Renaud Pinon
Développeur
Créateur de Tiny Code Studio

niveau 300

Arduino : on crée un mini-debugger !

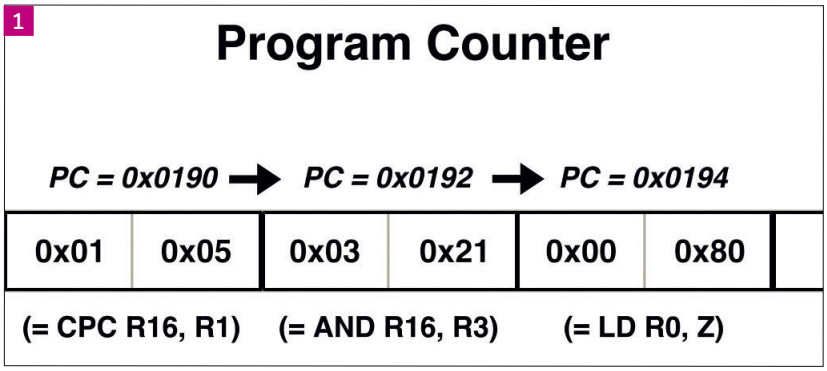
Après avoir vu le fonctionnement de la SRAM dans le précédent article (Programmez! 234), nous allons aujourd'hui nous concentrer sur les instructions stockées en mémoire Flash. Nous tenterons d'obtenir le numéro de l'instruction en cours du programme ou encore celui de l'instruction de retour d'une fonction. Nous créerons enfin un mini-debugger nous permettant d'arrêter un programme à une instruction donnée et d'afficher le contenu de la SRAM pour la fonction en cours.

Les explications données ici ont avant tout un but académique et devraient permettre de vous faire une idée du fonctionnement d'un programme du point de vue de la machine elle-même (ce qui est facilement transposable à d'autres hardwares).
Codes sources et images : https://github.com/renaudpinon/Arduino_miniDebugger

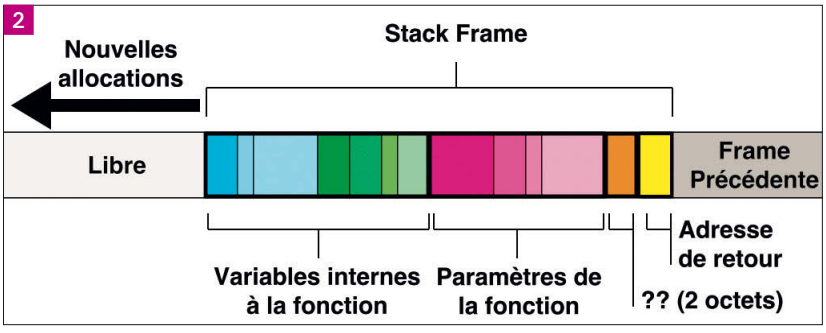
Rappels

Le programme, c'est à dire les instructions que vous avez écrites dans votre éditeur de code et envoyées à la carte, se situent dans la mémoire FLASH. Cette mémoire est inscriptible uniquement depuis votre ordinateur et est persistante même après extinction.
La SRAM quant à elle est la mémoire vive de la carte : elle contient les variables créées lors de l'exécution du programme qui sont stockées dans la "pile" (ou stack). Son début se trouve en fin de SRAM et est allouée "à l'envers", c'est-à-dire qu'à chaque nouvelle allocation, l'adresse de fin de pile est de moins en moins élevée et se rapproche du début de la SRAM.

