La Programmation Structurée

Table des Matières



Remerciements

Avant de commencer, je voudrais d'abord remercier Colette Michel qui m'a fortement inspiré lors de la rédaction du début de ce fascicule. En effet, lors de la rédaction de son ouvrage "Ecrire en dBase" en 1986, elle décrivait déjà parfaitement les méthodes de programmation structurée que nous utiliserons dans notre cours.

Qu’est-ce que Programmer ?

La réponse à cette question est tellement vaste, qu’il nous est impossible de donner une réponse complète. Nous allons toutefois essayer de donner un élément de réponse.

La programmation n’est pas simplement l’introduction de commandes et d’instructions correctes dans un micro-contrôleur, cet aspect-là n’étant que la partie émergée de l’iceberg. La programmation est un processus complexe, à plusieurs étapes, qui demande au préalable une planification soigneuse et, à tous les stades du travail, une vérification rigoureuse.

Un logiciel présente un niveau de difficulté très variable, depuis le simple programme qui fait clignoter une LED, jusqu’aux applications de grande envergure nécessitant des milliers d’heures de travail accomplies par des dizaines de programmeurs, tel que le programme de gestion d’un vidéo ou d’un GSM par exemple.

Plus le projet est important, plus il est essentiel de respecter les bases d’une bonne programmation. Même dans des applications très modestes, la mise en oeuvre de ces techniques ne peut qu’aider car elles rendent les programmes plus simples, plus clairs, plus lisibles, plus compréhensibles.

Aucun problème, n’admet de solution unique, traduisible par un programme défini. Il existe beaucoup de façons de le résoudre par des programmes efficients. Mais la plupart des spécialistes en programmation s’accordent sur la division du processus  en cinq étapes logiques :

1 ) - Définition du problème
2 ) - Conception de la solution
3 ) - Ecriture du programme
4 ) - Documentation
5 ) - Vérification et recherche des erreurs.

Vous constaterez que l’écriture (ou l’encodage) n’est qu’une étape, et certes pas la plus longue ni la plus difficile.

Qu’est-ce que Programmer ?

Etape n° 1 : Définition du problème

Le premier stade est de bien comprendre le problème qui se pose. Quel est le but de l’application ? Tâchez d’arriver à le formuler en une phrase. Si le problème est trop complexe, il faut alors le subdiviser et définir l’objectif à atteindre par chaque partie.

La première question à ce poser, pour bien appréhender le problème, est de ce demander, quels sont les résultats attendus ? C’est-à-dire de commencer par la fin. Interviewez le ou les utilisateurs et essayez de définir exactement quels sont leurs besoins. Demandez leurs ce qu’ils attendent de l’appareil que vous êtes en train de concevoir.

A ce stade, il est déjà temps de commencer la documentation de l’application, ce point trop souvent négligé et qui est pourtant vital dans l’avenir de votre travail. Gardez toutes vos notes sur les besoins et les avis de toutes les personnes concernées. Nous développerons plus loin la notion de documentation d’une application.

Vous pouvez ensuite passer à la deuxième Question. Quelles sont les entrées nécessaires pour permettre le déroulement du processus ? Quelles sont les informations nécessaires au traitement de l’information ?

La troisième question à se poser est : quel est le matériel que je vais devoir mettre en oeuvre ? Vous pouvez alors définir les interfaces d’entrées et de sorties que vous allez devoir ajouter à votre micro-contrôleur pour que l’application étudiée puisse tourner. Cette étape est très importante, car le bon choix des interfaces vous permettra parfois de simplifier le code utilisé par votre micro-contrôleur.

La quatrième question à se poser est : de quel micro-contrôleur vais-je avoir besoin pour résoudre le problème qui m’est posé ? Cette question est en fait liée à la précédente. En effet, si le choix des interfaces va déterminer la complexité du code, il va aussi peser lourdement sur le choix du micro-contrôleur. (Puissance, mémoire, nombre d’ I/O)

Un petit exemple pour illustrer cela : si dans votre application vous avez besoin de contrôler 16 leds, vous pouvez utiliser directement 16 sorties de votre micro-contrôleur et garder un code très simple, ou utiliser deux circuits de conversion série-parallèle qui n’utiliseront que 3 sorties du même micro-contrôleur mais qui rendra votre code plus complexe, (et aura donc besoin de moins de sorties mais de plus de mémoire)

Enfin, la dernière question à se poser est : le système définit est-il rentable ? En effet, si vous devez mettre en oeuvre un système à micro-contrôleur coûtant 30 Euros, pour réaliser une minuterie, alors qu’un simple circuit 555 coûtant 15 Euros peut réaliser la même fonction, ou pire, que cette minuterie "Made In Taiwan" est vendue toute faite au prix de 10 Euros, votre choix sera vite fait. (Il n’est pas nécessaire de réinventer le monde à chaque fois.)

Qu’est-ce que Programmer ?

Etape n° 2 : La conception de la solution

Le problème est maintenant défini, l’arrière-plan est établi. De nouveau, vous allez être tenté de commencer à écrire les programmes. Pourtant, ce n’est pas encore le moment : avant d’écrire quoi que ce soit, il faut concevoir la manière de résoudre le problème.

C’est en fait lors de cette étape que vous allez utiliser la programmation structurée pour écrire vos différents algorithmes. En fait, cette étape se subdivise en quatre sous-étapes : le traitement par modules, l’écriture des algorithmes, l’optimisation et la recherche d’erreurs.

Le traitement par modules :

Sauf si le problème à résoudre est trivial, sa solution comprendra plusieurs programmes s’appelant l’un l’autre pour former ensemble une séquence complexe qui accomplira la tâche demandée. Nous allons donc diviser l’application en modules élémentaires. Chaque module effectuera une tâche précise et sera suffisamment court pour être compris dans son entièreté en une fois.

L’écriture du module en programmation structurée nous aide à expliciter la tâche du module, et c’est en raffinant cette écriture que vous vous approcherez petit à petit de l’étape de l’écriture du programme : l’encodage.

Raffiner l’écriture, c’est réécrire successivement le module à un niveau de détail chaque fois plus élevé.

Le premier stade sera :

```
BEGIN
     <Fonction minuterie>
END
```

Il deviendra par la suite plus complexe :

```
BEGIN
     DO WHILE <Courant présent>
          IF <Touche appuyée>
               <Allumer la lampe>
               <Attendre 2 minutes>
               <Eteindre la lampe>
          ENDIF
     ENDDO
END
```

Cette manière de travailler permet de présenter le déroulement du processus à n’importe quel niveau de complexité. Elle permet la décomposition en fonctions indépendantes et est très utile pour la documentation du programme.

N’oubliez pas de maintenir une correspondance stricte entre les différentes écritures. Une liste détaillée doit expliciter exactement une ligne d’une liste moins détaillée. Sans cela votre travail sera tout à fait incohérent !

Qu’est-ce que Programmer ?

Etape n° 2 : La conception de la solution (suite)

Après avoir divisé le travail en étapes simples accomplies chacune par un module, vous pouvez constater que certains de ces modules ont une fonction tout à fait générale qui n’est pas liée à l’application en cours. Ces modules peuvent être utilisés dans d’autres programmes et vont servir à vous constituer une bibliothèque de modules qui seront réutilisés maintes et maintes fois.

Cette façon d’agir présente plusieurs avantages. Un module déjà utilisé fréquemment ne présente plus beaucoup de risques d’erreurs. Sa logique peut être considérée comme valable puisqu’elle a été bien souvent testée dans diverses applications. De plus, le temps d’écriture de ce module peut-être économisé, tout au plus nécessitera-t-il quelques adaptations.

