GIF-3002 Compilation, édition de liens et IDE

Ce cours discute des options offertes par un environnement de développement intégré (IDE). Il présente d'abord les environnements de développement intégrés et la création d'exécutables en général. Ensuite, il présente le fonctionnement général du compilateur et le fonctionnement général de l'éditeur. Enfin, ce cours présente les options les plus communes lorsque l'on compile et crée l'exécutable.

1 Fonctionnement général

1.1 Compilation, édition de liens et exécutable (rappel)

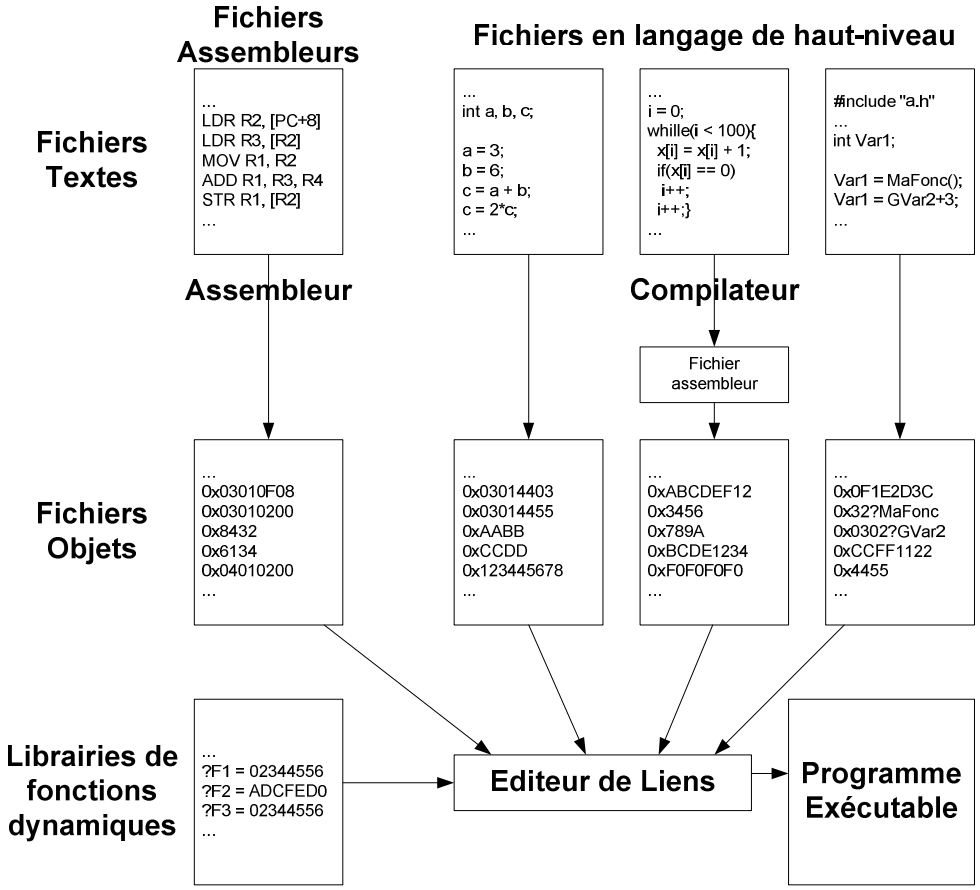

Figure 1 - Création d'exécutable

Tel qu'illustré dans la figure ci-dessus, un programme est d'abord fait à partir de fichiers textes. Les fichiers textes sont écrits avec des mnémoniques (assembleur) ou avec une syntaxe propre à un langage de programmation donné. Ils sont ensuite transformés en langage machine par le compilateur si la syntaxe est correct. Enfin, l'éditeur de liens regroupe les différents fichiers objets constituant un programme et les lie ensemble afin de constituer le programme ou l'exécutable.

