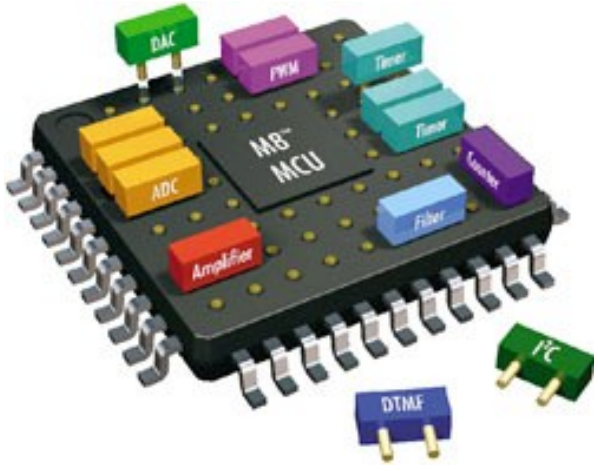
Architecture et programmation des systèmes embarqués sous le langage C

Informatique embarquée

1ère année master RSD

Dr: K.BARKA



1. Concepts de base sur le hardware d'un système embarqué

- **La programmation des systèmes embarqués impose aux programmeurs la maitrise du milieu physique du système et de se familiariser avec les différents composants électroniques.**

- **Ainsi, le programmeur doit pouvoir facilement identifier les éléments physiques d'un système embarqué et de les contrôler.**

- **Dans cette première partie, nous allons introduire quelques notions de base sur les composants électroniques, donner des exemples du hardware embarqué et aborder des concepts importants pour la programmation embarquée.**

1.1 Le schéma de flux de données

- **Afin d'écrire un programme dédié à un système embarqué, il faut d'abord avoir une idée sur le matériel sur lequel on va travailler.**

- **D'abord, on doit connaitre les fonctions de base du système à travers les services offerts par sa carte mère, et identifier ensuite le type de son processeur, ses mémoires de stockage et ses périphériques d'entrée et de sortie.**

- **En général, tous les dispositifs embarqués sont fournis avec un guide de fonctionnement qui facilite l'identification du matériel.**

- **Le guide de fonctionnement permet aussi de savoir comment les flux de données circulent entre les différents composants du système.**

- **Cela va permettre d'établir un schéma de flux de données qui définit l'interconnexion des composants à travers les bus de données et d'adresses.**

