

## 1. Student Guide

### 1.1. Contrôle des stations

Ce chapitre est destiné à expliquer les différents moyens de programmation mis en œuvre pour le contrôle des stations de manière indépendante. Avant toute chose, pour pouvoir lire l'état des capteurs du dispositif FESTO et pour pouvoir contrôler les différents actuateurs, il faut pouvoir lire et écrire dans les registres de notre carte d'entrées/sorties PIO-D48 pour les stations 2, 3 & 4.

Dans le cas de la station 1, c'est différent du fait que l'interfaçage est différent. Pour rappel, la station 1 est interfacée par l'intermédiaire du bus de terrain PROFIBUS.

#### 1.1.1. Ouverture du driver

Les drivers sont lancés par défaut au démarrage de la machine. Une fois lancés, ces derniers créent une entrée dans le répertoire /dev :

🕒 Le driver de la carte PIO-D48 créera l'entrée /dev/pio\_d48

🕒 Le driver PROFIBUS créera l'entrée /dev/applicomIO

##### 1.1.1.a) Interface parallèle :

Pour les stations 2, 3 & 4, pour pouvoir communiquer et appeler les fonctions du driver, il faut commencer par obtenir un descripteur sur le driver. Pour cela on utilise la fonction open() :

```
fd = open("/dev/pio_d48", O_RDWR);
```

Nous ouvrons donc le driver en lecture/écriture et nous récupérons un descripteur via la variable fd.

##### 1.1.1.b) Interface PROFIBUS :

Dans le cas de PROFIBUS, il ne faut pas faire un open() mais appeler une fonction d'initialisation qui est IO\_Init :

```
IO_Init(NUMCARTE, &status);
```

Le premier paramètre correspond à l'adresse de la carte pci PROFIBUS. L'adresse de notre carte PROFIBUS est 1.

Dans la variable status se trouve le code de retour de la fonction, ce code de retour vaut 0 quand tout se passe bien.

Une fois que tout est bien initialisé, nous pouvons commencer à communiquer avec les drivers et effectuer des entrées/sorties sur le matériel.

## 1.1.2. Utilisation des fonctions des drivers

Jetons un œil sur la manière de réaliser des entrées/sorties :

### 1.1.2.1 Réaliser une écriture

#### 1.2.1.a) Interface parallèle :

Pour envoyer des ordres de contrôle aux actuateurs des stations 2, 3 & 4, nous utilisons la fonction `devctl()` de la manière suivante :

```
ret = devctl(fd, WRITE_ACTUATEUR, S2_ACTUATEURS_BP, sizeof(S2_ACTUATEURS_BP), NULL);
```

Vous pouvez constater que nous passons à cette fonction le descripteur du driver. Vous constatez également que nous appelons la sous fonction `WRITE_ACTUATEUR` de la fonction `devctl()`. Le paramètre `S2_ACTUATEURS_BP` est en réalité un pointeur vers la variable entière qui contient les 3 bytes à écrire dans les 3 registres configurés en sortie sur la carte PIO-D48.

Une fois l'information écrite dans les registres de la carte PIO-D48, l'information sera alors répercutée sur la carte à relais et ensuite sur les actuateurs du FESTO. Terminons par préciser que le quatrième paramètre est la taille du pointeur du nombre entier passé en paramètre et que nous nous ne servons pas du 5<sup>e</sup> paramètre c'est pourquoi nous lui donnons la valeur `NULL`.

Jetons un petit coup d'œil à la structure de notre nombre entier pour mieux comprendre comment sont stockés les bytes à l'intérieur de notre variable de type `int` :

Sur notre carte d'entrées/sorties à 48 canaux, 24 canaux sont configurés en sortie soit 3 bytes. En informatique aucun type prédéfini n'occupe un espace de 3 bytes en mémoire. C'est pourquoi nous avons choisi de travailler avec des variables de type `int` qui occupe 4 bytes en mémoire. De ces 4 bytes, nous utilisons les 3 bytes de poids faibles pour contenir l'information que nous souhaitons écrire dans nos 3 registres configuré en sortie. Le byte de poids fort ne représente rien et n'a aucune signification.

- ⌚ Les bits allant de 0 à 7 constituent le byte qui sera écrit sur le port C du 8255 configuré en sortie
- ⌚ Les bits allant de 8 à 15 constituent le byte qui sera écrit sur le port B du 8255 configuré en sortie
- ⌚ Les bits allant de 16 à 23 constituent le byte qui sera écrit sur le port A du 8255 configuré en sortie
- ⌚ Les bits allant de 24 à 31 n'ont aucune signification.

En cas d'erreur la fonction retourne une valeur inférieure à 0.

## 1.2.1.b)Interface PROFIBUS

Pour envoyer des ordres aux actuators il faut dans un premier temps faire appel à la fonction suivante :

```
IO_WriteQByte(NUMCARTE, ADRESSECPVDIO, 0, 1, S1_ACTUATEURS_LSB_BP,&status);
```

- ⌚ NUMCARTE étant l'adresse de la carte PCU-DPIO située dans le pc de contrôle de la station 1. Pour rappel, dans notre cas, cette adresse vaut 1.
- ⌚ ADRESSECPVDIO n'est rien d'autre que l'adresse du module d'entrée/sortie PROFIBUS situé sur la station 1, dans notre cas, cette adresse vaut 64.
- ⌚ Le 3<sup>e</sup> paramètre vaut 0 et correspond à l'Offset à partir duquel nous souhaitons écrire.
- ⌚ Le 4<sup>e</sup> paramètre valant 1 correspond au nombre de byte à écrire.
- ⌚ S1\_ACTUATEURS\_LSB\_BP correspond à un pointeur pointant le byte qui contient l'information à écrire.
- ⌚ La variable status n'est rien d'autre que la variable qui va stocker le code de retour de la fonction. Quand tout se passe bien la fonction retourne 0.

Une fois que cette fonction a été appelée l'information ne circule toujours pas sur le bus en direction des actuators. En effet, il faut savoir que la carte PROFIBUS-DP que nous avons achetée travaille avec des mémoires tampon, et lorsque nous appelons la fonction IO\_WriteQByte(), nous mettons à jour le contenu de la variable tampon de notre carte mais nous ne mettons pas l'information à disposition sur le bus, pour mettre cette information à disposition sur le bus, il faut appeler la fonction suivante :

```
IO_RefreshOutput(NUMCARTE, &status);
```

- ⌚ Le premier paramètre est en fait le numéro de la carte PCU-DPIO présente dans le pc de contrôle de la station 1. Dans notre cas, NUMCARTE = 1.
- ⌚ La variable status dont nous passons l'adresse, contiendra le code de retour de la fonction. Quand tout se passe bien la fonction retourne 0.

## 1.1.2.2 Réaliser une lecture

### 1.2.2.a) Interface parallèle

Pour la lecture des capteurs sur les stations 2, 3 & 4, nous utilisons encore la fonction `devctl()` mais de la manière suivante :

```
ret = devctl(fd, READ_CAPTEUR, S2_CAPTEURS_BP, sizeof(S2_CAPTEURS_BP), NULL);
```

Vous constatez à nouveau que nous passons le descripteur du driver en paramètre. `READ_CAPTEUR` nous permet d'appeler la sous fonction `READ_CAPTEUR` de la fonction `devctl()`. `S2_CAPTEURS_BP` n'est rien d'autre qu'un pointeur vers une variable entière. Le driver va d'ailleurs placer les résultats des lectures des 3 registres dans ce nombre entier de la manière suivante :

L'information récupérée suite à la lecture du port A du 8255 configuré en entrée sera stockée sur les bits allant de 16 à 23. L'information contenue dans le registre du port B du 8255 configuré en entrée sera stockée sur les bits allant de 8 à 15 et l'information stockée dans le registre du port C du 8255 configuré en entrée sera stockée sur les bits allant de 0 à 7.

