Concept de développement des sequences BRUKER sous PV360

©Denis Grenier, CREATIS UMR CNRS 5220, Lyon, France

denis.grenier@creatis.insa-lyon.fr

Table of Contents

Contexte....2
- Concept de la partie interface utilisateur....2
- Concept de la partie déterministe (séquenceur)....3
- Suggestions de pratiques pour simplifier la création/modification des séquences IRM....3

Programmation de la partie PPL d’une méthode....4
- Les délais....5
- Notion de Listes....6
  - Definition d’une liste....6
  - Navigation dans une liste 6
- Les pulses....7
- Notion de pulse (power) list....8
- Acquisition....10
- Phase des pulses RF et de l’acquisition....12
- Boucles et tests de branchements....14
- Contrôle des gradients....18
    - grad_ramp....19
    - grad_shape....20
    - Contrôle temps réel des shims et preemphasis....21
    - Contrôle des gradients en parallèle....23
- Délais incompressibles....24
- TTL input/output....24

Partie interface utilisateur (langage C)....25
- Structure globale d’une méthode....25
- Structure de modification conseillée pour une méthode....26
- Création et visualisation de variables dans l’interface....27
  - Le fichier parsDefinition.h....27
    - Les relations et le fichier parsRelations.c....29
  - Le fichier ParsLayout.h....29
  - Fichier CallbackDefs.h....30
  - Le fichier .xml....32
  - Les Délais : Exemple ajout d’un délai....33
  - Programmer un module....34
  - Programmer des ajustements....35

Reconstruction....36
- Dump du schéma du pipeline existant....37
- Redirection et modification du PIPELINE....40
- Récupération « en vol » des données et traitement dédié....42
  - RecoSystemFilter....42
  - RecoMethodFilter....42

Contexte

Pour développer des séquences IRM Bruker se base sur deux contextes différents :

- Réaliser une acquisition nécessite une phase de prescription pendant laquelle l'utilisateur « joue » avec les paramètres d'une séquence pour l'adapter à ses besoins (modifie le TR, TE ...). Cette étape est une étape de préparation, d'interaction entre l'utilisateur et l'interface. Elle n'est pas déterministe en ce sens que les interactions homme-machine continuent jusqu'à ce que l'utilisateur soit satisfait de son setup.
- Cette étape terminée, l'utilisateur va lancer la séquence. Lors de cette étape, il donne le contrôle total à la machine pour exécuter d'une manière déterministe la séquence en utilisant les paramètres prescrit dans la phase précédente.

Programmer une séquence Bruker passe donc par 2 étapes totalement distinctes :

- Utiliser le langage C pour programmer l'interface de prescription des paramètres de la séquence.
- Utiliser un langage de bas niveau (Pulse Programming Language (PPL)) pour pouvoir programmer dans le séquenceur temps réel une description abstraite de la séquence.

Concept de la partie interface utilisateur

Pour éviter que l'utilisateur ne puisse accéder à des parties de codes propriétaire ou à des fonctionnalités qui pourraient être destructrices pour le matériel, la partie programmation de l'interface ne permet pas de créer un programme (qui contrôlerait tous les aspects du spectromètre et dont le code source contiendrait TOUTES les infos et secrets de programmation du spectromètre) mais juste une **librairie** qui DOIT comporter 2 fonctions :

- la fonction **void init(void)** qui, comme son nom l'indique sera appelée par le programme principal pour initialiser l'état du spectromètre de manière à ce que toutes les cases mémoires (variables) dont il a besoin pour tourner correctement aient une valeur (correcte).
- la fonction **void loadmeth(void)** qui, lorsqu'elle sera appelée par le programme principal va charger en mémoire le cas de réalisation courant prescrit par l'utilisateur.

Tout ce que vous aurez à programmer dans la partie interface d'une séquence seront des lignes de codes dans des fonctions, appelées par des fonctions … appelées par la fonction init() ou la fonction loadmeth().
Une fois compilée, la partie interface de la méthode maSpinEcho se retrouve sous la forme de la librairie maSpinEcho.so (ShareObject) qui contiendra les fonctions init() et loadmeth() (qui incluront les fonctions… qui incluront vos modifications.

Concept de la partie déterministe (séquenceur)

La partie déterministe de la séquence se programme dans un fichier PPG qui sera lui aussi compilé pour être injecté dans le séquenceur de l'IRM.

Cette partie consiste à décrire les actions qui devront être réalisées en temps réel par la machine, elle se compose en instructions qui invoquent différentes fonctions hardware :

- des **délais** définissant une durée (s).

- des **formes RF** ayant une **durée** d'application (s), une **forme normalisée** (variation d'amplitude/phase), une **phase** d'application (exprimée en degrés ou segments de cercles) (dans le plan xOy), une **fréquence** d'application (Hz) et une amplitude (W)

- Des **gradients** de champ magnétique caractérisés par : leur **durée** (s), leur **forme** normalisée, leur **direction** d'application dans le repère (R,P,S), leur **amplitude**.

- L'**acquisition** du signal par un des 8 « **jobs** » cumulables, une **durée** d'acquisition (s) pour chacun de ces « job », un **routing** (voie) pour chacun de ces jobs, une **phase** (°), un **offset en fréquence.**

- Des **boucles** et test de **branchements**.

- Des signaux de **trig** entrants ou sortants.

Suggestions de pratiques pour simplifier la création/modification des séquences IRM

Attention :
La première lettre d'une méthode « maison » doit impérativement commencer par une minuscule, c'est une manière de voir que ce n'est plus une methode développée par Bruker

On ne réalise pas une séquence « from scatch » il faut trouver dans la bibliothèque Bruker la méthode la plus proche de ce que l'on veut faire et utiliser la fonction copyMethod pour en créer une version que l'on pourra modifier.

La manière la plus conviviale de développer et surtout debugger une méthode Bruker est d'utiliser l'environnement de Eclipse.
Les gros avantages de Eclipse par rapport à l'environnement Bruker sont de permettre le debuggage pas à pas du code en utilisant de vrais outils (gdb), d'avoir une aide contextuelle puissante et de faire des recherches de textes et fonctions au sein de tous les fichiers du projet.
Pour faire tout cela, il faut :

1. d'abord utiliser l'interface Bruker pour faire le copyMethod
2. Lancer Eclipse et y importer la nouvelle méthode en utilisant File→Import→ C/C++ → Existing Code as MakeFile Project.
   Compléter le nom de projet (en lui donnant le nom de la méthode copiée pour plus de clarté mais tout autre nom est possible), faire pointer « Existing Code Location » vers votre méthode (sans doute

$PVPath/prog/curdir/<votreLogin>/Paravision/methods/src/<votreNomDeMethode>)

3. Configurer les directives de compilation dans Eclipse de manière à ce que vous n'ayez qu'à cliquer pour lancer la compilation et l'installation de la séquence.
4. Tant que la méthode n'est pas finalisée, il faut activer l'option PARCOMP_OPTS = -g dans le fichier Makefile de la méthode pour pouvoir utiliser gdb.

Au cours du développement d'une méthode, il est plus cohérent de **travailler d'abord sur la partie séquenceur de la méthode (.PPG)** cela permet de définir exactement ce que sera la séquence en termes de délais, RF, gradients, acquisition, boucles et branchements et **ensuite on travaille sur l'interface utilisateur** pour mettre en correspondance les cases mémoires encore vides du séquenceur avec les paramètres que l'utilisateur imposera à la machine.

Programmation de la partie PPL d'une méthode

Pulse Program Code

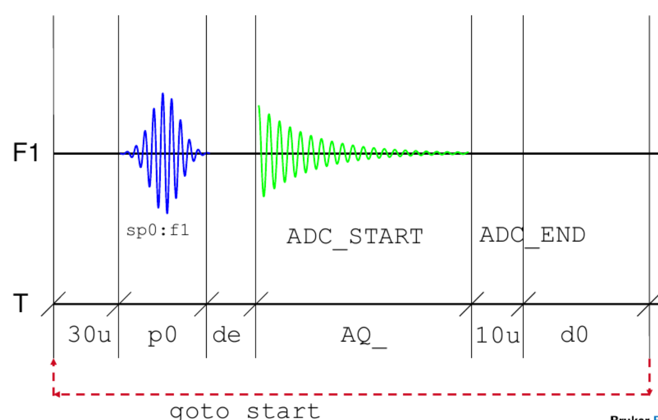
```
#include <MRI.include>

INIT_DEVICES
;-------------start of the
start, 30u
       (p0:sp0 ph0):f1                 ; excitation pulse
       ADC_INIT_(job0, ph1, ph
       AQ_(job0)   ADC_START_(j
       10u  ADC_END_(job0)             ; data
       d0                              ; repe
       lo to start times NR            ; repetition loop
       SETUP_GOTO(start)
exit
ph0 = 0                                hase for excitation
ph1 = 0                                hase for accumulation
```

Timing definitions
Transmitter Control Commands
Acquisition Control Commands
Comments
Labels and Loops
Phase definitions

9 Bruker BioSpin

Pulse Sequence Design



6 Bruker BioSpin

Les délais

Attention : Tous les délais du PPG sont exprimés en secondes (sauf evidement ceux qui sont dimensionnés : 1u, 2m

Dans un programme PPG les délais peuvent être indiqués de manière différente :

- Des délais constants :

```
0.3u , 1.2m , 0.5s
```

Attention : Ces variables d* ont des assignations « rituelles » => d4 = rise time, d0 filling time pour TR, … mais c'est juste une « vieille » convention !

- Des délais « prédéfinis à l'ancienne » :

```
d1,d2 … d64
```

Ces variables prédéfinies sont déjà connues du séquenceur, sous la forme d'un registre de variables de type float (16 ou 32 bits ?) dont le « synonyme » du côté de l'interface utilisateur sont le pointeur D[0] … D[64].

Attention : Le nom d'une variable dans le PPG ne doit pas dépasser 16 caractères sous peine d'être tronquée.
=> Dans le PPG, les variables
abcdefghijklmnopQRST
et
abcdefghijklmnopU
pointent vers la même case mémoire, celle d'une variable appelée : abcdefghijklmnop

- Des délais définis de manière « classique » :

```
define delay joe = {$PVM_Var}
```

Dans laquelle PVM_Var est une variable de type double définie dans la partie interface de la méthode. Ou encore :

```
define delay jack

« jack = td*dw »
```

- Des listes de délais :

Un délai appliqué peut être nul mais évidement pas négatif

```
define list<delay> william = {$PVM_mon_paramètre_1D}

define list<delay> averell  = { 0.12 40.0 52.1 }
```

L'horloge du spectromètre étant à 80MHz, les délais sont arrondits pour être des multiples de 12.5ns

Notion de Listes

Definition d’une liste

```
define list<delay> william = {$PVM_mon_paramètre_1D}

define list<delay> averell  = { 0.12 40.0 52.1 }
```

Attention :
si vous utilisez les listes de fréquences prédéfinies (`fq1-8`) elles ont un auto-incrément, ce qui n'est **pas** le cas des listes que vous définissez vous-même.

La même syntaxe est utilisée pour créer des délais, des fréquences, des amplitudes de pulses, des amplitudes de gradients et des indices de boucles de contrôle en utilisant les mots clefs **`delay`** (en s), **`frequency`** (en Hz), **`pulse`** (en µs), **`power`** (en W) **`gradient`** (en %) et **`loopcounter`**

Navigation dans une liste

`william.inc` (increment) passe à la valeur suivante

`william.dec` (decrement) revient à la valeur précédente

`william.res` (reset) revient au premier indice

`william.len` (length) retourne le nombre d'indices

`william.idx = 3` (index) fait pointer directement sur la troisième valeur de la liste.