1.2 Environnement de développement intégré (IDE)

L'environnement de développement intégré est un logiciel qui facilite la création et le déverminage d'exécutables. Cette application intègre plusieurs fonctions différentes:

- Support à l'édition de code : les environnements de développement intégrés offrent plusieurs fonctionnalités afin de développer efficacement un logiciel dans un langage de programmation donné. Ces fonctionnalités peuvent aller de la navigation parmi plusieurs fichiers, à la recherche de chaînes de caractères en passant par la complétion automatique de mots.
- Interface usager simple pour la création d'exécutable : l'IDE permet habituellement de compiler tous les fichiers textes d'un exécutable en appuyant sur un seul bouton et sans avoir à éditer manuellement un ensemble de makefiles. Il permet de lancer tout aussi facilement les éditeurs de liens.
- Programmation de la mémoire d'instruction : Habituellement, l'IDE possède les fonctionnalités nécessaires pour transférer le programme créé dans la mémoire d'instruction du microcontrôleur (ou de dans de la mémoire externe) automatiquement.
- Déverminage de code : L'IDE permet de déverminer le code créé, soit à l'aide d'un simulateur de microprocesseur, soit avec le microprocesseur lui-même, en mode debug.

2 Compilateur et Assembleur

Un compilateur ou un assembleur est un programme qui décode des mots ou des séquences de mots afin de les transformer en code machine. Ce programme remplace progressivement le texte d'un fichier par du code et alloue de la mémoire pour les variables.

2.1 Pré-compilation

La pré-compilation survient avant la compilation. Dans cette phase, le compilateur recherche certains mots spécifiques/prédéfini et exécute des commandes déterminées par ces mots que l'on nomme directive.

Les principales directives de pré-compilation sont les #include ou équivalent, les #define ou équivalent et les #ifdef/#endif ou équivalent.

- La directive #include "MonFichier.h" permet de remplacer la ligne #include "MonFichier.h" par le contenu intégral de MonFichier.h.
- La directive #define MA_CONSTANTE 4 ou #define MA_MACRO (a = b +c) permet de remplacer toutes les instances de MA_CONSTANTE par 4 avant la compilation et MA_MACRO par (a = b + c).
- La directive #ifdef MON_SYMBOLE permet de retirer/ajouter du texte dans le fichier à compiler en fonction d'options. Si MON_SYMBOLE n'est définit (avec #define MON_SYMBOLE par exemple), tout le texte entre #ifdef MON_SYMBOLE et le #endif correspondant sera retirer du fichier à compiler, avant la compilation.

2.2 Compilation

La compilation s'effectue sur un seul fichier à la fois, par étapes :

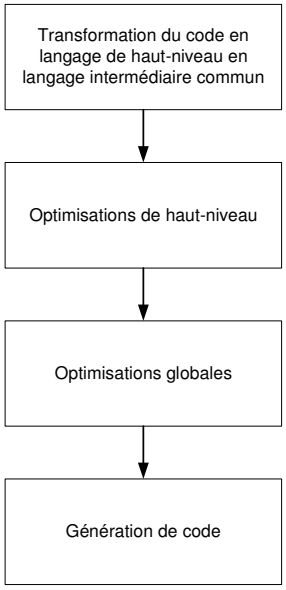

Figure 2 - Exemple d'étapes de compilation

2.2.1 Déclaration de variables et de fonctions

Une variable ou une fonction est déclarée lorsque le compilateur connaît la nature de cette variable ou de cette fonction.

Dans le cas d'une variable, la déclaration de la variable indique au compilateur : sa taille (en octets), sa portée (dépend de l'endroit de la déclaration), le type de mémoire alloué pour la mémoire et certains attributs de la variables (statique, volatile, no init, etc.).

Dans le cas d'une fonction, la déclaration de la fonction indique au compilateur : les paramètres de la fonction, la nature de la variable retournée par la fonction, la portée de la fonction (dépend de l'endroit de la déclaration) et certains attributs de la fonction (exemple : le jeu d'instruction utilisé pour coder la fonction).

Lors de la compilation, le compilateur retournera une erreur s'il rencontre un mot qui n'est pas réservé pour le langage de programmation ou déclaré : le compilateur ne sait pas quelle instruction doit être utilisée pour manipuler ce mot.

2.2.1.1 Déclaration de variables externes, globales

Il est possible de déclarer une variable ou une fonction provenant d'un autre fichier avec la directive appropriée (extern). Dans ce cas, le compilateur, même s'il connaît la nature de la variable ou de la fonction ne pourra pas transformer la totalité du texte en langage machine, car il ne connait pas les adresses finales ou relatives de ces variables et ces fonctions.

