GIF-3002 Mémoires, Logiciel

Ce cours discute de gestion de la mémoire dans les systèmes embarqués. Il discute d'abord d'allocation de mémoire par le compilateur et l'éditeur de lien. Il détaille où sont placées les instructions et les données de manière statique. Ensuite, le cours discute d'allocation dynamique de mémoire, c'est-à-dire pendant l'exécution d'un programme. La pile et les fonctions new/malloc seront présentées. Enfin, le cours discute de disposition des programmes en mémoire et de translation d'adresse. Un rappel sera fait sur les stratégies d'allocation de mémoire et divers concepts plus avancés seront présentés.

1 Allocation de mémoire par le compilateur et l'éditeur de liens, programme unique

1.1 Instructions pour accéder à la mémoire

Plusieurs instructions servent à accéder à la mémoire. Il y a au moins une instruction d'accès à la mémoire par plage d'adresse de mémoire, il y a au moins une instruction d'accès à la mémoire par mode d'adressage et, pour certains modes d'adressages, il y aura plusieurs instructions en fonction de la taille des paramètres.

Pour plusieurs microcontrôleur, il y a plusieurs plages d'adresses pour accéder à la mémoire d'instruction, à la mémoire de donnée interne, à la mémoire de donnée externe, aux entrées/sorties… Pour chacune de ces plages, vous retrouverez une ou des instructions différentes (opcode différent) permettant d'accéder à la plage.

> *Lorsque plusieurs plages d'adresses sont disponibles pour les données, il faut habituellement accoler un qualificatif aux variables globales pour indiquer au compilateur à quelle mémoire elles appartiennent. Par exemple, vous retrouvez des déclarations comme « xdata unsigned char MaVariableGlobaleEnRAMExterne » ou, encore, « idata int MaVariableGlobaleEnRAMInterne ».*

Pour chaque mode d'adressage, il y a une instruction unique d'accès à la mémoire : en fonction de l'opcode, le microprocesseur déterminera comment calculer l'adresse d'une variable. Ainsi, le LOAD pour lire une donnée à une adresse déterminée par un registre + une constante (i.e. LOAD R1, [R0 + 1234]; indirect indexé) aura un opcode différent du LOAD pour lire une donnée à une adresse déterminée par la somme de deux registres.

Enfin, l'opcode des instructions permettant d'accéder à la mémoire varie en fonction de la taille des opérandes. Lire ou écrire un octet n'est pas comme lire un mot sur quatre octets! La taille des constantes peut aussi changer l'opcode utilisé…

Heureusement, l'assembleur et le compilateur gèrent les différents opcodes en fonction des affectations effectuées. Cependant, il faut souvent insérer des directives ou des mots réservés dans le code afin de les aider.

1.2 Mémoire d'instructions

Dans un système embarqué, les instructions des programmes sont habituellement emmagasinées en mémoire comme de la FLASH plutôt que sur un disque dur. Il ne faut pas charger les instructions dans la mémoire vive pour quelles s'exécutent : les instructions sont déjà là.

Sauf exception voulue par le programmeur, le compilateur génère du code avec des emplacements relatifs : l'adresse de destination des branchements est calculée par rapport à l'adresse du branchement, l'adresse d'une constante est déterminée par rapport au nombre d'instructions qui séparent la constante de l'instruction en cours, l'adresse du code dans une fonction est calculé par rapport au début de la fonction… Le code est compilé pour être re-localisable aisément à l'intérieur du programme par l'éditeur de liens.

L'adresse de la première instruction exécutée est rarement l'adresse 0. Habituellement, l'adresse 0 contient la table des vecteurs d'interruptions… en mémoire non-volatile ou en RAM…

Certaines directives de compilation et d'édition de lien (habituellement #pragma ou ORG) permettent de placer des instructions ou des fonctions dans des segments spécifiques de la mémoire. Ces segments sont définis dans un fichier de commande pour l'éditeur de lien qui peut être modifié en fonction du microprocesseur et de la mémoire disponible.

Dans la plupart des systèmes microcontrôleurs, il est possible d'exécuter des programmes à partir de la RAM. D'une part, l'exécution d'instruction en RAM est parfois plus rapide que l'exécution d'instruction en ROM parce que la RAM est, un peu, plus rapide. D'autre part, permettre d'exécuter des programmes à partir de la RAM est fondamental si on veut pouvoir charger et exécuter des programmes de l'utilisateur situé dans des périphériques de stockages à l'accès très lents (disque dur...).

