

**NI660X SUPPORT LIBRARY****Diffusion :***Liste de diffusion*

Date	Rédacteur	Vérificateur	Approbateur	Modifications	Indice
	G.Abeillé				

Table des Matières

<b>1</b>	<b>Introduction .....</b>	<b>4</b>
<b>2</b>	<b>Présentation de la 6602.....</b>	<b>5</b>
<b>3</b>	<b>Principe d'utilisation.....</b>	<b>6</b>
<b>4</b>	<b>Configuration des modes .....</b>	<b>6</b>
<b>5</b>	<b>Acquisition Bufférisée.....</b>	<b>7</b>
5.1	Acquisition finie .....	7
5.2	Acquisition continue .....	7
<b>6</b>	<b>Génération d'impulsions .....</b>	<b>9</b>
6.1	Principe .....	9
6.2	Configuration.....	10
6.1	Continuous pulse train generation .....	12
6.2	Finite pulse train generation.....	13
6.3	Retriggerable pulse train generation.....	14
<b>7</b>	<b>Comptage d'événements.....</b>	<b>14</b>
7.1	Principe .....	14
7.2	Configuration.....	15
7.3	Simple event counting.....	15
7.4	Buffered event counting .....	16
<b>8</b>	<b>Mesure de position de moteur .....</b>	<b>19</b>
8.1	Principe .....	19
8.1.1	Encodeurs en quadrature .....	19
8.1.2	Encodeurs sur deux impulsions .....	20
8.2	Configuration.....	21
8.3	Simple position measurement.....	21
8.4	Buffered position measurement .....	21
<b>9</b>	<b>Routage interne .....</b>	<b>21</b>
<b>10</b>	<b>Gestion des erreurs.....</b>	<b>21</b>
10.1	Exceptions .....	21
10.2	Warnings.....	21
<b>11</b>	<b>Etats des opérations de comptage.....</b>	<b>21</b>
<b>12</b>	<b>Intégration dans un device TANGO.....</b>	<b>21</b>

12.1	Conversion des exceptions de NI660Xsl vers TANGO .....	21
12.2	Conversion des états de NI660Xsl vers TANGO .....	21
12.3	Exemple : « Continuous pulse generation » .....	21
12.3.1	init_device .....	21
12.3.2	Commandes.....	21
12.4	Exemple : acquisition bufferisée finie.....	21
12.4.1	Réception des données .....	21
12.4.2	init_device .....	21
12.4.3	read_attr_hardware.....	21
12.4.4	read_attr.....	21
12.4.5	Commandes.....	21
12.5	Exemple : acquisition bufferisée continue.....	21
12.5.1	Réception des données .....	21
12.5.2	Acquisition.....	21
12.5.3	init_device .....	21
12.5.4	read_attr_hardware.....	21
12.5.5	read_attr.....	21
12.5.6	Commandes.....	21
<b>13</b>	<b>Les annexes .....</b>	<b>21</b>

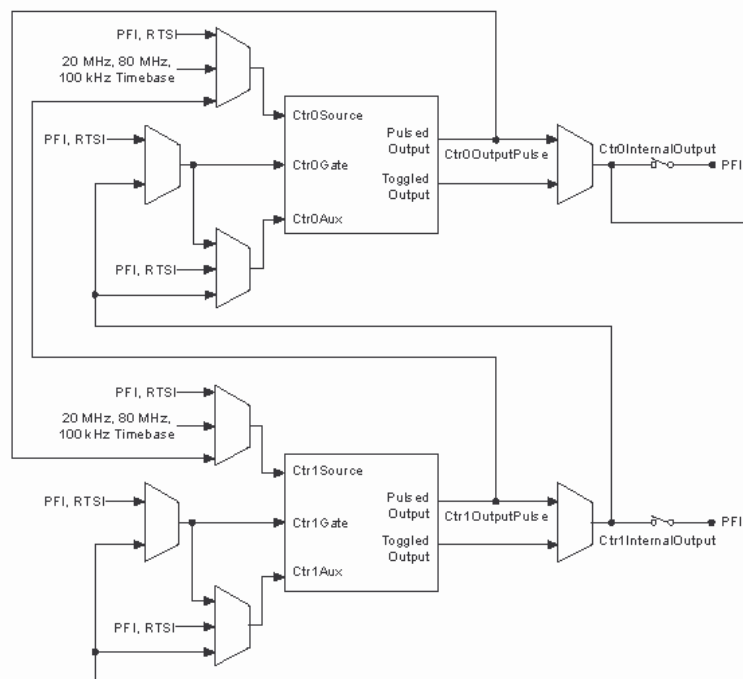
## 1 Introduction

Cette librairie, NI660XSL, a pour but premier d'intégrer la carte PXI6602 de chez National Instruments dans le système de contrôle de SOLEIL, TANGO. Elle supporte aussi les autres cartes de comptage 6601 et 6608 (mais celles-ci n'ont pas été testées). Cette librairie permet de masquer la complexité des drivers de la carte et propose une interface simplifiée permettant d'intégrer de façon plus rapide les principales fonctionnalités de la carte dans un device TANGO. Les fonctionnalités de la carte sont découpées en un certain nombre de « modes » qui correspondent à certains types d'applications. Les modes intégrés dans cette librairie sont la génération d'impulsions (*'Pulse Generation'*), le comptage d'événements (*'Event Counting'*) et la mesure de position de moteur (*'Position Measurement'*). Ces modes seront décrits un par un dans la suite du document. Bien que les diverses fonctionnalités proposées soient différentes, le principe d'utilisation de la carte sera toujours le même. Le protocole d'initialisation, configuration est toujours le même d'un mode à l'autre. Ce protocole sera décrit dans ce document.

NB : Ce document présente en détails comment implémenter une application avec cette librairie avec de nombreux exemples. Mais elle ne détaille pas toutes les méthodes et tous les paramètres. On se reportera à la documentation html fournie avec la librairie.

## 2 Présentation de la 6602

La 6602 est une carte de comptage comprenant 8 compteurs de 32 bits. L'horloge interne de la carte est de maximum 80 Mhz. Chaque compteur a comme entrées des voies appelées GATE, UP/DOWN et SOURCE. Ces voies peuvent être connectées de façon externe à la carte ou de façon interne en utilisant des fonctions logicielles de routage. Chaque compteur a une sortie, OUT. Les connections externes des signaux sont appelés PFI<0..39>. Il est possible de faire du routage entre signaux externes ou internes par logiciel. Le schéma suivant présente l'architecture de 2 compteurs :



La description exacte de la connectique et de toutes les fonctionnalités de la carte se trouvent dans le manuel fourni avec les cartes : *'6601/6602 User Manual'*.

Le driver utilisé pour développer cette librairie est NI-DAQmx (version NI-DAQ7.2). Pour le moment, ce driver ne fonctionne que sous Windows. L'accès du driver vers les connections physiques de la carte à travers les fonctions logicielles se fait de la façon suivante :

Ces connections appelées terminaux peuvent être internes à la carte (Gate, source, ...) ou accessible de l'extérieur par le connecteur de la 6602 (PFI<0..39>). Il faut d'abord accéder à la carte se trouvant dans un emplacement du châssis cPCI. Pour connaître l'identifiant de cette carte, on utilise le logiciel MAX (Measurement & Automation Explorer) fourni avec le driver de la carte. Dans l'onglet *Configuration >> périphériques et interfaces >> périphériques NI-DAQmx*, on retrouve les cartes présentes dans le châssis et fonctionnant avec ce driver ainsi que leurs noms. Par défaut, les cartes sont nommées *Dev1*, *Dev2*, ... Lorsque que l'on appellera les cartes à partir de la librairie, on utilisera donc la chaîne de caractères lui correspondant, par exemple *"Dev1"*.

Puis, on peut accéder aux terminaux de la carte, internes ou externes, présentés dans l'onglet *Routage du périphérique* lorsque l'on sélectionne la carte voulue dans MAX. Par exemple, la chaîne de caractères *"/Dev1/Ctr0Gate"* permet d'accéder à l'entrée GATE du compteur 0 de la carte *Dev1*. La chaîne *"/Dev1/PFI39"* permet d'accéder à la broche du connecteur PFI39 de la carte *Dev1*.

### 3 Principe d'utilisation

Chaque mode est représenté par une classe. Pour sélectionner un mode, il suffit d'instancier la classe voulue parmi les classes suivantes (ces classes sont décrites en détails dans la suite du document):

- `ni660Xsl::ContinuousPulseTrainGeneration`
- `ni660Xsl::FinitePulseTrainGeneration`
- `ni660Xsl::RetriggerablePulseTrainGeneration`
- `ni660Xsl::SimpleEventCounting`
- `ni660Xsl::SimplePositionMeasurement`

Pour les acquisitions bufférisées, il faut hériter sa propre classe (cf. §Acquisition Bufférisée) d'une des classes suivantes :

- `ni660Xsl::BufferedEventCounting`
- `ni660Xsl::BufferedPositionMeasurement`

Exemple :

```
ni660Xsl::ContinuousPulseTrainGeneration cont_train;
```

Le protocole d'utilisation de la carte est toujours le même pour tous les modes et comprend les étapes suivantes, dont l'ordre est à respecter :

- Initialisation du mode, crée un « handle » vers la carte :  
`cont_train.init();`
- Configuration du mode, configure la carte avec tous les paramètres qui auront été prédéfinis :  
`cont_train.configure();`
- Démarrage de l'acquisition :  
`cont_train.start();`
- Arrêt de l'acquisition :  
`cont_train.stop();`
- Release du mode, réinitialise le mode, la configuration est effacée :  
`cont_train.release();`

### 4 Configuration des modes

Chaque mode contient des « channels », qui permettent de le configurer. Chaque type de mode est configurable avec un ou plusieurs types de channel.

La génération d'impulsions est configurable avec les classes suivantes :

- `ni660Xsl::OutTimeChan`
- `ni660Xsl::OutFreqChan`
- `ni660Xsl::OutClockTicksChan`

Le comptage d'événements est configurable avec la classe suivante :

- `ni660Xsl::EventCountChan`

La mesure de position de moteur est configurable avec les classes suivantes :

- `ni660Xsl::LinearEncoderChan`
- `ni660Xsl::AngularEncoderChan`

Chacune de ces classes contient un certain nombre de champs auxquels il faut affecter des valeurs en fonction de l'acquisition que l'on souhaite faire.

Pour le comptage d'impulsions et la mesure de position de moteur, il est seulement possible d'ajouter un seul channel par mode. Par contre, il est possible de faire de la génération d'impulsions avec plusieurs voies pour une seule instance d'un mode.