L’écriture des algorithmes

Un des facteurs le plus important dans l’établissement de la solution est le choix de l’algorithme approprié. Un algorithme est une suite d’étapes clairement définies, décrivant l’action à réaliser.

L’algorithme spécifie les actions et les opérations d’un programme. Il peut y avoir une grande différence dans l’efficacité de plusieurs algorithmes aboutissant au même résultat, aussi le choix définitif d’un algorithme doit-il être soigneusement pesé.

Pour bien construire un algorithme, il faut d’abord bien définir les relations entre les paramètres du problème. Ensuite, concevoir l’algorithme en travaillant de haut en bas, c’est-à-dire commencer par la structure de contrôle principal et ensuite descendre vers les niveaux inférieurs en détaillant un peu chaque fois les opérations. Une première description ne devrait pas prendre plus d’une dizaine d’étapes-clés. Ensuite, chacune de ces étapes est raffinée jusqu’à arriver, par paliers, à décrire complètement le processus. Dans ce raffinement, recherchez toujours le traitement le plus simple. Et ne vous évertuez pas à réinventer ce qui a été mainte fois écrit. Beaucoup d’algorithmes efficients existent sur le marché et cela vaut souvent la peine de chercher dans la littérature existante, la solution à votre problème précis. (Internet est à ce sujet une manne quasi infinie d’inspiration)

Une fois l’algorithme établi et traduit par une liste de commandes de programmation structurée, il vous faut tester la logique de l’ensemble. Pour cela, reprenez votre liste depuis le début, et promenez-vous le long de tous les blocs sans réfléchir à ce qui devrait s’y accomplir, mais en exécutant chaque étape exactement comme elle se présente. Notez les valeurs que l’algorithme utilise et modifie. Si une étape du traitement demande de diminuer une valeur d’une unité, faites-le scrupuleusement même si la valeur précédente est déjà erronément négative. En effet, une des caractéristiques du micro-contrôleur est de faire exactement ce qu’il reçoit comme instruction, même si cela n’a aucun sens.

Cette étape finale de "ballade" à travers l’algorithme vous aidera à trouver les lacunes de logique. C’est une étape parmi les plus ennuyeuses, mais elle est vitale et c’est à vos risques et périls que vous l’esquiverez.

La structure modulaire et la conception de haut en bas sont deux éléments de la programmation structurée que nous traiterons en détail dans le chapitre suivant.

Qu’est-ce que Programmer ?

Etape n° 2 : La conception de la solution (suite)

L’optimisation

L’optimisation de votre programme est en fait très fortement liée à votre algorithmie et au matériel que vous aurez choisi. Nous avons dit précédemment que pour une tâche précise, il pouvait y avoir plusieurs algorithmes qui l’exécuteraient de manière identique. C’est lors de l’optimisation que vous allez décider exactement quel matériel vous allez utiliser, et quel algorithme sera le plus efficace en fonction du matériel choisi.

Le premier critère d’optimisation d’un algorithme, sera un critère de vitesse. En effet, si vous voulez réaliser un chronomètre, et que votre algorithme met une seconde et demi pour afficher le temps suivant (sur le matériel choisi) vous vous rendez compte tout de suite du problème. Il vous faudra alors soit aménager votre algorithme pour l’accélérer, soit définir ou choisir un autre algorithme, ou soit adapter le matériel pour vous permettre de réaliser la même tâche plus rapidement.

Un autre critère important qui influencera votre algorithme, est le critère financier. Les micro-contrôleurs rapides sont très coûteux, et vous devrez parfois utiliser des circuits plus lents, et donc adapter votre algorithme pour que les résultats soient acceptables.

La recherche d’erreurs.

Lors de la "ballade" à travers l’algorithme, vous aurez vraisemblablement trouvé un certain nombre d’erreurs, dus à votre algorithme, et que vous devriez avoir corrigé.

Mais le plus dur, est de projeter des erreurs "improbables" voire "impossible". Il vous faudra alors imaginer les cas les plus farfelus qui pourraient "planter" votre système. Puis vous pourrez ébaucher des solutions logicielles, et parfois même vous pourrez concevoir des solutions matérielles tel qu’un "Watch-dog".

Une fois toutes ces étapes terminées, vous vous retrouver avec un schéma qui est définitif, et les algorithmes écrits en programmation structurée qui s’y rapportent. Il est important d'insister sur le fait que votre électronique doit être complètement définie à ce stade. Vous pourrez alors enfin passer à l’étape suivante qui est la programmation de votre circuit. Un changement de l’électronique ensuite, vous obligera à recommencer depuis le début l’étape de conception de la solution.

Qu’est-ce que Programmer ?

Etape n° 3 : Ecriture du programme

Vous voici enfin au stade que vous attendez ! Maintenant, vous allez enfin programmer, pensez-vous ? En fait, vous programmiez depuis le début, ne l’oubliez pas. Vous arrivez à l’étape de la production des suites d’instructions qui réaliseront le traitement que vous avez soigneusement défini et conçu.

A votre disposition vous avez les spécifications du système, la description technique des entrées et des sorties, les listes écrites en programmation structurée des différents modules qui composent votre programme. Vos notes reprennent les souhaits et les remarques des utilisateurs du système. Avec tout cela, par où commencer ?

La réponse ne vous surprendra pas : Commencez par le sommet. Comme lors de la conception du système, il faut d’abord encoder le "programme principal" (qui forme le "squelette", la structure de l’application), c’est en général lui qui contrôle le déroulement de toute l’application.

Essayer d’écrire cette structure de la manière la plus complète possible, et dès le départ, définissez une routine "vide" vers laquelle pointera tous les "gosub". Puis c’est au fur et à mesure de l’écriture des différents modules, que vous remplacerez dans les "gosub", l’étiquette qui pointait vers la routine "vide" par le label qui pointera vers la routine définitive. Cette manière d’utiliser une structure "vide" permet de pouvoir tester votre application très tôt dans le processus d’écriture. Grâce à l’utilisation d’une électronique de test à base de mémoire de type flash ou Eeprom, vous pourrez tester votre application à chaque ajout d’une routine dans votre programme.

Si votre application a été bien conçue, il vous est théoriquement possible d’écrire les différentes routines de manière complètement autonome. Vous pouvez alors la tester seule sur votre électronique finale, puis ensuite, l’ajouter dans le corps de votre application principale en remplacement d’une routine "vide".

Lors de l’écriture des différentes routines, pensez à consulter les sources de programmes à votre disposition. Dans ce cours sur les micro-contrôleurs, et en annexe, vous trouverez un ensemble de petites routines qui vous permettront d’accomplir un nombre très diversifié de tâches. N’hésitez pas à vous en servir.

Dans la mesure du possible, lors de l’écriture du programme, essayez d’utiliser des noms de variables qui soient "chantant". Ceci n’est pas très important pour le micro-contrôleur, mais lorsque vous relirez votre programme l’expression "OUTPUT Relais3" sera plus lisible que l’expression "OUTPUT I"

Qu’est-ce que Programmer ?

Etape n° 3 : Ecriture du programme (suite)

Les commentaires.

Enfin, pour clôturer cette partie, je voudrais insister sur l’importance des commentaires utilisés dans le programme.