1



2



Program Counter

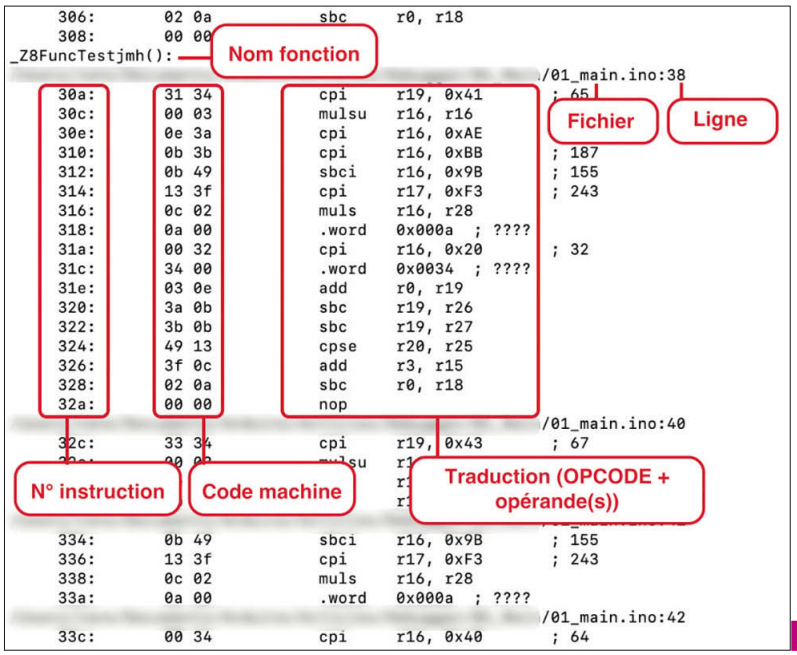
Dans la mémoire Flash, chaque instruction de votre programme a un numéro unique qui lui est attribué de façon incrémentale. Le Program Counter (compteur de programme en français, abrégé PC) est une simple variable contenant le numéro de l'instruction en cours. Elle est toutefois interne au processeur et ne peut donc être obtenue de façon directe (nous verrons un contournement possible tout à l'heure). Lorsque le processeur a fini d'exécuter une instruction en langage machine, il ajoute au PC le nombre d'octets constituant cette instruction (2 à 4 octets pour les instructions AVR), puis va lire l'instruction en mémoire FLASH ayant la nouvelle valeur de PC pour l'exécuter.
Voici une représentation schématique : 1
Dans ce schéma, les numéros d'instructions sont inscrits sur la ligne du haut (PC = 0x019X). Dans les cases se trouvent les octets de l'instruction en langage machine, et, en dessous, leur traduction en Assembleur. Dans cet exemple toutes les instructions font 2 octets : le program counter augmente donc ici de 2 à chaque fois. Evidemment les instructions d'un programme ne sont pas forcément à la suite les unes des autres : les branchements (appels de fonctions, conditions, for / while, …) changent la valeur du PC vers un numéro défini dans le branchement.
Nous utiliserons plus tard le program counter pour savoir à quel endroit du programme nous sommes et arrêter l'exécution s'il se trouve dans une plage bien précise.

Stack Frame : la théorie

Une Stack Frame correspond à la plage mémoire utilisée par une seule fonction dans la pile. La stack Frame contient donc les variables locales, les arguments passés à la fonction, les données générées par les instructions "PUSH" en Assembleur mais aussi le numéro de l'instruction de retour de la fonction.
Le format d'une stack frame dépend totalement du matériel exécutant le code, mais en ce qui concerne l'Arduino, on pourrait la représenter comme ceci : 2
N'oublions pas que les variables sont empilées de manière "descendantes", il faut donc lire le schéma de droite à gauche. On trouve tout d'abord, en jaune, l'adresse de retour de fonction (on y reviendra), d'une taille de 2 ou 3 octets suivant le modèle de carte (2 pour les Arduino UNO, 3 pour les MEGA). Puis se trouvent 2 octets "magiques" (je n'ai jamais réussi à trouver à quoi ils correspondent), puis les paramètres de la fonction et enfin les variables locales internes à la fonctions (variables + données de registres poussées lors de l'exécution de la fonction).

Adresse de retour : la théorie

Comme dit plus haut, l'adresse de retour de la fonction n'est pas une adresse en SRAM, mais un numéro d'instruction dans le programme, c'est-à-dire dans la mémoire Flash de l'Arduino. La suite d'outils AVR-GCC est capable, avec le programme *avr-objdump*, de désassembler le code compilé pour en sortir une représentation textuelle. En voici un exemple : 3



Les numéros d'instruction (les valeurs que peut prendre le Program Counter) se situent à gauche du listing. On voit bien qu'ils s'incrémentent à chaque ligne du nombre d'octets du code machine (+2 ici). Une "traduction" en Assembleur est ensuite présentée à droite, avec parfois à la fin de la ligne des commentaires précédés de ";". Le numéro de retour d'une fonction présent au début de la stack frame correspond donc à un numéro de la colonne de gauche. Pour obtenir ce numéro, on récupèrera les 2 (ou 3 suivant modèle) derniers octets de la stack frame. Cependant, la valeur qu'on obtiendra devra au préalable être transformée : premièrement, et contrairement à tout ce que j'ai pu vous dire jusqu'à présent, ses octets ne doivent PAS être inversés quand on les lit à la suite (c'est à dire que dans la réalité, il faut les inverser. Vous suivez ?).