## **5 Acquisition Bufférisée**

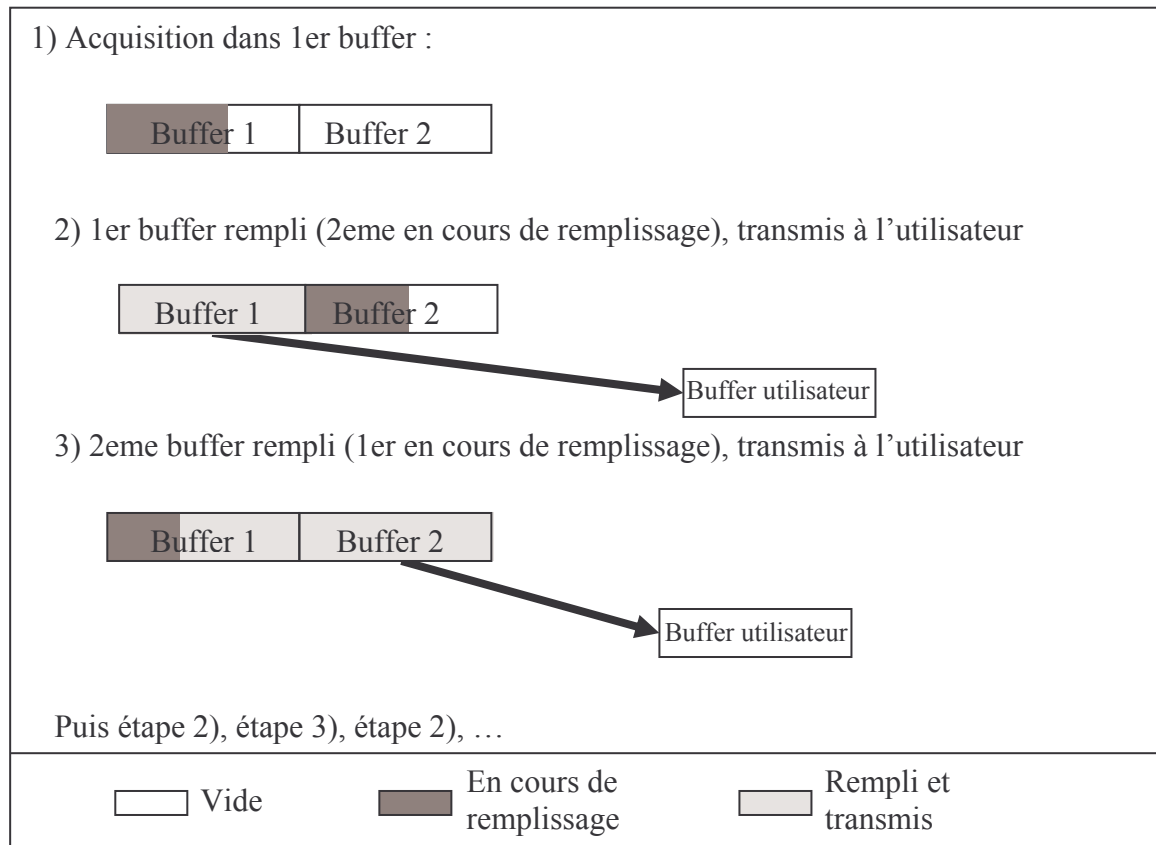
Pour les modes en entrée, comptage d'événements et position de moteur (la génération d'impulsions est un mode en sortie), on peut, soit faire de l'acquisition « simple », soit de l'acquisition « bufferisée ». En acquisition « simple » le compteur compte et l'utilisateur récupère la valeur courante du compteur quand il le souhaite. En acquisition « bufferisée », la valeur courante du compteur est sauvée dans un buffer à une fréquence fixe. Cela permet, par exemple de calculer la vitesse de comptage.

### **5.1 Acquisition finie**

Dans ce cas, on démarre l'acquisition, un buffer est rempli et l'acquisition s'arrête. Pour acquérir de nouvelles données, il faut redémarrer l'acquisition.

### **5.2 Acquisition continue**

Pour faire de l'acquisition continue, on fait ce que l'on appelle du « double-buffering ». Un double-buffer est considéré comme un buffer circulaire (donc infini) et est découpé en 2 parties. Chaque fois qu'un demi-buffer est rempli, le programme le renvoi à l'utilisateur. Le schéma suivant présente le principe du double-buffering :



Il peut arriver que le programme n'ait pas eu le temps de transmettre les données à l'utilisateur, la charge CPU étant trop importante. L'acquisition continuant de tourner, de nouvelles entrées sont stockées et les précédentes données sont effacées. On appelle ce phénomène un « overrun ».

Pour implémenter une acquisition bufferisée, il faut hériter une classe du mode bufferisé voulu. Par exemple si on veut faire du comptage d'événements bufferisé, on hérite de la classe `ni660Xsl::BufferedEventCounting`. Cette classe contient des méthodes virtuelles pures qu'il faut implémenter :

- `handle_data_lost`: cette méthode sera appelée par la NI660XSL quand un « overrun » se produit.
- `handle_timeout`: si NI660XSL ne reçoit pas de signal en entrée (horloge d'échantillonnage ou entrée du compteur absents) alors que l'acquisition est en cours, la librairie appelle cette fonction.
- `handle_raw_data`: méthode appelée quand un buffer utilisateur est plein et que l'on fait de l'acquisition avec la valeur du comptage. Le buffer, de type `ni660Xsl::InRawBuffer`, est passé en paramètre de cette fonction. Une fois ce buffer reçu, il ne faut pas oublier de le `delete`.
- `handle_scaled_data`: méthode appelée quand un buffer utilisateur est plein et que l'on fait de l'acquisition avec des valeurs scalées (la position du moteur en degrés par exemple). Le buffer, de type `ni660Xsl::InScaledBuffer`, est passé en paramètre de cette fonction. Une fois ce buffer reçu, il ne faut pas oublier de le `delete`.



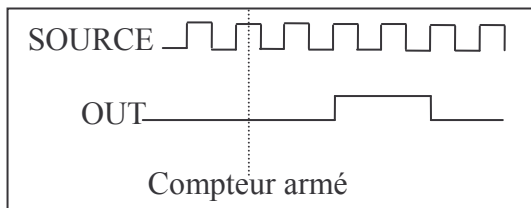
L'acquisition est paramétrable avec les fonctions suivantes :

- `set_sample_clock(clk_source, active_edge, max_rate)` : Paramètre l'horloge d'échantillonnage avec `clk_source`, la broche sur laquelle est branchée l'horloge (elle est forcément externe), `active_edge` est le front sur lequel l'horloge est active et `max_rate` est la fréquence maximum de l'horloge.
- `set_buffer_depth(nb_samples)` : Configure la taille du buffer en nombre d'échantillons renvoyé par les méthodes `handle_raw_data` et `handle_scaled_data`.
- `set_timing_mode(mode)` : Sélectionne le mode d'acquisition continu (`mode = ni::continuous`) ou le mode fini (`mode = ni::finite`). Si le mode fini est choisi, un seul buffer de la taille précisée par `set_buffer_depth` sera acquis.
- `set_timeout(timeout)` : Configure le temps, en secondes, à partir duquel on déclenche un timeout.
- `set_overrun_strategy(strategy)` : Permet de choisir le type de stratégie à adopter en cas d'overrun. Les valeurs possibles sont `ni::ignore`, `ni::trash`, `ni::notify`, `ni::restart`, `ni::abort` :
  - Si la stratégie choisie est « ignore », l'utilisateur reçoit les données mais il n'est pas notifié s'il y a eu des pertes de données.
  - Avec la stratégie « trash », les données sont détruites et l'utilisateur n'est pas notifié.
  - Si l'on choisi « notify », les données sont détruites et l'utilisateur est notifié à travers la fonction décrite ci-dessus, `handle_data_lost`.
  - Les données sont détruites, l'utilisateur est notifié par `handle_data_lost`, puis l'acquisition est redémarrée automatiquement avec la stratégie « restart ».
  - Les données sont détruites, l'utilisateur est notifié par `handle_data_lost`, puis l'acquisition est stoppée automatiquement avec la stratégie « abort ».

## 6 Génération d'impulsions

### 6.1 Principe

On génère des impulsions sur la sortie des compteurs, OUT. On paramètre la largeur des impulsions avec les fonctions de configuration. Le compteur se sert de la SOURCE comme référence pour calculer la largeur des impulsions. Ici, on génère une impulsion avec un délai :



Il est possible de faire plusieurs types de génération d'impulsions. Il y a d'abord la génération continue avec `ni660Xsl::ContinuousPulseTrainGeneration`, où l'on démarre la

génération d'impulsion avec la méthode `start()` et on l'arrête avec la méthode `stop()`. On peut aussi générer un train d'impulsions fini avec un nombre spécifique d'impulsions de 1 à n, c'est du `ni660Xsl::FinitePulseTrainGeneration`. Et enfin, on peut générer un train d'impulsions fini à chaque réception d'un trigger externe, c'est le `ni660Xsl::RetriggerablePulseTrainGeneration`.

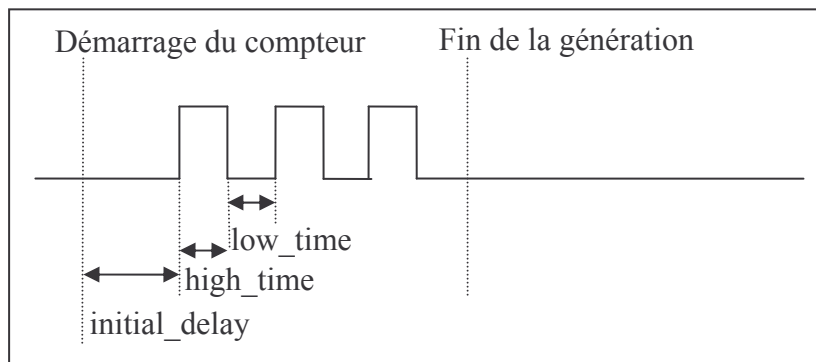
## 6.2 Configuration

Trois classes permettent de paramétrer la génération d'impulsions. On peut paramétrer la largeur des impulsions :

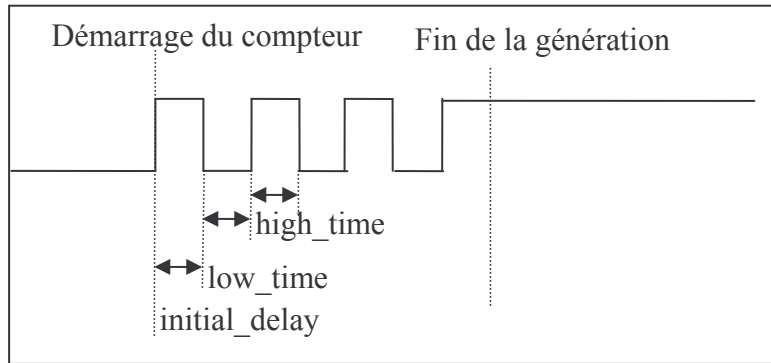
- en temps (secondes) avec `ni660Xsl::OutTimeChan`. L'horloge est forcément interne.
- en fréquence avec `ni660Xsl::OutFreqChan`. L'horloge est forcément interne.
- en coups d'horloge par rapport à l'horloge de référence avec `ni660Xsl::OutClockTicksChan`. L'horloge peut être interne (ex : `"/Dev1/80MhzTimebase"`) ou externe (ex : `"/Dev1/PFI12"`).

