

Utiliser p1q18-algocode – Les bases

Version 1.1

Remarques :

- Les images sont réalisées dans un environnement **WindowsTM** sous **XP** et ne correspondent pas forcément à ce que vous observez si vous êtes dans un autre environnement.
- Pour utiliser **piq18-algocode**, il faut avoir quelques notions d'algorithmie et d'architecture des micro-contrôleurs (notamment des pics de chez **MicrochipTM**). Vous devez notamment savoir configurer un pic (voir la documentation **MicrochipTM** de ce pic au chapitre " **Special Features Of The CPU** "). Cette configuration dépend du schéma électronique dans lequel est utilisé le pic et de ce que vous voulez pouvoir faire avec lui. Ce tutoriel suppose que vous avez ces notions.
- En général, une info-bulle d'aide accompagne tout élément actif.

Un minimum de compétences doivent être acquises pour utiliser **piq18-algocode** aisément :

- 1) Créer un projet.
- 2) Ecrire dans un port.
- 3) Lire un port.
- 4) Effectuer une opération de masquage.
- 5) Effectuer un test.
- 6) Utiliser une temporisation.
- 7) Utiliser une boucle.
- 8) Créer et utiliser un sous-programme.
- 9) Utiliser un composant.
- 10) Utiliser une interruption.

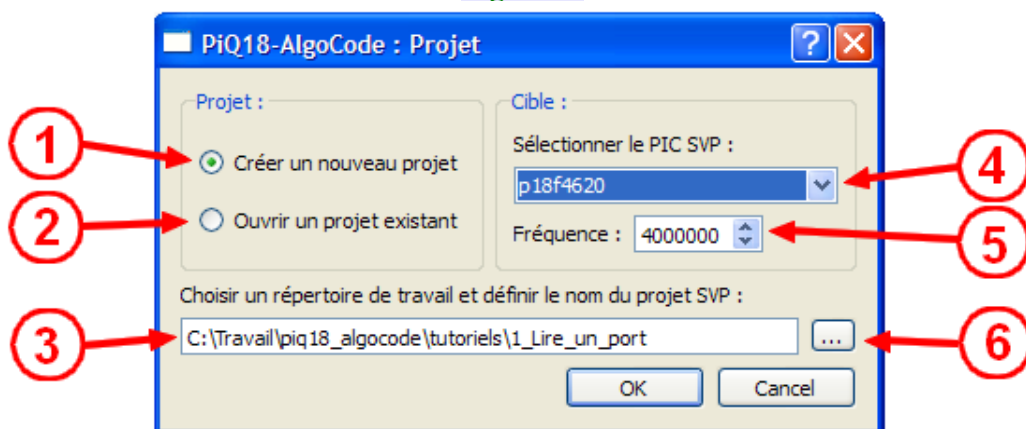
Les chapitres suivants se proposent de vous les faire acquérir.

I. Créer un projet :

La première fois, il faut utiliser **piq18-algocode** en double-cliquant sur son exécutable, ou, si vous avez fait les choses correctement à son installation, via son raccourci.

La fenêtre de la **figure 1** s'ouvre.

Figure 1.

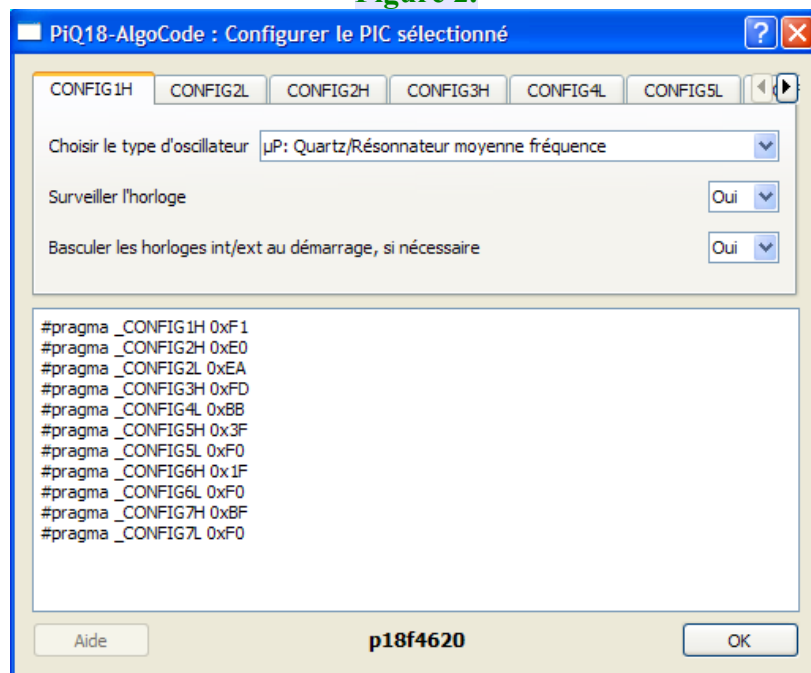


- 1) Normalement, puisqu'il s'agit de votre premier projet, vous devriez choisir l'option (1).
- 2) Ensuite vous devriez choisir les options pour (4) et (5).
- 3) Ensuite, il faut définir en (3) à l'aide de (6) le chemin où sera sauvegardé le projet ainsi que le nom du fichier projet (extension **.pac**).

→ **Remarque :** Vous aurez noté l'existence de l'option (2) qui permet d'aller rechercher un projet déjà défini.

Une fois la fenêtre de la **figure 1** validée et configurée, **pic18-algocode** s'ouvre en mettant en évidence la fenêtre de configuration du pic choisi (**figure 2**).

Figure 2.



Il vaut mieux la configurer maintenant. Si vous fermez cette fenêtre, elle vous sera proposée de nouveau au moment de la compilation pour produire le fichier **.hex**. Vous pouvez sinon le faire à tout moment via le menu " **Projet** " + " **Configurer...** ".

I.1. Remarque sur la gestion des projets :

Lorsqu'on utilise toujours le même pic, on l'insère en général toujours dans un même schéma électronique de base. La configuration est donc toujours la même ou varie très peu.

Il vaut mieux dans ce cas créer un " **projet type** " dans un " **dossier type** " (un répertoire qui contient tous les fichiers utiles) où sera enregistré un fichier projet **.pac** vierge où seule la configuration voulue aura été faite.

Ce projet sera recopié pour chaque nouveau projet en changeant simplement les noms (du dossier et des fichiers qu'il contient, au moins le fichier **.pac**).

- Il suffit en effet de double-cliquer sur un fichier **.pac** pour l'ouvrir avec **pic18-algocode**, à condition que l'extension **.pac** ait été associée à **pic18-algocode** bien sûr !).

Figure 3.

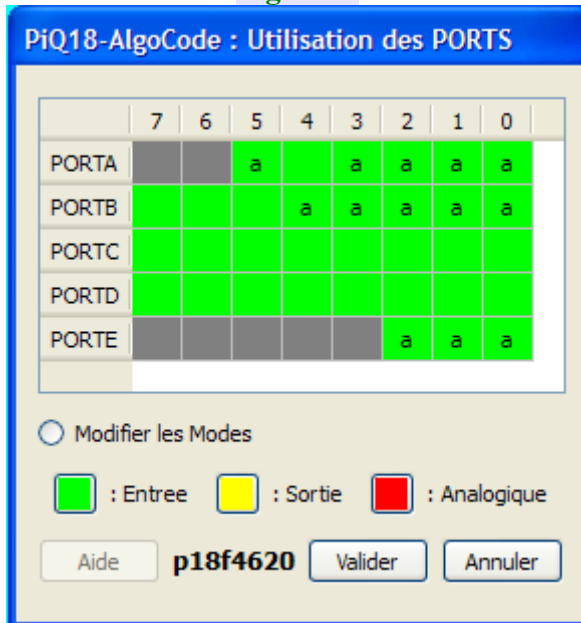
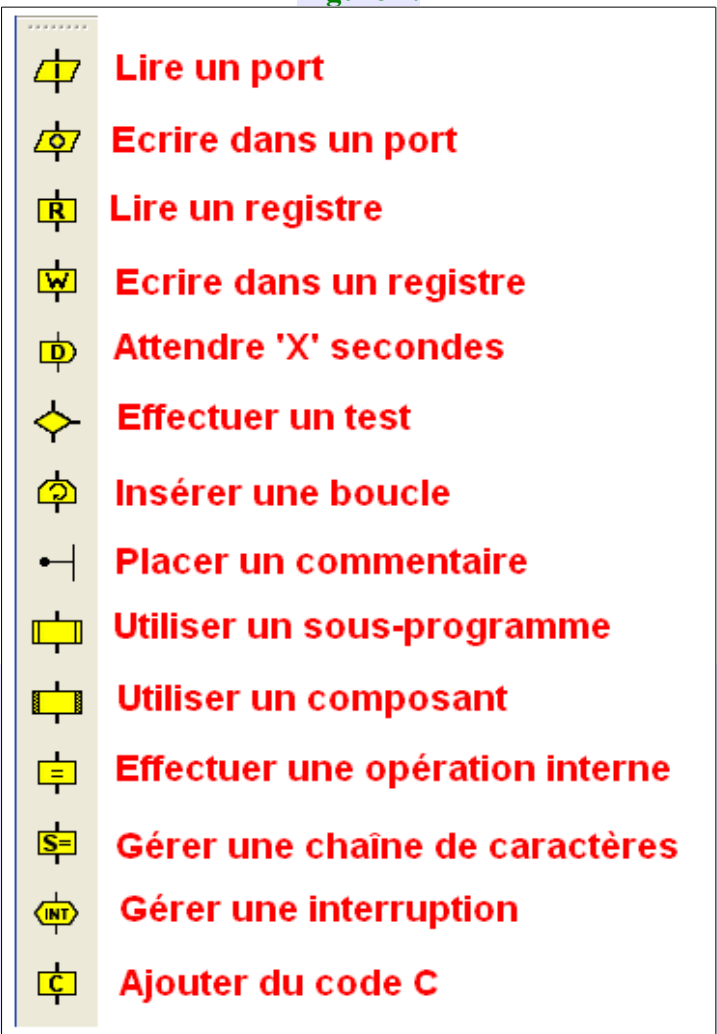


Figure 4.



Un **port** est un **registre** interne au micro-contrôleur [soit une case mémoire, de 8 bits pour les pics 18xxxxx de chez **Microchip**^(TM)] spécialisé dans les échanges avec l'extérieur. A ce titre, il est suivi d'un étage d'amplification (mais le courant reste limité à quelques mA seulement !).

Il y a des ports séries et des port parallèles. Par définition, chaque bit d'un port a un lien direct avec une des broches du pic.

On ne s'intéresse ici qu'aux ports parallèles. Ils sont visibles dans la fenêtre d'utilisation des ports (**figure 3**). Si celle-ci n'est pas visible, utiliser le menu " **Affichage** " + " **Utilisation des Ports...** ".

Notation : **RA0** est le bit 0 du **portA**, **RB3**, le bit 3 du **portB**, etc...

- ➔ Normalement, avant de commencer à programmer, il faut configurer les ports et les composants (ou les modules internes au pic qui permettent eux-aussi d'être en relation avec l'extérieur, comme le module **usart** par exemple). Les composants seront abordés plus tard.

La configuration des ports se fait via la fenêtre d'utilisation des ports (qui a donc un double-rôle : configurer les ports à l'initialisation du pic et montrer cette configuration). Pour cela, il suffit de :

- 1) Cliquer sur le bouton-radio " **Modifier les modes** ".
- 2) Choisir une couleur correspondant au mode souhaité (le mode " **Analogique** " place les broches marquée d'un " **a** " en mode **entrée analogique**).
- 3) Cliquer à l'intérieur de la fenêtre d'utilisation des ports sur les broches dont vous voulez changer le mode.
- 4) Valider la nouvelle configuration en cliquant sur le bouton " **Valider** ".

Important :

Il est à noter que la fenêtre d'utilisation des ports ne correspond qu'à la configuration que prendront les ports à l'initialisation du pic, soit peu après le démarrage du programme que vous aurez créé. Vous pouvez faire évoluer cette configuration ensuite par programmation, en utilisant un pictogramme " **Ecrire** "

dans un registre " (**fenêtre 4**) afin de modifier le registre de direction " **TRISx** " associé au port à modifier (voir documentation **Microchip**^(TM) du pic sélectionné, chapitre " **I/O Ports** ").
On peut aussi le faire à l'aide du pictogramme " **Ajouter du code C** " (cf. **figure 4**).

