'il est 0 ou 1 en passant à la fonction statique isBitSet() le char contenant les données et le nombre de bit en question – Fig. 10.1.12. On utilise l'opération logique de multiplication binaire. Pour chaque nombre de bite (0,1,2..7) on multiplie le char avec un nombre hexadécimal, dont le bit en question et le seul bit avec une valeur égale à 1, tous les autres sont égales à zéro. Dans cette façon la résultat de la multiplication sera différente de zéro seulement si le bit en question des données lues et en haut – 1.

```

10 ▼ bool BaseCommunicator::isBitSet(unsigned char x, int bitNumber)
11 {
12     switch (bitNumber) {
13     case 0:
14         return (x & 0x01) != 0;
15     case 1:
16         return (x & 0x02) != 0;
17     case 2:
18         return (x & 0x04) != 0;
19     case 3:
20         return (x & 0x08) != 0;
21     case 4:
22         return (x & 0x10) != 0;
23     case 5:
24         return (x & 0x20) != 0;
25     case 6:
26         return (x & 0x40) != 0;
27     case 7:
28         return (x & 0x80) != 0;
29     default:
30         return false;
31     }
32 }
33

```

Figure 10.1.12 Photo instantanée de la fonction statique isBitSet dans le BaseCommunicator

Les deux autres fonctions auxiliaires servent à maîtres des bites concrètes en bas ou en haut – setBitLow() et setBitHigh(). Pour maître un bit en bas on utilise la multiplication logique binaires en multipliant la valeur des données (unsigned char x) par de nombres hexadécimal, dont tous les bites sont en haut en dehors du bit, qui on veut maîtres en bas. Par contre quand on veut maître en haut un bit on exécute une addition logiques binaires en multipliant la variable x avec un nombre hexadécimal, dont tous les bits sont en bas que le bit qui on veut maître en haut.

```

66 ▼ void BaseCommunicator::setBitLow(unsigned char& x, int bitNumber)
67 {
68     switch (bitNumber) {
69     case 0:
70         x&=0xFE;
71         break;
72     case 1:
73         x&=0xFD;
74         break;
75     case 2:
76         x&=0xFB;
77         break;
78     case 3:
79         x&=0xF7;
80         break;
81     case 4:
82         x&=0xEF;
83         break;
84     case 5:
85         x&=0xDF;
86         break;
87     case 6:
88         x&=0xBF;
89         break;
90     case 7:
91         x&=0x7F;
92         break;
93     default:
94         break;
95     }
96 }
34 ▼ void BaseCommunicator::setBitHigh(unsigned char& x, int bitNumber)
35 {
36     switch (bitNumber) {
37     case 0:
38         x|=0x01;
39         break;
40     case 1:
41         x|=0x02;
42         break;
43     case 2:
44         x|=0x04;
45         break;
46     case 3:
47         x|=0x08;
48         break;
49     case 4:
50         x|=0x10;
51         break;
52     case 5:
53         x|=0x20;
54         break;
55     case 6:
56         x|=0x40;
57         break;
58     case 7:
59         x|=0x80;
60         break;
61     default:
62         break;
63     }
64 }

```

Figure 10.1.13 Photo instantanée des fonctions statiques setBitHigh et setBitLow dans le BaseCommunicator

10.2. Communication en parallèle

Il y a quatre fonction virtuelles pures dans la classe BaseCommunicator, qui chaque communicateur héritant la classe de base doit implémenter à sa part. Pour le communicateur parallèle seulement trois de ses fonctions ont une importance (la quatrième stopCommunication() est implémentée aussi, mais elle est vide).

Pour l'implémentation de la communication par le port parallèle on utilise la classe io.h de la module sys → <sys/io.h>, qui fait partie de la librairie libre GNU C (GNU C Library glibc) – logiciel libre.

```

17
18 ▼ void ParallelCommunicator::initiateCommunication()
19 {
20     if (iopl(3)) //reserve 0x378, 0x379
21         fprintf(stderr, "Couldn't get the port at %x\n", base), exit(1);
22 }

```

Figure 10.2.1 Photo instantanée de la fonction initiateCommunication dans le ParallelCommunicator

Dans la fonction initiateCommunication() on demande accès sur le port parallèle – port 379, pour lire et port 378 pour écrire sur le dispositif. En effet utilisant la fonction iopl() avec un argument 3 on resserve tous les ports d'entrée-sortie – Fig.10.2.1 .