1.3 Mémoire de données

1.3.1 Variables globales d'un fichier

De la mémoire RAM est allouée à chaque variable globale. Le compilateur indique la taille du symbole à l'éditeur de lien qui lui attribue une adresse.

Certaines directives de compilation et d'édition de lien (habituellement #pragma) permettent de placer des variables globales dans des segments spécifiques de la mémoire. Ces segments sont définis dans un fichier de commande pour l'éditeur de lien qui peut être modifié en fonction du microprocesseur et de la mémoire disponible.

Les variables statiques sont habituellement gérées comme des variables globales. Seule leur portée change : elle est limitée au fichier contenant la variable.

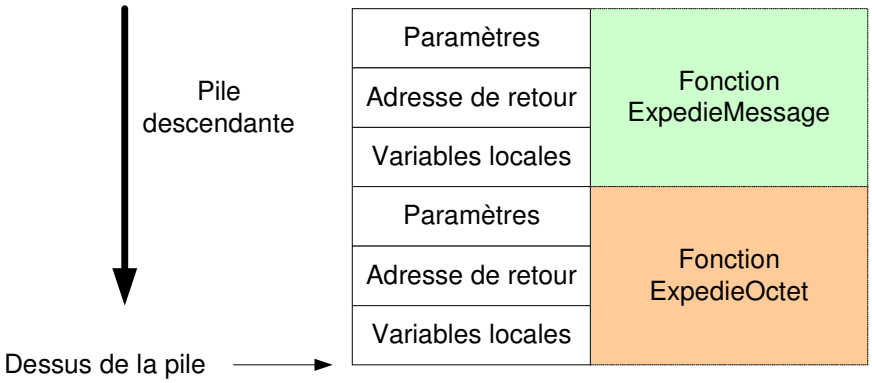
1.3.2 Variables locales et passages de paramètres

Il y a deux stratégies possibles pour emmagasiner temporairement les variables locales et les paramètres copiés d'une fonction : utiliser des registres ou utiliser la pile. Utiliser la pile est la meilleure stratégie et il s'agit de l'approche employée dans presque tous les cas.

> *Il est aussi possible de compiler les variables locales ou les copies des paramètres comme des variables globales, mais cette approche est peu utilisée de nos jours pour plusieurs raisons : les fonctions ne sont plus réentrantes (elles ne peuvent pas être appelées de deux contextes différents), une fonction ne peut plus s'appeler elle-même, le compilateur/éditeur de lien doit conserver une trace de tous les appels de fonctions afin de savoir quel espace de mémoire peut être réservé à chaque fonction et savoir quel espace de mémoire peut être réutilisé d'une fonction à l'autre… Bref, les variables locales étaient compilées comme des variables globales dans les systèmes ayant peu de registres, ayant une pile très petite et/ou ayant une pile dans une mémoire d'accès très lent par rapport à d'autres mémoires du système*[1].

Les variables locales peuvent être implémentées avec des registres (sauvegardés sur la pile lors de l'entrée de la fonction) et il est possible de faire du passage de paramètre avec les registres seulement. Cette méthode est très rapide et peut permettre un gain de temps notable. Cependant, il y a un nombre limité de registres et passer les paramètres avec des registres rend les fonctions non-réentrantes. Par ailleurs, le gain en rapidité est diminué par la sauvegarde et la récupération des registres.

La façon habituelle de passer les paramètres est d'utiliser la pile. Les variables locales sont également mises sur la pile habituellement. Par exemple, si on suppose un appel de la fonction ExpedieMessage qui appelle la fonction ExpedieOctet, la pile, descendante, aura le contenu suivant après l'appel de ExpedieOctet :



[1] Exemple : Une application devant être exécutée rapidement par un 8051 dans lequel certains Special Function Registers (SFR) sont utilisés pour contenir les variables locales. Les SFRs sont beaucoup plus rapide d'accès que la mémoire data externe du 8051 contenant la pile.