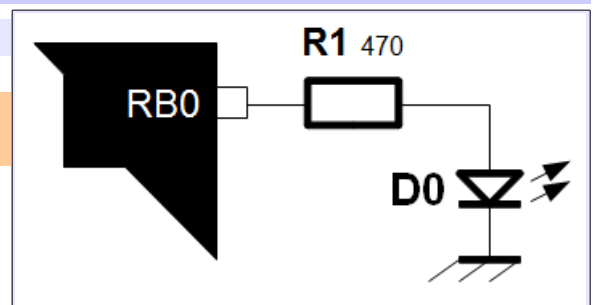
Remarques :

- Dans la fenêtre d'utilisation des ports, certaines broches sont grisées. Ceci n'indique pas que ces broches ne sont pas utilisables mais que leur mode est fixé à l'initialisation. La remarque précédente reste valable pour elles.
- Les broches **RA6** et **RA7** sont bloquées en mode sortie et entrée respectivement car ce sont les broches associées à l'horloge et, statistiquement parlant, elles ont plus de chance d'être utilisées par celle-ci (voir documentation **Microchip**^(TM), chapitre " **Oscillator Configurations** ").
- La broche **RE3** est bloquée en entrée, car dans la plupart des pics, elle est dans ce seul mode car multiplexée avec la patte **/MCLR**. Il y a certains pics cependant où c'est une autre patte qui est utilisée pour **/MCLR** (**RG5** par exemple) et elle peut être alors bidirectionnelle. Mais l'analyse de tous les cas est trop compliquée à faire.
- Les broches **RB6** et **RB7** sont de même associées au circuit de **programmation/débogage** du pic. Comme elles peuvent ne pas être utilisées à cette fin, elles ont été laissées libre de configuration à l'initialisation. Mais il faut se rappeler que si l'on souhaite utiliser cette possibilité, on doit faire attention au mode choisi pour elles (voir la documentation **Microchip**^(TM) du pic choisi, chapitre " **Special Features of the CPU** ", sous-section " **In-Circuit Debugger** ").

II. Ecrire dans un port :

Exercice : Allumer la led **D0** branchée sur la broche **RB0**.

→ Par sécurité, il vaut mieux laisser les broches inutilisées en entrée.



- 1) Configurer **RB0** en sortie (voir la procédure plus haut et **figure 5, points 2 à 4**).
- 2) **Pic18-algocode** présente un onglet nommé " **init_cpu** " (**figure 5, point 1**). Il correspond au sous-programme d'initialisation du pic. C'est ici normalement qu'il faut placer toutes les actions de configuration du pic à réaliser au démarrage du programme.
- 3) Dans la barre des pictogrammes, choisir le pictogramme " **Ecrire dans un port** " (cf. **figure 4**) et le placer dans le programme (clic gauche sur la flèche de liaison qui relie la bulle " **Début** " à la bulle " **Fin** ", **figure 5, point 5**).
- 4) Double-cliquer sur ce pictogramme, ce qui ouvre sa fenêtre de configuration.
- 5) Puisque l'on veut allumer la led dans la configuration du schéma électrique donné plus haut, il faut placer un " 1 " dans le bit 0 du **portB**. Ceci est fait en **figure 5**, par les **points 7 à 9**.
- 6) Valider en cliquant sur le bouton " **Ok** ".

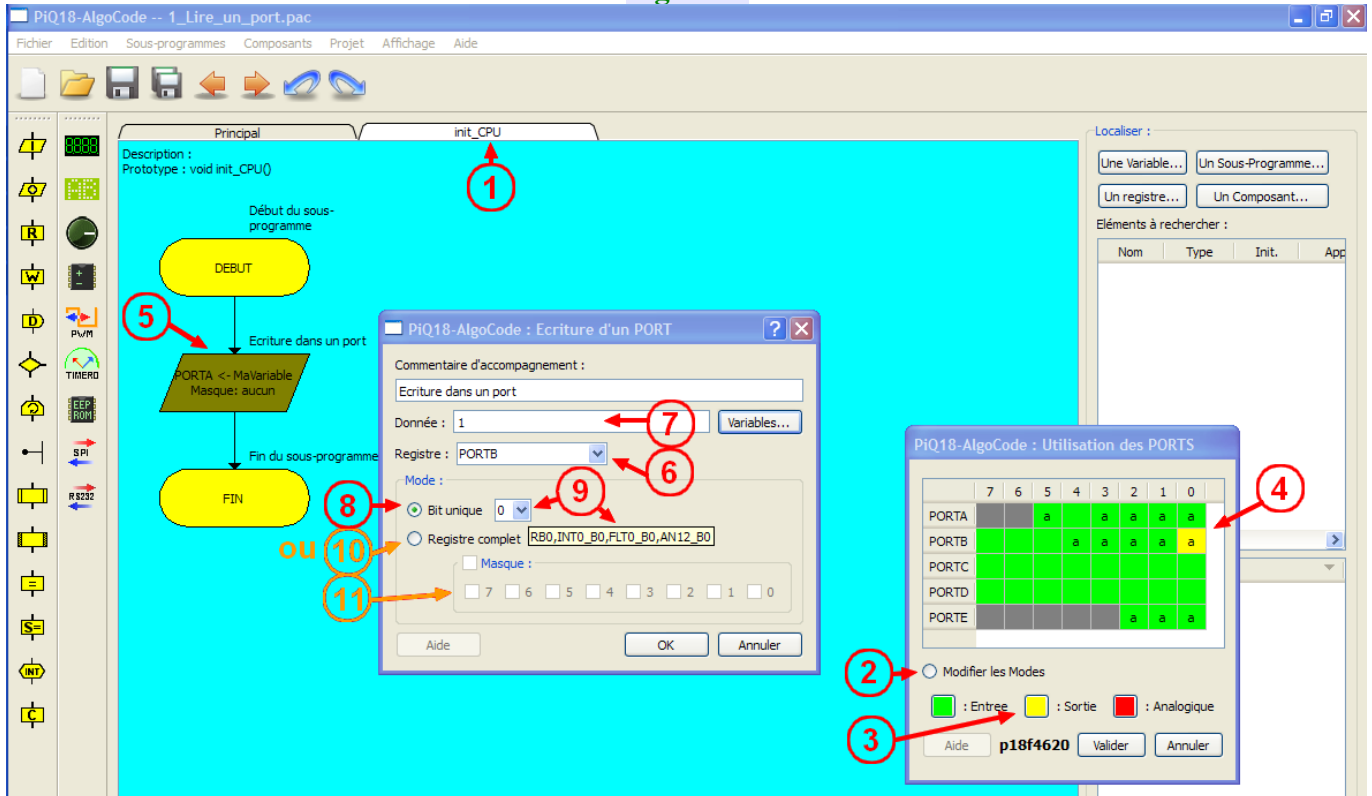
Remarques :

- En restant un moment sur (9), une info-bulle s'ouvre qui indique tous les noms de la broche sélectionnée en mode " **Bit unique** ".
- On aurait pu utiliser le bouton (10), auquel cas, la valeur tapée en (7) aurait été envoyée sur tous les bits du **portB**. Ceci est à éviter car, en général, les autres broches d'un port sont utilisées et donc ceci a des conséquences sur le fonctionnement du montage (ici la led **D0** se serait bien allumée, mais toutes les autres broches auraient été placées à " 0 " car, en binaire sur 8 bits, la valeur " 1 " (celle que l'on écrit dans le **portB**) s'écrit " 00000001 ". Il vaut donc mieux toujours utiliser (9) lorsqu'un seul bit d'un port est à affecter.
- A noter qu'au lieu de travailler en **décimal**, on peut travailler directement en **binaire** ou en **hexadécimal** (par exemple la valeur **décimale** " 123 " correspond à " 0b01111011 " en binaire et

" 0x7B " en hexadécimal). Le binaire est idéal quand on travaille avec des ports ou des registres car ceux-ci fonctionnent au niveau du bit (chaque bit a un rôle précis).

- Lorsqu'on a plus d'un bit à changer en même temps, utiliser (10) et (11) au lieu de (9). Les masques (11) servent à dire au compilateur d'écrire (ou de lire) la valeur 8 bits sans tenir compte des cases qui ne sont pas cochées : seules les broches du port dont les cases sont cochées en (11) seront affectées par le donnée placée en (7).

Figure 5.



→ Il faut maintenant compiler le projet : menu " **Projet** " + " **Compiler vers HEX...** " (ou touche " **F5** ").

Remarque : Certains pics, comme le **p18f4620** utilisé en exemple, ont des ports qui peuvent être utilisés dans des modes spéciaux, le mode **PSP** par exemple. Pour les activer en mode " **normal** " (c'est à dire en mode où chacun de leur bit est soit une entrée ou soit sortie logique), il faut alors effectuer des manipulations supplémentaires (et manuelles) sur certains registres.

- Par exemple, le **p18f4620** a son **port D** défini en mode **PSP** par défaut. Pour le passer en mode normal, il faut toucher au bit 4 du registre **TRISE** pour le placer à " 0 ". Ceci peut se faire au niveau du sous-programme " **init_CPU** " en utilisant un pictogramme " **Ecrire dans un registre** " (cf. **figure 4**) qui se manipule exactement comme celui qui vient d'être utilisé.
- La lecture de la documentation du pic que vous avez sélectionné est nécessaire pour déterminer si ce genre de manipulation est utile ou non. Il y a trop de pics pour que **piq18-algocode** puisse tenir compte de tous ces problèmes particuliers.

tutoriel 1.1.odt

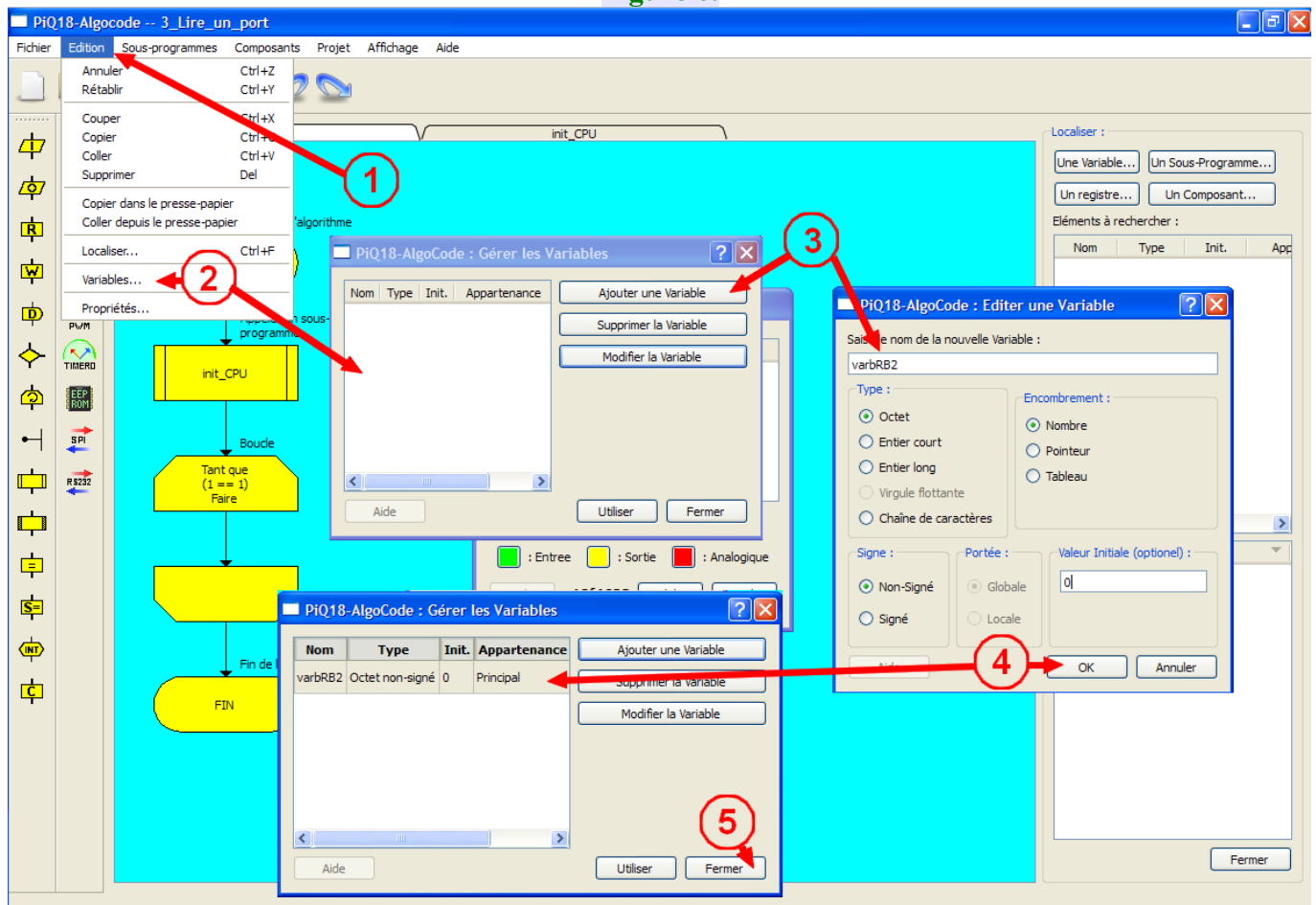
~~var~~
~~var1~~
~~var12~~
MAUVAIS

var01
var02
var11
VALABLE

III.2. Définir varbRB2 :

Voir la **figure 6**.

Figure 6.