Le quatrième paramètre passé à la fonction `devctl()` n'est rien d'autre que la taille du pointeur de la variable entière et le 5<sup>e</sup> paramètre n'est pas utilisé c'est pourquoi nous mettons le paramètre `NULL`.

En cas d'erreur la fonction `devctl()` retourne une valeur négative.

## 1.2.2.b) Interface PROFIBUS

Pour réaliser des entrées et des sorties en PROFIBUS il faut pouvoir jongler avec 4 fonctions.

Pour récupérer les informations en provenance des capteurs, il faut dans un premier temps faire appel à la fonction suivante :

```
IO_RefreshInput(NUMCARTE, &status);
```

- ⌚ Le premier paramètre est en fait le numéro de la carte PCU-DPIO présente dans le pc de contrôle de la station 1. Dans notre cas, NUMCARTE = 1.
- ⌚ La variable status dont nous passons l'adresse, contiendra le code de retour de la fonction. Quand tout se passe bien la fonction retourne 0.

Comme vu précédemment, notre carte PROFIBUS travaille avec des mémoires tampons avant de pouvoir lire l'état des capteurs il faut d'abord rafraichir l'état de la variable correspondante avec l'information qui circule sur le bus. Une fois cela fait nous pouvons lire le contenu de la variable à l'aide de la fonction suivante :

```
IO_ReadIByte(NUMCARTE, ADRESSECPVDIO, 0, 1, S1_CAPTEURS_LSB_BP, &status);
```

- ⌚ NUMCARTE étant le numéro de carte PCU-DPIO située dans le pc de contrôle de la station 1. Dans notre cas, NUMCARTE = 1.
- ⌚ ADRESSECPVDIO n'est rien d'autre que l'adresse du module d'entrée/sortie PROFIBUS situé sur la station 1, dans notre cas, cette adresse vaut 64.
- ⌚ Le 3<sup>e</sup> paramètre vaut 0 et correspond à l'Offset à partir duquel nous souhaitons écrire.
- ⌚ Le 4<sup>e</sup> paramètre valant 1 correspond au nombre de byte à lire.
- ⌚ S1\_CAPTEURS\_LSB\_BP correspond à un pointeur pointant le byte qui va recevoir l'information lue.
- ⌚ La variable status n'est rien d'autre que la variable qui va stocker le code de retour de la fonction. Quand tout se passe bien la fonction retourne 0.

Vous avez maintenant toutes les informations pour réaliser des entrées et des sorties aussi bien via l'interface parallèle que via PROFIBUS. Voyons maintenant comment libérer les ressources.

## 1.1.3. Libération des ressources

### 1.3.1.a) Interface parallèle

Pour les stations 2, 3 & 4, nous libérons les ressources du driver à l'aide de la fonction `close()` comme c'est le cas dans l'exemple suivant :

```
close(fd);
```

`fd` étant bien évidemment le descripteur que la fonction `open()` nous a retourné.

### 1.3.1.b) Interface PROFIBUS

Il est très important de libérer les ressources du driver PROFIBUS une fois que vous aurez terminé l'exécution de votre programme. Sans quoi vous serez obligé de redémarrer la machine, car plus aucune application ne parviendra à utiliser l'API du driver. En effet, le driver PROFIBUS n'autorise pas l'utilisation de son API à plus d'une application à la fois. Il faudra donc toujours faire correspondre un `IO_Exit()` à un `IO_Init()`.

Voici un exemple de libération des ressources du driver PROFIBUS :

```
IO_Exit(NUMCARTE, &status);
```

- ⌚ NUMCARTE permet de spécifier le numéro de la carte PROFIBUS. L'adresse de notre carte est l'adresse 1.
- ⌚ Le paramètre `status` porte bien son nom, en effet, il contiendra le code de retour de l'exécution de notre fonction. Nous récupérerons la valeur 0 dans `status` si aucune erreur ne survient, une autre valeur en cas de problème.

## 1.1.4. Schémas récapitulatif

Voici 2 schémas récapitulatif permettant de résumer comment réaliser des entrées/sorties via l'interface parallèle et via PROFIBUS :

Interface parallèle

PROFIBUS

Vous avez remarqué que nous écrivons et lisons des données d'un byte en PROFIBUS. Nous disposons pourtant de 15 capteurs et 10 actionneurs, il aurait été plus avantageux de lire et d'écrire directement des mots. Malheureusement après plusieurs tests, les instructions `IO_WriteQWord()` `IO_ReadIWord()` retournaient un code d'erreur, le manque de documentation ne nous a pas permis de déterminer d'où provenait l'erreur.

Nous sommes donc actuellement obligés de lire et d'écrire de simple byte et appeler 2 fois de manière successive les fonction `IO_WriteQByte()` et `IO_ReadIByte()` avec un offset différent de manière à lire et écrire tous les capteurs et actionneurs.



## 1.1.5. Option de compilation & headers indispensables

### 1.5.1.a) Interface parallèle

Vous devez absolument inclure à votre programme le fichier header « **pioD48\_qnx\_v2.2.h** » pour pouvoir appeler la fonction `devctl()` du driver de la carte d'entrée/sortie PIO-D48.

Il n'y a pas d'option de compilation particulière à prendre en compte dans le cas de l'interface parallèle.

### 1.5.1.b) Interface PROFIBUS

Pour pouvoir utiliser l'API de programmation de la carte d'interfaçage PROFIBUS-DP, vous devrez inclure le fichier « **appio.h** » dans votre programme.

Ce n'est pas tout, toujours dans le but d'utiliser l'API de programmation PROFIBUS, lors de la compilation vous devrez linker à votre programme la librairie statique « **libappio.a** ». Pour ce faire utiliser la directive de compilation suivante :

-Bstatic libappio.a
---------------------

## 1.1.6. Câblage

Station 1				
Actuateurs	Bit	Byte 0	Byte 1	/
	0	Sortir PP1	Bras gau./droi	/
	1	Rentrer PP1	/	/
	2	Sortir PP2	Aspiration	/
	3	Rentrer PP2	/	/
	4	Sortir PP3	/	/
	5	Rentrer PP3	/	/
	6	Rent/Sort P	/	/
	7	/	/	/
Capteurs	Bit	Byte 0	Byte 1	/
	0	Pouss. rentré	Pouss. sorti	/
	1	PP1 rentré	PP1 sorti	/
	2	PP2 rentré	PP2 sorti	/
	3	PP3 rentré	PP3 sorti	/
	4	Bras droite	Bras gauche	/
	5	Prés. Pièce 1	Pièce Aspirée?	/
	6	Prés. Pièce 2	24v sur I7	/
	7	Prés. Pièce 3	Table alignée	/

La station 1 est particulière du fait qu'elle est interfacée en PROFIBUS. Les fonctions de lecture et d'écriture rendue disponible par le driver ne permettent que de lire et d'écrire un seul byte. Vous retrouverez le byte 0 à l'offset 0 et le byte 1 à l'offset 1.

