Chapitre 3

Programmation en assembleur

3.1. Introduction à la programmation en assembleur

La programmation en langage assembleur constitue une étape fondamentale dans la maîtrise des microcontrôleurs. Contrairement aux langages de haut niveau qui masquent les détails matériels, l'assembleur offre une vision transparente du fonctionnement interne du processeur. Chaque instruction assembleur correspond directement à une instruction machine exécutable par le microcontrôleur, ce qui permet au programmeur de contrôler finement les ressources matérielles. Cette approche est particulièrement précieuse dans les systèmes embarqués où les contraintes de mémoire et de temps d'exécution sont souvent sévères. L'apprentissage de l'assembleur permet non seulement de développer des programmes efficaces, mais aussi d'acquérir une compréhension profonde de l'architecture du microcontrôleur, compréhension qui reste utile même lorsqu'on utilise ultérieurement des langages de plus haut niveau comme le C.

3.2. Structure d'un programme assembleur

Tout programme assembleur suit une organisation rigoureuse qui reflète l'architecture du microcontrôleur cible. La structure typique se divise en plusieurs sections distinctes. La section de code contient l'ensemble des instructions exécutables, organisées sous forme de routines et de fonctions. La section de données rassemble les variables et constantes, avec une distinction claire entre les données initialisées et non initialisées. Une section spécifique est dédiée à la gestion de la pile, qui sert à sauvegarder le contexte lors des appels de sous-programmes et à gérer les variables locales. Les directives d'assemblage, telles que celles définissant le point d'entrée du programme ou l'inclusion de fichiers, complètent cette structure. Les étudiants apprennent à organiser leur code de manière modulaire, en séparant les différentes fonctionnalités dans des fichiers distincts, ce qui facilite la maintenance et la réutilisation du code.

3.3. Gestion de la mémoire et de la pile

La maîtrise de la gestion mémoire est essentielle en programmation assembleur. La mémoire vive (RAM) est divisée en plusieurs zones : la zone des variables globales, la zone de pile et éventuellement la zone de tas. La pile, dont le pointeur est géré par un registre dédié, s'utilise selon un principe premier entré, dernier sorti. Les étudiants apprennent à manipuler explicitement la pile pour sauvegarder le contenu des registres avant un appel de sous-programme, pour transmettre des paramètres et pour stocker des variables locales. Une attention particulière est portée à la gestion des débordements de pile, qui constituent une source fréquente d'erreurs difficiles à diagnostiquer. La compréhension du modèle mémoire

permet également d'optimiser l'utilisation de l'espace RAM, souvent limité sur les microcontrôleurs bas de gamme.

3.4. Exemples progressifs : manipulation de ports et opérations arithmétiques

L'apprentissage pratique débute par des exemples simples qui mettent en œuvre les concepts fondamentaux. Le premier exemple type consiste à manipuler les ports d'entrées-sorties : configuration des registres de direction, écriture de valeurs sur les broches de sortie et lecture d'états sur les broches d'entrée. Ces exercices permettent de se familiariser avec l'adressage mémoire des registres spéciaux et avec les instructions de déplacement de données. Les exemples suivants introduisent les opérations arithmétiques de base (addition, soustraction) et logiques (ET, OU, XOR), en montrant comment les résultats sont stockés dans les registres et comment les indicateurs (flags) du registre d'état sont affectés. La gestion des boucles est abordée à travers des exemples de temporisations simples basées sur des décrémentations répétées, illustrant l'utilisation des instructions de saut conditionnel et inconditionnel.

3.5. Exemples progressifs : structures de contrôle et sous-programmes

Au-delà des séquences linéaires, les étudiants apprennent à implémenter les structures de contrôle fondamentales. Les instructions conditionnelles sont réalisées à l'aide d'instructions de comparaison suivies de sauts conditionnels, permettant de construire des structures if-then-else. Les boucles for et while sont implémentées par des combinaisons de compteurs, d'incrémentations et de sauts conditionnels vers l'arrière. Les sous-programmes sont introduits comme un moyen de structurer le code et d'éviter les duplications. Les étudiants maîtrisent les instructions d'appel et de retour, ainsi que les conventions de passage de paramètres (via registres ou via pile) et de sauvegarde du contexte. Des exemples concrets, tels que le clignotement d'une LED avec une temporisation paramétrable ou la gestion d'un afficheur 7 segments, permettent de mettre en œuvre ces concepts dans des applications réalistes.

3.6. Optimisation du code

L'optimisation du code constitue un aspect crucial de la programmation assembleur, directement lié aux contraintes des systèmes embarqués. Les étudiants apprennent à évaluer et à améliorer leurs programmes selon deux axes principaux : l'optimisation en taille et l'optimisation en vitesse. L'optimisation en taille vise à réduire l'empreinte mémoire du programme, ce qui permet d'utiliser des microcontrôleurs moins coûteux ou de libérer de l'espace pour des fonctionnalités supplémentaires. Elle repose sur des techniques telles que le choix d'instructions plus compactes (par exemple, utiliser une instruction incrémentale plutôt qu'une addition générique), la factorisation de code identique en sous-programmes réutilisables, ou encore l'utilisation de structures de données adaptées. L'optimisation en vitesse vise à réduire le temps d'exécution, essentielle pour les applications temps réel. Elle passe par des choix tels que l'utilisation de registres plutôt que de mémoire RAM, le déroulage de boucles (loop unrolling) pour réduire le nombre de sauts, ou l'exploitation judicieuse des modes d'adressage les plus rapides.

3.7. Compromis et bonnes pratiques

Les étudiants sont amenés à comprendre que l'optimisation implique souvent des compromis entre taille et vitesse, entre lisibilité et performance. Une bonne pratique consiste à commencer par un code clair et correct, puis à optimiser uniquement les parties critiques identifiées par une analyse des performances. L'utilisation de commentaires précis est particulièrement importante en assembleur, où le code peut rapidement devenir difficile à comprendre sans documentation adéquate. Les étudiants apprennent à documenter systématiquement l'objectif de chaque routine, les conventions d'utilisation des registres et les points critiques de leur code. Enfin, une introduction est faite aux outils d'analyse qui permettent de mesurer la taille du code généré et d'estimer les temps d'exécution, fournissant ainsi des retours objectifs pour guider les efforts d'optimisation.

3.8. Synthèse et perspectives

En conclusion de ce chapitre, les étudiants possèdent les compétences de base leur permettant d'écrire des programmes assembleur fonctionnels, structurés et optimisés. Ils comprennent le lien direct entre leur code et l'exécution matérielle, ce qui leur donne une longueur d'avance pour aborder les chapitres suivants consacrés aux périphériques avancés (interruptions, timers) et aux interfaces de communication. Ces compétences en assembleur constituent également un atout précieux pour la lecture et l'optimisation du code généré par les compilateurs C, ainsi que pour le débogage fin lors des phases de mise au point. L'approche progressive adoptée, allant des concepts fondamentaux aux techniques d'optimisation avancées, permet à chaque étudiant de progresser à son rythme tout en acquérant une maîtrise solide de la programmation bas niveau.