Utilisant les fonction inb() et passant le numéro du port comme argument on lit les entrées et retourne les données lues comme un char – Fig. 10.2.2 . Puis en utilisant la fonction outb avec des arguments – port et char à écrire.

```

29 ▼ QMap<int, bool> ParallelCommunicator::readInputs()
30 {
31     char b    = inb(base+1);
32     QMap<int, bool> result = charToInputs(b);
33     emit inputsRead(result);
34     return result;
35 }

```

Figure 10.2.2 Photo instantanée de la fonction readInputs dans le ParallelCommunicator

```

37 void ParallelCommunicator::writeOutputs(QList<Output> outputs)
38 {
39     unsigned char x = outputsToChar(outputs);
40     outb(x,base);
41 }
42

```

Figure 10.2.3 Photo instantanée de la fonction writeOutputs dans le PrarallelCommunicator

Schéma du dispositif – Fig. 10.2.4

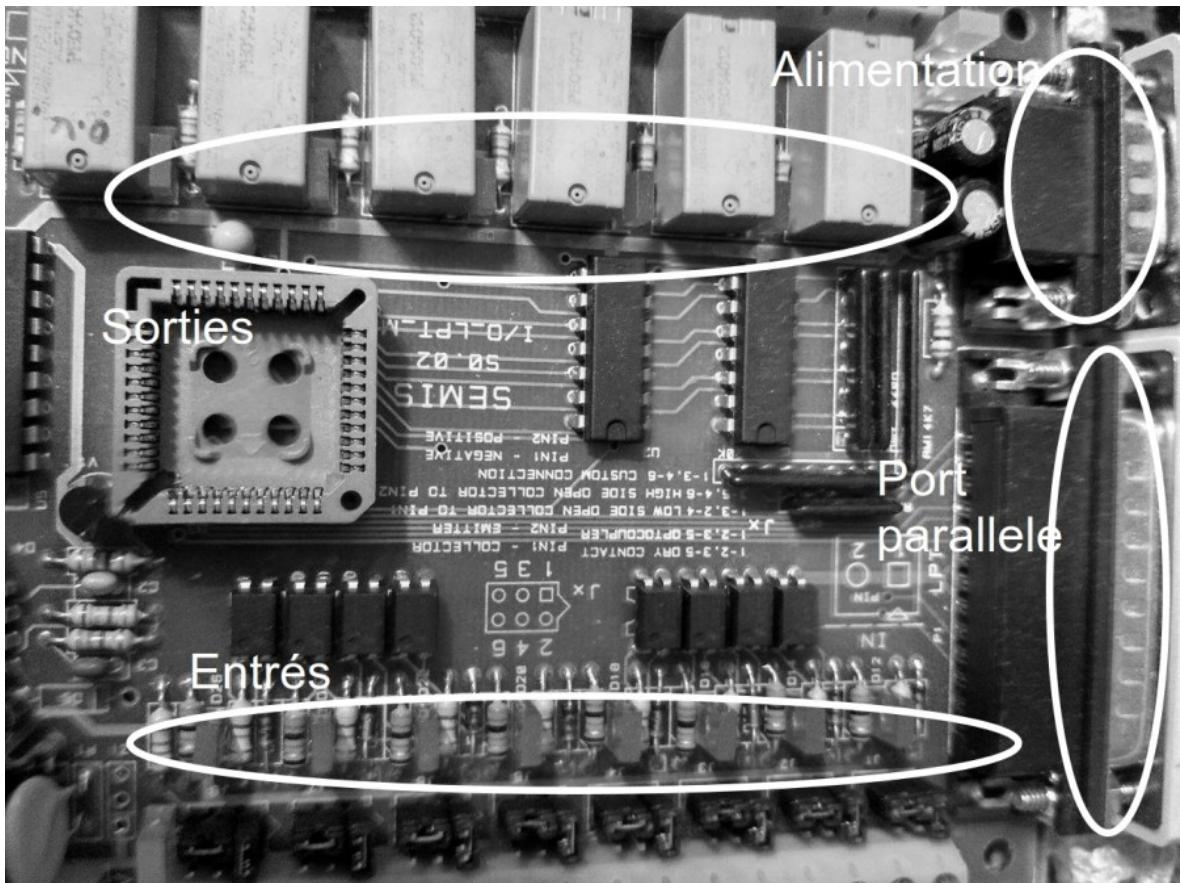


Figure 10.2.4 Photo du dispositif

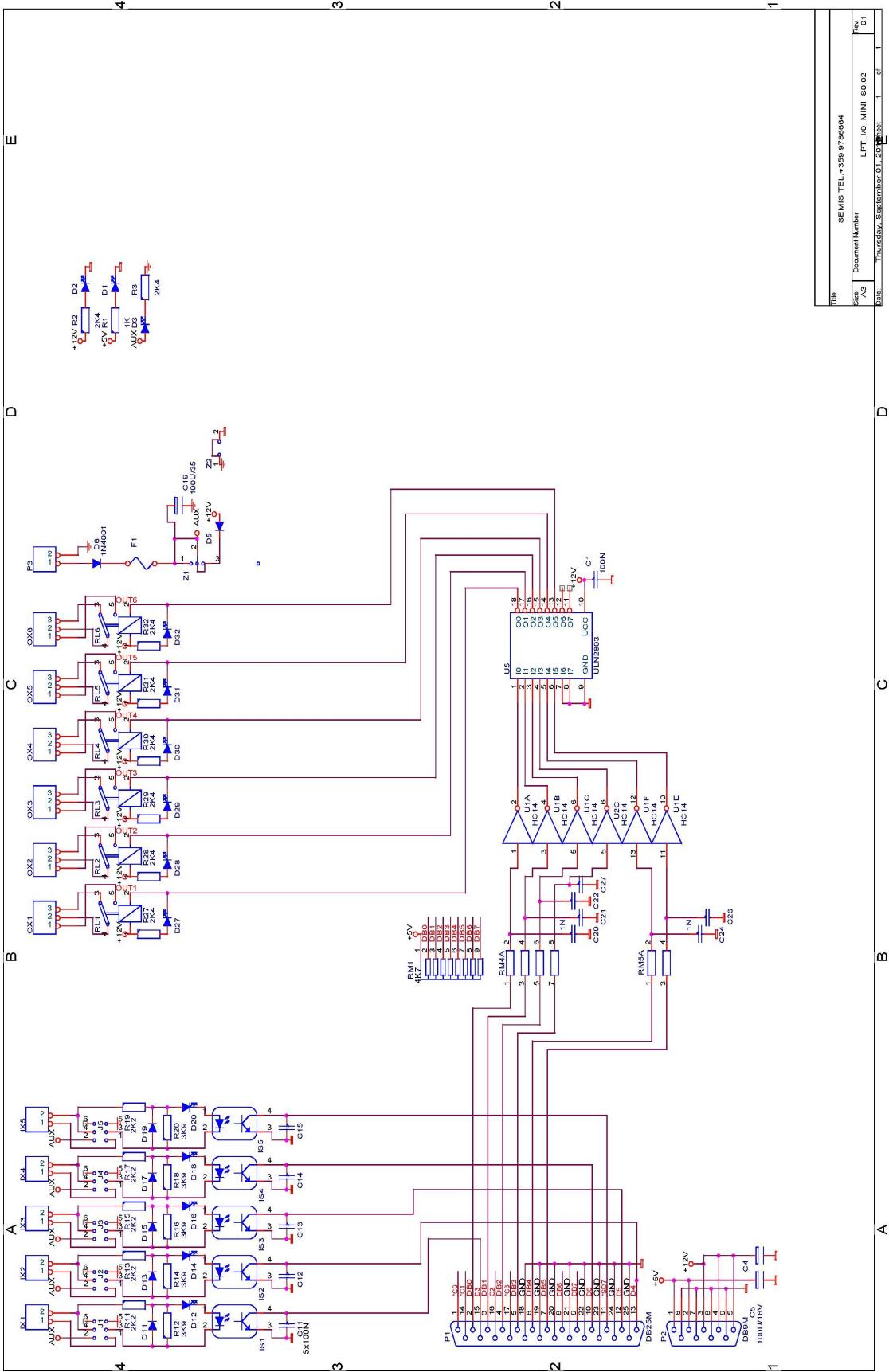


Figure 10.2.5 Schéma électrique du dispositif

10.3. Communication en série

Dispositif

Le dispositif en série comporte trois parties essentielles – des entrées, des sorties et un microcontrôleur PIC – Fig.10.3.1 . Le PC communique avec le PIC en utilisant un protocole définir par le développeur, qui écrit le logiciel sur le PIC.

Un microcontrôleur PIC est une unité de traitement et d'exécution de l'information à laquelle on a ajouté des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants annexes. Un microcontrôleur PIC peut donc fonctionner de façon autonome après programmation.