Le compilateur se contente alors de noter des ***symboles*** dans le fichier objet : il laisse des annotations qui diront à l'éditeur de liens de remplacer, dans le fichier compilé, à un endroit précis, un symbole par son adresse.

2.2.2 Définition de variables et de fonctions

Une variable ou une fonction est définie lorsque de la mémoire a été allouée pour cette variable ou fonction.

2.2.3 Création de fichiers objets

Le compilateur produit un fichier objet par fichier texte analysé. Le fichier objet contient, entre autres, la liste des variables/fonctions déclarées et définies dans le fichier analysé, la liste des symboles déclarés qui ne sont pas définis et plusieurs autres données pour l'éditeur de liens (la taille et la nature des fonctions, la taille et la nature des variables…).

Évidemment, le compilateur produit le code machine associé au fichier analysé. Cependant, les adresses (de variables ou de fonctions) contenues dans le code compilé sont toutes relatives au début du fichier : il n'y a pas d'adresse absolue (à moins de rencontrer une directive spécifique à cet effet!). De manière générale, les adresses sont toujours relatives (au début du fichier ou au début du programme ou à une adresse 0 fictive…) : cela permet de relocaliser les variables, les fonctions ou même le programme en mémoire…

Il existe plusieurs formats de fichiers objets, ces fichiers contenant davantage que du code machine. L'exemple ci-dessous illustre le passage d'un fichier texte à un fichier objet pour un fichier objet générique :

Toto.txt	Toto.o	Commentaires

int IntGlobal;
extern char CharGlobalExt;
int FunExterne(int Input);
 	
void FunToto(void)
{
IntGlobal = CharGlobalExt + 4;
IntGlobal = FunExterne(12);
}
	Liste des symboles	

	$IntGlobal déclaré içi	
	$CharGlobalExt requis	
	$FunExterne requis	
	$FunToto déclarée içi	
	Code de Toto.o, FunToto	

	PUSH R0, R1, R2
	MOV R0, 4
	LDR.B R1, [$CharGlobalEXT]
	ADD R2, R1, R0
	STR.W R2, [$IntGlobal]
	PUSH 12
	BL $FunExterne
	POP R2
	STR.W R2, [$IntGlobal]
	POP R0, R1, R2
		Fonction propre!
		
		LDR.B pour lire un BYTE/char
		
		STR.W pour écrire un WORD/int
		Passage de paramètres par la pile
		Appel de fonction FunExterne
		Passage de paramètres par la pile
		STR.W pour écrire un WORD/int
		Fonction propre!

Le compilateur doit savoir la nature des variables et fonctions pour utiliser les instructions appropriées, mais c'est l'éditeur de liens qui détermine les adresses.

2.2.4 Optimisations du compilateur

Nom de l'optimisation	Description	Exemple de situation	Après application
Haut-niveau	*Près du code source, inter procédural*		
Intégration de procédure	Remplacer des procédures par leur contenu	Main : SetMaVariable(4); void SetMaVariable(int Valeur) MaVariable = Valeur;	MaVariable = 4;
Local	*Au niveau d'un bloc de base*		
Elimination de sous-expressions communes	Remplacer deux instances du même calcul par une seule copie	a1 = b+c+d+e+f+g; a2 = b+c+d+e+f+g+h;	a1 = b+c+d+e+f+g; a2 = a1+h;

Propagation de constante	Remplacer toutes les instances d'une variable à laquelle est assignée une constante par la constante	a = 3;
		b = c+a;
		d = e+a;
			a = 3;
			b = c+3;
			d = e+3;

Réduction de la profondeur de pile	Réarranger une expression pour minimiser les ressources nécessaires à son évaluation	a = b+c+d+e+f+g;	a = (b+c)+(d+e)+(f+g);
Globale	*À travers des branchement*		
Elimination globale de sous-expressions communes	Comme local, mais à travers des branchements		
Propagation de copies	Remplacer toutes les instances d'une variable à laquelle a été assignée une expression par l'expression.	a = x;b = c+a;d = e+a; a = b + 4; a = a + 4;	b = c+x;d = e+x; a = b+ 8