Si le temps le permet, un exercice sur le passage de paramètres avec le LM3S9B92 sera présenté en classe. Par ailleurs, vous devez être capable d'expliquer pourquoi les paramètres de la fonction son mis sur la pile avant l'adresse de retour qui est elle-même mise avant les variables locales…

1.3.3 Initialisation des variables

Habituellement, toutes les variables sont initialisées à 0 par une boucle d'écriture de la RAM, au démarrage du système. Lorsqu'une variable globale reçoit une valeur initiale, le compilateur ajoute des instructions pour changer la valeur de la variable avant l'exécution du main ou retourne une erreur… Lorsqu'une variable locale reçoit une valeur initiale, le compilateur ajoute des instructions dans la fonction pour initialiser la variable correctement.

De manière générale, il faut utiliser une directive d'assembleur (no init) lors de déclaration de la variable globale pour éviter qu'elle ne soit écrite à 0 lors du démarrage du système. Le compilateur et l'éditeur de lien disposent habituellement les variables no init et celles avec init dans des segments différents de mémoire RAM.

Lorsqu'on ajoute une mémoire externe à un microcontrôleur, toutes les variables déclarées dans cette mémoire externe devraient être no init : il faut initialiser le lien de communication avec la mémoire externe avant d'y écrire une valeur !

1.3.4 Données en mémoire non-volatile

Parfois, certains paramètres d'opération du système doivent être configurables et sauvegardés en mémoire non-volatile. Pour insérer avoir des variables non-volatiles, plusieurs stratégies sont possibles:

- Ajouter une mémoire non-volatile externe comme de la EEPROM
    - Demande une mémoire externe !
- Utiliser une ou plusieurs pages de FLASH à l'intérieur de la mémoire d'instruction pour sauvegarder des données
    - Vous devez réserver un segment de mémoire code pour les variables.
- Mettre une batterie sur une mémoire volatile
    - Peu pratique pour plusieurs raisons comme la grosseur des batteries et leur maintenance
- Etc.

Dans tous les cas, il faudra probablement protéger les données en mémoire non-volatile des erreurs comme une faute d'alimentation pendant l'écriture ou d'une corruption des données…

Un CRC, Cyclic Redundancy Check, est une valeur, habituellement sur 16 bits, calculée à partir de tous les octets d'une mémoire ou d'un message. Le CRC est calculé après une écriture de la mémoire et emmagasiné dans celle-ci. Pour vérifier la mémoire, il suffit de recalculer le CRC et de comparer avec la valeur calculée lors de la dernière écriture de la mémoire.

Plusieurs évènements atypiques peuvent causer une corruption des données en mémoire non-volatile : ESD causant une transition indésirable sur une broche de communication avec la mémoire, ESD causant un saut dans le code qui fera écrire une donnée aléatoire, corruption de la FLASH après une faute d'alimentation, erreur dans la gestion des fautes d'alimentation survenant pendant l'écriture des données…

1.4 Quantité de mémoire requise

L'éditeur de lien génère, parfois sur requête, un fichier (un map file) qui décrit l'emplacement de toutes les variables du système et leur taille. Ce fichier peut décrire également l'emplacement des fonctions du système et leur taille. De ce fait, il a une valeur inestimable lorsque la quantité de mémoire du système est limitée par rapport aux applications exécutées par le microprocesseur.

Voici quelques items à noter par rapport aux quantités de mémoires disponibles :

- Utiliser des librairies peut augmenter de manière significative la quantité de code généré pour votre programme. Parfois, juste déclarer une variable de type "float" avec un microprocesseur ne traitant que les entiers ou, encore, utiliser l'opérateur new(), peut augmenter la taille du code de quelques kilooctets…
- Lorsque les mots de la mémoire sont grands (32-bits par exemple), il arrive que les petites variables (char, short) utilisent tout de même l'espace mémoire d'un mot selon les options de compilation. La taille des objets et des structures sera également très probablement alignée sur un multiple de la taille des mots.
- Optimiser la mémoire, soit manuellement, soit avec le compilateur (voir Compilation, Édition de lien et IDE.doc) permet de sauver de l'espace mémoire, mais peut introduire des erreurs dans le code… Idéalement, chaque partie du code est testée adéquatement avant l'optimisation et testée de nouveau ensuite.
- Il est habituellement possible de gagner de la mémoire en réduisant la taille de la pile ou du heap. Ces dernières structures de données sont souvent surdimensionnées par défaut… et par prudence.

2 Allocation dynamique de mémoire

Lorsqu'on utilise un langage de programmation orienté objet, l'allocation dynamique de mémoire est très importante. Dans les systèmes embarqués, cette allocation doit se faire rapidement. Elle doit aussi maximiser l'espace mémoire disponible car la quantité de mémoire est souvent limitée.