Exemple: si on lit 01 suivi de 2B, cela donne d'habitude le nombre 0x2B01. Mais ici, l'octet de poids faible est le deuxième, pas le premier, ce qui donne donc 0x012B. De plus, les bits doivent ensuite être décalés de 1 vers la gauche (ce qui change vraiment tout : dans notre cas cela devient 0x0256 !). Voici un exemple de fonction permettant d'obtenir la valeur correcte de l'adresse de retour :

```
uint16_t translatePC(uint16_t address)
{
  // On doit inverser les octets pour
  // Obtenir le PC (ex: 0x2B01 devient 0x012B):
  uint8_t buff[2];
  buff[0] = (address >> 8);
  buff[1] = ((address << 8) >> 8);
  memcpy(&address, buff, sizeof(address));

  // Enfin on décale les bits de 1 vers la gauche:
  return (address << 1);
}
```

Enfin, je précise que l'adresse de retour ne renvoie pas à l'instruction qui a appelé la fonction, mais à celle qui la suit. Si la fonction est de type void, alors l'instruction sera à la ligne suivante. En revanche si la fonction retourne une valeur qui est affectée à une variable, l'instruction suivante sera l'affectation et donc la même ligne (exemple : pour *int x = Sum(11, 12);* : l'instruction qui suit l'appel de fonction est l'affectation du résultat à x).

La (douloureuse) pratique

Malheureusement la pratique pour obtenir la stack frame ou l'adresse de retour se révèle bien plus complexe que la théorie !

Premièrement les listings donnés ici ne fonctionneront pas de base dans Arduino IDE. La faute en revient à l'optimisation du compilateur qui détruit complément la structure des programmes : certaines fonctions, courtes ou utilisées une seule fois, seront transformées en instructions "inline", c'est à dire directement intégrées à la suite sans branchement. Cela rend impossible l'obtention du numéro de l'instruction de retour car… il n'y a plus de fonction ! Pareil pour les paramètres de fonctions ou variables locales : chaque variable potentiellement inutile sera purement et simplement supprimée à la compilation si elles ne sont pas utilisées, voire remplacées par une constante (notamment si vous mettez des valeurs en "dur" dans les appels de fonction, du genre int x = Sum(12, 13); : les paramètres de la fonction ne seront jamais dans la pile).

Deuxièmement, il est possible de modifier Arduino IDE pour compiler avec un niveau d'optimisation beaucoup moins agressif et propice au debug, mais ce ne sera pas de tout repos ! Je vous renvoie au fichier *ArduinoIDE_procedure.rtf* du GitHub qui explique comment faire cette modification pour Windows ou macOS. En substance, il s'agit premièrement de modifier le fichier platform.txt d'Arduino IDE afin de changer les options de compilation d'AVR-GCC pour ajouter -fno-lto, -fno-inline et -g -Og. Mais il faut aussi remplacer les exécutables d'AVR-GCC fournis dans Arduino IDE par les officiels, ceux d'Arduino IDE n'acceptant pas de baisser le niveau d'optimisation ! (et la raison reste un mystère pour moi…).

Je vous invite donc à faire les modifications d'Arduino IDE avant d'essayer le moindre listing de cet article, sans quoi il est peu probable que vous obteniez le résultat escompté…

Passés ces obstacles, et pour en revenir à la Stack Frame, on peut penser qu'il est très simple d'obtenir son contenu quand on voit le schéma précédent : on récupère le contenu de la SRAM à l'adresse SP (pour "Stack Pointer", correspondant au bas de la pile) sur une longueur égale à celle de la Stack Frame… Oui mais comment obtenir cette longueur ? Existe-il une fonction toute prête ? Eh non! Peut-être a-t-on un indicateur de la longueur à l'adresse SP - 2 ou à l'adresse de retour - 2 ? Nope !