- 1) Dans le menu " **Edition** ",
- 2) Cliquer sur " **Variables...** ".
- 3) Une fenêtre de gestion des variables s'ouvre. Cliquer sur le bouton " **Ajouter une variable** ".
Une deuxième fenêtre s'ouvre qui sert à éditer les variables. Remplir les champs comme souhaité.
Ici : **nom** : varbRB2, **type** : octet non-signé, **portée** : globale et **valeur initiale** : 0.
Valider en cliquant sur le bouton " **OK** ".
- 4) La première fenêtre se met à jour et on peut y vérifier que les informations sont correctes. Sinon utiliser le bouton " **Modifier la variable** " et reprendre la procédure d'édition.
- 5) Fermer cette fenêtre.

Remarques :

- La définition de la portée d'une variable est très importante. Le compilateur **cpik** utilisé par **piq18-algocode** suit le standard ANSI C. Rapidement, cela signifie qu'il effectue tous les calculs intermédiaires dans la " **pile** " (zone spéciale de mémoire temporaire), ce qui permet d'économiser la taille mémoire **RAM**.
Une variable **locale**, par définition, est " locale au sous-programme en cours d'utilisation ". C'est à dire qu'elle ne peut être vue qu'à l'intérieur de ce sous-programme. Elle n'existe que le temps où ce sous-programme est exécuté puis " disparaît ". Une variable **locale** est stockée dans la **pile**.
Une variable globale quand à elle peut être vue à tout endroit du programme et est choisie **globale** pour cela. Elle ne peut donc être stockée dans la **pile**, et est conservée du début à la fin de l'exécution du programme. Elle occupe donc de la place même si elle n'est pas utilisée.
A moins qu'une variable soit destinée à être vue par tous les sous-programmes ou dans le programme principal, il vaut donc mieux travailler en portée locale.
- Au cours de l'utilisation des variables, **piq18-algocode** fait une vérification stricte sur leur type. Celles-ci ne doivent en effet pas être manipulées n'importe comment sous peine d'obtenir des résultats faux ou des erreurs lors de l'étape de compilation. Il faut donc faire très attention à bien choisir le type de vos variables.
- Si on ne donne pas de valeur initiale à une variable, celle-ci est assignée automatiquement à une valeur par défaut. Les variables sont donc toutes initialisées au démarrage du programme ou d'un sous-programme. C'est pourquoi un sous-programme d'initialisation des variables est inutile.

III.3. Type des tableaux et des éléments de tableaux :

Lorsque vous définissez une variable de type **tableau** (ensemble ordonné d'éléments de même type), le type choisi est celui de ses éléments. Après sa création, cette variable peut être utilisée pour obtenir un des éléments du tableau ou comme un pointeur sur le premier élément du tableau qu'elle représente.

Par exemple, si vous définissez la variable **tab** comme un tableau d'entiers non-signés de taille 2, alors cette variable est indiquée en tant que **tab[2]** dans toutes les listes de variables disponibles.

Vous pouvez alors :

- Utiliser les éléments de ce tableau, soit **tab[0]** ou **tab[1]** (le premier élément est toujours associé à l'indice 0, le suivant à l'indice 1, etc...). Chaque élément est alors du type défini lors de la déclaration du tableau, soit ici **Entier Non-Signé**. **Remarque :** Normalement, **piq18-algocode** vérifie les indices de tableau et indique une erreur s'ils sont non-valides.
- Utiliser le tableau lui-même comme un pointeur sur son premier élément. Pour cela : utiliser le nom du tableau sans lui accoler de crochets (et donc pas d'indice). Le type qui lui est donné est alors celui de sa déclaration mais considéré alors comme un pointeur (→ajout d'un " * " à la suite de son type dans le langage C).

III.4. Solution au problème posé :

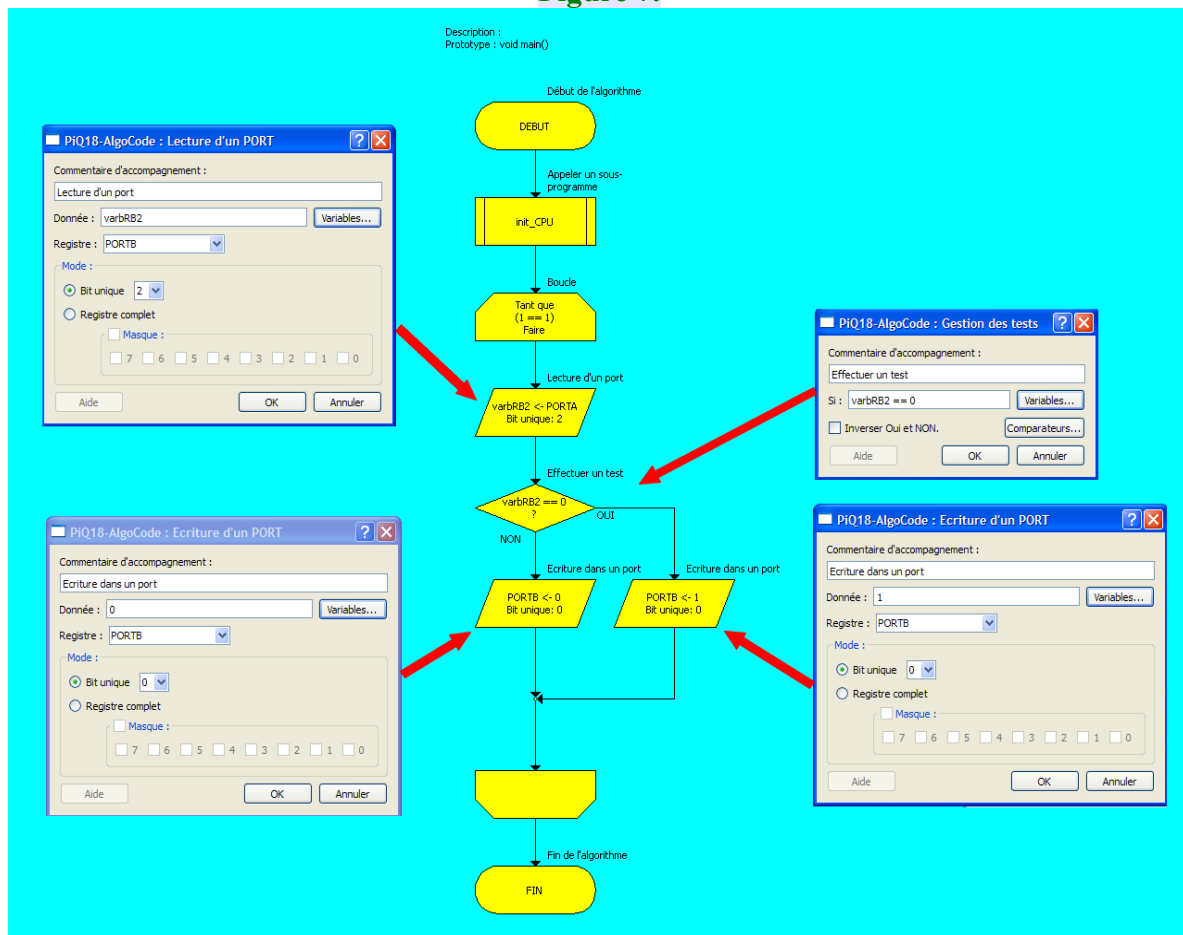
Maintenant que la variable **varbRB2** est définie, on peut s'attaquer à la solution du problème posé : allumer une led sur demande.

L'état du bouton-poussoir **BP2** doit être scruté régulièrement. Aussi, on ne doit plus placer le code dans le sous-programme d'initialisation (**init_cpu**). Ce code sera forcément placé dans la boucle sans fin du **programme principal**. Celui-ci devra donc (1) lire l'état du bouton-poussoir **BP2** puis (2) selon ce dernier, allumer ou non la led **D0**. Il y a donc une lecture de port (objet de ce chapitre – mais la moitié à déjà été fait), un test et une écriture dans le **portB** à faire.

Il suffit tout simplement de choisir les pictogrammes adéquats, de les placer et de les configurer correctement. La procédure est donc celle déjà vue pour l'écriture dans un port mais en l'adaptant à d'autres pictogrammes.

La solution finale est donnée en **figure 7**. (→ Image réalisée dans sa partie centrale avec l'exportation graphique de **piq18-algocode** accessible via le menu "Fichier" + " Enregistrer une image (.png) ").

Figure 7.



IV. Effectuer une opération de masquage :

Un masque permet de ne tenir compte que d'un certain nombre de bits lorsqu'on effectue une opération de lecture ou d'écriture dans un registre ou un port.

Avec **piq18-algocode**, c'est une opération très simple à effectuer puisqu'il suffit de choisir, dans la configuration des pictogrammes qui correspondent à ces opérations, le mode "**Masque**" et de cocher les bits dont on veut tenir compte.

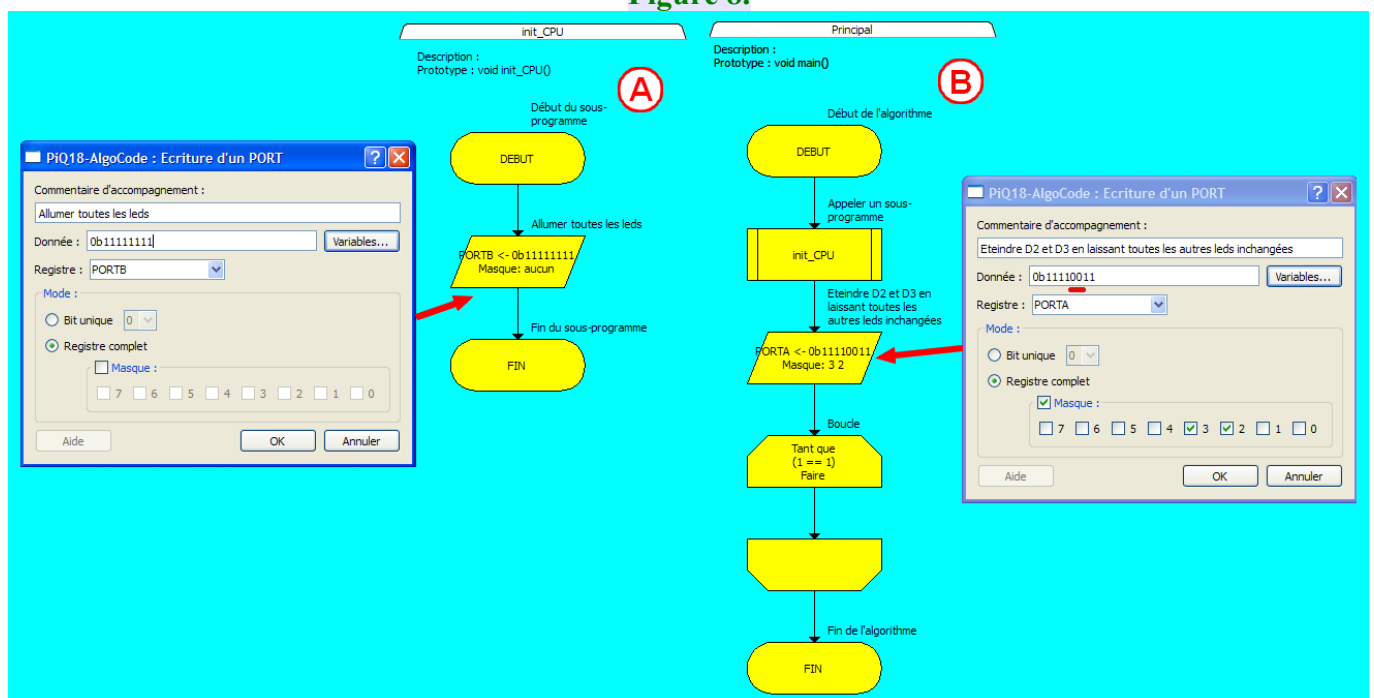
Exercice : Sur le **portB**, placer 8 leds à piloter indépendamment les uns des autres (→ led **D_i** sur broche **RBi**). La led **D₀** indique que le pic est en cours de fonctionnement sur **RB₀** et les autres leds doivent pouvoir être allumées ou éteintes à volonté en cours de déroulement du programme (schéma de câblage des leds semblable à celui utilisé au chapitre "**Ecrire dans un port**").

Comme vu précédemment au chapitre "**Ecrire dans un port**", la led **D₀** sera allumée au tout début du programme dans le sous-programme "**init_CPU**". On en profitera, afin de raccourcir l'exercice, pour allumer toutes les leds. On souhaite ensuite dans le programme principal éteindre les leds **D₂** et **D₃**. Pour cela, on ne peut tout simplement pas écrire la valeur "0" dans le **portB** puisque cela aura pour conséquence d'éteindre également la led **D₀** ainsi que toutes les autres leds.

→ Ce n'est pas ce que l'on veut puisque **D₀** doit rester allumée tout le temps que le pic fonctionne.

On procédera donc comme suit :

Figure 8.



Remarques :

- 1) Lorsqu'on travaille sur des registres (un port est un registre particulier) où les bits ont chacun leur rôle, il vaut mieux travailler en binaire comme cela a été fait ici.
- 2) Dans la donnée de la **partie B de la figure 8**, les "1" ne sont pas pris en compte. On aurait pu les remplacer tous, ou seulement certains par des "0" sans changer le résultat final.
- 3) Si on avait voulu allumer la led **D₂** et éteindre la led **D₃**, il aurait fallu remplacer les "00" de cette donnée par "01".