Prenez l’habitude de commencer tous vos programmes par leur nom et leur date de création. Pendant la période de mise au point, ajoutez à chaque modification la date du jour et éventuellement la version. Ensuite, décrivez en une ligne ou deux le but du programme, et éventuellement l’électronique nécessaire à son fonctionnement, et la manière de la raccorder. Dès que votre bibliothèque de programmes contiendra plusieurs modules, ce sera indispensable pour les identifier facilement. Constituez-vous un répertoire de modules, reprenant le nom du programme ainsi que ces commentaires décrivant leur but et gardez toujours ce répertoire à portée de main.

De même, dans le cours de l’encodage, insérez des lignes de commentaires, non pour traduire en français les instructions du micro-contrôleur, mais plutôt pour expliciter le but du traitement. Puisque vous avez découpé le traitement en étapes simples dans votre liste structurée, reprenez ce sectionnement pour insérer vos commentaires entre chaque bloc logique.

Une bonne façon de voir si vos commentaires sont efficaces est de relire le programme quelques semaines après l’avoir écrit. Si vous le comprenez encore sans devoir vous creuser la tête, il était probablement bien commenté. Sinon, essayez d’y ajouter ce qui en facilitera la compréhension ultérieure, le temps consacré à ce travail sera regagné lors du réemploi ou de la maintenance.

Un dernier point : n’oubliez pas de mettre à jour les commentaires lorsque vous modifiez le programme !

Qu’est-ce que Programmer ?

Etape n° 4 : Documentation

Vous avez remarqué que régulièrement, à la fin d'un paragraphe, nous insistons sur la documentation. C'est en effet un élément très important de la mise au point d'une application et c'est pourtant l'aspect le plus négligé.

Sans documentation, l'utilisateur est perdu dès qu'il a affaire à une procédure inhabituelle. Sans documentation, votre bibliothèque de programmes ne sera guère utilisable.

La documentation d'une application peut se diviser en deux catégories. La première est constituée des commentaires inclus dans chaque programme. Grâce à ceux-ci, la structure d'un programme restera claire, même des mois après sa conception.

La deuxième catégorie reprend les directives écrites, expliquant la façon de travailler du programme et son utilisation.

Mode d'emploi

Ne croyez surtout pas que la documentation soit le dernier stade de la mise au point d'une application. Elle en est une part intégrante qui doit commencer en même temps que la définition du problème. Cette façon d'agir apporte aussi d'autres avantages. Rédiger clairement le problème qui est posé, la solution envisagée et la liste des produits à obtenir, permet de disposer d'une base de contrat à proposer à l'utilisateur. Ce document servira aussi au développement et à l'encodage et constituera le départ du mode d'emploi final.

Dans la durée de vie d'une application, le mode d'emploi sert essentiellement à deux catégories de personnes : ceux qui utilisent le programme et ceux qui se chargent de sa maintenance. Les besoins de ces deux catégories ne sont pas les mêmes, aussi le mode d'emploi doit-il se diviser en deux parties.

Le manuel de l'utilisateur contiendra les informations non techniques. Il exposera la mise en route du système, et l'explication de son utilisation, les différentes visualisations et signalisations et éventuellement les différents codes d'erreurs. Il donnera les directives à suivre en cas de problèmes. Il expliquera à l'aide d'exemples un maximum de cas concrets. N'oubliez pas l'index pour que l'utilisateur s'y retrouve facilement.

Quant au manuel technique qui est conçu pour des techniciens, i1 inclura les descriptions du système et de ses interfaces, les différentes mesures à effectuer. Les états des entrées et des sorties et une description plus détaillée des éventuelles astuces utilisées. Les listes des programmes ("listings") ne sont pas intéressantes à inclure dans le mode d'emploi car elles prennent beaucoup de place et devraient être remises à jour à chaque modification, même superficielle. De plus, leurs présences risqueraient de voiler l'importance des autres informations. Par contre, un chapitre très important est la description de toutes les modifications apportées, une sorte de "journal de la maintenance" qui évoluera au fil du temps avec l'application.

Qu’est-ce que Programmer ?

Etape n° 4 : Documentation

Documentation du programme

Le programme lui-même est documenté de plusieurs façons. Il y a les commentaires directs, en en-tête pour donner les principales informations concernant la fonction du programme, et, plus loin dans le texte, pour éclaircir la structure, les principales boucles et les branchements importants.

Ecrivez-les au fur et à mesure de Votre encodage, quand le module et son déroulement sont bien présents à votre esprit. Vous pourrez alors décrire facilement le mode de fonctionnement.

Un en-tête significatif comprendra le nom du programme, le nom du programmeur, la date d'écriture et la date de la dernière mise à jour, une courte description du but et du fonctionnement, les informations nécessaires pour le bon déroulement et la maintenance du programme. Suivant les cas, il pourrait aussi contenir les noms des variables locales, des paramètres passés depuis un autre fichier de commandes, et la liste des programmes appelés. L'habitude de terminer le programme par une ligne de commentaires permet de remarquer immédiatement si une portion du fichier de commandes a été perdue.

Votre programme est aussi commenté de façon indirecte. L'indentation correcte des branchements et des boucles vous rend la structure visible au premier regard. De même, l'usage de noms significatifs, pour les variables et les routines, permet de mieux comprendre ce qui s'exécute.

La formule MOYENNE = SOMME/NOMBRE n'a pas besoin d'explication alors que A = B/C n'est pas très explicite

Si vous avez lu toutes les recommandations qui précèdent, vous pensez probablement que c'est beaucoup trop élaboré pour la plupart des petites applications. Et bien, détrompez-vous ! Il vous faudra pas mal de discipline pour obtenir une bonne documentation, mais vous en serez toujours récompensé. Pour vous en convaincre, essayez de remettre la main sur un programme que vous avez écrit voici plusieurs mois ou, si vous débutez en programmation, prenez un programme au hasard dans un livre et, tentez d'en comprendre le fonctionnement tout en calculant le temps nécessaire. S'il est documenté, vous constatez bien que c'est grâce à cela que vous arrivez à vous y retrouver.

L'essentiel est de s'habituer à commenter pour que cela devienne un réflexe dans votre méthode de travail.

Commentez vos programmes et utilisez la structure modulaire et la réalisation de haut en bas, même pour des applications très simples. Ces techniques doivent vous devenir naturelles et faire partie de votre style de programmation.

Qu’est-ce que Programmer ?

Etape n° 5 : Vérification et recherche des erreurs.

La vérification d'un programme, ainsi d'ailleurs que la documentation, ne sont pas des étapes logiques de la mise au point d'une application comme le sont l'analyse du problème, la conception de la solution et l'encodage des programmes. Ces trois étapes-là se suivent et la réalisation de chacune découle de la précédente.

Au contraire, la vérification et la documentation sont des méthodes de travail plutôt que des stades du processus. Chaque fois que vous parcourez un organigramme, vous vérifiez votre programme. Quand vous essayez les différentes sélections de votre menu principal, vous vérifiez votre application.

La vérification vise principalement 3 objectifs
1 - éliminer les erreurs,
2 - vérifier l'accord avec les spécifications,
3 - améliorer la vitesse.

La recherche d'erreurs prend un temps considérable très souvent sous-estimé. Les tests successifs de l'application servent à détecter les erreurs. Malheureusement il est presque impossible de prouver le bon fonctionnement d'une application par les tests. Le nombre de combinaisons des différentes options et des données possibles croît de façon exponentielle avec la quantité de branchements, si bien qu'on ne peut les essayer toutes.

Le choix des tests à accomplir est donc primordial. Ainsi, si on a contrôlé qu'une zone est numérique, a-t-on pensé à tester ou interdire les valeurs fractionnaires ou négatives ?