Attention :
`william.idx=1`
contient ce qui est à
`william[0]`

`william[2]` contient aussi la 3e valeur (la première est à l'indice 0)

toutes ces opérations se font sur un buffer circulaire :
william[max+1]=william[0] et william[0-1]=william[max]

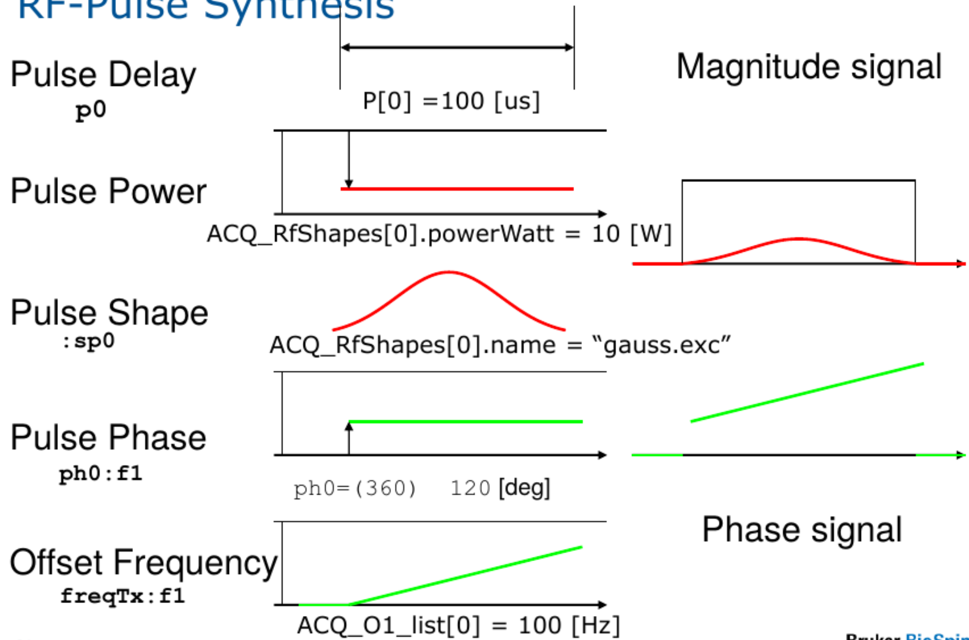
De même, si la liste n’a que 3 valeurs william[7]=william[1]

Les pulses

Attention : Dans un PPG les délais sont exprimés en secondes **SAUF** pour les pulses pour lesquels ils doivent êtres donnés en **µs**

Malgré le flou présent dans la doc Bruker, dans un PPG, un délai associé à un pulse **doit** s'appeler `p0..p64` car, là encore, les variables `p0-63` et `sp0-63` sont les synonymes d'adresses mémoires bien précises du côté de l'interface utilisateur : respectivement le pointeur `P[i]`, la structure `ACQ_RfShapes[i]` et la structure `ACQ_O1_list[i]`

```
1m          fq1:f1      ; prepare carrier frequency
(p1:sp0 ph0):f1        ; pulse delay p1 with shape sp0 (defined
                        ; in ACQ_RfShapes[0]) on channel f1 with
                        ; phase ph0
```

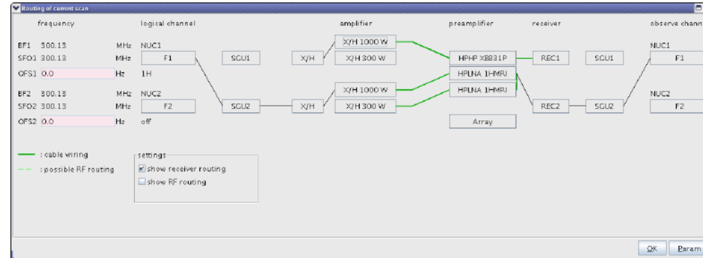


Attention : Les 8 canaux f1-f8 n'ont aucun lien avec les 8 « jobs » disponibles pour l'acquisition

`f1` défini le canal (routage) du signal d'émission/réception c'est-à-dire les synthétiseurs de fréquences et amplis utilisés en émission ainsi que les préamplis utilisés pour la réception

RF-Excitation: The Channel Concept

- Channels linked to selected nuclei and associated frequency
- RF Transmitters routed to RF Channels F1 … F8
- RF Channels are addressed by `:f1`(default)-`:f8` in pulse program
- Actions on separate channels can be performed simultaneously



Bruker BioSpin

Notion de pulse (power) list

Même si les pulses doivent s’appeler p0→p64 pour que leurs caractéristiques soient bien prises en concidération par le PPG, il existe aussi la notion de power lists : pl0→pl31[:f1-f8]

define list<power> <name> = { $<parametername> }

or

define list<power> <name> = { Watt <power value list> }

Attention : Les puissances utilisées dans ces listes sont des puissances « brutes » qui ne tiennent pas compte des valeurs de cortab utilisées pour linéariser la puissance envoyée.
Il faut donc utiliser cette technique seulement pour de faibles puissances correspondant à une zone de linéarité de l’ampli

Exemple

Considérons que

PLW[1] = 0.3W (30dB)

PLW[2] = 30W (10dB)

ACQ_RfShapes[0].power = 300W (0dB)

ACQ_RfShapes[1].power = 300W (0dB)

define list<power> pwUsr = { Watt 0.03 } ; (40dB)

120up ; default power level is value of PLW[1], i.e 30dB

100u pl2:f1 ; switch power level of channel 1 to PLW[2] i.e. 10dB

120up ; next blockpulse generated with pl2

100u pwUsr:f1 ; switch power level of channel 1 to 40 dB

120up:sp0(currentpower) ; l'instruction currentpower lui dit d'utiliser la puissance courante (40 dB) plutôt que ACQ_RfShapes[0].power définie par défaut pour sp0

100u ; after shape return to default power

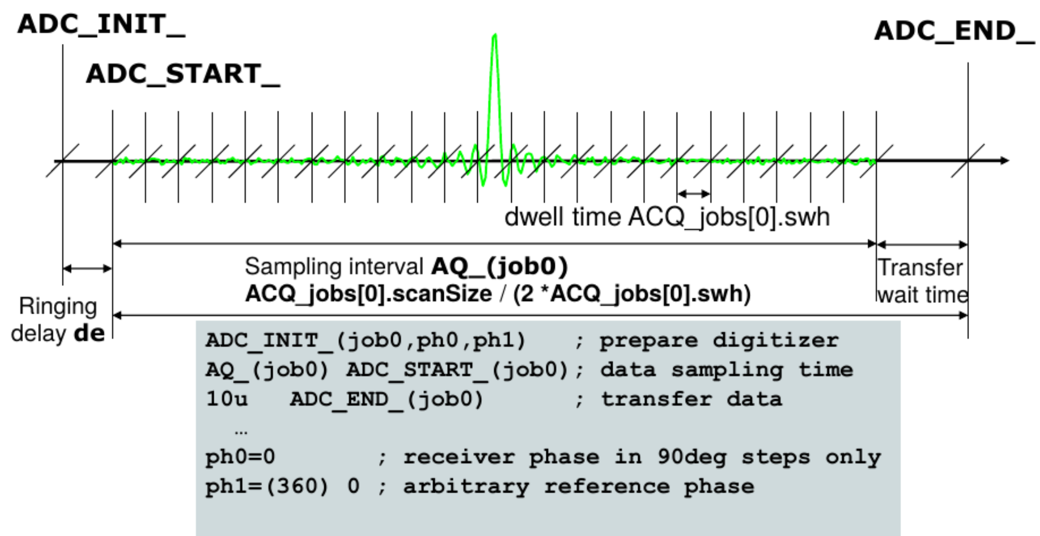
120up:sp1 ; use power level defined for shape1: 0 dB!

exit

Acquisition

L’acquisition se fait par le biais de « job ». Une séquence peut comporter jusqu’à 8 jobs différents chacun d’entre eux possédant ses propres caractéristiques de durée d’acquisition, nombre de points, bande passante, fréquence, phase ainsi que son propre pipeline de traitement post digitalisation : filtrage, fft … sauvegarde dans un fichier ou non …

Data Sampling Overview



43 Bruker BioSpin

Data Sampling Command Summary

- As a convention, for image data the title `job0` is used.
- `ADC_INIT_(<job>,<receiver phase>,<reference phase>)`
  - Prepare digitizer and set receiver (data combination) phase and reference signal phase
  - Has implicit delay **de** (parameter **DE**)
- `ADC_START_(<job>)`
  - Marks sampling of first data point – minimum delay `AQ_(<job>)`
- `ADC_END_(<job>)`
  - Initializes data transfer
  - needs minimum delay of 10u

44 Bruker BioSpin

La macro **SWITCH_(job)** permet de passer d'un job d'acquisition à un autre. Elle prend comme argument le prochain job qui doit être acquis. Elle est obligatoire si vous utilisez plus d'un job dans un PPG. La commande va charger les paramètres de filtrage matériel et commuter le gain du récepteur à la valeur associée au job considéré dans le paramètre ACQ_job[n].receiverGain.

Comme les étages de gain du récepteur ont un temps de stabilisation non négligeable, la macro **SWITCH_** est nécessaire pour permettre un contrôle total et un timing optimisé pour cet événement.

Exemple :

```
define delay minrgtime;
"minrgtime = 60u - de"
define delay minwait = {$ACQ_RxFilterInfo[0].minTime}
"minwait = minwait / 1e6"
minrgtime SWITCH_(job0) ; sets receiver gain for job0
ADC_INIT_(job0,ph0,ph1) ; lasts de
AQ_(job0) ADC_START_(job0)
minwait ADC_END_(job0) ;
60u SWITCH_(job1); sets receiver gain for job1
```

Phase des pulses RF et de l’acquisition

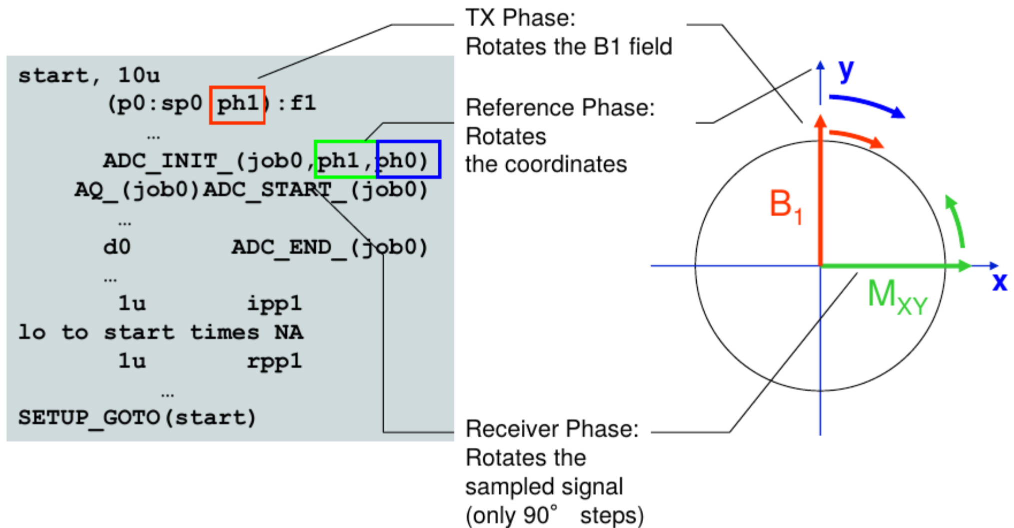
Phase Lists

- **Phase** lists can be used to provide phase **offsets** to **transmitter** or **receiver**
- Phase lists **ph0-ph31** are defined at the **end** of the **pulse program**:

  ph<x> = (<divisor / defaults to 4 when missing>)
  < whitespace separated list of phase values
  meaning multiples of 360/divisor >
- The **next** member of a phase list can be selected with the **ipp0 – ipp31** command
- A phase list can be **reset** to the first value by the **rpp0 – rpp31** command
- Example

```
ph0=       0 1 2 3  ;  0   90 180 270 degrees
ph1= (90)  0 1 2 3  ;  0   90 180 270 degrees
ph2=(360)  0 1 2 3  ;  0    1   2   3 degrees
```

