

HIWARE

Innovation for your Success

PIAU J.P.

Notice d'utilisation

Version 2.0

Mai 1999

SOMMAIRE

I. LE GESTIONNAIRE DE PROJET : HIWARE TOOLS.....	6
I.1. CONFIGURATION :	7
I.1.1. Nouveau projet :	7
I.1.2. Définition du projet :	7
I.1.2.1. Nom du projet :	7
I.1.2.2. Définition de l'éditeur de texte :	8
I.1.2.3. Définition des outils accessibles :	8
II. L'ÉDITEUR DE TEXTE : WINEDIT32 :	9
II.1. L'INTERFACE GRAPHIQUE DE WINEDIT :	9
II.2. LES COMMANDES :	9
<i>Commandes de gestion des fichiers :</i>	<i>9</i>
II.2.2. <i>Commandes d'édition :</i>	<i>9</i>
II.2.3. <i>Commandes de mise au point :</i>	<i>10</i>
III. L'ASSEMBLEUR :	11
III.1. L'INTERFACE GRAPHIQUE DE L'ASSEMBLEUR :	11
III.2. LA BARRE D'OUTILS :	11
III.3. LE MENU FICHIER / LANCEMENT DE L'ASSEMBLAGE :	12
III.4. LES OPTIONS D'ASSEMBLAGE :	13
III.4.1. Options "Output" :	13
III.4.2. Options "Input" :	13
III.4.3. Options "Host" :	13
III.4.4. Options "Code generation" :	14
III.4.5. Options "Messages" :	14
III.4.6. Options "Various" :	14
III.5. ORGANISATION DE LA MÉMOIRE :	14
III.5.1. Attributs des sections :	14
III.5.2. Types des sections:	14
III.5.2.1. Sections absolues :	14
III.5.2.2. Sections relogeables :	15
III.5.2.3. Avantages des sections relogeables :	15
III.6. SYNTAXE :	15
III.6.1. Le champs des étiquettes :	15
III.6.2. Le champs mnémonique :	15
III.6.3. Le champs opérande :	15
III.6.4. Les modes d'adressages :	16
III.6.5. Les symboles :	16
III.6.6. Le type des constantes :	16
III.6.7. Les opérateurs et les expressions:	16
III.6.8. Les directives d'assemblages :	17
III.6.8.1. Définition de sections:	17
III.6.8.2. Définition de constantes :	17
III.6.8.3. Allocation de donnée :	17
III.6.8.4. Importation et exportation des symboles :	17
III.6.8.5. Contrôle de l'assemblage :	17
III.6.8.6. Contrôle du fichier listing :	18
III.6.8.7. Contrôle des macros :	18
III.6.8.8. Assemblage conditionnel :	18
III.6.9. Les macros :	18
III.6.9.1. Définition :	18
III.6.9.2. Appels :	18
III.6.9.3. Arguments :	18
III.6.9.4. Étiquettes dans les macros :	19
IV. LE COMPILATEUR C / C++ :	20
IV.1. L'INTERFACE GRAPHIQUE DU COMPILATEUR :	20
IV.2. LA BARRE D'OUTILS :	20
IV.3. LE MENU FICHIER / LANCEMENT DE LA COMPILATION :	21

DÉFINIR LA CONFIGURATION :	21
IV.4. CONFIGURATION DU COMPILATEUR :	22
IV.4.1. Paramétrage des types standards :	22
IV.4.2. Les options de compilation :	22
IV.4.2.1. Options "Optimizations" :	23
IV.4.2.2. Options "Output" :	23
IV.4.2.3. Options "Input" :	24
IV.4.2.4. Options "language" :	24
IV.4.2.5. Options "Code Generation" :	24
IV.4.2.6. Option "Host" :	25
IV.4.2.7. Options "Messages" :	25
IV.4.3. Contrôle de l'optimisation, "Smart Sliders" :	25
IV.5. LES MACROS :	25
IV.6. LES PRAGMAS :	26
IV.7. SYNTAXE DU COMPILATEUR :	26
IV.7.1. ANSI-C :	27
IV.7.1.1. Mots clés :	27
IV.7.1.2. Directives de compilations :	27
IV.7.1.3. Codage des nombres réels :	27
IV.7.1.4. Champs de bits :	27
IV.7.1.5. Segmentation de la mémoire :	28
IV.7.2. Motorola HC12 :	28
IV.7.2.1. Mémoire non-paginée :	28
IV.7.2.2. Mémoire paginée :	28
IV.7.2.3. Types des variables :	28
IV.7.2.4. Appels des fonctions :	29
IV.7.2.5. Les fonctions et la pile :	30
IV.7.2.6. Les fonctions d'interruptions :	30
IV.7.2.7. Segmentation mémoire :	30
IV.7.2.8. Conseils de programmation :	31
IV.7.3. Insertion de code assembleur :	31
IV.7.3.1. Syntaxe :	31
IV.7.3.2. Particularités :	31
IV.8. LIBRAIRIES :	32
IV.8.1. ANSI-C :	32
V. L'ÉDITEUR DE LIENS :	34
V.1. L'INTERFACE GRAPHIQUE DE L'ÉDITEUR DE LIENS :	34
V.2. LA BARRE D'OUTILS :	35
V.3. LE MENU FICHIER / LANCEMENT DE L'ÉDITION DE LIENS :	35
V.4. LES OPTIONS DE L'ÉDITION DE LIENS :	36
V.4.1. Options "Output" :	36
V.4.2. Options "Input" :	36
V.4.3. Options "Host" :	36
V.4.4. Options "Messages" :	36
V.5. FORMAT DES FICHIERS :	37
V.5.1. Fichiers d'entrées :	37
V.5.2. Fichiers de sorties :	37
V.6. PARAMÉTRAGE DE L'ÉDITION DE LIENS :	37
V.6.1. Nom du fichier absolu :	37
V.6.2. Fichiers à lier :	37
V.6.3. Segmentation de la mémoire :	38
V.6.3.1. Segments physiques :	38
V.6.3.2. Segments virtuels :	38
V.6.3.3. Attributs des segments :	38
V.6.3.4. Alignement des segments :	38
V.6.3.5. Initialisation des segments :	38
V.6.4. Placement des segments :	39
V.6.5. Sections pré-définies :	39
V.6.6. Initialisation de la table des vecteurs :	40
V.6.7. Initialisation de la pile :	40
V.6.8. Point d'entrée du programme :	41
V.6.9. Édition de liens des programmes assembleurs :	41
VI. ENVIRONNEMENT DE SIMULATION ET DE DEBUGAGE HIWAVE :	42
VI.1. INTRODUCTION :	42

VI.2. INTERFACE GRAPHIQUE :	42
VI.2.1. <i>Les menus</i> :	42
VI.2.1.1. Le menu File :	43
VI.2.1.2. Le menu View :	43
VI.2.1.3. Le menu Run :	43
VI.2.1.4. Le menu Target :	44
VI.2.1.5. Le menu component :	44
VI.2.1.6. Le menu Window :	44
VI.2.2. <i>Le barre d'outils</i> :	44
VI.2.3. <i>Menus associés aux objets</i> :	45
VI.2.4. <i>Possibilités de l'interface</i> :	45
VI.3. LES OBJETS OU COMPOSANTS :	45
VI.3.1. <i>Ouverture des objets</i> :	45
VI.3.2. <i>SOURCE</i> :	45
VI.3.2.1. Les menus :	46
VI.3.2.2. Le glisser/déposer :	46
VI.3.3. <i>ASSEMBLY</i> :	47
VI.3.3.1. Les menus :	47
VI.3.3.2. Le glisser/déposer :	47
VI.3.4. <i>Procédure</i> :	48
VI.3.4.1. Le Menu :	48
VI.3.4.2. Le glisser/déposer :	48
VI.3.5. <i>Register</i> :	48
VI.3.6. <i>Data</i> :	49
VI.3.6.1. Les menus :	49
VI.3.6.2. Le glisser/déposer :	50
VI.3.7. <i>Memory</i> :	50
VI.3.7.1. Les menus :	50
VI.3.7.2. Le glisser/déposer :	51
VI.3.8. <i>Coverage</i> :	51
VI.3.9. <i>Inspector</i> :	51
VI.3.10. <i>IO_LED</i> :	52
VI.3.11. <i>Component</i> :	52
VI.3.12. <i>Profiler</i> :	52
VI.3.13. <i>Recorder</i> :	52
VI.3.14. <i>SoftTrace</i> :	52
VI.3.15. <i>Command Line</i> :	53
VI.3.16. <i>TestTerm</i> :	53
VI.3.16.1. Fonctionnement :	53
VI.3.16.2. Utilisation :	53
VI.3.17. <i>Terminal</i> :	54
VI.3.18. <i>Les utilitaires de visualisation</i> :	54
L'objet LED :	54
VI.4. CONTRÔLE DE L'EXÉCUTION :	55
VI.4.1. <i>Exécution et arrêt</i> :	55
VI.4.2. <i>Les points de contrôle</i> :	55
VI.4.2.1. Édition des points d'arrêts :	55
VI.4.2.2. Édition des points de regard :	56
VI.5. STIMULATION DES ENTRÉES/SORTIES :	57
VI.5.1. <i>Syntaxe des fichiers de stimulation</i> :	57

HIWARE TOOLS : NOTICE D'UTILISATION.

HIWARE TOOLS est un ensemble d'outils pour la programmation des microcontrôleurs Motorola et en particulier, pour le 68HC12. On a accès aux différents outils depuis un gestionnaire de projet. Ceux-ci sont :

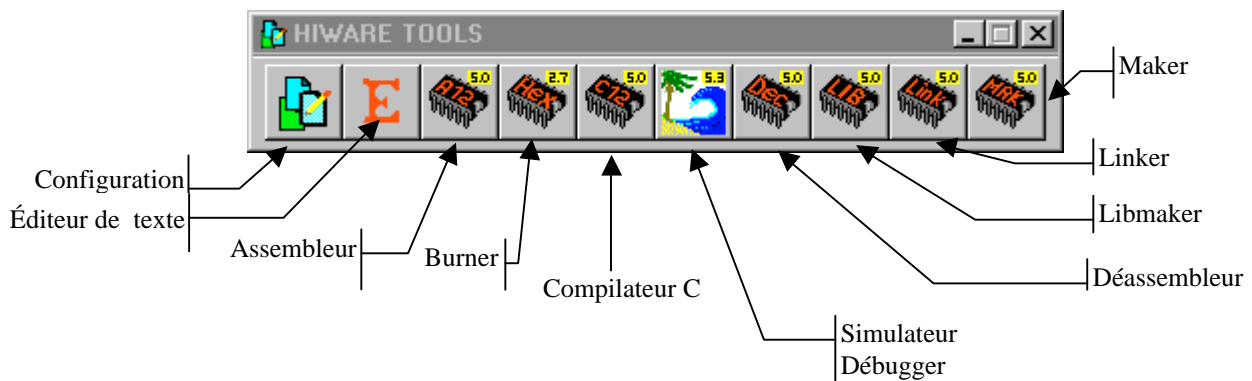
- Un assembleur : AHC12.exe ;
- Un compilateur C/C++ : CHC12.exe ;
- Un éditeur de liens : Linker.exe ;
- Un *Maker* : Make.exe ;
- Un générateur de fichiers de programmation : Burner.exe ;
- Un désassembleur : Decode.exe ;
- Un gestionnaire de bibliothèques : Libmaker.exe ;
- Un débogueur : Hiwave.exe ;

Tous ces outils peuvent être lancés depuis le gestionnaire de projet à l'aide de raccourcis. De plus, on peut avoir directement accès à un éditeur de texte pour éditer les programmes.

I. LE GESTIONNAIRE DE PROJET : HIWARE TOOLS.



Le gestionnaire de projet permet de définir l'environnement de programmation. On y accède par un icône dans le bureau de Windows, ou par le menu *démarrer / programmes / Hiware / programmes / HIWARE TOOLS*. La fenêtre du gestionnaire de projet contient plusieurs icônes correspondant aux différents outils accessibles pour le projet en cours :



Un projet permet de réunir dans un seul répertoire tous les fichiers utilisés lors du développement d'un programme. Il définit un environnement de travail dont les fichiers de configuration sont localisés dans le répertoire du projet. On doit donc retrouver dans ce répertoire les fichiers suivants :

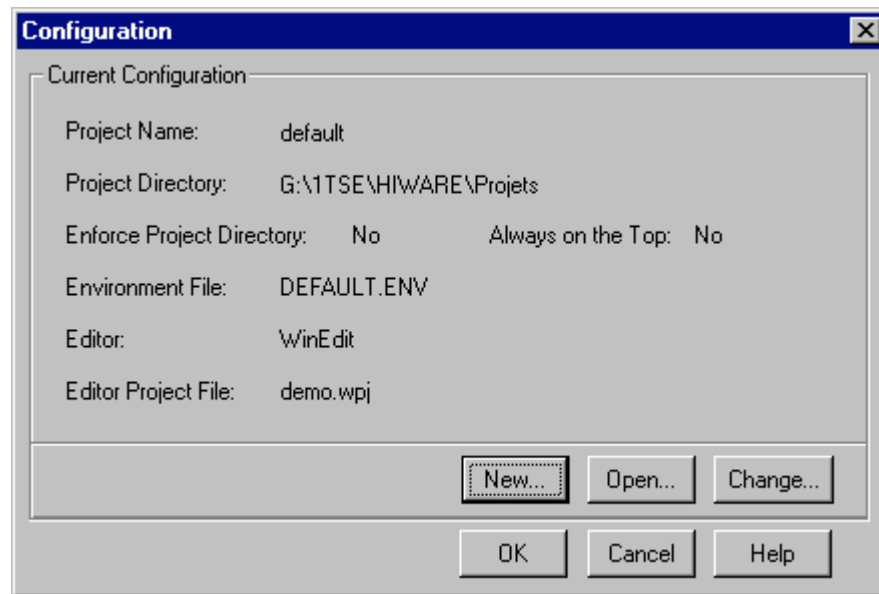
- DEFAULT.ENV : contient les informations sur l'environnement (chemins des fichiers...) ;
- PROJECT.INI : contient les informations de configuration des différents outils ;
- PROJECT.VPJ : contient les informations de configuration de l'éditeur de texte WinEdit32 ;

Chaque projet est personnalisable.

Un répertoire unique doit correspondre à chaque projet.

I.1. Configuration :

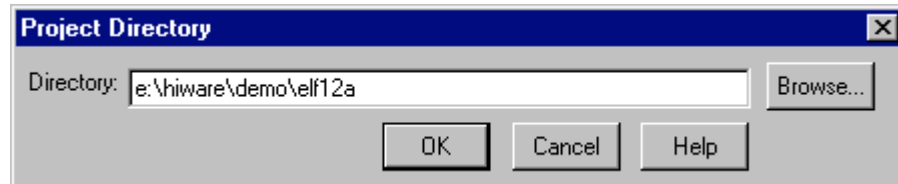
On lance la configuration en cliquant sur le premier icône à gauche de la fenêtre.



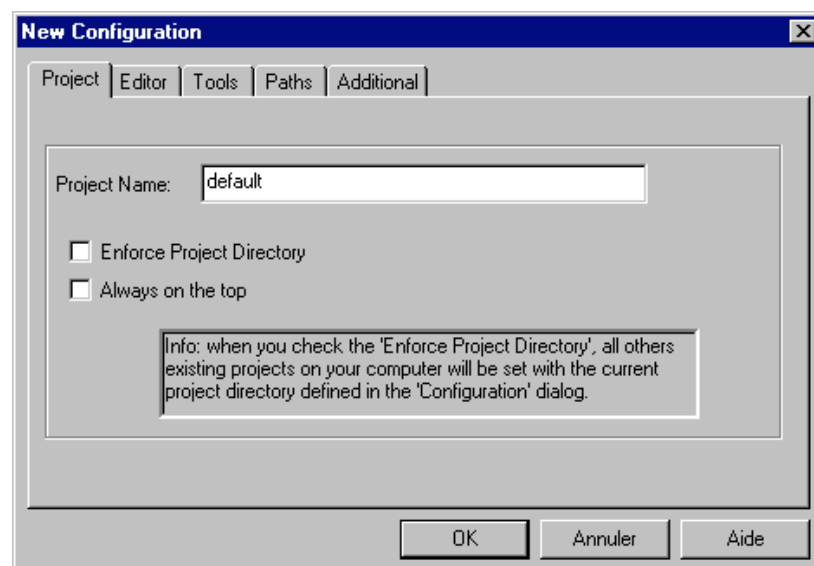
Dans cette fenêtre, on peut lire la configuration courante et on peut définir un nouveau projet (*New...*), ouvrir un projet existant (*Open...*) ou changer la configuration du projet courant (*Change...*).

I.1.1. Nouveau projet :

Pour définir un nouveau projet il suffit d'indiquer le chemin de son répertoire. Si le répertoire n'existe pas, on le crée en tapant son nom et son chemin dans la fenêtre prévue à cet effet :

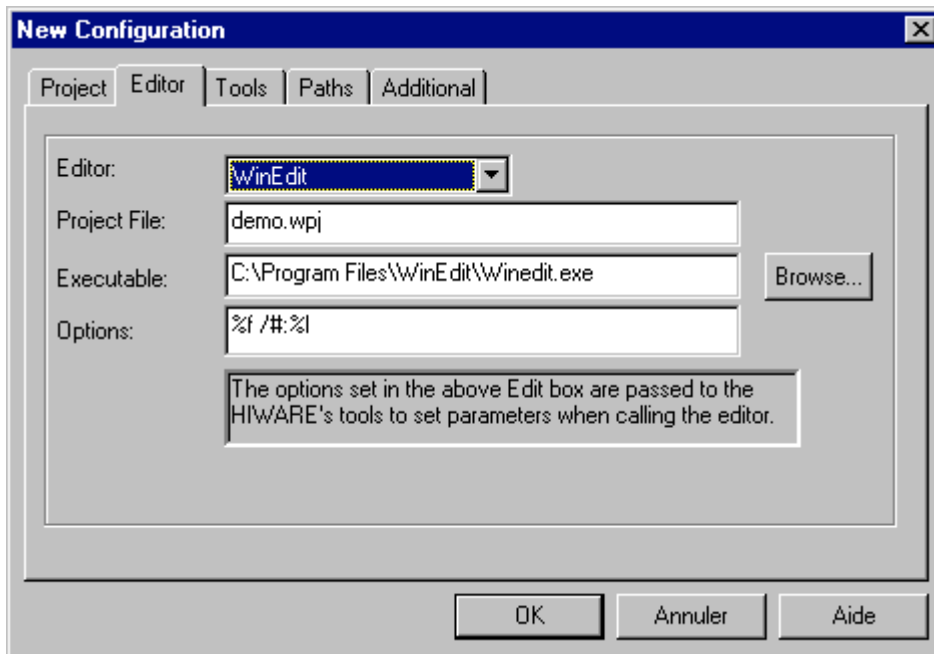


Il faut ensuite définir la configuration du projet.

I.1.2. Définition du projet :**I.1.2.1. Nom du projet :**

I.1.2.2. Définition de l'éditeur de texte :

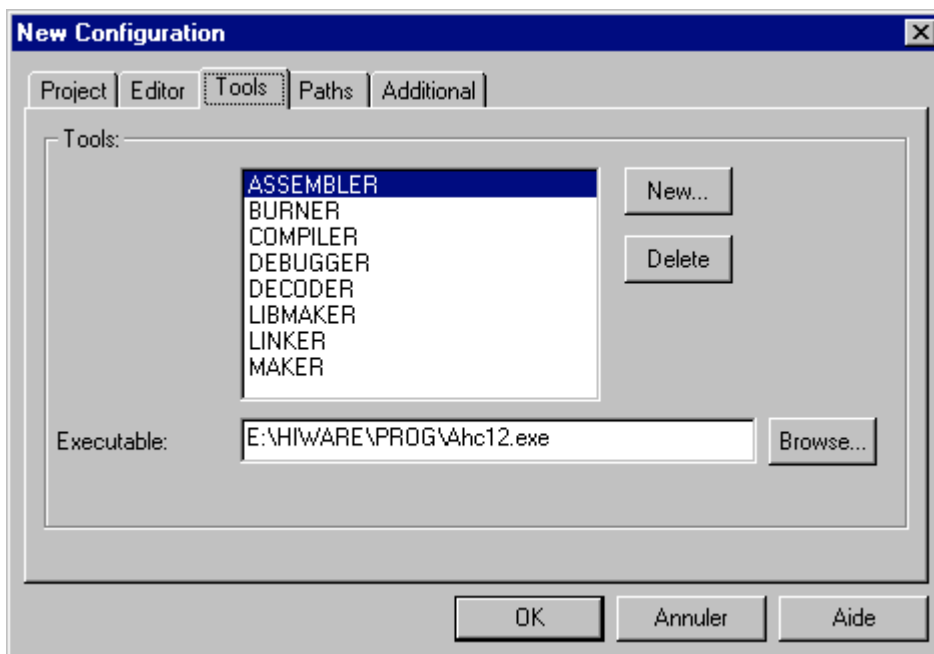
Pour définir l'éditeur de texte qui sera utilisé pour l'édition des programmes, on passe par cette fenêtre :



Pour utiliser WinEdit, on utilisera la configuration précédente. On pourra changer le nom du fichier de configuration de WinEdit (demo.wpj).

I.1.2.3. Définition des outils accessibles :

C'est dans cette fenêtre que l'on ajoute ou que l'on retire les objets qui seront accessibles à partir de la fenêtre de projets.



On peut aussi définir les chemins des différents répertoires qui seront utilisés dans le projet à l'aide de l'onglet "Paths". On peut éventuellement définir des paramètres personnalisés dans l'onglet "Additional".

II. L'ÉDITEUR DE TEXTE : WINEDIT32 :

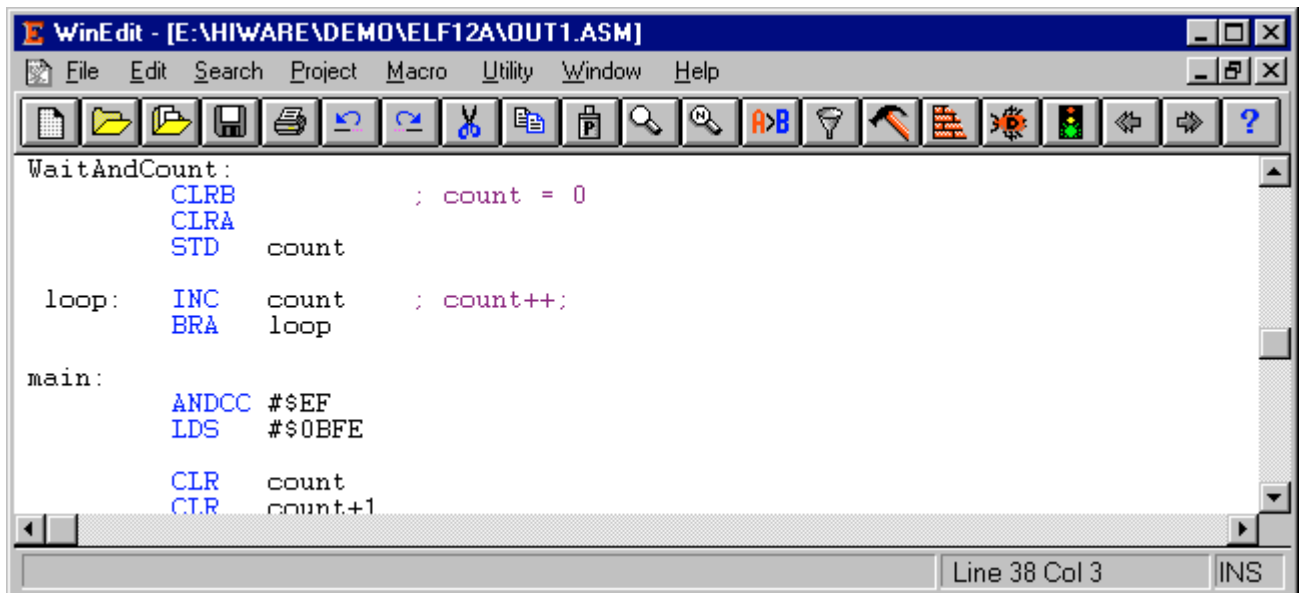
N'importe quel éditeur, tel que le *BlockNote* de *Windows*, peut être utilisé pour éditer un programme. Dans la suite de ce document, nous utiliserons l'éditeur de texte évolué **WinEdit**.

WinEdit 96



Copyright © 1990-96 Steve Schauer

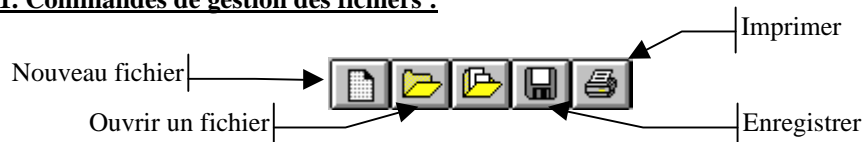
II.1. L'interface graphique de WinEdit :



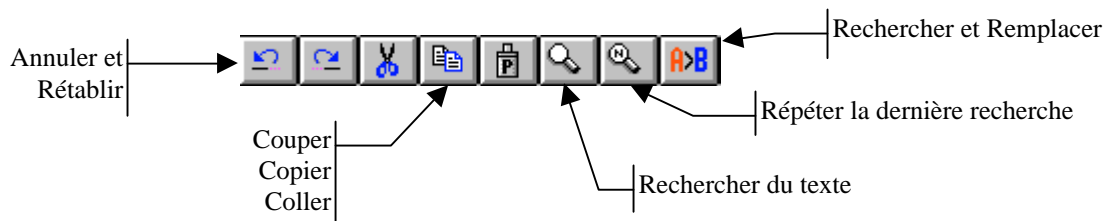
Les mots réservés des différents langages gérés apparaissent avec des couleurs différentes. Les commentaires sont également mis en évidence. Cela permet de repérer rapidement les erreurs de syntaxes et les fautes de frappe.