Il est préférable que les tests soient effectués par d'autres personnes que le programmeur. Celui-ci est généralement trop proche de son travail pour être à même de repérer les erreurs. De plus, il connaît le fonctionnement du programme et s'attend à ses réactions ; comme tout être humain, il aura tendance, même inconsciemment, à influencer ses tests en fonction de cela, Une personne étrangère au processus de mise au point de l'application sera "vierge" de toute information et testera plus facilement dans toutes les directions.

Voici quelques règles pour mener à bien la vérification

- Testez par module. Prenez-les un par un, indépendamment de l'application globale. Fournissez des entrées au module et vérifiez que le résultat produit est correct en essayant tous les cheminements possibles.

- Testez les frontières. Dans chaque module, pour tous les cas où un domaine est prévu pour les entrées, voyez ce qui se passe avec les limites du domaine de validité. S'il s'agit d'une entrée analogique, testez les entrées minimales et maximales, voire négative. Pour une entrée digitale, jouez avec la tension d'alimentation, imaginez l'influence du monde extérieur sur votre application.

- Testez les "erreurs". Nous voulons dire par-là, introduisez volontairement des erreurs pour voir comment le système réagit et comment les erreurs sont traitées. Sont-elles validées, les indications d'erreurs sont-elles correctes ? Le programme peut-il continuer à se dérouler après rectification.

Qu’est-ce que Programmer ?

Etape n° 5 : Vérification et recherche des erreurs. (suite)

Ces diverses vérifications seront effectuées par le programmeur. Ensuite celui-ci mettra au point un plan de test à réaliser par un collègue sur l'application entière.

Tous ces tests nous ont permis de déceler certaines erreurs. Il nous reste encore à les localiser exactement pour pouvoir les corriger.

Différents types d'erreurs se rencontrent couramment

Les erreurs s'introduisent à tous les stades de l'élaboration d'une application. Quand elles proviennent de l'analyse et de la conception de la solution, les corriger tardivement coûte assez cher, car elles requièrent souvent une remise au point et une réécriture de modules importants.

La cause de ces erreurs est généralement un manque de communication et d'échanges d'informations entre les "demandeurs" et les "concepteurs", aussi soyez très prudents quand vous travaillez pour quelqu'un d'autre. Attachez beaucoup d'importance à l'analyse du problème. Dans la mesure du possible, faites un prototype et soumettez-le à l'utilisateur. Cela l'aidera à cerner ce qu'il souhaite. Dans la phase de mise au point de la solution, ne vous hâtez pas pour arriver plus vite à l'écriture proprement dite. Si vous vous trompez à ce stade, vous produirez des modules incorrects qu'il faudra recommencer et réécrire depuis le début.

Au moment de l'encodage, vous risquez de commettre des erreurs de syntaxe, que l'interpréteur vous aidera à appréhender et des erreurs de logique qui peuvent aussi provenir de l'étape de mise au point. Vérifiez bien les critères de branchement et les critères de boucle, ainsi que les modes d'incrémentations.

Ceci termine notre survol des bases d'une bonne programmation, visant à optimiser la maintenance et à en minimiser le coût. Nous avons divisé le processus en cinq étapes qui s'interpénètrent. Le respect de cette procédure vous aidera à produire un travail de qualité. Et avec le temps qui passe, vous apprécierez de plus en plus cette méthode de travail appelée "Programmation Structurée".

# La Programmation Structurée

La programmation structurée a pour but de définir un "algorithme" dans un langage universel qui pourra aisément être retranscrit dans n’importe quel langage et pour n’importe quel micro-contrôleur.

Tous s’accordent à reconnaître dans cette manière de procéder une des meilleures méthodes de programmation, mais très peu l’utilisent réellement. La programmation structurée est simplement un ensemble de techniques qui rend le processus de programmation plus facile à accomplir. Dans ce cas, pourquoi n'est-il pas plus utilisé, direz-vous ?

Tout d’abord, la programmation structurée peut-être un travail ardu : elle requiert une préparation plus attentive que les autres techniques. Elle demande de la discipline. Il est plus simple de s’asseoir devant le clavier et de commencer à taper quelques lignes de programme, que de traverser tout le processus de la définition, de la conception, de la mise au point, des enquêtes auprès des utilisateurs. Il est plus facile d’esquisser des organigrammes et de concevoir dans un premier jet des systèmes de manière non structurée.

Mais un programme qui travaille mal (à condition qu’il fonctionne) est souvent le résultat d’un encodage imprévoyant et doit alors être corrigé ou réécrit au prix d’une perte de temps et d’argent. Il en va ainsi des projets bâclés qui engendrent des organigrammes et des programmes confus, difficiles à lire et à comprendre et souvent plus compliqués à modifier qu’à réécrire.

L’origine de la programmation structurée remonte aux années 66-68, quand il a été démontré par E.Dijkstra, G. Galopini et C. Bohm, que la qualité des programmes diminuaient avec le nombre de GOTO utilisés et que n’importe quel système pouvait être écrit sans instruction GOTO, en utilisant trois structures de programmation.

L’instruction GOTO qui est présente dans de nombreux langages, a pour effet un branchement inconditionnel à une instruction située ailleurs. Ce branchement sans restriction donne un style décousu. Un organigramme non structuré peut évoluer à ce point que ses flèches s’entrecroisent en tout sens et le font ressembler à un plat de spaghetti ! Dans un pareil cas, nous ne vous souhaitons pas devoir retracer la logique du programme, ne fût-ce que quelques jours plus tard !

Règles de la programmation structurée :

1 ) Un programme est toujours faux.

2 ) Un programme est une structure de base qui est divisée en modules (les sous-routines), et chacun des modules ne doit effectuer qu'une seule fonction.

3) Le programme, comme chacun de ses modules, ne contient qu'un seul point d'entrée et un seul point de sortie.

4) Les programmes n'utilisent que trois structures : les enchaînements, les choix et les boucles.

5 ) Un programme se lit toujours du haut vers le bas.

Règle n°1 : Un programme est toujours faux.

Cette affirmation peut faire sourire, et pourtant. Lors de l'écriture d'un programme, il est quasiment impossible au concepteur de penser à tout, de tout prévoir, de tout planifier. Il existe toujours des cas que le programmeur à omis par négligence, par oubli, ou consciemment en pensant que ce cas de figure était "impossible". Lors de l'écriture de votre programme ayez toujours humblement en tête cette affirmation, et pensez aux factures astronomiques que certains clients reçoivent de leurs fournisseurs, qui se justifient en disant que "c'est la faute à l'informatique".

Règle n°2 : La division en modules

Lors du cheminement qui vous amènera à la réalisation de votre programme, vous préparerez les objectifs du programme, les sources d'information qu'il utilisera, les algorithmes de conversion de données, les valeurs intermédiaires nécessaires et les résultats prévus. En suivant ce cheminement vous êtes déjà sur la voie de la programmation structurée. Il vous reste à concevoir vos organigrammes d'une façon simple et structurée. Vous disposez alors d'une bibliothèque de modules d'intérêts généraux qui pourront vous resservir au besoin dans d'autres applications. Vous voyez l'intérêt d'un module simple qui accomplit une tâche unique, clairement définie. Les modules seront courts, normalement pas plus de deux pages de codes structurés.