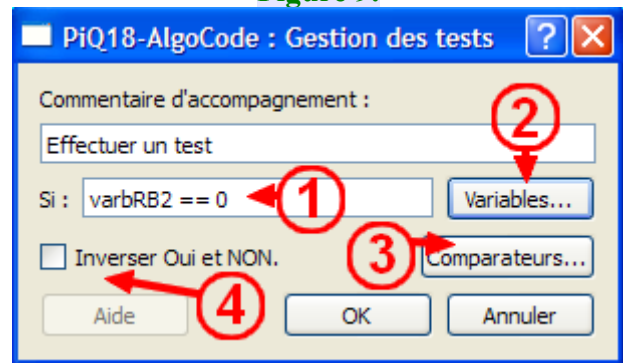
V. Effectuer un test :

Un test a déjà été utilisé dans le chapitre " Lire un port ".

Dans **pic18-algocode**, le test à effectuer est à entrer à l'emplacement (1) de la **figure 9**, et doit obligatoirement avoir la structure suivante :

- 1) " **variable1** **comparateur** **variable2** " ou
 - 2) " **variable** **comparateur** **valeur** ".
- C'est à dire que le premier élément d'un test doit toujours être une **variable**. Le bouton (2) peut aider dans ce cas.
- La variable ou la valeur à droite du comparateur doit être compatible avec le type de la variable placée à gauche du comparateur.

Figure 9.



La liste des comparateurs disponible peut être obtenue en cliquant sur le bouton (3). La liste des comparateurs est donnée en **figure 10**. C'est celle du langage C. Vous noterez la notation du test d'égalité (" == " et non " = " simplement !).

En cas d'erreur sur l'utilisation d'un test, on peut inverser les branches " **oui** " et " **non** " à l'aide du bouton (4).

Figure 10.

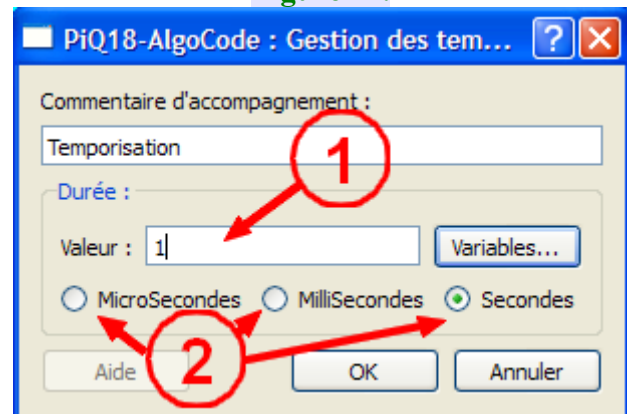
Test	Description
==	Egal ?
!=	Inégal ?
>	Supérieur ?
>=	Supérieur ou égal ?
<	Inférieur ?
<=	Inférieur ou égale ?

VI. Utiliser une temporisation :

Les temporisations servent à faire perdre du temps au pic. On le fait tout simplement " tourner en rond " un certain nombre de fois. Il ne peut donc rien faire d'autre lorsqu'on a lancé une temporisation (à moins d'utiliser les interruptions – voir plus loin – mais la temporisation risque alors d'être moins précise).

Il suffit de glisser le pictogramme " **Attendre 'X' secondes** " (cf. **figure 4**) dans le programme et de la configurer (cf. **figure 11**).

Figure 11.



Remarques :

- La durée (1) peut être définie soit par une valeur ou une variable et doit être de type compatible avec un entier court non-signé (soit une valeur entière comprise entre 0 et 65535).
- Le premier bouton de la gamme (2) s'adapte à la fréquence du pic. Il n'est pas rare de voir le choix " **x10uS** " ou " **x100uS** " s'afficher au lieu de " **microsecondes** " lorsqu'elle est peu élevée. Par exemple, si la gamme choisie est " **x100uS** " et que le chiffre entré en (1) est 5, alors la durée de temporisation sera de 5 x 100µS soit 500µS.
- Il est impossible de réaliser un code qui donne des temporisations très précises, car celles-ci sont basées sur des boucles et que le compilateur n'a pas toujours le même comportement en fonction de la valeur entrée et du nombre de boucles générées. Aussi, ne vous étonnez pas de voir des durées de temporisation précises à ±20% (l'erreur engendrée est en générale plus grande lorsque la durée de la temporisation est faible) !
- Si on veut vraiment une précision absolue, il faut utiliser les modules " **timer** " intégrés dans tous les pics (voir la documentation **Microchip**^(TM)).

Exercice : Faire clignoter la led **D0** de l'exercice du chapitre " **Ecrire dans un port** " à la fréquence de 1Hz.

La configuration est la même que pour cet exercice.

La solution est simple : dans la boucle infinie du programme principal, (1) allumer la led **D0**, (2) attendre 500ms, (3) éteindre la led **D0** puis (4) attendre encore 500ms.

La **figure 12** montre la solution. Un " **copier-coller** " peut être utilisé pour réduire les réglages à effectuer.

VII. Utiliser une boucle :

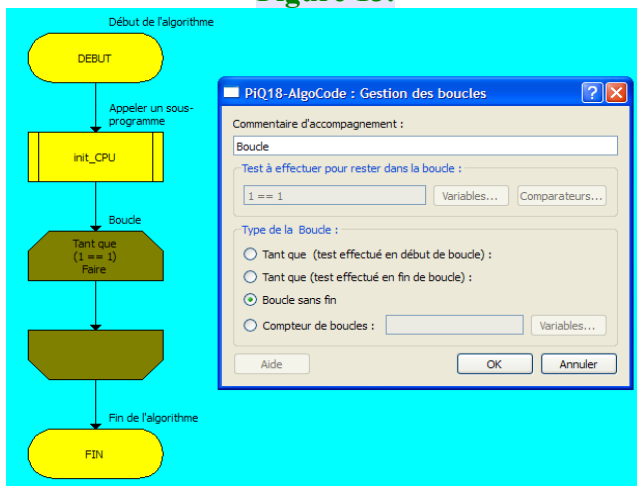
Les boucles sont les éléments les plus puissants d'un langage de programmation, qu'il soit graphique ou non.

Elles consistent à répéter un nombre de fois donné l'exécution des actions qu'elles contiennent, ce nombre étant imposé une fois pour toute ou déterminé par le résultat d'un test. Dans ce cas, tant que ce test est vérifié, le pic continue à exécuter les actions incluses dans la boucle.

piq18-algocode gère quatre types de boucles.

VII.1. Boucle sans fin :

Figure 13.

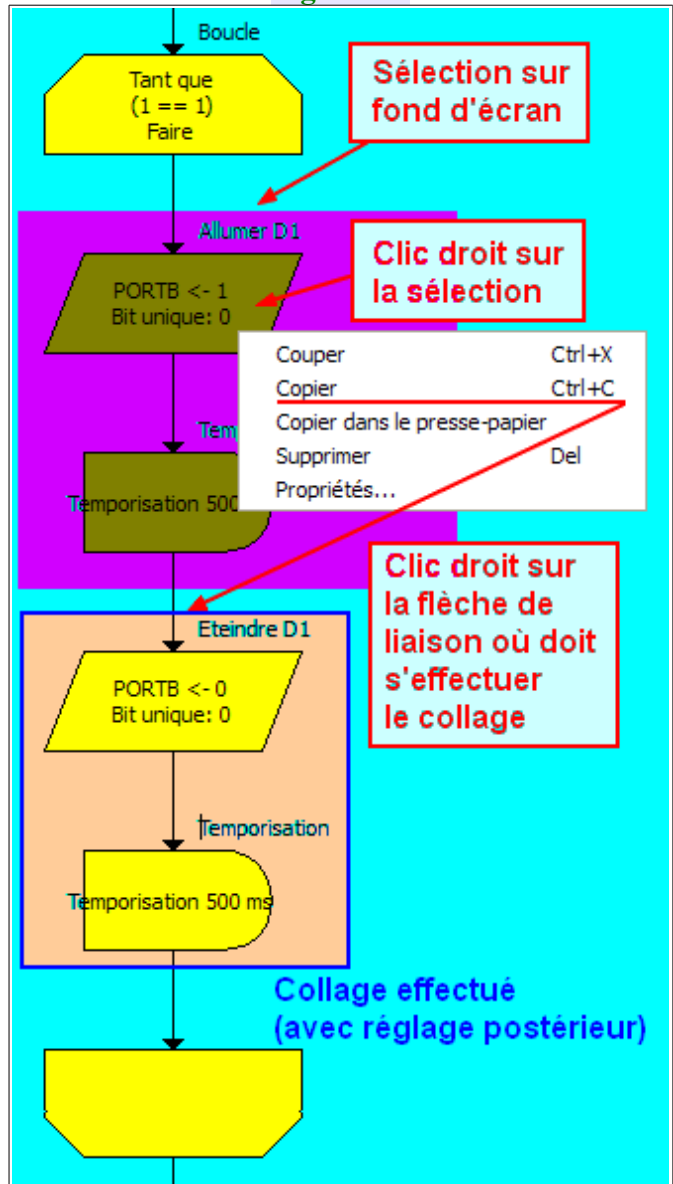


Elle permet, comme son nom l'indique, d'effectuer sans fin les actions qu'elle encadre.

Son but est souvent d'empêcher le programme d'atteindre la bulle " Fin " du programme principal. En effet, un microcontrôleur ne peut jamais s'arrêter, sauf dans le mode spécial de veille (ou sommeil), s'il existe (c'est le cas des pics).

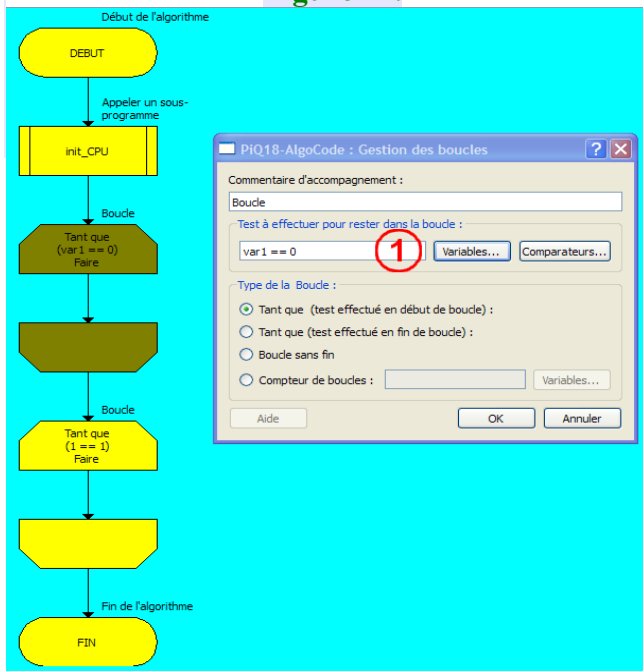
→ Il n'y a aucun réglage à faire.

Figure 12.



VII.2. Boucle " Tant que " avec test effectué en début de boucle :

Figure 14.



Dans ce type de boucles, un test est effectué au tout début de la boucle.

Tant que son résultat est vrai, les actions encadrées par la boucle sont exécutées, sinon, le programme saute au pictogramme situé juste après la boucle.

- Les actions insérées dans la boucle peuvent donc ne pas être exécutées.

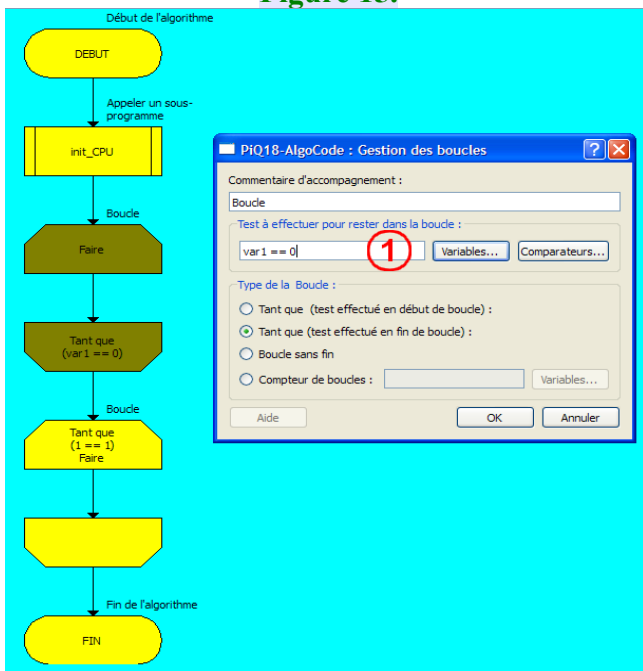
Le test est à entrer en (1).

Il doit avoir la structure :

- " **variable comparateur variable** " ou
- " **variable comparateur valeur** ".
- Tous les tests dans **piq18-algocode** doivent commencer par une variable.