Station 2				
Actuateurs	Bit	PA	PB	PC
	0	PP ON/OFF	/	Allumer Start
	1	Ascen. Desc.	Ascen. Monte	Allumer Q1
	2	Coussin Air	/	Allumer Q2
	3	/	/	Allumer Rese
	4	/	/	Allumer Q4
	5	/	/	Allumer Q5
	6	/	/	Allumer Q6
	7	/	Table alignée	Allumer Q7
Capteurs	Bit	PA	PB	PC
	0	PP rentré	/	Start enfoncé
	1	Ascen. Bas.	Ascen. Haut	Stop non enf.
	2	/	/	Reset enf.
	3	/	/	Strt/Stp/Res
	4	Présence P.	/	24v sur I4
	5	Detect P. Met.	/	24v sur I5
	6	Barrière Opto.	24v sur I7	24v sur I6
	7	Hauteur P.	Table alignée	Clé Tournée

Station 3				
Actuateurs	Bit	PA	PB	PC
	0	Car. ON/OFF	/	Allumer Start
	1	Desc. Fraise	Monter Fraise	Allumer Q1
	2	Fraise ON/OFF	/	Allumer Q2
	3	Sol. H. ON/OFF	/	Allumer Rese
	4	Sol. V. ON/OFF	/	Allumer Q4
	5	Ejecteur Pièce	/	Allumer Q5
	6	/	/	Allumer Q6
	7	/	Table alignée	Allumer Q7
Capteurs	Bit	PA	PB	PC
	0	Car. en pos?	/	Start enfoncé
	1	Fraise bas	Fraise haut	Stop non enf.
	2	Pièce Fraisée?	/	Reset enf.
	3	/	/	Start/Stop/Res
	4	/	/	24v sur I4
	5	Prés. Pièce 1	/	24v sur I5
	6	Prés. Pièce 2	24v sur I7	24v sur I6
	7	Prés. Pièce 3	Table alignée	Clé Tournée

Station 4				
Actuateurs	Bit	PA	PB	PC
	0	Rentrer Bras H.	Sortir Bras H.	Bit 0 Position
	1	Bras V. Ren/So	/	Bit 1 Position
	2	Pince Ouv/Fer.	/	Bit 2 Position
	3	/	/	Bit 3 Position
	4	/	/	Start
	5	/	/	/
	6	/	/	Amplificateur
	7	/	Table alignée	Regulateur
Capteurs	Bit	PA	PB	PC
	0	Bras H. rentré	Bras H. sorti	Bouton Start
	1	Bras V. rentré	Bras V. sorti	Bouton Stop
	2	/	/	Bouton Reset
	3	/	/	Strt/Stp/Res.
	4	/	/	Arret Urg. Tiré
	5	Ready to Op.	/	/
	6	Alim Active	/	/
	7	En Position	/	B Auto/Man

## 1.1.7. Pilotage du grappin de la station 4

Pour pouvoir déplacer le grappin de la station 4, il faut faire passer le moteur en mode de positionnement lors de la phase d'initialisation de votre programme. Pour ce faire vous devez activer l'amplificateur ainsi que le régulateur du moteur en les plaçant à 1.

Une fois que vous avez fait passer le moteur en mode de positionnement, le grappin va se calibrer et ensuite venir se placer en position 0. Le calibrage peut durer plusieurs secondes, pensez à temporiser.

A la fin de l'exécution de votre programme, n'oubliez pas de faire repasser l'amplificateur et le régulateur du grappin à 0 afin de sortir du mode de positionnement.

### A quoi correspondent les positions ?

- ⌚ Position 0 = au dessus du carrousel de la station 3
- ⌚ Position 1 = au dessus du tube 1
- ⌚ Position 2 = au dessus du tube 2
- ⌚ Position 3 = au dessus du tube 3

### Comment envoyer des ordres et faire déplacer le grappin ?

Il est possible d'enregistrer jusqu'à 16 positions dans le moteur c'est pourquoi 4 bits sont prévus pour le positionnement du grappin. Les 4 positions enregistrées (carrousel et tubes) sont codées logiquement sur les bits 0 et 1 :

Position	Bit 3 Position	Bit 2 Position	Bit 1 Position	Bit 0 Position
Position 0 – Plateau	0	0	0	0
Position 1 – Tube 1	0	0	0	1
Position 2 – Tube 2	0	0	1	0
Position 3 – Tube 3	0	0	1	1

Une fois que vous avez enregistré la position à rejoindre en positionnant les bits de position, il faut envoyer un front montant sur le bit de START pour faire effectuer le déplacement. Notez, qu'il est nécessaire d'effectuer une tempo d'environ 150ms entre le moment où vous positionnez les bits de position et le moment où vous envoyez le front montant.

## 1.2. Photon Application Builder – PhAB

### 1.2.1. Introduction

Pendant notre cursus scolaire, nous avons l’habitude de développer des interfaces graphiques en passant par des environnements de développement comme Visual Studio, Netbeans ou encore Qt Creator.

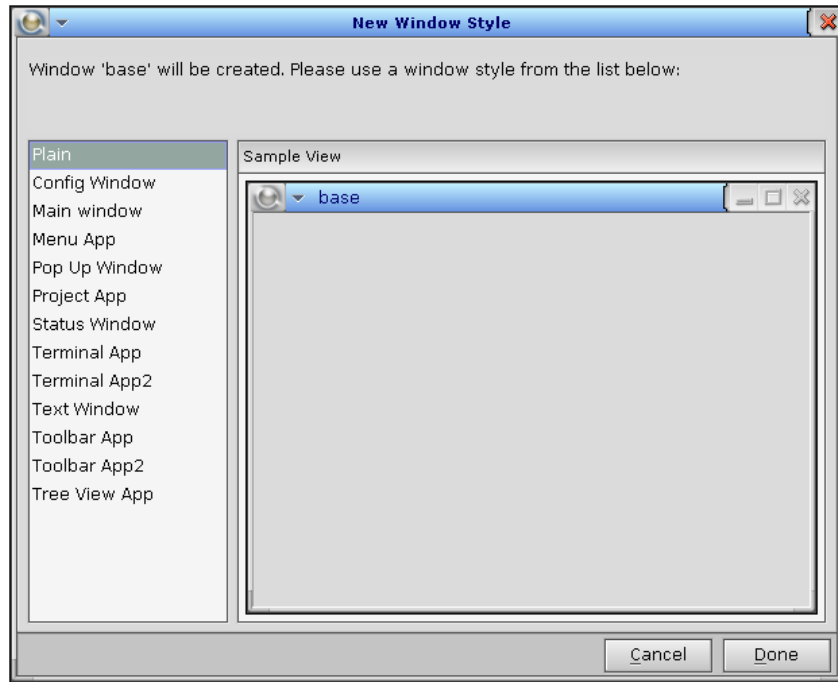
Sous QNX, aucun de ces environnements de développement ne tournent. Toutefois, il est possible de compiler Qt Creator d’une certaine manière pour qu’il puisse tourner sur la plateforme QNX.

Cependant, pour des raisons de facilité à l’école, et dans le cas où une machine doit être réinstallée nous avons préféré utiliser directement l’environnement de développement intégré à QNX : Photon Application Builder. De plus, c’était l’occasion pour moi de découvrir un nouvel environnement de développement.

Photon Application Builder est donc un environnement de développement graphique qui va nous permettre d’implémenter nos interfaces graphiques en prenant en charge le langage C ou le langage C++. Photon Application Builder est aussi un outil qui va nous permettre de gagner un temps considérable, en effet, créer une interface graphique devient un véritable jeu d’enfant alors que cela nous aurait prit beaucoup plus de temps de créer notre application graphique entièrement à la main. La gestion des Widgets, des Modules (fenêtres, dialogues,...) et des fonctions de rappels (Callback) est déjà prise en charge par défaut par l’environnement de développement. De ce fait, sans écrire une seule ligne de code, il est possible de construire une interface graphique et voir à quoi elle ressemblera.