II.2. Les commandes :

II.2.1. Commandes de gestion des fichiers :

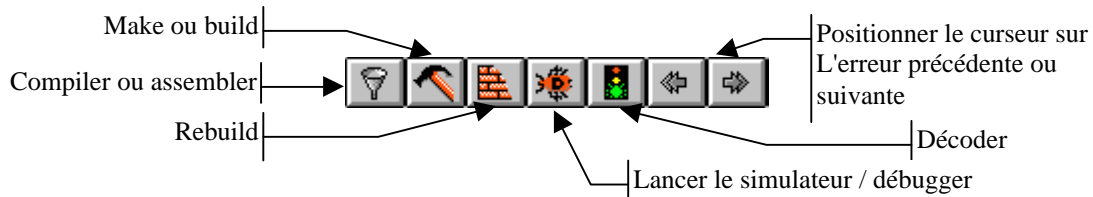


II.2.2. Commandes d'édition :



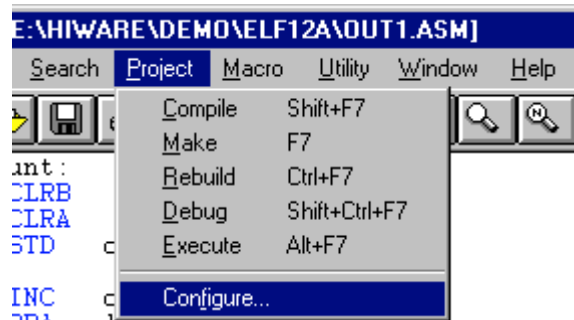
II.2.3. Commandes de mise au point :

Un des avantages de WinEdit est de pouvoir lancer la compilation ou l'assemblage depuis son interface graphique à l'aide d'icônes ou d'un menu.

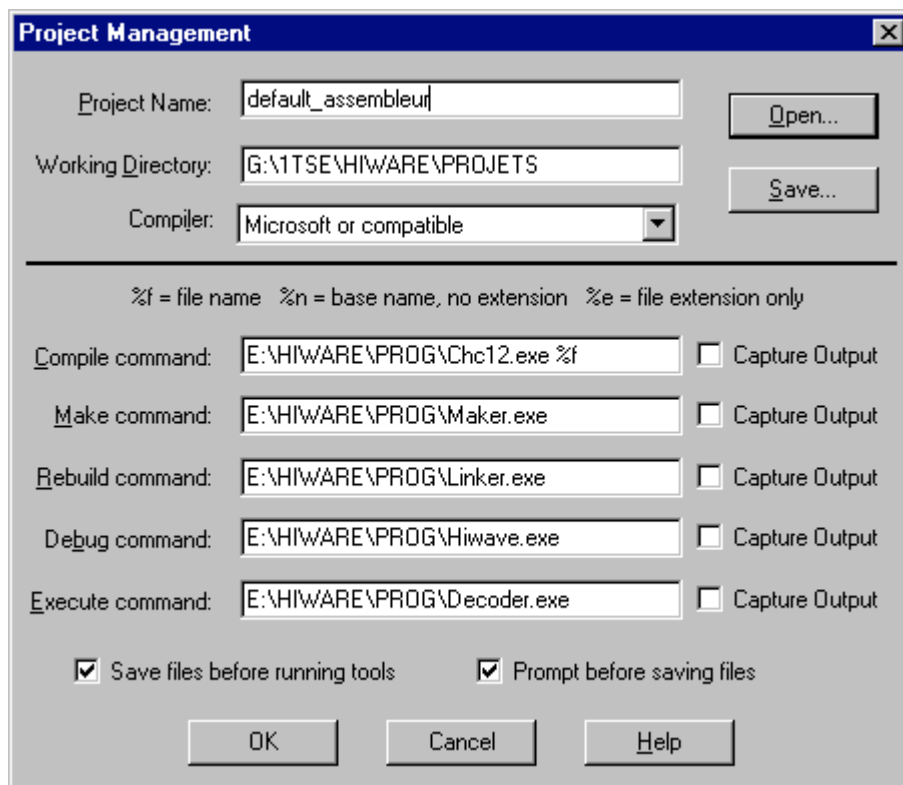


C'est à l'aide de ces icônes que l'on lance les outils correspondants.

Pour définir les outils à lancer, il faut lancer la configuration à l'aide du menu projet :



La fenêtre suivante apparaît :



On peut définir un nom de projet. Attention, le projet est en fait la sauvegarde de la configuration de WinEdit mais pas du projet Hiware.

Pour utiliser la commande **compile** avec un programme assembleur, il faut changer la ligne " Compile command :" en remplaçant **Chc12.exe** (qui est l'exécutable du compilateur C) par **Ahc12.exe**.

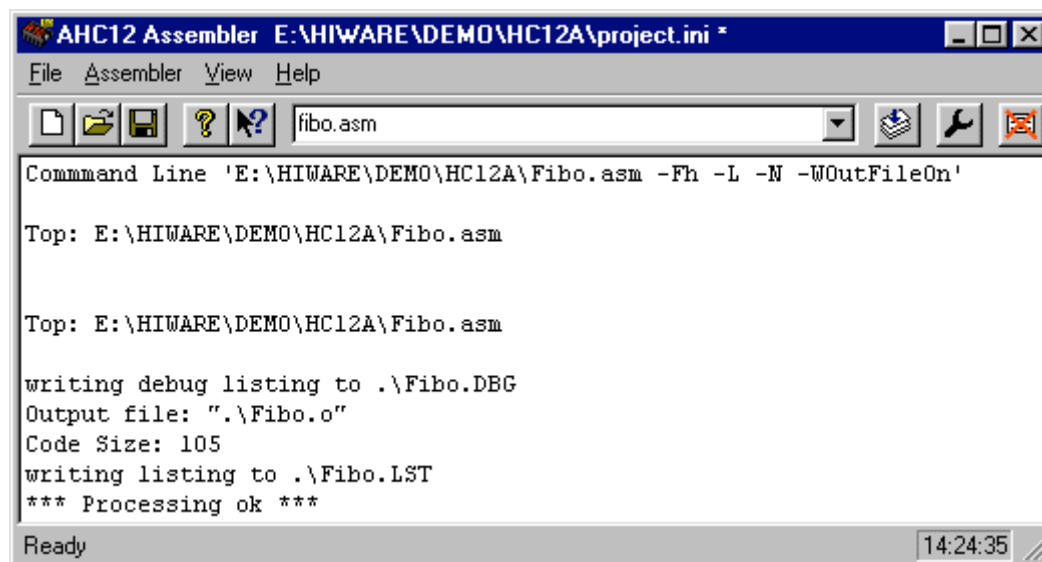
On peut ensuite enregistrer la configuration dans le répertoire du projet Hiware. (fichier *.vpj).

III. L'ASSEMBLEUR :

Pour démarrer l'assembleur, on peut soit cliquer sur l'icône de l'assembleur dans le gestionnaire de projet, soit cliquer sur l'icône de compilation dans WinEdit (si l'outil correspondant à été changé dans la configuration de WinEdit).

III.1. L'interface graphique de l'assembleur :

Lorsque l'assembleur est lancé sans avoir spécifié un nom de fichier dans la ligne de commande, la fenêtre principale de l'assembleur apparaît. Si le paramètre %f est ajouté à la ligne de commande dans l'éditeur de texte WinEdit, l'assembleur se lance avec le fichier qui était édité et l'assemble sans faire apparaître l'interface graphique...



Elle se présente comme toutes les applications Windows, avec une barre de titre, de menu, d'outils et d'état. La zone de texte affiche les différentes informations concernant le processus en cours. Ces informations sont :

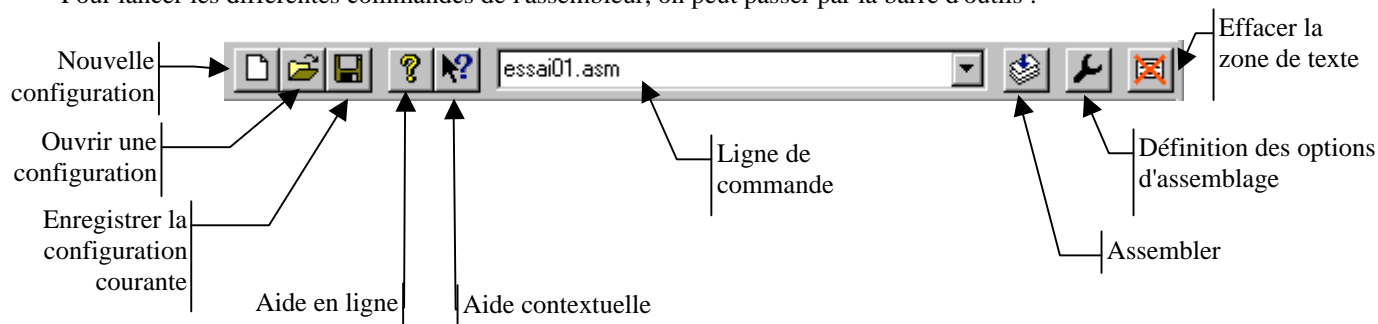
- le nom du fichier assemblé ;
- le nom complet des fichiers effectivement assemblés, fichier principal et tous les fichiers inclus ;
- le nom et le type des fichiers créés : (ex : debug listing (*.DBG) : fichiers listing pour la simulation) ;
- la liste des erreurs, warning et informations générés lors de l'assemblage ;
- la taille du code du fichier objet créé.

Des menus popup contextuels sont accessibles d'un clique droit de la souris dans la zone de texte. Ceux-ci permettent :

- d'obtenir l'aide principale ;
- d'obtenir une aide contextuelle sur l'élément pointé ou sélectionné par la souris. On peut, par exemple, obtenir de l'aide sur une erreur générée à l'assemblage ;
- d'ouvrir soit le fichier pointé ou sélectionné par la souris, soit le fichier source contenant l'erreur sélectionnée. Le fichier s'ouvre alors dans l'éditeur de texte WinEdit et le curseur est positionné sur la ligne contenant l'erreur ;
- de copier en mémoire le texte sélectionné.

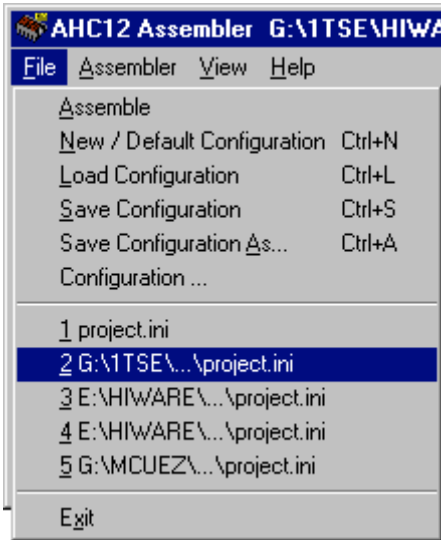
III.2. La barre d'outils :

Pour lancer les différentes commandes de l'assembleur, on peut passer par la barre d'outils :



Au démarrage de l'assembleur, la ligne de commande contient le nom du dernier fichier assemblé en passant par la ligne de commande. Si le dernier fichier a été assemblé sans passer par celle-ci, il n'apparaîtra pas dans la liste des dernier fichiers assemblés.

III.3. Le menu fichier / lancement de l'assemblage :



Pour assembler un fichier source, deux solutions sont possibles :

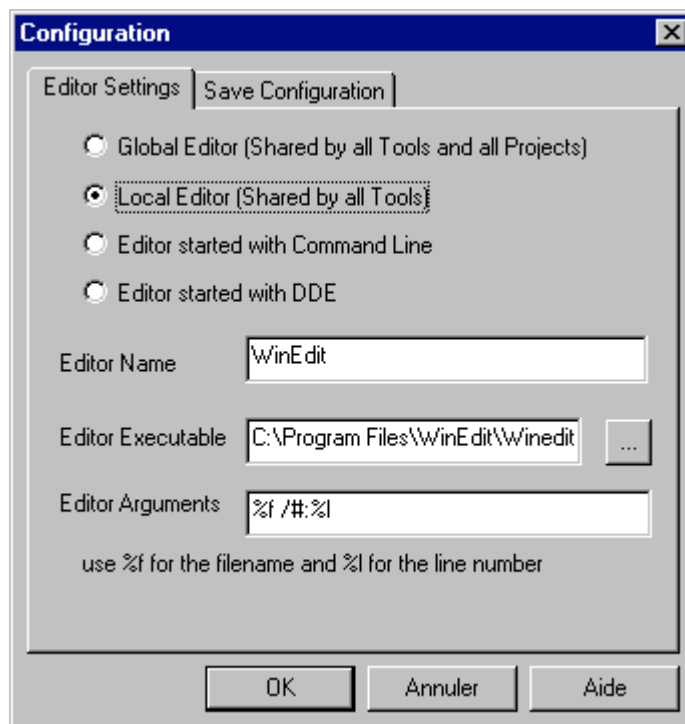
- Entrer le nom du fichier dans la ligne de commande de la barre d'outils puis sélectionner l'icône "assembler". Le fichier doit alors être dans le répertoire du projet ;
- Sélectionner la commande "Assemble" du menu fichier. Une fenêtre apparaît alors permettant de sélectionner le fichier dans le répertoire du projet. L'assemblage est lancé une fois le fichier sélectionné.

A partir de ce menu il est également possible de :

- Créer, ouvrir et sauver la configuration de l'assembleur pour le projet en cours ;
- Ouvrir une configuration au choix parmi les dernières ouvertes ;
- Définir la configuration :

Définition de la configuration :

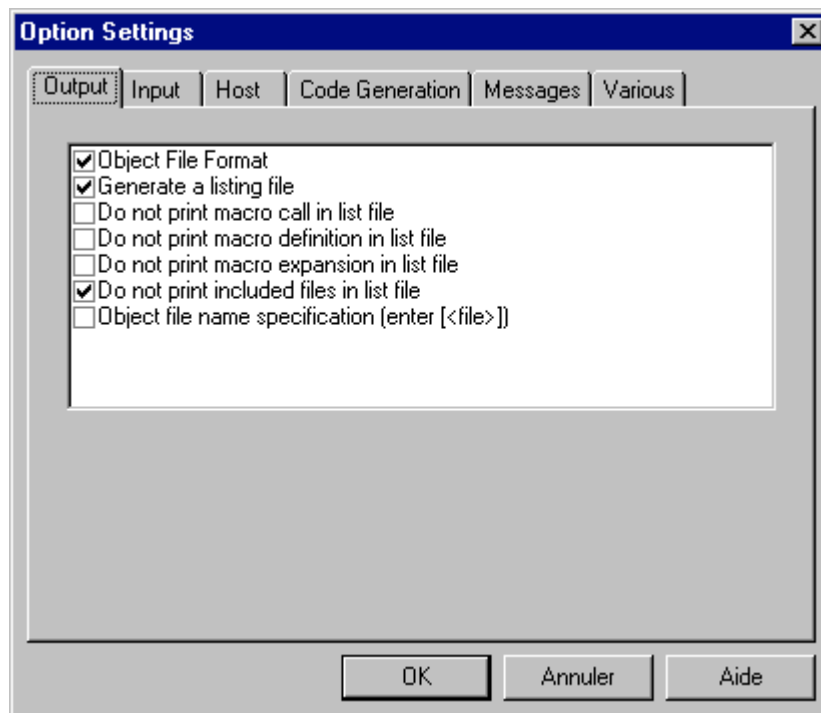
Elle se définit à l'aide de la fenêtre suivante :



On peut définir l'éditeur de texte qui sera utilisé pour éditer le fichier source afin de corriger les éventuelles erreurs. L'éditeur que nous utilisons est WinEdit, donc on pourra vérifier que notre configuration est semblable à celle-ci. Dans l'onglet "Save Configuration", on choisit les éléments qui devront être sauves dans la configuration. On peut, par exemple, choisir de d'enregistrer les options d'assemblages qui auront été définies précédemment.

III.4. Les options d'assemblage :

Pour définir les options d'assemblage, il faut sélectionner la commande "Options" du menu "Assembler" ou cliquer sur l'icône correspondant dans la barre d'outils. La fenêtre suivante apparaît :



Les options d'assemblages sont réparties en six groupes. On définit une option en cochant la ligne correspondante et en complétant éventuellement d'autres informations correspondant à l'option choisie. Chaque option cochée correspond en fait à un paramètre passé à la ligne de commande lançant l'assemblage. Exemple :

Command Line 'fib0.asm -Fh -L -N -WOutFileOn'

III.4.1. Options "Output" :

Ce sont les options de contrôle des fichiers de sorties :

- "Object File Format" : Création du fichier objet. Trois formats de fichier objet sont possibles :
 Hiware : format propriétaire de HIWARE ;
 Elf/Dwarf 2.0 ou Compatible : format permettant aux fichiers objets d'être utilisés par d'autres logiciels que ceux d'Hiware ;
 On a aussi la possibilité de créer directement le fichier exécutable .ABS.
- "Generate a listing file" : Création d'un fichier listing contenant toutes les informations d'assemblage, les macros sont développées, le code des instructions affiché ainsi que les adresses relatives ou absolues. Les adresses non résolues sont notées par des XXXX et seront complétées par l'éditeur de liens.
- "Do not print ..." : Ce sont des options de contrôle de la génération du fichier listing.
- "Object file name specification" : Permet de définir le nom du fichier objet lorsque l'on veut qu'il soit différent du nom du fichier source (nom_source.asm donne par défaut nom_source.o).

III.4.2. Options "Input" :

Ce sont les options relatives aux fichiers sources :

- "Case insensitivity on label names" : L'assembleur ne fera pas la différence entre les majuscules et les minuscules dans les noms d'étiquettes. Option disponible uniquement si l'on génère directement un fichier exécutable .abs.
- "Support for structured types" : Option intéressante uniquement lors de projet contenant des fichiers sources assembleur et en langage C-ansi;.

III.4.3. Options "Host" :

"Set environment variable" : Permet de définir une variable d'environnement.

III.4.4. Options "Code generation" :

L'option "Memory model" permet de spécifier le mode d'utilisation du 68HC12, c'est à dire, avec le modèle de mémoire réduit "SMALL" correspondant aux 64ko, ou avec le modèle de mémoire étendu "BANK" ou paginée. Il est important de préciser le modèle de mémoire lorsque l'on utilise surtout lorsque le projet contient des fichiers sources assembleur et C afin que l'assembleur et le compilateur utilisent un modèle de mémoire identique.

III.4.5. Options "Messages" :

C'est ici que l'on définira les options d'affichages des messages d'erreurs et warning et messages d'informations. On notera principalement les options suivantes :

- "Show notification box in case of error" : Une fenêtre d'information apparaît si des erreurs sont détectées à l'assemblage. C'est particulièrement utile lorsqu'on lance l'assemblage sans ouvrir l'interface graphique (en donnant le nom du fichier source en paramètre...).

III.4.6. Options "Various" :

Ce sont les options de compatibilité avec d'autres assembleurs : MCUasm et Avocet.

III.5. Organisation de la mémoire :

L'organisation de la mémoire dépend du système dans lequel le programme va être implanté. Pour développer celui-ci il est nécessaire de connaître le plan mémoire du système. Toutefois, la mémoire physique est généralement constituée par un espace ROM, un espace RAM, espace réservé aux registres internes du microprocesseur et un espace d'entrées-sorties. Ces espaces sont soit accessibles en lecture, soit en lecture et écriture. Donc, physiquement, l'espace mémoire est déjà divisé en différentes parties.

On peut également segmenter l'espace mémoire physique en segments plus petits et indivisibles. Ces segments permettront d'organiser la mémoire pour y distinguer des espaces particuliers et définis par l'utilisateur. On peut ainsi définir un segment CODE où sera implanté le programme, avec une taille définie, dans la mémoire ROM du système. Les segments n'ont pas à être adjacents et ne doivent pas se chevaucher. Au niveau du langage assembleur, on utilise le terme de section, plutôt que segment, pour ne pas les confondre avec les segments de mémoire définis à l'édition de liens.

III.5.1. Attributs des sections :

Les sections peuvent avoir un de ces trois attributs, en fonction de ce qu'elles devront contenir :

- **Sections de code :**
Ce sont des sections qui seront allouées dans l'espace ROM du microprocesseur. Il ne devra y avoir que des instructions ou éventuellement des constantes.
- **Sections de constantes :**
Ce sont des sections qui seront allouées dans l'espace ROM du microprocesseur. On utilisera cet attribut pour différencier les constantes et les instructions du programme.
- **Sections de variables :**
Ces sections seront allouées dans l'espace RAM du microprocesseur. On utilisera cet attribut pour spécifier où les variables seront stockées.

III.5.2. Types des sections:**III.5.2.1. Sections absolues :**

L'adresse de base d'une section absolue est connue dès l'assemblage. Une section absolue est définie à l'aide de la directive **ORG**. L'opérande suivant le ORG détermine l'adresse de base de la section.

Exemple :

```

XDEF entry
ORG $A00 ; Section de constantes absolu.
cst1:DC.B $A6
cst2:DC.B $BC
...
ORG $800 ; Section de variables absolu.
var: DS.B 1
ORG $C00 ; Section de code absolu.
entry:
LDAA cst1 ; Charge la valeur de cst1
ADDA cst2 ; Additionne la valeur de cst2
STAA var ; range le résultat dans var
BRA entry

```

Quand on utilise des sections absolues, c'est au programmeur de s'assurer qu'il n'y a pas de chevauchement entre les différentes sections définies dans le projet. La directive DC (Define Constant) située dans une section définie par ORG, permet de reconnaître que c'est une section de constantes absolue. La directive DS (Define Space) permet de réserver de l'espace mémoire dans une section de variable absolue. Lorsque des instructions sont placées à la suite d'un ORG, c'est donc que la section est une section de code absolue.

III.5.2.2. Sections relogeables :

L'adresse de base d'une section relogeable est définie au moment de l'édition de liens, grâce aux informations enregistrées dans le fichier de configuration de l'éditeur de liens. Une section relogeable est définie grâce à la directive SECTION :

```
XDEF entry
constSec: SECTION ; Section de constantes relogeable.
cst1:    DC.B $A6
cst2:    DC.B $BC
...

dataSec: SECTION ; Section de variable relogeable.
var: DS.B 1

codeSec: SECTION ; Section de code relogeable.
entry:
LDAA cst1 ; Charge la valeur de cst1
ADDA cst2 ; Additionne la valeur de cst2
STAA var ; range le résultat dans var
BRA entry
```

III.5.2.3. Avantages des sections relogeables :

Les avantages des sections relogeables sont divers. Le premier est qu'elles permettent une grande modularité dans la conception des programmes, ce qui permet de diviser facilement le programme en plusieurs sections qui peuvent être distribuées dans plusieurs fichiers sources. Et cela sans trop se soucier de l'endroit où vont être mémorisés physiquement ces portions de code. De ce fait, cette modularité permet la conception d'un programme par plusieurs personnes. Chaque personne ne s'occupant que de certaines parties. Les seules choses que chaque développeur devra connaître des autres, sont les noms des fonctions et des variables qui sont utilisées et développées. Il les inclura dans son fichier source et la correspondance entre les différents modules sera faite au moment de l'édition des liens.

De plus, lorsque le programme, plus tard, devra être modifié, on aura pas à se soucier de l'espace mémoire disponible dans la limite, évidemment, de la taille mémoire physiquement disponible. On pourra donc modifier le source et vérifier pendant l'édition de liens s'il y a suffisamment de place en mémoire. Enfin, puisque les problèmes de mémoires sont résolus à l'édition de liens, le programme est alors portable. C'est à dire qu'il pourra être utilisé dans un autre système possédant un plan mémoire différent.

III.6. Syntaxe :

Sans entrer dans les détails de la syntaxe utilisée par cet assembleur, nous n'allons présenter ici que les spécificités de la syntaxe de celui-ci. Comme dans tous les assembleurs, une ligne de code est composée de 3 ou 4 champs que l'on peut séparer seulement par un espace, mais on préférera les séparer à l'aide d'une ou de plusieurs tabulations. On s'arrangera pour que les champs soient tous alignés afin d'améliorer la lisibilité du programme.

III.6.1. Le champs des étiquettes :

Les étiquettes se trouvent dans le premier champs. Pour éviter de les confondre avec les noms de macros générés par le macro-assembleur, les noms d'étiquettes ne doivent pas commencer par "_". Ce champs est aussi utilisé pour la définition des symboles de sections, de constantes et de variables.

III.6.2. Le champs mnémonique :

Ce champs contient les mnémoniques des différentes instructions du microcontrôleur ainsi que les directives d'assemblages et les instructions du macro-assembleur.

III.6.3. Le champs opérande :

Le champs opérande peut faire référence à l'opérande d'une instruction mais également :

- la valeur donnée à un symbole déclaré par la directive EQU ;
- l'adresse d'une section de programme ou de données absolue précisé par la directive ORG ;
- le contenu des constantes définies par DC ;
- l'espace mémoire réservé pour les variables défini par la directive DS ;

III.6.4. Les modes d'adressages :

Les modes d'adressages et leur notation sont résumés dans le tableau suivant :

Mode d'adressage	Notation
Inhérent	Pas d'opérande
Direct	< adresse 8-bit >
Étendu	< adresse 16-bit >
Relatif	< relatif PC, offset 8-Bit > ou < relatif PC, offset 16-Bit >
Immédiat	#< expression immédiate 8-bit > ou #< expression immédiate 16-bit >
Indexé, 5-bit offset	<5-bit offset>, xysp
Indexé, pré-décrément	<3-bit offset>, -xys
Indexé, pré-incrément	<3-bit offset>, +xys
Indexé, post-décrément	<3-bit offset>, xys-
Indexé, post-incrément	<3-bit offset>, xys+
Indexé, offset accumulateur	abd, xysp
Indexé, 9-bit offset	<9-bit offset>, xysp
Indexé, 16-bit offset	<16-bit offset>, xysp
Indexé - Indirect, 16-bit offset	[<16-bit offset>, xysp]
Indexé - Indirect, offset accumulateur D	[D, xysp]

Avec les notations suivantes :

- **xysp** : pour un des registres X, Y, SP, PC ou PCR ;
- **xys** : pour un des registres X, Y ou SP ;
- **abd** : pour un des accumulateurs A,B ou D.