Ces classes contiennent un certain nombre de champs qui permettent de spécifier l'allure du train d'impulsions. Un train d'impulsion peut être généré avec un délai initial, ce délai est paramétré par le champ `initial_delay`. La polarité du signal généré est paramétrée par le champ appelé `idle_state`. Quand il est paramétré avec la valeur `ni::low`, la génération des impulsions est à l'état bas pendant le délai initial. A l'inverse, s'il est égal à `ni::high`, alors la génération d'impulsions commence à l'état haut.

Voici un exemple avec délai initial et `idle_state = ni::low`, et la largeur de l'impulsion spécifiée en temps avec `low_time` et `high_time`:



Voici un exemple avec `idle_state = ni::high`:



Exemples :

```
try{

ni660Xsl::OutTimeChan chan;

chan.chan_name = "/Dev1/ctr0"; //use ctr0 of board Dev1 for this channel
chan.idle_state = ni::low; //the signal is low at the beginning of generation
chan.initial_delay = 0.0; //the delay at the beginning of the generation in
//secs
chan.low_time = 0.0002; //the time when signal is low in secs
chan.high_time = 0.0001; //the time when the signal is high in secs
chan.output_terminal = "/Dev1/PFI24"; //optional: add another output to the
//default one

ni660Xsl::OutClockTicksChan chan2;

chan2.chan_name = "/Dev1/ctr3"; //use counter 3 of Dev1
chan2.clk_source = "/Dev1/PFI39"; //clock is on PFI39
chan2.idle_state = ni::low; //the signal is low at the beginning of generation
chan2.initial_delay = 4; // //the delay at the beginning of the generation in
//clk ticks
chan2.low_ticks = 2; // the time when signal is low in clk ticks
chan2.high_ticks = 2; // the time when the signal is high in clk ticks
chan2.output_terminal = "/Dev1/PFI20"; //optional: add another output to the
//default one

ni660Xsl::OutFreqChan chan3;

chan3.chan_name= "/Dev1/ctr4";
chan3.idle_state = ni::low;
chan3.initial_delay = 0.0; //secs
chan3.frequency = 10;
chan3.duty_cycle = 0.5;

}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

NB : Pour la gestion des exceptions cf. §Gestion des erreurs.

On ajoute ensuite les voies configurées au mode créé avec les méthodes suivantes :

```
- add_time_channel(ni660Xsl::OutTimeChan _chan)
```

- `add_frequency_channel(ni660Xsl::OutFreqChan _chan)`
- `add_clock_ticks_channel(ni660Xsl::OutClockTicksChan _chan)`

Il est possible d'ajouter plusieurs channels en mode génération d'impulsions.

Exemple, suite de l'exemple précédent :

```
try{
ni660Xsl::ContinuousPulseTrainGeneration cont_train;
cont_train.add_time_channel(chan) ;
cont_train.add_clock_ticks_channel(chan2) ;
cont_train.add_frequency_channel(chan3);
}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

## 6.1 Continuous pulse train generation

Pour générer un train d'impulsions continu, on procède de la façon suivante. On instancie d'abord la classe `ni660Xsl::ContinuousPulseTrainGeneration`. On configure et on ajoute ensuite les voies souhaitées. Puis on suit le protocole commun à tous les modes : `init`, `configure`, `start`, `stop`...

Il est possible d'utiliser une fonction d'attente `wait_finished(time)` où `time` est le temps en secondes avant d'arrêter la génération. Si par exemple, on veut attendre 5 secondes, on aura :

```
try{
ni660Xsl::ContinuousPulseTrainGeneration cont_train ;
ni660Xsl::OutTimeChan chan;

chan.chan_name = "/Dev1/ctr0";
chan.idle_state = ni::low;
chan.initial_delay = 0.0;
chan.low_time = 0.0002;
chan.high_time = 0.0001;

cont_train.add_time_channel(chan);
cont_train.init();
cont_train.configure();
cont_train.start()
cont_train.wait_finished(5);
cont_train.stop();
cont_train.release();

}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

En mode continu, il est aussi possible de changer les paramètres de largeur d'impulsion, lorsque la génération d'impulsions est en cours, pour faire, par exemple du PWM (Pulse Width Modulation). Pour cela, on utilise les méthodes suivantes :

- `change_freq_values(ni660Xsl::OutFreqChan _chan)` où l'on peut changer les champs `frequency` et `duty_cycle` de `_chan`
- `change_time_values(ni660Xsl::OutTimeChan _chan)` où l'on peut changer les champs `low_time` et `high_time` de `_chan`.
- `change_clock_ticks_values(ni660Xsl::OutClockTicksChan _chan)` où l'on peut changer les paramètres `low_ticks` et `high_ticks` de `_chan`.

On peut se servir d'un trigger pour faire du « start trigger » ou du « pause trigger ». Ce trigger est connecté sur la GATE du compteur, il n'est donc possible d'utiliser que l'un des deux triggers. Si on utilise le « start trigger », on le paramètre avec la fonction `set_start_trigger(trigger_source, active_edge)`. Dans l'exemple suivant, l'entrée du trigger est sur la pin PFI38, et il est actif sur front montant :

```
cont_train.set_start_trigger("/Dev1/PFI38", ni::rising_edge);
```

Le « pause trigger » permet de faire des pauses de la génération d'impulsions. On le peut configurer avec la fonction `set_pause_trigger(trigger_source, pause_when)`. Dans l'exemple suivant, le trigger vient de la pin 38 et la génération d'impulsions s'arrête quand celui-ci est au niveau bas :

```
cont_train.set_pause_trigger("/Dev1/PFI38", ni::low);
```

Ces fonctions de configuration de trigger doivent être appelées **avant** les fonction `init()` et `configure()`.

## 6.2 Finite pulse train generation

Pour générer un train d'impulsions fini, il faut instancier la classe `ni660Xsl::FinitePulseTrainGeneration`. On paramètre le nombre d'impulsions à générer avec la méthode `set_nb_pulses(nb_pulses)`. Si le nombre d'impulsions du train est égal à 1 alors la carte n'utilisera que **1 compteur**. Mais si le nombre est supérieur à 1 alors la carte va utiliser **2 compteurs**. Si on génère un train d'impulsions avec le compteur 0, `ctr0`, la carte va automatiquement utiliser le compteur `ctr1`. Les paires de compteurs seront toujours utilisé de la façon suivante : `ctr0` et `ctr1`, `ctr2` et `ctr3`, `ctr4` et `ctr5`, `ctr6` et `ctr7`. La fonction `wait_finished(-1)` permet d'attendre la fin de la génération du train.

On peut utiliser le « start trigger » pour démarrer la génération de ce train sur trigger externe avec la fonction `set_start_trigger(trigger_source, active_edge)`.

Voici un exemple :

```
try{
ni660Xsl::FinitePulseTrainGeneration finite_train ;
ni660Xsl::OutTimeChan chan;

chan.chan_name = "/Dev1/ctr0";
chan.idle_state = ni::low;
chan.initial_delay = 0.0;
chan.low_time = 0.0002;
chan.high_time = 0.0001;

finite_train.add_time_channel(chan) ;
finite_train.set_nb_pulses(10);
```

```
finite_train.set_start_trigger("/Dev1/PFI38", ni:: rising_edge);

finite_train.init();
finite_train.configure();
finite_train.start()
finite_train.wait_finished(-1); //wait for the generation to be finished
finite_train_stop();
finite_train.release();
} catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

### 6.3 Retriggerable pulse train generation

Il est possible de générer un train d'impulsions fini avec chaque trigger reçu. Pour cela il faut instancier la classe `ni660Xsl::RetriggerablePulseTrainGeneration`. Il faut paramétrer la source du trigger avec la fonction `set_trigger_source(trigger_source, active_edge)`. Le nombre de compteurs utilisé est le même que en « finite pulse generation ».

Exemple :

```
try{
ni660Xsl:: RetriggerablePulseTrainGeneration retrig_train ;
ni660Xsl::OutTimeChan chan;

chan.chan_name = "/Dev1/ctr0";
chan.idle_state = ni::low;
chan.initial_delay = 0.0;
chan.low_time = 0.0002;
chan.high_time = 0.0001;

retrig_train.add_time_channel(chan) ;

retrig_train.set_nb_pulses(10);

retrig_train.set_trigger_source("/Dev1/PFI38", ni:: rising_edge);

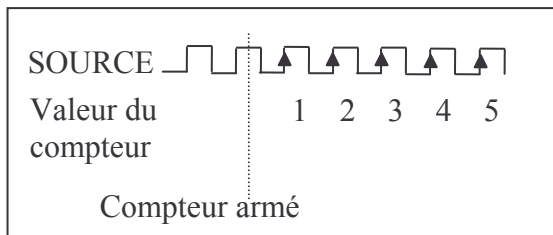
retrig_train.init();
retrig_train.configure();
retrig_train.start()
retrig_train.wait_finished(3); //wait 3 seconds.
retrig_train_stop();
retrig_train.release();
} catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

## 7 Comptage d'événements

### 7.1 Principe

Le compteur compte les événements (fronts montants) arrivants sur la SOURCE après que le compteur ait été armé. L'entrée UP/DOWN peut servir à contrôler la direction de

comptage. Quand UP/DOWN est au niveau haut, le compteur incrémente sur réception des événements. Quand UP/DOWN est au niveau bas, le compteur décrémente.



Il est possible de faire du « SimpleEventCounting ». Dans ce cas, l'utilisateur récupère la valeur du compteur quand il le souhaite. On peut aussi faire du « BufferedEventCounting », les données sont alors enregistrées dans un registre ou buffer avec une fréquence d'échantillonnage précise.

## 7.2 Configuration

On paramètre un channel pour le comptage d'événements avec la classe `ni6602Xsl::EventCountChan`. Il n'est possible d'ajouter que un channel à une opération de comptage (obligation du driver). Appeler deux fois la fonction `add_input_channel` renvoie une exception.

Exemple :

```
try{
ni6602Xsl::SimpleEventCounting count;
chan.chan_name = "/Dev1/ctr0"; //use ctr0 of board Dev1
chan.edge = ni::rising_edge; //detect events on rising edges
chan.initial_count = 0; //count from value 0
chan.count_direction = ni::count_up; //count up
count.add_input_channel(chan); //add « chan » to « count »
}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

## 7.3 Simple event counting

Pour faire du comptage d'événements simple, on instancie la classe `ni6602Xsl::SimpleEventCounting`, on configure la voie, puis on appelle les méthodes `init`, `configure`, `start`. On peut récupérer la valeur courante du compteur grâce à la méthode `get_current_raw_value()`.

