

# Quelques pensées sur les conventions d'appel

*Qt by Nokia*

par Thiago Macieira traducteur : Thibaut Cuvelier ([Site web](#)) ([Blog](#)) Qt Labs

Date de publication : 16/08/2009

Dernière mise à jour : 03/10/2010

**Après avoir dégrossi les ABI**, nous allons cette fois nous attaquer aux conventions d'appel, une partie des ABI, mais néanmoins très importante.

Cet article est une traduction autorisée de  **Some thoughts on calling convention**, par Thiago Macieira.

N'hésitez pas à commenter cet article !

I - L'article original.....	3
II - Introduction.....	3
III - De quoi parlons-nous ?.....	3
IV - Comment sont effectués les appels de fonction.....	4
V - Aujourd'hui.....	5
VI - Autres architectures.....	6
VI-A - SPARC.....	6
VI-B - ARM.....	6
VI-C - Itanium (IA-64).....	7
VII - Conclusion.....	8
VIII - Divers.....	8

## I - L'article original

Les *Qt Labs Blogs* sont des blogs tenus par les développeurs de Qt, concernant les nouveautés ou les utilisations un peu extrêmes du framework.

Nokia, Qt et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction du billet  **Some thoughts on calling convention**, par Thiago Macieira.

## II - Introduction

Eh oui, c'est encore moi. Et ce n'est pas encore la série que je vous avais promise. Non, j'ai décidé d'étendre un peu la situation de la compatibilité binaire et des ABI, en explorant un concept encore plus détaillé.

Ceci est encore un pur produit de cerveau, n'en attendez pas de conclusion. Ce texte pourrait recevoir des appréciations de "truc sympa, tout ce que voulais savoir sans jamais oser le demander", à "quelque chose d'amusant", à "Tu m'as perdu", à "C'est encore moi". Vous aurez été prévenus !

Cependant, avant de rentrer dans le vif du sujet, laissez-moi m'exprimer à propos de mon post précédent. Il y manquait une conclusion. Il y a une raison à cela : le texte n'est pas un essai. Mais, au delà, j'avais écrit un gros morceau du texte que je devais encore éditer avant de publier. Je pensais pouvoir le faire avant d'aller au lit. Deux heures plus tard. Je n'avais pas encore fini. Je n'avais pas encore de conclusion. Il était plus que l'heure de dormir. Ainsi j'ai publié.

Si vous voulez une conclusion, la voici : la compatibilité binaire est une chose assez dure. Nous le faisons pour vous, vous ne devez donc pas l'apprendre. À moins que vous écriviez, vous aussi, une librairie...

Je veux dire que la compatibilité binaire est un jeu d'essais-erreurs. C'est un art que, patiemment, nous avons amélioré, au fil des années, maintenant nombreuses. Qt s'est débrouillé pour rester une librairie C++ binaires compatible pour quelques années, tout en restant en développement constant, subissant extensions et améliorations. L'utilisation généralisée des *d-pointers* y est pour beaucoup. Nous avons aussi des règles à suivre, concernant **ce que nous pouvons faire et ce que nous ne pouvons pas faire**, ainsi que des **tests automatiques** (merci à Harald) pour repérer de tels problèmes. J'ai même, assez récemment, utilisé **un outil de vérification de compatibilité binaire pour Linux** sur Qt.

Comme je l'ai dit, ce processus n'est qu'une suite d'essais et d'erreurs. Nous réglons les problèmes quand nous les voyons. Par exemple, dans Qt 4.4.0, par accident, nous avons cassé la compatibilité de QByteArray avec les compilateurs Microsoft. Nous avons retiré un const du type de retour de char QByteArray::at(int) const. Nous l'avons corrigé pour la 4.4.1, ce qui a restauré la compatibilité. En fait, nous avions le patch pour la 4.4.0, mais, à la suite d'une faute de frappe, il n'a pas été appliqué dans les paquets proposés au téléchargement. Pire encore, j'avais reconstruit ces paquets, et oublié de les mettre sur le FTP... Dans Qt 4.5.0, nous avons introduit une autre cassure accidentelle, mais, cette fois, nous avons décidé de "faire avec". C'est **ce problème** qui m'a demandé d'ajouter une nouvelle règle à la page sur la Techbase, d'écrire les exemples, de lire quelques nouvelles informations, et, au final, d'écrire ces quelques posts.