VII.3. Boucle " Tant que " avec test effectué en fin de boucle :

Figure 15.



Dans ce type de boucles, un test est effectué à la fin de la boucle.

Tant que son résultat est vrai, les actions encadrées par la boucle sont exécutées à nouveau, sinon, le programme saute au pictogramme situé juste après la boucle.

- Les actions insérées dans la boucle sont toujours exécutées au moins une fois.

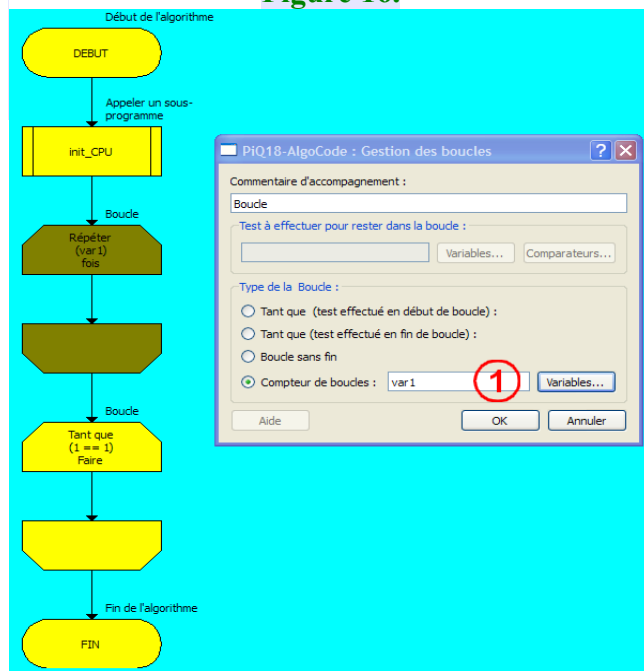
Le test est à entrer en (1).

Il doit avoir la structure :

- " **variable comparateur variable** " ou
- " **variable comparateur valeur** ".
- Tous les tests dans **piq18-algocode** doivent commencer par une variable.

VII.4. Boucle " Compteur " :

Figure 16.



Dans ce type de boucles, les actions encadrées par la boucle sont exécutées un nombre de fois précis, puis, le programme saute au pictogramme situé juste après la boucle.

- Ce nombre doit être entier, positif ou nul.
- Si le nombre vaut 0, les actions insérées dans la boucle ne sont pas exécutées.
- **pic18-algocode** limite la valeur du compteur à 65535 (si une variable est utilisée, le type compatible est au plus " Entier court non-signé ").

La valeur à donner au compteur est à entrer en (1). Entrer une variable est possible également :

- "valeur" ou
- "variable".

Exercice : Pour le branchement de l'exercice du chapitre " **Ecrire dans un port** ", faire clignoter la led **D0** à 1hz, 8 fois seulement.

- Il suffit de reprendre la solution établie pour faire clignoter la led **D0** continuellement. Le plus simple est de l'insérer dans une boucle de type " **Compteur** ", dont la valeur sera fixée à 8. Pour empêcher le programme d'atteindre la bulle " **Fin** " du programme principal, on gardera la boucle sans fin placée initialement à l'ouverture d'un projet (cf. figure 17, réalisée avec l'exportation graphique de **piq18-algocode** accessible via le menu "Fichier" + " **Enregistrer une image (.png)** ").

VIII. Créer et utiliser un sous-programme :

Le rôle du programme principal consiste normalement à distribuer le travail à effectuer à chaque sous-programme suivant sa " spécialité ". Ce n'est que " l'organisateur " ou le " chef d'orchestre ".

En effet, un projet peut être volumineux. On ne peut pas créer le code en une seule fois. Il vaut mieux décomposer le problème en " petits morceaux " spécialisés dans une tâche à accomplir et les appeler au moment opportun. Ce sont les **sous-programmes**.

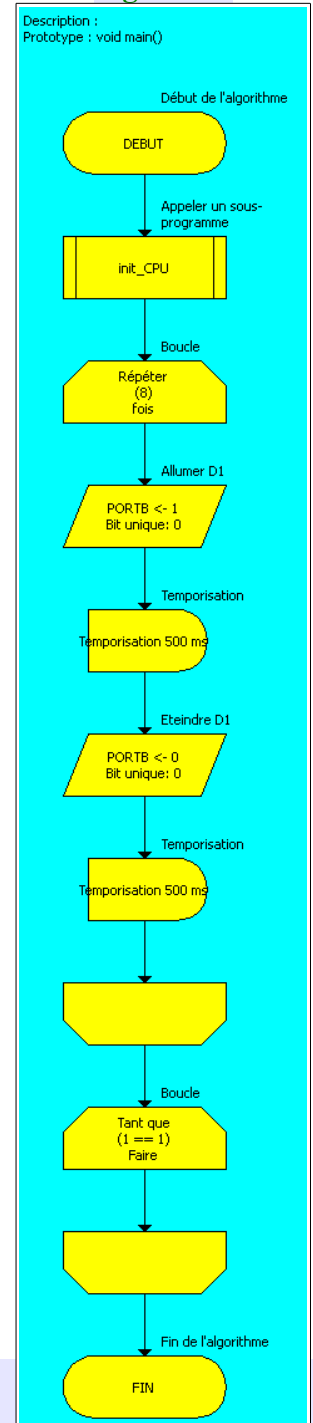
Leurs avantages sont nombreux :

- Faciliter la création, la mise au point et le suivi du programme complet : on ne se consacre plus qu'au code à réaliser dans le sous-programme et aux échanges qu'il doit réaliser avec le programme principal (et éventuellement d'autres sous-programmes).
- Permettre une réutilisation aisée du code pour corrections ou améliorations ultérieures.
- Compacter le code final, car le code d'un sous-programme n'a pas à être réécrit pour pouvoir utiliser le sous-programme plusieurs fois. Ceci se fait néanmoins par une légère perte de rapidité, car l'appel à un sous-programme, puis son retour, prennent quelques instructions machines.
- Constituer éventuellement des bibliothèques de sous-programmes.

- **piq18-algocode** représente chaque **sous-programme** comme un onglet dans sa fenêtre centrale (voir par exemple la **figure 5**, où est indiqué en (1) le sous-programme " **init_CPU** ").
- Il le transforme à la compilation en une fonction du langage C.
- On peut donc tout comme les fonctions du C, lui attribuer des **arguments** et une **valeur de retour**.
- Ces notions sont assez faciles à comprendre car dans vos études vous les avez déjà manipulées, par exemple en voulant calculer le sinus d'un angle de 45°. 45° est ici la valeur à donner à l'argument de la fonction sinus, qui correspond alors à un angle. Cette fonction va renvoyer une valeur $(\frac{\sqrt{2}}{2})$ que vous voudrez peut-être réutiliser pour effectuer un autre calcul. Vous allez alors stocker cette valeur dans une variable. La formule de calcul sera alors : $var1 = \sin(\frac{45 \cdot \pi}{180})$ car généralement l'argument de la fonction sinus doit être exprimé en radian.
- Lorsque c'est à vous d'écrire un **sous-programme** (une fonction), vous ne pouvez pas savoir à l'avance ce que va taper l'utilisateur de votre fonction en argument au moment où il va s'en servir. Vous allez donc considérer une variable qui va stocker cette valeur. Votre code utilisera alors cette variable comme si c'était la valeur entrée. Du point de vue informatique, c'est cela un argument. → Un **argument** est une variable locale au sous-programme que vous définissez au moment de décrire la structure de votre sous-programme.
- Un **sous-programme** (une fonction) peut avoir plusieurs **arguments**, mais il n'aura toujours au plus qu'une seule **valeur de retour**. C'est à dire qu'il accomplira les actions décrites dans son code, mais sans nécessairement retourner une valeur à la fin (par exemple, un sous-programme " **clignote** " peut se résumer aux actions élémentaires pour faire clignoter une led, soit allumer la led, attendre un moment, l'éteindre pour attendre à nouveau, mais ne renvoie pas de valeur).
- Ce même **sous-programme** n'a pas non-plus besoin **d'arguments** (paramètres) pour fonctionner. On pourrait lui en donner un cependant : la durée pendant laquelle la led doit s'allumer, voir deux en lui rajoutant une durée d'extinction distincte ou encore trois, en lui précisant quel bit du port il doit utiliser, etc....

Exercice : Reprendre l'exercice du chapitre " **Utiliser une temporisation** " mais en écrivant le code de clignotement de la led **D0** à l'aide d'un sous-programme. Ce sous-programme devra permettre le réglage de la durée d'allumage de **D0** ainsi que la durée de son extinction séparément.

Figure 17.



->On appellera le sous-programme " **clignote** ". Il lui faut deux arguments, un pour préciser la durée d'allumage, l'autre la durée d'extinction de la led **D0**. Il ne doit rien renvoyer.

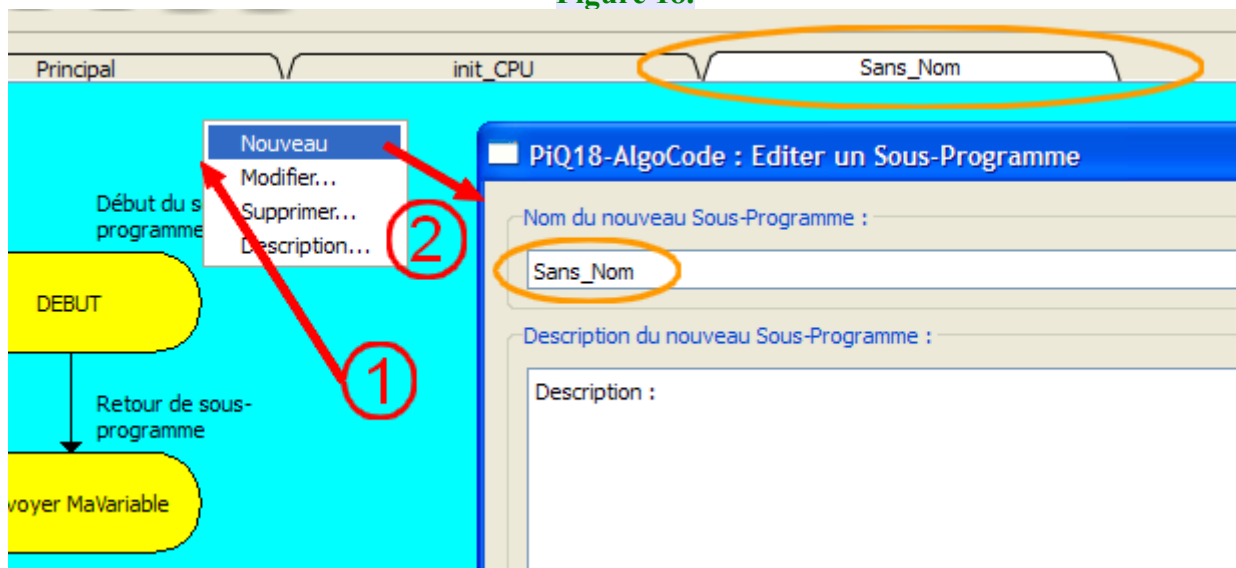
La création du programme doit alors se faire en trois temps :

- 1) Création de la structure du sous-programme " **clignote** ".
- 2) Ecriture du code associé au sous-programme " **clignote** "
- 3) Utilisation du sous-programme " **clignote** ".