Alors peut-être existe-t-il une fonction pour obtenir un pointeur vers le premier octet de la stack frame (c'est à dire les 2 octets entre l'adresse de retour et les paramètres de fonction) ? Eh bien oui ! Mais en fait non.

La suite GCC donne accès à la fonction ***void* __builtin_frame***

_address(int level) qui est censée faire juste cela : elle crée un pointeur vers le tout premier octet créé pour une stack frame. Le paramètre "level" permet de demander une stack frame précise : 0 pour la fonction en cours, 1 pour la fonction parente, 2 pour la fonction parente de la parente, etc. Si la fonction n'existe pas, elle est censée renvoyer 0 ou un nombre aléatoire. Malheureusement l'implémentation est totalement libre suivant l'architecture et en avr elle renvoie simplement la valeur de SP (c'est-à-dire la valeur la plus en bas de la pile). Et si on change le level à 1 ou plus, le résultat est incorrect ! Dommage.

Il va donc falloir ruser. Une possibilité pour obtenir la longueur est de stocker dans une variable globale la valeur de SP juste avant l'appel d'une fonction (donc avant que la stack frame ne soit construite), puis de retrancher la valeur de SP une fois qu'on est dans la fonction (donc après que la stack frame soit créée). Ce n'est pas très pratique, et, en plus, quand on revient dans la fonction parente, notre variable globale n'a plus la bonne valeur de SP. De plus, impossible d'obtenir la longueur de la stack frame dans la fonction loop() ou setup() (à moins de modifier le fichier cores/main.cpp des sources d'Arduino, mais je ne le conseille évidemment pas).

Nous allons cependant voir une autre technique, qui, même si elle présente quelques inconvénients, reste bien meilleure. Elle ne vient pas de moi, je l'ai trouvée à l'adresse suivante :

https://www.stderr.nl/static/files/Hardware/Electronics/Arduino/debug.cpp

Elle consiste à récupérer la variable Assembleur "__stack_usage" créée par avr-gcc à chaque "prologue" (phase de création d'une fonction en langage machine). Voici comment récupérer cette valeur :

```
uint8_t stack_usage;
__asm__ __volatile__("ldi %0, .L__stack_usage" : "=r"(stack_usage));
```

Si vous exécutez ce code dans n'importe quelle fonction, stack_usage contiendra la longueur de la stack frame. J'insiste sur le fait qu'il ne faut pas créer une fonction de debug qui contiendrait ces instructions et l'appeler à chaque fois qu'on en a besoin : dans ce cas la longueur retournée ne serait évidemment pas celle de la fonction qui nous intéresse mais celle de notre fonction de debug, ce qui n'a rien à voir ! On aurait pu en temps normal faire une fonction ayant l'attribut "inline" - ce qui a pour effet de copier le contenu de la fonction tel quel à chaque fois qu'elle est appelée - mais comme nous avons été obligé de modifier les arguments de compilation avec fno-inline, cela ne fonctionnera pas. Une solution viable serait alors de faire une macro qui affecterait la valeur de retour à une variable globale au lieu de l'affecter à stack_usage (les macros sont exécutées de façon "inline").

Attention : stack_usage contient la taille des données "prévisibles". Si vous déclarez un tableau de taille variable en plein milieu d'une fonction (disons : *int monTableau[variable];*), alors la taille renvoyée ne contiendra pas les octets du tableau ! L'idée pour résoudre cela est de calculer et stocker l'adresse du haut de la stack frame au tout début de la fonction : ainsi même si de nouveaux octets sont poussés dans la pile et même si on crée des tableaux de longueur variable, cela ne changera jamais cette valeur.