III.6.5. Les symboles :

Les symboles permettent d'identifier les variables, les constantes ou les sections.

Pour que les symboles utilisés dans une partie de programme puisse être utilisé dans une autre partie située dans un fichier source différent, il faut le définir comme étant externe grâce à la directive XDEF. XDEF permet de déclarer un symbole pour un fichier donné. De même, pour pouvoir utiliser un symbole défini dans un autre fichier source, il faut le déclarer comme étant une référence à un symbole externe par la directive : XREF.

III.6.6. Le type des constantes :

Les constantes gérées par l'assembleur peuvent être de deux types :

- type entier :
 - décimal : 5 ; 512 ; 1024... ;
 - hexadécimal : \$5 ; \$200 ; \$400... ;
 - octal : @5 ; @1000 ; @2000... ;
 - binaire : %101 ; %1000000000 ; %100000000000...

Par défaut, le format des entiers est décimal.
- type chaîne de caractères :

Elles sont composées par des caractères ASCII délimités par des " ou des ' , exemple : "Abcd" ; 'Abcd' ;

III.6.7. Les opérateurs et les expressions:

Les différents opérateurs arithmétiques et logiques sont reconnus par l'assembleur. Ils peuvent être utilisés dans les opérandes des instructions et des directives ainsi que pour former les expressions. Pour plus de précisions, on se référera à la documentation en ligne.

III.6.8. Les directives d'assemblages :

Il existe différents groupes de directives d'assemblage. Nous nous intéresserons qu'à celles que nous utiliserons le plus souvent. On pourra obtenir des informations sur les autres par l'aide en ligne.

III.6.8.1. Définition de sections:

Elles ont été vues précédemment :

Directive	Description
ORG	Définit une section absolue
SECTION	Définit une section relogeable
OFFSET	Définit une section d'offset

III.6.8.2. Définition de constantes :

Il y en a deux :

- EQU : Associe une étiquette à la valeur qui suit la directive. Le symbole est défini définitivement.
 SET : Même chose sauf que le symbole peut être redéfini.

III.6.8.3. Allocation de donnée :

Ces directives réservent de l'espace mémoire à l'endroit où les données seront rangées. Elles peuvent aussi permettre la définition de constante :

Directive	Description
DC	Définit une constante
DCB	Définit un block de constantes
DS	Réserve de l'espace mémoire pour une variable

Elles sont complétées par un paramètre qui permet de préciser la taille de la donnée à mémoriser :

- .B : réserve 1 octet ;
 .W : réserve 2 octets ;
 .L : réserve 4 octets.

- Exemples : label : DC.B 55,255 réserve 2 octets de mémoire consécutifs à partir de l'adresse de 'label' et leur affecte la valeur 55 et 255 ;
 Lab2 : DC.W "ABCD" réserve 2 mots de 16 bits consécutifs à partir de Lab2 et leur affecte les valeurs ascii équivalentes de "AB" et "CD"
 lab3 : DCB.B 3,\$FF Réserve un block de 3 octets et affecte la valeur \$FF à chaque emplacement.
 Lab4 : DS.W 3 Réserve 3 mots consécutifs à partir de l'adresse de lab4 ;

III.6.8.4. Importation et exportation des symboles :

Ces directives permettent de définir et d'utiliser des symboles externes. Elles sont traitées au moment de l'édition de liens :

Directive	Description
ABSENTRY	Définit le point d'entrée du programme pour la génération directe du fichier absolu (.ABS)
XDEF	Rend un symbole public (utilisable par d'autres modules)
XREF	Importe la référence à un symbole externe (permet d'utiliser un symbole d'un autre module)
XREFB	Importe la référence à un symbole externe situé dans la mémoire de page direct.

III.6.8.5. Contrôle de l'assemblage :

Directive	Description
ALIGN	Définit une contrainte d'alignement
BASE	Spécifie une adresse de base pour la définition d'une constante
END	Définit la fin d'un module
EVEN	Définit une contrainte d'alignement de 2 octets
FAIL	Génère un message d'erreur ou de warning défini par l'utilisateur
INCLUDE	Inclut le texte provenant d'un autre fichier
LONGEVEN	Définit une contrainte d'alignement de 4 octets

III.6.8.6. Contrôle du fichier listing :

Ces directives permettent de contrôler ce qui apparaîtra dans le fichier listing :

CLIST, LIST, LLEN, MLIST, NOLIST, NOPAGE, PAGE, PLEN, SPC, TABS, TITLE.

III.6.8.7. Contrôle des macros :

Directive	Description
ENDM	Fin de macro
MACRO	Début de macro
MEXIT	Sortie de macro

III.6.8.8. Assemblage conditionnel :

Directive	Description
ELSE	
ENDIF	Fin d'un block conditionnel.
IF	Début d'un block conditionnel. Une expression logique doit suivre cette directive.
IFC	Teste si deux expressions de texte sont égales.
IFDEF	Teste si un symbole est défini.
IFEQ	Teste si une expression est nulle.
IFGE	Teste si une expression est supérieure ou égale à 0.
IFGT	Teste si une expression est supérieure à 0.
IFLE	Teste si une expression est inférieure ou égale à 0.
IFLT	Teste si une expression est inférieure à 0.
IFNC	Teste si deux expressions de texte sont différentes.
IFNDEF	Teste si un symbole n'est pas défini.
IFNE	Teste si une expression est non nulle.

III.6.9. Les macros :

Les macros permettent de définir un nom pour une partie de code qui pourra être utilisé à plusieurs reprises dans le programme. Ce ne sont pas des sous-programmes mais elles peuvent accepter des paramètres. Elles sont traitées par le macro-assembleur qui interprète toutes les directives d'assemblage. Celui-ci remplace toutes les occurrences de la macros par le segment de code associé et défini au préalable.

III.6.9.1. Définition :

La définition d'une macro comprend quatre parties :

- Définition du nom de la macro à l'aide de la directive MACRO ;
- Le corps de la macro consistant en une suite séquentielle d'instructions assembleur pouvant aussi contenir des arguments ;
- La directive ENDM qui définit la fin de la macro ;
- Éventuellement des directives de sortie de macro peuvent être utilisées dans le corps de la macro à l'aide de la directive MEXIT.

Les macros peuvent utiliser des arguments afin de les rendre paramétrable et ainsi le code remplacé par le macro-assembleur pourra être contextuel.

III.6.9.2. Appels :

La syntaxe de l'appel d'une macro est la suivante :

[label :] nom [.taille_des_arguments] [argument1 [,argument2] ...]

Toutes les macros doivent être définies avant leur appel. Le nom de la macro doit apparaître dans le champs mnémonique. Les arguments de la macro doivent être placés dans le champs opérande séparés par des virgules. L'appel de la macro produit du code assembleur à l'endroit de l'appel. Le code est ensuite assemblé comme le reste du programme.

III.6.9.3. Arguments :

Les macros peuvent utiliser jusqu'à 36 différents arguments. Ces paramètres seront remplacés par leur valeur donnée au moment de l'appel. Un argument est défini par un antislash (\), suivi par un chiffre (0 à 9) ou une lettre en majuscule (A à Z). L'argument \0 est spécifique et correspond à la taille des arguments qui suit le nom de la macro.

Exemple :

La macro suivante :

```
MaMacro :  MACRO
           DC.\0      \1,\2
           ENDM
```

Appelée par la ligne suivante :

```
MAMacro.B $10,$56
```

Produira à l'assemblage le code suivant :

```
DC.B      $10,$56
```

III.6.9.4. Étiquettes dans les macros :

Pour éviter le problème des étiquettes définies plusieurs fois parce qu'utilisée dans une macro, on demande à l'assembleur de définir un nom unique d'étiquette à chaque appel de la macro. Pour cela, les étiquettes définies dans une macros doivent l'être à l'aide de \@. Chaque étiquette créée à partir d'une macro et définie à l'aide de \@label sera définie dans le code assembleur par _nnnnnlabel. "nnnnn" est un nombre à 5 chiffre généré automatiquement par l'assembleur en commençant par 00000.

Exemple :

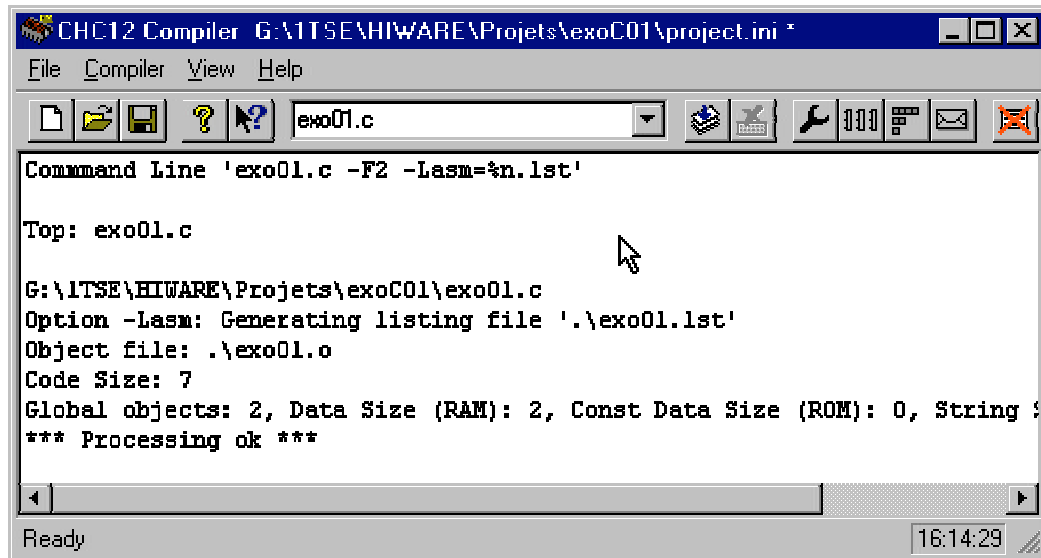
Ce code :	Sera remplacé par :
Clear : MACRO	LDX temporary
LDX \1	LDAB #16
LDAB #16	_00001loop : CLR 0,X
\@loop : CLR 0,X	INX
INX	DECB
DECB	BNE _00001loop
BNE \@loop	LDX data
ENDM	LDAB #16
Clear temporary	_00002loop : CLR 0,X
Clear data	INX
	DECB
	BNE _00002loop

IV. LE COMPILATEUR C / C++ :

Pour démarrer le compilateur, on peut soit cliquer sur l'icône du compilateur dans le gestionnaire de projet, soit cliquer sur l'icône de compilation dans WinEdit.

IV.1. L'interface graphique du compilateur :

Lorsque le compilateur est lancé sans avoir spécifié un nom de fichier dans la ligne de commande, la fenêtre principale du compilateur apparaît. Si le paramètre %f est ajouté à la ligne de commande dans l'éditeur de texte WinEdit, le compilateur se lance avec le fichier qui était édité et le compile sans faire apparaître l'interface graphique...



Elle se présente comme toutes les applications Windows, avec une barre de titre, de menu, d'outils et d'état. La zone de texte affiche les différentes informations concernant le processus en cours. Ces informations sont :

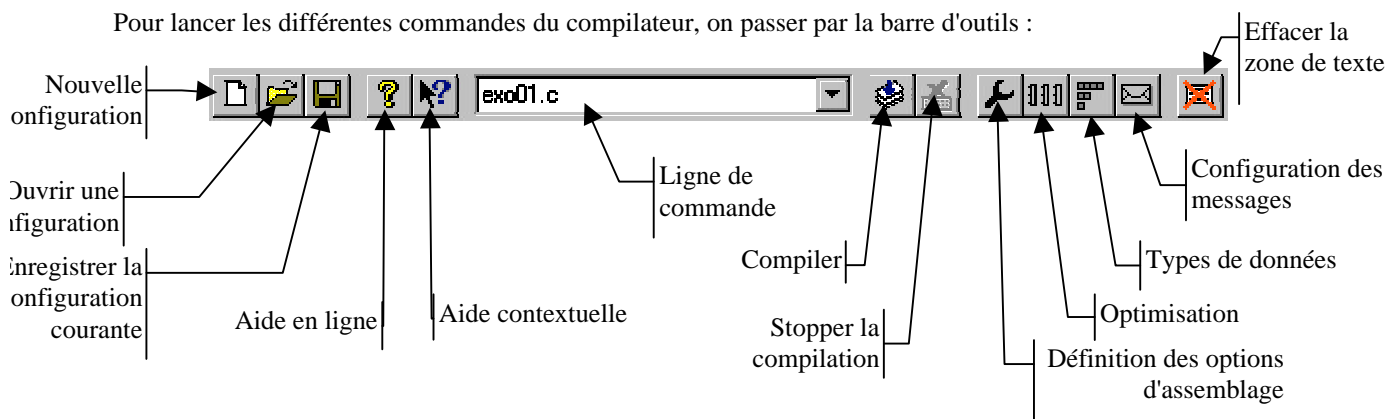
- le nom du fichier compilé ;
- le nom complet des fichiers effectivement compilés, fichier principal et tous les fichiers inclus ;
- le nom et le type des fichiers créés ;
- la liste des erreurs, warning et informations générés lors de la compilation ;
- la taille du code du fichier objet créé.

Des menus popup contextuels sont accessibles d'un clic droit de la souris dans la zone de texte. Ceux-ci permettent :

- d'obtenir l'aide principale ;
- d'obtenir une aide contextuelle sur l'élément pointé ou sélectionné par la souris. On peut, par exemple, obtenir de l'aide sur une erreur générée lors de la compilation ;
- d'ouvrir soit le fichier pointé ou sélectionné par la souris, soit le fichier source contenant l'erreur sélectionnée. Le fichier s'ouvre alors dans l'éditeur de texte WinEdit et le curseur est positionné sur la ligne contenant l'erreur ;
- de copier en mémoire le texte sélectionné.

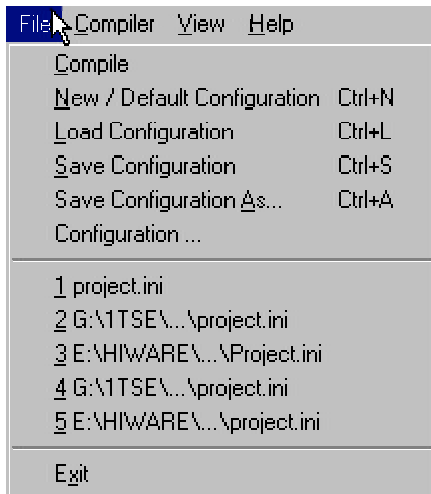
IV.2. La barre d'outils :

Pour lancer les différentes commandes du compilateur, on passe par la barre d'outils :



Au démarrage de l'assembleur, la ligne de commande contient le nom du dernier fichier assemblé en passant par la ligne de commande. Si le dernier fichier a été assemblé sans passer par celle-ci, il n'apparaîtra pas dans la liste des derniers fichiers assemblés.

IV.3. Le menu fichier / lancement de la compilation :



Pour compiler un fichier source, deux solutions sont possibles :

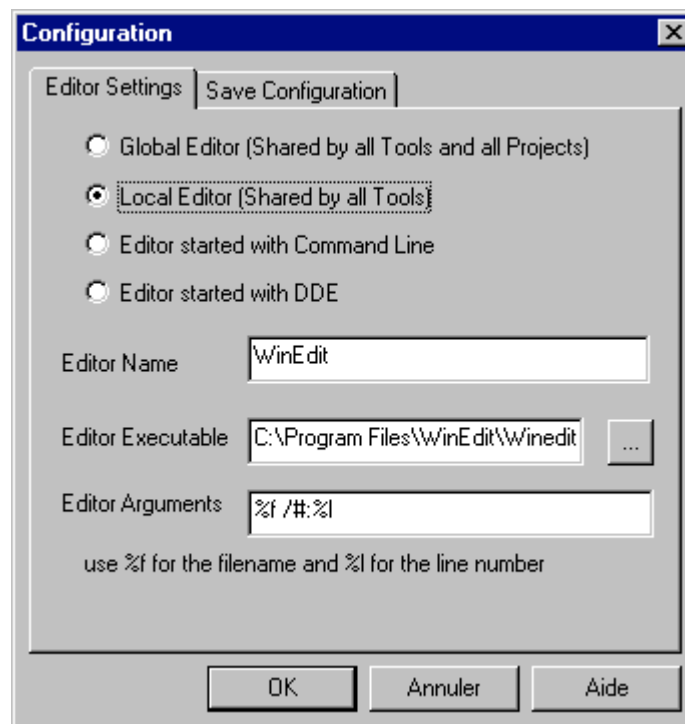
- Entrer le nom du fichier dans la ligne de commande de la barre d'outils puis sélectionner l'icône "compiler". Le fichier doit alors être dans le répertoire du projet ;
- Sélectionner la commande "Compile" du menu fichier. Une fenêtre apparaît alors permettant de sélectionner le fichier dans le répertoire du projet. La compilation est lancée une fois le fichier sélectionné.

A partir de ce menu il est également possible de :

- Créer, ouvrir et sauver la configuration du compilateur pour le projet en cours ;
- Ouvrir une configuration au choix parmi les dernières ouvertes ;

Définir la configuration :

Elle se définit à l'aide de la fenêtre suivante :



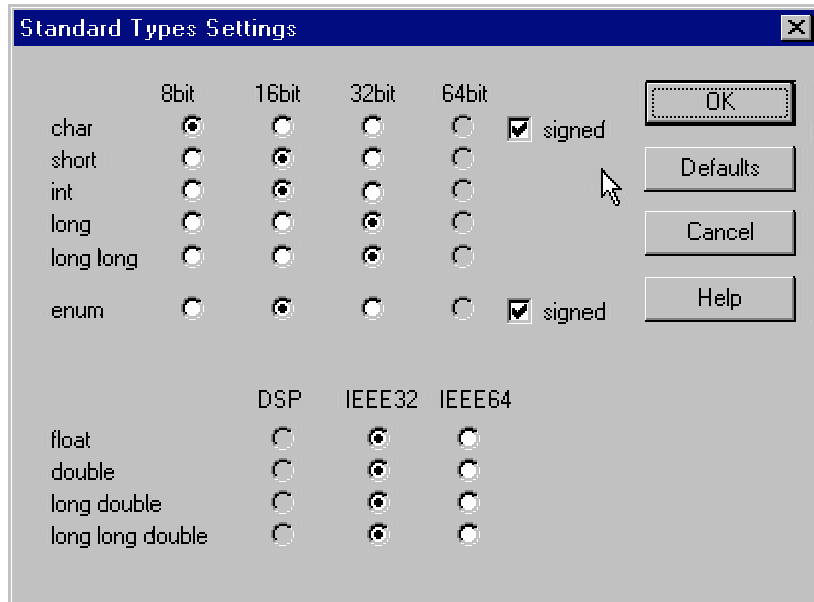
On peut définir l'éditeur de texte qui sera utilisé pour éditer le fichier source afin de corriger les éventuelles erreurs. L'éditeur que nous utilisons est WinEdit, donc on pourra vérifier que notre configuration est semblable à celle-ci. Dans l'onglet "Save Configuration", on choisit les éléments qui devront être sauves dans la configuration. On peut, par exemple, choisir de enregistrer les options de compilation qui auront été définies précédemment.

IV.4. Configuration du compilateur :

Plusieurs paramètres peuvent être changés pour influencer le code produit par le compilateur. Ces paramètres sont accessibles par le menu "**Compiler / Options...**".

IV.4.1. Paramétrage des types standards :

On peut définir le nombre de bits associés à chaque type standard du C-ANSI :




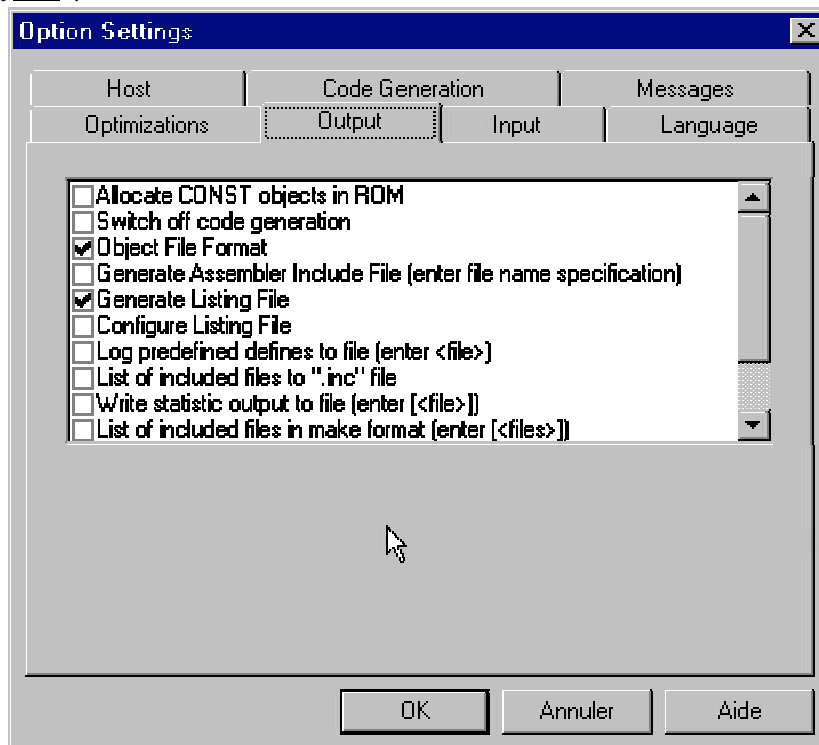
En temps normal, il n'y a pas à changer ces paramètres. Dans le cas où ils ont à être changés, les règles suivantes doivent être respectées :

```
sizeof(char)  <= sizeof(short)
sizeof(short) <= sizeof(int)
sizeof(int)   <= sizeof(long)
sizeof(long)  <= sizeof(long long)
sizeof(float) <= sizeof(double)
sizeof(double)<= sizeof(long double)
sizeof(long double) <= sizeof(long long double)
```

Les valeurs par défaut des différents types standards du C-ANSI dépendent du microprocesseur cible.

IV.4.2. Les options de compilation :

Pour définir les options de compilation, il faut sélectionner la commande "**Compiler / Options / Advanced**" ou cliquer sur l'icône correspondant  :



Les options de compilation sont réparties en sept groupes. On définit une option en cochant la ligne correspondante et en complétant éventuellement d'autres informations correspondant à l'option choisie. Chaque option cochée correspond en fait à un paramètre passé à la ligne de commande lançant la compilation. Exemple :

Command Line 'exo01.c -F2'

IV.4.2.1. Options "Optimizations" :

Ces options contrôlent l'optimisation du code généré. On se référera à la documentation en ligne pour plus de détails sur l'optimisation du code. On pourra, par exemple, retenir ces options :

- **"No integral promotion on characters" :** Permet d'augmenter la densité du code. En C-ANSI, les calculs sur les variables de type *char* se font en sur des entiers. Il y a donc une conversion implicite de *char* vers *int* et les calculs se font alors sur des variables 16bits au lieu de 8... En sélectionnant cette option, cette conversion n'est plus faite. Par contre, dans ce cas, le code n'est plus compatible avec le C-ANSI.
- **"Main Optimize Target" :** Permet de sélectionner le type d'optimisation principale. On peut choisir de donner la priorité à la taille du code ou à la vitesse d'exécution... Par défaut, le compilateur optimise le code par rapport à sa taille.
- **"Optimize Dead Assignments" :** Le compilateur retire les assignements de variables qui ne sont pas utilisées après celui-ci.
- **"Try to Keep Loop Induction Variables in Registers" :** Avec cette option, le compilateur essaie de faire en sorte que les variables utilisées comme index dans les boucles soient associées aux registres. Cela permet de réduire le temps d'exécution. On peut indiquer le nombre de registres utilisés.
- **"Allocate local variables into registers" :** Cette option permet de faire en sorte que les variables locales (*char* et *int*) soient allouées dans les registres. Ceci augmente la rapidité d'exécution mais augmente aussi la taille du code...

IV.4.2.2. Options "Output" :

Ce sont les options de contrôle des fichiers de sortie. On pourra retenir :

- **"Allocate Constant Objects into ROM" :** Si cette option n'est pas validée, les variables déclarées avec *const*, sont traitées comme les autres variables. Avec, cette option, ces constantes sont placées dans le segment ROM_VAR qui sera assigné à une ROM. Cette option n'est utile que pour le format d'objet *HIWARE*, dans le format *ELF/DWARF*, les constantes sont automatiquement placées dans le segment *".rodata"*. On peut faire la même chose avec le pragma *"INTO_ROM"*.
- **"Object File Format" :** Après compilation, le compilateur écrit le code et des informations pour le débogage dans un fichier objet (*.o). Ce fichier objet peut avoir différents formats. Les formats disponibles sont alors :
 - *HIWARE* : format propriétaire comportant quelques restrictions mais produisant un code plus compacte;
 - *Strict HIWARE V2.7* : autre format propriétaire...
 - *ELF/DWARF* : format supporté par d'autres compilateurs et éditeurs de liens. On a alors le choix entre la compatibilité avec la version 1.1 du format ou avec la version 2.0...
- **"Generate Assembler Include File" :** Option à utiliser lorsque le pragma *"CREATE_ASM_LISTING"* permettant de générer un fichier à inclure dans les sources assembleurs pour faire référence à des objets (fonctions, variables...) définis dans des sources C. C'est avec cette option que l'on définit le nom du fichier généré qui aura l'extension *.inc*.
- **"Generate Listing File" :** Cette option permet de générer directement un fichier listing en assembleur. La configuration de cette opération se fait avec l'option *"Configure Listing File"*.
- **"Include Files in Make File List" :** Cette option permet de créer un fichier texte contenant la liste des fichiers inclus dans le fichier source à l'aide de la directive *#INCLUDE*. Cette liste est au format *make* et on pourra alors la réutiliser tel quel dans un fichier destiné au *maker*.
- **"Object file name specification" :** On peut préciser le nom du fichier objet créé par le compilateur à l'aide de cette option.