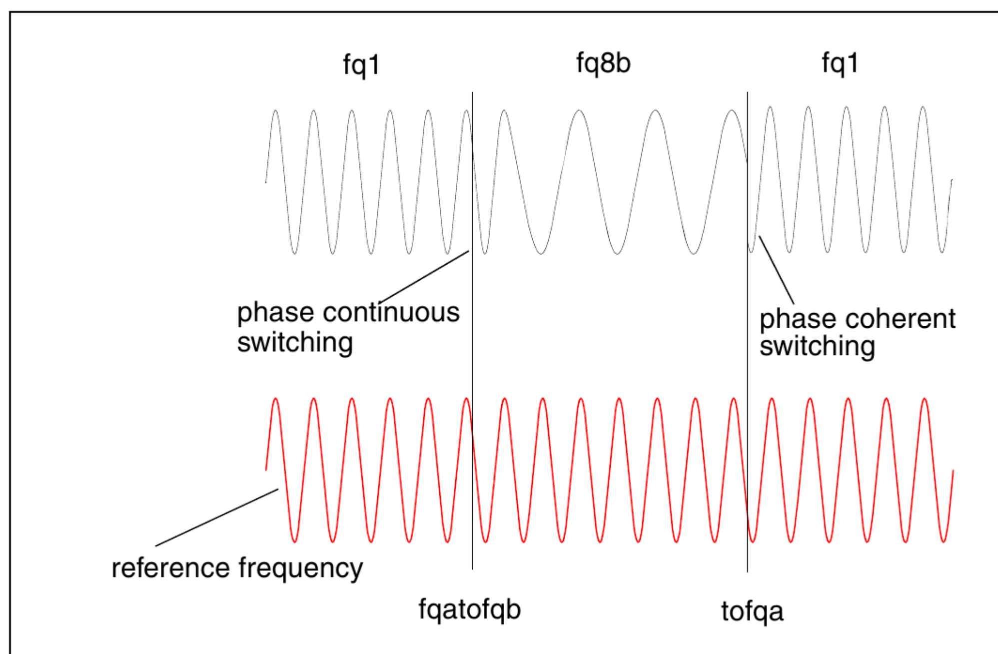
Phase Cycling, interpretation



```
Start, 10u
(p0:sp0 ph1):f1
ADC_INIT_(job0, ph2,ph0)
AQ_(job0) ADC_START_(job0)
d0    ADC_END_(job0)
```

Par exemple si **ph1** = 0 cela fait faire une rotation autour de l'axe 0 (c.a.d. *x* par défaut) ce qui, dans le cas d'une impulsion 90° place alors M_0 sur *y*. *S*i l'on veut acquérir un signal correctement phasé (c.a.d. partie réelle maximum) son acquisition doit alors se faire suivant y et il faudrait **ph2**=1 .

ph0 le second paramètre de ADC_INIT_ est là pour donner la phase « absolue » du signal de référence indépendamment des variations que peut lui faire subir des *switch* en fréquence :



Boucles et tests de branchements

```
joe,
1m
loop to joe times NA
3u
if « I0 < 5 » goto joe
5m
if « I0 > 6 » goto jack
2m
jack,
exit
```

Les assignements utilisent les opérateurs classiques : + - * / = += -= *= /= plus une bonne liste de fonctions mathématiques : abs, acos, asin, …, sqrt, pow, trunc

Les tests se font aussi avec les opérateurs standards : ==,!=, &&, ||, <, >, > =, < =, !

Le point primordial est de bien garder en mémoire que si les conditions ne sont pas mises entre des guillemets, les conditions sont testées avant le lancement du scan et les parties qui ne remplissent pas les conditions ne seront jamais exécutées par le séquenceur.

Les conditions qui sont mises entre guillemets sont testées à chaque fois que le séquenceur les rencontre pendant l'acquisition.

Par exemple si on a décidé dans l'interface d'activer des bandes de saturation il est inutile de tester le branchement vers le module de sat à chaque répétition => pas de guillemets pour ce test.

Mais si par exemple on veut faire des dummy scans tant que le trig n'est pas OK on a intérêt à tester l'état du trig à chaque répétition => test entre guillemets

```
<label>,
↓
loop to <label> times <loopcompteur>
↓
if <condition> goto <label>
if <autre condition>
{
↓
}
else
{
↓
}
```

```
define loopcounter monCompteur {$MonParametre}
start,
if monCompteur ==3
{
1s
« monCompteur = monCompteur -1 »
if « monCompteur > 0 » goto start
}
else
{
1m
}
exit
```

Dans l'exemple ci-dessus, si *MonParametre* a été initialisé à 3 dans l'interface utilisateur, la séquence durera 3 secondes et *monCompteur* vaudra 0 à la fin.

Si *MonParametre* a été initialisé à une valeur ≠3 dans l'interface, la même séquence durera 1 milliseconde et à la fin *monCompteur* vaudra toujours x

Très important : à partir de PV360 (PV7?) la notion des boucles imbriquées ayant implicitement une fonction précise (=> NA nbr d'accumulation, NS nbr de scans, NE, NI, NAE, NR) n'est plus vraie. Les séquences continuent souvent par habitude à les utiliser mais ce ne sont plus eux qui dictent ce qui

se passe pour le signal dans une boucle précise. Cette notion est maintenant déléguée à un mécanisme plus moderne :

Counting Scans

- The number of scans to be acquired in an experiment is defined by the number how often `ADC-` commands are called.
- The `exit` command stops acquisition
- The total number of scans acquired must match the value of **ACQ_jobs[].nTotalScans**
- Standard ACQP parameters can be used as loop counters to implement default Cartesian acquisition loops:
  - **NSLICES** number of slices
  - **NI** number of image objects, e.g. echoes
  - **NS/NA** number of averaged scans/encoding steps
  - **NAE** number of averaged experiments
  - **NR** number of repetitions

52 Bruker BioSpin

Loop programming

```
; pulse program
start,
    lo to start times NI
    lo to start times NA
    lo to start times L[0]
    lo to start times NAE
    lo to start times NR
```

```
// method code

appendLoop(NI,LOOP_SETUP);
appendLoop(NA,LOOP_AVERAGE);
appendLoop(L[0]);
appendLoop(NAE,LOOP_AVERAGE);
appendLoop(NR);
```

```
ACQ_jobs[0]:    nTotalScans        NI*NA*L[0]*NAE*NR
                transactionBlocks  NI
ACQ_ScanPipeJobSettings[0]:        innerAccumLoops   NI
                                   outerAccumLoops   NI*L[0]
                                   nAccumScans       NI*L[0]*NR
ACQ_ScanInnerAccumWeights[0][0]: NA
ACQ_ScanOuterAccumWeights[0][0]: NAE
```

53 Bruker BioSpin

la structure de boucle que l'on utilise dans le PPG doit être reflétée pour les mêmes variables et dans le même ordre du côté de l'interface.

Dans l’exemple ci-dessus, NI est juste un entier qui définit par exemple le nombre d’échos acquis dans une séquence multi-échos. Dans ce PPG il est associé à la boucle la plus interne c’est-à-dire celle qui sera exécutée au total le plus grand nombre de fois.

Dans l’interface utilisateur la première commande :

- **job0->appendLoop(NI,LOOP_SETUP)**

  Initialise pour le système d’acquisition le fait que NI intervient comme multiplicateur dans le nombre de fois que les job0 va acquérir le signal. Le keyword LOOP_SETUP indique que, en mode setup, le job0 n’est concerné à chaque TR que par ce loop là.

  À la sortie de cette boucle, on a en mémoire un objet de dimension NbrDePointsCplxAcquis * NI

- **job0→appendLoop(NA,LOOP_AVERAGE)**

  Cette seconde commande indique au mécanisme de gestion de boucle que la seconde boucle est une boucle accumulant NA fois les NI signaux acquis.

  À la sortie de cette boucle on a toujours un objet de dimension NbrDePointsCplxAcquis * NI

- **job0→appendLoop(L[0])**

  Cette troisième commande indique à l’acquisition qu’il y a une troisième boucle (sans comportement spécial).

  À la sortie de cette boucle on a donc un objet de dimension NbrDePointsCplxAcquis * NI * L[0]

- **job0→appendLoop(NAE,LOOP_AVERAGE)**

  Cette quatrième commande indique une nouvelle boucle pour laquelle l’objet de taille NbrDePointsCplxAcquis * NI * L[0] sera accumulé NAE fois.

  À la sortie de cette boucle on a toujours un objet de dimension NbrDePointsCplxAcquis * NI * L[0]

- **job0→appendLoop(NR)**

  Cette dernière boucle permet de créer un objet de taille

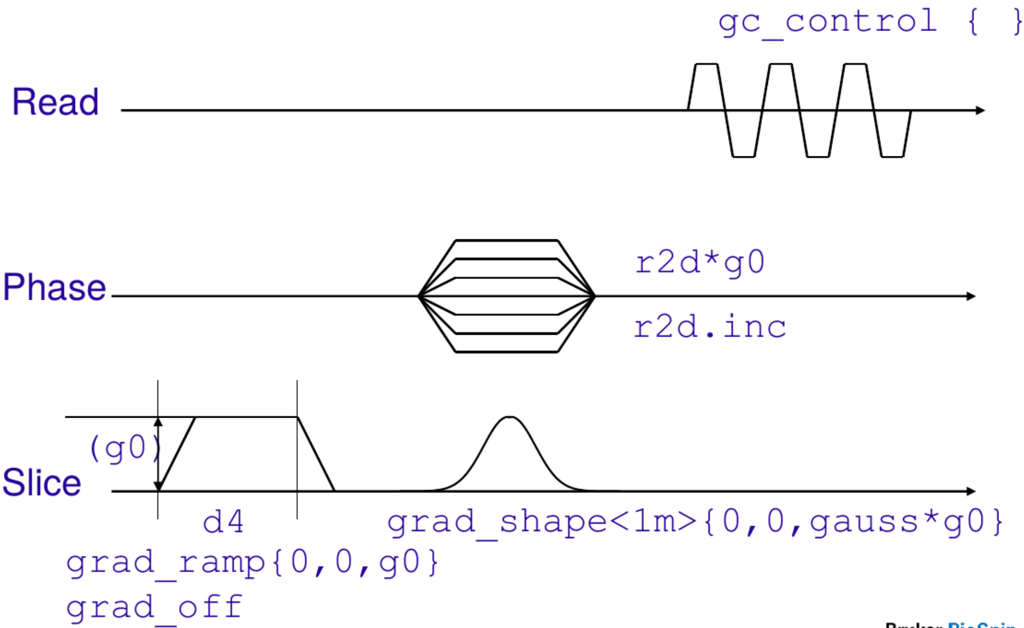
  NbrDePointsCplxAcquis * NI * L[0] * NR .

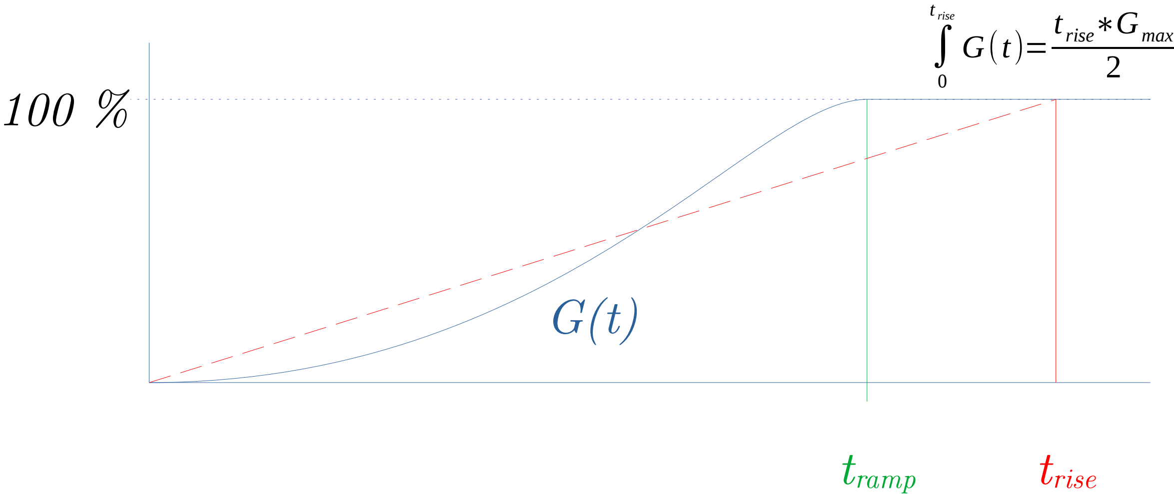
En théorie, chaque job est associé à son propre système de boucles
=> A votre charge de faire un PPG qui puisse gérer la cohérence des acquisitions de tous vos jobs à la fois !!!!
(bonne chance)

Job0 s’arrêtera automatiquement quand il aura réalisé :

ACQ_jobs[0] = NI * NA * L[0] * NAE * NR acquisitions de NbrDePointsCplxAcquis

Contrôle des gradients





grad_ramp

```
d1 grad_ramp { <read> | <phase> |<slice>}
```

L'amplitude des gradients et un certain nombre de fonctions peuvent être placées dans <read>, <phase>, <slice> :

- Des scalaires : `5` représente une amplitude de 5 %

- Des variables : `g0...g99` qui pointent sur les valeurs définies dans la partie method en utilisant `ACQ_gradient_amplitude[0-99]`

- Des listes de gradients normalisées à 1, prédéfinies :
    - `r1d, r2d, r3d` sont des listes d'amplitudes dont le nombre et la valeur des éléments est déduite des valeurs de **ACQ_phase_encoding_mode[0-2]**, **ACQ_phase_encoding_start[0-2]**, **ACQ_spatial_phase_0-2**.
    - `sin()` et `cos()`, fonctions allant de 0 à 2pi (exclus), le nombre d'élément doit être défini avant
    - `sinp()` va de 0 a pi.
    - `gauss<truncval>()` gaussienne tronquée à truncval
    - `step()` rampe linéaire de 0 à 1
    - sin90 ?
    - cos90
    - Cosp ?
    - Plusm ?