Les PIC intègrent une mémoire programme non volatile (FLASH), une mémoire de données volatile (SRAM), une mémoire de donnée non volatile (E2PROM), des ports d'entrée-sortie (numériques, analogiques, MLI, UART, bus I2C, Timers, SPI, etc.), et même une horloge, bien que des bases de temps externes puissent être employées.

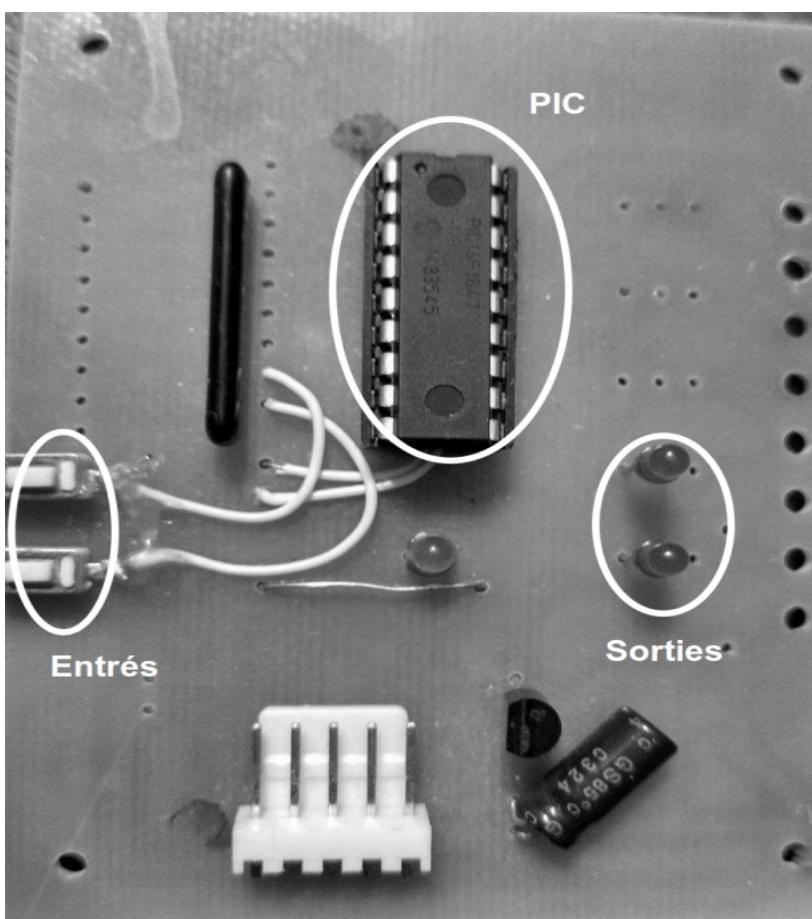


Figure 10.3.1 Photo du dispositif série

Les PIC se conforment à l'architecture Harvard : ils possèdent un mémoire de programme et un mémoire de données séparées. La plupart des instructions occupent un mot de la mémoire de programme. La taille de ces mots dépend du modèle de PIC, tandis que la mémoire de données est organisée en octets.

Les PIC sont des processeurs dits RISC, c'est-à-dire processeur à jeu d'instruction réduit. Plus on

réduit le nombre d'instructions, plus facile et plus rapide en est le décodage, et plus vite le composant fonctionne. Cependant, il faut plus d'instructions pour réaliser une opération complexe.

La communication PC - microcontrôleur est de type master – esclave. Le protocole de communication avec le PIC prend les formes suivantes:

- Lire les entrées

PC → PIC

adresse du Pic * commandement lire * vérification
1*3*255

PIC → PC

adresse du Pic * commandement lire * données * données * vérification
1*3*0*0*255

- Écrire sur les sorties

PC → PIC

adresse du Pic * commandement écrire * données * données **vérification
1*6*164*164*255

PIC → PC

adresse du Pic * commandement écrire * données * données * vérification
1*6*164*164*255

La vérification as toujours la valeur 255. Le commandement lire a la valeur – 3, et le commandement écrire – 6. Les données sont répéter pour pouvoir vérifier l'apparition d'une erreur pendant la transmission.

Pour la réalisation de la communication par le port série on utilise la classe de Qt `QtSerialPort`. Dans le constructeur de la classe `SerialCommunicator` on connecte le signal `readyRead()` de la classe `QtSerialPort` au slot du communicateur `readData()`. La communication via le `QtSerialPort` est asynchrone et c'est pourquoi on utilise le système des signaux et slots.