### 1.2.2. Création d’un nouveau projet

Pour créer un nouveau projet, cliquez sur le menu « File » en haut à gauche et ensuite cliquez sur « New ». Une fenêtre pop up apparaîtra vous permettant de choisir le type de fenêtre désiré pour votre application :

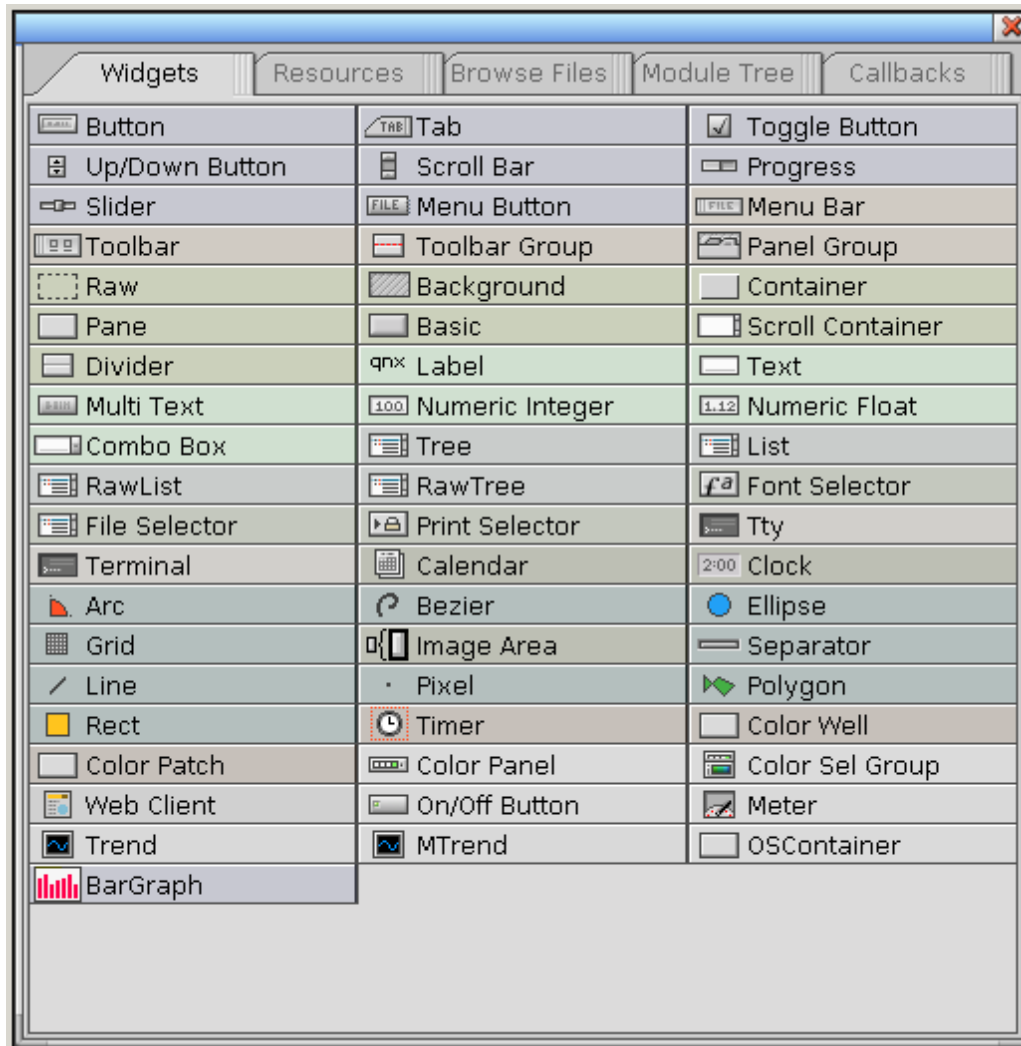


Appuyez sur « Done » pour valider votre choix.

Ensuite, quatre fenêtres vont vous permettre de personnaliser votre interface graphique. Il s'agit des fenêtres « Widgets », « Callbacks », « Resources » et « Module Tree ».

### 1.2.3. La fenêtre « Widgets »

La fenêtre « Widgets » affiche tous les Widgets mis à notre disposition et prêt à être utilisé. Il suffit de cliquer sur le Widget que l'on souhaite utiliser et ensuite de le placer dans notre interface graphique. Si la fenêtre « Widgets » n'est pas directement disponible par défaut au démarrage de Photon Application Builder, vous pouvez la faire apparaître en cliquant dans la barre de menu sur « Window », ensuite cliquez sur « Show Templates » et terminez par sélectionner « Show Widgets ».



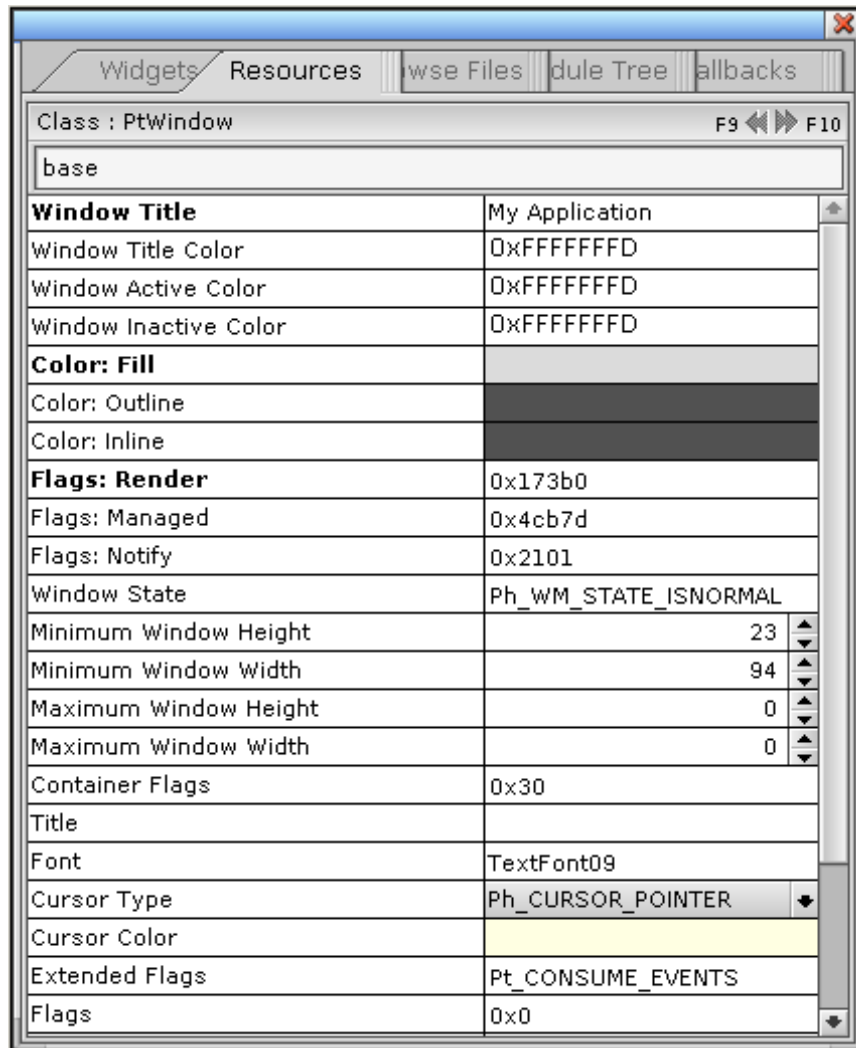


## 1.2.4. La fenêtre « Resources »

La fenêtre « Resources » permet de personnaliser entièrement vos widgets, elle correspond au menu « Propriétés » d'un widget de Visual Studio. En effet, vous pourrez y modifier la couleur, le texte, la police et de multiples autres propriétés de votre widget.

Pour savoir à quoi correspond chaque paramètre, pointez le paramètre en question avec votre souris pendant 2 secondes, un petit texte d'aide apparaîtra alors à l'écran.