Voici un exemple :

```
#define DBG_RET_SIZE    2
#ifdef __AVR_3_BYTE_PC__
 #define  DBG_RET_SIZE    3
#elif defined(__AVR_2_BYTE_PC__)
 #define  DBG_RET_SIZE    2
#endif

int FuncTest(int a, int b)
{
  // début : on stocke l'adresse de départ :
  uint8_t stack_usage;
  __asm__ __volatile__("ldi %0, .L__stack_usage" : "=r"(stack_usage));
  uint16_t topSP = SP + stack_usage + DBG_RET_SIZE;

  int ret = a + b;
  // TODO: actions...
  PrintVariable(SP, topSP - SP + 1);

  return ret;
}

void PrintVariable(uint16_t address, int len)
{
  // On place un pointeur à l'adresse :
  uint8_t* pointer = (uint8_t*)address;
  for (int i = 0 ; i < len ; i+= 16)
  {
    for (int j = 0 ; j < 16 && i + j < len ; j++)
    {
      uint8_t val = *(pointer+i+j); // valeur d'1 octet à imprimer.
      Serial.print( (val < 16 ? "0" : "") + String(val, HEX) + " ");
    }
    Serial.println(""); // CRLF;
  }
  Serial.println(""); // CRLF
}
```

__AVR_3_BYTE_PC__ et __AVR_2_BYTE_PC__ sont déclarées à la compilation et permettent de savoir si on a une carte avec un program counter sur 2 ou 3 octets (dans la pratique : toute carte ayant une mémoire Flash > 128 ko a un PC de 3 octets). En tout début de FuncTest() on obtient la taille de la stack frame et on calcule son adresse de début. Cette adresse ne bougera pas quoiqu'on déclare par la suite.

Pour réaliser l'impression de la stack frame, j'ai repris la fonction PrintVariable() de mon précédent article puisqu'elle fait juste ce qu'il faut : imprimer les données à partir d'une adresse mémoire et pour une longueur donnée.

Obtenir l'adresse de retour d'une fonction

Maintenant que nous avons la stack frame, nous pouvons obtenir l'adresse de retour d'une fonction. Cela est utile pour obtenir ce qu'on appelle la "call stack", c'est à dire la pile des appels de fonctions : par exemple loop() à la ligne 40 appelle func1() à la ligne 80, qui à la ligne 83 appelle etc… Pour obtenir la valeur, on va

chercher les 2 derniers octets de la stack frame et on effectue la transformation avec la fonction TranslatePC() un peu plus haut.
Même si cela fonctionne très bien, il faut savoir qu'il existe une méthode plus simple pour obtenir cette adresse : la fonction intégrée ***void* __builtin_return_address(int level)***. Ouf, contrairement à ***__builtin_frame_address()***, celle-ci fonctionne ! Et comme précédemment, level est la fonction appelante : 0 pour la fonction courante, 1 pour la fonction parente, etc. Cependant seul le level 0 semble fonctionner en AVR. Attention : elle renvoie *void** et on serait tenté de croire qu'elle renvoie un pointeur, mais en fait elle renvoie une variable de type int ou long suivant la machine. Cette dernière représente l'adresse de retour de la fonction **avant** la transformation. A noter qu'il existe aussi une fonction censée donner cette valeur sans que l'on ait besoin de la transformer : ***void* __builtin_extract_return_addr(void* address)***. On est censé lui passer en paramètre le résultat de ***__builtin_return_address()***. Mais malheureusement, il semble bien qu'elle soit inopérante pour les AVR puisqu'elle ne renvoie jamais la bonne valeur, on se contentera donc de ***__builtin_return_address()***.
Voici un exemple d'utilisation :

```
uint16_t retAddress = (uint16_t)__builtin_return_address(0);
// TODO: transformer le résultat.
```

Obtenir le PC

On a obtenu l'adresse de retour d'une fonction. Mais comment obtient-on, en plein milieu d'une fonction, le numéro d'instruction exact où nous sommes ? C'est en fait possible et même très simple ! Après tout, en créant une fonction utilisée juste pour le debug et à laquelle on demande d'obtenir sa propre adresse de retour, si on l'appelle dans le programme "principal" alors elle nous renverra bien le numéro de l'instruction en cours. Bien sûr, en réalité, on n'obtient pas le PC réel puisque celui-ci est actuellement irrémédiablement dans notre fonction de debug, mais cela ne change rien pour nous.

```
uint16_t GetPC(uint16_t spValue)
{
  // StackFrame:
  uint16_t sp = SP;
  uint16_t diff = spValue - sp;

  uint16_t tmpAddress = 0;
  uint8_t* pointer = (uint8_t*)(sp + diff - 1);

  memcpy(&tmpAddress, pointer, sizeof(tmpAddress));

  uint8_t buff[2];
  buff[0] = (tmpAddress >> 8);
  buff[1] = ((tmpAddress << 8) >> 8);
  memcpy(&tmpAddress, buff, 2);

  return (tmpAddress << 1);
}
```