Règle n°3 : Une seule entrée, une seule sortie

Comme nous venons de le dire, les modules ne doivent contenir qu'un seul point d'entrée et un seul point de sortie. Bien sûr, ils disposeront de points de décision, des endroits où l'action peut prendre une direction parmi plusieurs. Mais ces branchements seront entièrement contenus dans le module (si c'est un choix entre plusieurs traitements) ou formeront les jonctions avec des modules de niveaux inférieurs.

Sont à proscrire les modules accessibles par plusieurs entrées ou, encore pire, avec plusieurs points de branchements vers d'autres modules. Retenez : une seule entrée, une seule sortie.

Lors de l'utilisation des instructions de programmation structurée, ce principe ne pose pas de problèmes au niveau de l'entrée : une routine appelée ne peut commencer que par la première ligne. Mais il faut veiller à ce que la possibilité de sortie soit unique.

Règles de la programmation structurée :

Règle n°4 : Les trois structures de base

N’oubliez pas que ces trois structures de base (l’enchaînement, la décision et boucle) permettent de décrire n’importe quelle structure logique aussi complexe soit-elle. Pour décrire les trois structures de base, nous allons utiliser les instructions de la programmation structurée. Ces instructions seront détaillées et commentées par la suite.

A ) La structure d’enchaînement

```
BEGIN
     <Traitement A>
     <Traitement B>
     <Traitement C>
END
```

Ces traitements représentent, par exemple, l’introduction d’une donnée, l’extraction d’une racine, la lecture d’un clavier, l'enclenchement d’un relais, l’allumage d’une lampe ou n’importe quelle tâche simple. Vous remarquerez : une entrée, une sortie.

B ) La structure de décision

```
BEGIN
     IF <condition> THEN
          <Traitement A>
     ELSE
          <Traitement B>
     ENDIF
     <Traitement C>
END
```

Elle comporte aussi un seul point d’entrée et un seul point de sortie, mais elle est un peu plus complexe. Le <Traitement A> sera réalisé si la <condition> est vraie, alors que le <Traitement B> est réalisé si la <condition> est fausse. Par contre le <Traitement C> sera toujours effectué. Par extension, il est possible de concevoir une structure à décision multiple ayant la forme suivante :

```
BEGIN
     IF <condition 1> THEN
          <Traitement 1>
     ELSE
          IF <condition 2> THEN
               <Traitement 2>
          ELSE
               ....
               IF <condition n> THEN
                    <Traitement n>
               ELSE
                    <Traitement n+1>
               ENDIF
          ENDIF
     ENDIF
END
```

Règles de la programmation structurée :

Règle n°4 : Les trois structures de base (suite)

B ) La structure de décision (suite)

Dans cette structure, le <Traitement 1> est effectué si la <condition 1> est rencontrée, le <Traitement 2> est effectué si la <condition 2> est rencontrée..... et le <Traitement n> est effectué si la <condition n> est rencontrée. Si aucune des conditions n'est rencontrée, c'est le <Traitement n+1> qui sera accompli. Notez que dans cette structure, chaque traitement n'est effectué qu'une seule fois.
Une autre structure assimilée à décision multiple est la structure ON GOSUB.

```
BEGIN
     <Traitement A>
     ON <variable> GOSUB <routine0>,<routine1>,<routine2>,...
END

SUB <routine0>
     <Traitement B>
ENDSUB

SUB <routine1>
     <Traitement C>
ENDSUB

SUB <routine2>
     <Traitement D>
ENDSUB

etc...
```

Ici, il y a une évaluation numérique de l'expression <variable>. Si elle est égale à zéro, la <routine0> est lancée, si elle est égale à 1 c'est la <routine1>, si elle est égale à 2 c'est la <routine2>, et ainsi de suite... Si la valeur de la variable dépasse le nombre de GOSUB, alors, aucune routine n'est lancée et le programme se termine. Ici aussi, chaque traitement n'est effectué qu'une seule fois. Mais la différence avec la structure précédente, c'est qu'ici, si la variable a une valeur qui ne correspond pas, aucun traitement n'est effectué.

Règles de la programmation structurée :

Règle n°4 : Les trois structures de base (suite)

C ) La structure de boucle.

Cette troisième et dernière structure peut se diviser en deux parties. La boucle à décision préalable, et la boucle à décision postérieure.

a ) Boucle à décision préalable :

```
BEGIN
     DO WHILE <condition>
          <Traitement A>
     ENDDO
END
```

Dans cette boucle, le <Traitement A> est effectué tant que la <condition> est vraie. Notez que comme le test se fait au début de la boucle, si la condition est fausse dès le début, le <Traitement A> ne sera même pas effectué une seule fois. Faites attention en définissant ce genre de structure que lors du <Traitement A>, l'élément utilisé lors du test de <condition> soit modifié, sinon, vous risquez de tomber dans une boucle infinie.

b ) La boucle a décision postérieure :

```
BEGIN
     DO
          <Traitement A>
     UNTIL <condition>
END
```

Contrairement au cas précédent, ici, le <Traitement A> est toujours effectué au moins une fois, puisque la décision de bouclage est prise après le <Traitement A>. Notez aussi, qu'ici, la boucle est inversée, c'est-à-dire qu'elle se réalise tant que la <condition> est fausse, et la sortie de boucle se fait lorsque la <condition> est vraie. Ici aussi, il faut veiller, lors de la conception de la boucle, à ne pas tomber dans un cas ou la boucle est infinie.

Avec la combinaison de ces trois structures, vous pouvez concevoir n'importe quel système. Grâce à cela, vos programmes seront plus faciles à relire. Vous pourrez sans problème les modifier au gré de l'évolution des conditions ou les inclure dans d'autres programmes futurs.

Votre productivité aussi s'améliorera en fonction de la modularité et de la simplicité de vos programmes, puisque vous en viendrez à écrire plus de modules dans le même laps de temps et sans travailler plus.

Une remarque encore : au début, la conception structurée semble parfois donner des systèmes peu maniables, combinaisons multiples de nombreux modules simples, là où un seul, plus grand et plus complexe, aurait pu accomplir le travail. Cela peut simplement signifier que vous n'avez pas étudié le problème à fond. La complexité et la confusion de votre organigramme reflètent alors la complexité et la confusion de vos pensées sur la question.

Règles de la programmation structurée :

Règle n°4 : Les trois structures de base (suite)

Parfois aussi, il est vrai, le travail pourrait se faire à l'aide d'un seul programme au lieu d'une combinaison de modules structurés. Mais cela entraîne deux conséquences : ce seul programme sera plus complexe ; si un problème se pose, il sera d'autant plus difficile à localiser. En outre, ce programme complexe restera probablement artisanal, c'est-à-dire qu'il ne pourra pas figurer en bonne place dans votre bibliothèque de routines standards. Autrement dit, la prochaine fois qu'un système devra accomplir une tâche similaire, il faudra reprendre tout le travail depuis le début.

Règle n°5 : La conception de haut en bas

Peut-être ne voyez-vous pas bien ce que recouvre le terme de "conception de haut en bas" ? Cela signifie simplement que nous allons débuter le travail par le haut, c'est-à-dire par le programme principal, celui qui contrôle le déroulement de tout le processus ; ensuite, passer à la description du niveau inférieur et ainsi de suite en descendant les échelons jusqu'à avoir détaillé les traitements les plus simples de niveaux inférieurs.

Dans ce contexte, le programmeur est amené à envisager le problème de manière la plus large possible. Ce n'est pas nécessairement intuitif. Les programmeurs ont souvent tendance à commencer par le bas, en invoquant de mauvaises raisons : " Pour gagner du temps ! " ou " Si le résultat est correct, c'est que le programme est bon. " Une approche de bas en haut aboutira en un tas de modules que l'on va chercher à relier entre eux tant bien que mal.