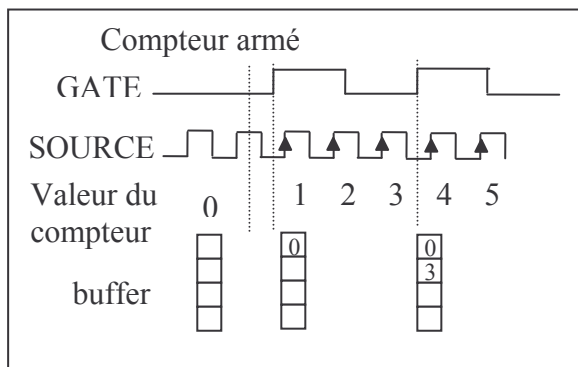
Il est possible d'utiliser un trigger pour faire soit du « start trigger » soit du « pause trigger ». On peut démarrer l'acquisition sur un « start trigger ». On le paramètre avec la méthode `set_start_trigger(trigger_source, active_edge)`. Pour faire du pause trigger (c'est-à-dire faire des pauses de l'acquisition sur réception d'un trigger externe), on le configure avec la méthode `set_pause_trigger(trigger_source, pause_when)`.

Exemple :

```
try{
ni6602Xsl::SimpleEventCounting count;
chan.chan_name = "/Dev1/ctr0";
chan.edge = ni::rising_edge;
chan.initial_count = 0;
chan.count_direction = ni::count_up;
count.add_input_channel(chan);
cout.set_pause_trigger("/Dev1/PFI23", ni::high);
count.init();
count.start();
count.wait_finished(2); //wait 2 seconds
count.stop();
int val = count.get_raw_data(); //val contains the current value of counter
}catch(const ni660Xsl::DAQException& e)
{/*manage exception*/}
```

## 7.4 Buffered event counting

Les données sont enregistrées dans un buffer à une fréquence donnée par la GATE :



Pour faire de l'acquisition bufferisée (cf. §Acquisition Bufferisée), on doit hériter sa propre classe de la classe `ni660Xsl::BufferedEventCounting`. L'exemple suivant montre une implémentation des méthodes virtuelles pures. Dans le cas du «event counting» les buffers passés dans méthodes `handle_scaled_data` ou `handle_raw_data` contiendront les même types de données, c'est-à-dire les valeurs du compteur. Il très important de ne pas oublier de **deleter les buffers**.

```
class MyMeasurement: public ni660Xsl::BufferedPositionMeasurement
{
public:
    MyMeasurement(void)
        :ni660Xsl::BufferedPositionMeasurement(),
        overrun(0),
        timeout(0)
    {};
    virtual ~MyMeasurement(void)
    {};
    void handle_data_lost(void)
    {
```



```

        overrun++;
        std::cout<<"overrun nb :"<<overrun<<std::endl;
    };
    void handle_timeout(void)
    {
        timeout++;
        std::cout<<"timeout nb :"<<timeout<<std::endl;
    };
    void handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long& _samples_read)
    {
        std::cout<<"raw"<<std::endl;
        for (int j =0; j <buffer->depth(); j = j+100)
            std::cout<<"val["<<j<<"] : "<<(*buffer)[j]<<std::endl;
        delete buffer;
    };
    void handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer, long&
                                _samples_read)
    {
        std::cout<<"scaled"<<std::endl;
        for (int j =0; j <buffer->depth(); j = j+100)
            std::cout<<"val["<<j<<"] : "<<(*buffer)[j]<<std::endl;
        delete buffer;
    };
private:
    int overrun;
    int timeout;

};

```

Il faut ensuite dans l'application principale procéder comme pour le « simple event counting » : configurer les voies, ... Il faut aussi, pour de l'acquisition bufferisée, configurer l'horloge d'échantillonnage avec la fonction `set_sample_clock`, le timeout avec `set_timeout`, la stratégie d'overrun avec `set_overrun_strategy`, et la profondeur du buffer avec `set_buffer_depth` et sélectionner le mode continu ou fini avec `set_timing_mode` (cf. §Acquisition Bufferisée).

Il n'y a pas, malheureusement, dans le driver, de mécanisme d'interruption qui permet de paramétrer une fonction de callback. Cette fonction pourrait alors appeler automatiquement les méthodes `handle_data`. Il faut donc pour recevoir les données dans les méthodes `handle_raw_data` ou `handle_scaled_data`, faire des boucles en appelant les méthodes `get_raw_buffer()` ou `get_scaled_buffer()`.

Exemple en mode fini :

```

try{
    MyMeasurement count;
    //-----config channel-----
    ni660Xsl::EventCountChan chan;

    chan.chan_name = "/Dev1/ctr0";
    chan.edge = ni::rising_edge;
    chan.initial_count = 0;
    chan.count_direction = ni::count_up;

    count.add_input_channel(chan);
    count.set_timeout(1.0); //seconds
    count.set_overrun_strategy(ni::abort);
    //-----config buffer-----
    count.set_buffer_depth(1000);
}

```

```
count.set_timing_mode(ni::finite);
//-----config sample clk (gate of counter)-----
count.set_sample_clock("/Dev1/PFI38", ni::rising_edge, 15000000);
//-----init-----
count.init();
//-----config hw-----
count.configure();
//-----start acq-----
count.start();
//-----get input data-----
count.get_raw_buffer(); //get data once because finite acquisition
//-----stop-----
count.stop();
//-----release hw-----
count.release();
}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

### Exemple en mode continu:

```
try{
    MyMeasurement count;
    //-----config channel-----
    ni660Xsl::EventCountChan chan;

    chan.chan_name = "/Dev1/ctr0";
    chan.edge = ni::rising_edge;
    chan.initial_count = 0;
    chan.count_direction = ni::count_up;

    count.add_input_channel(chan);
    count.set_timeout(1.0); //seconds
    count.set_overrun_strategy(ni::abort);
    //-----config buffer-----
    count.set_buffer_depth(1000);
    count.set_timing_mode(ni::continuous);
    //-----config sample clk (gate of counter)-----
    count.set_sample_clock("/Dev1/PFI38", ni::rising_edge, 15000000);
    //-----init-----
    count.init();
    //-----config hw-----
    count.configure();
    //-----start acq-----
    count.start();
    //-----get input data-----
    for (int i =0; i <100; i++)
    {
        //get data continuously because continuous mode
        count.get_raw_buffer();
    }
    //-----stop-----
    count.stop();
    //-----release hw-----
    count.release();
}catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

## 8 Mesure de position de moteur

### 8.1 Principe

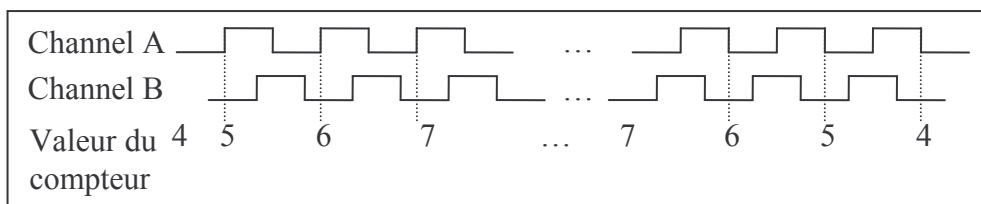
Pour faire de la mesure de position de moteur, on relie l'encodeur à la carte de comptage. Il est possible de faire de la mesure sur deux types d'encodeurs, les encodeurs en quadrature et les encodeurs sur deux impulsions.

#### 8.1.1 Encodeurs en quadrature

Un encodeur en quadrature peut avoir jusqu'à trois signaux : A, B, Z. Quand le channel A devance le channel B en quadrature, le compteur incrémente. Quand le channel B devance le channel A en quadrature, le compteur décrémente. Le taux de comptage par cycle dépend du type d'encodage : X1, X2, X4.

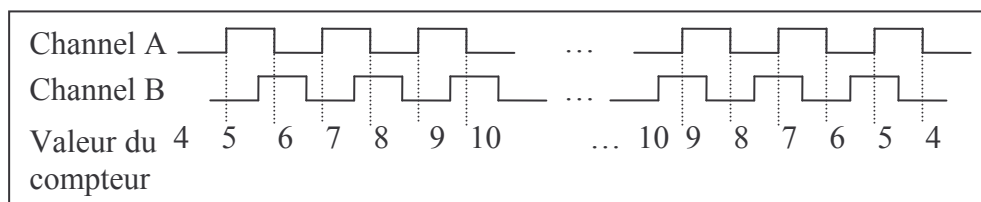
##### 8.1.1.1 Encodage X1

Quand le channel A devance le channel B, le compteur incrémente sur front montant de channel A. quand le channel B devance le channel A, le compteur décrémente sur front descendant de channel A.



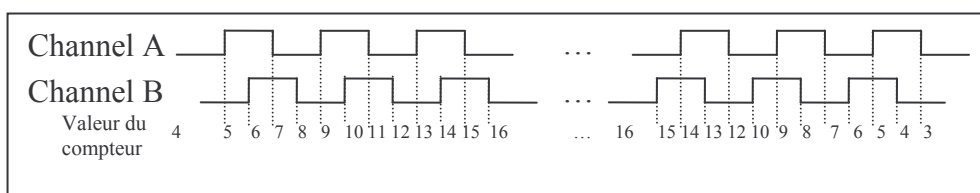
##### 8.1.1.2 Encodage X2

C'est le même comportement que pour l'encodage X1, excepté que le compteur incrémente et décrémente sur chaque front de channel A, en fonction de quel channel précède l'autre. Il y aura deux incréments ou décréments par cycle, comme le montre le schéma suivant :



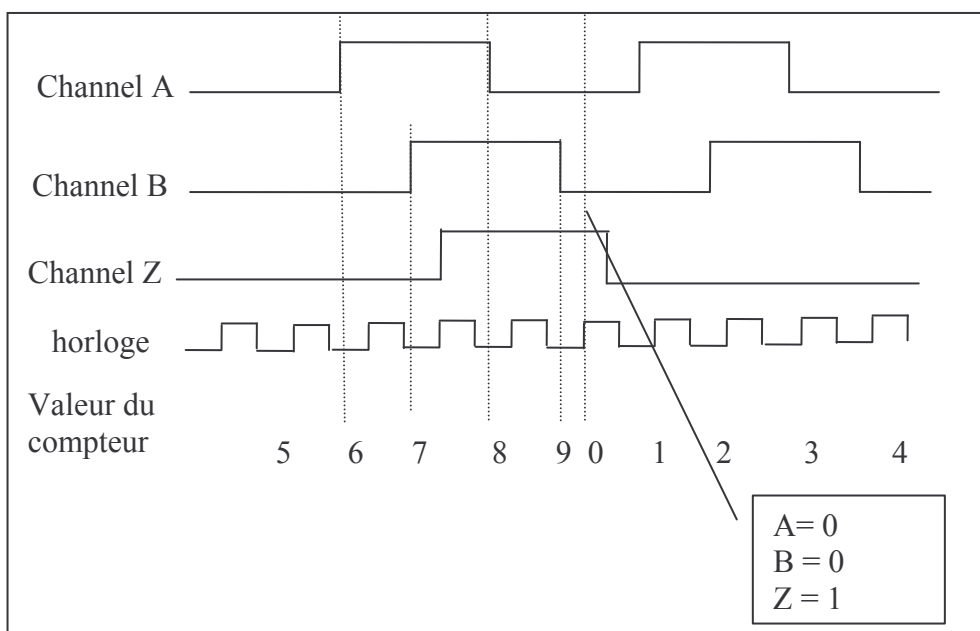
##### 8.1.1.3 Encodage X4

Le compteur incrémente ou décrémente sur chaque front de channel A et channel B. Suivant qu'un channel mène l'autre ou pas, le compteur incrémente ou décrémente. Il y a quatre incréments ou décréments par cycle :