Puis dans la fonction d'initialisation de la communication on configure les paramètres de `QtSerialPort`, dont on a besoin pour ouvrir le canal de communication avec le dispositif – parité, nom de port, baud rate, stop bit, data bits – Fig. 10.3.2.

```
25
26 void SerialCommunicator::initiateCommunication()
27 {
28     m_serialPort.setPortName(currentSettings.name);
29     m_serialPort.setBaudRate(currentSettings.baudRate);
30     m_serialPort.setDataBits(currentSettings.dataBits);
31     m_serialPort.setParity(currentSettings.parity);
32     m_serialPort.setStopBits(currentSettings.stopBits);
33     m_serialPort.setFlowControl(currentSettings.flowControl);
34     if (!m_serialPort.open(QIODevice::ReadWrite))
35     {
36         qDebug() << "Could not open port";
37     }
38 }
```

Figure 10.3.2 Fonction `initiateCommunication` dans la classe `SerialCommunicator`

La structure currentSettings est une structure, qui on utilise pour garder l'information sur le port. On initialise cette structure aussi dans le constructeur de la classe SerialCommunicator. Les paramètres utilisés par le prototypes sont baude rate – 9600, aucune parité, data bits - 8, stop bit – 1. Tous ça et bien importante pour pouvoir se comprendre avec le dispositif série.

```

39
40 ▼ QMap<int, bool> SerialCommunicator::readInputs()
41 {
42     QByteArray readCommand;
43     readCommand.append(s_address);
44     readCommand.append(s_commandRead);
45     readCommand.append(s_check);
46
47     readCommand.resize(s_commandReadLength);
48
49     m_isReadingSession = true;
50     m_serialPort.write(readCommand);
51
52     return QMap<int, bool>();
53 }
54

```

Figure 10.3.3 Fonction readInputs dans la classe SerialCommunicator

Les deux fonctions – écrire et lire, exigent que le logiciel premièrement écrit une commande (commande lire ou commande écrire) sur le dispositif – Fig. 10.3.3. et Fig. 10.3.4. Pour construire les commandes on utilise la classe QByteArray, qui fourni une tableau des octets. On ajoute consécutivement les chars représentants l'adresse, le type de commande, les données (seulement dans le cas d'écriture) et la vérification. On sauvegarde dans une variable booléenne si la session actuelle est une session d'écriture ou lecture. Et si c'est une section de lecture dans le slot readData() on émit comme un signal les données lues.

```

55 ▼ void SerialCommunicator::writeOutputs(QList<Output> outputs)
56 {
57     m_dataToWrite = outputsToChar(outputs);
58     writeData();
59 }

```

Figure 10.3.4 Fonction writeOutputs dans la classe SerialCommunicator

```

69 ▼ void SerialCommunicator::writeData()
70 {
71     QByteArray writeCommand;
72     writeCommand.append(s_address);
73     writeCommand.append(s_commandWrtie);
74     writeCommand.append(m_dataToWrite);
75     writeCommand.append(m_dataToWrite);
76     writeCommand.append(s_check);
77
78     writeCommand.resize(s_commandWrtieLength);
79
80     m_isWrittingSession = true;
81     m_serialPort.write(writeCommand);
82 }
83