Au contraire, la conception de haut en bas va établir une hiérarchie entre tous les modules. Créer une hiérarchie est très important dans la description du système. Elle décrit le contrôle du système en ordonnant les différentes parties. Un module est appelé par un module au-dessus de lui et peut appeler ceux qui sont en dessous. Un module renvoie le contrôle à celui qui l'a appelé dès qu'il a achevé sa tâche.

Spécifier clairement ce cheminement entre les modules en limitant les passages trop diversifiés permet de réduire les interactions entre les modules. De la sorte, une modification d'un module n'affectera que celui qui est directement au-dessus de lui. Si tous les modules peuvent s'appeler l'un l'autre, les erreurs seront bien difficiles à cerner et une seule modification pourra entraîner une réaction inattendue de plusieurs modules apparemment non concernés. Exactement comme dans un plat de spaghetti, si vous tirez un spaghetti d'un côté, cela peut faire bouger l'autre bout du plat. N'en déduisez tout de même pas que les adeptes de la programmation structurée sont allergiques aux spaghettis !

La conception de haut en bas présente de nombreux avantages : Elle encourage et même elle requiert l'organisation dès le début de l'effort de réflexion vers le but exact de l'application et vers la façon de l'atteindre. Elle vous force à mettre d'abord au point les grandes lignes de votre système, le nud de contrôle de tout l'ensemble, le programme qui sera le plus souvent en opération. Réalisé le premier, il sera aussi le mieux testé. Et vous pourrez effectuer tous les changements logiques au système avant d'avoir gaspillé des heures à mettre au point des routines qui ne figureront peut-être plus dans le travail final.

Liste des instructions de programmation structurée

Conventions

Nous allons maintenant décrire les quelques instructions nécessaires à l’écriture de la programmation structurée. Pour cela, nous allons imposer quelques conventions :

- Les instructions seront toujours écrites en majuscule.

- Les noms de variables et les labels seront eux définis en minuscule.

Instructions de remarque dans le programme

REM ou '

Pour différencier les remarques et la documentation du reste des instructions, elles seront toujours précédées de l'instruction "REM" ou d'une apostrophe.

L’ensemble des remarques constitue la documentation du programme. Cette documentation doit être la plus abondante possible. Nous en avons suffisamment parlé auparavant.

Liste des instructions de programmation structurée

Instructions de commande du programme

BEGIN

Déclare le début du programme, c’est toujours la première instruction d’un programme

END

Termine l’exécution d’un programme. C’est toujours la dernière instruction d’un programme.

PAUSE

Suspend l’exécution d’un programme pendant un certain temps définit après le mot PAUSE

Exemple : PAUSE 1S ’Arrête l’exécution du programme pendant 1 seconde. PAUSE 10mS ’Arrête l’exécution du programme pendant 10 millisecondes.

Liste des instructions de programmation structurée

Instruction de déclaration de variables et constantes

DECLARE

Cette commande permet de définir les "variables" et les "constantes" et les "tableaux" Les variables sont des zones mémoires qui changent d'état pendant l'exécution du programme, alors que les constantes gardent toujours la même valeur. Les tableaux sont des variables ou constantes avec plusieurs éléments. Ils sont définis comme une variable avec un nombre entre parenthèses (l'index) qui défini le nombre de variables contenues dans le tableau. L'index peut-être une autre variable.

Les différents types de valeurs sont :

- INTEGER : valeurs entières varient de -128 à 127 ou -32768 à 32767
- FLOATING : valeurs définies en virgule flottante
- ALPHA : valeurs alphanumériques
- BIT : un bit égal à zéro ou un
- NIBBLE : 4 bits (varie de 0 à 15)
- BYTE : 8 bits (varie de 0 à 255)
- WORD : 16 bits (varie de 0 à 65535)

Par convention, le suffixe $ peut-être utilisé pour les chaînes de caractère et le suffixe % pour les variables entières. L'explication

Format courant de l'instruction :

```
DECLARE <Lieu>-<Nom> AS <Type>

Exemples :
DECLARE global-i% AS INTEGER       'Variable "i%" définie en valeur entière
DECLARE global-message$ AS ALPHA   'Variable "message" définie comme chaîne
                                   'de caractère.
DECLARE b AS BYTE                  'Variable "b" définie comme un octet.
DECLARE drapeau AS BIT             'Variable "Drapeau" définie comme un bit
                                   '(0 ou 1).
DECLARE liste(9) AS BYTE           'Variable "liste" définie comme un
                                   'tableau de 10 octets.
                                   'Les différents éléments du tableau sont
                                   'Liste(0),Liste(1)..Liste(8) et Liste(9)
```

Les variables BIT, NIBBLE, BYTE peuvent être définies par rapport à un élément supérieur. On utilise alors un qualifiant qui va préciser quel élément est défini dans la variable de départ. On peut donc définir une variable comme étant le Xiéme "BIT" d'un "BYTE" ou d'un "WORD", ou comme le "BYTE" de poids faible (LSB) ou de poids fort (MSB) d'un "WORD" ou encore comme "NIBBLE" de poids faible (LSB) ou de poids fort (MSB) d'un "BYTE".

Liste des instructions de programmation structurée

Instruction de déclaration de variables et constantes

DECLARE (suite)

Pour plus de clarté, nous allons subdiviser une variable "WORD" dans ses différents éléments, et montrer les différentes manières de déclarer les variables attenantes.

Considérons une variable "WORD" qui est composé de 16 "Bits" :

| Variable | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LSB-BYTE-WORD | | | | | | | | | X | X | X | X | X | X | X | X |
| MSB-BYTE-WORD | X | X | X | X | X | X | X | X | | | | | | | | |
| BYTE0-WORD | | | | | | | | | X | X | X | X | X | X | X | X |
| BYTE1-WORD | X | X | X | X | X | X | X | X | | | | | | | | |
| LSB-NIBBLE-LSB-BYTE-WORD | | | | | | | | | | | | | X | X | X | X |
| MSB-NIBBLE-LSB-BYTE-WORD | | | | | | | | | X | X | X | X | | | | |
| LSB-NIBBLE-MSB-BYTE-WORD | | | | | X | X | X | X | | | | | | | | |
| MSB-NIBBLE-MSB-BYTE-WORD | X | X | X | X | | | | | | | | | | | | |
| NIBBLE0-WORD | | | | | | | | | | | | | X | X | X | X |
| NIBBLE1-WORD | | | | | | | | | X | X | X | X | | | | |
| NIBBLE2-WORD | | | | | X | X | X | X | | | | | | | | |
| NIBBLE3-WORD | X | X | X | X | | | | | | | | | | | | |
| BIT0-WORD | | | | | | | | | | | | | | | | X |
| BIT1-WORD | | | | | | | | | | | | | | | X | |
| BIT2-WORD | | | | | | | | | | | | | | X | | |
| BIT3-WORD | | | | | | | | | | | | | X | | | |
| BIT4-WORD | | | | | | | | | | | | X | | | | |
| BIT5-WORD | | | | | | | | | | | X | | | | | |
| BIT6-WORD | | | | | | | | | | X | | | | | | |
| BIT7-WORD | | | | | | | | | X | | | | | | | |
| BIT8-WORD | | | | | | | | X | | | | | | | | |
| BIT9-WORD | | | | | | | X | | | | | | | | | |
| BIT10-WORD | | | | | | X | | | | | | | | | | |
| BIT11-WORD | | | | | X | | | | | | | | | | | |
| BIT12-WORD | | | | X | | | | | | | | | | | | |
| BIT13-WORD | | | X | | | | | | | | | | | | | |
| BIT14-WORD | | X | | | | | | | | | | | | | | |
| BIT15-WORD | X | | | | | | | | | | | | | | | |