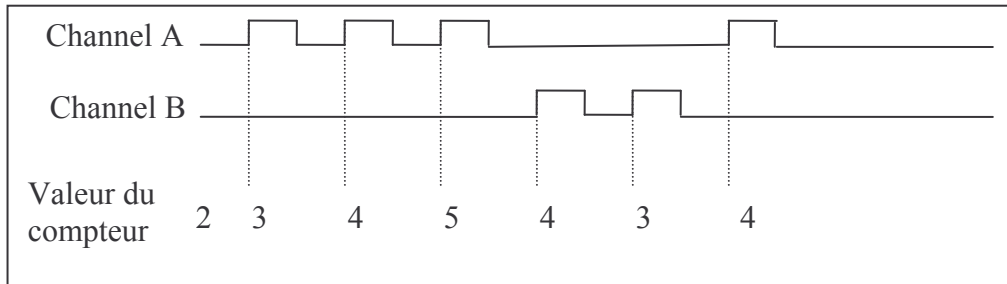
#### 8.1.1.4 Channel Z

Certains encodeurs ont un troisième channel, Z, qui est un channel d'index. Le niveau haut du channel Z permet de recharger le compteur pour une phase spécifique de channel A et channel B. Il est possible de configurer la phase. Une des configurations possibles est que channel A et channel B doivent être à l'état bas quand channel Z est à l'état haut pour recharger le compteur :



#### 8.1.2 Encodeurs sur deux impulsions

Ce type d'encodage (« two-pulse encoder ») supporte deux voies : channel A et channel B. Une impulsion sur le channel A incrémente le compteur. Une impulsion sur le channel B décrémente le compteur.



## 8.2 Configuration

On peut configurer la mesure d'impulsions de deux façons différentes. On peut soit utiliser un encodeur angulaire qui permet de connaître la position du moteur en angle (degrés par exemple), soit on utilise un encodeur linéaire pour connaître la position du moteur converti en distance (mètres, par exemple). Il n'est seulement possible d'ajouter une voie à une instance d'une classe de mesure de position. Appeler deux fois les méthodes `add_angular_channel` ou `add_linear_channel`, renvoie une exception.

Pour un encodeur linéaire on utilise la classe `ni660Xsl::AngularEncoderChan`. Voici un exemple :

```
try
{
    ni660Xsl::SimplePositionMeasurement position;
    ni660Xsl::AngularEncoderChan chan;
    chan.chan_name = "/Dev1/ctr0"; //use ctr0 of Dev1
    chan.decoding_type = ni::two_pulse; //two-pulse encoder
    chan.initial_angle = 0.0; //the initial angle is 0 degrees.
    chan.pulse_per_revolution = 24; //there are 24 pulses per revol. Of the
                                   //motor
    chan.z_idx_enable = true; //enable z indexing
    chan.z_idx_phase = ni::AHigh_BLow; // the Z phase is A high and B low.
    chan.z_idx_val = 0; // the value of the counter when reset.
    chan.units = ni::degrees; //the units used are degrees.
    position.add_angular_encoder(chan); //add 'chan' to 'position'
} catch (const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

Pour utiliser un encodeur linéaire, c'est-à-dire quand le mouvement circulaire du moteur est transformé en un mouvement linéaire, on utilise la classe `ni660Xsl::SimplePositionMeasurement`. Exemple :

```
try
{
    ni660Xsl::SimplePositionMeasurement position2;
    ni660Xsl::LinearEncoderChan chan2;
    chan2.chan_name = "/Dev1/ctr0";
    chan2.decoding_type = ni::x1; //x1 encoding
    chan2.distance_per_pulse = 1.0; //each step of the encoder represents
                                   // a movement of 1 meter.
    chan2.initial_position = 0; //the initial position is 0 meters.
    chan2.units = ni::meters; //use units meters.
    position2.add_linear_encoder(chan2); //add chan 2 to position 2
} catch (const ni660Xsl::DAQException& e)
```

```
{/*manage exception*/}
```

### 8.3 Simple position measurement

On peut faire de la mesure de position simple de moteur avec la classe `ni660Xsl::SimplePositionMeasurement`. On récupère la valeur courante mesurée les méthodes suivantes :

- `get_current_raw_value()` : retourne la valeur courante du compteur (la valeur de comptage brute, un entier).
- `get_current_scaled_value()` : récupère la position courante du moteur dans l'unité spécifié dans la configuration (degrés par exemple).

Il est possible de démarrer l'acquisition sur un trigger externe. Pour l'utiliser on fait appel à la fonction `set_start_trigger(trigger_source, active_edge)`.

Exemple :

```
try
{
    ni660Xsl::SimplePositionMeasurement position;
    //-----config chan-----
    ni660Xsl::AngularEncoderChan chan;
    chan.chan_name = "/Dev1/ctr0";
    chan.decoding_type = ni::x4;
    chan.initial_angle = 0.0;
    chan.pulse_per_revolution = 24;
    chan.z_idx_enable = false;
    chan.z_idx_phase = ni::AHigh_BHigh;
    chan.z_idx_val = 0;
    chan.units = ni::degrees;

    position.add_angular_encoder(chan);
    //-----optional-----
    position.set_start_trigger("/Dev1/PFI33", ni::rising_edge);
    //-----init task-----
    position.init();
    //-----config hw -----
    position.configure();
    //-----start measuring position-----
    position.start();
    //-----read counter-----
    for (int i= 0; i<100; i++)
    {
        double p = position.get_current_scaled_value();
        //do something with data...
    }
    //-----stop task-----
    position.stop();
    position.release();
} catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

### 8.4 Buffered position measurement

C'est le même principe que l'acquisition simple sauf que les données sont enregistrées dans un buffer à une fréquence définie par une horloge (cf. §Acquisition Bufférisée).

Pour implémenter une acquisition bufférisée de position de moteur, on doit hériter sa propre classe de la classe `ni660Xsl::BufferedPositionMeasurement`. La méthode `handle_raw_data` permet de récupérer les données de comptage, des entiers. La méthode `handle_scaled_data` permet de récupérer les données dans l'unité configurée par l'utilisateur (degrés par exemple). Et, il faut absolument **deleter les buffers reçus**. Voici un exemple où la classe `MyMeasurement` hérite de `ni660Xsl::BufferedEventCounting` :

```
class MyMeasurement: public ni660Xsl::BufferedEventCounting
{
public:
    MyMeasurement(void)
        :ni660Xsl::BufferedEventCounting(),
        overrun(0),
        timeout(0)
    {};
    virtual ~MyMeasurement(void)
    {};
    /**
    * count the number of overrun
    */
    void handle_data_lost(void)
    {
        overrun++;
        std::cout<<"overrun nb :"<<overrun<<std::endl;
    };
    /**
    * count the number of timeout
    */
    void handle_timeout(void)
    {
        timeout++;
        std::cout<<"timeout nb :"<<timeout<<std::endl;
    };
    void handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long& _samples_read)
    {
        std::cout<<"raw"<<std::endl;
        for (int j =0; j <buffer->depth(); j = j+10)
            std::cout<<"val["<<j<<"] : "<<(*buffer)[j]<<std::endl;
        delete buffer;
    };
    void handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer, long&
                               _samples_read)
    {
        std::cout<<"scaled"<<std::endl;
        for (int j =0; j <buffer->depth(); j = j+10)
            std::cout<<"val["<<j<<"] : "<<(*buffer)[j]<<std::endl;
        delete buffer;
    };
private:
    int overrun;
    int timeout;
};
```

Dans l'application principale, on paramètre l'acquisition et on la démarre. La lecture des données se fait avec les fonctions `get_raw_data` si on veut récupérer les données dans `handle_raw_data` ou avec `get_scaled_data` si on veut avoir les données dans la fonction `handle_scaled_data`.

Exemple d'acquisition finie :

```
try
{
    //-----create a Buffered Event Counting operation-----
    MyMeasurement position;

    //-----config channel-----
    ni660Xsl::AngularEncoderChan chan;
    chan.chan_name = "/Dev1/ctr0";
    chan.decoding_type = ni::two_pulse;
    chan.initial_angle = 0.0;
    chan.pulse_per_revolution = 24;
    chan.z_idx_enable = false;
    chan.z_idx_phase = ni::AHigh_BLow;
    chan.z_idx_val = 0;
    chan.units = ni::degrees;

    position.add_angular_encoder(chan);
    //-----config timeout-----
    position.set_timeout(1.0);
    position.set_overshoot_strategy(ni::abort);
    //-----config buffer-----
    position.set_buffer_depth(2000);
    position.set_timing_mode(ni::finite);
    //-----config sample clk (gate of counter)-----
    position.set_sample_clock("/Dev1/PFI38", ni::rising_edge, 15000000);
    //-----init-----
    position.init();
    //-----config hw-----
    position.configure();
    //-----start acq-----
    position.start();
    //-----read input data-----
    //get data only once because timing mode is finite.
    position.get_scaled_buffer();
    //-----stop-----
    position.stop();
    //-----release hw-----
    position.release();
} catch (const ni660Xsl::DAQException& e)
{ /*manage exception*/ }
```

Exemple d'acquisition continue :

```
try
{
    //-----create a Buffered Event Counting operation-----
    MyMeasurement position;

    //-----config channel-----
    ni660Xsl::AngularEncoderChan chan;
    chan.chan_name = "/Dev1/ctr0";
    chan.decoding_type = ni::two_pulse;
    chan.initial_angle = 0.0;
```



```

chan.pulse_per_revolution = 24;
chan.z_idx_enable = false;
chan.z_idx_phase = ni::AHigh_BLow;
chan.z_idx_val = 0;
chan.units = ni::degrees;

position.add_angular_encoder(chan);
//-----config timeout-----
position.set_timeout(1.0);
position.set_overnrun_strategy(ni::abort);
//-----config buffer-----
position.set_buffer_depth(2000);
position.set_timing_mode(ni::continuous);
//-----config sample clk (gate of counter)-----
position.set_sample_clock("/Dev1/PFI38", ni::rising_edge, 15000000);
//-----init-----
position.init();
//-----config hw-----
position.configure();
//-----start acq-----
position.start();
//-----read input data-----
for (int i =0; i <100; i++)
{
    //get data continuously
    position.get_scaled_buffer();
}
//-----stop-----
position.stop();
//-----release hw-----
position.release();
} catch(const ni660Xsl::DAQException& e)
{ /*manage exception*/ }

```

## 9 Routage interne

Il est possible de faire du routage dans une carte sans avoir à instancier une classe pour un mode, de façon statique. On peut alors relier plusieurs compteurs entre eux ou relier les compteurs avec des connections extérieures.

