http://www.didel.com/
info@didel.com
www.didel.com/C/MiseAuPoint.pdf

Mise au point des programmes Arduino/C

Mettre au point un programme peut prendre beaucoup plus de temps que de l'écrire. Chacun développe sa propre technique de programmation, chaque application a des contraintes spécifiques et il est difficile de documenter une méthodologie de mise au point universelle.
Les professionnels utilisent des outils performants pour gérer leurs projets ; ils incluent un dévermineur (debugger) et ces outils sont actuellement moins coûteux, voir gratuits.
Arduino est facile à utiliser, mais il ajoute des fonctionnalités non compatibles avec les compilateur C qui ont un dévermineur. On se trouve donc un jour à vouloir transporter ses programmes Arduino sur un IDE plus performant quand le programme devient gros et difficile à déverminer. Nos conseils de bien séparer les contraintes technologiques et fonctions de base et la fonctionnalité du programme devraient vous faciliter la transistion.
L'objectif ici est d'aider les débutants qui apprécient Arduino, Pinguino ou Energia, et qui ont, comme tout le monde, des problèmes de mise au point.

Bien programmer
La première règle est naturellement de bien écrire le programme. Il doit être structuré avec les définitions regroupées et vérifiées avec les initialisations. On écrit des programmes de test en faisant attention aux conditions limites.
Des noms clairs facilitent la relecture ; il faut les choisir avec beaucoup de soin. Un commentaire devant chaque bloc est essentiel. Une ligne vide est un excellent commentaire, si elle montre que le bloc a plusieurs composantes. Bien commenter n'est pas répéter ce que fait chaque instruction, mais dire à quoi servent les groupes d'instructions et détailler les paramètres de fonctions !
Les fonctionnalités de base sont confiées à des définitions et fonctions, et on progresse vers la complexité avec des fonctions qui appellent les fonctions précédemment définies et testées. Le programme principal appelle ces fonctions en ajoutant des instructions "ciment" pour les relier. Le programme principal doit être court ou comporter des étapes successives bien distinctes.
Avec un bon découpage, retrouver les erreurs dans chaque étape est rapide et utilise si nécessaire l'une des techniques que l'on va voir.

1er conseil
Il est important de conserver tous les programmes de test et les variantes successives. Quand le programme se complexifie, il arrive souvent que l'on "perde les pédales", on n'a plus confiance dans le matériel, on soupçonne des fonctions mal testées.
En reprenant des programmes précédents qui ont fonctionné, on peut faire des tests supplémentaires, et retourner par étapes vers le programme final en se mettant dans le bain de tous les rouages. On converge vers le bug et on finit par voir comment le mettre en évidence, puis le corriger.

Erreurs de compilation
Le premier problème souvent est d'avoir une compilation correcte; il est en général difficile d'interpréter les messages du compilateur. La plupart des fautes se trouvent en relisant attentivement le programme. Les compilateurs ont un mode "verbose" qui dit à quelles lignes le compilateur a détecté des anomalies et ce qu'il a déduit, mais il faut avoir compris comment interpréter ce charabia.
Les erreurs les plus fréquentes sont d'oublier un point-virgule ou une virgule, remplacer une virgule par un point, mettre l'accolade de fin une instruction pas assez ou trop loin.
Les messages d'erreur dépendent de l'instruction incorrecte et parfois de ce qui suit ou précède. Par exemple :

; oublié *expected `;' before 'delay'* -- delay(); est l'instruction suivante
'63' cannot be used as a function – 0b00111111 n'était pas suivi d'un ;
'kk' cannot be used as a function – int kk=0 sans ;
, oublié *too few arguments to function 'void pin*
expected constructor, destructor, or type conversion before '(' token
typos *'high' was not declared in this scope* – minuscules au lieu de majuscules
expected `;' before ':' token – : au lieu de ;
{ } ( ) en trop, manquant *expected declaration before '}'*
(inspiré de http://diyroboticslab.wordpress.com/2009/06/05/correcting-arduino-compiler-errors/

Prenez un programme et créez des erreurs, cela vous aidera à vous y retrouver par la suite.

2e conseil
Ignorez les messages d'erreur et relisez attentivement TOUT le programme, c'est très utile pour rester dans le bain et bien souvent on se focalise et tourne en rond sur un endroit qui n'est pas la source du problème.

Erreurs d'exécution
Il arrive que le compilateur accepte votre programme, mais il n'a pas créé le code que l'on croit. Il peut y avoir des erreurs de type, de conversion de type (on n'en a pas parlé), un "volatile" oublié. Ou une accolade fermée trop vite. Ou un 0x oublié devant un nombre hexa, mais comme il n'avait que des chiffres de 0 à 9, ni vous ni le compilateur bien sûr n'a remarqué. Cela se remarquera à l'exécution. Très souvent, un = simple traîne dans une condition; la condition est alors toujours vraie.
Si vous appelez une fonction sans paramètre, FaireCeci(); et que vous oublié la (), le résultat n'est pas toujours correct.