  Exemple : define list<grad_scalar,128> sinp
  => sinp devient une liste de 128 valeurs réparties entre sin(0) et sin(pi)

- Des listes de gradients définies par l'utilisateur :

```
define list<grad_scalar> mygradlist = {amplitude1,...,amplitude n}
define list<grad_scalar> mygradlist = {$PVM_MaGradList}
```

Les indices de listes sont gérés par les préfixes :

increment index → .inc
decrement index → .dec
reset index → .res
store current index → .store
restore stored index → .restore

Ex :

```
d1 grad_ampl{ (g0+5)*2.1 | r2d*sin | mygradlist *1.0 *r3d}
3u r2d.inc sin.res mygradlist.dec r3d.store
```

La fonction *ramp_time* peut prendre des options :

grad_ramp<ramptime> impose un ramptime en µs (qui doit rester supérieur à celui du système de gradients)

grad_ramp<coordinates> indique que les gradients doivent être appliqués dans le repère de l'aimant (XYZ) → ***magnet_coord*** ou dans le repère classique (RPS) → ***object_coord***

grad_ramp<ramptime,coordinates>

Example

```
1u grad_ramp<120u>{100,0,0} ; leads to a runtime error,
;when the system ramp time is >120u
1u grad_ramp<120u, magnet_coord> {0,-10,0} ; switch Y
; direction to -10 percent within 120u
```

grad_off est un raccourcit pour la commande grad_ramp{0|0|0}

grad_shape

La fonction grad_shape permet de définir la forme du gradient au cours du temps

la résolution minimale d'une forme de gradient est 8us

define list<grad_shape> myshape = { -1, 0, 1 }

define list<grad_shape> myshape = {$myPVMarray1D}

100u grad_shape{100*myshape(),0,0}

1u grad_shape<100u>{100*myshape(),0,0}

Les fonctions suivantes peuvent être utilisées pour définir des formes et des listes de gradients. Lorsque n points sont requis, les fonctions sont appliquées à la plage donnée de manière équidistante.

sin: sine function from sin(0) to sin((n-1)/n*pi)

sinp: sine function from sin(0) to sin(pi)

sin90: sine function from sin(0) to sin(pi/2)

cos: cosine function from sin(0) to sin((n-1)/n*pi)

cosp: cosine function from sin(0) to sin(pi)

gauss<truncation>: gauss function truncated at given truncation level, e.g. gauss5

truncates at 5%level

plusm: alternating function, switching between +1 and -1 in each step

step: linear ramp from 0 to 1

Dans les versions précédentes, les fonctions de rampe r1d, r2d et r3d étaient définies et implicitement liées aux paramètres ACQ_spatial_phase_0-2. Comme ce lien n'était pas simple, les fonctions intégrées ne sont plus disponibles. Dans certains programmes d'impulsion, les macros r1d-r3d sont définies.

#define r1d ACQ_spatial_phase_0

Ils se comportent de la même manière que les anciennes commandes de rampe.

Contrôle temps réel des shims et preemphasis

- Commandes pour recharger les gradients durant le PPG

  relgrad; reload les gradients pendant la partie setup si GS_typ = Gradients ou Preemphasis.

  Pendant un GSP on peut utiliser reload B0 :

  - 12u reload B0 ; reload B0 values

- Avec nos preemphasis il est possible d'agir sur la valeur de B0 pour conserver la fréquence de résonance au cours d'une manip très longue : Au début de l'acquisition une valeur de B0 est chargée, un pipeline la recalcule au cours du temps et injecte régulièrement une valeur correcte dans la GCU. Cette nouvelle valeur est chargée à chaque appel de la commande

  reload B0

- le même principe peut être utilisé avec les shims X,Y,Z,Z0 avec l'instruction reload_shims :

  - 10u reload shims ; reload shims
  - 10u reload all; reload shim and B0 value

- ctrlgrad <Data>

  On AVIII (HD) systems it is possible to send control information to the Digital Preemphasis Unit :

  - ctrlgrad 1; Blank X Gradient
  - ctrlgrad 2; Blank Y Gradient
  - ctrlgrad 4; Blank Z Gradient
  - ctrlgrad 8; Blank B0 Gradient

- ctrlgrad 15; Blank all Gradients
- ctrlgrad (1*16); resets all gradients immediately to zero
- ctrlgrad (2*16); ramp down all gradients to zero
- ctrlgrad (4*16) ; does the same as the reload B0 command
- ctrlgrad (8*16) ; does the same as the reload shims command

The following are commands to active shim sets in a dynamic shim experiment.

- ctrlgrad (1*256) ; activate first shim set
- ctrlgrad (2*256) ; activate next shim set
- ctrlgrad (3*256) ; activate previous shim set
- ctrlgrad (4*256) ; save index of actual shim set
- ctrlgrad (5*256) ; restore shim set

Contrôle des gradients en parallèle

Le timing des gradients peut être contrôlé indépendamment de celui des RF et de l'acquisition.

L'instruction *gc_control* permet de créer un bloc d'instructions liées aux gradients qui se déroulera en parallèle des instructions suivantes (qui devront donc ne pas faire intervenir les gradients).

Exemple :

```
2u gc_control

{; Begining of gc_ control block

d1 grad_ramp{ 20,0,0}; Gradients statements

d2 groff ; within gc_control

loop L[10] ; loop within gc_control

{

d1 grad_ramp{ 10,0,0 }

d2 grad_ramp{ -10,0,0 }

}

if (Spoiler); Conditional compiling is supported

{

d3 grad{ 20, 20, 20 }

}

groff ; last gradient without delay

}; End of gc_ control block

d4 ADC_INIT(ph0, ph1); TCU continues at this point after 2us
```

Délais incompressibles

Timing Restraints

When you develop a pulse sequence, you must bear in mind certain restrictions for the timing. The following table gives safe values, which can be used for all hardware configurations. Shorter timing may be possible on certain systems. References to parameters which are typically used to describe corresponding time intervals are given, where they exist:

Pulse program action	Timing	Parameter
Resolution for delays	12.5 ns	-
Minimum length of delay	50 ns	
Minimum length of pulse delay	100 ns	
Time resolution for pulse shapes	100 ns	-
Minimum time between two shaped pulses	4 us	-
Minimum time between a pulse and data acquisition	ca. 6 us	DE
Minimum time between end and start of acquisition	ca. 700 us	-
Minimum time between start and end of scan		DEOSC
Length of Gating pulse: (depends on amplifier)	100 us	D[8]
Time for phase switch to become effective	4 us	
Time for amplitude switch to become effective	2 us	
Time for frequency switch to become effective	2 us	

Table 3.2: Typical timing for pulse program actions

You should note, that timing restraints in most cases are hardware specific.

TTL input/output

ADVANCE NEO permet d’utiliser 2 lignes TTL en output

TTL1 et TTL2

exemple d’envoi d’un TTL de 10us sur la voie 1 :

10u TTL1_HIGH

0.0125u TTL1_LOW

à la fin d’une manip les trig sont automatiquement remis à leur valeur de base qui est HIGH

de même il existe des lignes permettant l’injection de signaux TTL :

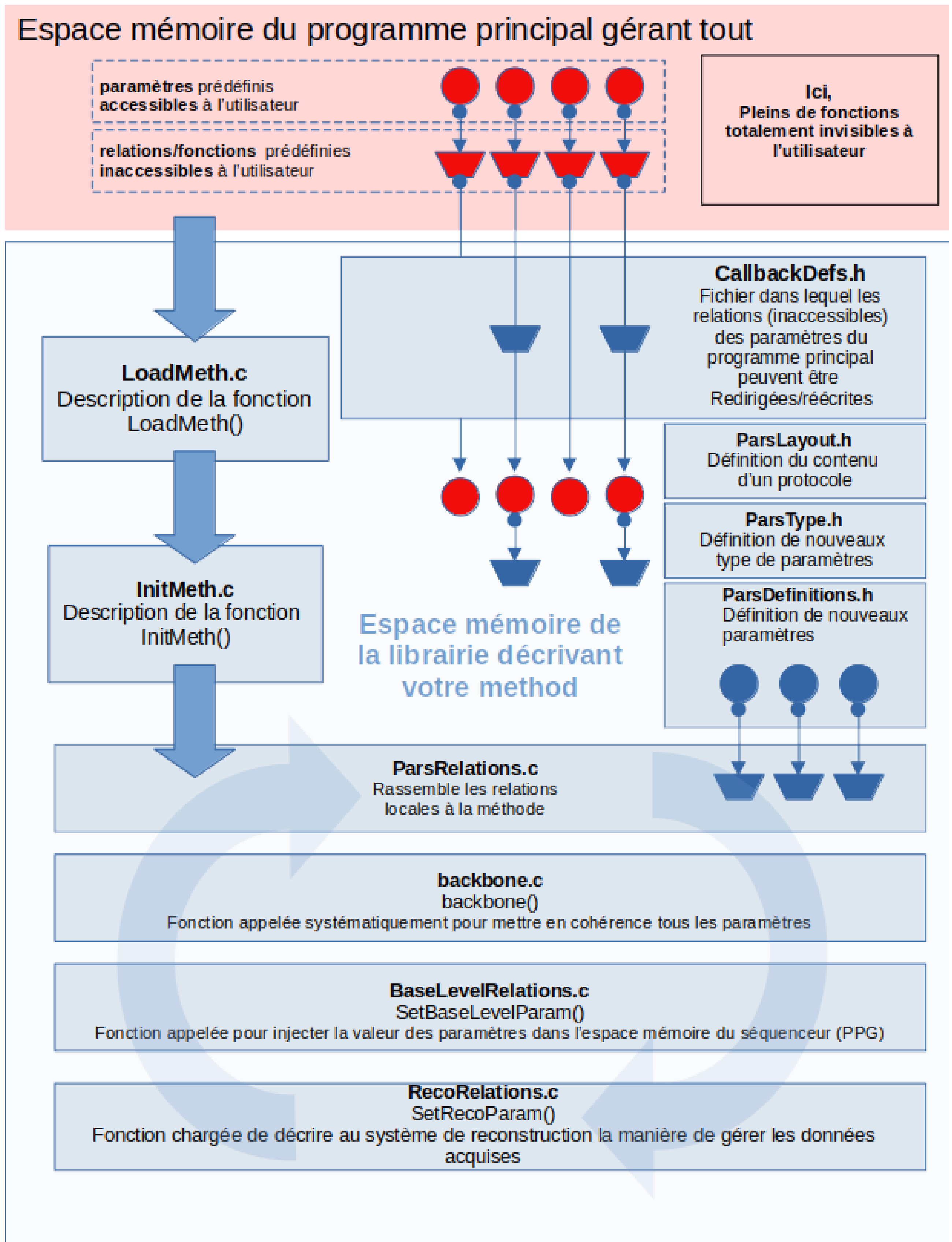
trignl1 (ECG TRIG), trignl2 (TTL IN 1), trignl4 (TTL IN 2)

ex :