Pour faire ce routage, on utilise une méthode statique de la classe `ni660Xsl::SystemProperties` :

```
route_terminals(source, destination, invert_polarity) ;
```

Voici, un exemple de routage entre la sortie du compteur 0 et la gate du compteur 1.

```
ni660Xsl::SystemProperties::route_terminals(«/Dev1/ctr0InternalOutput»,
                                             «/Dev1/ctr0Gate», false ) ;
```

Ici, on relie la sortie 0 à la broche PFI6 :

```
ni660Xsl::SystemProperties::route_terminals(«/Dev1/ctr0InternalOuput»,
                                             «/Dev1/PFI6», false ) ;
```

Les routages faits ne s'enlèvent pas automatiquement. Il faut « reseter » la carte pour les enlever. On utilise la fonction statique `reset_device` en lui passant en paramètre la carte à reseter. Ici, on reset la carte Dev1 :

```
ni660Xsl::SystemProperties::reset_device («Dev1») ;
```

Il est donc fortement conseillé de **reseter le device à chaque fois que l'on démarre une nouvelle application**, au cas où des restes de routage resteraient d'une application différente précédemment exécutée sur la carte.

## 10 Gestion des erreurs

Le driver NI-DAQmx et la librairie NI660Xsl produisent des erreurs et des warnings que l'on peut gérer de la façon suivante.

### 10.1 Exceptions

La librairie génère des exceptions qu'il faut « catcher » à chaque fois qu'une erreur se produit. Les exceptions produites, sont de la classe `ni660Xsl::DAQException`. La classe `DAQException` contient une liste contenant des objets de type `ni660Xsl::Error` avec les champs suivants :

- `reason` : La raison de l'erreur.
- `desc` : La description de l'erreur. Si l'erreur vient du driver, c'est la chaîne de caractères renvoyée par le driver.
- `origin` : La méthode dans laquelle s'est produite l'erreur.
- `code` : Le code d'erreur renvoyé par le driver (un entier négatif).
- `severity` : La sévérité de l'erreur (`WARN`, `ERR`, `PANIC`).

Toutes les erreurs sont empilées dans la liste d'erreurs, il est donc possible de retrouver toutes les erreurs.

Exemple:

```
try
{
//methods called in ni660Xsl
}
catch(const ni660Xsl::DAQException &e)
{
for (int i = 0; i < e.errors.size(); i++)
{
std::cout<<"- reason:\n \t"<< e.errors[i].reason.c_str()<<std::endl;
std::cout<<"- desc:\n"<< e.errors[i].desc.c_str()<<std::endl;
std::cout<<"- origin:\n \t"<< e.errors[i].origin.c_str()<<std::endl;
std::cout<<"- code:\n \t"<< e.errors[i].code<<std::endl;
std::cout<<"- severity:\n \t"<< e.errors[i].severity<<std::endl;
}
```

```

}
}

```

## 10.2 Warnings

Le driver génère aussi des warnings que l'utilisateur de NI660XSL peut choisir de gérer ou non. Les méthodes générant des exceptions peuvent aussi générer des warnings. Il est possible de savoir si un warning est arrivé dans la dernière méthode appelée avec la méthode `warn_occured()` qui retourne vrai si un warning a été généré. On récupère la chaîne de caractères de ce warning avec la méthode `get_warn_string()`.

Exemple :

```

ContinuousPulseTrainGeneration cont_train;
try
{
    cont_train.init() ;
    if(cont_train.warn_occured())
    {
        std::cout<<cont_train.get_warn_string()<<std::endl;
    }
}
catch(const ni660Xsl::DAQException &e)
{
    //manage exception
}

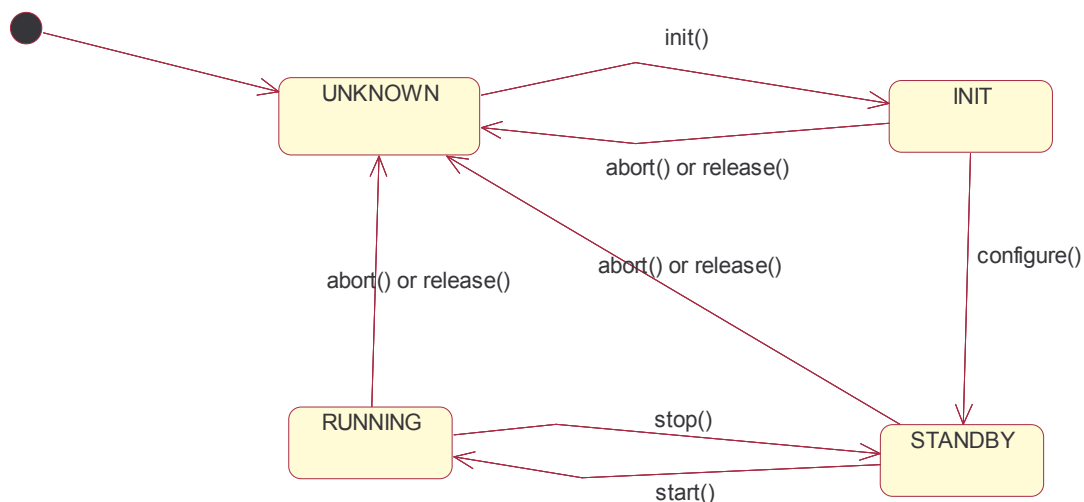
```

## 11 Etats des opérations de comptage

Chaque opération de ni660Xsl comporte des états. Les états possibles sont :

- UNKNOWN : l'état de l'opération est inconnu.
- INIT : l'opération a été initialisée correctement.
- STANDBY : l'opération est arrêtée.
- RUNNING : l'opération est en train de fonctionner.

On peut résumer la séquence des états au diagramme suivant :



On connaît l'état avec la méthode `state()` présente dans toutes les opérations.

## 12 Intégration dans un device TANGO

**NB : Pour que le device fonctionne correctement, il faut inclure le fichier ACE.h (fourni avec la distribution TANGO de SOLEIL) dans le main du device : `#include <ace/ACE.h>`**

### 12.1 Conversion des exceptions de NI660Xsl vers TANGO

Il est aussi très simple de convertir les exceptions `ni660Xsl::DAQException` en exception TANGO, `Tango::DevFailed`. Voici une méthode à intégrer dans un device :

```
Tango::DevFailed MeasurePosition::daq_to_tango_exception(const ni660Xsl::DAQException&
de)
{
    Tango::DevErrorList error_list(de.errors.size());
    error_list.length(de.errors.size());

    for(int i = 0; i < de.errors.size(); i++)
    {
        error_list[i].reason = CORBA::string_dup(de.errors[i].reason.c_str());
        error_list[i].desc = CORBA::string_dup(de.errors[i].desc.c_str());
        error_list[i].origin = CORBA::string_dup(de.errors[i].origin.c_str());
        switch(de.errors[i].severity)
        {
            case ni660Xsl::WARN:
                error_list[i].severity = Tango::WARN;
                break;
            case ni660Xsl::PANIC:
                error_list[i].severity = Tango::PANIC;
                break;
            case ni660Xsl::ERR:
            default:
                error_list[i].severity = Tango::ERR;
                break;
        }
    }
    return Tango::DevFailed(error_list);
}
```

### 12.2 Conversion des états de NI660Xsl vers TANGO

Les états de `ni660Xsl` existent aussi dans TANGO. On peut mettre à jour les états et les status du device avec, par exemple la méthode suivante :

```
ni660Xsl ::SimplePositionMeasurement* meas_pos =
    new ni660Xsl ::SimplePositionMeasurement() ;
...
```

...

```
void MeasurePosition::set_internal_state(void)
{
    switch(meas_pos->state())
    {
        case ni660Xsl::SimplePositionMeasurement::STANDBY:
            this->set_state(Tango::STANDBY);
            this->set_status("The acquisition is stopped");
            break;
        case ni660Xsl::SimplePositionMeasurement::RUNNING:
            this->set_state(Tango::RUNNING);
            this->set_status("The acquisition is running");
            break;
        case ni660Xsl::SimplePositionMeasurement::INIT:
            this->set_state(Tango::INIT);
            this->set_status("The acquisition is initialized");
            break;
        case ni660Xsl::SimplePositionMeasurement::UNKNOWN:
        default:
            this->set_state(Tango::UNKNOWN);
            this->set_status("The acquisition is in an unknown state");
            break;
    }
}
```

### 12.3 Exemple : « Continuous pulse generation »

Voici un exemple de device qui génère un train continu d'impulsions. Les commandes et les propriétés du device sont les suivantes :

- Commandes :
  - o Start : démarre la génération du train d'impulsions.
  - o Stop : Stoppe la génération.
  - o Abort : Abort la génération (en cas de problème).
- Attributs : aucun
- Propriétés :
  - o device\_name : le nom de la carte donné dans MAX (Measurement & Automation Explorer).
  - o low\_ticks : la largeur du niveau bas des impulsions en clock ticks.
  - o high\_ticks : la largeur du niveau haut des impulsions en clock ticks.

Le .h du device contient les lignes suivantes :

```
protected :
ni660Xsl::ContinuousPulseTrainGeneration* cont_train;
Tango::DevFailed daq_to_tango_exception(const ni660Xsl::DAQException& de);
void set_internal_state(void);
```

## 12.3.1 init\_device

Voici le contenu de la méthode `init_device`. Ici, on initialise la carte avec les propriétés du device et on démarre la génération des impulsions :

```
// Initialise variables to default values
//-----
cont_train = 0;

// Initialise variables to default values
//-----
get_device_property();

try
{
    cont_train = new ni660Xsl::ContinuousPulseTrainGeneration();
    if(cont_train == 0)
    {
        Tango::Except::throw_exception(
            (const char*)"out of memory",
            (const char*)"out of memory error",
            (const char*)"SampleClock::init_device");
    }

    ni660Xsl::SystemProperties::reset_device(device_name);

    ni660Xsl::OutClockTicksChan chan;

    chan.chan_name = "/" + device_name + "/ctr0";
    chan.clk_source = "/" + device_name + "/80MHzTimebase"; //use internal clock
    chan.idle_state = ni::low;
    chan.initial_delay = 0; //clk ticks
    chan.low_ticks = low_ticks; // property of the device
    chan.high_ticks = high_ticks; // property of the device

    cont_train->add_clock_ticks_channel(chan);

    cont_train->init();

    cont_train->configure();

    cont_train->start();
}
catch(const ni660Xsl::DAQException& de)
{
    Tango::DevFailed df = daq_to_tango_exception(de); //see §12.1
    ERROR_STREAM << df<<endl;
    this->delete_device();
    Tango::Except::re_throw_exception(df,
        (const char*)"device initialization failed",
        (const char*)"caught ni660Xsl::DAQException",
        (const char*)"PulseGeneration::init_device");
}
catch(...)
{
    ERROR_STREAM << " PulseGeneretion::init_device::unknown exception
caught"<<std::endl;
    this->delete_device();
    Tango::Except::throw_exception(
        (const char*)"device initialization failed",
        (const char*)"unknown exception caught",
        (const char*)" PulseGeneration::init_device")
}
```

```
    );  
}  
  
this->set_internal_state(); //see §12.2
```

### 12.3.2 Commandes

Voici le contenu de la commande Start:

```
try  
{  
    cont_train->start();  
    this->set_internal_state(); //see §12.2  
}  
catch(const ni660Xsl::DAQException& de)  
{  
    Tango::DevFailed df = daq_to_tango_exception(de); //see §12.1  
    ERROR_STREAM << df<<endl;  
    Tango::Except::re_throw_exception(df,  
        (const char*)("start failed"),  
        (const char*)("caugth ni660Xsl::DAQException"),  
        (const char*)(" PulseGeneretion::start")  
    );  
}  
catch(...)  
{  
    ERROR_STREAM << " PulseGeneretion::start::unknown exception caught"<<std::endl;  
  
    Tango::Except::throw_exception(  
        (const char*)("start failed"),  
        (const char*)("unknown exception caugth"),  
        (const char*)(" PulseGeneretion::start")  
    );  
}
```

Pour les commandes Stop et Abort c'est la même chose, il suffit de remplacer la ligne :

```
cont_train->start();
```

Par :

```
cont_train->stop(); ou cont_train->abort();
```

## 12.4 Exemple : acquisition bufferisée finie

Dans cet exemple on fait de l'acquisition bufferisée de position de moteur.