2.1 Aspects généraux

Pour allouer de la mémoire lors de création d'objets ou pour d'autres besoins, l'éditeur de lien réserve habituellement de la mémoire appelée « heap » ou tas (en français). Le tas contiendra les variables créées dynamiquement, des informations sur les segments de tas alloués et des informations sur les espaces de mémoire disponibles.

Il y a plusieurs façons d'allouer de la mémoire et plusieurs façons de libérer de l'espace mémoire. Autrement dit il y a plusieurs façons d'implémenter les fonctions new/delete (en C++) ou malloc/free (en C).

2.2 Bassin de Mémoire

Les algorithmes d'allocations dynamiques pour les RTOS doivent être plus rapides (donc plus simples) que ceux pour les OS généraux. Une approche habituellement adoptée est d'utiliser un bassin de mémoire (*memory pool*) avec des blocs de mémoire de taille fixe (il est possible d'avoir plusieurs tailles de bloc, mais la taille d'un bloc de change pas).

Avec un memory pool, l'allocateur alloue un bloc de mémoire libre (le premier disponible) à chaque nouvelle variable créée. L'allocateur maintient également une liste de blocs libres et chaque bloc de mémoire est référencé avec un numéro de référence (handle).

Les principaux avantages des memory pools sont la simplicité et la rapidité. Pour allouer un bloc de mémoire, il suffit de prendre un bloc dans la liste des blocs libres et de retourner un pointeur ou un numéro de référence sur le bloc réservé. Pour libérer un bloc de mémoire, il suffit de remettre un bloc dans la liste des blocs libres…

Les principaux désavantages des memory pools sont la fragmentation interne (peu importe la taille de l'object/variable instancié, la quantité de mémoire utilisée est un multiple de la taille des blocs) et la nécessité de connaître les applications qui seront exécutées par le RTOS : il faut estimer le nombre maximal et la taille maximale des objets alloués afin d'avoir un memory pool de format adéquat.

2.2.1 Exemple d'implémentation

Un tas a été divisé en 32 blocs de 128 octets pour allouer de la mémoire dynamiquement avec un algorithme de MemoryPool. Ce tas est accompagné de la variable BlocLibres, sur 32 bits, pour identifier les blocs du tas disponible pour allouer de la mémoire.

Comment seront implémentées les fonctions new et free pour ce système avec char Tas[32][128] et int32 BlocLibres?

New	Free

void* new(int taille)
{
  int i;
  int Masque;
  if(taille > 128)
  return 0;
  for(i = 0; i < 32; i++)
  {
  Masque = 1 << i; //Bit du bloc choisi
  if(BlocLibres & Masque)
  {
  BlocLibres &= ~Masque; //Prend le bloc
  return &Tas[i][0];
  }
  }
  return 0; //Pas de bloc libre
}
	void Free(void* AdresseDeBloc)
	{
	  int iBloc;
	  int Offset;
	  int Masque;
	
	  Offset = AdresseDeBloc - &Tas[0][0];
	  iBloc = Offset/128;
	  Masque = 1 << iBloc;
	  BlocLibres |= Masque;
	}

2.3 dlMalloc

Le populaire algorithme de M. Doug Lea sera présenté en classe pour expliquer l'allocation dynamique de mémoire. Voir le site web suivant :

http://g.oswego.edu/dl/html/malloc.html

Si le temps le permet, les modifications à l'algorithme de M. Doug Lea apportées principalement par M. Wolfram Gloger pour supporter les systèmes d'exploitation avant plusieurs threads seront présentés en classe (voir http://www.malloc.de/en/index.html)

3 Allocation de mémoire pour plusieurs programmes

3.1 Mémoire virtuelle et algorithme d'allocation de mémoire

Voir le cours PartageDuTempsDeCPU.doc.

3.2 Protection de la mémoire

La protection de la mémoire consiste à interdire l'accès en lecture et/ou en écriture de certains segments de la mémoire à certains processus.

La protection de la mémoire est habituellement implémentée en ajoutant un champs de bits décrivant chaque segment de la mémoire. Ces bits disent : si le segment de la mémoire est réservé au système d'exploitation ou aux programmes de l'usager, si le segment est en lecture seule, si le segment de mémoire peut être partagé entre plusieurs processus, si…
Le Memory Protection Unit du ARM Cortex M3 sera présenté en classe.