Maintenant, finies les digressions, retournons aux conventions d'appel.

## III - De quoi parlons-nous ?

Comme je l'ai fait remarquer plus tôt, j'ai écrit un assez long texte pour l'autre article. Il contenait des parties du texte que je place ici. Et une autre raison de le faire maintenant est que j'ai dû, en plein mois de juillet, essayer de faire compiler QtWebKit sur Solaris / UltraSPARC / Sun Studio (CC 5.9), ainsi que sur AIX / POWER6 / xLC 7.0. Longue

histoire courte : il compile sur les deux, avec une grande partie des patches. Il ne lie pas sur AIX. Mais il fonctionne bien sur Solaris, à l'exception d'un crash, que j'ai essayé de déboguer.

Pendant ce débogage, j'ai dû apprendre un peu de l'assembleur SPARC et sa convention d'appel.

Donc : qu'est-ce que la convention d'appel ? Quand vous appelez une fonction, dans n'importe quel langage, les paramètres que vous lui passez doivent être placés quelque part où l'appelé peut les trouver, dans le bon ordre. Aussi, l'appelé peut retourner des informations, ce qui est aussi défini. De plus, il y a des ressources sur le CPU auxquelles l'appelé ne peut toucher, et d'autres qu'il peut modifier à son gré.

La convention d'appel est juste une autre partie de l'ABI. C'est la plus importante partie en C, à côté des tailles et alignements des types. En C++, comme mon précédent article l'a montré, les ABI doivent en faire beaucoup plus.


## IV - Comment sont effectués les appels de fonction

Commençons, une fois encore, avec un leçon d'histoire. Dans les premiers jours de l'informatique, la **mémoire** primait, et la technologie des transistors était toujours chère. Ainsi, les processeurs disposaient de peu de registres, de taille limitée, et très peu de mémoire. L'**Intel 4004**, par exemple, avait 16 registres de quatre bits, trois niveaux de pile, et pouvait adresser 4096 mots de quatre bits (soit une mémoire de deux kilooctets au plus).

Le temps avançant, les choses se sont améliorées. Nous avons plus de RAM, des registres plus grands, et la limite de pile levée. Ce dernier changement était assez important, nous verrons pourquoi un peu plus tard.

À ce moment-là, nous avons eu droit à un processeur 16 bits, l'**Intel 8086**, les bases de la convention d'appel que nous utilisons aujourd'hui ont été imposées. Cette plateforme avait huit registres généraux de seize bits, même si un avait son utilisation propre : le pointeur de pile (en anglais, *stack pointer*, SP). Les autres registres avaient de drôles de limites dans la manière dont on pouvait en user : certaines instructions ne pouvaient fonctionner que dans un registre précis.

Comme son nom l'indique, le pointeur de pile est un pointeur sur une pile. Ce qui signifie que nous avons une implémentation LIFO (*Last-In First-Out* ; premier entré, dernier sorti), en ayant un pointeur sur la dernière entrée. Quand on met quelque chose sur la pile, on déplace le pointeur vers la saisie suivante disponible.

 *De manière intéressante, la pile est implémentée dans toutes les architectures de ma connaissance avec une stratégie de grow downwards : mettre quelque chose sur la pile entraîne une décrémentation du pointeur de pile.*

Le pointeur de pile était utilisé pour supprimer la limitation du nombre d'appel effectuels. L'instruction CALL, en plus de transférer le contrôle à l'adresse spécifiée, met l'adresse de retour (l'instruction à exécuter après l'appel) sur la pile. De la même manière, l'instruction RET lit une adresse de la pile et saute dessus.


Maintenant, la pile est utilisée pour de nombreuses choses, notamment pour les variables automatiques dans une fonction (celles avec le mot-clé auto, depuis longtemps tombé dans l'oubli). Si vous avez une fonction récursive (qui s'appelle elle-même), il est facile de voir que l'ABI doit trouver une manière de placer ces variables dans une place différente de l'instance extérieure.