J'aurais pu utiliser ***__builtin_extract_return_addr()*** mais j'ai préféré rester sur les valeurs de SP. Par simplicité je renvoie une valeur sur 2 octets, mais le programme que vous trouverez sur le GitHub permet bien d'obtenir un numéro d'instruction sur 2 ou 3 octets. On appelle la fonction en la plaçant juste avant l'instruction qui nous intéresse :

```
int pc = GetPC(SP); digitalWrite(13, HIGH);
```

Désassemblage

Maintenant que nous avons le numéro d'instruction, voyons comment désassembler un programme pour ensuite corréler l'instruction à une ligne. Pour cela, il suffit juste de savoir où se trouve le fichier .elf créé par AVR-GCC. Avec Arduino IDE on peut l'obtenir très simplement : il suffit de passer l'éditeur en mode "verbose". Pour cela, ouvrez les préférences du logiciel et cochez les cases "Compilation" et "Téléversement" du champ "Afficher les résultats détaillés pendant :". Ensuite, si vous compilez un programme, vous verrez dans la console certaines commandes (par exemple avr-size) qui utilisent et affichent le chemin du fichier .elf (pour moi sur MacOs :

```
"/var/folders/zg/ldrv10ws4t113vw7hnf_32hh0000gn/T/arduino_build_219222/test.ino.elf").
```

Copiez le chemin dans le presse-papier, puis ouvrez une commande DOS ou Terminal. Notre but est d'exécuter le fichier avr-objdump qui se trouve dans le dossier hardware/tools/avr/bin d'Arduino IDE (sur macOS : il faudra faire un clic droit sur l'application puis "Afficher le contenu du paquet" et enfin aller dans le dossier Contents/java).
On exécute le programme dans notre Terminal / commande DOS :

```
chemin/applications/avr/avr-objdump -D -C -l chemin/fichier/arduino.ino.elf
```

(attention : -l est un L minuscule, pas un i !)

Si tout va bien, un listing de désassemblage du programme, qui ressemble à la capture présentée en début d'article, apparaît. Attention : à la moindre modification du programme et recompilation, il faudra penser à refaire cette commande !

Mini Debugger

Je vous propose maintenant de passer aux travaux pratiques avec la création d'un mini-debugger. Il devra être capable de :

- Communiquer avec l'utilisateur via le port série pour arrêter le programme à l'instruction en cours ;
- Demander au programme de s'arrêter tout seul quand il arrive dans une plage d'instructions ;
- Demander au programme de s'arrêter dès que possible ;
- Obtenir l'état de l'exécution (en cours, arrêté, arrêt prochain à l'instruction XXXX, etc.) ;
- Obtenir l'adresse de retour de la fonction en cours ;
- Reprendre l'exécution du programme ;
- Afficher dans le moniteur série le contenu de la stack frame lorsque le programme s'arrête.

Les étapes que devra suivre l'utilisateur seront les suivantes :

- Insérer dans son programme, à chaque début de fonction, un code permettant de calculer l'adresse de départ de la stack frame ;
- Insérer à chaque instruction un appel à une fonction de debug créant un "point d'arrêt" ;
- Compiler le code dans Arduino IDE ;

- Désassembler si besoin le programme avec l'utilitaire avr-objdump (pour vérifier que les numéros de lignes correspondent bien aux instructions et adresses de retour ;)) ;
- Ouvrir le moniteur série et taper :
- STOP pour arrêter le programme à l'instruction en cours,
- STOP XXXX YYYY pour arrêter le programme lorsqu'il arrive dans une plage d'instructions précise (XXXX étant l'adresse de départ et YYYY l'adresse de fin en hexa),
- NEXT pour relancer le programme s'il est stoppé et l'arrêter à l'instruction suivante,
- PLAY pour reprendre l'exécution,
- RETURN pour afficher l'adresse de retour de la fonction actuelle,
- STATE pour obtenir l'état actuel (arrêté, en cours, etc.).