Si l'exécution ne se fait pas correctement, il faut stopper l'exécution à un instant critique et observer l'état interne du processeur, la valeur des variables, les signaux sur les entrées sorties. Les dévermineurs (débugger) font cela parfaitement bien, mais on perd évidemment les aspects de temps réel puisque l'on doit interrompre le programme pour observer l'état.
Si vous êtes débutant et fan d'Arduino, Pinguino ou Energia, et ne voulez pas apprendre de suite à utiliser un EDI plus performant, il faut savoir utiliser quelques trucs bons à connaître de toute façon.

Leurrer et espionner
Un programme peut mal réagir pour certaines valeurs des entrées ou variables fournies par une fonction. On remplace la fonction qui marche par une fonction "bidon" qui assigne les variables qu'elle gère d'une façon prédéfinie. Cela permet d'avoir dans la suite du programme des états de variables connus, et de voir si cela marche avec ces variables, en particulier pour des valeurs extrèmes.
Par exemple, une fonction qui retourne une valeur 16 bits est mise en commentaire et on écrit à la place

```
int MaFonction () {
    return 37 ;
}
```

Si cette fonction modifie des variables globales, on leur assigne des valeurs sans appeler la fonction.

Observer des variables
Une solution si le programme déraille est de bloquer son exécution à différents endroits pour permettre de vérifier quel est l'état interne des variables. Avec un débugger c'est facile, mais c'est aussi lourd.
Pour bloquer l'exécution, on a vu qu'il suffit d'insérer un `while(1){}`. On peut auparavant allumer une led ou envoyer un message au terminal.

On insérera par exemple une fonction `Stop(var);` pour afficher la variable var qui n'a peut-être pas la valeur imaginée.
On peut naturellement vouloir continuer après avoir affiché et suivre à la trace l'exécution avec des commentaires disant dans quelle partie de programme on se trouve.

On peut naturellement en supprimant le `while(1)` ne pas bloquer l'exécution et afficher un numéro qui permet de reconnaître où le programme a passé et avec quelle valeur de variable :

```
void  AffiVar (byte nn,int vv) {
    Serial.print (nn);
    Serial.print (" ") ;
    Serial.println (vv) ;  // décimal
}
```

Chaque l'on affiche par exemple AffiVar (3,variableTruc) l'exécution est interrompue pour 10 millisecondes environ.

Si on veut continuer l'exécution après un arrêt avec affichage, on peut remplacer le while(1) par une attente sur un poussoir pressé puis relâché.

```
while (!PousOn)  {delay (20); }
while  (PousOn)  {delay (20); }
```

Inclusions conditionnelles
Pour enlever provisoirement ces affichages, car c'est souvent utile de les remettre en service, on peut utiliser le menu Arduino "comment/uncomment" dans le menu "Edit". On peut aussi utiliser une richesse du C beaucoup utilisée par les professionnels pour générer des variantes de programmes: les inclusions conditionnelles.
On encapsule les fonctions debug par un `#ifdef Debug . . . #endif`
et on déclare au début du programme `#define Debug`. Si Debug n'a pas été déclaré, les blocs `ifdef` sont ignorés.
Vérifions, avec un exemple aussi simple que possible

```
#define Debug
void setup()  {
   pinMode (Led, OUTPUT);
}
void loop()  {
   #ifdef Debug
   LedOn; delay (200);
   LedOff; delay (200);
   #endif
}
```

Si la ligne #define Debug est mise en commentaire, cela ne clignote plus. Vérifiez.

Temps réel
Avec un oscilloscope, on peut faire beaucoup de vérifications en dégradant à peine les timings. Sur une sortie libre, on active la sortie au début d'une fonction et on désactive à la fin. On voit ainsi si la fonction est appelée, et quelle est sa durée.

Plutôt qu'utiliser le terminal série, on peut câbler un registre série avec 8 Leds. La procédure que l'on a vue pour remplir le registre permet avec 3 fils d'afficher une variable en étant 1000 fois plus rapide que le serial.print.

Une autre solution un peu plus longue à programmer est de copier les variables à observer dans une table. Elle ne prend que quelques instructions. En sortie de la partie critique du programme, on envoie sur le terminal les valeurs qui ont été interceptées.

Ce qui perturbe le moins le programme est d'insérer une instruction `setBit ou clearBit` sur un pin libre du microcontrôleur, ou écrire un mot binaire sur un port (s'il y en a un de libre). Cela ralentit de 0.2 microsecondes et on peut allumer ou éteindre une ou plusieurs Leds qui montrent par où le programme passe. Un oscilloscope permet alors d'observer

plus précisément les durées des impulsions générées et d’ajuster les paramètres. Une application type est le décodage d’impulsions infrarouges ou radio reçues. Chaque impulsion déclenche une action logicielle ; on active un bit pendant l'action et on peut s’assurer que l'action est terminé avant l’impulsion suivante, même en cas d’interruptions intempestives.

On voit que chaque programme complexe nécessite un travail supplémentaire pour le dépanner et, même s’il fonctionne apparemment du premier coup, il faut le certifier, c’est-à-dire garantir son bon fonctionnement dans toutes les conditions d’utilisation.

jdn 131012/131110