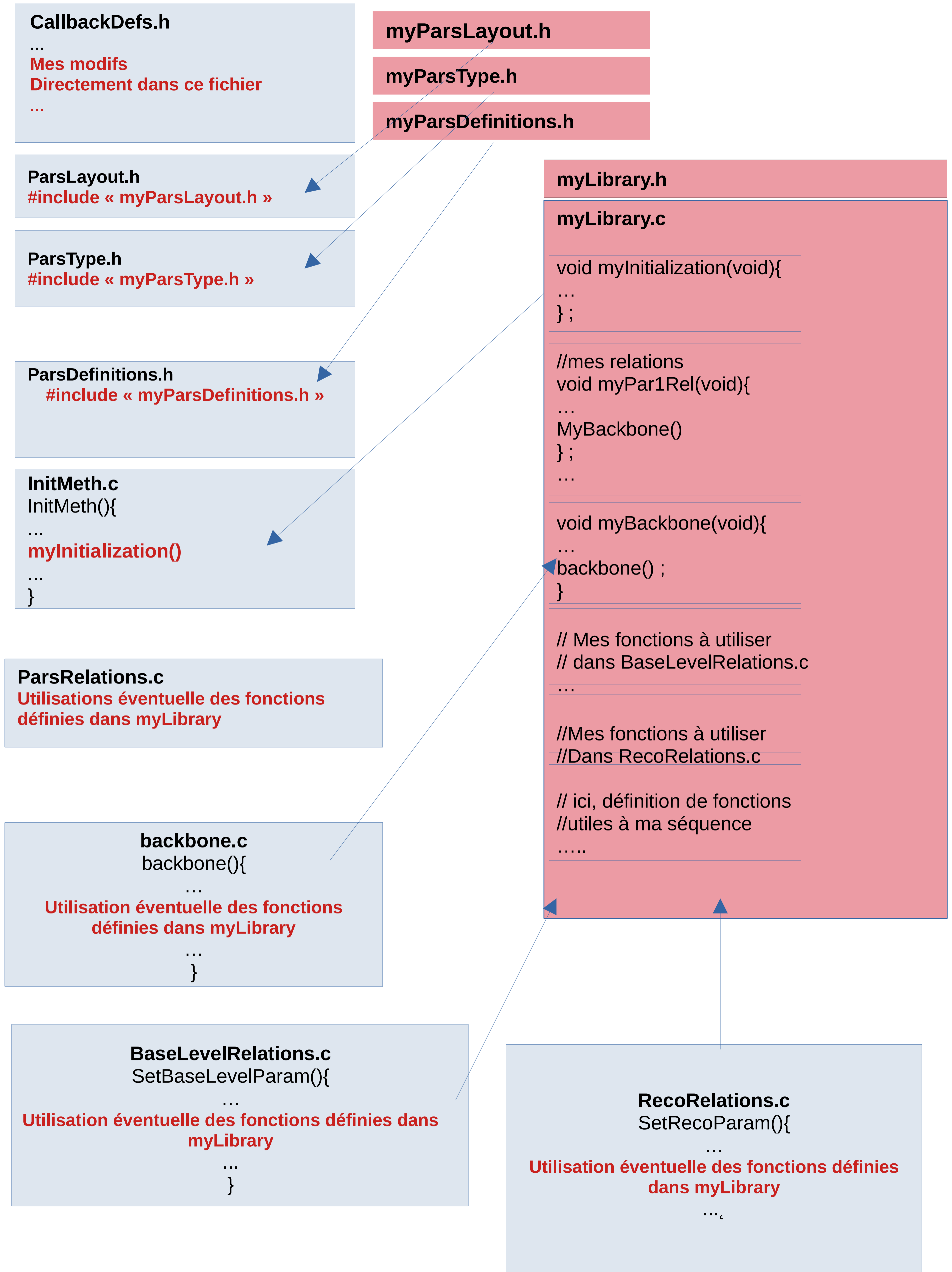
if « trignl1 » goto start

Partie interface utilisateur (langage C)

Structure globale d'une méthode



Structure de modification conseillée pour une méthode



Création et visualisation de variables dans l’interface

Pour utiliser/modifier les propriétés des paramètres, on utilise les fonctions :

```
void ParxRelsParRelations(const char *name, YesNo ForceDefault)
```

Lance la relation associée à ce paramètre.

Si ForceDefault =Yes c’est la relation par défaut qui est utilisée

Si ForceDefault=No c’est la relation éventuellement redirigée par CallbackDef.h qui sera lancée.

« setters » :

```
void ParxRelsMakeEditable (const char *name)
void ParxRelsMakeNonEditable (const char *name)
void ParxRelsShowInEditor(const char * name)
void ParxRelsHideInEditor(const char * name)
void ParxRelsShowInFile(const char * name)
void ParxRelsHideInFile(const char * name)
void PARX_change_dims(const char *name, int size1, …)
```

Change la (multi-)dimension du paramètre

« getters » :

```
int ParxRelsParHasValue(const char *name)
```

renvoit 1 si le paramètre a une valeur et 0 sinon

```
int ParxRelsVisibleForEdit(const char * name)
int ParxRelsParIsEditable(const char * name)
int ParxRelsVisibleInFile(const char * name)
unsigned int PARX_get_dim(const char *name, int dim)
```

renvoi la dimension du parametre

Le fichier parsDefinition.h

Pour créer une variable dans l’interface utilisateur il faut d’abord définir le type de cette variable dans le fichier **parsDefinition.h** ou plutôt dans un fichier **myParsDef.h** que vous rajoutez comme ***include*** dans le fichier **parDefinition.h**.

C’est à cet endroit que sont définies les variables et paramètres globaux de votre method.

Il existe un certain nombre de type de variables prédéfinis :

Si vous définissez la taille d'un pointeur elle sera alors fixe :

```
int parameter mypar[5];
```

Si vous laissez cette valeur vide, sa taille pourra être modifié dans la séquence en fonction de vos besoins :

```
int parameter mypar[];
```

int,int[], int[][], ... ,

double,double[], double[][], ...

char,char[],char[][],...

YesNo,

void

Ainsi que des types prédéfinis basés sur des *struct* et des *enum*.

exemple :

```
double GradValdouble ;  //variable standard en C
double parameter GradValmonpar ;   /* variable C dont le nom et la valeur peuvent être sauvés dans les fichiers décrivant les acquisitions (acqp, method, reco, …) */
double parameter
{
display_name "Gradient value";
format "%.2f";
units "mT/m";
relations GradValRelations;
} GradVal; * variable C pouvant aussi être visualisée dans l'interface utilisateur*
```

options :

toutes ces options sont optionnelles, cependant je vous encourage à remplir systématiquement celles qui sont pertinentes vis-à-vis du paramètre que vous voulez définir

display_name “string”;	Name displayed on the Parameter Card
short_description ”string”;	Text displayed in the “tool tip”
format ”%.3f”;	Format string as in printf function
minimum <value> outofrange origval;	Minimum value. When a lower value is entered,the editor returns to the original value.
minimum <value> outofrange nearestval;	Minimum value. When a lower value is entered, the editor sets it to minimum.
maximum <value> outofrange origval;	Maximum value. When a higher value is entered, the editor returns to the original value.
maximum <value> outofrange nearestval;	Maximum value. When a higher value is entered, the editor sets it to maximum.
widget slider;	A slider is available to modify the parameter value
units ”string”;	Definition of units presented in the editor

store true/false;	Parameter will (not) be stored in the method file
visible true/false;	Parameter will (not) be visible in the editor
editable true/false;	Parameter will (not) be editable (gray)
relations <function>;	Name of the relations function
style inline_array;	Elements displayed in one line
maxdim once <max_size>	Size limit for dynamic arrays

toutes ces options sont optionnelles, cependant je vous encourage à remplir systématiquement celles qui sont pertinentes vis-à-vis du paramètre que vous voulez définir

Les relations et le fichier parsRelations.c

Dans pratiquement tous les cas, vous devrez associer à votre paramètre une relation c'est-à-dire une fonction qui sera appelée à chaque fois que votre paramètre sera modifié, et ceci que ce soit directement dans l'interface par l'utilisateur ou bien indirectement par une autre fonction.

Ces fonctions sont rituellement définies dans le fichier parsRelations.c, là encore je vous encourage vivement à les placer dans un fichier du style myparsRelations.c et à inclure ce fichier dans parsRelations.c

Le but de ces fonctions/relations est de décrire l'impact de la modification de votre paramètre sur le reste de la séquence, c'est à vous de faire cet exercice car c'est vous seul qui savez quel paramètre impacte quel autre.

Le fichier ParsLayout.h

Par le passé, ce fichier permettait de déterminer quel paramètre était visualisé ou dans l'interface.

Maintenant ce rôle est tenu par le fichier .xml associé à chaque méthode.

Pourtant ce fichier joue encore un rôle assez obscur mais vital pour le bon fonctionnement d'une séquence dans la mesure ou il sert encore d'interface entre la partie visualisation et action de l'interface, c'est-à-dire :

Attention ce point est une cause récurrente de grosse perte de temps.
Pensez bien à le faire

Si un paramètre et sa relation ont été définis mais que ce paramètre n'apparaît pas directement ou comme faisant partit d'une classe incluse dans la classe « mère » MethodClass alors, quand on compilera la séquence, on ne créera pas de lien bilatéral entre la visualisation et les fonctions lancées.

En particulier, une valeur modifiée dans l'interface reviendra à sa valeur initiale quand on fera un aller-retour entre prescription/validation.

Il faut donc systématiquement vérifier que cela est bien fait :

```
……
extend pargroup
{
  PVM_EchoTime;
….
  FaExp;
  MonParametre ;
   ouBienMaClasseQuiRegroupeTousMesParamètres ;
  ...
  RecoveryDelay;
} MethodClass;
……
```

Fichier CallbackDefs.h

Il sert à modifier le comportement par défaut des relations associés aux paramètres prédéfinis dans le programme principal.

Exemple ;

```
relations PVM_NAverages LocalNArelations;
```

à partir de ce moment-là, lorsque la valeur du paramètre PVM_NAverages est changée, c'est la relation `void LocalNArelations(void)` que nous aurons définie qui sera lancée.

En plus de ce comportement associé à des **paramètres**, il est possible aussi de rediriger des **actions ou des familles de paramètres :**

Event	Function parameter
Change on the System Card (e.g. different coil operation mode)	PVM_SysConfigHandler
Start of the scan	PVM_AcqScanHandler
Start of the reconstruction	RecoUserUpdate
End of reconstruction (deriving Visu parameters)	VisuDerivePars
Start of an adjustment	PVM_AdjHandler
End of an adjustment	PVM_AdjResultHandler

Group	ImageGeometry
Members	Parameters describing image FOV, slice thickness, resolution and orientation of slice packages, numbers of slices etc. Most of them can be set graphically in the Geometry Editor.
Handler	PVM_ImageGeometryHandler
Initializer	STB_InitImageGeometry
Updater	STB_UpdateImageGeometry

Group	Nuclei
Members	Nuclei names and enumerations, gradient calibration constant, frequency parameters
Handler	PVM_NucleiHandler
Initializer	STB_InitNuclei
Updater	STB_UpdateNuclei

Group	Voxel_Geometry
Members	Parameters describing voxel geometry in localised spectroscopy: number of voxels, sizes, positions and angles.
Handler	PVM_VoxCallBack
Initializer	STB_InitVoxelGeometry
Updater	STB_UpdateVoxelGeometry

Group	Spectroscopy
Members	Parameters describing spectral dimensionality, resolution, and matrix size.
Handler	PVM_SpecHandler
Initializer	STB_InitSpectroscopy
Updater	STB_UpdateSpectroscopy

La liste exhaustive de ces groupes de paramètre se trouve dans le fichier

prog/include/methodClassDefs.h

Le fichier .xml

Attention :
D’habitude je conseille de créer des fichiers séparés et de les intégrer sous forme de #include.
Dans ce cadre précis il est déconseillé de le faire car le Makefile de la séquence n’est pas prévu pour cela et il gérera ensuite très mal un copyMethod que l’on pourrait vouloir faire par la suite.
i.e. le chemin allant vers le fichier xml que vous aurez défini sera invariant par copyMethod et pointera donc toujours sur un même fichier.