VIII.1. Création de la structure du sous-programme " clignote " :

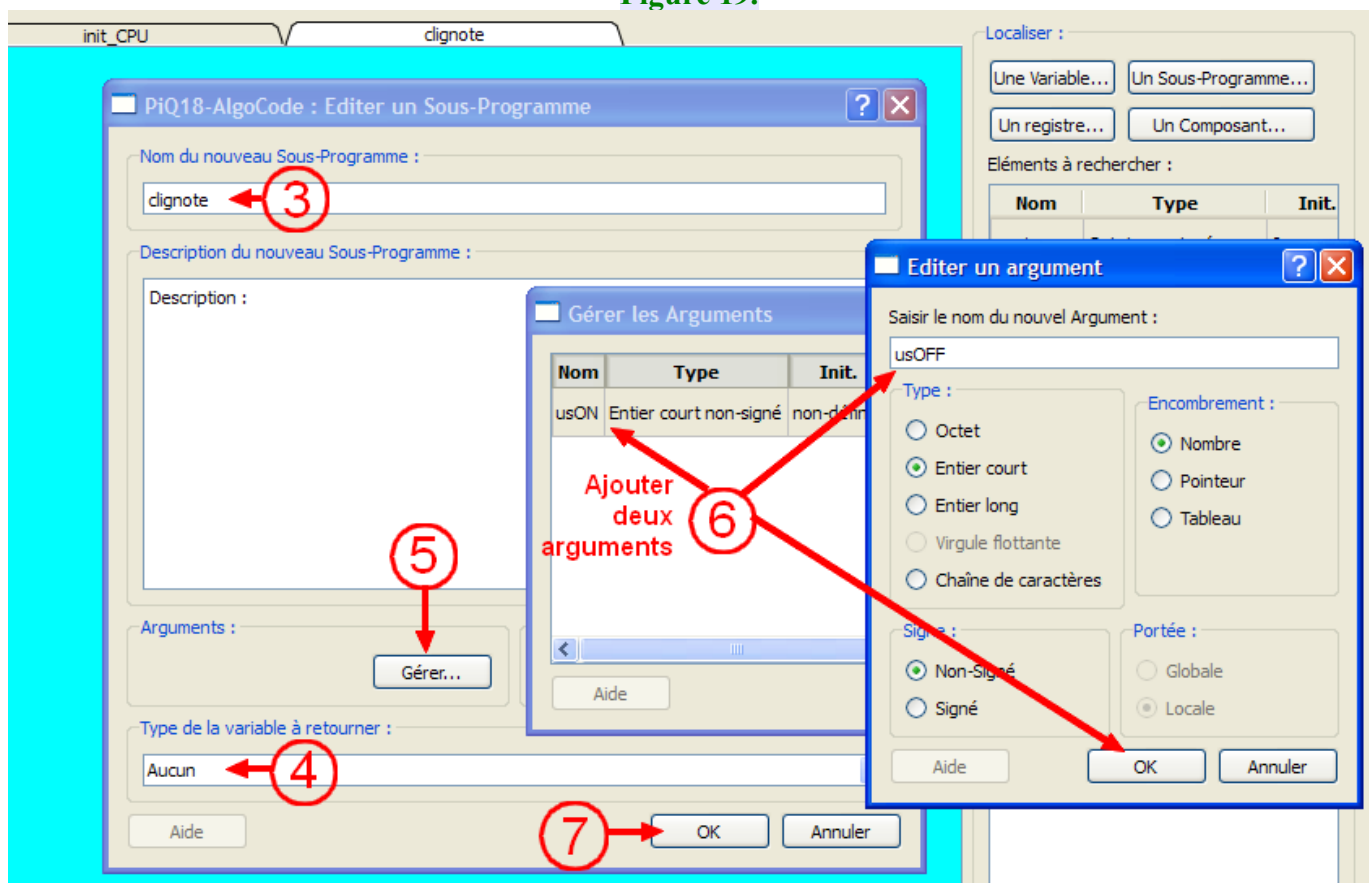
Créer un projet vierge.

Figure 18.



- 1) Cliquer-droit sur le fond bleu du programme principal (ou de tout autre sous-programme déjà créé).
- 2) Choisir " **Nouveau...** " dans le menu contextuel qui apparaît. Ceci ouvre une fenêtre d'édition d'un nouveau sous-programme, et crée temporairement un nouvel onglet (création réelle si validation de la fenêtre d'édition).

Figure 19.



- 3) Donner le nom " **cligote** " au sous-programme.
- 4) En valeur de retour, laisser la valeur par défaut (" **Aucun** " puisque l'on ne souhaite pas renvoyer de valeur).
- 5) Appuyer sur le bouton "Gérer..." de la zone " **Arguments** : ", qui ouvre une fenêtre d'édition des paramètres.
- 6) Si on souhaite régler des durées d'allumage ou d'extinction de la led **D0** (de 0ms à quelques secondes), il faut choisir des arguments de type " **Entier court non-signé** " et utiliser par la suite dans le code du sous-programme, des temporisations dans la gamme des millisecondes, qui utiliseront ces arguments. Valider.

Remarques :

- L'ordre des arguments est important pour la suite. Il est gérable dans la fenêtre d'édition des sous-programmes (**figure 19**).
- Par la suite, les arguments apparaissent en jaune dans toutes les fenêtres qui concernent les variables. Les paramètres ne peuvent être gérés qu'à partir de leur sous-programme propriétaire (cliquer sur l'onglet du sous-programme, puis, dans le menu "Sous-Programmes", choisir " **Modifier** ". On a alors accès à nouveau aux paramètres via le bouton " **Gérer...** " de la zone " **Arguments** : ".

Figure 19.

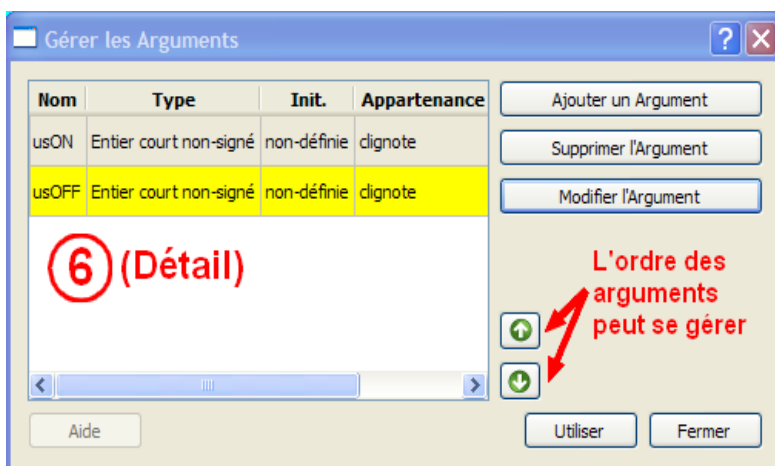
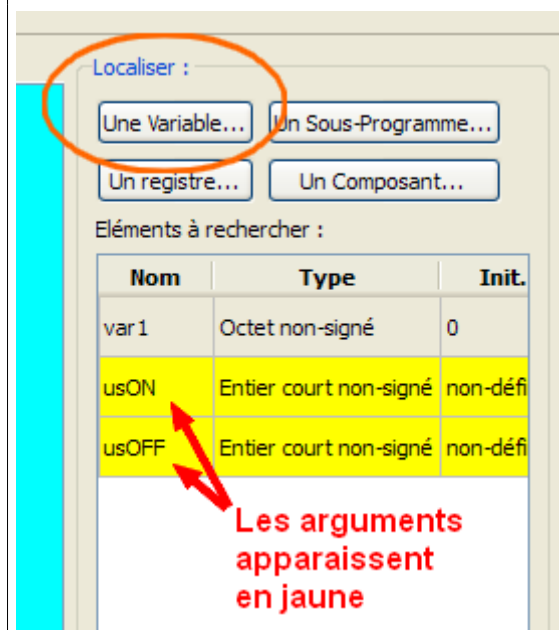


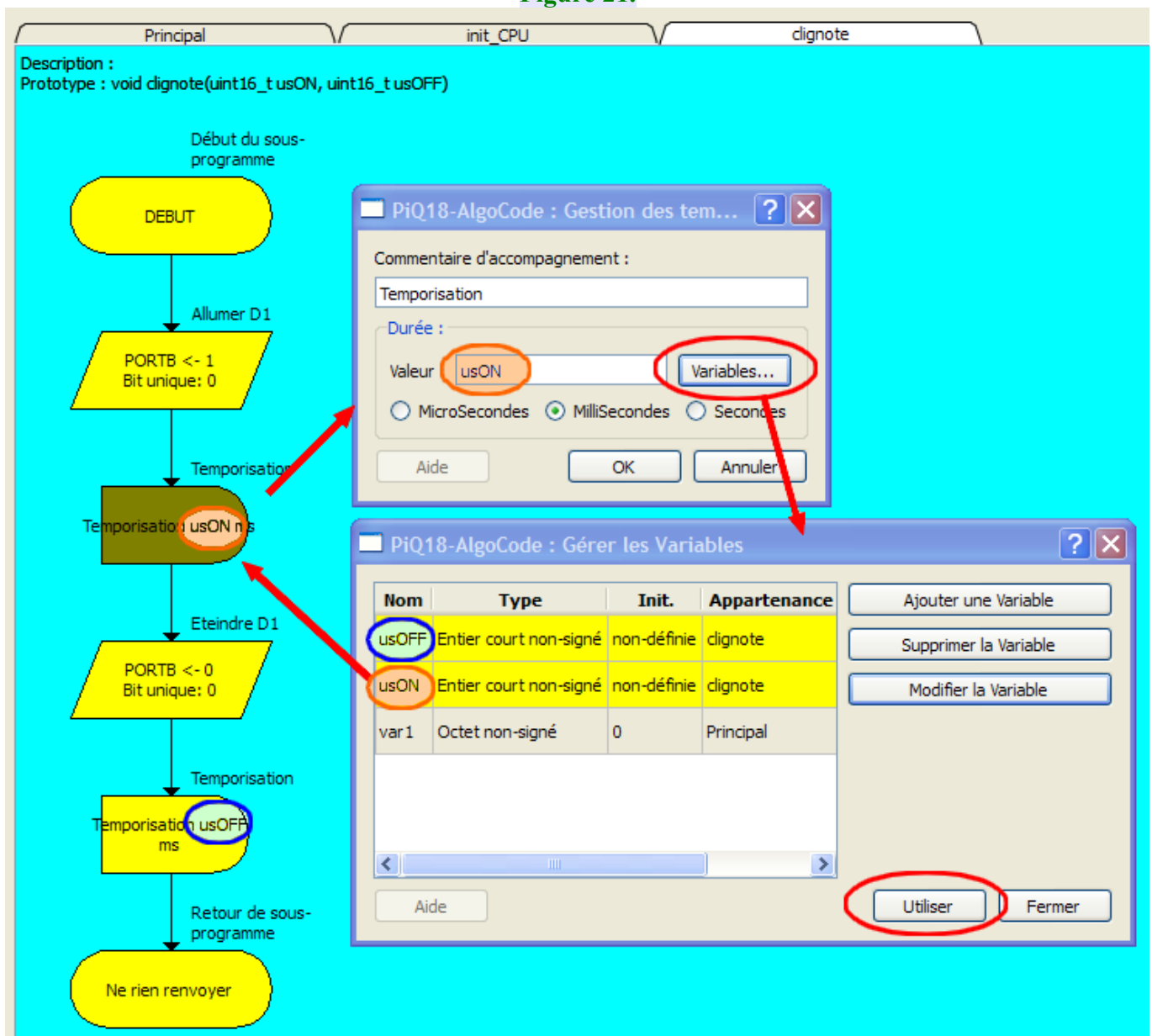
Figure 20.



VIII.2. Ecriture du code associé au sous-programme " clignote " :

- Dans l'onglet " **clignote** " qui vient d'être créé, reprendre le code de la temporisation en précisant bien qu'il faut maintenant utiliser les paramètres du sous-programmes comme **variables** de temporisations.

Figure 21.



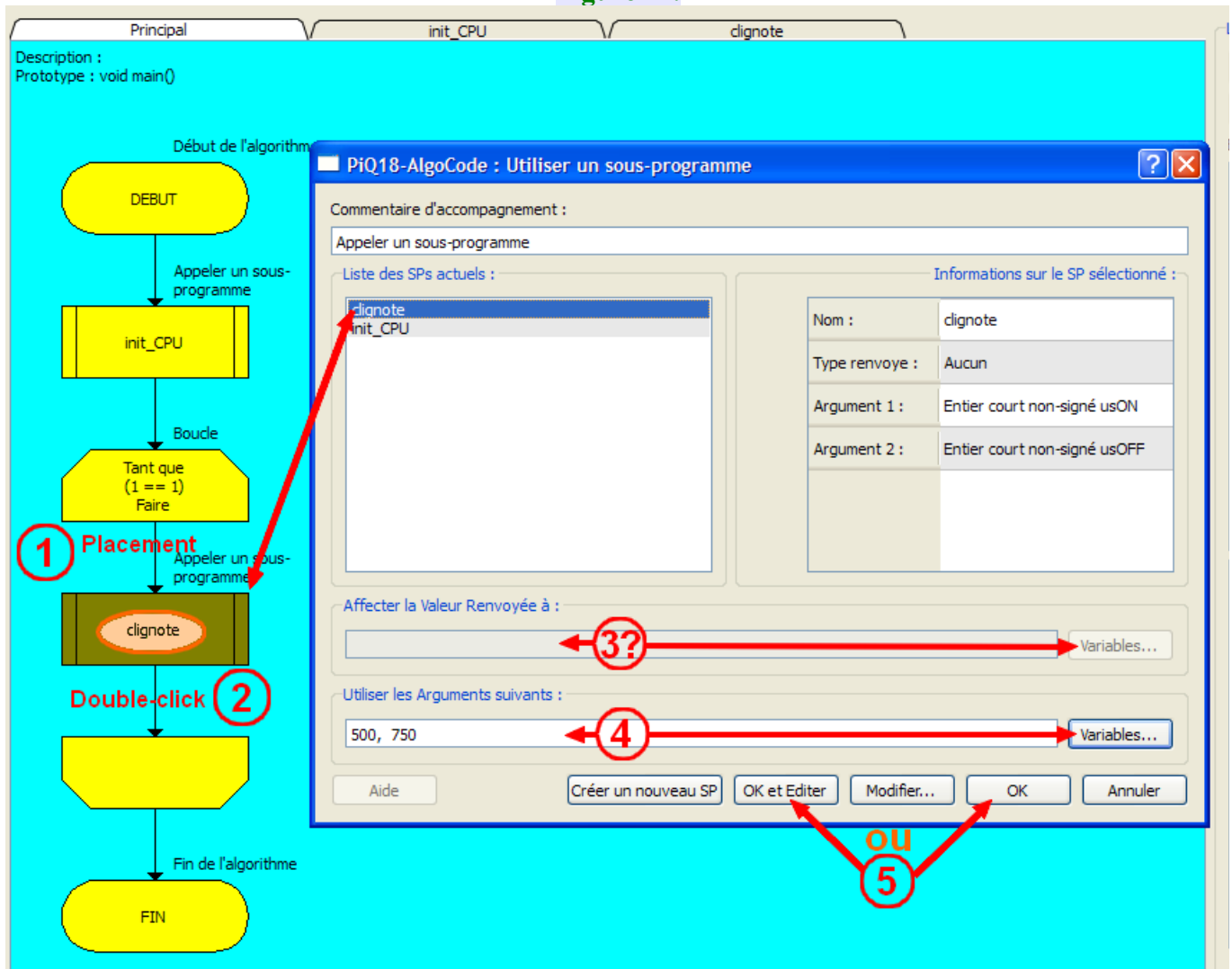
VIII.3. Utilisation du sous-programme " clignote " :

→ cf. figure 22.