Déplacement de code	Ôter du code d'une boucle qui est calculé à chaque itération	for(i=0;i<100;i++)
		{x[i] = i;
		if(i ==3) x[i] = 21;}
			for(i=0;i<100;i++)
			{x[i] = i;}
			x[3] = 21;

Élimination de variable d'induction	Simplifier/éliminer les calculs d'adressage des tableaux dans les boucles	for(i=0;i<100;i++) {x[i+1] = a;}	for(i=1;i<101;i++) {x[i] = a;}
Dépendant du processeur	*Au niveau d'un bloc de base*		

Réduction de puissance	Remplacer des expressions complexes par des instructions plus simples ayant le même résultat	x = y/32; x = y%8;	x = y >> 5; x = y & 0x07;

Planification de pipeline	Ré-arranger les instructions pour améliorer les performances du pipeline	a = b + c;
		d = a + e;
		g = h + i;
			a = b + c;
			g = h + i;
			d = a + e;

Optimisation des distances de saut	Choisir le saut le plus court pour simplifier l'instruction de saut	LJUMP Madestination	JUMP Madestination

Optimisation du code avec les instructions spéciales du microprocesseur	Remplacer une ou plusieurs instructions par une instruction plus efficace, spécifique au microcontrôleur	LD R1, [100]
		LD R2, [104]
		LD R3. [108]
			LDM R1-R3, [100]-[108]

Dans la plupart des optimisations de compilateur, il y souvent possibilité d'échanger de la mémoire contre de la vitesse d'exécution ou de la vitesse d'exécution contre de la mémoire. Un exemple de cette idée est le déroulement de boucle : une boucle déroulée s'exécutera plus rapidement qu'une boucle non-déroulée, mais elle prendra beaucoup plus d'espace de code.

Les optimisations de compilateur sont toujours à faire avec soin. Il est recommandé de travailler avec un code non-optimisé, tester/déverminer, optimiser au besoin, puis tester de nouveau:

- Un code optimisé est plus difficile à déverminer parce qu'il n'est plus identique au code écrit par le programmeur. Par exemple, si vous exécutez le code pas à pas, des lignes seront ignorées ou des sauts inattendus surviendront.
- L'optimisation d'un code peut produire des erreurs, soit parce qu'elles existaient déjà (mais elles étaient cachées), soit parce que le programmeur n'a pas fourni les informations pertinentes à l'optimisation (exemple: variable volatile), soit parce que le compilateur a fait une erreur (ce qui est rarement le cas, mais…).
- Les erreurs causées par l'optimisation peuvent survenir à plusieurs niveaux : haut-niveau, local, global. Le compilateur optimise parfois la mémoire et la vitesse d'exécution en retirant du code inutile dans une fonction, des variables "inutiles", des fonctions inutiles voire même des modules inutiles…

Le terme "volatile" est appliqué à une variable pour indiquer au compilateur que cette variable peut être modifiée dans un autre contexte, et, de ce fait, ne doit pas être optmisée. Voici un exemple où CountS est décrémenté dans une interruption de timer:

Code de base (Cdb)	Cdb optimisé (incorrect)	Code de base modifié, optimisé correct

Int CountS;
 	
char AttendReponse() {
char RepRecue = 0;
CountS = 10;
while(!ReponseRecue && (CountS>0)) {
RepRecue = LitReponse();
}
…}
	int CountS;
	
	char AttendReponse() {
	CountS = 10;
	char RepRecue = 0;
	while(!ReponseRecue) {
	RepRecue = LitReponse();
	}
	…}
		volatile int CountS;
		
		char AttendReponse() {
		CountS = 10;
		char RepRecue = 0;
		while(!ReponseRecue && (CountS>0)) {
		RepRecue = LitReponse();
		}
		…}

L'optimisation de code la plus efficace est toujours celle faites par le programmeur lui-même! *Si vous manquez d'espace ou de temps de calcul, la première étape devrait être de revoir votre code. Ensuite, si vous ne voyez pas de gain évident, essayez avec le compilateur… Le compilateur est plus méthodique que vous, mais beaucoup moins intelligent !*

3 Édition de liens et exécutable

L'éditeur de liens relie ensembles les fichiers objets : il fait le lien entre les déclarations de variables/fonctions globales d'un fichier et les définitions dans les autres fichiers. Il relie aussi les déclarations des fichiers aux librairies dynamiques.