En suivant l’arborescence xml on peut créer un nouvel onglet qui intégrera tous nos paramètres ou encore les incorporer dans un onglet existant :

```
<parameterCard displayName="MonOnglet">
        <column>
                <parameter name="myParam_1"/>
        ...
                <parameter name="myParam_n"/>
                <textLine text=""/>
                <parameter name="myParam_n+1"/>
        ...
                <parameter name="myParam_n+x"/>
        </column>
        <column>
                <parameter name="myOtherParam_1"/>
        ...
                <parameter name="myOtherParam_n"/>
        </column>
</parameterCard>
```

Une fois ce fichier modifié il est fortement conseillé d’utiliser les utilitaires fournis par l’interface de programmation Bruker pour valider la syntaxe de ce fichier vis-à-vis du standard xml et ensuite le valider par rapport à son extension Bruker

Les Délais : Exemple ajout d'un délai

Les variables Bruker sont souvent définies en ms mais parfois en s ou µs. Pour éviter beaucoup de pb de debuggage, je vous conseille très fortement de nommer vos variables avec une dimension associée :

myVar_s

myVar_ms

myVar_us

DG_myLibrary.h

```
void DG_Delay_Relations (void) ;
void DG_initMeth(void) ;
void DG_ImpactSurEchoTime(void) ;
void DG_ImpactDesAutresParamètresSurDG_Delay(void) ;
……
```

DG_myLibrary.c

```
void DG_Delay_Relations (void){
DG_Delay = MAX_OF(DG_Delay, 1.0) ;
DG_Delay = MIN_OF(DG_Delay, 100.0) ;
backbone() ;
}

// cette fonction est à faire appeler dans la fonction
initMeth.c
void DG_initMeth(void) {
if (ParxRelsParHasValue("DG_Delay") == No)
DG_Delay=10.0;
}

// cette fonction est à faire appeler par la fonction
backbone()
void DG_ImpactSurEchoTime(void){
// Ici vos instructions pour faire en sorte
// que le temps d'écho tienne compte
// de votre délai
}

//cette fonction est aussi à faire appeler par le
backbone()
void DG_ImpactDesAutresParamètresSurDG_Delay(void){
//code décrivant l'impact des autres paramètres sur
//le délai que j'ai défini
}
```

DG_parsDefinition.h

```
double parameter{
display_name "Mon délai";
format "%.2f";
units "ms";
relations DG_Delay_Relations
} DG_Delay;
```

Dans le .PPG je rajoute les lignes
`myDelay = {$DG_Delay}`
et j'utilise ensuite
`myDelay`
dans le PPG

parsLayout.h
je rajoute dans `MethodClass` la ligne
`DG_Delay ;`

parsDefinition.h
`#include <DG_parsDefiniton.h>`

Dans le .xml je rajoute
<parameter name="`DG_Delay`"/>

method.h
`#include <DG_myLibrary.h>`

Makefile

```
OBJS        =       \
                    initMeth$(OBJEXT)  \
                    loadMeth$(OBJEXT)  \
                    backbone$(OBJEXT)  \
                    parsRelations$(OBJEXT)  \
                    BaseLevelRelations$(OBJEXT)  \
                    RecoRelations$(OBJEXT)\
                    DG_myLibrary$(OBJEXT)
```

Gestion des Interactions entre paramètres

Souvent changer la valeur d'un paramètre peut entraîner une cascade de changements et ajustements pour d'autres paramètres, ces comportements sont gérés par l'utilisateur de manière déterministe dans la relation liée à ce paramètre ou d'une manière plus flexible au niveau du *backbone*.

Pour que cette gestion du changement soit faite la plus intelligemment possible par le *backbone* il est nécessaire de pouvoir savoir quel paramètre a été changé.

Pour cela on utilise la fonction **UT_RelContext_ParName()**

parsDefinition.h:

```
double parameter
{
    relations A_Relation;
} A;
```

```
double parameter
{
    relations B_Relation;
} B;
```

parsRelations.c:

```
void A_Relation(void)
{
   // A rangechecking
   (…)
   backbone();
}
```

```
void B_Relation(void)
{
   // A rangechecking
   (…)
   backbone();
}
```

backbone.c:

```
if ( UT_RelContext_ParName() == "A" )
    <code that changes B>

if (UT_RelContext_ParName() == "B" )
    <code that changes A>
```

On peut en plus utiliser les paramètres suivants pour préciser le contexte :

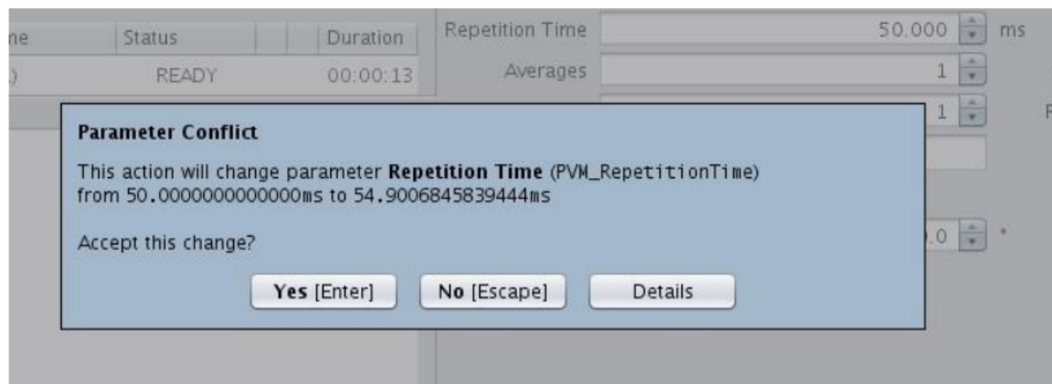
- UT_RelContext_ParOrigValue => Previous value of changed parameter
- UT_RelContext_ParOrigDim => Previous dimension of redimensioned array parameter
- UT_RelContext_ParStructNames => Name of changed element in struct parameter
- UT_RelContext_ParArrayIndices => Index of changed element in array parameter

De la même manière, on peut imaginer que plusieurs paramètres appellent la même relation qui va se comporter différemment suivant le paramètre qui l'a appelée, c'est particulièrement utile quand on crée plusieurs représentations d'un même paramètre comme l'amplitude d'un pulse qui peut être exprimée à la fois en W, dB, µT, Hz

Indication visuelle des conflits entre paramètres

Changer un paramètre peut sournoisement en impacter un autre, pour que l'utilisateur soit bien au courant de l'impact que peut avoir le changement d'un paramètre on peut utiliser le mécanisme de « conflict handling ». Il suffit de rajouter le paramètre dans le group *conficts* de *parsLayout.h*

```
// parameters that should be tested after any editing
conflicts
{
PVM_EchoTime;
PVM_RepetitionTime;
PVM_Fov;
PVM_SliceThick;
};
```



Programmer des ajustements

Un ajustement est une méthode qui peut s'auto-piloter.

c'est une méthode classique qui dispose en plus de fonctions lui permettant d'étudier le résultat de sa propre acquisition et éventuellement itérer les périodes d'acquisition/étude jusqu'à convergence.

L'ajustement peut être de deux type :

- Il peut utiliser la méthode qui le lance comme typiquement l'ajustement du receiver gain qui n'a de sens que s'il utilise la méthode qui le lance.
- Il peut utiliser une méthode autre que celle qui a besoin du résultat de l'ajustement comme c'est le cas pour le calcul de la 90° qui utilise la méthode *RefGain* quelle que soit la méthode qui lance cet ajustement.

Principe d'une méthode de type ajustement.

Quand une méthode de type ajustement est lancé, les acquisitions sont contrôlées par un pipeline spécifique, le pipeline ***Auto*** qui agit de la manière suivante :

- La séquence tourne en continue en mode GS
- **ET** chaque TR, les paramètres modifiés du côté de l'interface sont envoyés à la partie hardware.

Le point important est que chaque acquisition déclenche l'incrément d'un paramètre précis nommé *AutoCounter* dont la modification déclenche après chaque acquisition le lancement de la relation qui lui est associé.

le principe est donc de (re)définir ce paramètre **ET** sa relation en fonction de nos besoins :

l'instruction ci-dessous indique au pipeline Auto que c'est la variable ***myAutoCounter*** qu'il devra modifier à chaque acquisition.

```
strcpy(ACQ_SetupAutoName,"myAutoCounter");
```

Cette redirection de paramètre implique aussi qu'à chaque modification de ce paramètre (chaque acquisition) la relation ***myAutoCounterRel*** (qui lui est associée lors de sa définition dans le fichier parsDefinition.h) sera automatiquement lancée :

```
int parameter
{
display_name "Auto Counter";
relations myAutoCounterRel;
} myAutoCounter;
```

À notre charge de coder dans la méthode, la fonction/relation ***myAutoCounterRel*** qui incorporera les mécanismes de capture de donnée acquises (cf pipelines) ainsi que les traitements et les aspects décisionnels rendant cette fonction « décideuse ».

On utilise aussi le paramètre myAutoCounter pour interagir avec le pipeline « Auto » :

- Quand l'ajustement est initialisé, l'*autocounter* doit être mis à 1 ce qui déclenchera pour la première fois de l'ajustement la relation ***myAutoCounterRel***. Si l'*autocounter* n'est pas mis à 1 pour cette initialisation, le pipeline considérera qu'il y a eu un problème et ne démarrera pas.
- Pendant l'ajustement, l'*autocounter* est automatiquement incrémenté à chaque acquisition ce qui déclenche à chaque fois la relation ***myAutoCounterRel.***
- Quand ***myAutoCounterRel*** est content du résultat, on lui fait mettre l'*autocounter* à 0 ce que le pipeline « Auto » interprétera comme : mission accomplie. Quand le pipeline « Auto » détecte le succès de l'ajustement il s'arrête et lance ensuite automatiquement la relation ***PVM_AdjResultHandler*** il est logique d'utiliser le fichier callbackdef.h pour rediriger cette relation vers le *backbone*, ainsi, les paramètres modifiés par l'ajustement seront injectés et mis en cohérence dans la méthode qui a eu besoin de l'ajustement.
- Si ***myAutoCounterRel*** ne converge pas ou détecte un pb on lui fait mettre l'*autocounter* à -1 ce que le pipeline « Auto » interprète comme : pb, on arrête avec un éventuel message d'erreur envoyé au GUI.

Fabrication d'un ajustement utilisant des données persistantes (type GOP)

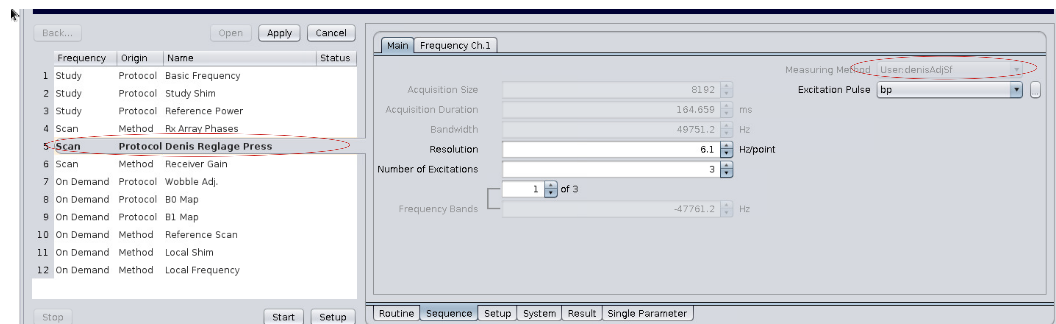
Dans le cas évoqué plus haut, une fois l'ajustement terminé, les paramètres en RAM sont ajustés, mais il ne reste aucune trace pérenne de l'ajustement.