IV.4.2.3. Options "Input" :

Ce sont les options relatives aux fichiers sources. On peut à l'aide de l'option **"Include File Path"**, indiquer un chemin supplémentaire pour l'inclusion des fichiers. Ce chemin sera alors prioritaire sur les chemins définis dans les variables d'environnement définies à la configuration du projet. L'option **"Include files only once"** permet d'empêcher que les fichiers en-tête (*.h) soient lus plusieurs fois. En effet, le langage C ne permet pas que des structures (*struct*), les énumérations (*enum*) et les définitions de type (*typedef*) soient définies plus d'une fois. Or les fichiers en-tête incluent souvent d'autres fichiers et l'imbrication des fichiers en-tête entraîne souvent plusieurs ouvertures d'un même fichier. Cette option évite alors les erreurs de compilation dues à ces multiples ouvertures.

IV.4.2.4. Options "language" :

Ces options permettent de contrôler la compilation en fonction du langage utilisé. En effet, le compilateur peut compiler des fichiers sources en langage C, à la norme C-ANSI ou non, ou même en langage C++. Certaines de ces options définissent des variables de compilation qui peuvent être utilisées dans le fichier source pour contrôler la compilation en fonction des différentes options avec lesquelles le compilateur sera lancé. Cela permet d'écrire des programmes portables dans un langage ou dans un autre. On utilise le plus souvent ces variables dans les fichiers destinés à être réutilisés par d'autres programmes, pour les fichiers en-tête et les bibliothèques. On pourra retenir plus particulièrement ces options :

- **"Strict ANSI" :** Cette option force le compilateur à suivre strictement la conversion du langage C-ANSI. Elle définit la variable **"__STDC__"** ;
- **"Enable C++ Support" :** Cette option force le compilateur à se comporter comme un compilateur C++. On a alors le choix entre trois types différents de C++ :
 - **Full ANSI Draft C++ :** supporte la définition du C++ ANSI ;
 - **Embedded C++ (EC++) :** supporte une définition réduite du C++ adaptée aux systèmes embarqués ;
 - **compactC++ (cC++) :** supporte une version du C++ que l'on peut configurer à l'aide de l'option **Disable compactC++ features** ;

IV.4.2.5. Options "Code Generation" :

Ces options permettent au compilateur de générer un code qui prenne en compte des particularités du microprocesseur cible. Certaines options sont toutefois valables pour tous les microprocesseurs. On pourra retenir :

- **"Float IEEE32, doubles IEEE64" :** Par défaut, le codage des *float* et des *double* est à la norme IEEE32. Cette option permet de coder les *float* à la norme IEEE32 et les *double* à la norme IEEE64.
- **"DPAGE / EPAGE / PPAGE Register is used for paging" :** Options particulières au 68HC12, elles permettent de spécifier l'adresse des registres DPAGE, EPAGE et PPAGE pour l'utilisation de la mémoire paginée. Les variables définies par ces options sont : **"__DPAGE__"** ou **"__NO_DPAGE__"**, **"__EPAGE__"** ou **"__NO_EPAGE__"** et **"__PPAGE__"** ou **"__NO_PPAGE__"** ;
- **"Memory Model" :** Le 68HC12 possède deux modèles de mémoire :
 - **"SMALL" :** c'est le plan mémoire par défaut correspondant aux 64 ko accessibles avec les 16 lignes d'adresses. Cette option définit la variable **"__SMALL__"** ;
 - **"BANKED" :** c'est plan mémoire paginé qui permet d'étendre la mémoire au-delà des 64 ko. Cette option définit alors la variable **"__BANKED__"** ;
- **"Use EDIV instruction" :** Le 68HC12 possède une instruction EDIV qui permet de réaliser la division d'une valeur 32bits par une valeur 16bits et donnant le résultat sous forme d'un quotient 16bits et d'un reste 16bits également (La division signée se fait avec EDIVS). Cette option permet d'utiliser cette instruction plutôt qu'une fonction ce qui augmente la rapidité d'exécution et diminue la taille du code. Mais il faut faire attention à son utilisation, car en cas de dépassement de capacité (du quotient entre autre), le résultat n'est alors pas significatif. Le programmeur doit donc s'assurer qu'il n'y a pas de dépassement pour utiliser convenablement cette option ;

- **"Hardware restrictions for specifics CPUs :**

Cette instruction oblige le compilateur à ne pas générer des instructions et des blocs de code qui ne fonctionnent pas avec tous les processeurs disponibles. En effet, le 68HC12 possède des instructions spécifiques : TBNE, TBEQ, IBNE, IBEQ, DBNE, DBEQ ; La variable `"__PROCESSOR_X4__"` est alors définie.

IV.4.2.6. Option "Host" :

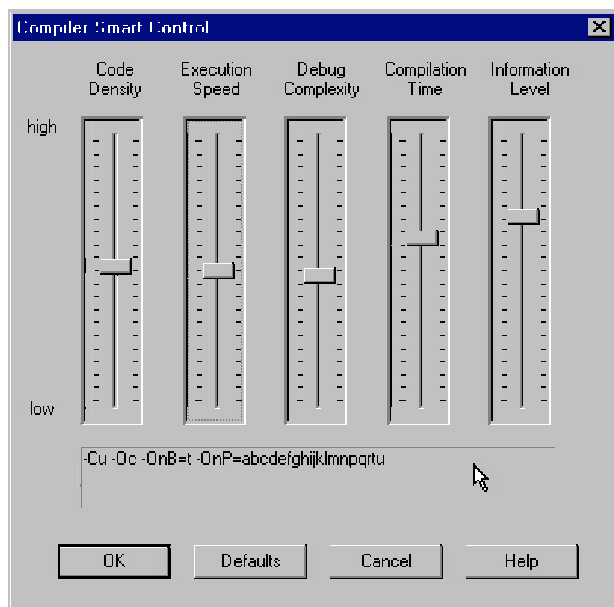
On peut définir une variable d'environnement directement à partir de la ligne de commande du compilateur. Pour cela il suffit de définir cette variables dans l'option **"Set Environment Variable"**.

IV.4.2.7. Options "Messages" :

Ces options permettent de contrôler l'affichage des messages lors de la compilation. On peut ainsi être prévenu d'une erreur de compilation par une fenêtre d'information (**"Show Notification box in case of errors"**, utile principalement lors d'une compilation lancée par le maker). On peut décider de ne pas voir s'afficher les messages d'information et d'avertissement (WARNING). On peut définir le format d'affichage des messages, définir le nombre maximum d'informations, d'avertissements et d'erreurs (dans le cas des erreurs, lorsque le nombre maximum est atteint, la compilation s'arrête).

IV.4.3. Contrôle de l'optimisation, "Smart Sliders" :

L'interface graphique du compilateur HIWARE propose un moyen pratique pour choisir les optimisations appropriées pour une application donnée. Elle fournit une fenêtre de configuration se composant de cinq curseurs permettant de choisir le niveau d'optimisation :

**"Code Density" :**

affiche le niveau de densité du code souhaité, un haut niveau donnera une taille de code plus petite ;

"Execution Speed" :

affiche le niveau de vitesse d'exécution souhaité, un haut niveau donnera un code plus rapide ;

"Debug Complexity" :

affiche le niveau de complexité du code généré, un haut niveau donnera un code plus complexe à déboguer ;

"Compilation Time" :

affiche le niveau de temps de compilation souhaité, un haut niveau donnera un temps de compilation plus long dû à une plus grande optimisation ;

"Information level" :

affiche le niveau d'information souhaité pendant la compilation ;

Un lien direct existe entre les quatre premiers curseurs : lorsque l'on change la position d'un curseur, les trois autres sont mis à jours en relation avec la modification effectuée.

La ligne de commande de la compilation est mise à jours en fonction des options de compilation définies par la position des différents curseurs.

IV.5. Les macros :

Le standard ANSI pour le langage C oblige le compilateur à définir un certain nombre de variables (considérées ici comme des macros). Ces variables peuvent être utilisées pour définir des conditions de compilation et contrôler ainsi la compilation d'un fichier source. Le compilateur fournit des variables prédéfinies :

- `__LINE__` : Numéro de ligne dans le fichier source courant ;
- `__FILE__` : Nom du fichier source ;
- `__DATE__` : Date de compilation ;
- `__TIME__` : Heure de compilation ;
- `__STDC__` : Définie à 1 si l'option du compilateur -ANSI a été choisie (Option Language | Strict ANSI);

Le compilateur fournit également d'autres variables qui dépendent des options de compilations choisies. Les variables suivantes non pas de valeur, mais sont soit définies (test : `#IF DEFINED ***` ou `IFDEF`) soit non définies (test : `#IFNDEF`) :

- `__cplusplus` : Définie si langage C++ ;
- `__CNI__`, `__OPTIMIZE_FOR_TIME__`, `__OPTIMIZE_FOR_SIZE__` : Définies en fonction des options d'optimisation ;
- `__HIWARE_OBJECT_FILE_FORMAT__`, `__ELF_OBJECT_FILE_FORMAT__` : Définies selon l'option choisie ;

D'autres variables sont définies par le compilateur et concernent la taille maximale des objets et la taille des différents types de variables. Ces variables permettent de connaître dans le fichier source et dans les fichiers inclus la taille des objets.

Exemples : `__CHAR_IS_8BIT__`, `__SHORT_IS_16BIT__`, `__INT_IS_16BIT__`, `__LONG_IS_32BIT__` ...

Enfin, les variables `__BITFIELD_MSBIT_FIRST__`, `__BITFIELD_LSBIT_FIRST__`, `__BITFIELD_MSBYTE_FIRST__` et `__BITFIELD_LSBYTE_FIRST__` permettent de connaître comment le compilateur alloue les champs de bits. Cela permet de rendre le code compatible avec d'autres compilateur même si l'allocation des champs de bits se fait différemment. On peut noter que si l'utilisation des champs de bits pour accéder à des registres d'entrées - sorties est possible, elle n'est ni portable ni efficace... Il est recommandé d'utiliser plutôt les opérations régulières au niveau des bit, le ET (AND → `&`) et le OU (OR → `|`) pour l'accès aux ports d'entrées - sorties... Toutefois, pour le 68HC12, les variables `__BITFIELD_LSBIT_FIRST__` et `__BITFIELD_MSBYTE_FIRST__` sont définies par défaut.

IV.6. Les Pragmas :

Un *pragma* est un moyen de passer des informations au compilateur sur la façon d'interpréter les instructions qui le suivent. Les effets d'un *pragma* sur le code généré commencent à partir de sa définition et finissent à la fin de la fonction suivante, sauf pour les *pragmas* **ONCE** et **NO_STRING_CONSTR** qui sont valides jusqu'à la fin du fichier source.

La syntaxe des *pragmas* est :

#pragma nom_du_pragma [arguments_optionnels]

Les *pragmas* suivants sont valables pour tous les microprocesseurs, les *pragmas* spécifiques au 68HC12 seront vus dans le chapitre IV.7.2 Motorola HC12 (page 28).

- **#pragma ONCE :** Lorsqu'il est placé dans un fichier en-tête, celui-ci n'est ouvert et lu qu'une seule fois, ce qui peut accélérer la compilation ;
- **#pragma NO_STRING_CONSTR :** Ce *pragma* annule la prise en compte du caractère "#" comme étant une définition de chaîne de caractères (utilisé dans les macros pour passer des chaînes de caractères en paramètre). Il est tout particulièrement utile lorsqu'une macro doit contenir des instructions assembleur avec un adressage immédiat (exemple : `asm { LDX #$F0 }`). Il est valide pour le reste du fichier source.
- **#pragma NO_RETURN :** Il supprime la génération de l'instruction de retour (retour de sous-programme et d'interruption (RTS et RTI)) pour la fonction définie à la suite. Il peut être utile si le programmeur prévoit lui-même le retour ou si le code doit aller à la première instruction de la fonction suivante.
- **#pragma INLINE :** Il demande au compilateur de développer la fonction suivante dans le fichier source à chaque fois qu'elle apparaît.
- **#pragma INTO_ROM :** Ce *pragma* force la prochaine définition de variable à être constante (de type *const*).
- **#pragma TRAP_PROC :** Ce *pragma* marque la fonction suivante comme étant une fonction d'interruption.

IV.7. Syntaxe du compilateur :

La compilation se réalise en deux étapes principales :

Le code est d'abord interprété en fonction du langage choisit. Lors de cette première étape, appelée "**Front End**", le compilateur lit le fichier source, vérifie la syntaxe et produit une représentation intermédiaire du programme qui est passé à la deuxième étape. La première étape ne dépend pas du microprocesseur cible mais dépend du langage de programmation. Le compilateur d'HIWARE accepte le langage C ANSI (**ANSI-C Front End**) et le langage C++ (**C++ Front End**).

La deuxième étape dépend du microprocesseur cible. Le compilateur génère le code machine correspondant au programme. Cette étape est aussi appelée "**Back End**" (Motorola H12 Back End pour le 68HC12).

IV.7.1. ANSI-C :

Le langage C-ANSI est supporté selon le standard X3J11. Le document de référence est : "*American Standard for Programming Languages - C*", ANSI/ISO 9899-1990.

IV.7.1.1. Mots clés :

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

IV.7.1.2. Directives de compilations :

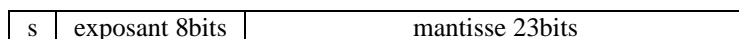
#if, #ifdef, #ifndef, #else, #elif, #endif
 #define, #undef
 #include
 #pragma
 #error, #line

Les opérateurs du pré-processeur defined, # et ## sont également supportés.

IV.7.1.3. Codage des nombres réels :

Les nombres réels correspondent aux type *float* et *double*. Le compilateur supporte trois codages différents pour ces nombres : IEEE32 et IEEE64 pour les types à la norme IEEE, et le format DSP. Ces formats sont décrits ci-après :

Format IEEE 32bits :

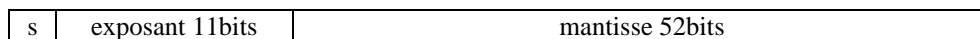


bit de signe

$$\text{valeur} = (-1)^s * 2^{(\text{exp}-127)} * 1, \text{mant}$$

$$5,8 * 10^{-39} < |\text{valeur}| < 3,8 * 10^{38}$$

Format IEEE 64 bits :

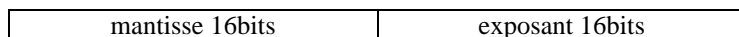


bit de signe

$$\text{valeur} = (-1)^s * 2^{(\text{exp}-1023)} * 1, \text{mant}$$

$$1,12 * 10^{-308} < |\text{valeur}| < 1,797 * 10^{308}$$

Format DSP :



$$\text{valeur} = \text{mant} * 2^{(\text{exp})}$$

Par défaut, le type *float* est codé avec le format IEEE32 et le type *double* avec le format IEEE64.

IV.7.1.4. Champs de bits :

Il n'existe pas de standard pour l'utilisation des champs de bits. Leur allocation dans un octet peut varier d'un compilateur à un autre. De plus, l'utilisation des champs de bits pour accéder aux ports d'entrées sorties n'est ni portable ni efficace. Il est plutôt recommandé d'utiliser des masques de bits à l'aide des opérateurs ET et OU.

Si l'on utilise ces champs de bits, il faudra faire attention à leur type : les champs de bits peuvent être signés et dans ce cas, leur valeur est soit -1 soit 0. On utilisera donc de préférence les champs de bits non signés.

IV.7.1.5. Segmentation de la mémoire :

L'éditeur de liens d'HIWARE permet de diviser l'espace mémoire en plusieurs segments. Le compilateur permet d'attribuer un segment mémoire particulier à un ensemble de variables ou de fonctions qui seront alors placés dans le segment par l'éditeur de liens.

Il existe deux type de segments de base, *code* et *data*, qui peuvent être spécifié à l'aide de ces deux pragmas :

```
#pragma CODE_SEG [SHORT] <nom_du_segment>
```

```
#pragma DATA_SEG [SHORT] <nom_du_segment>
```

Il existe également un troisième pragma pour les constantes :

```
#pragma CONST_SEG [SHORT] <nom_du_segment>
```

Tous ces pragmas sont valides jusqu'au prochain pragma du même type. Si aucun segment n'est précisé, le compilateur utilise deux segments par défaut : DEFAULT_ROM (le segment code par défaut) et DEFAULT_RAM (le segment data par défaut). Pour rendre explicitement ces segments actifs, il suffit d'utiliser le nom de segment DEFAULT :

```
#pragma CODE_SEG DEFAULT
```

```
#pragma DATA_SEG DEFAULT
```

```
#pragma CONST_SEG DEFAULT
```

Les segments peuvent être déclarés avec l'attribut SHORT. Cela oblige le compilateur à utiliser un adressage court (adresse sur 8 ou 16 bits selon le processeur cible).

IV.7.2. Motorola HC12 :

La deuxième étape de la compilation adaptée au 68HC12 correspond au **Motorola HC12 Back End**.

IV.7.2.1. Mémoire non-paginée :

Le compilateur pour le MC68HC12 supporte deux modèles de mémoire différents. Le modèle par défaut est le modèle de mémoire non-paginée SMALL qui correspond aux 64 KB de mémoire accessibles directement avec les 16 lignes d'adresses du CPU. Si, par contre, on étends la mémoire au delà des 64 KB, le compilateur utilise le modèle BANKED de la mémoire paginée.

IV.7.2.2. Mémoire paginée :

Fonctions near et far :

Le 68HC12 a la possibilité d'adresser de la mémoire étendue pour augmenter la taille de la mémoire adressable. Il existe plusieurs pages de mémoire étendue. La page active est déterminée par des registres dédiés en mémoire (registres de pages). Une partie de la mémoire n'est pas paginée et est accessible à partir de toutes les pages étendues.

Une fonction située dans la mémoire étendue doit être appelée différemment d'une fonction située dans la mémoire non étendue. En particulier, un changement de page doit être effectué :

- le numéro de page courante doit être sauvé ;
- le numéro de page de la fonction appelée doit être chargé dans le registre de page ;
- la fonction doit être appelée.

Pour s'affranchir de tout ce travail, les fonctions sont séparées en deux classes : **far** et **near**. Une fonction **far** est toujours appelée avec un *CALL* alors qu'une fonction **near** est simplement appelée avec un *JSR/BSR*. De plus, si une fonction **near** est appelée, elle doit être située soit dans l'espace mémoire non étendu, soit dans la même page que la fonction appelante. Avec le modèle de mémoire étendu (BANKED), toutes les fonctions sont par défaut des **far**. Pour déclarer explicitement une fonction en *near* ou *far*, il suffit de le préciser à la déclaration de la fonction :

```
static int far fonct(int *p) ;
```

```
static int near fonct(int *p) ;
```

Mémoire non étendue :

Certaines parties d'une application doivent être dans la zone de mémoire non paginée, en particulier :

- * le code *prestart* (segment _PRESTART) ;
- * le code de démarrage (*startup* dans le segment NON_BANKED) et les variables de démarrage (*startup descriptor* dans le segment STARTUP) ;
- * toutes les routines "runtime" (segment NON_BANKED) ;
- * toutes les fonctions d'interruption (les vecteurs d'interruption n'ont que 16bits...) ;

Normalement, des instructions sont nécessaires pour activer la mémoire étendue. Ces initialisations pourront être ajoutées à la fonction *startup*.

IV.7.2.3. Types des variables :

Types scalaires :

Tous les types de bases peuvent être redéfinis avec l'option de compilation -T (Flexible Type Management). Tous les types scalaires sans précision, sont par défaut signés : "int" est équivalent à "signed int".

Le format des différents types scalaires est donné dans la table suivante :

Type	Format par défaut	Valeurs		Formats disponibles avec l'option -T
		Min.	Max.	
char (signed)	8bit	-128	127	8bit, 16bit, 32bit
signed char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit
signed short	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned short	16bit	0	65535	8bit, 16bit, 32bit
enum (signed)	16bit	-32768	32767	8bit, 16bit, 32bit
signed int	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned int	16bit	0	65535	8bit, 16bit, 32bit
signed long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long	32bit	0	4294967295	8bit, 16bit, 32bit
signed long long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long long	32bit	0	4294967295	8bit, 16bit, 32bit

Les variables réelles :

Deux formats aux normes IEEE sont supporté par le compilateur : IEEE32 et IEEE64. Par défaut, le type pour tous les flottants (*float*, *double*, *long double* et *long long double*) est le format IEEE32 mais on peut changer le format de chaque type à l'aide de l'option -T. Les variables réelles sont donc comprises entre 1.17549435E-38 et 3.402823466E+38.

Les pointeurs et les pointeurs de fonction :

Le format des pointeurs dépend du modèle de mémoire sélectionné :

Type	Modèle de mémoire	
	SMALL	BANKED
Pointeur	2 octets	2 octets
Pointeur de fonction	2 octets	3 octets
pointeur <i>far</i>	3 octets	3 octets

Champs de bits :

La longueur maximale d'un champs de bits est de 32 bits. L'unité d'allocation est l'octet, le compilateur utilise le mot (16bits) uniquement pour les champs de bits de longueur supérieur à 8 bits ou si des octets laisseraient plus de deux bits inutilisés. L'allocation se fait du bit de poids le plus faible au bit de poids le plus fort dans l'ordre de la déclaration.

IV.7.2.4. Appels des fonctions :

Passage des arguments :

Pour les fonctions avec un nombre fixe de paramètres, la convention d'appel du Pascal est utilisée : les paramètres sont stockés dans la pile de gauche à droite. Après l'appel, ils sont retirés de la pile.

La convention d'appel du C n'est utilisée que pour les fonctions avec un nombre de paramètres variable. Dans ce cas, ils sont stockés dans la pile de droite à gauche.

Si le dernier paramètre d'une fonction avec un nombre fixe d'argument a un type simple, il n'est pas stocké dans la pile mais dans un registre. Cela donne un code plus petit. Le registre utilisé dépend du format de ce dernier argument :

Format du dernier paramètre	Exemple de type	Registre utilisé
1 octet	char	B
2 octets	int, tableaux	D
3 octets	pointeur de données <i>far</i>	X(L), B(H)
4 octets	long	D(L), X(H)

Les paramètres ayant un format supérieur à 4 octets sont passés dans la pile.

Valeurs retournées :

Les valeurs retournées par les fonctions sont normalement passées par les registres, sauf si une fonction renvoie une valeur de format supérieur à 1 mot (2 octets). Le registre utilisé dépend du format de la valeur retournée :

Format de la valeur retournée	Exemple de type	Registre utilisé
1 octet	char	B
2 octets	int	D
3 octets	pointeur de données <i>far</i>	X(L), B(H)
4 octets	long	D(L), X(H)

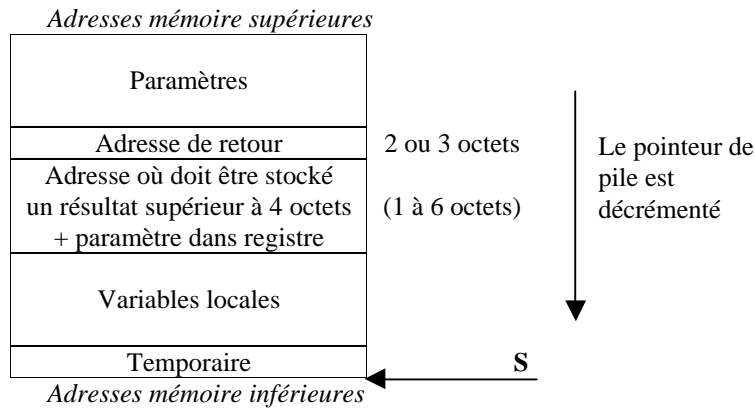
Les fonction qui renvoient un résultat de format supérieur à un mot sont appelées avec un paramètre supplémentaire : l'adresse où le résultat devra être stocké.

IV.7.2.5. Les fonctions et la pile :

Chaque fonction possède une partie de la pile contenant ses variables locales. Le compilateur utilise le pointeur de pile comme adresse de base pour accéder aux variables locales.

Si un des pragmas NO_ENTRY, NO_EXIT ou NO_FRAME est actif, le compilateur ne génère pas le code pour initialiser cette partie de la pile. Dans ce cas, la fonction ne doit pas avoir de variable locale ni de paramètres.

Le plan de cette partie de pile réservé à une fonction est décrit dans la figure suivante :



IV.7.2.6. Les fonctions d'interruptions :

Les fonctions d'interruptions sont appelées différemment des autres fonctions. Premièrement, le retour se fait par un RTI. Deuxièmement, tous les registres modifiés doivent être sauves. Les registre D, X et Y sont sauves automatiquement. Mais le compilateur doit aussi sauver les registres de pages s'ils peuvent être modifiés dans la fonction. Il faut donc distinguer les fonctions d'interruptions des autres. Pour cela on utilise un pragma :

#pragma TRAP_PROC

Les registres de pages qui doivent être sauves sont déterminés par le pragma. Sa syntaxe est la suivante :

```
#pragma TRAP_PROC [SAVE_ALL_REGS | SAVE_NO_REGS]
```

Si TRAP_PROC SAVE_ALL_REGS est utilisé, tous les registres de pages sont sauves, même s'ils ne sont pas utilisés dans la fonction d'interruption. Si TRAP_PROC SAVE_NO_REGS est utilisé, aucun registre de page n'est sauve. Si TRAP_PROC est utilisé sans paramètres, les registres de pages spécifiés avec l'option -Cp (Specify D|E|P PAGE Register) sont sauvegardés.