Dans les archaïques jours du DOS, comme je l'ai dit précédemment, il n'y avait que de l'édition statique des liens. L'entiereté de ces bibliothèques étaient fournies par le compilateur, ainsi il pouvait choisir la meilleure manière d'allouer les ressources à la main (registres et pile) pour passer les paramètres. Mais, même avec cette flexibilité, le compilateur devait être *déterministe* : lors d'un appel de fonction, il devait savoir ce qu'il décidait dans la fonction appelée concernant l'endroit réservé aux paramètres.

La solution la plus simple était, très basiquement, pour chaque compilateur sur x86, de mettre les valeurs sur la pile. Ainsi, il n'y a plus d'ambiguïté. Enfin... C'est ce que vous pensez. Premièrement, doit-on les mettre de gauche à droite ou de droite à gauche ? Et qui s'occupera du nettoyage : l'appelant ou l'appelé ? Le besoin, en C, d'appels

de fonction à longueur variable dictait, et on les mettait de droite à gauche, et l'appelant nettoyait. Mais d'autres langages, comme le Pascal, ont choisi l'exact opposé. À cette époque, on pouvait choisir sa convention d'appel : `__cdecl` (*C declaration*), `__pascal` ou `__fortran`.

Mais, même dans ces jours où les tables de l'assembleur venaient avec le nombre de cycles que chaque instruction prenait, et où vous pouviez compter dessus, les accès à la mémoire étaient très chers. Pour lire ou écrire dans la mémoire, on devait ajouter un cycle au temps d'exécution. L'étape suivante, dans les conventions d'appel, était d'utiliser des registres pour passer des données, puisque leur temps d'accès était nul. Ceci n'a jamais réellement été standardisé, entraînant la création de myriades de conventions d'appel, toutes différentes, qui avaient des noms comme `fastcall`, `stdcall`, `syscall`, et différaient entre les compilateurs. La raison de tout ceci était claire : il n'y avait pas assez de registres.

 *La distinction entre pointeurs lointains (*far*) et proches (*near*) existe pour une raison tout à fait différente. Pour accéder à l'ensemble de son mégaoctet de ROM et de RAM adressable, l'Intel 8086 avait une **mémoire segmentée**, avec quatre bits additionnels d'adressage en provenance des registres de segments, ce qui s'ajoutait aux seize bits des registres généraux. Les programmes écrits pour des systèmes 16 bits plus simples avaient des pointeurs de seize bits, mais le x86 avait besoin de plus de bits, une option était donc donnée au développeur pour choisir quelle taille de pointeur il désirait.*

## V - Aujourd'hui


Avec l'introduction du 80386, la plateforme x86 est passée du 16 au 32 bits. Peu a changé dans les registres, à l'exception des registres généraux, qui sont passés, eux aussi, à une largeur de trente-deux bits. Ainsi, la convention d'appel sur les x86 à 32 bits reste principalement la même, comme c'était dans les années 80 : les arguments sont passés sur la pile, mis de droite à gauche, et l'appelant se charge du nettoyage.

La seule chose supplémentaire était d'avoir quelques registres appelés *registres de travail*, et d'autres, *registres préservés*. Les registres de travail sont ceux que l'appelé peut utiliser à volonté, il ne doit pas préserver la moindre valeur. Si l'appelant n'a rien qu'il veuille préserver, il doit sauvegarder la valeur (d'où le fait que ces registres soient aussi appelés *registres caller-save*). Les registres préservés sont à l'opposé : l'appelé doit sauvegarder la valeur s'il a besoin du registre, et la restaurer par après (d'où leur autre nom, *registres callee-save*)

Ceci amène aussi des incompatibilités : sur certains systèmes d'exploitation, un jeu de registres donné est sauvegardé ; sur un autre, il s'agira d'un autre jeu.

Avec les architectures AMD64 et x86-64, le nombre de registres généraux a finalement été augmenté. Des huit originaux, ils sont passés à seize (appelés *r8*, *r9*... jusqu' *r15*). J'espère juste qu'ils auront renommés les huit les plus anciens *r0* à *r7*. Avec quelques registres supplémentaires, le système d'exploitation sur x86-64 a décidé de les rendre par défaut pour passer les arguments.