L'éditeur de liens dispose aussi les variables et les fonctions du programme dans l'espace mémoire du programme. Il génère un map file indiquant l'emplacement de chaque composante du système.

> *Le map file généré par l'éditeur de liens contient des informations très utiles. On y retrouve la quantité d'octets utilisés, les tailles et les adresses des fonctions/variables et plusieurs autres données pertinentes.*

Enfin, l'éditeur de liens génère le fichier exécutable. Le fichier exécutable est plus qu'un ensemble d'instructions en binaire : il contient souvent des informations nécessaires pour l'exécution/relocalisation de l'exécutable par le système d'exploitation ou des informations nécessaires afin de programmer convenablement la mémoire d'instructions du microprocesseur visé.

> *Il y a plusieurs formats de fichiers exécutables. Si le temps le permet, des formats seront présentés en exemple. En fait, le format ELF, tiré des notes de cours de systèmes d'exploitation (IFT-2001) sera présenté.*

3.1 Disposition des variables dans la mémoire

L'éditeur de liens et le compilateur peuvent choisir quelle adresse de mémoire attribuer à une variable en fonction de sa nature et de paramètres donnés par le programmeur.

Les variables locales, par exemple, peuvent être compilées comme étant des registres. Il est aussi possible de réserver de l'espace sur la pile pour toutes les variables locales (c'est une stratégie très courante). Enfin, on pourrait, même si cela se fait qu'exceptionnellement (c'est une mauvaise pratique pour les fonctions réentrantes ou récursives), déclarer toutes les variables locales comme étant globales.

Les variables globales, elles, peuvent être disposées de plusieurs façons dans la mémoire de données. Elles peuvent être: regroupées par module, regroupées par segments de mémoire définis par le programmeur, regroupées par ordre d'utilisation dans le code…

Il y a généralement deux segments de mémoire alloués pour les variables: un segment pour les variables à initialiser et un segment pour les variables qui ne doivent pas être initialisé automatiquement au début du programme. Vous devez éviter d'initialiser les variables dans votre EEPROM comme s'il s'agissait de variables en RAM (cela créera des exceptions!!!) en déclarant vos variables conséquemment. Vous retrouverez des déclaration du genre "__no_init unsigned char MaVariable"...

Plus de détails sur la disposition des variables et fonctions dans la mémoire sera vue dans le cours Mémoire-Logiciel.

3.1.1 Directives à l'éditeur de liens pour placer les variables

Il existe souvent des options pour déterminer comment seront compilées et disposées les variables dans le code.

Par exemple, il existe une directive de compilation/édition de liens qui indique à l'éditeur de liens dans quel segment de mémoire placer une variable ou une fonction. Cette directive dépend du compilateur utilisé (exemple pour CCS : #pragma DATA_SECTION (NomVariable, "NomDuSegment").

Habituellement, les segments de mémoire sont définis dans un fichier de commande (.cmd dans le cas de CCS) pour l'éditeur de liens.

4 Environnement de développement intégré

Avant les premiers IDEs, le développement de code se faisait avec des logiciels spécialisés. D'autres logiciels permettaient de compiler et de construire les programmes. De plus, un troisième type de logiciels permettait de programmer la mémoire d'instruction. Enfin, des logiciels spécialisés, simulateurs ou débugger temps réel, permettaient de déverminer efficacement les applications. Les IDEs intègrent toutes ces fonctionnalités.

4.1 Support pour l'écriture de code

L'IDE propose habituellement une interface graphique permettant d'éditer efficacement les fichiers textes d'un programme. L'IDE offre souvent des fonctionnalités similaires à celles d'un traitement de texte, mais avec quelques différences :

- L'IDE permet de gérer l'ensemble des fichiers appartenant à un projet.
- L'IDE fournit des outils de recherche et de navigation dans le code adapté au langage de programmation. Il permet, par exemple, de chercher un mot à travers tous les fichiers d'un projet ou d'aller à la définition d'un symbole.
- L'IDE propose automatiquement des tabulations et espaces adaptés au langage de programmation (gestion de l'indentation).
- L'IDE permet habituellement de commenter des blocs de code
- L'IDE permet parfois de compléter automatiquement des mots.
- L'IDE offre des marque-pages (bookmarks) pour identifier des lignes de code
- L'IDE peut souligner ou mettre en évidence le code modifié ou les ajouts
- …

4.2 Construction d'application : Makefile et IDE

4.2.1 Makefile

Un makefile est un fichier texte qui décrit comment construire une application.