Exemple :

```
DECLARE Sorties AS WORD            '- Définit la variable "Sorties" comme
                                   '  mot (16 bits).
DECLARE Drapeau AS BIT12-Sorties   '- Définit la variable "Drapeau" comme
                                   '  étant le 12ième bit de "Sorties".
DECLARE Relais AS NIBBLE2-Sorties  '- Définit la variable "Relais" comme
                                   '  étant le Nibble composé des bits 8 à
                                   '  11 de la variable "Sorties".
DECLARE Leds AS LSB-BYTE-Sorties   '- Définit la variable "Leds" comme
                                   '  étant l'octet composé des bits 0 à 7
                                   '  de la variable "Sorties".
```

Liste des instructions de programmation structurée

Instruction de déclaration de variables et constantes

DECLARE (suite)

De même, nous pouvons considérer une variable "BYTE" qui est composé de 8 "Bits". Elle sera subdivisée en différents éléments de la manière suivante:

```
Variable            7 6 5 4 3 2 1 0
BYTE                X X X X X X X X
LSB-NIBBLE-BYTE               X X X X
MSB-NIBBLE-BYTE     X X X X
NIBBLE0-BYTE                  X X X X
NIBBLE1-BYTE        X X X X
BIT0-WORD                           X
BIT1-WORD                         X
BIT2-WORD                       X
BIT3-WORD                     X
BIT4-WORD                   X
BIT5-WORD                 X
BIT6-WORD               X
BIT7-WORD           X
```

Exemple :

```
DECLARE Entrées AS BYTE                 '- Définit la variable "Entrées" comme
                                        '  un octet de 8 bits.
DECLARE Détecteur AS BIT3-Entrées       '- Définit la variable "Détecteur" comme
                                        '  étant le bit N° 3 d'"Entrées".
DECLARE Clavier AS NIBBLE1-Entrées      '- Définit la variable "Clavier" comme
                                        '  étant le Nibble composé des bits 4 à
                                        '  7 de la variable "Entrées".
DECLARE Alarme AS LSB-NIBBLE-Entrées    '- Définit la variable "Alarme" comme
                                        '  étant le Nibble composé des bits 0 à
                                        '  3 de la variable "Entrées".
```

Liste des instructions de programmation structurée

Instruction de définition des variables et constantes

LET

Après avoir déclaré les variables et les constantes, nous pouvons leur définir une valeur. Cette valeur sera donnée par l'instruction LET. L'instruction LET est aussi utilisée pour modifier une valeur.

Lorsque nous définissons ou modifions une valeur, nous utiliserons le préfixe "$" devant la valeur si la variable est utilisée en Hexadécimal, "%" devant la valeur si la variable est utilisée en binaire et sans préfixe si elle est utilisée en mode décimal classique. Notez bien qu'il s'agisse ici de préfixe qui s'utilise dans une valeur. A ne pas confondre avec le suffixe du nom de la variable qui définit une variable alphanumérique ou entière.

```
Exemple :
DECLARE Entrées AS BYTE                 '- Définit la variable "Entrées" comme
                                        '  un octet de 8 bits.
DECLARE Message AS ALPHA                '- Définit la variable "Message" comme
                                        '  variable alphanumérique.

LET Entrées = 41
LET Entrées = $29
LET Entrées = %00101001                 '- Ces trois lignes sont équivalentes.
                                        '  Elles donnent la valeur 41 en décimal
                                        '  (ce qui correspond à $29 en
                                        '  Hexadécimal et à %101001 en binaire)
                                        '  à la variable "Entrées".
LET Message = "Bonjour"                 '- La variable Message est initialisée
                                        '  avec le mot "Bonjour". Notez que le
                                        '  mot défini est mis entre guillemets.
```

Les Opérateurs

Lors de la définition ou la modification d'une variable, ou lors d'évaluation d'expression logique, nous pouvons utiliser un ou des opérateurs pour transformer la variable ou la combiner avec d'autres. Nous distinguerons quatre types d'opérateurs : les opérateurs numériques, les opérateurs binaires, les opérateurs alphanumériques et les opérateurs logiques.

Opérateurs numériques : Les opérateurs numériques sont des modifiants qui se rapportent aux nombres en général.

| Opérateur | Description |
|---|---|
| X + Y | Addition des nombres X et Y |
| X - Y | Soustraction des nombres X et Y |
| X * Y | Multiplication des nombres X et Y |
| X / Y | Division du nombre X par le nombre Y |
| X**Y | Nombre X élevé à la puissance Y |
| SQR(X) | Racine carrée de X |
| SIN(X) | Sinus de X |
| COS(X) | Cosinus de X |
| TAN(X) | Tangente de X |
| ABS(X) | Valeur absolue de X |

Liste des instructions de programmation structurée

Instruction de définition des variables et constantes

LET (suite)

Opérateurs binaires : Les opérateurs binaires se rapportent à la structure binaire des nombres.

| Opérateur | Description |
|---|---|
| X & Y | Combinaison ET bit à bit |
| X \| Y | Combinaison OU bit à bit |
| X ^ Y | Combinaison OU exclusif bit à bit |
| INV X | Inversion bit à bit du nombre X |
| >> X | Décalage à droite des bits du nombre X |
| << X | Décalage à gauche des bits du nombre X |

Voici le tableau des opérateurs pour les différentes tables de vérité.

| X | Y | X & Y | X \| Y | X ^ Y | INV X |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Opérateurs Alphanumériques : Les opérateurs alphanumériques modifient les variables alphanumériques.

| Opérateur | Description |
|---|---|
| LEFT$(X$,Y) | Extraction des Y caractères de gauche de la variable X$ |
| RIGHT$(X$,Y) | Extraction des Y caractères de droite de la variable X$ |
| MID$(X$,Y,Z) | Extraction des Z caractères à droite du Yème dans X$ |
| ASC(X$) | Extraction du code ASCII du 1er caractère de X$ |
| CHR$(X) | Donne le caractère dont le code ASCII est X |

Opérateurs logiques : Les opérateurs logiques sont utilisés lors de l’évaluation d’expression logique ou de conditions lors d’instruction de test.

| Opérateur | Description |
|---|---|
| A AND B | Vrai si A ET B sont vrais |
| A OR B | Vrai si A OU B ou les deux sont vrais |
| A XOR B | Vrai si A OU B mais pas les deux est vrais |
| NOT A | Vrai si A est faux |
| A > B | Vrai si A est plus grand que B |
| A < B | Vrai si A est plus petit que B |
| A <> B | Vrai si A est différent de B |
| A = B | Vrai si A est égal à B |
| A >= B | Vrai si A est plus grand ou égal à B |
| A <= B | Vrai si A est plus petit ou égal à B |

Liste des instructions de programmation structurée

Instructions d’envoi et de retour de "sous-routines"

```
SUB <Label>
     ...
ENDSUB
```

```
GOSUB <Label>
```

```
ON <variable> GOSUB <label1>, <label2>, .....
```

Une sous-routine (Subroutine en anglais) est en fait un petit programme qui accomplit une tâche en particulier. En général, la sous-routine n'effectue qu'une seule tâche et une seule. C'est en fait le "fameux" module dont nous parlions dans le chapitre consacré à la programmation structurée. Ce petit programme doit répondre aux règles de la programmation structurée à une exception près, c’est qu’en place de débuter par l’instruction BEGIN et terminer par l’instruction END, il commence par l’instruction SUB <Label> et se termine par l’instruction ENDSUB .