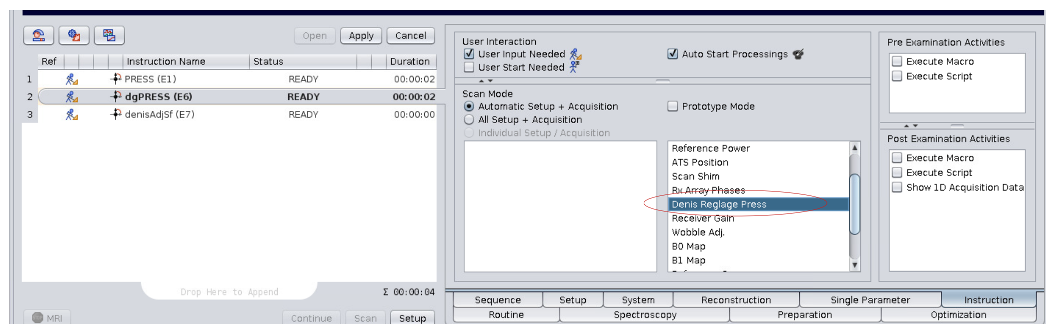
Parfois on veut aussi que le résultat de l'ajustement soit une « vraie » acquisition qui puisse être analysée/réutilisée tout au long de l'expérience, typiquement une carte B0 (fieldmap) ou B1, la mesure de trajectoires ou une carte de déformation quelconque qui pourra être utilisé par d'autres séquences ou même en post processing déporté.

Dans ce cas-là, le mécanisme ***ACQ_SetupAutoName*** n'est pas utilisé et les données sont enregistrées comme pour un scan ordinaire. Cela se fait en lançant un appel de la fonction ***PvSysManRequestNewExpno()***.

Pour récupérer le chemin d’accès à ces données, un paramètre de type ***AdjProcnoResultType*** doit être défini dans la méthode et inclus dans la liste des ajustements. Si l’ajustement est un succès, le *path* sera renvoyé dans ce paramètre et peut être récupéré sous forme de string en utilisant la fonction ***PvAdjManProcnoResultPath()***.

Requête d’ajustements par une méthode





Une fois qu’un ajustement a été fabriqué, il faut que la méthode demandeuse puisse y accéder et adapter l’ordre ou la manière dont il est lancé. Ainsi on peut voir dans la figure ci-dessus que les ajustements peuvent être :

- *On Demand*
- automatiquement lancés à chaque nouveau *Scan*
- automatiquement lancés à chaque nouvelle *Study.*

On remarque aussi que le *Receiver Gain, reference Scan* et *Local Frequency* sont (évidemment réalisés en lançant la *Method* qui les appelle) Quand le reste des ajustements visibles sont liés à des *Protocol* (méthodes externes).

ceci est décidé par les valeurs passées à la fonction ***PTB_AppendAdjustment***.

Dans l’exemple ayant donné les figure ci-dessus, j’ai placé dans le *backbone* l’instruction :

```
PTB_AppendAdjustment("DenisReglage",            // chp 1
                     "Denis Reglage Press",         // chp 2
                     "BlaBla Denis Reglage Press",  // chp 3
                     per_scan,                      // chp 4
                     "denisAdjSf");                 // chp 5
```

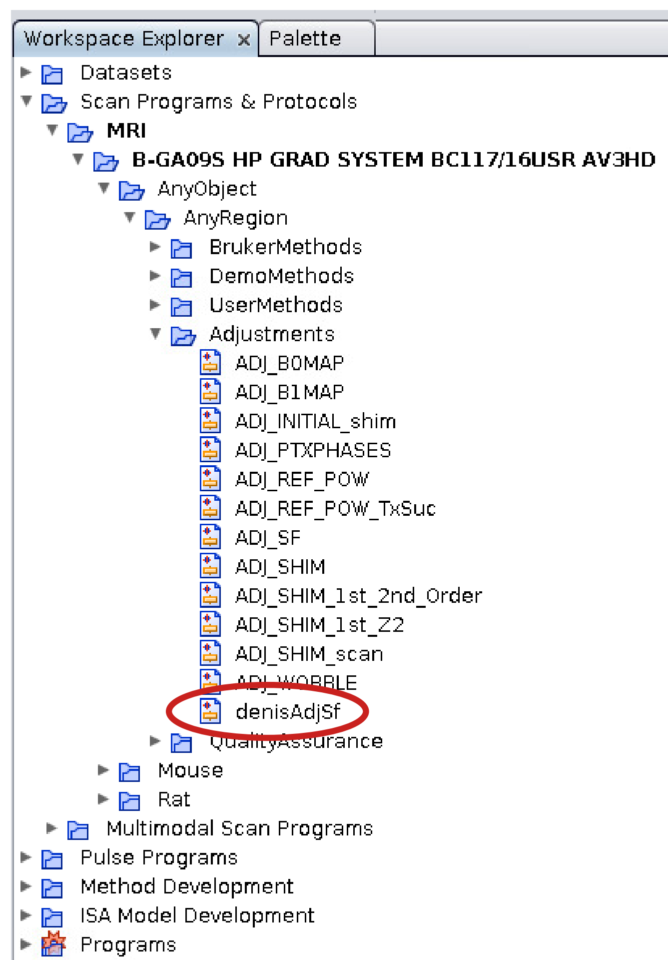
chp 1 : contient le nom servant à la méthode pour référencer en interne cet ajustement précis

chp 2 : contient le texte affiché décrivant la fonction de l'ajustement

chp 3 : contient des explications plus précises s'affichant dans une bulle d'aide

chp 4 : sert à préciser si l'ajustement est à effectuer sur demande/à chaque nouveau scan/à chaque nouvelle study

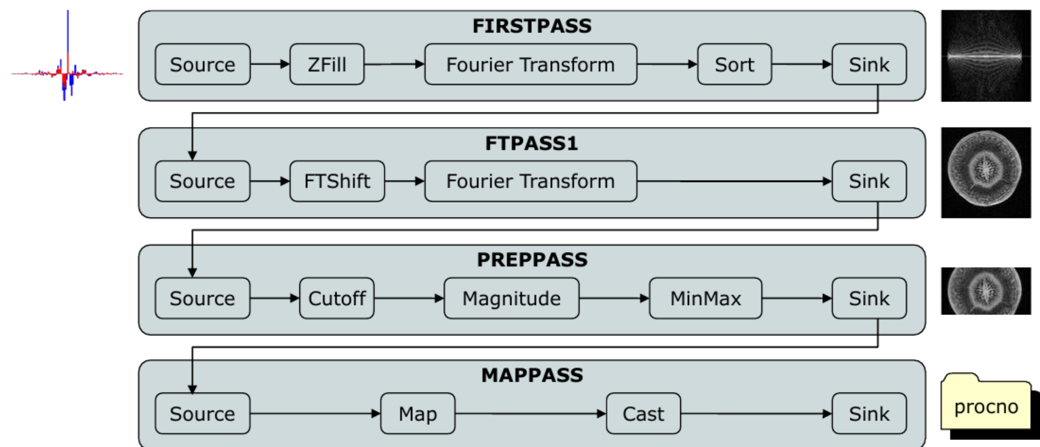
chp 5 : contient le nom de la méthode servant pour l'ajustement, si ce champ est vide, la méthode s'auto-utilise. **Si ce champ n'est pas vide il doit contenir le nom d'un protocole => une méthode qui a été sauvée comme protocole dans le « répertoire » ajustment :**



Reconstruction

La reconstruction du signal est orchestré par la notion de *passes* distinctes.

Chaque *passe* doit décrire une série d'instructions répétitives, déclenchées par des évènements précis exemple :



FIRSTPASS : Chaque fois que cette *passe* dispose d'une ligne de l'espace des k complète (array 1D), cette ligne subie alors un ZEROFILLING et une FFT et elle est ensuite triée pour être placée au bon endroit d'une matrice 2D décrivant l'espace des k en 2D.

FTPASS1 : quand la matrice 2D est remplie, on fait un FFTSHIFT et une autre FFT suivant l'autre dimension.

PREPASS : l'image 2D complexe retaillée, est transformée en image de magnitude et on en extrait les valeurs min et max

MAPPASS : l'image est « mappé » sur 16bits en fonction des valeurs min et max trouvées et elle est envoyé en écriture sur le HDD

Ces morceaux de pipelines sont caractérisés par un connecteur d'entrée (Source), un connecteur de sortie (Sink) et tous les processus qui sont insérés entre ces 2 étapes.

Dump du schéma du pipeline existant

De la même manière que l'on ne programme pas des méthodes « from scratch », on va en général se baser sur le pipeline de reconstruction existant déjà pour la séquence et on va le modifier en fonction de nos besoins.

Mais là encore, de la même manière que les instructions ne sont chargées sur le séquenceur que lorsque l'étape de prescription est totalement terminée, les informations sur la reconstruction ne sont chargées dans le système de reconstruction que lorsque le système atteint le status « **runtime** » (cf CallbackDef.h **events**). Cet **event** « **runtime** » déclenche le setup de toute la partie reconstruction en lançant la **relation** associée au paramètre ***RecoUserUpdate*.**

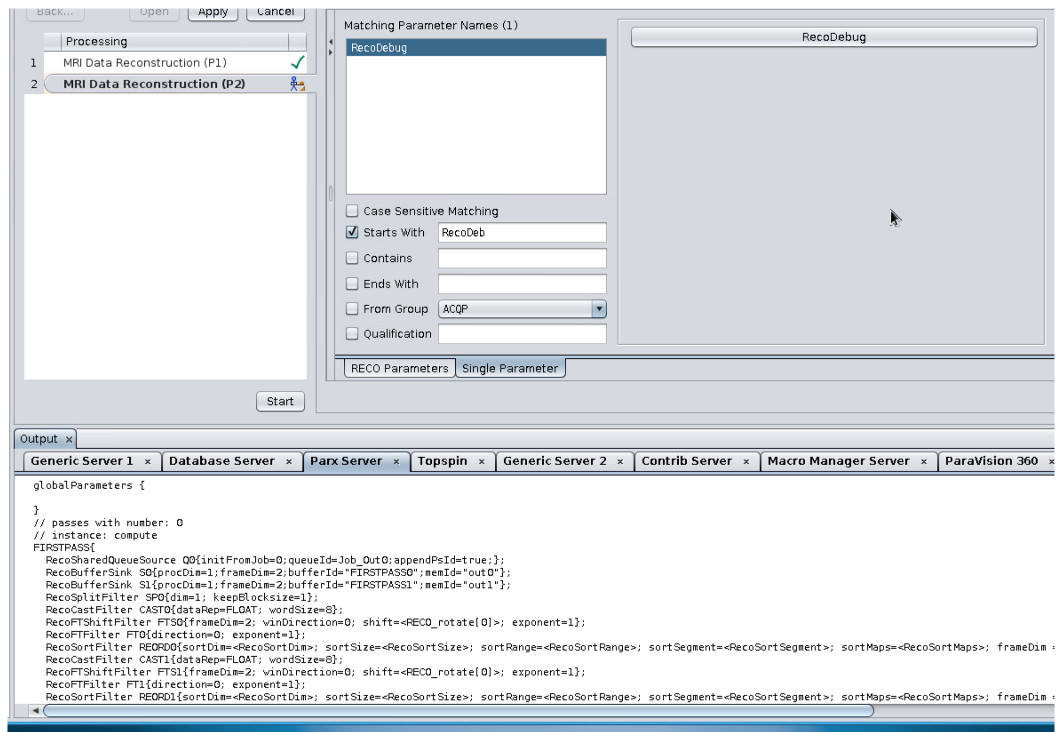
Cette relation est un pipeline générique de reconstruction 3D qui est prédéfini dans l'espace mémoire du programme principal et donc, qui nous est interdit d'accès. Heureusement <u>nous pouvons utiliser les mécanismes de redirection de relations utilisant CallbackDef.h pour rediriger cette relation vers une autre relation que nous pourrons modifier</u>.

Attention : lancer manuellement RecoUserUpdate ne permettent de « dumper » que la fonction <u>initiale</u> liée au paramètre RecoUserUpdate.
Si vous partez d'une séquence dans laquelle vous voyez dans CallbackDef.h que cette relation a <u>déjà</u> été redirigée vers une autre fonction, alors, vous devrez vous contenter de scruter le code de cette dernière fonction pour voir ce qu'elle fait et y insérer votre code aux endroits judicieux

La manière la plus simple pour observer le schéma du PIPELINE est de lancer manuellement la relation associée au paramètre ***RecoUserUpdate*** en utilisant aussi le mode mis en place par le paramètre ***RecoDebug***

Pour cela il suffit d'aller dans l'interface et d'utiliser le « Single Parameter Editor » pour modifier ces paramètres de manière à lancer leurs relations.