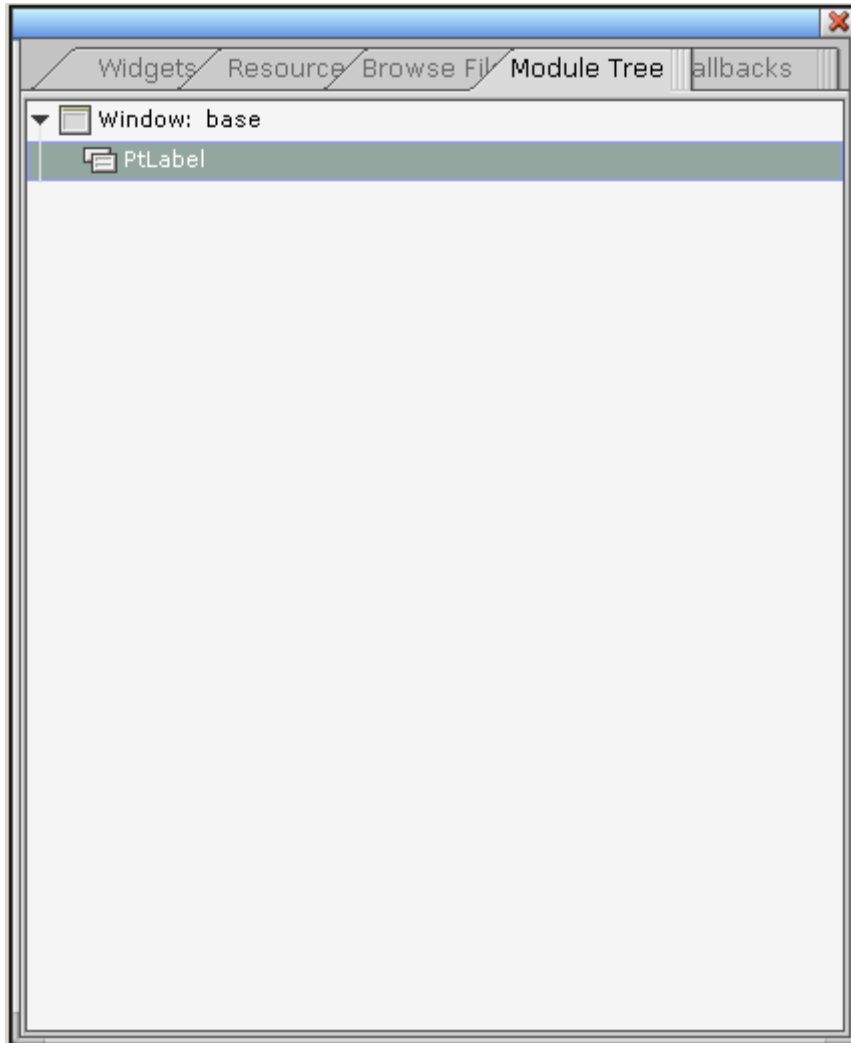
Si la fenêtre « Resources » n'apparaît pas par défaut au démarrage de Photon Application Builder, cliquez sur « Window » dans la barre de menu, ensuite cliquez sur « Show Resources » ou appuyez sur F12.



## 1.2.5. La fenêtre « Module Tree »

La fenêtre « Module Tree » affiche la hiérarchie des composants graphiques de notre interface en cours de construction. Dans le cas illustré ci-dessous, nous voyons que nous avons mis un label dans la fenêtre principal de notre application.

Si la fenêtre « Module Tree » n'apparaît pas par défaut au démarrage de Photon Application Builder, cliquez sur « Window » dans la barre de menu, ensuite cliquez sur « Show Module Tree ».



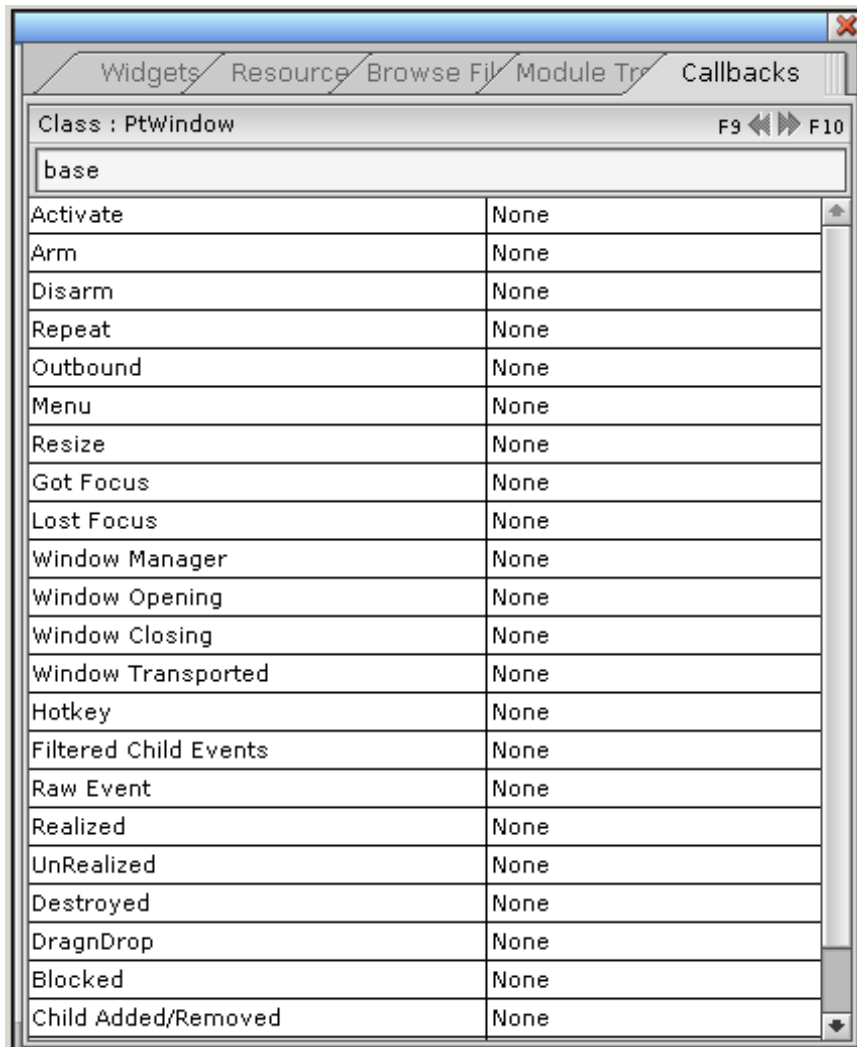
## 1.2.6. La fenêtre « Callbacks »

La fenêtre « Callbacks » nous affiche l'ensemble des fonctions de rappels possibles pour un Widget donné. C'est par l'intermédiaire de la fenêtre « Callbacks » que nous associerons du code à un évènement.

L'évènement qui nous sera le plus utile sera l'évènement « Activate » qui correspond à un clic sur un Widget. Par exemple, dans le cas d'un bouton si nous voulons exécuter du code lorsque nous cliquons sur ce bouton, nous devons référencer notre fichier c dans l'évènement « Activate ».