- Commandes :
  - o Start
  - o Stop
  - o Abort
- Attributs : `position`. C'est un spectrum contenant les acquisitions des positions en degrés.
- Propriétés :

- `device_name` : le nom du device donnée par MAX.
- `decoding_type` : le type de décodage (cf. §8.2)
- `pulse_per_revolution` : le nombre de pas par tour.
- `initial_angle` : l'angle de départ.
- `buffer_depth` : la profondeur du buffer.
- `sample_clock` : l'horloge d'échantillonnage.

#### 12.4.1 Réception des données

Comme il faut hériter de la classe `ni660Xsl::BufferedPositionMeasurement`, on crée une autre classe à coté du device, `DataHandler`. Cette classe permet principalement de réceptionner les données de l'acquisition venant du driver. Le buffer de données reçu est copié dans un buffer temporaire public de la classe `DataHandler`. Ce buffer (`position_buffer`) sera ensuite lu par le device pour mettre à jour l'attribut « position ». Comme ce buffer peut être lu et écrit en même temps on utilise un mutex (`data_lock_`). On connaît le nombre de buffers reçus avec la donnée membre `data_counter`.

Voici les méthodes de la classe `DataHandler` dans le `.h` :

```
class DataHandler:
public ni660Xsl::BufferedPositionMeasurement,
public Tango::LogAdapter //LogAdapter for Tango logging
{
public:
    DataHandler(Tango::DeviceImpl * dev, unsigned long buffer_depth);
    virtual ~DataHandler(void);
    /**
     * count the number of overrun
     */
    void handle_data_lost(void);
    /**
     * count the number of timeout
     */
    void handle_timeout(void);
    void handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long& _samples_read);
    void handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer,
                              long& _samples_read);

    inline void lock_data(void)
    {
        this->data_lock_.acquire();
    };

    inline void unlock_data(void)
    {
        this->data_lock_.release();
    };
    ni660Xsl::InScaledBuffer* position_buffer;
    int data_counter;
private:
    int overrun;
    int timeout;
    ACE_Thread_Mutex data_lock_;

};
```



On reçoit les données de position de moteur dans la méthode `handle_scaled_data`. Dans cette méthode, on copie le buffer reçu dans la variable `position_buffer`. Voici le .cpp de `DataHandler`:

```
DataHandler::DataHandler(Tango::DeviceImpl * dev, unsigned long buffer_depth)
:
ni660Xsl::BufferedPositionMeasurement(),
Tango::LogAdapter(dev), //for tango logging
overrun(0),
timeout(0),
position_buffer(0)
{
    position_buffer = new ni660Xsl::InScaledBuffer(buffer_depth);
    if(position_buffer == 0)
    {
        //TODO: throw exception
    }
}
DataHandler::~DataHandler(void)
{
    if(position_buffer)
    {
        delete position_buffer;
        position_buffer = 0;
    }
}
void DataHandler::handle_data_lost(void)
{
    overrun++;
    std::cout<<"overrun nb :"<<overrun<<std::endl;
}
void DataHandler::handle_timeout(void)
{
    timeout++;
    std::cout<<"timeout nb :"<<timeout<<std::endl;
}
void DataHandler::handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long&
_samples_read)
{
    delete buffer;
}
void DataHandler::handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer, long&
_samples_read)
{
    //obtain mutex
    this->lock_data();

    data_counter++;
    //copy buffer to a position_buffer.
    ACE_OS::memcpy(this->position_buffer->base(), buffer->base(), buffer->size());

    this->unlock_data();

    delete buffer;
}
```

#### 12.4.2 init\_device

Voici l'implémentation du `init_device` dans lequel on initialise `DataHandler`. `position_buffer` est le buffer qui va écrire dans l'attribut `position` du device. On initialise l'acquisition de position du moteur puis on démarre l'acquisition (`buf_pos->start()`).

```
try
{
    //in the .h : DataHandler* buf_pos
    buf_pos = new DataHandler(this, buffer_depth);
    if(buf_pos == 0)
    {
        Tango::Except::throw_exception(
            (const char*)"device initialization failed"),
            (const char*)"out of memory",
            (const char*)"PositionBuffered::init_device"
        );
    }
    //in the .h : double* position_buffer
    position_buffer = new double[buffer_depth];
    if(position_buffer == 0)
    {
        this->delete_device();
        Tango::Except::throw_exception(
            (const char*)"device initialization failed"),
            (const char*)"out of memory",
            (const char*)"PositionBuffered::init_device"
        );
    }
    ni660Xsl::AngularEncoderChan chan;
    chan.chan_name = "/" + device_name + "/ctr0";
    if(decoding_type == "X1") //decoding_type is a property of the device
    {
        chan.decoding_type = ni::x1;
    }
    else if(decoding_type == "X2")
    {
        chan.decoding_type = ni::x2;
    }
    else if(decoding_type == "X4")
    {
        chan.decoding_type = ni::x4;
    }
    else if(decoding_type == "2P")
    {
        chan.decoding_type = ni::two_pulse;
    }
    chan.initial_angle = initial_angle; //initial_angle is a property of the device
    chan.pulse_per_revolution = pulse_per_revolution; //property of the device

    chan.z_idx_enable = false;
    chan.z_idx_phase = ni::AHigh_BHigh;
    chan.z_idx_val = 0;
    chan.units = ni::degrees;

    buf_pos->add_angular_encoder(chan);
    buf_pos->set_buffer_depth(buffer_depth);
    buf_pos->set_timing_mode(ni::finite);
    buf_pos->set_sample_clock("/"+device_name+"/"+sample_clock, ni::rising_edge,
                             150000000);
    buf_pos->set_overrun_strategy(ni::restart);

    buf_pos->set_timeout(1.0);
    buf_pos->init();
    buf_pos->configure();
    buf_pos->start();
}
```

```

    }
    catch(const ni660Xsl::DAQException& de)
    {
        Tango::DevFailed df = daq_to_tango_exception(de);
        ERROR_STREAM << df<<endl;
        this->set_internal_state();
        this->delete_device();
        Tango::Except::re_throw_exception(df,
            (const char*)("device initialization failed"),
            (const char*)("caugth ni660Xsl::DAQException"),
            (const char*)("PositionBuffered::init_device")
        );
    }
    catch(...)
    {
        ERROR_STREAM << "PositionBuffered::init_device::unknown exception
            caught"<<std::endl;
        this->set_internal_state();
        this->delete_device();
        Tango::Except::throw_exception(
            (const char*)("device initialization failed"),
            (const char*)("unknown exception caugth"),
            (const char*)("PositionBuffered::init_device")
        );
    }

    this->set_internal_state();

```

### 12.4.3 read\_attr\_hardware

Dans cette méthode, on lance l'acquisition du buffer. Comme `read_attr_hardware` peut être appelé plusieurs fois alors que l'acquisition finie ne récupère qu'un seul buffer, on ne lance qu'une seule fois l'acquisition du buffer grâce à la ligne de code `if(buf_pos->data_counter == 0)`. En effet quand `data_counter` est à 0, cela veut dire que aucune donnée n'a été encore reçue. On recopie ensuite le buffer temporaire de la classe `DataHandler` (`buf_pos->position_buffer`) dans une variable locale (`this->position_buffer`) en utilisant le mutex :

```

try
{
    if(buf_pos->data_counter == 0 && this->state() == Tango::RUNNING)
    {
        buf_pos->get_scaled_buffer();
        this->set_internal_state();
        buf_pos->lock_data();

        ACE_OS::memcpy(this->position_buffer,
            buf_pos->position_buffer->base(),
            buf_pos->position_buffer->size());

        buf_pos->unlock_data();
    }
}
catch(const ni660Xsl::DAQException& de)
{
    Tango::DevFailed df = daq_to_tango_exception(de);
    ERROR_STREAM << df<<endl;
    this->set_internal_state();
    Tango::Except::re_throw_exception(df,
        (const char*)("read failed"),
        (const char*)("caugth ni660Xsl::DAQException"),

```

```

        (const char*)("AcqMotorPosition::read_attr_hardware")
    );
}
catch(...)
{
    ERROR_STREAM << "AcqMotorPosition::read_attr_hardware : unknown exception
                    caught"<<std::endl;
    this->set_internal_state();
    Tango::Except::throw_exception(
        (const char*)("read failed"),
        (const char*)("unknown exception caught"),
        (const char*)("AcqMotorPosition::read_attr_hardware")
    );
}

```

#### 12.4.4 read\_attr

Il ne reste plus qu'à mettre `this->position_buffer` dans l'attribut du device :

```

if (attr_name == "position")
{
    attr.set_value(this->position_buffer, buffer_depth);
}

```

#### 12.4.5 Commandes

Dans les commande Stop et Abort, on arrête l'acquisition avec les méthodes `buf_pos->stop()` ou `buf_pos->abort()`:

```

try
{
    buf_pos->stop();
    this->set_internal_state();

}
catch(const ni660Xsl::DAQException& de)
{
    //manage exception
}
catch(...)
{
    //manage exception
}

```

Dans la commande Start, on démarre l'acquisition puis le thread :

```

try
{
    buf_pos->start();
    this->set_internal_state();
    //- run the thread for getting buffer
    if (thread->run() == -1)
    {

        cout<<"Thread::run failed\n"<<endl;
        this->delete_device();
        Tango::Except::throw_exception(
            (const char*)("start failed"),
            (const char*)("enable to start thread for getting buffer"),
            (const char*)("PositionBuffered::start")
        );
    }
}

```

```

        );
    }

}
catch(const ni660Xsl::DAQException& de)
{
    //manage exception
}
catch(...)
{
    //manage exception
}

```

## 12.5 Exemple : acquisition bufferisée continue

Dans cet exemple on fait de l'acquisition bufferisée de position de moteur.