Le code est disponible sur le GitHub donc je n'imprimerai pas la totalité ici. J'ai créé deux fichiers s'appelant Debugger.h et Debugger.cpp qu'il suffit d'ajouter à n'importe quel projet (copiez-les simplement dans le même dossier que votre fichier .ino et ils seront considérés lors de la compilation par Arduino IDE).
Pour résumer, Debugger.cpp contient une fonction pour interpréter les commandes reçues sur le port série, une fonction pour obtenir l'adresse de retour d'une stack frame, une fonction pour imprimer le contenu d'une variable en hexa sur une longueur de X octets (vue dans l'article précédent) et une fonction DebugBP() pour "créer" un point d'arrêt.
Cette dernière va vérifier l'adresse de l'instruction actuelle et sortir directement si les conditions d'arrêt ne sont pas réunies. Dans le cas contraire (on est à l'instruction voulue ou on doit s'arrêter à la prochaine instruction), elle rentre dans une boucle while() infinie, ce qui va "stopper" l'exécution du programme. Le programme vérifiera tout de même à chaque itération de cette boucle s'il y a des données envoyées par l'utilisateur sur le port série.

```
void DebugBP(uint16_t spValue, uint16_t previousSpValue, String file, uint16_t line)
{
  uint32_t pc = DebugGetPC(SP, spValue);

  do {
    if (_dbgState == DebugStateStopNext
        || (pc >= _dbgStopMin && pc <= _dbgStopMax))
    {
      if (_dbgState != DebugStatePause)
      {
        _dbgState = DebugStatePause;

        Serial.println(String(F("Stop fichier ["))
            + file + String(F("] : "))
            + String(line));
        Serial.println(String(F("PC = 0x"))
            + String(pc, HEX));

        // Adresse retour:
        _dbgReturnAddress = DebugGetPC(spValue, previousSpValue);

        Serial.println(F("Stack frame: "));
        int stackSize = (int)(previousSpValue - spValue + 1);
        DebugPrintVariable(spValue, stackSize);
      }
    }
    DebugSerialRead();
  } while (_dbgState == DebugStatePause);
}
```

La fonction accepte 4 arguments : spValue, previousSPValue, file et line. Les deux premiers sont la plage de mémoire de la stack frame, file est le nom du fichier actuel (macro __FILE__), et line est le numéro de ligne dans le fichier (macro __LINE__). Voici un exemple d'appel juste avant une instruction quelconque (DBGStackStart est une variable locale contenant l'adresse de départ de la stack frame) :

```
DebugBP(SP, DBGStackStart, __FILE__, __LINE__); delay(1000);
```

Ici un "point d'arrêt" est créé avant l'appel à delay(1000). Si l'utilisateur a demandé de s'arrêter à cet endroit (par une commande STOP, STOP XXX YYY ou NEXT), alors le nom du fichier et la ligne s'afficheront, suivis du contenu de la stack frame.
Pour plus de simplicité j'ai créé une macro au nom un peu plus court qui appellera cette fonction avec les arguments voulus :

```
#define BP()    (DebugBP(SP, DBGStackStart, __FILE__, __LINE__))
```

Du coup la ligne précédente devient :

```
BP(); delay(1000);
```

Ce qui est plus lisible et surtout moins fatigant s'il faut l'écrire à chaque ligne ;)
Il ne faut pas oublier à chaque début de fonction d'obtenir le début de stack frame. Il est un peu plus compliqué de créer une seule macro puisque tout doit rester "inline". Et une macro ne peut faire qu'une ligne. J'ai donc opté donc pour deux macros :

```
#define STACKLEN() __asm__ __volatile__("ldi %0, .L__stack_usage" : "=r"(_dbg
CurrentFrameSize))
#define STACKGETSTART() uint16_t DBGStackStart = SP + _dbgCurrentFrameSize + DBG
_RET_SIZE
```