Mais, manifestement, ils ne se mettent pas d'accord sur lesquels utiliser. Au moins, ils ont essayé. [www.x86-64.org](http://www.x86-64.org) propose une spécification d'ABI, qui est utilisée par GCC (et, par conséquent, les systèmes d'exploitation où GCC est le compilateur par défaut). Cette ABI prévoit que le premier argument est passé dans le registre *RDI*, le second dans *RSI*, le troisième dans *RDX*, puis *RCX*, *R8*, *R9*, tant que les paramètres sont des entiers et tiennent dans les registres

 *S'ils avaient utilisé une numérotation numérique des registres, les registres *r7*, *r6*, *r2*, *r1*, *r8* et *r9* auraient été utilisés. La raison pour cet ordre bizarre est que le pointeur de pile est *r4*, tandis que *r3* et *r5* sont les registres préservés (*rbx* et *rbp*), et que les nouveaux registres requièrent plus d'octets dans l'instruction pour être accédés.*

Sous Windows, les registres sont utilisés différemment. Et il y a bien d'autres registres dans l'architecture, comme les registres des flottants, les registres SSE, et autres registres vectoriels.

## VI - Autres architectures

### VI-A - SPARC

Nous arrivons au point de la raison de parler des conventions d'appel. J'en viens à l'architecture SPARC. Le nom *SPARC* signifie *Scalable Processor ARChitecture*, et lui vient de sa pile de registres. L'architecture SPARC a trente-deux registres généraux, d'une largeur de soixante-quatre bits pour les processeurs UltraSPARC. Ces trente-deux registres sont divisés en quatre groupes : les registres globaux, les registres d'entrée, les registres locaux, les registres de sortie.

Les registres globaux sont une sorte normale de registres, en comparaison des autres architectures : leurs valeurs, une fois écrites, restent là. L'appel ou le retour de fonction ne change pas leurs valeurs.

Les trois autres groupes font partie d'un jeu de rotation. Le processeur, habituellement, possède plusieurs banques de huit registres, mais, tout le temps, seulement trois peuvent être accédés par le programme. Il y a deux instructions spéciales dans l'assembleur SPARC qui modifient ce jeu : *save* et *restore*. Lors de l'appel à *save*, le jeu tourne de deux : une des trois banques d'origine reste (à une autre position), et deux nouvelles apparaissent. Lors de l'appel à *restore*, le jeu tourne de deux dans le sens inverse : deux banques disparaissent, la troisième bouge, et les deux banques d'origine sont restaurées.

Ceci a pour conséquence que chaque fonction partage huit registres avec ses appelées, huit registres avec ses appelants, et huit registres non partagés. D'où la distribution en sortie, entrée et local.

Par conséquent, c'est très simple de décider qui, de l'appelant ou de l'appelé, doit sauvegarder : le jeu d'instruction en prend grand soin (vous devez toujours tirer au sort les registres globaux). Le pointeur de pile est choisi comme registre *%o6*. Par une astuce du moteur, le pointeur de pile est automatiquement sauvegardé à l'appel de fonction en tant que registre *%i6*, et automatiquement restauré à la fin. La sauvegarde est souvent appelée *pointeur de trame*.

Le problème avec le design de l'architecture SPARC (au delà du saut de slot - qui a pensé qu'exécuter une instruction *supplémentaire* après un appel ou un saut était une bonne idée ?), c'est qu'elle est peu économique. Chaque fonction a un jeu fixé de quinze registres de travail de soixante-quatre bits (registres locaux et d'entrée, sans le pointeur de trame vu son utilité spéciale). Beaucoup de fonctions n'ont pas besoin d'autant de registres, c'est pour ça qu'on leur en assigne autant. Et le moteur des registres ne fonctionne pas par magie : quand il n'y a plus de banque de registres, le noyau doit sauvegarder les registres en mémoire. Ce qui signifie qu'un appel profond en pile requiert plusieurs interventions du noyau.