- Commandes :
  - Start
  - Stop
  - Abort
- Attributs : `position`. C'est un spectrum contenant les acquisitions des positions en degrés.
- Propriétés :
  - `device_name` : le nom du device donnée par MAX.
  - `decoding_type` : le type de décodage (cf. §8.2)
  - `pulse_per_revolution` : le nombre de pas par tour.
  - `initial_angle` : l'angle de départ.
  - `buffer_depth` : la profondeur du buffer.
  - `sample_clock` : l'horloge d'échantillonnage.

### 12.5.1 Réception des données

Comme il faut hériter de la classe `ni660Xsl::BufferedPositionMeasurement`, on crée une autre classe à coté du device, `DataHandler`. Cette classe permet principalement de réceptionner les données de l'acquisition venant du driver. Le buffer de données reçu est copié dans un buffer temporaire public de la classe `DataHandler`. Ce buffer sera ensuite lu par le device pour mettre à jour l'attribut « position ». Comme ce buffer peut être lu et écrit en même temps on utilise un mutex(`data_lock_`).

Voici les méthodes de la classe `DataHandler` dans le `.h` :

```

class DataHandler:
public ni660Xsl::BufferedPositionMeasurement,
public Tango::LogAdapter //LogAdapter for Tango logging
{
public:
    DataHandler(Tango::DeviceImpl * dev, unsigned long buffer_depth);
    virtual ~DataHandler(void);
    /**

```

```

    * count the number of overrun
    */
    void handle_data_lost(void);
    /**
    * count the number of timeout
    */
    void handle_timeout(void);
    void handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long& _samples_read);
    void handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer,
                              long& _samples_read);

    inline void lock_data(void)
    {
        this->data_lock_.acquire();
    };

    inline void unlock_data(void)
    {
        this->data_lock_.release();
    };
    ni660Xsl::InScaledBuffer* position_buffer;
private:
    int overrun;
    int timeout;
    ACE_Thread_Mutex data_lock_;

};

```

On reçoit les données de position de moteur dans la méthode `handle_scaled_data`. Dans cette méthode, on copie le buffer reçu dans la variable `position_buffer`. Voici le .cpp de `DataHandler`:

```

DataHandler::DataHandler(Tango::DeviceImpl * dev, unsigned long buffer_depth)
:
ni660Xsl::BufferedPositionMeasurement(),
Tango::LogAdapter(dev), //for tango logging
overrun(0),
timeout(0),
position_buffer(0)
{
    position_buffer = new ni660Xsl::InScaledBuffer(buffer_depth);
    if(position_buffer == 0)
    {
        //TODO: throw exception
    }
}
DataHandler::~DataHandler(void)
{
    if(position_buffer)
    {
        delete position_buffer;
        position_buffer = 0;
    }
}
void DataHandler::handle_data_lost(void)
{
    overrun++;
    std::cout<<"overrun nb :"<<overrun<<std::endl;
}
void DataHandler::handle_timeout(void)
{
    timeout++;
    std::cout<<"timeout nb :"<<timeout<<std::endl;
}

```

```

void DataHandler::handle_raw_buffer(ni660Xsl::InRawBuffer* buffer, long&
_samples_read)
{
    delete buffer;
}
void DataHandler::handle_scaled_buffer(ni660Xsl::InScaledBuffer* buffer, long&
_samples_read)
{
    //obtain mutex
    this->lock_data();

    //copy buffer to a position_buffer.
    ACE_OS::memcpy(this->position_buffer->base(), buffer->base(), buffer->size());

    this->unlock_data();

    delete buffer;
}

```

### 12.5.2 Acquisition

Pour que les données arrivent en continu dans la classe `DataHandler`, il faut faire une boucle sur la méthode `get_scaled_buffer()` qui renvoie ces données à l'utilisateur. Il faut donc créer un `Thread` qui permet de faire cette boucle à par du device. Pour cela, on hérite de la classe `ni660Xsl::Thread`. Dans cette classe on fera la boucle d'acquisition dans la méthode `svc`.

En voici l'implémentation :

```

class AcquisitionThread : public ni660Xsl::Thread
{
public:
    AcquisitionThread (ni660Xsl::BufferedPositionMeasurement* measure,
                      Tango::DeviceImpl * dev)
        : ni660Xsl::Thread(),
        device(dev),
        Tango::LogAdapter(dev), //for logging in device
        meas_pos(measure)
    {}
    virtual ~AcquisitionThread (){};
protected:
    virtual ACE_THR_FUNC_RETURN svc (void* arg)
    {
        //- enter almost infinite loop
        do
        {
            try
            {
                meas_pos->get_scaled_buffer();
            }
            catch(const ni660Xsl::DAQException& e)
            {
                device->set_state(Tango::UNKNOWN);
                device->set_status("An error as occurred while getting data");
                //manage exception
            }
            catch(...)
            {
                //manage exception
            }
        }while (quit_requested() == false);

        //- return result (here just return <arg> back)
        return ACE_static_cast(ACE_THR_FUNC_RETURN, arg);
    };
}

```

```
ni660Xsl::BufferedPositionMeasurement* meas_pos;  
Tango::DeviceImpl* device;  
};
```

### 12.5.3 init\_device

Voici l'implémentation du `init_device` dans lequel on initialise `DataHandler` et `AcquisitionThread`. `position_buffer` est le buffer qui va écrire dans l'attribut `position` du device. On initialise l'acquisition de position du moteur puis on démarre l'acquisition (`buf_pos->start()`) et le thread (`thread->run()`) qui permet de récupérer les données.

```
try  
{  
    //in the .h : DataHandler* buf_pos  
    buf_pos = new DataHandler(this, buffer_depth);  
    if(buf_pos == 0)  
    {  
        Tango::Except::throw_exception(  
            (const char*)"device initialization failed"),  
            (const char*)"out of memory",  
            (const char*)"PositionBuffered::init_device"  
        );  
    }  
    //in the .h : AcquisitionThread* thread  
    thread = new AcquisitionThread(buf_pos);  
    if(thread == 0)  
    {  
        this->delete_device();  
        Tango::Except::throw_exception(  
            (const char*)"device initialization failed"),  
            (const char*)"out of memory",  
            (const char*)"PositionBuffered::init_device"  
        );  
    }  
    //in the .h : double* position_buffer  
    position_buffer = new double[buffer_depth];  
    if(position_buffer == 0)  
    {  
        this->delete_device();  
        Tango::Except::throw_exception(  
            (const char*)"device initialization failed"),  
            (const char*)"out of memory",  
            (const char*)"PositionBuffered::init_device"  
        );  
    }  
    ni660Xsl::AngularEncoderChan chan;  
    chan.chan_name = "/" + device_name + "/ctr0";  
    if(decoding_type == "X1") //decoding_type is a property of the device  
    {  
        chan.decoding_type = ni::x1;  
    }  
    else if(decoding_type == "X2")  
    {  
        chan.decoding_type = ni::x2;  
    }  
    else if(decoding_type == "X4")  
    {  
        chan.decoding_type = ni::x4;  
    }  
    else if(decoding_type == "2P")  
    {  
        chan.decoding_type = ni::two_pulse;  
    }  
}
```



```

chan.initial_angle = initial_angle; //initial_angle is a property of the device
chan.pulse_per_revolution = pulse_per_revolution; //property of the device

chan.z_idx_enable = false;
chan.z_idx_phase = ni::AHigh_BHigh;
chan.z_idx_val = 0;
chan.units = ni::degrees;

buf_pos->add_angular_encoder(chan);
buf_pos->set_buffer_depth(buffer_depth);
buf_pos->set_timing_mode(ni::continuous);
buf_pos->set_sample_clock("/"+device_name+"/"+sample_clock, ni::rising_edge,
                        150000000);
buf_pos->set_overrun_strategy(ni::restart);

buf_pos->set_timeout(1.0);
buf_pos->init();
buf_pos->configure();
buf_pos->start();

//- run the thread for getting buffer
if (thread->run() == -1)
{
    ERROR_STREAM << "Thread::run failed"<<endl;
    this->delete_device();
    Tango::Except::throw_exception(
        (const char*)("device initialization failed"),
        (const char*)("enable to start thread for getting buffer"),
        (const char*)("PositionBuffered::init_device")
    );
}

}

catch(const ni660Xsl::DAQException& de)
{
    Tango::DevFailed df = daq_to_tango_exception(de);
    ERROR_STREAM << df<<endl;
    this->set_internal_state();
    this->delete_device();
    Tango::Except::re_throw_exception(df,
        (const char*)("device initialization failed"),
        (const char*)("caugh ni660Xsl::DAQException"),
        (const char*)("PositionBuffered::init_device")
    );
}

catch(...)
{
    ERROR_STREAM << "PositionBuffered::init_device::unknown exception
                    caught"<<std::endl;
    this->set_internal_state();
    this->delete_device();
    Tango::Except::throw_exception(
        (const char*)("device initialization failed"),
        (const char*)("unknown exception caught"),
        (const char*)("PositionBuffered::init_device")
    );
}

this->set_internal_state();

```

#### 12.5.4 read\_attr\_hardware

Dans cette méthode on recopie le buffer temporaire de la classe `DataHandler` (`buf_pos->position_buffer`) dans une variable locale (`this->position_buffer`) en utilisant le mutex :

```
buf_pos->lock_data();
ACE_OS::memcpy(this->position_buffer, buf_pos->position_buffer->base(),
               buf_pos->position_buffer->size());
buf_pos->unlock_data();
```

### 12.5.5 read\_attr

Il ne reste plus qu'à mettre `this->position_buffer` dans l'attribut du device :

```
if (attr_name == "position")
{
    attr.set_value(this->position_buffer, buffer_depth);
}
```

### 12.5.6 Commandes

Dans les commande Stop et Abort, on arrête d'abord le thread (`thread->quit_and_join()`) puis l'acquisition (`buf_pos->stop()` ou `buf_pos->abort()`) :

```
try
{
    //- ask the thread to quit
    if (thread->quit_and_join() == -1)
    {
        Tango::Except::throw_exception(
            (const char*)"stop failed",
            (const char*)"enable to stop thread",
            (const char*)"PositionBuffered::stop"
        );
    }
    buf_pos->stop();
    this->set_internal_state();
}
catch(const ni660Xsl::DAQException& de)
{
    //manage exception
}
catch(...)
{
    //manage exception
}
```

Dans la commande Start, on démarre l'acquisition puis le thread :

```
try
{
    buf_pos->start();
    this->set_internal_state();
    //- run the thread for getting buffer
    if (thread->run() == -1)
    {
        cout<<"Thread::run failed\n"<<endl;
        this->delete_device();
    }
}
```

```
        Tango::Except::throw_exception(
            (const char*)("start failed"),
            (const char*)("enable to start thread for getting buffer"),
            (const char*)("PositionBuffered::start")
        );
    }

}

catch(const ni660Xsl::DAQException& de)
{
    //manage exception
}

catch(...)
{
    //manage exception
}
```

## 13 Les annexes

Cette librairie a été développée avec les drivers NI-DAQmx fournis avec la version NI-DAQ7.2.

Documentation :

- « 6601/6602 User Manual » fourni avec les cartes.
- « NI-DAQmx Help », documentation html fournie avec les drivers.
- Documentation html de la librairie dans le répertoire doc. de NI660Xsl.

Code complet des devices :