Le compilateur donne une possibilité (non ANSI) de donner le numéro de vecteur d'interruption dans le fichier source :

```
Void interrupt 0 ResetFunction(void) { }
```

La correspondance entre le numéro de vecteur et l'adresse du vecteur est la suivante :

Numéro de vecteur	Adresse du vecteur	Taille du vecteur
0	0xFFFFE, 0xFFFF	2
1	0xFFFFC, 0xFFFFD	2
3	0xFFFFA, 0xFFFFB	2
...
N	0xFFFF - (n*2)	2

IV.7.2.7. Segmentation mémoire :

Le compilateur permet d'attribuer un nom de segment à certaines variables ou fonctions qui pourront alors être placées dans ce segment par l'éditeur de liens. Puisqu'il y a deux type de segments de base, segments code et données, il y a aussi deux pragmas pour spécifier les segments :

```
#pragma CODE_SEG [NEAR|FAR|SHORT] <name>
```

```
#pragma DATA_SEG [DPAGE|PPAGE|EPAGE|SHORT] <name>
```

Ces deux pragmas sont valides jusqu'au prochain pragma du même type. Si aucun nom de segment n'est donné, le compilateur prend deux segments par défaut DEFAULT_ROM (le segment code par défaut) et DEFAULT_RAM (le segment de variables par défaut). On peut explicitement rendre ces segments actifs en utilisant leur nom : DEFAULT

Exemple : #pragma CODE_SEG DEFAULT

Le mot clé SHORT permet de préciser au compilateur que le segment sera alloué dans la page 0 (adresses de 0x0000 à 0x00FF) : #pragma CODE_SEG SHORT <name>

Utiliser la page 0 permet de produire un code plus dense car l'adressage direct peut être utilisé plutôt que l'adressage étendu.

Les mots FAR et NEAR sont utilisés pour spécifier la convention d'appel d'une fonction. L'appel à des fonctions FAR initialise le registre PPAGE. Les fonctions NEAR doivent être dans la même page. Dans le modèle de mémoire étendu (BANKED), toutes les fonctions sont FAR par défaut. Dans le modèle de mémoire non étendu (SMALL), les fonctions sont NEAR par défaut. Les mots clés DPAGE, EPAGE et PPAGE sont utilisés pour préciser le registre de page pour les variables paginées.

IV.7.2.8. Conseils de programmation :

Le 68HC12 est un processeur 8/16 bits qui n'a pas été conçu pour être programmé avec un langage de haut niveau. Aussi, il faudra penser à certains points pour produire un code raisonnablement efficace :

- allouer les variables statiques souvent utilisées dans la page 0 en utilisant des segments SHORT ;
- utiliser, dans la mesure du possible, des variables de type *char* ; Toutefois, il faut considérer que les expressions contenant des *char* et des *int* sont considérées comme ne contenant que des *int* car les variables *char* doivent être converties en *int*. De même pour certaines expressions ne contenant que des *char* (ex : `char a,b,c,d ; a = (b + c)/d ;`) car ces expressions doivent être évaluées avec un format de 16 bits pour suivre la norme ANSI.
- L'utilisation de type non signé est meilleur que le type signé, dans le cas des conversion *char* vers *int*, et de *int* vers *long* ;
- N'utiliser les types *long*, *float* ou *double*, que si cela est absolument nécessaire. Ils produisent beaucoup de code !
- Éviter d'utiliser des parties de pile réservées aux fonctions plus grandes que 256 octets ;
- Éviter les structures de plus de 256 octets si les champs sont accédés par un pointeur.

IV.7.3. Insertion de code assembleur :

Le compilateur HIWARE permet d'écrire directement du code assembleur dans le fichier source en C. Cela permet de ne pas avoir un fichier source assembleur séparé qu'il faudrait assembler et lier avec les autres fichiers sources. Pour la syntaxe de l'assembleur, on se référera à l'assembleur du 68HC12 (III.6 Syntaxe :).

IV.7.3.1. Syntaxe :

Le code assembleur peut apparaître partout où des instructions en langage C peuvent apparaître. Pour écrire du code assembleur, il faut utiliser le mot clé "asm". On peut l'utiliser de deux façons différentes :

```
"asm" <instruction assembleur> " ;" ["/" commentaires "*" /]
```

```
"asm" <instruction assembleur> " ;" ["/" commentaires]
```

```
exemple :   asm CLI      ;      /* autorisation des interruptions */
```

ou

```
"asm" "{ "
    { <instruction assembleur> [ " ;" commentaires "\n" ]
    } "
exemple :
```

```
asm {
    NOP
    CLI      ; autorisation des interruptions
}
```

Si on utilise la première forme, plusieurs instructions peuvent être mise sur une même ligne en les délimitant par "asm" et le point virgule " ;". Les commentaires sont alors délimités comme en langage C.

Si on utilise la deuxième forme, plusieurs instructions assembleur peuvent apparaître dans le bloc `asm { ... }`, mais il ne peut y avoir qu'une seule instruction assembleur par ligne. Dans ce cas, on peut utiliser les commentaires assembleurs commençant par " ;".

IV.7.3.2. Particularités :

Mots réservés :

L'assembleur possède des mots réservés qui ne doivent pas être confondues avec des noms de variables définies par l'utilisateur. Ces mots sont :

- tous les mnémoniques (LDAA, STX ...) ;
- tous les noms de registre (A, B, D, X, Y, CCR, SP) ;
- le mot PAGE ;

Pour ces mots réservés, l'assembleur ne fait pas la différence entre majuscules et minuscules. Ainsi, LDAB est identique à lDaB ou ldab... Pour tous les autres identificateurs (étiquettes, nom de variable etc...), l'assembleur fait la différence.

Variables :

L'assembleur permet d'accéder aux variables définies dans le programme en langage C simplement en utilisant leur nom. Les variables globales sont traduit en adressage direct ou étendu selon le segment dans lequel elles sont stockées.

Expressions constantes :

Les expressions constantes peuvent être utilisées partout où une valeur immédiate est attendue. Ces expressions peuvent contenir les opérateurs "+", "-", "*", "/" et "/". La syntaxe est la même que les expressions en C.

Adresse d'une variable :

On peut utiliser l'adresse d'une variable globale ou l'offset d'une variable locale :

Adresse = "@" variable

Exemple: LDD @glo ; charge X avec l'adresse de la variable glo
 LDY @loc ; charge Y avec l'offset (dans la partie de pile
 réservé à la fonction) de la variable locale ou du
 paramètre

On peut aussi accéder aux champs d'une structure (*struct*) ou d'une union (*union*) en utilisant la notation du langage C :

 LDD r.f ; Charge D avec le contenu du champs f de la structure r

Pour accéder au mot de poids faible d'une variable *long* ou *float*, on a la possibilité d'utiliser un offset par rapport à l'adresse de la variable :

 LDY @g :2 ; charge Y avec l'adresse de g + 2
 LDD g :2 ; charge X avec la valeur stockée à l'adresse de g + 2
 LDD r.f :2 ; Charge D avec le mot de poids faible du champs f

On peut utiliser cette possibilité pour accéder aux éléments d'un tableau :

```
int tab[20] ;
asm {LDD tab :24   ;charge tab[12] dans D }
```

IV.8. Librairies**IV.8.1. ANSI-C**

V. L'ÉDITEUR DE LIENS :

L'édition de liens est l'opération qui consiste à assigner de la mémoire à tous objets d'une application (fonctions, variables, constantes etc...) et à réunir tous ces objets dans un format approprié pour être chargé dans le système hôte. C'est lors de cette opération que tous les liens entre les différents modules d'un programmes sont réalisés, que toutes les adresses restant à définir après l'assemblage (ou la compilation) sont initialisées. A la fin de l'édition de liens, on obtient un fichier exécutable de l'application qui pourra être utilisé pour la simulation ou la mise au point sur une carte d'évaluation. Par contre, pour l'implanter dans une EPROM, le fichier doit avoir un format particulier que l'on obtient avec le *Burner*.

L'éditeur de liens d'HIWARE est un éditeur de liens "intelligent" ou "smart linker": il ne s'occupe que des objets réellement utilisés dans l'application. Les fonctions et variables non utilisées n'occuperont pas d'espace mémoire dans le système hôte. De plus, un certain nombre d'optimisations sont réalisés dans l'optique d'un système ne possédant pas de beaucoup de mémoire : les variables globales sont mémorisées dans une forme compacte et pour les chaînes de caractères égales, l'espace mémoire n'est réservé qu'une seule fois.

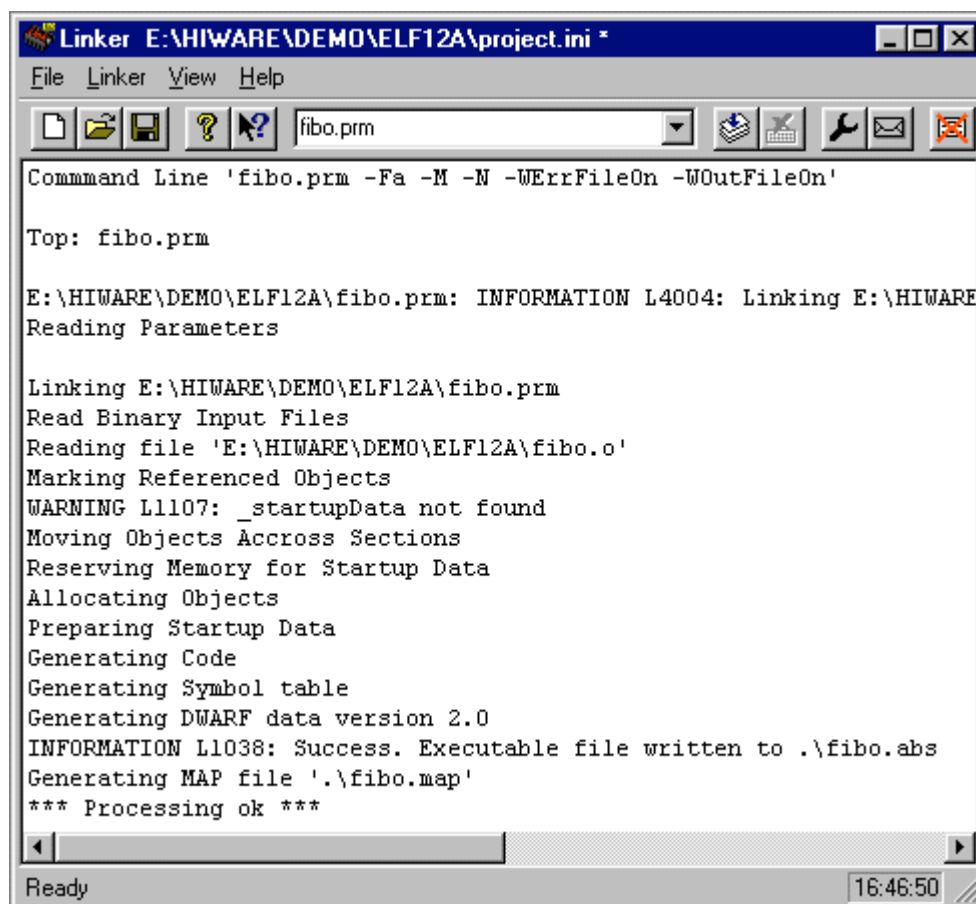
Les caractéristiques les plus importantes de l'éditeur de liens sont :

- Un contrôle complet du placement des objets dans la mémoire : la gestion de la segmentation de la mémoire est complète ;
- Réalisations des liens vers les objets dont l'édition de liens a déjà été réalisée précédemment. (Utilisation de bibliothèques de fonctions en ROM) ;
- Fonction d'initialisation (Startup) personnalisée : Le code de cette fonction est accessible dans un fichier séparé en assembleur et peut être facilement adapté ;
- Édition de liens de langage différents : il est possible de mixer des fichiers objets assembleur et C dans la même application ;
- L'initialisation des vecteurs d'interruption peut se faire directement à l'édition de liens.

Pour lancer l'éditeur de liens, on peut soit cliquer sur l'icône du linker dans le gestionnaire de projet, soit cliquer sur l'icône "rebuild" de la barre d'outils de WinEdit.

V.1. L'interface graphique de l'éditeur de liens :

Lorsque l'éditeur de liens est lancé sans avoir spécifier un nom de fichier dans la ligne de commande, la fenêtre principale apparaît.



L'interface graphique se présente comme toutes les applications Windows, avec une barre de titre, de menu, d'outils et d'état. La zone de texte affiche les différentes informations concernant le processus en cours. Ces informations sont :

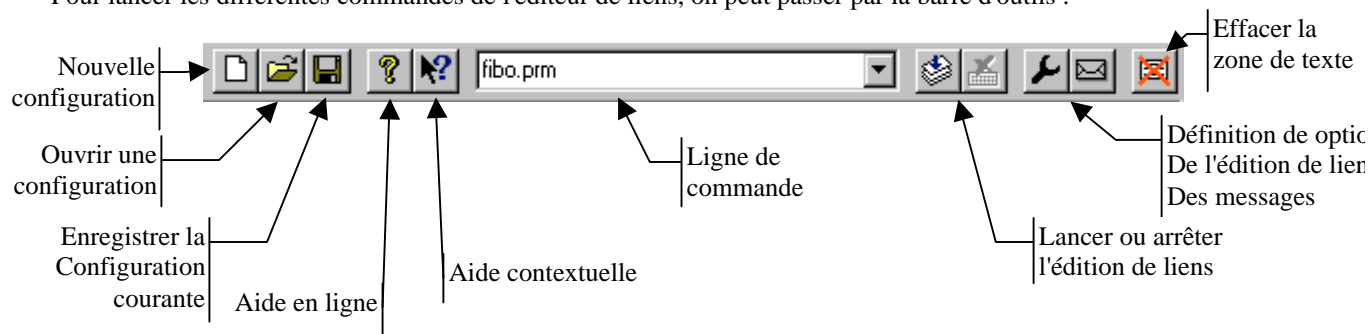
- le nom du fichier actuellement traité par l'éditeur de liens ;
- le nom complet des fichiers concernés ;
- la liste des erreurs, warning et informations générés lors de l'édition de liens ;
- le nom et le type des fichiers créés.

Des menus popup contextuels sont accessibles d'un clique droit de la souris dans la zone de texte. Ceux-ci permettent, en fonction du texte pointé par la souris :

- d'obtenir l'aide principale
- d'obtenir une aide contextuelle sur l'élément pointé par la souris. On peut, par exemple, obtenir de l'aide sur une erreur survenue lors de l'édition de liens ;
- d'ouvrir, soit le fichier pointé ou sélectionné par la souris, soit le fichier .PRM contenant l'erreur sélectionnée. Le fichier s'ouvre alors dans l'éditeur de texte WinEdit ;
- de copier dans le presse papier de Windows, le texte sélectionné.

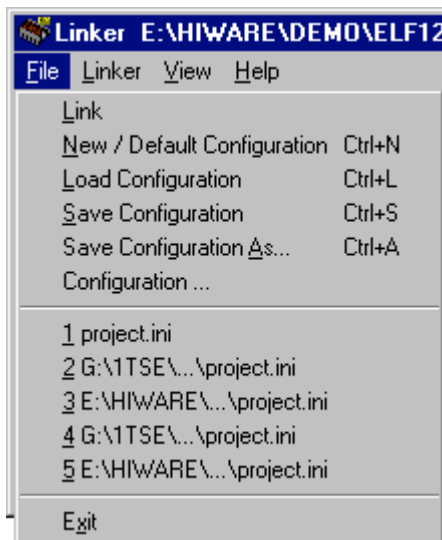
V.2. La barre d'outils :

Pour lancer les différentes commandes de l'éditeur de liens, on peut passer par la barre d'outils :



Au démarrage de l'éditeur de liens, la ligne de commande contient le nom du dernier fichier concerné par la dernière édition de liens lancée par cette ligne de commande.

V.3. Le menu fichier / lancement de l'édition de liens:



Pour lancer l'édition de liens, deux solutions sont possibles :

- Entrer le nom du fichier dans la ligne de commande de la barre d'outils puis sélectionner l'icône "link". Le fichier doit alors être dans le répertoire du projet ;
- Sélectionner la commande "link" du menu fichier. Une fenêtre apparaît alors permettant de sélectionner le fichier dans le répertoire du projet. L'édition de liens est lancée une fois le fichier sélectionné.

A partir de ce menu il est également possible de :

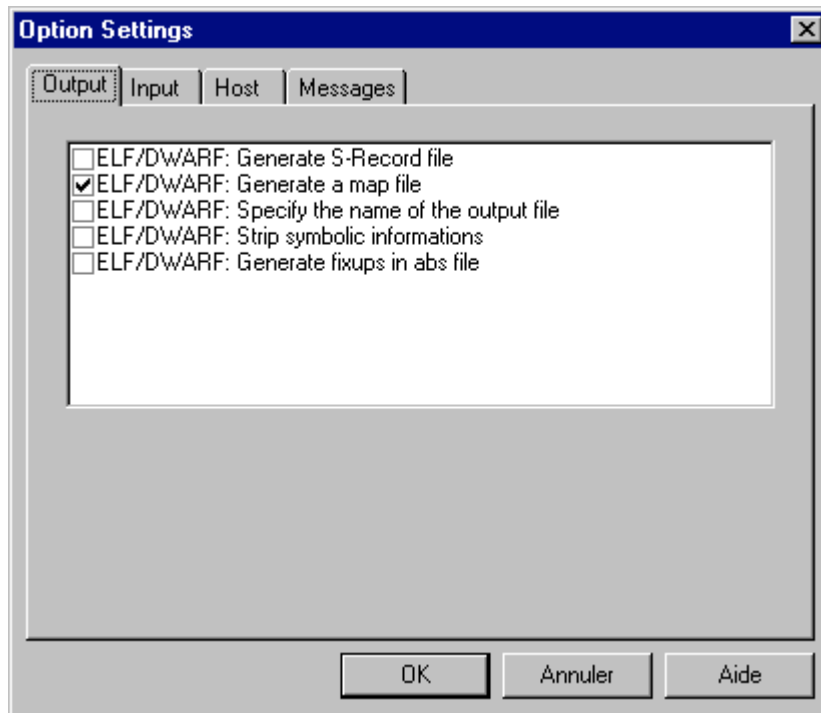
- Créer, ouvrir et sauver la configuration de l'éditeur de liens pour le projet en cours ;
- Ouvrir une configuration au choix parmi les dernières ouvertes ;
- Définir la configuration :

Définition de la configuration :

Pour définir la configuration, on se reportera au chapitre correspondant dans la partie traitant de l'assembleur.

V.4. Les options de l'édition de liens :

Pour définir les options de l'édition de liens, il faut sélectionner la commande "Options" du menu "linker" ou cliquer sur l'icône correspondant dans la barre d'outils. La fenêtre suivante apparaît :



Les options d'édition de liens sont réparties en quatre groupes. On définit une option en cochant la ligne correspondante et en complétant éventuellement d'autres informations correspondant à l'option choisie. Chaque option cochée correspond en fait à un paramètre passé à la ligne de commande lançant l'édition de liens. Par exemple :

Command line 'fiboprm -Fa -M -N -WerrFileOn -WoutFileOn'

V.4.1. Options "Output" :

Ce sont les options de contrôle des fichiers de sorties :

- "ELF/DWARF : Generate S-Record file" : génère un fichier au format S-record en plus du fichier absolu .abs.
- " ELF/DWARF : Generate a map file" : Génère un fichier listing contenant les informations de l'édition de liens et nommé fichier map. On y trouve, entre autres, les informations relatives au type de microprocesseur, aux fichiers concernés, à l'allocation des segments et des objets en mémoire.
- " ELF/DWARF : Specify the name of the output file" : Permet de définir le nom du fichier absolu lorsque l'on veut qu'il soit différent du fichier source.
- " ELF/DWARF : Strip symbolic informations" : Permet de ne pas sauver les symboles dans le fichier absolu. Le debugage symbolique ne sera alors pas possible.
- " ELF/DWARF : Generate fixups abs file" :

V.4.2. Options "Input" :

Ce sont les options relatives au fichiers d'entrées, objets ou de paramétrage :

- " ELF/DWARF : Specify the name of the startup function" : Permet de spécifier le nom de la fonction d'initialisation.
- "Object file format" : Permet de spécifier le format des fichier objets à lier.
- " ELF/DWARF : Add a path to the search path " :

V.4.3. Options "Host" :

"Set environment variable" : Permet de définir une variable d'environnement.

V.4.4. Options "Messages" :

C'est ici que l'on définira les options d'affichage des erreurs et des warning et des messages d'information.

V.5. Format des fichiers :

V.5.1. Fichiers d'entrées :

Fichier de paramétrage :

L'édition de liens se fait par l'intermédiaire d'un fichier de paramétrage dans lequel on définit les fichiers à lier, le plan mémoire et les vecteurs d'interruptions. Ce fichier peut avoir n'importe quelle extension mais il est recommandé d'utiliser l'extension **.prm**. Ce fichier ne doit être constitué que par des caractères ASCII.

Fichiers objets :

L'éditeur de liens trouve le nom des fichiers objets à lier dans le fichier de paramétrage. Ces fichiers doivent être des fichiers objets au format compatible ELF/DWARF (**.o**), des fichiers absolus (**.abs**) ou des fichiers de bibliothèques (**.lib**).

V.5.2. Fichiers de sorties :

Fichiers absolus :

Après une édition de liens réussie, l'éditeur génère un fichier absolu contenant le code du programme avec des informations de débogage. Ces fichiers absolus ont l'extension **.abs**.

Fichiers MAP :

Après une édition de liens réussie, l'éditeur génère un fichier listing, ou fichier map, contenant les informations relatives au déroulement de l'édition de liens. Ces fichiers ont l'extension **.map**.

Fichiers d'erreurs :

Quand une erreur est survenue lors de l'édition de liens, l'éditeur peut générer, si l'on a coché l'option correspondante, un fichier listing rapportant les erreurs rencontrées. Ces fichiers peuvent être utiles lorsqu'on travaille mode non interactif (la fenêtre de l'éditeur n'est pas ouverte...).

V.6. Paramétrage de l'édition de liens :

Le paramétrage de l'édition de liens se fait par l'intermédiaire d'un fichier de paramétrage. C'est un fichier texte comportant plusieurs parties. Comme les autres application HIWARE, l'éditeur de liens est capable de gérer deux formats de fichiers : Le format portable ELF/DWARF et le format spécifique d'HIWARE. Le premier permet de générer des fichiers compatibles avec des applications autres que celles d'HIWARE. Ce format a le désavantage de créer des fichiers plus importants que le format HIWARE qui produit du code plus compact. Nous verrons le format ELF/DWARF en premier, et nous ajouterons les modifications à apporter pour le format HIWARE.

V.6.1. Nom du fichier absolu :

On indique le nom du fichier absolu généré par l'édition de liens à l'aide de la commande **LINK**. Le fichier absolu doit avoir l'extension **.abs**.

Exemple : **LINK** test.abs

V.6.2. Fichiers à lier :

Le premier block du fichier de paramétrage permet de préciser quels sont les fichiers à lier. Il existe deux syntaxe pour les préciser : NAMES et ENTRIES. En général, NAMES est suffisant.

NAMES : Liste tous les fichiers constituant l'application et qui sont à lier.

ENTRIES : Liste tous les fichiers à lier avec l'application même s'ils ne font pas partie de l'application. En utilisant cette syntaxe, on désactive l'édition de liens "intelligente" ou "smart linking". Cette commande n'est pas disponible dans le format HIWARE.

Exemple :

NAMES myfile1.o myfile2.o mylib.lib **END**

ENTRIES myfile1.o :* myfile2.o :* **END**

*Tous les objets référencés dans les fichiers
myfile1.o et myfile2.o seront liés avec l'application.*

V.6.3. Segmentation de la mémoire :

La segmentation de la mémoire consiste à définir un nom à différents espaces mémoire continus de la mémoire du système. Cette segmentation est optionnelle mais elle permet d'augmenter la lisibilité du fichier de paramétrage. Les segments se définissent à l'aide de la syntaxe **SEGMENTS** pour le format ELF et par **SECTIONS** pour HIWARE. On peut définir deux types de segments :

V.6.3.1. Segments physiques :

Les segments physiques sont reliés directement à la mémoire physique disponible dans le système. On peut, par exemple, créer un segment pour l'espace mémoire ROM et un autre pour l'espace mémoire RAM.

Exemple :

```
SEGMENTS
    RAM_AREA = READ_ONLY    0x08000 TO 0x0FFFF ;
    ROM_AREA = READ_WRITE  0x00000 TO 0x07FFF ;
END
```

V.6.3.2. Segments virtuels :

Un segment physique peut être divisé en plusieurs segments virtuels, permettant de mieux structurer l'allocation des objets en mémoire. On peut, par exemple, créer des segments pour les seuls espaces utilisables par l'utilisateur d'une carte d'évaluation M68HC12A4EVB :

Exemple :

```
SEGMENTS
    USER_ON_CHIP_RAM      =    READ_WRITE    0x0800 TO 0x09FF ;
    USER_ON_CHIP_EEPROM  =    READ_ONLY      0x1000 TO 0x1FFF ;
    USER_EXTERN_RAM       =    READ_WRITE     0x4000 TO 0x7FFF ;
    USER_EXTERN_EPROM1    =    READ_ONLY      0x8000 TO 0x9FFF ;
    USER_EXTERN_EPROM2    =    READ_ONLY      0xFF00 TO 0xFF7F ;
END
```

V.6.3.3. Attributs des segments :

Différents attributs sont possibles pour les segments :

READ_ONLY : Seule la lecture est possible dans le segment concerné. Les objets situés dans un segment de ce type sont initialisés au chargement de l'application.

READ_WRITE : La lecture et l'écriture sont possibles dans les segments concernés. Les objets situés dans un segment de ce type sont initialisés au démarrage de l'application.

NO_INIT : La lecture et l'écriture sont possibles dans les segments concernés. Les objets situés dans un segment de ce type ne sont pas changés au démarrage de l'application. On pourra utiliser cet attribut pour définir des segments référant à une mémoire RAM avec sauvegarde par batterie. Avec cet attribut, les segments ne doivent pas contenir de variables initialisées.