- 1) Dans le programme principal, glisser simplement un pictogramme " **Utiliser un sous-programme** " à l'endroit approprié.
- 2) Double-cliquer dessus, ce qui ouvre une fenêtre d'utilisation des sous-programmes. Choisir ici le sous-programme " **clignote** ".
- 3) Donner des valeurs adéquates aux arguments (des variables sont possibles).
- 4) Vérifier si le sous-programme ne renvoie pas de valeur (ce qui n'est pas le cas ici), auquel cas il faut déclarer la variable à utiliser pour la récupérer.
- 5) Valider (le bouton " **Ok & Editer** ", valide les réglages, ferme la fenêtre d'édition du sous-programme sélectionné et ouvre l'onglet de ce sous-programme).
- 6) Compiler le programme (menu " **Projet** " + " **Compiler vers HEX...** ").

Remarque :

- Un **sous-programme** ne peut être supprimé que s'il n'est pas utilisé ailleurs, tout comme les **variables** (rendre actif le sous-programme via son onglet si cela est nécessaire, puis menu " **Sous-programmes** " + " **Supprimer...** ", ou tout simplement par clic-droit sur son fond bleu et choix du menu contextuel " **Supprimer...** ").

Figure 22.**IX. Utiliser un composant :**

Cela se passe en trois temps :

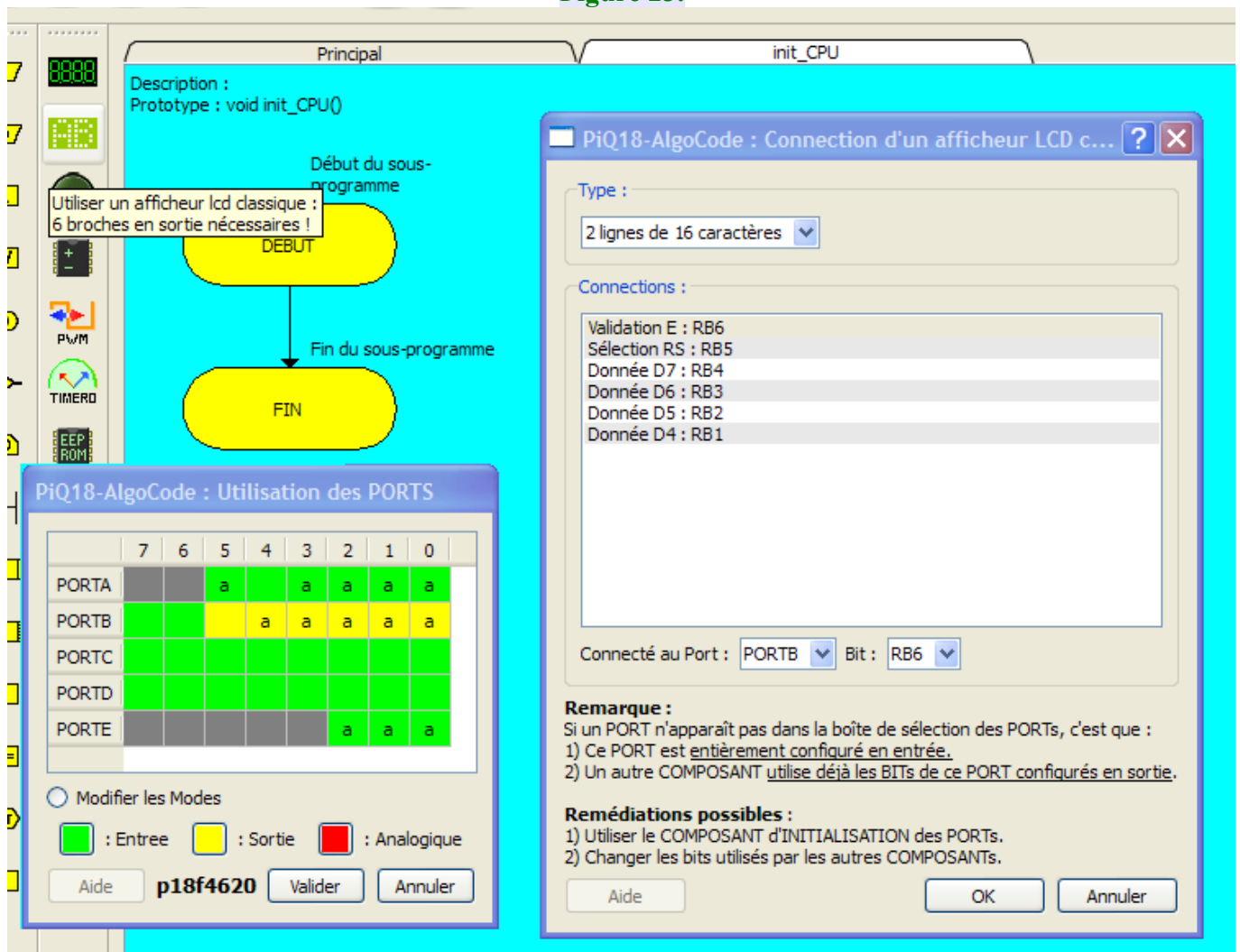
- 1) Cliquer, dans la barre des composants (deuxième barre verticale à gauche de la fenêtre centrale), sur l'icône du composant souhaité. Ceci ouvre une fenêtre de configuration de ce composant.
- 2) Configurer cette fenêtre et valider.
- 3) Dans le programme, placer un pictogramme " **Utiliser un composant** " (cf. figure 4) à l'endroit du programme où vous souhaitez avoir accès au composant. Double-cliquer dedans pour ouvrir une boîte de dialogue qui donne accès à des bibliothèques de fonctions. Une bibliothèque (indiquée normalement à la fermeture de la boîte de configuration du composant) gère normalement le composant voulu.

Exercice : Afficher " Essais AFFICHEUR LCD " sur les deux lignes d'un afficheur LCD (2 lignes, 16 colonnes).

- Créer un projet vierge.
- Dans la barre des composants (deuxième barre verticale à partir de la gauche), cliquer sur l'icône du composant " **Utiliser un afficheur LCD classique** ". Un avertissement vous informe qu'il faut au moins six broches configurées en sortie pour utiliser l'afficheur LCD. Soit ! Initialisons le pic en conséquence. Plaçons les broches **RB5 à RB0** en sortie à l'aide de la fenêtre de configuration/visualisation des ports (cf. chapitre " **Créer un projet** ").
- Cliquer à nouveau sur l'icône du composant " **Utiliser un afficheur LCD classique** ". Une fenêtre de configuration s'ouvre (figure 23). Normalement il n'y a pas de réglages à faire, sauf si on tient à assigner les broches très précisément.
- Valider en cliquant sur le bouton " **OK** ". Une fenêtre d'avertissement vous informe que pour utiliser le composant il suffit de glisser un pictogramme " **Utiliser un composant** " (cf. figure 4) dans le programme et d'utiliser la bibliothèque associée. Faisons-le donc !

Remarque : Pour d'autres composants, les avertissements peuvent être plus nombreux.

Figure 23.



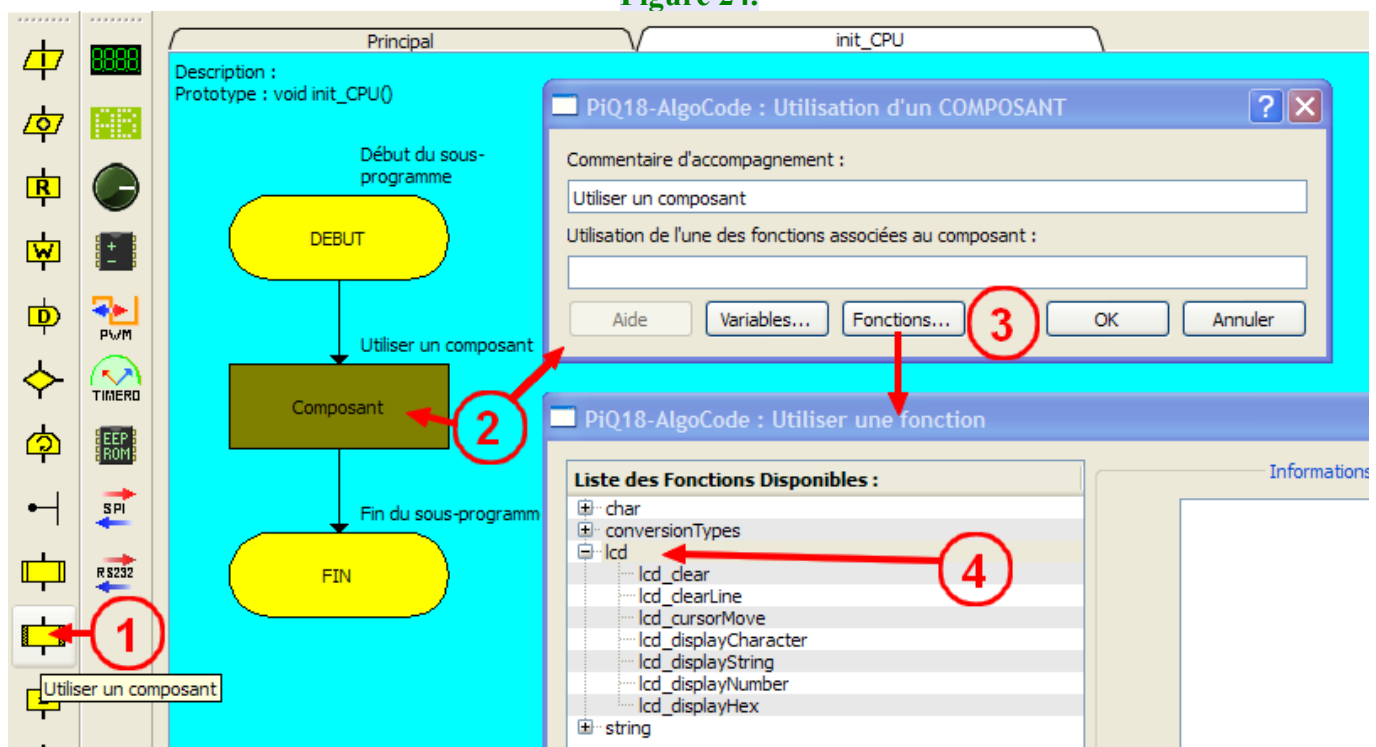
- Le fait d'utiliser un composant en cliquant sur son icône, dit à **piq18-algocode** de créer son code d'initialisation.

L'utilisation des composants se fait à travers des fonctions regroupées dans des bibliothèques. La notion de fonctions a été vue dans le chapitre " **Créer et utiliser un sous-programme** ". Nous n'y revenons donc pas. Il faut juste se souvenir qu'une fonction peut avoir besoin **d'arguments** et peut **retourner une valeur**, à stocker alors dans une **variable** du type approprié.

Remarque : Ici, l'affichage doit se faire une seule fois. Nous allons donc créer le code dans le sous-programme " **init_cpu** ".

- 1) Placer un pictogramme " **Utiliser un composant** " dans le sous-programme " **init_cpu** ".
- 2) Double-cliquer dessus. Cela ouvre une fenêtre d'utilisation d'un composant.
- 3) Cliquer sur le bouton " Fonctions... ".
- 4) Ouvrir la liste correspondant à la bibliothèque **lcd**. Toutes les fonctions de cette bibliothèque apparaissent alors.

Figure 24.



- 5) Parcourir la liste des fonctions pour avoir des détails sur leur utilisation. Ici, les fonctions " **lcd_displayString()** " et " **lcd_moveCursor()** " nous intéressent. Vérifier à chaque fois si la fonction retourne une valeur et si c'est le cas, sa signification et son type. Il faudra alors déclarer la variable à utiliser pour stocker cette valeur (utiliser le bouton " **Variables...** " en face de la ligne d'édition " **Affecter la valeur de retour à :** ").
- 6) Vérifier à chaque fois si la fonction a besoin d'arguments et si c'est le cas, leur signification et leur type. Il faudra alors entrer une donnée ou une variable sur la ligne d'édition " **Utiliser les arguments suivants :** " (→ Séparation des arguments à l'aide d'une virgule s'il y en a plusieurs à donner). Voir les **figures 25 et 26**.