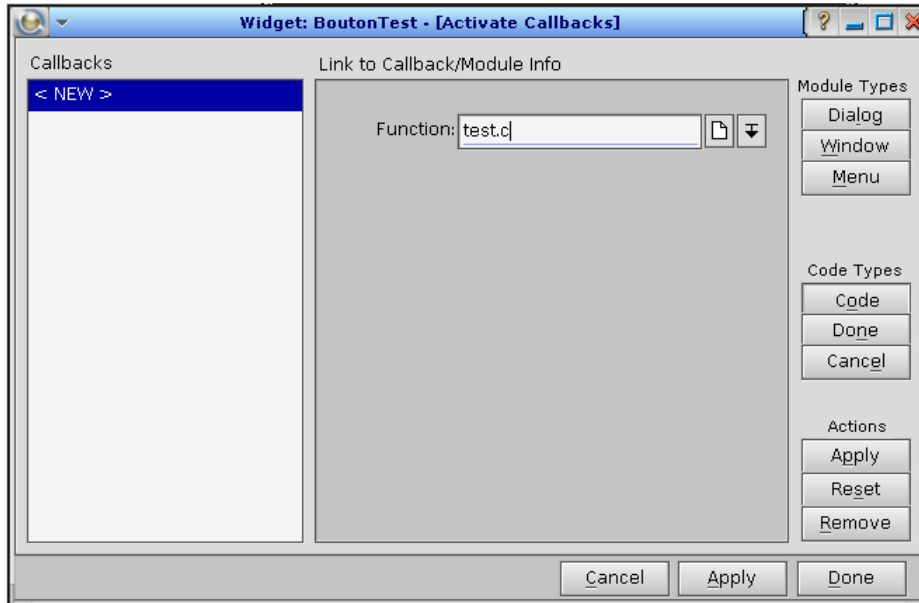
Pour connaître la signification d'un évènement ou plutôt pour savoir quand un évènement est déclenché, pointez avec votre souris l'évènement désiré dans la fenêtre « Callbacks » et un petit texte explicatif apparait au bout de 2 secondes d'inactivité.

Si la fenêtre « Callbacks » n'apparait pas par défaut au démarrage de Photon Application Builder, cliquez sur « Window » dans la barre de menu, ensuite cliquez sur « Show Callbacks » ou appuyez simultanément sur SHIFT – F12.

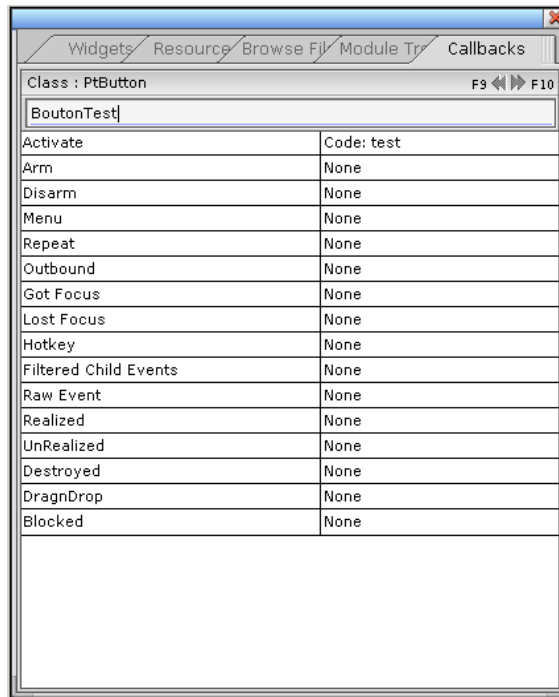


## 1.2.7. Associer du code à un évènement

Pour associer du code à un évènement, dans la fenêtre « Callbacks », il faut commencer par cliquer sur la case « None » situé en face de l'évènement que nous souhaitons implémenter. La fenêtre suivante apparait alors à l'écran :



Le fichier que nous référençons contiendra la fonction exécutée lorsque l'évènement sera déclenché. Dans l'exemple ci-dessus, il ne restera plus qu'à implémenter le corps de la fonction se trouvant à l'intérieur du fichier test.c. Le fichier et le canevas de la fonction de callback seront générés automatiquement lorsque nous appuyons successivement sur « Apply » et puis sur « Done ». Sur la capture d'écran suivante qui est capture d'écran de la fenêtre « Callbacks », vous pouvez constater que la fonction de rappel est bien prise en compte. Le type de rappel est également affiché, dans ce cas ci c'est du code, ensuite, vous trouvez le nom de la fonction :



## 1.2.8. Structure d'un projet Photon Application Builder

Si le nouveau projet créé s'appelle « mon\_projet », voici la structure de projet mise en place par Photon Application Builder :















Dans cette arborescence, seul le répertoire « src » nous intéresse. En effet, lorsque nous mentionnons une nouvelle fonction de callback, c'est dans le répertoire src que le fichier c correspondant sera généré.

De plus, si nous souhaitons ajouter une librairie personnelle à notre projet, la seule manière pour que Photon Application Builder prenne notre librairie en compte et la compile dans notre projet est de mettre ses fichiers sources (.c et .h) dans le répertoire « src ».

Nous verrons également plus tard, qu'il est possible d'utiliser un fichier header global à l'application et qu'une fonction d'initialisation peut également être appelée. Ces fichiers devront également être placés dans le répertoire src.

Dans ce répertoire « src », vous trouverez également toute une série de fichiers générés auxquels il ne faut absolument pas toucher.

Dans le cas de notre projet d'exemple, vous remarquerez la présence du fichier test.c dans le répertoire « src » de notre projet :

Name	Date modified	Type	Size
 abdefine.h	28/05/2012 17:39	H File	1 KB
 abevents.h	28/05/2012 17:39	H File	1 KB
 abimport.h	28/05/2012 17:39	H File	1 KB
 abLfiles	28/05/2012 17:39	File	1 KB
 ablibs.h	28/05/2012 17:39	H File	1 KB
 ablinks.h	28/05/2012 17:39	H File	1 KB
 abmain.c	28/05/2012 17:39	C File	2 KB
 abvars.h	28/05/2012 17:39	H File	1 KB
 abWfiles	28/05/2012 17:39	File	1 KB
 abwidgets.h	28/05/2012 17:39	H File	1 KB
 indLfiles	28/05/2012 17:39	File	1 KB
 proto.h	28/05/2012 17:39	H File	1 KB
 test.c	28/05/2012 17:37	C File	1 KB
 Uerrmsg	28/05/2012 17:39	File	2 KB

## 1.2.9. Implémentation fonction de callback

Si vous ouvrez le fichier test.c, vous tomberez sur un code similaire à celui-ci :

```
/* Your Description */
/*                               AppBuilder Photon Code Lib */
/*                               Version 2.03 */

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Local headers */
#include "ablibs.h"
#include "abimport.h"
#include "proto.h"

int test( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
    /* Ajouter votre code */

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    return( Pt_CONTINUE );
}
```

Le code qui est actuellement sous vos yeux est généré automatiquement à l'exception évidemment du commentaire « Ajouter votre code » qui indique clairement où placer votre code. Le code que vous placez à cet endroit sera donc exécuté lorsque l'évènement est invoqué.