Nous introduisons ici la notion de "label". Un "label" ou "étiquette", est un nom que l’on donne pour identifier le petit bout de code contenu dans la sous-routine. Ce "label" sera donc utilisé dans l’instruction d’appel GOSUB <Label> et dans l’instruction de définition de la sous-routine SUB <Label>.

Les différentes syntaxes utilisées avec les sous-routines sont les suivantes :

SUB <Label> Pour déclarer la sous-routine.

ENDSUB Equivalent à "END" dans le programme principal. Termine la sous-routine.

GOSUB <Label> Instruction d'appel de la sous-routine.

ON <variable> GOSUB <label1>, <label2>, ..... Instruction d'appel indexé.

Au début de la sous-routine, il est possible de retrouver des déclarations de variables locales. Ces variables ne sont utilisées que dans la sous-routine et n'existent pas à l'extérieur de celles-ci.

Liste des instructions de programmation structurée

Instructions de test

```
IF <condition> THEN
    ...
ELSE
    ...
ENDIF
```

Comme nous l'avons décris plus haut, cette instruction effectue un test. Ce test est une expression logique au sens large. Cette expression ne peut avoir que deux états de sortie : VRAI ou FAUX. Elle peut-être composée de plusieurs expressions logiques reliées par des opérateurs logiques (AND, OR, XOR).

L'instruction de test s'utilise de la manière suivante :

```
IF <condition> THEN
     <liste d’instructions 1>
ELSE
     <liste d’instructions 2>
ENDIF
```

Dans un premier temps l'expression logique contenue dans la <condition> est évaluée. Si cette expression est vraie, alors la <liste d'instructions 1> est effectuée. Si elle est fausse, alors c'est la <liste d'instruction 2> qui est effectuée. ou

```
IF <condition> THEN
     <liste d’instructions>
ENDIF
```

Ici, seule la <condition> vraie est envisagée. Si la <condition> est fausse rien ne se passe et le programme suit sont cours sans avoir effectué la <liste d'instructions>.

Liste des instructions de programmation structurée

Instructions de boucle

```
DO WHILE <condition>
     ...
ENDDO

DO
     ...
UNTIL <condition>
```

Comme pour l'instruction de test, la <condition> est une expression logique
Comme nous l'avons décris plus haut, il existe deux sortes de boucles :

La boucle à décision préalable :

```
DO WHILE <condition>
     <liste d'instructions>
ENDDO
```

Dans cette boucle, la <liste d'instructions> n'est effectuée que si la <condition> est vraie en rentrant dans la boucle. Si la <condition> est fausse, alors la <liste d'instructions> n'est pas effectuée.

ou

La boucle à décision postérieure :

```
DO
     <Liste d'instructions>
UNTIL <condition>
```

Par contre, ici, comme la <condition> n'est évaluée qu'à la fin de la boucle, la <liste d'instruction> est toujours effectuée au moins une seule fois.

Cette structure est à la base d'un grand nombre de "plantages". En effet, il faut toujours veiller à inclure dans la boucle, les évolutions nécessaires à sa sortie, sinon, la condition de sortie ne sera jamais remplie, et l'on aura une "boucle infinie". Un autre piège de ce type de structure est qu'à l'intérieur d'une boucle, une sous-routine est appelée, et dans cette routine, l'élément de l'évaluation de la condition de sortie est modifié. Ici aussi, nous allons nous retrouver dans une "boucle infinie".

Annexe A
Liste des instructions de programmation structurée

```
Remarque dans le programme
     REM ou ’

Commande du programme
     BEGIN
     END
     PAUSE

Déclaration de variables et constantes
     DECLARE <Lieu>-<Nom> AS <Type>
     <Lieu>    : Global ou Local
     <Nom>     : Nom de la variable
     <Type>    : - INTEGER    : valeurs entières
                 - FLOATING   : valeurs en virgule flottante
                 - ALPHA      : valeurs alphanumériques
                 - BIT        : un bit égal à zéro ou un
                 - NIBBLE     : 4  bits (varie de 0 à 15)
                 - BYTE       : 8 bits (varie de 0 à 255)
                 - WORD       : 16 bits (varie de 0 à 65535)

     LET <Nom> = <Valeur ou Expression>

Envoi et de retour de "subroutines"
     SUB <Label>
     <Liste d'instructions>
     ENDSUB

     GOSUB <Label>

     ON <variable> GOSUB <label0>, <label1>, <label2>, .....


Instructions de test
     IF <condition> THEN
          <liste d’instructions 1>
     ELSE
          <liste d’instructions 2>
     ENDIF

Instructions de boucle
     DO WHILE <condition>
          <liste d’instructions>
     ENDDO

     DO
          <Liste d’instructions>
     UNTIL <condition>
```

Annexe B
Liste des opérateurs en programmation structurée

Opérateurs numériques : Les opérateurs numériques sont des modifiants qui se rapportent aux nombres en général.

| Opérateur | Description |
|---|---|
| X + Y | Addition des nombres X et Y |
| X - Y | Soustraction des nombres X et Y |
| X * Y | Multiplication des nombres X et Y |
| X / Y | Division du nombre X par le nombre Y |
| X**Y | Nombre X élevé à la puissance Y |
| SQR(X) | Racine carrée de X |
| SIN(X) | Sinus de X |
| COS(X) | Cosinus de X |
| TAN(X) | Tangente de X |
| ABS(X) | Valeur absolue de X |

Opérateurs binaires : Les opérateurs binaires se rapportent à la structure binaire des nombres.

| Opérateur | Description |
|---|---|
| X & Y | Combinaison ET bit à bit |
| X \| Y | Combinaison OU bit à bit |
| X ^ Y | Combinaison OU exclusif bit à bit |
| INV X | Inversion bit à bit du nombre X |
| >> X | Décalage à droite des bits du nombre X |
| << X | Décalage à gauche des bits du nombre X |

Opérateurs Alphanumériques : Les opérateurs alphanumériques modifient les variables alphanumériques.

| Opérateur | Description |
|---|---|
| LEFT$(X$,Y) | Extraction des Y caractères de gauche de la variable X$ |
| RIGHT$(X$,Y) | Extraction des Y caractères de droite de la variable X$ |
| MID$(X$,Y,Z) | Extraction des Z caractères à droite du Yème dans X$ |
| ASC(X$) | Extraction du code ASCII du 1er caractère de X$ |
| CHR$(X) | Donne le caractère dont le code ASCII est X |

Opérateurs logiques : Les opérateurs logiques sont utilisés lors de l’évaluation d’expression logique ou de conditions lors d’instruction de test.

| Opérateur | Description |
|---|---|
| A AND B | Vrai si A ET B sont vrais |
| A OR B | Vrai si A OU B ou les deux sont vrais |
| A XOR B | Vrai si A OU B mais pas les deux est vrais |
| NOT A | Vrai si A est faux |
| A > B | Vrai si A est plus grand que B |
| A < B | Vrai si A est plus petit que B |
| A <> B | Vrai si A est différent de B |
| A = B | Vrai si A est égal à B |
| A >= B | Vrai si A est plus grand ou égal à B |
| A <= B | Vrai si A est plus petit ou égal à B |