```

Figure 10.3.4 Fonction dans la classe SerialCommunicator

11.Mode d'emploi

1. Charger un fichier de logique ladder et un fichier de configuration – Fig. 11.1

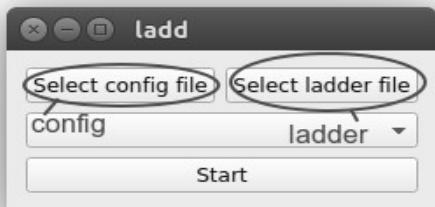


Figure 11.1 Interface utilisateur

2. Choisir un mode de communication – parallèle ou série – Fig. 11.2

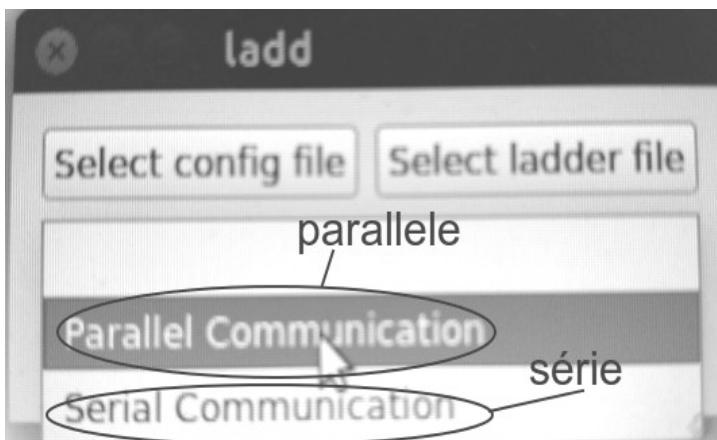


Figure 11.2 Interface utilisateur

3. Démarrer la communication – Fig. 11.3

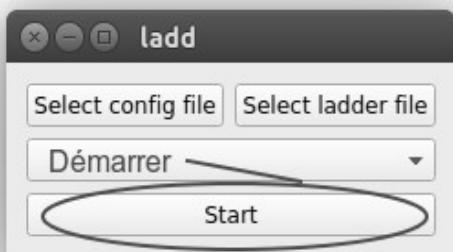


Figure 11.3 Interface utilisateur

Exemple de fonctionnement

Configuration parallèle

```
<parport>
    <inputs>
        <input>
            <id>10.1</id>
            <bitnumb>3</bitnumb>
            <!--LSB!-->
            <initialValue>1</initialValue>
        </input>
        <input>
            <id>10.2</id>
            <bitnumb>4</bitnumb>
            <!--LSB!-->
            <initialValue>1</initialValue>
        </input>
        <input>
            <id>10.3</id>
            <bitnumb>5</bitnumb>
            <!--LSB!-->
            <initialValue>1</initialValue>
        </input>
        <input>
            <id>10.4</id>
            <bitnumb>6</bitnumb>
            <!--LSB!-->
            <initialValue>1</initialValue>
        </input>
    </inputs>
    <outputs>
        <output>
            <id>70.1</id>
            <bitnumb>0</bitnumb>
            <!--LSB!-->
        </output>
        <output>
            <id>70.2</id>
            <bitnumb>1</bitnumb>
            <!--LSB!-->
        </output>
    </outputs>
</parport>
```