L'option ***RecoDebug*** a pour effet de *dumper* toute la structure du pipeline dans la fenêtre parxServer :



English/pvman/html/developer/RecoPublicStages.html

Intéressons-nous à ce qui se passe pendant FIRSTPASS

```
FIRSTPASS{
  RecoSharedQueueSource Q0{initFromJob=0;queueId=Job_Out0;appendPsId=true;};
  RecoBufferSink S0{procDim=1;frameDim=2;bufferId="FIRSTPASS0";memId="out0"};
  RecoBufferSink S1{procDim=1;frameDim=2;bufferId="FIRSTPASS1";memId="out1"};
  RecoSplitFilter SP0{dim=1; keepBlocksize=1};
 ...
  Q0->SP0;
  SP0->CAST0;
  SP0→CAST1;
...
}
// instance: input
FIRSTPASS{
  RecoFileSource FS0{initFromRawJob=0;procDim=1;numChan=2;contData=true;};
  RecoSharedQueueSink JQ0{queueId=Job_Out0;appendPsId=true;initQueue=true;};
  FS0->JQ0;
}
// instance: job0
FIRSTPASS{
  RecoSharedQueueSource
Q0{queueId=Job_In0;initQueue=true;appendPsId=true;dim=2;procDim=1;sizes={256,71};nr=1;dataRep=SIGNED;baseF
ield=COMPLEX;wordSize=4};
  RecoSharedQueueSink QS{queueId=Job_Out0;initQueue=true;appendPsId=true};
  RecoFileSink FS{jobNdx=0;writeAcqPars=true;filename="/opt/nmrdata/PV-360.2.0.pl.1/data/denis/
20201007_141920_fantomeresolution_fantomeresolution_1_1/21/rawdata.job0"};
  RecoProgressMonitorFilter PM{reportScanProgress="1/1"};
  RecoCastFilter CastToFloatMain{dataRep=FLOAT;wordSize=8};
```

```
  RecoAcqOutFilter
Av{displayEachAccumulation=1;displayEachScan=0;displayEachPeStep=0;cnt=1;ndx=0;procnoPath="/opt/nmrdata/PV-360.2.0.pl.1/data/denis/20201007_141920_fantomeresolution_fantomeresolution_1_1/21/pdata/1/";displayObjects=71;modalityType=1;innerObjects=1;outerObjects=1;average=1;};
  RecoCastFilter CastToIntMain{dataRep=SIGNED;wordSize=4};
  RecoTeeFilter TeeStorage;

  Q0->PM;
  PM->CastToFloatMain;
  CastToFloatMain->Av;
  Av->CastToIntMain;
  CastToIntMain->TeeStorage;
  TeeStorage->QS;
  TeeStorage->FS;
}
```

- Nous remarquons tout d'abord qu'il y a 3 FIRSTPASS qui vont donc se faire en parallèle :
    - La première effectue des opérations standards sur les donnés (filtrage, sorting, fft-1D).
    - La seconde semble lire un fichier associé à la manière dont est codée l'acquisition de données brutes.
    - La troisième gère la sauvegarde des données brutes dans le fichier rawdata.job0

- Nous remarquons aussi que ces FIRSTPASS sont associées à un même job qui est dans ce cas-là le job 0 :   initFromJob=0
  Donc si nous utilisons plusieurs jobs d'acquisition (ex : navigator + echo) il faudra construire un mécanisme de PASS pour chacun des jobs.

- Nous remarquons enfin que l'acquisition faite lors du job0 est ensuite décomposée en 2 ce qui correspond au nombre de canaux d'acquisition utilisés.
  Il faudra donc tenir compte séparément du nombre de canaux utilisés lorsque l'on modifie un pipeline.

Redirection et modification du PIPELINE

Comme nous venons de le voir plus haut, le PIPELINE générique de reconstruction 3D cartésien multicanaux est déjà relativement complexe. Nous allons donc nous contenter d'en reprendre la structure globale et d'y insérer d'autres actions ou d'en supprimer d'autres.

Nous partirons donc systématiquement du *dump* du PIPELINE pour nous informer sur ces éléments et savoir lesquels enlever ou autour desquels rajouter des actions intermédiaires.

Donc première étape, rediriger la relation associée au paramètre ***RecoUserUpdate*** :

- Dans **callbackDefs.h**

  relations RecoUserUpdate myRecoMode ;

- Dans **RecoRelations.c**

```
void myRecoMode()
{
/* we first launch the creation of the default
relation for standard PIPELINE */
ParxRelsParRelations("RecoUserUpdate", Yes);

if (RecoUserUpdate==No ||
ACQ_scan_type==Setup_Experiment)
return;
/* manipulate reco PIPELINE here */
...
}
```

Pour cela on utilise les fonctions ci-dessous dont la doc se trouve dans **English/pvman/html/developer/d2/d9e/group__PvOvlTools.html**

RecoNrActiveReceivers

RecoComputeAddStage

RecoComputeConnectStages

RecoComputeDisconnectStages

RecoComputeAppendStage

RecoComputeInsertStage

RecoComputeRemoveStage

Attention : Parfois, si la method n'a pas eu a utiliser ces fonctions, le makefile n'a pas été écrit pour « linker » en utilisant ces librairies => on a des erreurs du genre RecoRelations.c ...undefined reference to 'RecoComputeAppendStage'

Pour résoudre ce pb il faut rajouter dans le makefile dans LIBS = … $(SHLIBDIR)/LibPvOvlTools$(SHARELIBEXT) ...

(Obtenu en faisant un kdiff3 entre deux makefile de 2 sequences dont par exemple la ISIS)

Exemple d'insertion :

```
RecoComputeInsertStage(
"FIRSTPASS",            //PASS we want to modify
0,                      // Channel touched
"FT",                   // our stage is inserted before the one named FT
"RecoFTShiftFilter",    // we insert this function
"FTSQ",                 // This stage will be named FTSQ
"shift=0.25"            // The function RecoFTShiftFilter take shift=0.25
                        // as argument
);
```

En n'oubliant pas de faire la même opération sur tous les canaux :

```
void myRecoMode(void)
{
/* default relation for standard network */
ParxRelsParRelations("RecoUserUpdate", Yes);
if (RecoUserUpdate == No || ACQ_scan_type == Setup_Experiment)
return;
/* remove original FOV/2 Fourier shift filter
and replace it by a 1/4 Fourier shift */
for (int chan = 0; chan < RecoNrActiveReceivers(); chan++)
{
// removes FTS0,FTS1,FTS… stages in FIRSTPASS
RecoComputeRemoveStage("FIRSTPASS",chan,"FTS");
// insert FTSQ0,FTSQ1, … just before FT0, FT1, ...
RecoComputeInsertStage("FIRSTPASS",chan,"FT","RecoFTShiftFilter",
"FTSQ","shift=0.25");
}
}
```

Les fonctions utilisables comme étape d'une PASS sont disponibles dans la doc html :
English/pvman/html/developer/RecoPublicStages.html

Récupération « en vol » des données et traitement dédié

On est souvent amené à construire soi-même un traitement n'utilisant pas les fonctions standard ci-dessus.
Il est possible d'utiliser 2 fonctions pour nous simplifier la vie :

RecoSystemFilter

Cette commande permet de copier les donnés dans un fichier d'échange et de faire lancer une commande par l'OS (linux) (sur ce fichier d'échange :

```
RecoComputeAppendStage("PREPPASS", 0, "G", "RecoSystemFilter",
"OctaveProc", "cmd=\"/tmp/PvCallOctave.sh
<PROCNO>/infile\";exchangeFile=\"infile\";");
```

Attention :

Il est important de noter que ces manips sont faites ici seulement pour les données du canal 0

Si vous voulez faire la même chose pour les autres canaux en parallèle il faut créer « manuellement »les scripts, fichiers, paramètres associés.

RecoMethodFilter

Cette fonction permet, par un mécanisme de mémoire partagée, de faire pointer un pointeur de type double C standard vers des données acquises.

Dans **parsDefinition.h**

```
int parameter PvmFilterCnt;
int parameter {
   relations bufRelation;
 } PvmFilterBuffer;
```

Dans **RecoRelations.c**

```
RecoMethodFilter MyF{cntParameter=<PvmFilterCnt>;
                      bufParameter=<PvmFilterBuffer>};
```

ou bien :

```
RecoComputeAppendStage(
"PREPPASS",                              //PASS
0,                                       //Canal
"BlaBla",                                // à placer avant BlaBla
"RecoMethodFilter",                      // fonction à lancer
"MyF",                                   // nom de mon stage
"cntParameter=<PvmFilterCnt>",           //paramètre taille
"bufParameter=<PvmFilterBuffer>");       //paramètre ptr vers données
```

Dans **RecoRelations.c** aussi :

```
void bufRelation(){
RecoGlobalMemory MyF(PvmFilterBuffer);
double* data = (double*) MyF.getAddress();//pointeur vers les
                                          //données
```

```
int cnt  = PvmFilterCnt / sizeof(double); //nombre de points

for (int i = 0; i < cnt; i++) data[i] += 2.0;
}
```

voir aussi des fonctions comme ***ATB_GetRecoDataPoints*** qui insère les étapes de ***RecoTeeFilter***, ***RecoCastFilter***, et ***RecoParamterSink*** après une étape existante spécifiée. Le ***RecoParameterSink*** écrit les points de données dans le paramètre ***PARX PVM_RecoDataPoints***.

Interaction de RecoMethodFilter avec d'autres langages/ordinateurs

Quand vous avez programmé la fonction qui accède aux données par le pointeur (dans RecoRelations.c), rien ne vous empêche d'utiliser les mécanismes Unix de communication inter process (IPC) pour faire traiter vos données par un autre programme (Python, Matlab, ….) situé sur la console ou même sur un autre ordinateur du réseau qui renverra ensuite le résultat à votre fonction initiale Paravision.

On peut utiliser les mécanismes de *memory map* (cf mmap) mais je vous conseille d'utiliser plutôt les mécanismes mieux bordés de *sockets* (que ce soit intra ou inter ordinateurs).

Les différents types d'IPC que vous pouvez utiliser sont très bien décrit dans le bouquin (gratuit) inter-process_communication_in_linux.pdf

Je vous conseille aussi de définir la partie que vous avez écrite dans PV comme étant la partie ***cliente*** de votre socket (programmée forcément en C car codée dans votre méthode Paravision) et de définir la partie extérieure à PV comme la partie **serveur,** Partie programmée dans le langage qui fera le traitement déporté c.f. :

https://docs.python.org/3.8/library/socket.html pour python

et

https://fr.mathworks.com/help/instrument/communicate-using-tcpip-server-sockets.html pour Matlab.

Interaction de variables avec l 'extérieur

Dans le cadre par exemple d'ajustements, vous voudrez que votre programme extérieur puisse à la fois accéder aux données pour effectuer un traitement, mais aussi modifier des ***parameter*** dans un mécanisme de *feedback.*

Hint :

Utilisez la notion de struct pour qu’une seule relation socket puisse gérer en même temps tous les paramètres avec lesquels vous voullez intéragir.

Là encore, vous pouvez utiliser le mécanisme de socket mais en l’implémentant cette fois dans une/des relations de paramètres.

Pendant le déroulement de l’ajustement, à chaque fois que la relation ***myAutoCounterRel*** est lancée, faites lui aussi lancer la relation utilisant le socket.

Attention :

Dans le cadre d’ajustements il faut bien entendu que la durée entre l’envoi des données, et l’utilisation par PV des nouvelles valeurs des paramètres soit compatible avec le timing de votre séquence (qui est différent de TR). Si ce n’est pas le cas il faut programmer le socket du pipeline (celui qui envoi les données à traiter vers le serveur) à ne pas envoyer de nouvelles données vers le serveur tant que la réponse n’est pas arrivée, sinon, à cause de la bufferisation des données, le côté serveur fera des ajustements à partir de données qui ne tiennent pas encore compte des paramètres qu’il a fait modifier par PV.