### VI-B - ARM

Il s'agit d'une architecture que j'apprends encore. C'est une très bonne architecture : il y a seize registres généraux de trente-deux bits, chaque instruction fait exactement trente-deux bits de largeur, et les instructions peuvent être suffixées avec un drapeau pour désactiver l'exécution (ce qui s'appelle la *predication*). Puis il devient un peu plus désordonné quand on considère le jeu d'instructions *Thumb*. Basiquement, ce n'est que l'ARM avec la moitié des registres et des instructions de moitié réduites.

Comme toute architecture RISC moderne, ARM passe les arguments dans les registres : *r1*, *r2*... Il est intéressant de remarquer qu'il dispose aussi d'un moteur de sauvegarde des registres. Son fonctionnement m'est inconnu, mais, à l'appel de fonction, on déclare quels registres on va utiliser et qui ne sont pas membres de la liste d'arguments en entrée. À la fin de l'appel, on les restaure.

Une chose intéressante que fait ARM est qu'un appel ne met pas l'adresse de retour sur la pile. À la place, l'adresse de retour est sauvée dans un registre spécial, qui est accédé par l'instruction de retour. Puisque le registre est frappé, si une fonction appelle, elle doit sauvegarder le registre de retour - habituellement dans un autre registre.

## VI-C - Itanium (IA-64)

Maintenant, c'est une plateforme qui ne supporte pas la moindre comparaison. C'est malheureux qu'elle n'ait pas reçu plus d'attention, parce qu'elle a le potentiel d'être encore meilleure que tout ce que j'ai déjà vu.

Tandis que x86 a huit registres généraux, lx86-64 en a seize, ARM en a seize, SPARC en a trente-deux, Itanium possède cent vingt-huit registres généraux (avec quelques-uns en réserve). De ces registres, trente-deux sont fixes (comme les registres globaux de SPARC), et nonante-six tournent. Mais, à l'inverse de SPARC, les fenêtres de registres ont des tailles variables.

En plus des cent vingt-huit registres généraux, il y a aussi cent vingt-huit registres flottants d'octante-deux bits, aussi distribués en trente-deux fixes et nonante-six en rotation (même s'ils tournent ici pour une autre raison). Et même, il y a cent vingt-huit autres registres spécifiques, les *registres d'application* (en fait, ce sont les registres *ar0* à *ar128*, mais moins que cent vingt-huit registres sont implémentés). En plus, il y a soixante-quatre registres de prédicat, qui peuvent être utilisés pour empêcher toute instruction d'exécution, permettant, comme sur ARM, d'exécuter du code sous condition, sans devoir utiliser des sauts. En plus, il y a aussi huit registres de branche.

La seule chose bizarre dans cette architecture est la taille des instructions : quarante-et-un bits.

Oui, 41 bits. En fait, ce qu'il fait, c'est regrouper trois instructions dans seize octets (cent vingt-trois bits au total), et utiliser les cinq bits supplémentaires comme modificateurs d'instruction.

La convention d'appel, sur Itanium, utilise les registres généraux en rotation. À l'appel de fonction, les registres, qui débutent à *r32*, sont les registres d'entrée. Si la fonction prend trois entiers en paramètres, il seront dans *r32*, *r33* et *r34*. À l'appel de fonction, on déclare la taille du *jeu local* (entrée et local, comme SPARC), et la taille du *jeu de sortie*. Lors d'un appel, le moteur de la pile de registres déplace automatiquement le jeu local, en mettant le premier registre de sortie dans *r32*.

Des trente-deux registres fixes, il y a un couple de règles sur ce qui est effacé et ce qui est sauvegardé. À l'inverse de SPARC, le pointeur de pile est sauvegardé dans un registre global (*r32*). Ensuite, nous avons *r1*, un registre spécial : à la fois */dev/null* et */dev/zero*. Les lectures ont pour résultat zéro, et les écritures sont perdues (tout comme *%g0* de SPARC). *r1* est réservé au pointeur global : une valeur définie par l'appelant, qui est utilisée par l'appelé pour implémenter le PIC (*Position Independent Code*, code indépendant de toute position). *r2* et *r3* sont des registres de travail. *r4* à *r7* sont préservés. *r8* à *r11* sont des registres de travail. *r12* est le pointeur de pile. *r13* est le pointeur de thread. *r14* à *r31* sont des registres de travail.