Fichier ladder

RD 10.0
RD.STK 10.1
AND 10.2
OR.STK
WRT 70.1

RD 10.2
RD.NOT.STK 10.3
OR 10.4

AND.STK
WRT 70.2
RD 10.0
RD.STK 10.1
AND 10.2
OR.STK
WRT 70.1

10.0 – active
→ 70.1 – active – Fig. 11.4

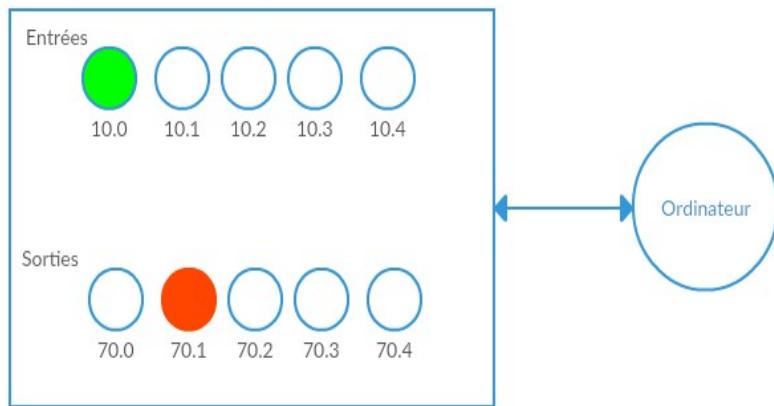


Figure 11.4 Exemple de fonctionnement

10.1 – active AND 10.2 – active
→ 70.1 – active – Fig. 11.5

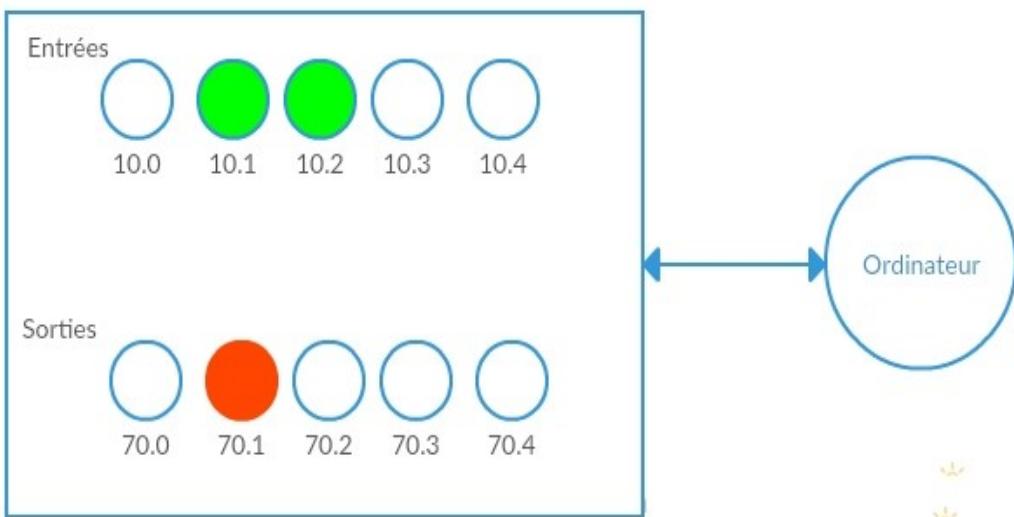


Figure 11.5 Exemple de fonctionnement

RD 10.2

RD.NOT.STK 10.3

OR 10.4

AND.STK

WRT 70.2

10.2 – active AND (!10.3) – active

→ 70.2 – active – Fig. 11.6

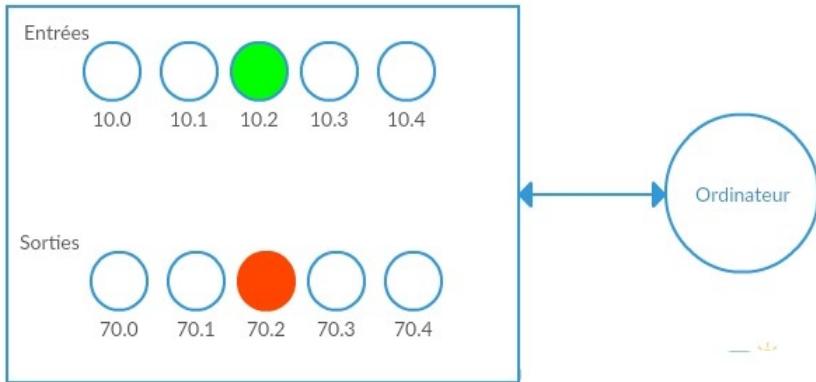


Figure 11.6 Exemple de fonctionnement

10.2 – active AND (!10.3) – pas active

→ 70.2 – pas active – Fig. 11.7

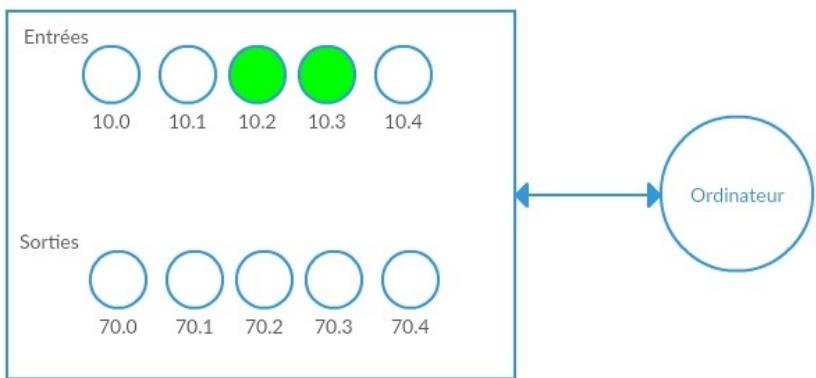


Figure 11.7 Exemple de fonctionnement

10.2 – active AND (!10.3) – pas active OR 10.4 - active)

→ 70.2 – active – Fig. 11.8

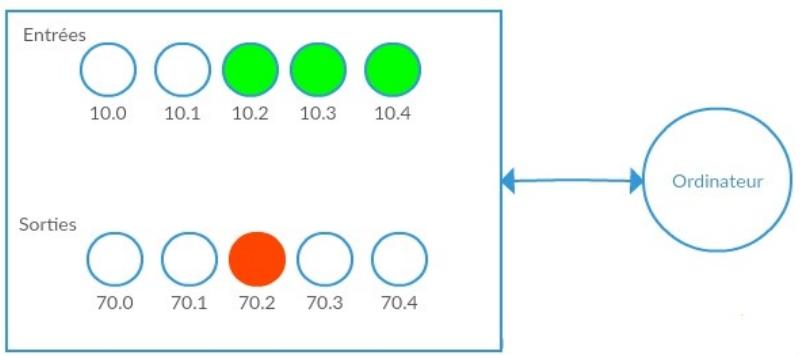


Figure 11.8 Exemple de fonctionnement

Conclusion

Dans ce projet de fin d'études on a fait une recherche sur :