Figure 25.

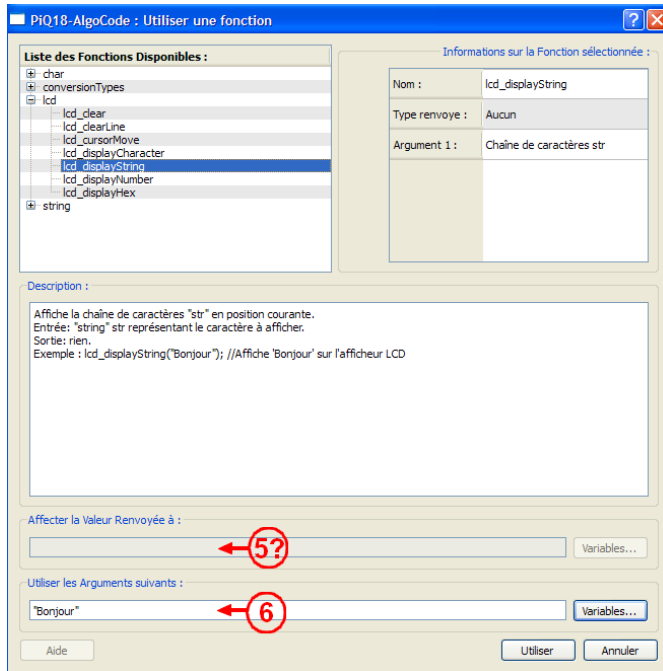
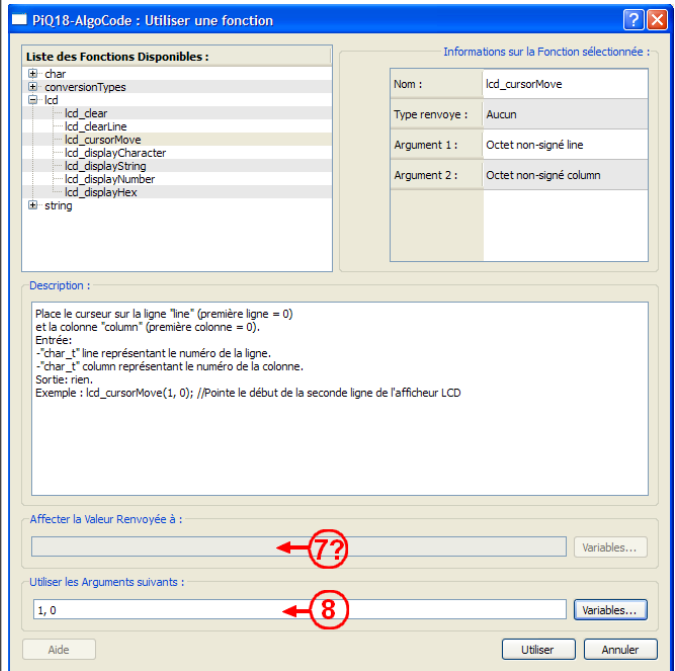


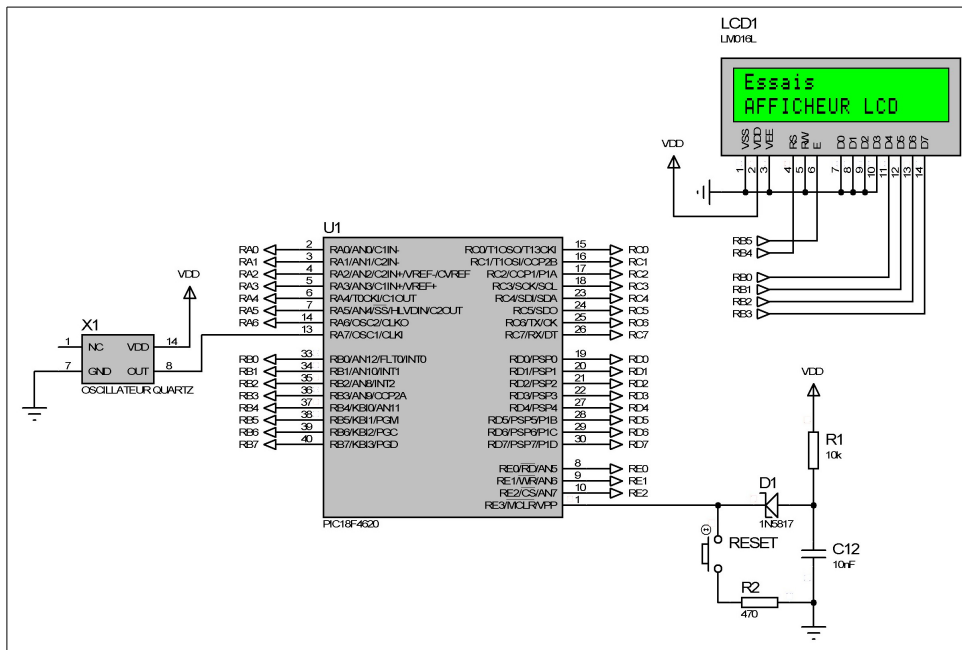
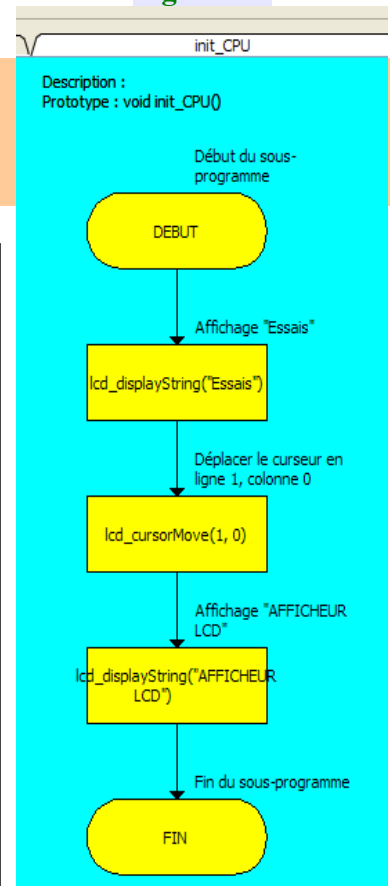
Figure 26.



On obtient alors le programme de la **figure 27**, qu'il faut ensuite compiler.

Remarque : Si on veut supprimer le composant : il faut avant tout supprimer tous les pictogrammes qui l'utilise puis, dans le menu **" Composants "**, cliquer simplement sur le bouton qui correspond au composant.

Figure 27.



X. Utiliser une interruption :

Une interruption est un processus très pratique qui permet à un composant externe au pic d'interrompre à tout moment le fonctionnement du programme par un événement qu'il déclenche sur l'une broche. Le pic achève alors l'instruction en cours, mémorise l'emplacement de l'instruction suivante et lance un sous-programme appelé **sous-programme d'interruption**. Lorsque le sous-programme d'interruption s'achève, le pic se replace à l'emplacement qu'il avait mémorisé et poursuit alors le programme initial.

- Pratiquement tous les modules internes au pic peuvent également déclencher une interruption, généralement lorsqu'ils ont accompli la tâche qui leur a été assignée.

Les pics 18xxxxx ont deux niveaux d'interruptions : **bas** et **haut**. Une interruption de **priorité haute** peut venir interrompre le sous-programme d'interruption associé à une interruption de **priorité basse**.

L'inverse n'est pas vrai.

La plupart des fenêtres de configuration des composants (ou plutôt des modules internes) proposent de gérer les interruptions qui leur sont associées. Ensuite, pour pouvoir les invalider ou les valider de nouveau, il faut faire appel aux fonctions **stop()** ou **restart()** des bibliothèques associées à ces composants.

Remarques :

- **piq18-algocode** valide la gestion des interruptions par priorités au début du programme principal, après la phase de configuration et d'initialisation des ports. Ensuite, il met à " 1 " les bits de validation des interruptions globales et périphériques (**GIE** et **PEIE**). Il gère ensuite au cas par cas les bits de validation d'interruption de chaque module (bit **IE**).
- La politique de gestion des interruptions associées aux modules internes du pic sélectionné est de les valider dès l'initialisation de ces composants, avec la priorité choisie (basse ou haute), bien sûr, uniquement si les interruptions sont validées lors de la configuration de ces modules.
- Les bibliothèques associées à chaque module possèdent une fonction **stop()** et une fonction **restart()**. La fonction **stop()** arrête le module et invalide la source d'interruption qui lui est associée. La fonction **restart()** redémarre le module et rétablit la validation de l'interruption si ce module a été configuré avec les interruptions validées.

Pour les autres interruptions, il existe un pictogramme réservé uniquement à cela : le pictogramme

" **Gérer une interruption** " (cf. **figure 4**).

- La gestion de ces interruptions est différente : on valide ou non l'interruption choisie en écrivant respectivement " 1 " ou " 0 " dans le bit de validation qui lui est associé, à l'endroit même où est placé ce pictogramme.

Dans tous les cas :

- Seuls les **sous-programmes sans arguments et sans paramètres** existants peuvent être choisis comme **sous-programme d'interruption**.
- Si l'onglet actif est un tel sous-programme, il n'apparaît pas dans la liste des sous-programmes d'interruptions potentiels, pour éviter qu'il s'appelle lui-même (boucle sans fin extrêmement dangereuse).
- A chaque retour du sous-programme d'interruption, le **drapeau** (bit **IF**) associé est remplacé automatiquement à " 0 ", ce qui permet de repartir en interruption aussitôt après si nécessaire. De plus :
 - Pour les modules, il l'est également à chaque utilisation des fonctions **stop()** et **restart()**. Ce comportement peut être changé en modifiant le code des fonctions de la bibliothèque de chaque module (répertoire /librairies). Mais c'est à vos risques et périls !
 - Pour les interruptions non-gérées via un composant, le bit de drapeau est mis à " 0 " avant chaque validation ou invalidation de l'interruption.

Exercice : Reprendre l'exercice du chapitre " Lire un port " en utilisant l'interruption " Externe 2 (INT2) " associée à la broche **RB2** (régler cette interruption pour qu'elle soit déclenchée sur un front descendant).

- Il suffit de créer un sous-programme d'interruption appelé " **toggle_RB0** " (donc sans argument et sans valeur de retour) qui sera appelé via un pictogramme "Gérer une interruption" placé dans le sous-programme " **init_cpu** ". Régler sa configuration alors comme demandé.

Figure 28.

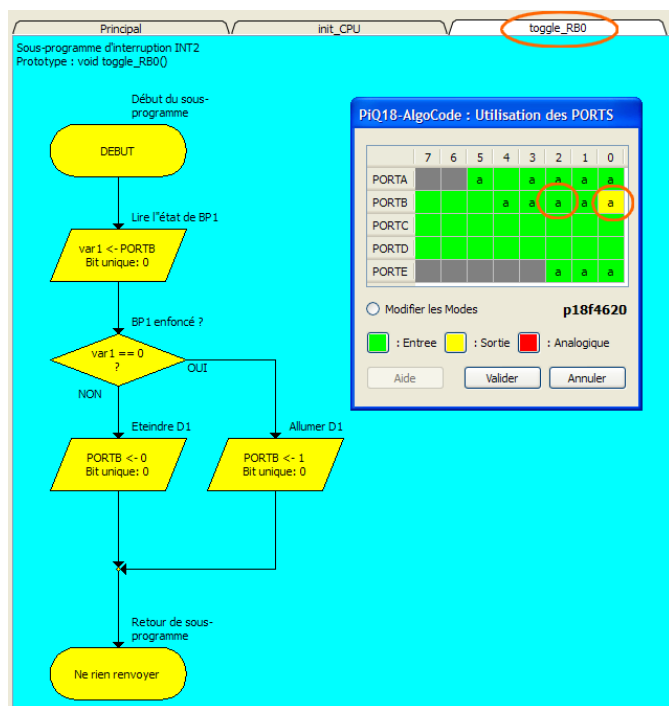
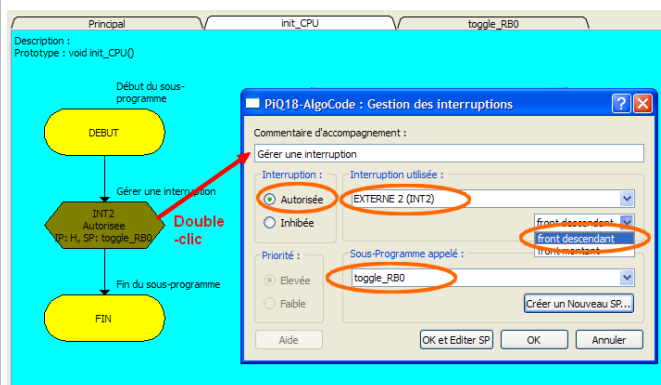


Figure 29.



Compiler le programme final.