Le nombre de registres de travail est très important ( $2 + 4 + 18 = 24$ ), pour que les fonctions qui n'appellent pas puissent stocker aisément des valeurs temporaires dans les registres, sans forcer le moteur de pile de registre à faire quelque chose. Les fonctions qui ont besoin de registres préservés peuvent tout simplement utiliser la partie locale du jeu en rotation (c'est-à-dire les registres qui commencent à *r32* jusqu'au premier registre de sortie).

Comme ARM, un appel de fonction ne met pas la valeur de retour dans la pile : elle est simplement sauvée dans un des registres de branche (par convention, *b0*). En fait, cela signifie que, sur Itanium, il est possible de nicher plusieurs appels sans toucher, ne fût-ce qu'une fois, à la pile - ni la pile normale, ni la pile utilisée par le moteur de pile de registres quand il a besoin de tourner et qu'il n'y a plus d'autres registres.

Quelqu'un m'a demandé dans mon article précédent pourquoi il n'était pas suffisant qu'Intel se limite à publier le jeu d'instructions. Voici une raison : c'est un système très complexe, et les développeurs avaient besoin de guidage pour les meilleures pratiques. Une autre est que, avec Intel qui publiait le **Software Conventions & Runtime Architecture Guide**, la firme a presque forcé tout le monde à s'entendre sur une séquence d'appel. À l'inverse de la situation actuelle sur x86-64, sur Itanium, les rôles de l'appelant et de l'appelé sont très bien définis (au moins en C).

Les deux documents supplémentaires qu'ils avaient préparés (**Processor-specific** ( et **C++ ABI**) voulaient standardiser par delà les plateformes. Microsoft a simplement ignoré ces deux documents - de toute façon, la *ps ABI* était prévue pour les systèmes d'exploitation basés sur le style de *System V* - mais ils ont été suivis par d'autres.



Quand il a migré son système d'exploitation de PA-RISC vers Itanium, HP a adopté les instructions de la *ps* ABI, choisi ELF comme format d'exécutable, et l'ABI C++.

Tout le monde, pourtant, a ignoré la suggestion d'Intel concernant l'emplacement des bibliothèques.

## VII - Conclusion

Comment ceci est-il lié au C++ ? En fait, les appels de fonction C++ sont juste des appels normaux C, même si le compilateur de Sun se plaint d'un anachronisme. Quand on appelle une fonction membre, un paramètre supplémentaire est inséré en première position (le pointeur *this*), le reste est identique.

Connaître la séquence d'appel est utile en cas de débogage en bas niveau. Seul, il n'amènera personne bien loin, mais c'est un début. Pour pouvoir déboguer un peu plus, il est nécessaire de connaître l'agencement des objets en mémoire, par exemple, au cas où les symboles ne sont pas à portée de main (par exemple, en cas de débogage d'un module de Qt, si gros que les éditeurs de liens n'ont plus de mémoire disponible lors d'une édition de liens avec les symboles de débogage). C'est ce que je faisais sur Solaris avec SPARC, jusqu'à ce que je remarque que l'éditeur de liens 64 bits pouvait lier avec les symboles.

C'est aussi utile de savoir comment le compilateur va passer les paramètres quand on décide d'implémenter une fonction. Un exemple est le cas de `QLatin1String`, abordé dans [un autre article](#).

On peut dire qu'il s'agit de microoptimisation, ce que ça pourrait bien être. Mais, même aujourd'hui, le cache de niveau 1 (L1) prend à peu près trois cycles, quand les registres n'en prennent aucun. Si la fonction est appelée *très* souvent, la différence devient flagrante.

Ou, vous pouvez aussi étaler vos connaissances lors de telles parties au bureau (on peut même gagner des prix en se souvenant de l'ordre des registres du x86-64 après trois bières).

## VIII - Divers

Merci à [yan](#) et [superjaja](#) pour leurs relectures et encouragement lors de la traduction, et à [koopajah](#) pour sa relecture orthographique !

Cet article n'est qu'une partie de l'iceberg ABI dévoilé par Thiago Macieira. Il a d'abord commencé par décrire [la compatibilité binaire](#).