PAGED : La lecture et l'écriture sont possibles dans les segments concernés. Les objets situés dans un segment de ce type ne sont pas changés au démarrage de l'application. Des objets situés dans deux segments **PAGED** (paginés) peuvent se recouvrir. Cet attribut est utilisé pour définir des segments situés dans des espaces mémoires paginés et où le passage d'une page à une autre est fait par le programmeur. Avec cet attribut, les segments ne doivent pas contenir de variables initialisées.

V.6.3.4. Alignement des segments :

La règle d'alignement par défaut dépend du microprocesseur et du modèle de mémoire utilisé. Le 68HC12 n'a pas besoin d'alignement des données et du programme. Mais on peut choisir de définir ses propres règles d'alignement pour un segment. On se référera à l'aide en ligne. (Option disponible uniquement avec le format ELF).

V.6.3.5. Initialisation des segments :

La valeur d'initialisation de la mémoire est par défaut le caractère null. Mais on peut choisir d'initialiser tout un segment avec une valeur personnalisée. Pour cela on ajoute à la ligne de définition du segment la syntaxe **FILL**. (Option disponible uniquement avec le format ELF).

Exemple :

```
SEGMENTS
    RAM_AREA1 =    READ_WRITE    0x00000 TO 0x000FF    FILL 0xAA ;
    RAM_AREA2 =    READ_WRITE    0x00100 TO 0x07FFF    FILL 0x22 ;
    ROM_AREA  =    READ_ONLY     0x08000 TO 0x0FFFF ;
END
```

V.6.4. Placement des segments :

Le block de placement des segments permet de placer physiquement les différentes sections définies dans les fichiers sources, dans les segments de mémoire définis dans le fichier de paramétrage de l'édition de liens. Les sections à placer peuvent être les sections définies par le programmeur dans le fichier source mais il existe aussi des sections pré-définies. Le placement se fait à l'aide du block **PLACEMENT**.

On peut placer plusieurs sections dans un seul segment. Elles seront alors placées dans l'ordre de leur énumération.

Exemple :

```

LINK test.abs
NAMES      test.o startup.o END

SEGMENTS
    RAM_AREA =      READ_WRITE      0x00100 TO 0x002FF ;
    STK_AREA  =      READ_WRITE      0x00300 TO 0x003FF ;
    ROM_AREA  =      READ_ONLY 0x08000 TO 0x0FFFF ;
END
PLACEMENT
    DEFAULT_RAM, dataSec1, dataSec2 INTO RAM_AREA ;
    DEFAULT_ROM, mycode              INTO ROM_AREA ;
    SSTACK                          INTO STK_AREA ;
END
```

On peut placer des sections dans plusieurs segments. Elles seront placées dans l'ordre de leur énumération, dans le premier segment jusqu'à qu'il soit plein, puis dans le second etc... jusqu'à que toutes les sections soient placées.

Exemple :

```

LINK test.abs
NAMES      test.o startup.o END

SEGMENTS
    RAM_AREA      =      READ_WRITE      0x00100 TO 0x002FF ;
    STK_AREA       =      READ_WRITE      0x00300 TO 0x003FF ;
    ROM_AREA1 =      READ_ONLY 0x08000 TO 0x0FFFF ;
    ROM_AREA2 =      READ_ONLY 0x10000 TO 0x100FF ;
    ROM_AREA3 =      READ_ONLY 0x40000 TO 0x4FFFF ;
END
PLACEMENT
    DEFAULT_RAM, dataSec      INTO RAM_AREA ;
    DEFAULT_ROM, mycode       INTO ROM_AREA1,ROM_AREA2,ROM_AREA3;
    SSTACK                   INTO STK_AREA ;
END
```

V.6.5. Sections pré-définies :

Si dans les programmes en langage assembleur la définition des segments est nécessaire pour faire la différence entre les variables, les constantes et le code du programme, en langage C, ces sections peuvent être complètement transparente. Aussi, dans ces programmes, il n'est pas nécessaire de préciser chaque sections. Il existe donc des sections pré-définies pour que l'éditeur de liens puisse placer les différents objets en mémoire. Ces sections ont des noms différents dans le format ELF et le format HIWARE. Ces noms correspondent à des sections équivalentes mais il est toutefois nécessaire de préciser les deux syntaxes :

.rodata^(ELF) / STRINGS^(HIWARE) :

Toutes les chaînes de caractères sont placées dans cette section. Si cette section est définie comme étant **READ_WRITE**, les chaînes de caractères sont chargées de la ROM vers la RAM à l'initialisation du programme.

.rodata^(ELF) / ROM_VAR^(HIWARE) :

Toutes les constantes non définies dans des sections utilisateur, sont placées dans cette section. Généralement cette section est définie comme étant **READ_ONLY**. Si elle n'est pas mentionnée dans le block **PLACEMENT**, ces constantes sont placées à la suite de la section **.text**.

.copy^(ELF) / COPY^(HIWARE) :

Toutes les variables possédant une valeur d'initialisation (exemple : int a=2 ; en C), sont placées dans cette section. A l'initialisation du programme, ces variables sont chargées de la ROM à la RAM. Donc, si la section **.rodata1** est définie **READ_WRITE**, elle est en fait placée dans la section **.copy**.

.stack^(ELF) / SSTACK^(HIWARE) :

La pile possède sa propre section **.stack**. Elle doit toujours être définie comme **READ_WRITE**.

.data^(ELF) / DEFAULT_RAM^(HIWARE) :

C'est la section par défaut pour les objets qui doivent être placés en RAM. Toutes les variables appartenant à aucune section ou à une section non mentionnée dans le block PLACEMENT, sont par défaut placées dans cette section. Si la section **.stack** n'est pas associée explicitement à un segment, elle automatiquement placée dans le segment de **.data** à la suite de la section **.data**.

.text^(ELF) / DEFAULT_ROM^(HIWARE) :

C'est la section par défaut pour toutes les fonctions. Si une fonction n'appartient pas à une section ou si elle appartient à une section non mentionnée dans block PLACEMENT, elle sera automatiquement placée dans cette section. Si les sections **.rodata**, **.rodata1**, **.startData** ou **.init** ne sont pas associées explicitement à un segment, elles sont incluses dans le segment de la section **.text** dans l'ordre :

.init | **.startData** | **.text** | **.rodata** | **.rodata1**

.startData^(ELF) / STARTUP^(HIWARE) :

C'est dans cette section que sont placées les données initialisées par l'éditeur de liens et utilisées par la fonction d'initialisation. Cette section doit être READ_ONLY.

.init^(ELF) / _PRESTART^(HIWARE) :

Le point d'entrée du programme est mémorisé dans cette section qui doit être READ_ONLY.

Les sections **.text^(ELF) / DEFAULT_ROM^(HIWARE)** et **.data^(ELF) / DEFAULT_RAM^(HIWARE)** doivent toujours être associées à un segment dans le block PLACEMENT.

V.6.6. Initialisation de la table des vecteurs :

La table des vecteurs d'interruption peut être initialisée à l'aide de la commande **VECTOR**.

On peut initialiser un vecteur d'interruption par son numéro, seulement si l'adresse de base de la table des vecteurs est 0x0000. On utilise alors la syntaxe "**VECTOR** numéro_de_vecteur".

Sinon, on peut toujours initialiser un vecteur à l'aide d'une des trois syntaxe suivante :

VECTOR ADDRESS 0xFFFFE 0x1000 : La valeur 0x1000 sera mémorisée à l'adresse 0xFFFFE ;

VECTOR ADDRESS 0xFFFFE nomF : L'adresse de la fonction **nomF** sera mémorisée à l'adresse 0xFFFFE ;

VECTOR ADDRESS 0xFFFFE nomF +2 : L'adresse de la fonction **nomF** plus 2 sera mémorisée à l'adresse 0xFFFFE ;

V.6.7. Initialisation de la pile :

L'initialisation de la pile ne se fait au moment de l'édition de liens que pour les programmes en langage C. Dans les programmes en assembleur, la pile doit être initialisée dès le début du programme dans le fichier source. Pour initialiser la pile, on dispose de deux commandes qui ne peuvent pas être utilisées en même temps :

STACKSIZE : Cette commande est optionnelle. Elle définit la taille de pile. On utilise cette commande de préférence lorsqu'on ne se soucie pas de l'endroit où se trouve la pile, mais uniquement de sa taille. Lorsque cette commande est utilisée seule, la pile est placée après la section **.data^(ELF) / DEFAULT_RAM^(HIWARE)**. Si on utilise cette commande associée au placement de la section **.stack^(ELF) / SSTACK^(HIWARE)**, la pile commencera à l'adresse de base du segment où aura été placé la section, incrémenté de la taille de la pile.

Exemple :**SECTIONS**

```
MY_STK   = NO_INIT      0xB00      TO  0xBFF ;
MY_RAM   = READ_WRITE  0xA00      TO  0xAFF ;
MY_ROM   = READ_ONLY   0x800      TO  0x9FF ;
```

END**PLACEMENT**

```
DEFAULT_ROM INTO MY_ROM ;
DEFAULT_RAM INTO MY_RAM ;
SSTACK      INTO MY_STK ;
```

END

STACKSIZE 0x60

Dans cet exemple, la section SSTACK est placée de l'adresse 0xB5F jusqu'à 0xB00. La pile est initialisée à l'adresse 0xB5E.

STACKSTOP : Cette commande est optionnelle. Elle définit l'adresse d'initialisation de la pile. Lorsque cette commande est utilisée seule, la taille de la pile est définie par défaut, et est assez grande pour contenir le PC. Si on associe cette commande avec le placement de la section **.stack^(ELF)** / **SSTACK^(HIWARE)**, la pile commence à l'adresse spécifiée et est définie jusqu'à l'adresse de base du segment où a été placée la section.

Exemple :

```
SECTIONS
    MY_STK   = NO_INIT      0xB00      TO 0xBFF ;
    MY_RAM   = READ_WRITE  0xA00      TO 0xAFF ;
    MY_ROM   = READ_ONLY   0x800      TO 0x9FF ;

END
PLACEMENT
    DEFAULT_ROM INTO MY_ROM ;
    DEFAULT_RAM INTO MY_RAM ;
    SSTACK      INTO MY_STK ;

END
STACKSTOP      0xB7E
```

Dans cet exemple la pile est définie depuis l'adresse 0xB7E jusqu'à l'adresse 0xB00.

V.6.8. Point d'entrée du programme :

L'éditeur de liens, quand on ne lui précise pas le point d'entrée du programme, cherche une fonction nommée '**_Startup**' et la définit comme étant le point d'entrée dans le programme. Pour définir une autre fonction, il est nécessaire pour les programmes en assembleur, mais optionnel pour les autres, de la définir à l'aide de la commande **INIT**.

Lorsque la fonction d'initialisation est lancée et a fini son travail, elle cherche la fonction principale du programme et la lance. Cette fonction doit par défaut s'appeler '**main**' (fonction main en C) mais si l'on veut que la première fonction lancée après la fonction **_Startup** soit différente, on peut indiquer à l'éditeur de liens un nom différent à l'aide de la commande **MAIN**.

V.6.9. Édition de liens des programmes assembleurs :

Lorsque l'application n'est composée que de fichiers sources en assembleur, le fichier de paramétrage de l'édition de liens peut être simplifié :

- Aucune fonction d'initialisation n'est nécessaire ;
- La pile n'a pas à être initialisée puisqu'elle doit l'être directement dans le fichier source ;
- Aucune fonction 'main' n'est nécessaire ;
- Par contre, le point d'entrée du programme doit être précisé.

Exemple :

```
LINK test.abs
NAMES test.o test2.o END

SEGMENTS
    DIRECT_RAM   = READ_WRITE  0x00000 TO 0x000FF ;
    RAM_AREA     = READ_WRITE  0x00100 TO 0x003FF ;
    ROM_AREA1    = READ_ONLY   0x08000 TO 0x0FFFF ;
    ROM_AREA2    = READ_ONLY   0x10000 TO 0x100FF ;
    ROM_AREA3    = READ_ONLY   0x40000 TO 0x4FFFF ;

END
PLACEMENT
    DEFAULT_RAM, dataSec INTO RAM_AREA ;
    DEFAULT_ROM, mycode  INTO ROM_AREA1,ROM_AREA2,ROM_AREA3;
    Myregister           INTO DIRECT_RAM ;

END
INIT Start ; Point d'entrée du programme
VECTOR ADDRESS 0xFFFFE Start ; Initialisation du vecteur de reset
```

Dans cet exemple, le point d'entrée et le vecteur de reset sont initialisés avec l'adresse de l'étiquette **Start** qui doit donc être définie en tant que symbole externe dans le fichier source avec la directive **XDEF Start**. L'éditeur de liens générera deux Warning liés à la fonction d'initialisation **_startup** et la fonction **main**, qui pourront être ignorés.

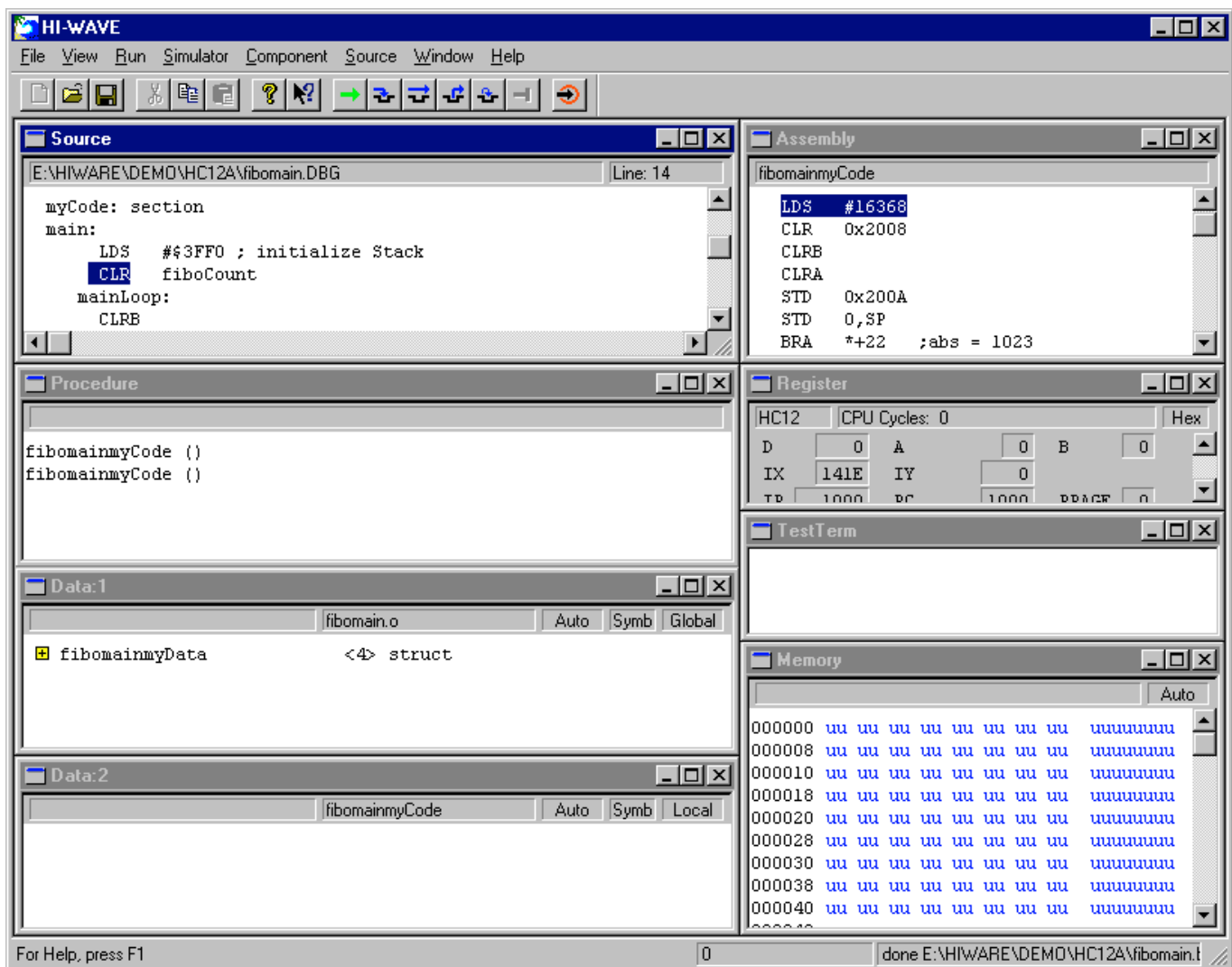
VI. ENVIRONNEMENT DE SIMULATION ET DE DEBUGAGE HIWAVE :

VI.1. Introduction :

HIWAVE est un environnement de simulation et de débogage pour microcontrôleurs 68HC12. Cet environnement est composé de divers objets présentés sous forme de fenêtres Windows et regroupés dans une seule fenêtre. Tous ces objets forment un système dont le cœur est le moteur HIWAVE. C'est lui qui gère tous ces objets qui ont tous une fonction particulière. Chaque environnement peut être adapté aux besoins de chaque programmeur. De plus, l'utilisateur peut développer lui-même ses propres objets (grâce à un logiciel spécifique avec programmation orientée objet C++) et les utiliser dans son système.

VI.2. Interface graphique :

L'interface graphique de HIWAVE est semblable à celle de toutes les applications Windows. Elle possède une fenêtre principale qui regroupe toutes les fenêtres des objets qui composent le système.



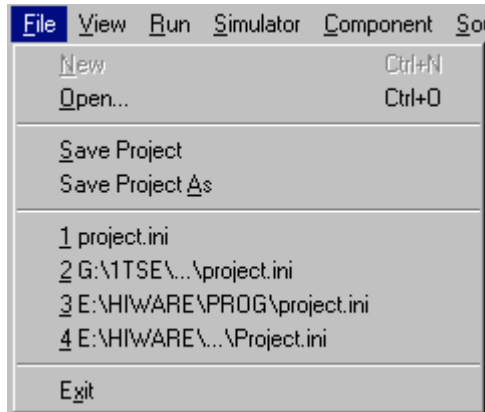
VI.2.1. Les menus :

La barre de menu se présente ainsi :

File View Run Simulator Component Source Window Help

Elle est composée de menus de base : File ; View ; Run ; Simulator ; Component ; Window et Help.

Elle contient également un menu contextuel selon l'objet dont la fenêtre est active (exemple : Source).

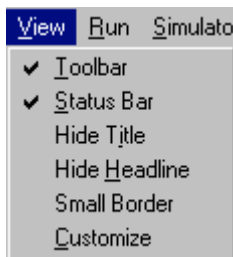
VI.2.1.1. Le menu File :

Ce menu est dédié à la gestion des projets HIWAVE.

On peut charger un fichier .HWP existant correspondant à la configuration d'un environnement de simulation précédent (objets composants le système, taille et position des fenêtres...).

On peut aussi sauvegarder le projet ou en créer un nouveau.

Pour quitter HIWAVE, on peut choisir la commande Exit.

VI.2.1.2. Le menu View :

Avec ce menu, on peut contrôler l'affichage des composants des fenêtres Windows. On peut choisir de montrer ou de cacher la barre d'outils, la barre d'état, la barre de titre ou d'information.

On peut aussi choisir des bordures plus petites pour toutes les fenêtres.

Enfin, on peut personnaliser la barre d'outils à l'aide de la commande Customize.

VI.2.1.3. Le menu Run :

Ce menu est associé aux commande de simulation ou de débogage.

- Start/Continue :** Commence ou continue l'exécution de l'application.
- Restart :** Recommence l'exécution de l'application depuis son point d'entrée.
- Halt :** Interrompt et arrête l'application en cours d'exécution. On peut alors examiner l'état de chaque variable de l'application, poser des points d'arrêt et de "regard" et étudier le code source.
- Single Step :** Si l'application est arrêtée, cette commande exécute le programme instruction par instruction (du code source en C) en entrant dans les sous-programmes (ou fonctions...). Elle correspond au mode pas à pas.
- Step Over :** Même chose que le mode pas à pas, mais exécute les sous-programmes sans entrer dedans.
- Step Out :** Si l'application est arrêtée à l'intérieur d'une fonction, cette commande exécute toutes les instructions jusqu'à la première instruction suivant l'appel de la fonction.

- Asssembly Step :** Cette commande est équivalente à l'instruction **Single Step** mais au niveau du code assembleur.
- Asssembly Step Over :** Cette commande est équivalente à l'instruction **Step Over** mais au niveau du code assembleur. On entre pas dans les sous-programmes.
- Asssembly Step Out :** Cette commande est équivalente à l'instruction **Step Out** mais au niveau du code assembleur.
- Breakpoints... :** Cette commande ouvre la fenêtre d'éditations des points d'arrêts et permet de consulter la liste des différents points d'arrêts positionnés dans l'application et de modifier leurs propriétés.

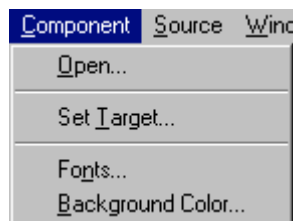
Watchpoints... :

Cette commande ouvre la fenêtre d'éditions des points de "regard" et permet de consulter la liste des différents points de "regard" positionnés dans l'application et de modifier leurs propriétés.

VI.2.1.4. Le menu Target :

Ce menu est apparent lorsqu'aucune cible n'est précisée dans le fichier d'initialisation PROJECT.INI et qu'aucune cible n'a été choisie. La cible correspond au système sur lequel doit être exécuté l'application. Quand aucune cible n'a déjà été choisie, la commande **Load** permet de la préciser, et on a le choix entre 5 cibles différentes : Ascimon, Bdi12, Motosil, Sim et Trace32. Chaque cible possède son propre mode de fonctionnement qu'il faudra trouver dans la documentation correspondante. Nous utiliserons que deux de ces cibles :

- **Ascimon** : Cible correspondant au débogage avec une carte d'évaluation du type HC12A4EVB avec le mode DBUG12. Lorsque cette cible est utilisée, le menu **target** est remplacé par le menu **Monitor**.
- **Sim** : Cible correspondant à la simulation. Lorsque cette cible est utilisée, le menu **target** est remplacé par le menu **Simulator**. C'est dans ce mode que nous allons découvrir la suite.

VI.2.1.5. Le menu component :

Ce menu correspond aux commandes liées aux différents objets que l'on peut retirer ou ajouter au système.

On peut charger un objet dans le système.

On peut aussi redéfinir la cible à l'aide de la commande **Set Target...**

Enfin, on peut définir certaines propriétés des objets comme choisir une police de caractère particulière ou définir la couleur de fond de l'objet.

VI.2.1.6. Le menu Window :

Ce menu permet de définir l'arrangement général des différentes fenêtres dans la fenêtre principale.

Dans le menu **Options**, on peut :

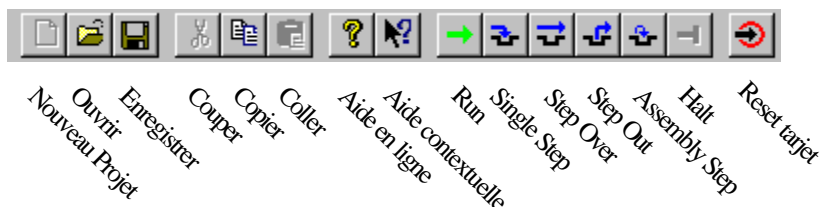
- faire en sorte que la taille des fenêtres s'ajuste automatiquement à la taille de la fenêtre principale en cochant **Autosize** ;
- Permettre l'affichage du menu associé à l'objet sélectionné dans la barre de menu principale avec **Component Menu** ;

Dans le menu **Layout**, on peut sauvegarder ou charger la configuration de ces arrangements.

On peut également rendre active la fenêtre d'un objet directement à partir de ce menu.

VI.2.2. Le barre d'outils :

La barre d'outil permet de lancer les commandes de simulation en cliquant sur des icônes. La barre d'outils par défaut se présente ainsi :



VI.2.3. Menus associés aux objets :

Chaque objet chargé dans le système, possède ses propres commandes qui sont accessibles par des menus. Deux sortes de menus cohabitent ensembles pour chaque objet. Le premier est le menu accessible par la barre de menu principale. Le second est accessible par un clique droit dans la fenêtre de l'objet. Ces menus sont contextuels et dépendent de la position de la souris au moment de l'appel au menu. Ces menus seront décrits pour chaque objet dans le chapitre qui leur est attribué.

VI.2.4. Possibilités de l'interface :

L'interface graphique de HIWAVE utilise au maximum les possibilités du glisser/déposer de la souris. Ces possibilités sont bien sûr contextuelles et dépendent de la fenêtre source et de la fenêtre destination. Chaque combinaison de glisser/déposer possibles seront abordées pour chaque objet.

VI.3. Les objets ou composants :

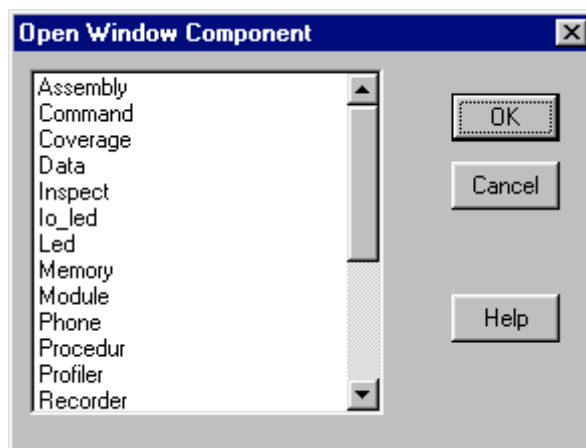
Le fonctionnement d'HIWARE repose sur des composants moteurs sur lesquels les composants de bases se greffent. D'autres objets personnalisés peuvent ensuite être intégrés dans le système. Les composants moteurs sont :

- **L'objet CPU :** C'est l'objet qui s'occupe de simuler (ou de suivre) le fonctionnement de l'unité centrale du microcontrôleur. Il met à jour les registres, exécute les instructions et les désassemble, etc... Cet objet est spécifique au microcontrôleur utilisé.
- **L'objet tarjet :** C'est l'objet qui gère les cibles.