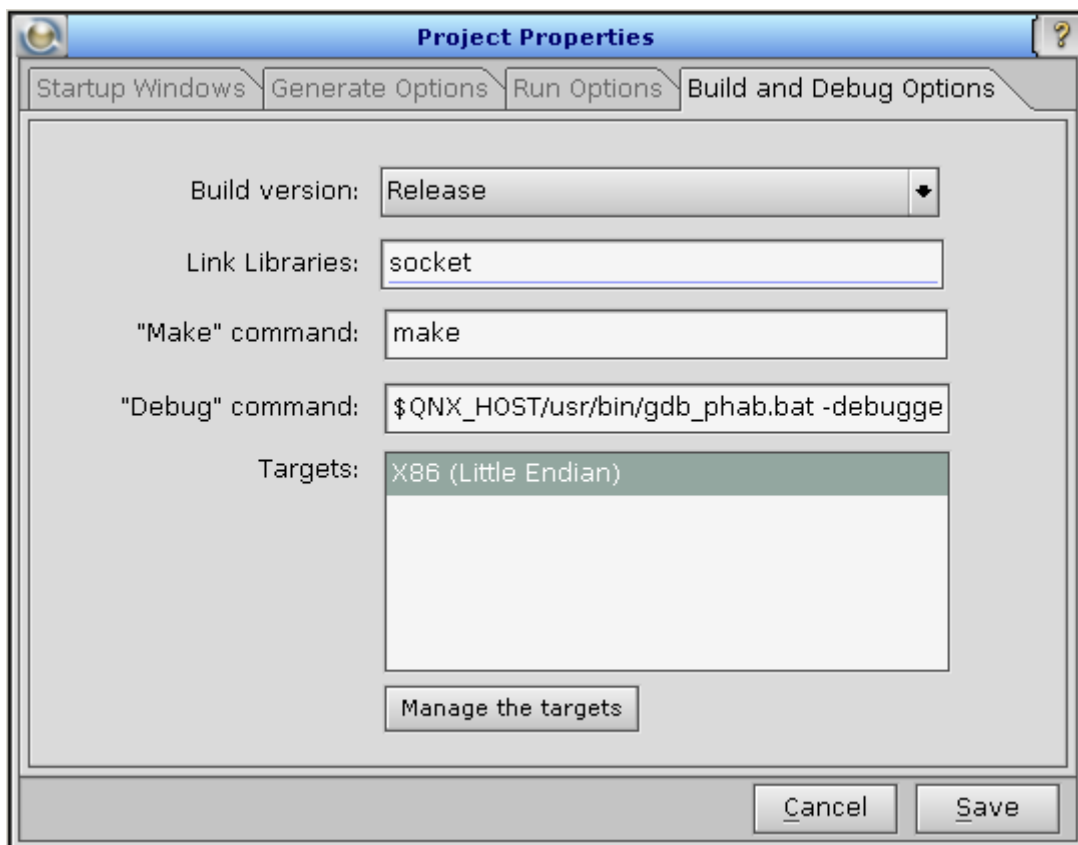
## 1.2.10. Comment linker une librairie lors de la compilation

Il se peut pour une raison ou pour une autre que vous ayez besoin de linker une librairie lors de la compilation. Par exemple, dans notre cas, nous utilisons des primitives telles que `socket()`, `bind()`, `listen()`, `accept()` ou encore `connect()`. Lorsque nous utilisons ces fonctions nous devons linker notre code avec la librairie `socket`.

Dans un premier temps, nous avons directement essayé de modifier le `makefile` généré par Photon Application Builder. Lors de l'ouverture de ce `makefile`, ce fut la surprise, des tonnes de variables d'environnement figuraient à tout va, et pas de cible dans le `makefile`. La moindre petite modification peut également faire planter la compilation de votre interface graphique.

La solution consiste à aller dans le menu « Project » et ensuite de cliquer sur « Properties ». Vous arrivez ainsi dans les paramètres du projet. Si vous vous dirigez dans l'onglet « Build and Debug Options » vous apercevrez un champ « Link Libraries ». C'est dans ce champ que nous pouvons mentionner toutes les librairies que nous souhaitons linker à notre application.

Par exemple, si nous souhaitons linker la librairie « `socket` » nous aurions du, dans un `makefile` classique, ajouter le paramètre `-l socket` à la directive de compilation, ici il faut simplement indiquer le nom de la librairie dans le champ « Link Libraries ». Dans notre exemple, nous ajouterons simplement « `socket` » :





## 1.2.11. Implémentation de l'interface graphique

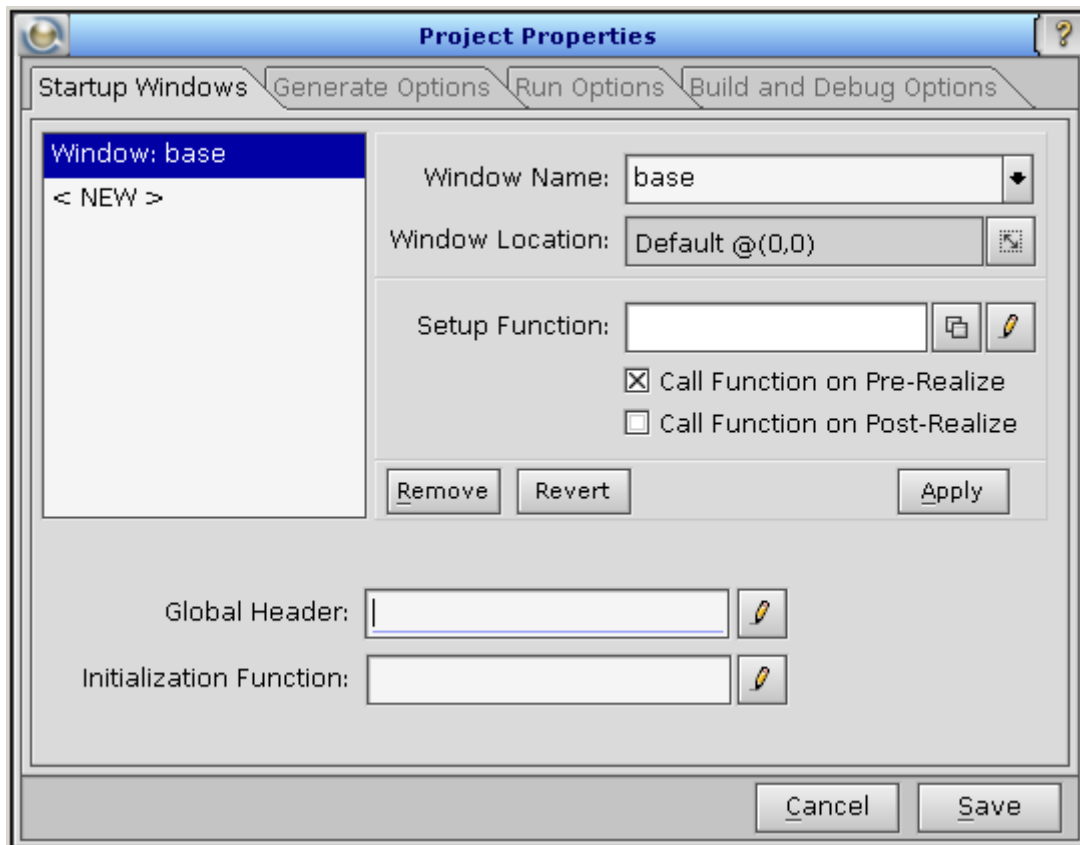
Durant notre cursus nous avons prit l'habitude de travailler avec des environnements de développement comme Visual Studio, Netbeans et Qt Creator.

Dans ces environnements de développement, notre main() instancie notre interface graphique, à partir de là nous implémentons notre interface graphique dans les différents handlers correspondant aux différents évènements.

Avec Photon Application Builder, tout cela est bien différent, il n'y a pas de fonction main() qui instancie notre interface graphique, cette dernière est directement lancée après la compilation.

Pour lancer du code en background, nous pouvons référencer une fonction d'initialisation qui sera exécutée juste avant l'instanciation de l'interface graphique.

Pour référencer cette fonction d'initialisation, cliquez sur le menu « Project », ensuite sélectionnez « Properties ». Là bas, allez dans l'onglet « Startup Windows » et dans le champ « init » tapez le nom de la fonction d'initialisation, ensuite appuyez sur le petit crayon pour générer le fichier ainsi que le canevas de base de la fonction. Vous retrouverez ce fichier dans le répertoire « src » de votre projet. Vous pouvez maintenant librement implémenter votre fonction d'initialisation à partir d'un éditeur tout à fait classique.



Vous remarquerez également la présence d'un champ « Global Header » dans ce menu. Il vous permet de créer et de référencer un fichier qui contiendra des variables globales à l'ensemble de l'application. Il faut savoir qu'à chaque fois qu'un évènement se produit, un nouveau thread est lancé pour exécuter le code de la fonction callback.