STACKLEN() stocke la longueur de stack frame actuelle dans la variable globale *_dbgCurrentFrameSize*. *STACKGETSTART()* crée une variable **locale** *DBGStackStart* qui est l'adresse du haut de la stack frame. A noter qu'après cela il ne faut plus utiliser_dbgCurrentFrameSize de toute la fonction car sa valeur pourrait être incorrecte.
On appellera ces deux macros au tout début de chaque fonction :

```
void MaFonction()
{
  STACKLEN();
  STACKGETSTART();
  // TODO: instructions suivantes...
}
```

Un programme exemple très simple (allumage et extinction de la led de la carte) se trouve sur le GitHub. Voici le résultat dans le moniteur série après lui avoir demandé de s'arrêter entre deux numéros d'instructions :

```
Bienvenue dans le mini-debugger.
Tapez:
- STATE pour connaître l'état actuel,
- STOP pour arrêter le programme à l'instruction actuelle,
- STOP XXXX YYYY pour arrêter le programme lorsqu'il arrive dans une plage d'adresse,
- PLAY pour reprendre l'exécution,
- RETURN pour obtenir l'adresse de retour de la fonction actuelle (état stop),
- NEXT pour continuer jusqu'a la prochaine instruction.
Ceci est un test d'addition : 255
Ceci est un test d'addition : 255
executed: stop 1034 1060
Ceci est un test d'addition : 255
Stop fichier [/XX/XXXXX/XXXXXXX/XXXXXXX/test.ino] : 48
PC = 0x1052
Stack frame:
29 af 03 43 00 43 00 21 c5 00 08 cf
```

Les lignes s'affichent dans la sortie, mais si vous voulez savoir à quoi correspondent les numéros d'instructions, il faudra désassembler le programme. Il suffira de rechercher ".ino:[numero de ligne]" pour trouver les instructions de la ligne qui vous intéressent, et de rechercher "XXXX:" pour trouver l'adresse hexadécimale de l'instruction que vous désirez.

Mais où sont mes variables ?

Effectivement vous vous rendrez peut-être compte que les variables ne sont pas toujours présentes dans la stack frame d'une fonction. Pourtant vous les utilisez avec succès, mais elles ne sont pas là. Pourquoi ? Encore une fois, la faute en revient aux optimisations de compilation. Si on a une fonction int Sum(int a, int b) que l'on appelle avec deux paramètres en dur (exemple : *int value = Sum(10, 15);*), alors vous pouvez être sûr que le compilateur va créer des constantes qui n'iront pas dans la pile. Le seul véritable moyen est d'utiliser d'une façon ou d'une autre leur adresse. On peut par exemple écrire dans la fonction *Sum()* :

```
DebugPrintVariable((uint16_t)&param1, sizeof(param1));
DebugPrintVariable((uint16_t)&param2, sizeof(param2));
```

Ce qui aura pour effet d'imprimer le contenu des variables dans le moniteur. Vous pourrez constater ensuite que leurs valeurs se trouvent bien dans la stack frame.

Conclusion

Nous avons vu beaucoup de choses aujourd'hui. La théorie est très simple, mais la pratique se révèle plus difficile du fait des nombreuses optimisations à la compilation qui dénaturent totalement le programme. Une grande partie des concepts que j'ai exposés ici ne serviront jamais dans la vie d'un développeur lambda, mais j'ose espérer avoir donné les clés nécessaires à ceux qui aiment bidouiller et auront envie de tester le fonctionnement interne de leurs machines préférées (ce que j'ai expliqué valant pour de nombreux hardwares différents).

Ces concepts sont grosso-modo ceux que j'utilise dans mon application Tiny Code Studio pour obtenir la pile des appels des fonctions, arrêter l'exécution du code aux points d'arrêts définis par l'utilisateur et obtenir la valeur des variables. Bien-sûr, l'utilisation est bien plus poussée puisqu'il n'y a pas qu'un seul point d'arrêt, et le logiciel affiche une représentation des valeurs des variables suivant leur type. Mais les principes restent les mêmes.

Ceux qui le veulent pourront essayer d'améliorer ce mini-debugger et l'utiliser dans leurs projets pour se simplifier le déboggage (adieu Serial.print() !). On pourrait par exemple faire une macro qui, écrite après la déclaration d'une variable, afficherait l'adresse de cette dernière et sa taille dans le moniteur série, permettant ainsi à l'utilisateur, lorsque le code est arrêté, de demander sa valeur en hexadécimal. •