Les makefiles contiennent des séquences de commandes (des lignes de textes) exécutées en lot. Par exemple, un makefile qui construit le programme abc.exe à partir des fichiers a.txt, b.txt et c.txt contiendra au moins trois commandes pour compiler a.txt, b.txt et c.txt. Il contiendra aussi au moins une commande pour relier a.obj, b.obj et c.obj ensemble afin de produire abc.exe.

Dans le meilleur des cas, l'édition des makefiles est pénible. Il faut connaître très bien chaque commande et choisir les paramètres/options qui l'accompagneront. Par ailleurs, les interactions entre plusieurs makefiles d'un projet peuvent devenir rapidement complexes.

4.2.2 Construction d'application dans l'IDE

Heureusement, les IDEs créent automatiquement les makefiles. À partir du projet et des options du projet, un IDE créera un makefile (ou équivalent) et l'exécutera lorsque vous demanderez de compiler. Voici diverses options communes que l'on retrouve dans l'IDE qui permettent de gérer la compilation et l'édition de liens. Ces options sont habituellement des paramètres pour les commandes du makefile. Elles sont généralement disponibles en faisant un right-click sur le projet:

- Choix du groupe d'options (target)
    - Les options qui suivent sont toutes reliées à une cible. Si la cible change, les options changes. Habituellement, il existe les cibles "Debug" et "Release"
- Choix du microprocesseur visé
- Choix du chemin et du nom de l'exécutable produit.
- Activer ou désactiver la génération de fichiers de liste (code assembleur correspondant au C ou autre)
- Chemin des "#include"
    - Cette option dit où trouver (dans quel répertoire) un fichier à inclure
- Symboles prédéfinis
    - il est possible de mettre des #define dans les fichiers textes, mais ils n'auront qu'une portée locale, c'est-à-dire que le symbole défini n'existera que pour le fichier. Définir un symbole dans le projet permet de le définir pour tous les fichiers du projet.
- Niveau d'optimisation du code
    - Voir la section sur l'optimisation du code.
- Actions à effectuer avant le build et après
    - Très utile pour insérer des numéros de version (SVN par exemple) dans un code compilé ou pour ajouter des CRCs/Checksum dans le code compilé.
- Format de l'exécutable produit
- Paramètres pour l'agencement du code et des variables en mémoire
    - Taille de la mémoire (instructions et données)
    - Taille de la pile (stack) et du monceau (heap)
    - Définition de segments de mémoire pour y mettre des items précis
- Activer ou désactiver la génération du fichier de memory map

- Options pour la compilation et la localisation des variables locales, statiques, globales.
- …

4.3 Programmation de la mémoire du SMI et Déverminage de code embarqué

Les IDEs offrent plusieurs options pour la programmation de la mémoire de SMI et pour le déverminage de code. Comme ces options dépendent très souvent de la nature du système avec mémoire ou de liens de communication avec le système, seul le JTAG sera présenté ici. Le JTAG permet à la fois de programmer la mémoire d'instruction et de déverminer une application dans celle-ci.

4.3.1 Déverminage d'application dans les SMI

Depuis les tout premiers microprocesseurs, des circuits logiques à l'intérieur même du microprocesseur permettent de déverminer les applications. Ces circuits permettent d'arrêter l'exécution d'instruction (break point) et, par la suite, de lire (voire d'écrire), les valeurs des registres du microprocesseur tout comme la mémoire du système.

Le circuit de debug permet aussi de tester le microprocesseur, il a des interruptions (trap) qui lui sont propres, parfois des broches dédiées et il fait partie intégrante du microprocesseur. Pour accéder au circuit de debug, plusieurs interfaces peuvent être utilisées : le port série (RS232), le port parallèle, …, et le JTAG.

4.3.2 JTAG

Les éléments de http://en.wikipedia.org/wiki/Joint_Test_Action_Group seront présentés en classe. Le JTAG est une interface commune pour accéder aux fonctions de debug des microcontrôleurs.