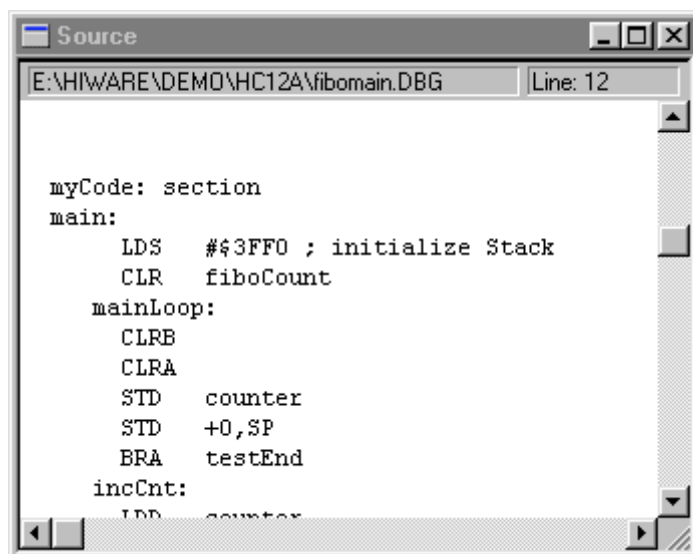
Ces deux objets sont automatiquement chargés dans le système à l'ouverture d'un fichier exécutable absolu **.abs**. Les objets de bases sont décrits dans le chapitre suivant.

VI.3.1. Ouverture des objets :

Pour ouvrir un objet, on passe par le menu **component** de la barre de menu principale :



VI.3.2. SOURCE :






Cet objet affiche le code source de l'application.

Il permet de voir, changer et contrôler la ligne actuellement exécutée dans le programme.

La couleur du texte dépend de ce qui est affiché. Les mots clés du langage, les commentaires et les chaînes de caractères sont affichés dans une couleur différente (pour le langage C uniquement).

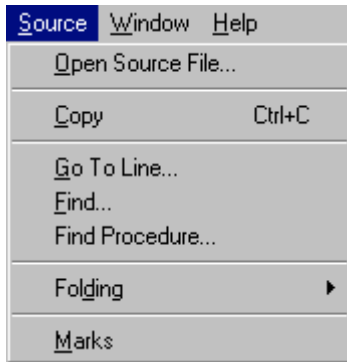
La ligne du programme correspondant à la valeur courante du PC est **affichée sur un fond bleu**. Cette ligne correspond à la prochaine instruction à être exécutée.

Si des points d'arrêts ou des points de "regard" ont été placés, ils sont marqués par des symboles en début de ligne.

Lorsque le fichier source est en langage C, il est possible de cacher une partie du code affiché correspondant à des blocks d'instructions (par exemple, fonctions, blocks délimités par des { }). Lorsque c'est possible, le block d'instructions est encadré par  et par . Pour cacher du code, il suffit de double-cliquer sur l'un des deux symboles. Les parties de codes ne pouvant être montrées sont délimitées par des . Pour montrer le code caché, il suffit de double-cliquer sur le symbole.

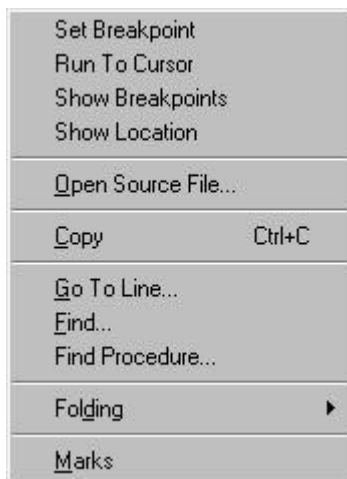
VI.3.2.1. Les menus :

Le menu de la barre de menu :



- Open Source File :** Ouvre une boîte de dialogue listant tous les fichiers sources de l'application. On choisit le fichier source à afficher dans cette fenêtre ;
- Copy :** Copie le texte sélectionné dans le presse papier de Windows ;
- Go To Line... :** Pose un point d'arrêt temporaire à l'endroit où le curseur se trouve et continue l'exécution de l'application jusqu'au point d'arrêt (sauf s'il a été posé à un endroit où un point d'arrêt avait été invalidé) ;
- Find... :** Ouvre une boîte de dialogue permettant de rechercher un texte dans le code source ;
- Find Procedure... :** Ouvre une boîte de dialogue permettant de positionner l'affichage sur une procédure (ou fonction) ;
- Folding... :** Commandes permettant de cacher ou de montrer tout ou partie du texte afficher ;

Le menu Contextuel :



- Run To Cursor :** Pose un point d'arrêt temporaire à l'endroit où le curseur se trouve et continue l'exécution de l'application jusqu'au point d'arrêt (sauf s'il a été posé à un endroit où un point d'arrêt avait été invalidé) ;
- Set Breakpoint :** Pose un point d'arrêt permanent à la position la plus proche du curseur. Cette commande n'est disponible que si aucun point d'arrêt n'a déjà été posé ;
- Enable/Disable Br :** Valide ou invalide un point d'arrêt ;
- Delete Breakpoint :** Supprime un point d'arrêt. Cette commande n'est disponible que si un point d'arrêt existe à la position du curseur ;
- Show Breakpoint :** Ouvre la fenêtre d'édition des points d'arrêt ;
- Show Location :** Met en valeur la portion de code assembleur (dans la fenêtre assembleur) correspondant à la ligne de code source sélectionnée ;
- Marks :** Bascule l'affichage des positions où les point d'arrêts peuvent être posés. Des petits triangles marquent ces positions.

VI.3.2.2. Le glisser/déposer :

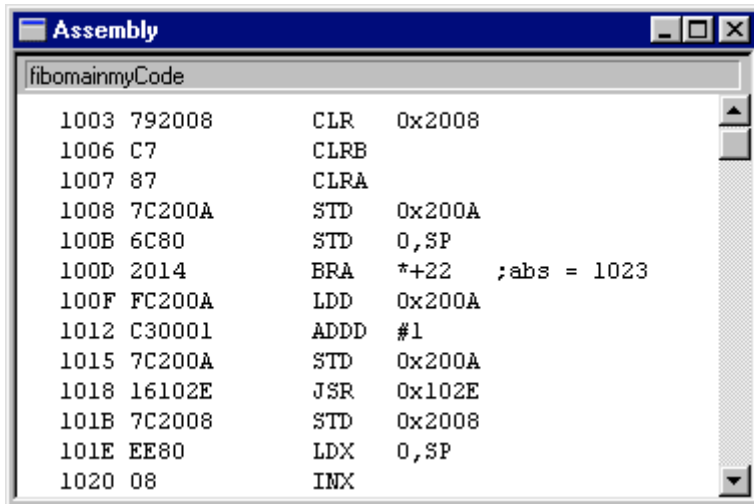
Ces différentes commandes sont accessibles en sélectionnant des éléments dans une fenêtre et en glissant la sélection jusque dans une autre fenêtre. La commande dépend de la fenêtre source et de la fenêtre destination.

Glisser/déposer de la fenêtre *source* vers une autre fenêtre :

Fenêtre destination	Commande
<i>Assembly</i>	Affiche le code désassemblé de la ligne sélectionnée en le mettant en valeur.
<i>Register</i>	Charge le registre de destination avec la valeur du PC de la première instruction sélectionnée.
<i>Data</i>	La sélection dans la fenêtre source est considérée comme une expression dans la fenêtre Data.

Glisser/déposer d'une autre fenêtre, vers la fenêtre *source* :

Fenêtre source	Commande
<i>Assembly</i>	La fenêtre source défile jusqu'à la ligne correspondant à l'instruction assembleur sélectionnée.
<i>Memory</i>	Affiche la ligne de code source commençant à la première adresse mémoire sélectionnée. Les instructions correspondants à l'espace mémoire sélectionné sont grisées.
<i>Module</i>	Affiche le code source du module sélectionné.

VI.3.3. ASSEMBLY :

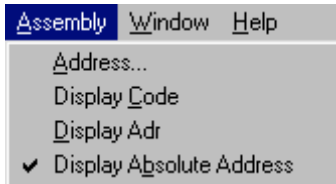
Cet objet affiche le code désassemblé de l'application.

Il possède des commandes similaires à la fenêtre source mais à un niveau d'abstraction plus faible.

Cette fenêtre peut afficher les informations suivantes : l'adresse, le code machine, l'instruction et les adresses absolue des instructions de branchement.

Les points d'arrêt positionnés dans la fenêtre source sont également affichés dans cette fenêtre.

Si l'exécution de l'application est arrêtée, l'instruction courante est **affichée sur un fond bleu.**

VI.3.3.1. Les menus :**Le menu de la barre de menu :****Address... :**

Permet d'entrer une adresse dans une boîte de dialogue. Le contenu de la mémoire commençant à l'adresse spécifiée est interprété et est affiché en tant que code assembleur ;

Display Code :

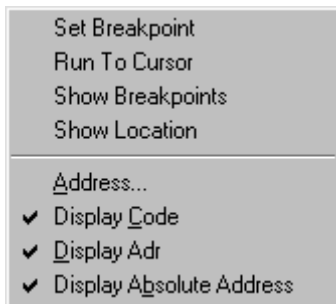
Affiche le code machine en face de chaque instructions désassemblée ;

Display Adr :

Affiche l'adresse de chaque instructions désassemblée ;

Display

Absolute Address : Affiche les adresses absolues dans les instructions de branchement.

Le menu contextuel :**Run To Cursor :**

Pose un point d'arrêt temporaire à l'endroit où le curseur se trouve et continue l'exécution de l'application jusqu'au point d'arrêt (sauf s'il a été posé à un endroit où un point d'arrêt avait été invalidé) ;

Set Breakpoint :

Pose un point d'arrêt permanent à la position la plus proche du curseur. Cette commande n'est disponible que si aucun point d'arrêt n'a déjà été posé ;

Enable/Disable Br : Valide ou invalide un point d'arrêt ;

Delete Breakpoint : Supprime un point d'arrêt. Cette commande n'est disponible que si un point d'arrêt existe à la position du curseur ;

Show Breakpoints : Ouvre la fenêtre d'édition des points d'arrêt ;

Show Location : Met en valeur la portion de code assembleur (dans la fenêtre assembleur) correspondant à la ligne de code source sélectionnée ;

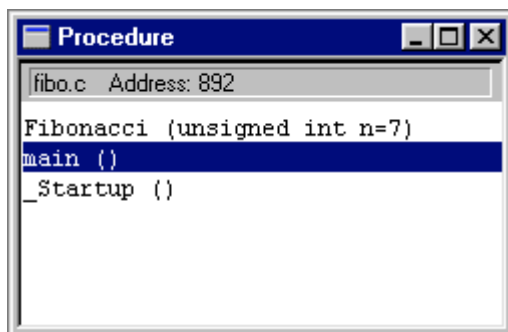
VI.3.3.2. Le glisser/déposer :

Glisser/déposer de la fenêtre *Assembly* vers une autre fenêtre :

Destination	Commande
<i>Command Line</i>	L'objet <i>Command Line</i> affiche l'adresse de l'instruction pointée ;
<i>Memory</i>	Affiche la zone mémoire commençant à l'adresse de l'instruction sélectionnée ;
<i>Register</i>	Charge le registre de destination avec l'adresse de l'instruction sélectionnée ;
<i>Source</i>	La fenêtre <i>source</i> défile jusqu'à l'instruction source correspondante.

Glisser/déposer d'une autre fenêtre, vers la fenêtre *source* :

Fenêtre source	Commande
<i>Source</i>	Affiche les instructions désassemblées commençant à la première instruction du code source sélectionnée. Toutes les instructions correspondantes sont mises en valeur ;
<i>Memory</i>	Affiche les instructions désassemblées commençant à la première adresse mémoire sélectionnée. Les instructions correspondants à l'espace mémoire sélectionné sont grisées.
<i>Register</i>	Affiche les instructions désassemblées commençant à l'adresse mémoire chargée dans le registre.
<i>Procedure</i>	Met en valeur les instructions désassemblées correspondants au symbole sélectionné.

VI.3.4. Procedure :

Cet objet affiche la liste des procédures ou des fonctions appelées au moment où l'exécution de l'application a été arrêtée. Cette liste est aussi appelée chaîne d'appel. Les fonctions de cette chaîne d'appel sont affichées dans l'ordre inverse d'exécution, depuis la dernière (la plus récente en haut) jusqu'à la première.

La valeur et le type des paramètres sont aussi affichés.

La barre d'information contient le nom du fichier source et l'adresse de la fonction sélectionnée.

En effectuant un double-clic sur le nom d'une fonction, entraîne l'affichage d'information sur cette fonction dans toutes les fenêtres des objets ouvertes : La fenêtre *source* affiche le code source de la fonction, la fenêtre des variables locales affiche les variables et paramètres de la fonction, l'instruction courante de la fonction sélectionnée est mise en valeur dans la fenêtre d'assemblage.

VI.3.4.1. Le Menu :

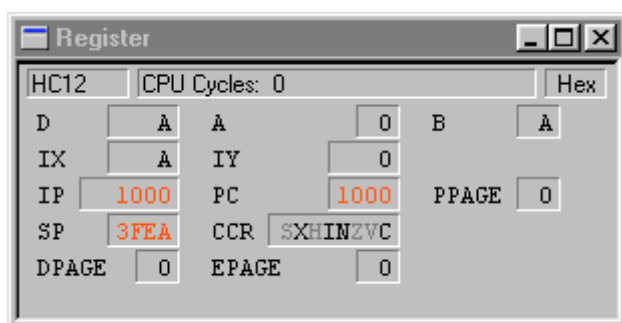
Un seul menu est disponible pour cette objet :

- **Show Values :** Affiche la valeur des paramètres des fonctions ;
- **Show Types :** Affiche le type des paramètres des fonctions ;

VI.3.4.2. Le glisser/déposer :

On ne peut pas faire un glisser/déposer dans cette fenêtre mais seulement à partir de cette fenêtre.

Destination	Commande
<i>Data / Local</i>	Affiche les variables locales de la fonction sélectionnée.
<i>Source</i>	Affiche le code source de la fonction sélectionnée. L'instruction courante est mise en valeur sur un fond bleu.
<i>Assembly</i>	L'instruction courante à l'intérieur de la fonction est mise en valeur.

VI.3.5. Register :

Cet objet affiche le contenu des registres de l'unité centrale du microcontrôleur (CPU12).

Le contenu des registres peut être affiché en binaire ou en hexadécimal et on peut changer leur valeurs.

Les bits du registre d'états (CCR) sont affichés en noir quand ils sont à 1 et en gris quand ils sont à 0. On peut changer la valeur d'un bit en double-cliquant dessus.

Pendant l'exécution de l'application, le contenu des registres qui ont été changés depuis le dernier rafraîchissement sont affichés en rouge, sauf pour le registre d'état. La barre d'information affiche le nom du microcontrôleur ainsi que le nombre de cycles machines.

Pour changer le contenu d'un registre, il suffit de double-cliquer dessus et une fenêtre de dialogue apparaît où le contenu peut être changé.

Pour changer l'affichage de la valeur en binaire ou en hexadécimal, un clic sur le bouton droit de la souris fait apparaître un menu où il est possible de choisir le type d'affichage.

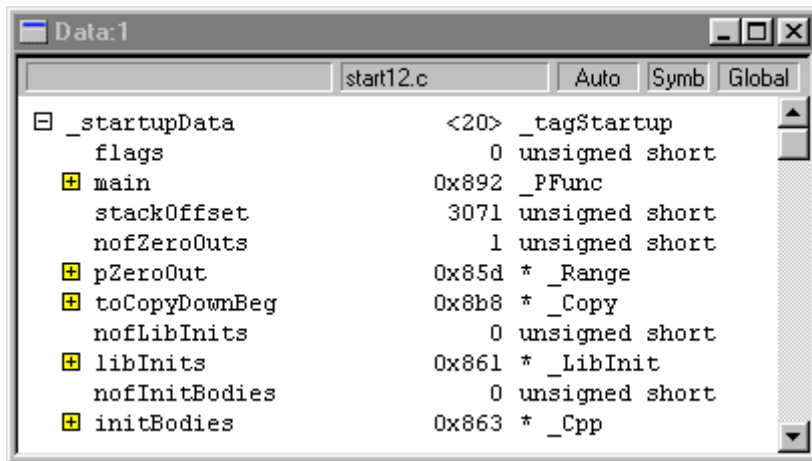
Le glisser/déposer :

Glisser/déposer de la fenêtre *Register* vers une autre fenêtre :

Destination	Commande
<i>Assembly</i>	La fenêtre d'assemblage défile jusqu'à l'instruction correspondant à la valeur déposée et interprétée comme une adresse.
<i>Memory</i>	Affiche l'espace mémoire correspondant l'adresse contenue dans le registre sélectionné.
<i>Command Line</i>	L'adresse contenue dans le registre sélectionné est ajouté à la commande en cours.

Glisser/déposer d'une autre fenêtre, vers la fenêtre *Register* :

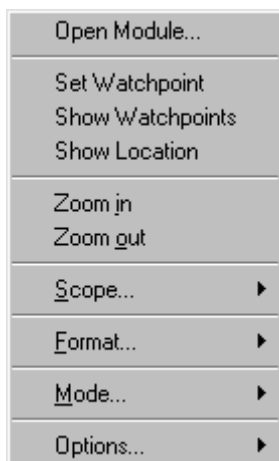
Fenêtre source	Commande
<i>Source</i>	Charge le registre de destination avec l'adresse de la première instruction sélectionnée.
<i>Memory</i>	Charge le registre de destination avec l'adresse de base du block mémoire sélectionné.
<i>Assembly</i>	Charge le registre de destination avec l'adresse de la première instruction sélectionnée.
<i>Data</i>	Si le nom d'une variable est déposé, le registre destination est chargé avec l'adresse de la variable sélectionnée, si c'est la valeur, alors le registre destination est chargé avec celle-ci.

VI.3.6. Data :

Cet objet affiche le nom, la valeur et le type de toutes les variables globales et locales du module (ou des fonctions d'un fichier source) en cours d'exécution.

La barre d'information contient l'adresse et la taille de la variable sélectionnée ainsi que le nom du module (ou fichier source...), le mode et le format d'affichage et la portée des variables affichée (globale, locale).

On peut définir des points de "regard" (Watchpoints) sur des variables.

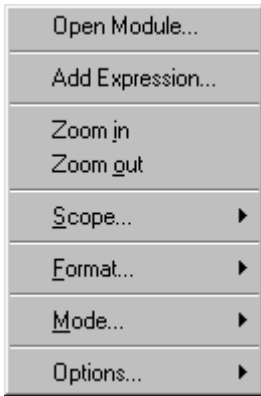
VI.3.6.1. Les menus :

- Open Module... :** Ouvre une boîte de dialogue listant tous les fichiers sources de l'application. Les variables globales du fichier sélectionné sont alors affichées dans la fenêtre. Commande disponible uniquement si la portée sélectionnée est *Global* ;
- Set Watchpoint :** Pose un point de "regard" en lecture/écriture sur la variable sélectionnée. Cette commande n'est disponible que si aucun point de "regard" n'a déjà été posé ;
- Delete Watchpo. :** Supprime un point de "regard". Cette commande n'est disponible que s'il y en a un ;
- Enable/disable Watchpoint :** Valide ou invalide un point de "regard". Cette commande n'est disponible que s'il y en a un ;
- Show watchpoints:** Ouvre la fenêtre d'édition des point de "regard" ;
- Show Location :** Force l'affichage des informations sur la variable sélectionnée dans toutes les fenêtres des objets ouverts ;
- Zoom in :** Développe la structure sélectionnée ;
- Zoom out :** Retourne au niveau de développement précédent ;

Scope... : Permet de sélectionner l'affichage des variables globales, locales ou utilisateurs ;

Format... : Permet de choisir le format d'affichage des variables entre *Symbolic* (le format dépend du type de la variable), **Hexadécimal**, **Octal**, **Binaire**, **Décimal signé** ou **décimal non signé (Udec)** ;

Mode... : Permet de choisir le mode de mise à jour des variables : **Automatique** (mise à jour à l'arrêt de l'application), **Périodique** (période à définir multiple de 100ms), **Vérouillé** ou **Locked** (à l'arrêt de l'application) ou **Gelé** ou **Frozen** (pas de mise à jour) ;



Add Expression... : Permet, à l'aide d'une boîte de dialogue, de définir une expression. Une expression utilise les variables déjà présentes dans la fenêtre *Data* et utilise la syntaxe du C-ANSI. Exemple :

```
((variable_1 << variable_2) + 0xFF) <= 0x1000  donnera un résultat booléen ;
(variable_1 >> variable_2) * 0x1000             donnera un résultat entier ;
```

Edit Expression... : Permet de modifier une expression déjà existante ;

Delete Expression : Supprime l'expression pointée par la souris ;

VI.3.6.2. Le glisser/déposer :

Glisser/déposer de la fenêtre *Data* vers une autre fenêtre :

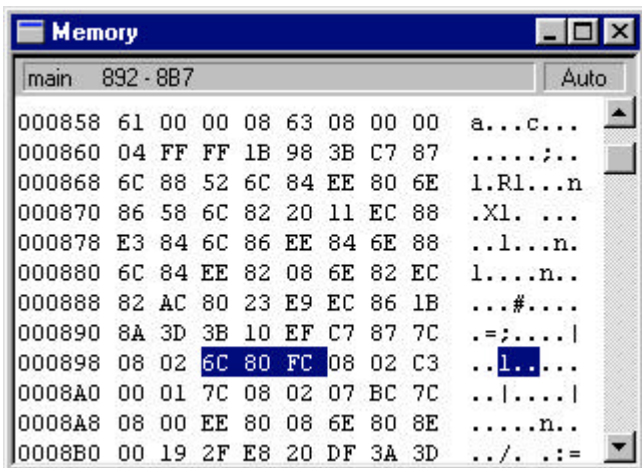
Destination	Commande
<i>Command Line</i>	Si c'est le nom de la variable sélectionné qui est déposé, alors l'adresse de la variable est ajouté à la ligne de commande, si c'est la valeur, c'est elle qui sera ajouté ;
<i>Memory</i>	Affiche et sélectionne l'espace mémoire correspondant à la variable sélectionnée ;
<i>Register</i>	Si le nom d'une variable est déposé, le registre destination est chargé avec l'adresse de la variable sélectionnée, si c'est la valeur, alors le registre destination est chargé avec celle-ci.

Glisser/déposer d'une autre fenêtre, vers la fenêtre *Data* :

Fenêtre source	Commande
<i>Source</i>	La sélection dans la fenêtre source est considérée comme une expression dans la fenêtre <i>Data</i> ;
<i>Module</i>	Affiche les variables globales du module dans la fenêtre de données (<i>Data</i>).

Plusieurs fenêtre de variables peuvent être ouvertes en même temps. On pourra ainsi utiliser une fenêtre pour visualiser les variables globales et locales de chaque fichier source...

VI.3.7. Memory :



Cet objet affiche le contenu de la mémoire.

On peut choisir d'afficher le contenu de la mémoire avec différents formats (Byte, Word et Double 32bits) et différentes bases (Binaire, octal, hexadécimal, décimal et décimal non signé).

On peut définir des point de "regard" (Watchpoints).

La fenêtre peut être divisée en trois parties : adresses, contenus et équivalence ASCII.

Le contenu d'une zone de mémoire peut être initialisée avec une valeur particulière.

La barre d'information affiche le nom de la fonction ou de la variable, le champs de la structure et les adresses de début et de fin correspondant au premier mot du block de mémoire sélectionné.

La valeur "uu" correspond à une case mémoire non initialisée ;

La valeur "--" correspond à une case mémoire non disponible.

VI.3.7.1. Les menus :



Word Size... : Ouvre un sous-menu permettant de choisir le format d'affichage de la valeur entre *Byte*, *Word* et *Double* ;

Format... : Ouvre un sous-menu permettant de choisir la base des valeurs affichées ;

Mode... : Permet de choisir le mode de mise à jour des valeurs : *Automatique*, *périodique* ou *"Frozen"* ;

Display... : Permet de valider l'affichage des adresses et du code ASCII ;

Address... : Ouvre une boîte de dialogue permettant de saisir l'adresse de base de la zone mémoire que l'on veut visualiser ;

Fill... : Ouvre une fenêtre de dialogue permettant de saisir l'adresse de début et de fin de la zone mémoire que l'on veut initialiser avec une valeur, que l'on saisit également.

On a également accès à un menu pop-up contextuel qui nous permet d'avoir accès aux commandes liées aux points de "regard" ou *Watchpoints*. Le fonctionnement de ces commandes est le même que dans l'objet *Data* mais non plus associé à une variable mais sur une zone mémoire sélectionnée...

Les modes de mise à jour de la mémoire sont les mêmes que l'objet *Data* et on se référera à celui-ci pour plus de détails.

VI.3.7.2. Le glisser/déposer :

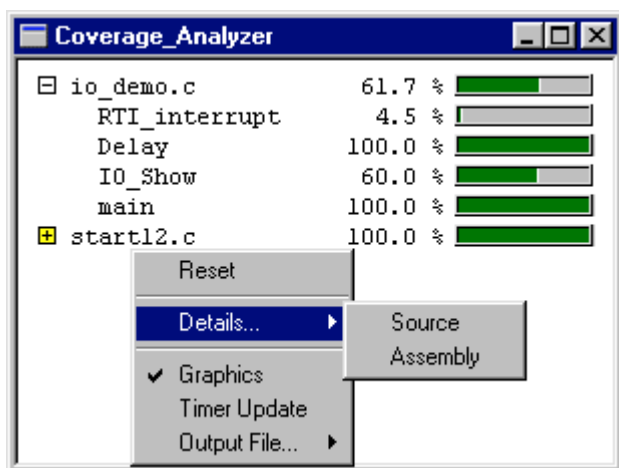
Glisser/déposer de la fenêtre *Memory* vers une autre fenêtre :

Destination	Commande
<i>Assembly</i>	Affiche les instructions désassemblées commençant à la première adresse mémoire sélectionnée. Les instructions correspondants à l'espace mémoire sélectionné sont grisées.
<i>Command Line</i>	Ajoute l'espace mémoire sélectionné à la ligne de commande courante.
<i>Register</i>	Charge le registre de destination avec l'adresse de base du block mémoire sélectionné.
<i>Source</i>	Affiche la ligne de code source commençant à la première adresse mémoire sélectionnée. Les instructions correspondants à l'espace mémoire sélectionné sont grisées.