## 1.2.12. Modification des ressources par programmation

Nous avons vu tout à l'heure qu'il existait une fenêtre « Ressources » permettant de paramétrer nos Widgets. Il serait intéressant de pouvoir modifier ces ressources durant le cours d'exécution de notre programme, c'est-à-dire dans notre code.

Il faut bien être conscient que la librairie Photon est une librairie écrite en C qui n'est pas un langage orienté objet. Nous ne disposons donc pas de classe, l'exemple de code suivant n'est donc pas envisageable en Photon :

```
monLabel.Text = « Hello World » ;
```

Pour modifier une ressource d'un widget, prenons l'exemple d'un label s'appelant « monLabel », nous appellerons la fonction :

```
PtSetResource(widget, type, value, len) ;
```

Avant d'étudier les différents paramètres pris par la fonction, il faut savoir que Photon Application Builder, crée pour chaque widget un pointeur que l'on désigne par le préfixe ABW\_ suivi du nom que nous avons donné au widget. Dans le cas de notre exemple, notre label sera pointé par le pointeur ABW\_monLabel.

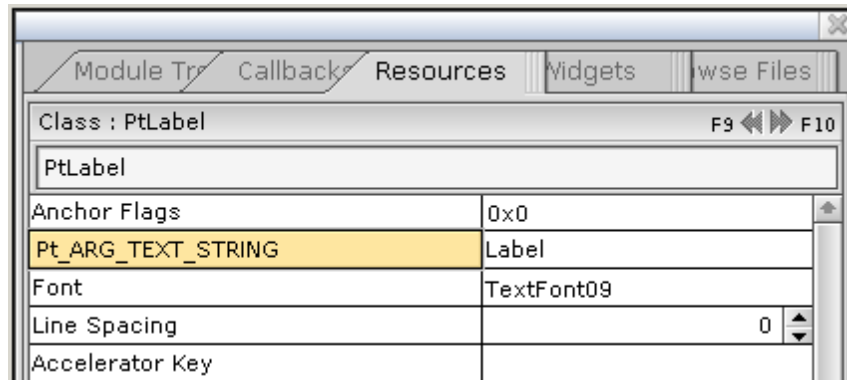
Les paramètres étant les suivants :

<i>widget</i>	Le pointeur vers le widget (ABW_monLabel)
<i>type</i>	Type de paramètre à modifier
<i>value</i>	Nouvelle valeur de la ressource
<i>len</i>	Taille du paramètre value

Afin de mieux comprendre et illustrer l'appel de la fonction, voici un exemple dans lequel nous modifions le texte affiché par notre label:

```
PtSetResource(ABW_monLabel, Pt_ARG_TEXT_STRING, « Bonjour monsieur Starzak », 0) ;
```

Pour connaître le paramètre *type* à envoyer à notre fonction PtSetResource(), allez dans la fenêtre « Ressources » et pointez le paramètre que vous aimeriez modifier avec votre souris, le paramètre à mentionner s'affichera alors à l'écran.



La fonction que nous venons de voir ne nous permet de modifier qu'une seule ressource à la fois, cependant nous pourrions être amené à devoir modifier plusieurs ressources au même moment. On pourrait penser appeler successivement la fonction `PtSetRessource()` mais cela ne serait pas très optimisé ni propre. C'est pourquoi l'API Photon met à notre disposition la fonction `PtSetResources()`. Le prototype de cette fonction est le suivant :

```
PtSetResources(PtWidget_t *widget, int n_args, PtArg_t const *args) ;
```

Les paramètres correspondant à :

<i>widget</i>	Le pointeur vers le widget (ABW_monLabel)
<i>n_args</i>	Le nombre de propriétés à modifier
<i>args</i>	Un pointeur vers une variable <code>PtArg_t</code> qui contient les changements à effectuer

Un complément d'information sur la variable de type `PtArg_t` s'avère nécessaire. `PtArg_t` est en fait une structure qui représente une propriété d'un widget. La structure `PtArg_t` est en fait constituée de 3 champs :

<i>type</i>	Type de paramètre à modifier
<i>value</i>	Nouvelle valeur de la ressource
<i>len</i>	Taille du paramètre value

Vous remarquerez que les champs de la struct `PtArg_t` correspondent aux 3 derniers paramètres que nous passons à la fonction `PtSetResource()` qui nous permettait de modifier une propriété d'un widget.

Les données de la structure `PtArg_t` peuvent être modifiées à l'aide la fonction `PtSerArg()`. Cette fonction dispose du prototype suivant :

```
PtSetArg(PtArg_t *arg, long type, long value, long len)
```

Les paramètres étant les suivants :

<i>arg</i>	Pointeur vers la variable PtArg_t que l'on souhaite modifier
<i>type</i>	Type de paramètre à modifier
<i>value</i>	Nouvelle valeur de la ressource
<i>len</i>	Taille du paramètre value

Afin de bien fixer les idées, voici un petit exemple avec un label dont on modifiera la couleur et le texte au même moment:

```
PtArg_t args[2] ;

PtSetArg(&args[0], Pt_ARG_TEXT_STRING, « Changement de texte », 0) ;
PtSetArg(&args[1], Pt_ARG_COLOR, Pg_GREEN, 0) ;

PtSetResources(ABW_monLabel, 2, &args) ;
```

En résumé, nous avons donc défini un tableau de structures PtArg\_t. Ensuite, nous faisons appel à la fonction PtSetArg() qui va nous permettre de modifier les valeurs des propriétés des widgets. Nous terminons par appeler la fonction PtSetResources() qui va opérer toutes les modifications réelles dans l'interface graphique.

### 1.2.13. Les threads en Photon

Selon la documentation Photon, les fonctions de l'API Photon ne sont pas thread-safe. Autrement dit, le seul thread qui pourrait utiliser la librairie Photon pour modifier l'affichage dans l'interface graphique, est le thread qui a appelé la fonction PtInit() et qui a donc initialisé l'application graphique.

Cela ajoute des contraintes supplémentaires à notre programmation et la rend beaucoup plus compliquée. D'autant plus qu'à chaque occurrence d'évènement, c'est un nouveau thread qui est créé. Si par malheur, deux threads venaient à utiliser au même moment des fonctions de l'API Photon, cela provoquerait le plantage immédiat de l'application.

Il existe heureusement deux fonctions qui viennent à notre secours pour résoudre ce problème. Il s'agit des fonctions PtEnter() et PtLeave(). En fait, ces fonctions nous permettent de prendre et de relâcher une sorte de mutex global qui agit sur la librairie.

En bref, si un thread est en train d'exécuter des fonctions de l'API Photon, l'appel de PtEnter() sera bloquant jusqu'à ce que ce thread fasse appel à la fonction PtLeave(). Lorsque PtEnter() réveille un thread en attente, ce dernier peut alors appeler toutes les fonctions de l'API Photon sans aucun problème jusqu'à ce qu'il relâche le verrou avec la fonction PtLeave().

En résumé, lorsque nous lançons un thread dans une application Photon Application Builder et que ce thread doit effectuer des modifications dans l'interface, il devra utiliser les fonctions de l'API Photon entre l'appel de PtEnter() et de PtLeave().

Pour bien illustrer l'utilisation de PtEnter() et PtLeave(), voici un petit exemple :

```
void *maFctThread(void *p)
{
```

```
/* Initialisation */  
[...]  
  
/* Prise du mutex */  
PtEnter(0) ;  
  
/* Modification dans l'interface graphique */  
PtSetResource(ABW_monLabel, Pt_ARG_TEXT_STRING, «Bonjour monsieur Starzak », 0 ) ;  
  
/* Relacher le mutex */  
PtLeave(0) ;  
  
[...]  
}
```