- des systèmes temps réel et ses applications dans la pratique
- des systèmes d'exploitation temps réel
- des moyennes de transformer Linux dans un système avec des caractéristiques d'un système d'exploitation temps réel

Puis on a réalisé un prototypes du logiciel qui :

- fait la déserialisation des fichiers *.xml et *.LAD
- communique avec un dispositif via l'interface parallèle et l'interface série
- traite les données d'entrées selon la logique ladder décrite dans le fichier *.LAD

Les fonctionnalités réalisées peuvent être seulement le début d'un projet plus grand. Dans la future on envisage les pas de développement suivants :

- réalisation de la communication simultanée avec un dispositif parallèle et un dispositif série
- élargissement de la capacité du logiciel – maintenant on peut traiter seulement huit (un octet) bits d'entrées et huit bits de sortie. Ce nombre doit être beaucoup plus grand (32, 64 etc.)
- réalisation de la communication par l'interface USB et Ethernet

Le projet futur peut être utilisé dans l'industrie en contrôlant des processus temps réel automatiques et en même temps offrant tous les avantages du système d'exploitation Linux.

Bibliographie

1. S.M. Kuo, B.H. Lee, and W. Tian, "Real-Time Digital Signal Processing: Implementations and Applications",
2. Shin, K.G.; Ramanathan, P. (Jan 1994). "Real-time computing: a new discipline of computer science and engineering"
3. C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. Journal of the ACM
4. Windows and Real-Time, Daniel Terhell
5. Tanenbaum, Andrew (2008). Modern Operating Systems. Upper Saddle River
6. "Context switching time". Segger Microcontroller Systems.
7. "RTOS performance comparison on emb4fun.de".
8. Yodaiken, Victor (1999). "The RTLinux Manifesto". Published in the 5th Linux Conference Proceedings
9. Barabanov, Michael (1996). "A Linux Based Real-Time Operating System"
10. Yodaiken, Victor (1996). "Cheap Operating systems Research" Published in the Proceedings of the First Conference on Freely Redistributable Systems, Cambridge MA, 1996
11. Dougan, Cort (2004), "Precision and predictability for Linux and RTLinuxPro", Dr. Dobbs Journal, February 1, 2004
12. Comparison of real-time operating systems at DMOZ
13. https://en.wikipedia.org/wiki/Real-time_computing#Criteria_for_real-time_computing
14. <http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/ACCOV/TRPP1Intro.pdf>
15. <https://www.micrium.com/hardware-accelerated-rtos-%C2%B5cos-iii-hw-rtos-and-the-r-in32m3/>
16. <http://www.memoireonline.com/04/10/3271/Le-multitche-avec-pic18f452-a-laide-dun-noyau-temps-reel.html>
17. https://hegdesuhas.wordpress.com/2011/04/13/installing-the-real-time-linux-rt_preempt-ubuntu/
18. http://www.site.uottawa.ca/~nrahmani/CEG4566_H13/notes_cours/Chap_I_H13.pdf
19. <http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-152/Virtualisation-et-systemes-embarques-l-exemple-ADEOS>
20. http://pficheux.free.fr/articles/lmf/hs24 realtime/linux_realtime_reloaded_final_www.pdf
21. <http://syncor.blogspot.bg/2011/07/signal-processing-on-windows-in-real.html>
22. <https://www.osr.com/nt-insider/2014-issue3/windows-real-time/>
23. <http://askubuntu.com/questions/4762/how-to-set-up-video-and-audio-players-to-use-realtime-or-close-priority>
24. <http://www-igm.univ-mly.fr/~dr/XPOSE2014/JavaTempsReel/plateformes.html>
25. <https://sourceware.org/gdb/onlinedocs/gdb/Server.html>
26. <http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/>
27. <http://www.thegeekstuff.com/2014/04/gdbserver-example/>
28. <https://www.gnu.org/software/gdb/>
29. <http://doc.qt.io/qtcreator/creator-debugger-operating-modes.html>