Glisser/déposer d'une autre fenêtre, vers la fenêtre *Memory* :

Fenêtre source	Commande
<i>Assembly</i>	Affiche la zone mémoire commençant à l'adresse de l'instruction sélectionnée ;
<i>Data</i>	Affiche et sélectionne l'espace mémoire correspondant à la variable sélectionnée ;
<i>Register</i>	Affiche l'espace mémoire correspondant l'adresse contenue dans le registre sélectionné.
<i>Module</i>	Affiche l'espace mémoire commençant à l'adresse de la première variable globale du module. La zone mémoire où cette variable est située est sélectionnée.

VI.3.8. Coverage :

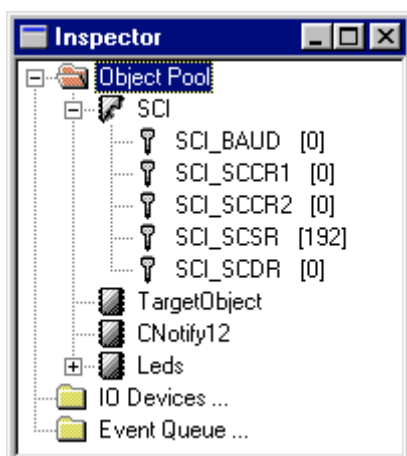


Cet objet permet de réaliser des statistiques sur l'exécution des différentes fonctions. Il indique des pourcentages représentant les proportions dans lesquelles les fonctions ont été exécutées. Il fait le rapport du nombre d'instructions exécutées sur le nombre total d'instruction dans la fonction.

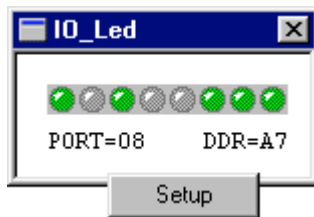
Cet objet peut fonctionner en parallèle avec l'objet *source* et *assembly*

Pour plus d'informations, on se référera à l'aide en ligne.

VI.3.9. Inspector :

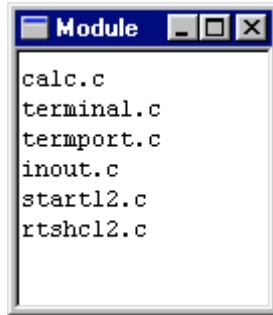


Cet objet affiche la liste des objets chargés dans la fenêtre principale. A chaque fois qu'un nouvel objet est ouvert, les informations sur l'objet sont transmises à l'inspecteur. Il affiche alors les informations et les valeurs associées à chaque objet.

VI.3.10. IO_LED :

Cette fenêtre représente 8 leds utilisées pour manipuler et visualiser le contenu de la mémoire à l'adresse spécifiée dans la boîte de dialogue associée. La couleur des leds sont définies à l'adresse du PORT et mis à 1 ou à 0 par le biais du registre DDR dont l'adresse est spécifiée dans la boîte de dialogue.

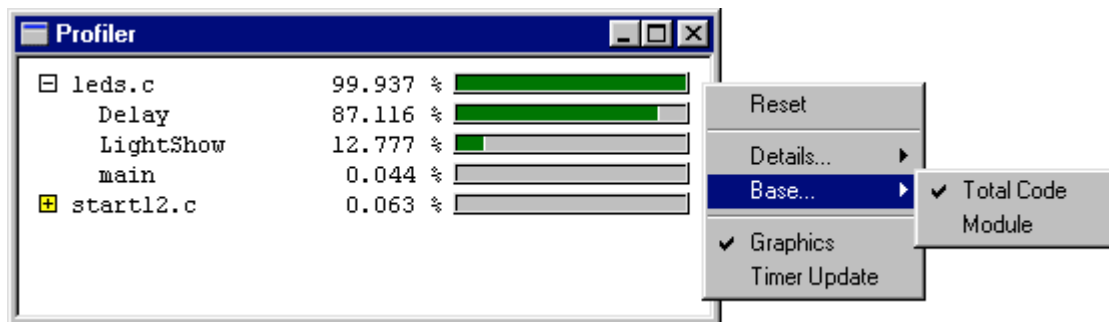
Le menu *Setup* ouvre la boîte de configuration de l'objet. On peut définir l'adresse du IO_LED PORT et du IO_LED DDR.

VI.3.11. Component :

Cet objet affiche tous les fichiers sources (ou modules) composants l'application.

En double-cliquant sur un des modules, on lance l'affichage des informations sur le module : la fenêtre source affiche le code source et la fenêtre de variables (*Data*) affiche toutes les variables globales.

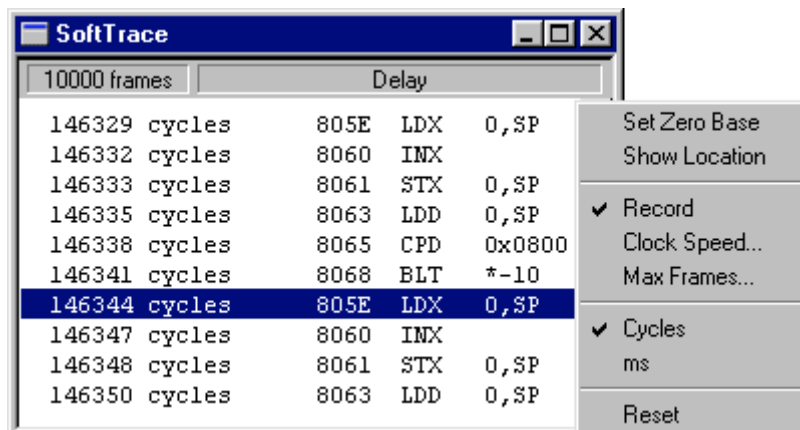
Le glisser/déposer est possible dans cet objet et on se référera aux objets précédents pour connaître les commandes disponibles.

VI.3.12. Profiler :

Cet objet affiche le nom des différents fichiers sources et des différentes fonctions auxquels correspondent des pourcentages représentant le temps passé dans chaque fonctions et module. Cet objet fonctionne globalement comme l'objet *coverage*. On se référera à l'aide en ligne pour plus de précisions.

VI.3.13. Recorder :

L'objet *recorder* permet d'enregistrer toutes les commandes exécutées lors d'une session de débogage. On se référera à l'aide en ligne pour son utilisation.

VI.3.14. SoftTrace :

L'objet *SoftTrace* enregistre et affiche les instructions exécutées en fonction du temps. Le temps peut être affiché en millisecondes ou en nombre de cycles machine.

On peut fixer une instruction particulière comme étant la référence du temps et toutes les autres instructions seront repérées par rapport à celle-ci.

Le nombre maximum d'instructions enregistrées ainsi que la fréquence d'horloge du microprocesseur peuvent être fixées grâce à un menu.

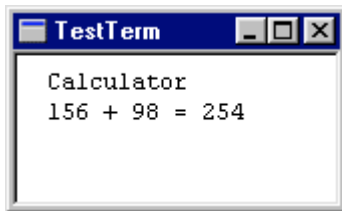
On se référera à l'aide en ligne pour plus de précisions.

VI.3.15. Command Line :

Cet objet permet d'entrer des commandes du simulateur. Chaque objet possède des commandes accessibles sous par des menu, soit par des raccourcis souris-clavier (non développés ici) soit par des glisser/déposer. Une partie de ces commandes sont accessibles aussi à l'aide de l'objet "ligne de commande" où il suffit de taper de nom associé à la commande. Les commandes disponibles pour chaque objet sont décrites dans l'aide en ligne.

On peut aussi créer un fichier de commande (au format texte) et lancer son exécution par le menu.

Des paramètres peuvent être passés par glisser/déposer depuis les fenêtres *Assembly*, *Data*, *Memory* et *Register*. On trouvera les combinaisons possibles aux chapitres correspondants à chaque objet.

VI.3.16. TestTerm :

L'objet *TestTerm* est un terminal d'entrée/sortie fonctionnant sur une interface SCI qui est indépendante de la cible. Il émule une interface de communication série basée à l'adresse 200h, et par conséquent, simule les cinq registres de contrôle de cette interface.

VI.3.16.1. Fonctionnement :

Les cinq registres de contrôle sont résumés dans le tableau suivant :

Nom du Registre	Fonction	Adresse
BAUD	Contrôle de la vitesse de communication	0x200
SCCR1	Registre de contrôle de la communication série	0x201
SCCR2	Registre de contrôle de la communication série	0x202
SCSR	Registre d'état	0x203
SCDR	Registre de données	0x204

Dans le registre d'état, les bits suivants sont utilisés :

- **TDRE** : Transmit Data Register Empty (valeur du masque : 0x80) ;
- **RDRF** : Receive Data Register Full (valeur du masque : 0x20) ;

L'écriture et la lecture dans les registres **BAUD**, **SCCR1**, **SCCR2** et **SCSR** n'ont aucun effet dans la fenêtre du terminal, mais ils sont nécessaires pour garder la compatibilité avec les interfaces SCI. Les entrées/sorties du terminal n'ont pas besoin d'être initialisés. Dans le fichier **termio.c**, les registres BAUD et SCSR sont initialisés pour être compatibles avec les vraies interfaces SCI.

Le registre SCDR est accessible en lecture et en écriture. Quand une valeur est lue, le bit RDRF est mis à 0. De plus, lorsque l'utilisateur entre un caractère par le clavier alors que la fenêtre du terminal est active, le bit RDRF est mis à 1 et la valeur ASCII du caractère saisi est rangée dans le registre de donnée SCDR.

Quand une valeur est écrite dans le registre SCDR par l'application, le bit TDRE est mis à 0. Quand la transmission est fine, le bit TDRE est à nouveau mis à 1. Comme l'objet *TestTerm* est juste une émulation d'une interface d'entrée/sortie, aucun temps de retard n'est simulé et donc, le bit TDRE est immédiatement mis à 1 lors de l'écriture d'une valeur dans le registre SCDR.

VI.3.16.2. Utilisation :

Pour utiliser le terminal, on peut utiliser les fonctions pré-définies dans le fichier **termport.h** :

```
char GetChar ( void ) ;          void PutChar ( char ch ) ;
void PutString ( char *str ) ;   void InitTermIO ( void ) ;
```

Exemple de définition de GetChar() :

```
typedef struct {
    unsigned char BAUD ;
    unsigned char SCCR1 ;
    unsigned char SCCR2 ;
    unsigned char SCSR ;
    unsigned char SCDR ;
} SCIStruct ;

#define SCI (*(( SCIStruct* )( 0x0200 )))

char GetChar(void)
{
    while ( !( SCI.SCSR & 0x20 )) ; /* attend une entrée */
    return SCI.SCDR;
}
```

VI.3.17. Terminal :

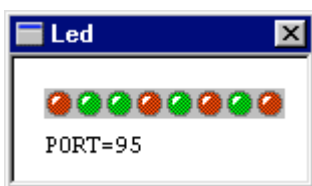
L'objet *Terminal* utilise les fonctionnalités d'une interface SCI fournie par l'environnement de débogage. Ce terminal ne fonctionne que si une interface SCI est présente ou si d'autres objets simulants des entrées/sorties sont présents. Il simule les entrées/sorties avec les outils associée à l'interface SCI.

Pour utiliser ce terminal, il faut donc qu'une interface SCI soit présente dans l'environnement de débogage. Pour fonctionner il faut vérifier que la cible (Simulateur ou carte d'évaluation), possède une interface SCI. Pour cela on peut ouvrir l'objet *Inspector* pour visualiser tous les objets présents. Il faut trouver un objet dont le nom est "**sci0**". Si aucun objet de ce type existe, alors aucune entrée/sortie par le terminal ne sera possible.

VI.3.18. Les utilitaires de visualisation :

A coté de tous ces objets de bases, il existe une famille d'utilitaires plus proches du système dans lequel l'application qui est développée sera implantée. Ces utilitaires doivent permettre de simuler le fonctionnement du système dans son ensemble. Ils sont donc développés à la demande.

Parmi ces objets, des utilitaires de visualisation existent pour visualiser les valeurs, les registres, les cases mémoires, etc., d'une façon graphique. Dans la version de base d'HIWARE, seuls trois utilitaires sont définis, mais deux ne sont pas finis et peuvent être utiles comme base de nouveaux utilitaires à développer. Nous ne verrons donc que l'objet *LED* :

L'objet LED :

Cet objet permet de visualiser le contenu d'une case mémoire de 8bits, le MSB étant à gauche. Les 1 sont codés par du rouge et les 0 par du vert. On définit l'adresse de la case mémoire à visualiser à l'aide d'une boîte de dialogue que l'on fait apparaître à l'aide d'un menu.

VI.4. Contrôle de l'exécution :

VI.4.1. Exécution et arrêt :

Les commandes d'exécution et d'arrêt de l'application sont accessibles soit par le menu *Run*, soit par des icônes de la barre d'outils. Ces commandes sont détaillées au chapitre **VI.2.1.3**.

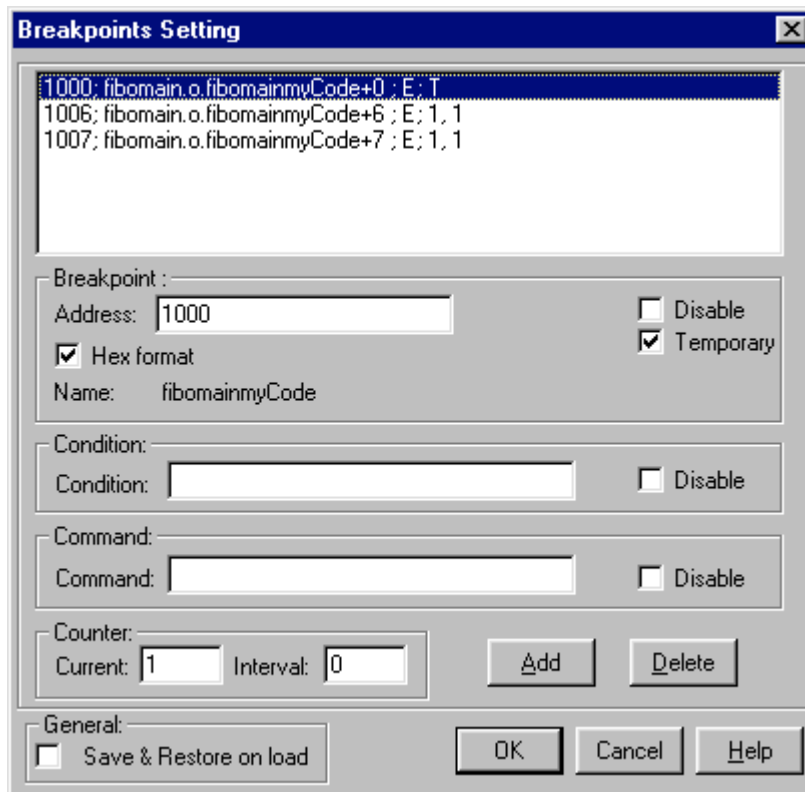
VI.4.2. Les points de contrôle :

Il existe deux type de points de contrôle : Les point d'arrêts ou "Breakpoints" et les points de regard ou "Watchpoints". Les points d'arrêts sont localisés à une adresse et ont donc une adresse. Ils peuvent être permanent ou temporaires. Les points de regard sont localisés dans une zone de mémoire.

On peut valider ou invalider un point de contrôle, imposer une condition et définir une commande associée.

VI.4.2.1. Édition des points d'arrêts :

On peut éditer tous les points d'arrêt définis dans une application à l'aide de la commande **Breakpoints...** ou **Show Breakpoints...** qui apparaît dans certains menus. Cette commande ouvre alors la fenêtre d'édition des points d'arrêt :



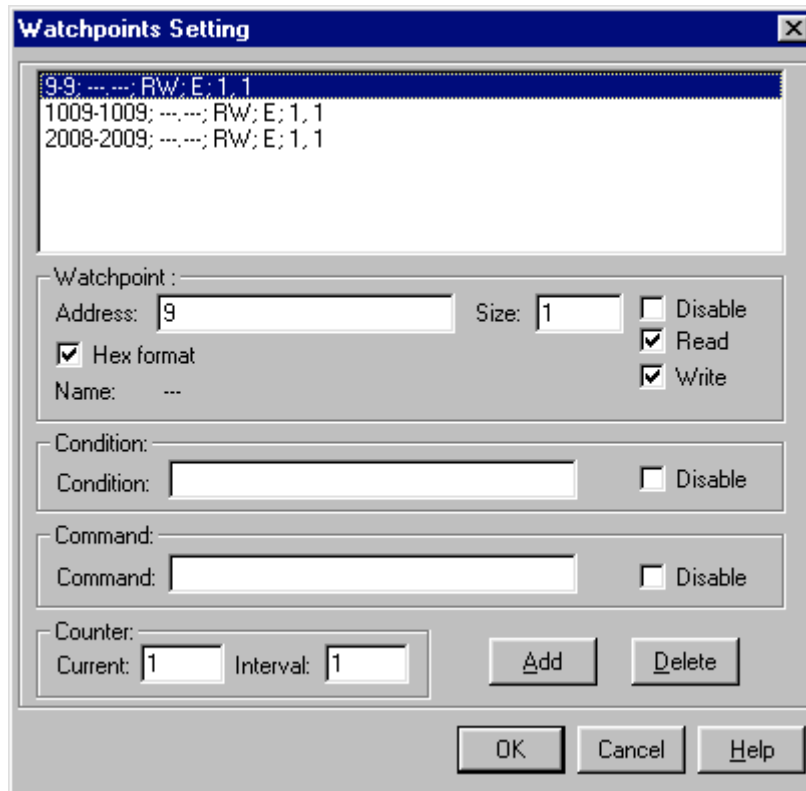
Cette fenêtre est divisée en six parties :

- Une zone de texte où apparaît la liste des points d'arrêt déjà définis avec leurs caractéristiques ;
- Quatre groupes correspondants aux caractéristiques du point d'arrêt sélectionné :
 - "Breakpoint :" : Affiche l'adresse du point d'arrêt, le nom de la fonction où il apparaît, l'état courant (validé ou non) et le type de point d'arrêt (permanent ou temporaire) ;
 - "Condition :" : Affiche et permet d'éditer une condition sur le point d'arrêt. Si la condition n'est pas réalisée, le programme ne s'arrête pas sur ce point d'arrêt. On peut invalider la condition ;
 - "Command :" : Affiche et permet d'éditer la commande associée au point d'arrêt. Lorsque le programme s'arrête sur ce point d'arrêt on peut lui demander d'exécuter une commande. On peut invalider la commande ;
 - "Counter :" : affiche et permet de définir la valeur du compteur et de l'intervalle associé au point d'arrêt. A chaque fois que le programme passe par le point d'arrêt, la valeur du compteur est décrémentée en partant de la valeur de l'intervalle. Le programme ne s'arrête que si la valeur du compteur est égale à 0.
- Lorsque la case "Save & Restore on load" est cochée, tous les points d'arrêts définis sont enregistrés dans un fichier .BPT du même nom que le fichier .ABS, et sont automatiquement redéfinis lorsque le fichier absolu .ABS correspondant est ouvert dans le simulateur.

Enfin, on peut ajouter ou supprimer des points d'arrêt à partir de cette fenêtre.

VI.4.2.2. Édition des points de regard :

On peut éditer tous les points de regard définis dans une application à l'aide de la commande **Watchpoints...** ou **Show Watchpoints...** qui apparaît dans certains menus. Cette commande ouvre alors la fenêtre d'édition des points de regard :



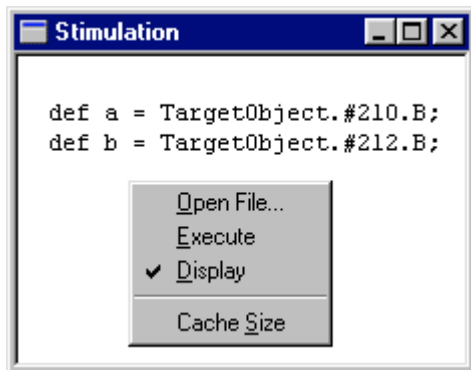
Cette fenêtre est divisée en cinq parties :

- Une zone de texte où apparaît la liste des points de regard déjà définis avec leurs caractéristiques ;
- Quatre groupes correspondants aux caractéristiques du point de regard sélectionné :
 - "Watchpoint :" : Affiche l'adresse du point de regard, sa taille, le nom de la fonction où il apparaît, son état (validé ou non), accès en lecture et/ou en écriture (validés ou non) ;
 - "Condition :" : Affiche et permet d'éditer une condition sur le point de regard. Si la condition n'est pas réalisée, le programme ne s'arrête pas sur ce point de regard. On peut invalider la condition ;
 - "Command :" : Affiche et permet d'éditer la commande associée au point de regard. Lorsque le programme s'arrête sur ce point de regard on peut lui demander d'exécuter une commande. On peut invalider la commande ;
 - "Counter :" : affiche et permet de définir la valeur du compteur et de l'intervalle associé au point de regard. A chaque fois que le programme passe par le point de regard, la valeur du compteur est décrémentée en partant de la valeur de l'intervalle. Le programme ne s'arrête que si la valeur du compteur est égale à 0.

On peut ajouter ou supprimer des points de regard depuis cette fenêtre.

VI.5. Stimulation des entrées/sorties :

On peut simuler des événements provenant de la partie Hardware comme le changement d'état des registres ou simuler l'apparition d'interruptions à l'aide d'un objet particulier : **IO Stimulation**. Cet objet utilise des fichiers textes dans lesquels sont enregistrés des commandes dont la syntaxe sera vue plus loin. Il interprète chaque commande et change les paramètres correspondant pendant la session de simulation.



Pour lancer la stimulation, il faut d'abord ouvrir l'objet :

⇒ menu Component | Open | *Stimulat*.

Ensuite, il faut ouvrir un fichier de stimulation .TXT :

⇒ menu *S*timulation | Open File...

Puis, on peut démarrer l'exécution de la stimulation :

⇒ menu *S*timulation | Execute

La stimulation commencera au démarrage de l'exécution de l'application.

VI.5.1. Syntaxe des fichiers de stimulation :

Les fichiers de stimulation sont des fichiers textes dont l'extension est **.TXT**.

Ils sont composés d'instructions mises les unes à la suite des autres et terminées par un " ;". Les instructions sont regroupées en trois groupes différents :

• Déclarations :

La déclarations des cases mémoires ou des registres affectés par la stimulation doit d'abord se faire pour pouvoir y faire référence :

def *nom_de_var* = *Object.champ* ;

- *nom_de_var* : nom de la variable ;
- *Object.champ* : référence à la case mémoire ou au registre concerné. Cette référence doit apparaître dans l'objet *inspector*. Le format de ce champ est : *ObjectName.FieldName*

ObjectName : Nom de l'objet, par exemple **TargetObject** ;

FieldName : Nom du champs concerné : un identificateur ou une adresse hexadécimale. Il peut aussi y avoir une indication de format : **.B**, **.W** ou **.L**.

Exemple :

def a = TargetObject.#210.B ;

Dans cet exemple, a représente l'octet (.B) de mémoire d'adresse 210h (# pour préciser hexadécimal) dans l'objet *TargetObject* correspondant à la cible utilisée.

On peut aussi faire référence uniquement à des bits. Il suffit d'ajouter [*startbit* : *nombre_de_bits*]

Par exemple : champs de 3 bits à partir du 5^{ème} bit, [5 : 3] donc les bits b5,b4 et b3.

• Événements :

On peut définir un événement pour qu'il apparaisse au bout d'un certain temps. Un événement peut être un changement de valeur dans ne case mémoire ou un registre, l'apparition d'une interruption.

Changement de valeur :

Temps *nom_de_var* = *Expression* ;

- *Temps* : temps représenté par un nombre de cycle machine. Plusieurs possibilité :
Avec un "+" devant : temps relatif à la dernière spécification de temps ;
Avec un "#" devant : temps absolu, temps écoulé depuis le démarrage de l'application.
Par défaut, c'est à dire sans rien devant : temps écoulé depuis que l'exécution du fichier de stimulation à démarré.
- *Expression* : Expression avec la même syntaxe que le C-ANSI ;

Exemples : #10000 a = var2 + 3 ;
+1000 a = 0 ;

Interruption :

Temps **RAISE** *vecteur , priorité , "nom" ;*

- *Temps* : temps représenté en cycle machine. Voir plus haut ;
- *vecteur* : numéro de vecteur. Il faudra se référer aux interruptions du microcontrôleur ;
- *priorité* : numéro de priorité ;
- *nom* : champs optionnel, permet de nommer l'interruption simulée.

Exemple : 200000 RAISE 7, 3, "test_interrupt" ;

Dans cet exemple, 200000 cycles machine après le démarrage de l'exécution du fichier de stimulation (menu Stimulation | execute), l'interruption de vecteur 7 (l'interruption RTI) ;

• **événements périodiques :**

On peut faire en sorte que des événements se répètent un certain nombre de fois. Les événements sont du même type que les événements précédents.

PERIODICAL *debut , nombre_de_fois :*
 événements ;

END

- *debut* : temps passé, à partir du temps initial, avant que l'événement périodique ne soit exécuté ;
- *nombre_de_fois* : nombre de fois que doit être exécuter l'événement périodique ;

Plusieurs événements peuvent être mis dans le block des événements périodiques, ils doivent être mis avant le END.