1.1 Le schéma de flux de données

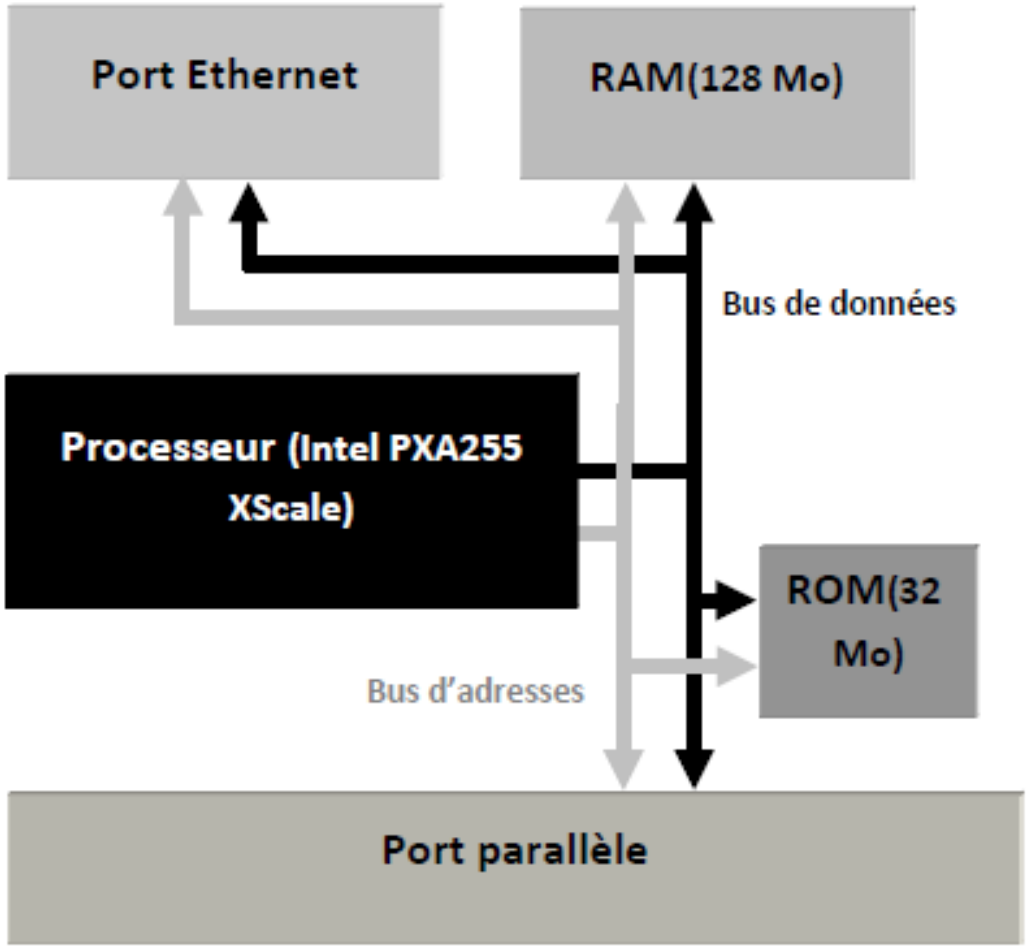


Schéma du diagramme de flux

1.1 Le schéma de flux de données

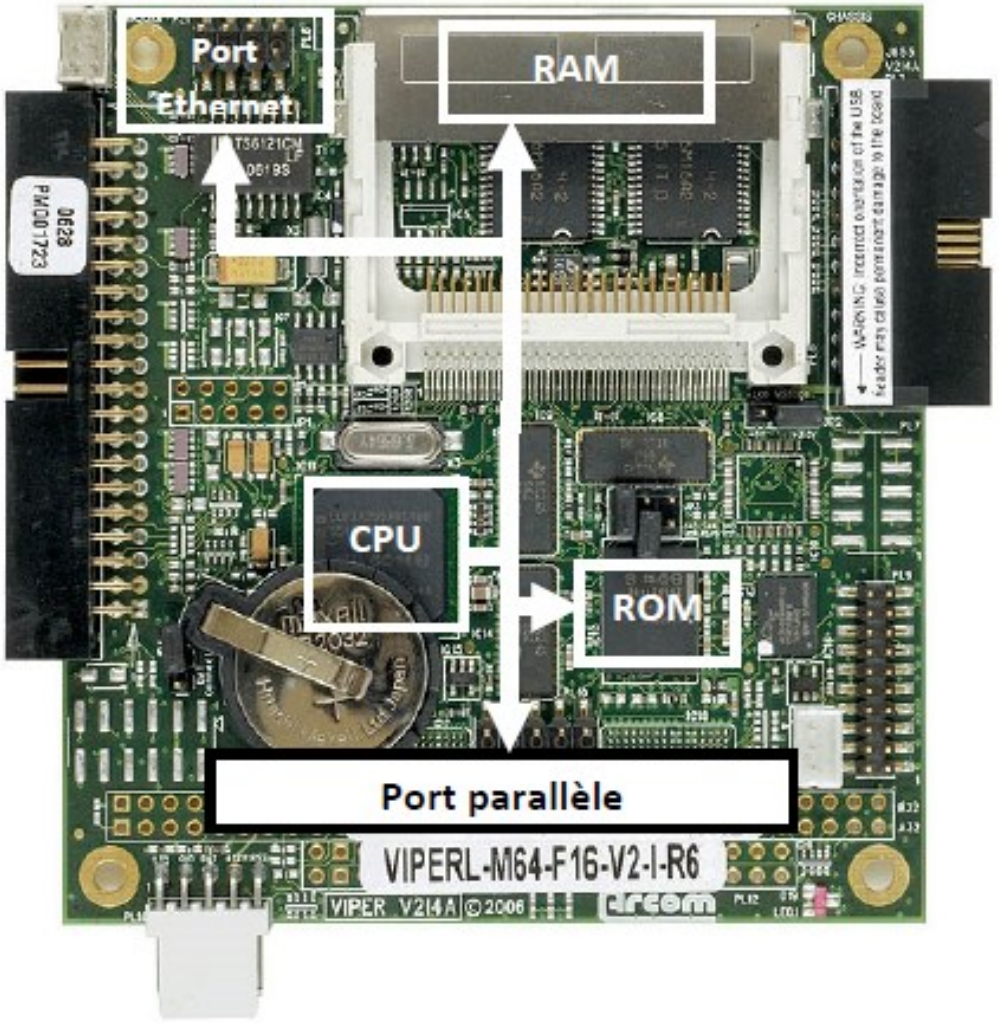
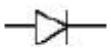
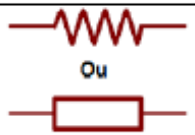
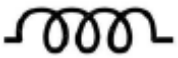
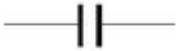
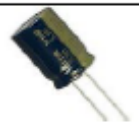


Schéma du diagramme de flux sur une carte mère Arcom VIPER-Lite

1.2 Le schéma des composants électroniques

Composant	Préfixe	Symbole	Exemple réel
Diode simple	D		
Diode lumineux	D		
Résistance	R		
Inducteur	L		
Condensateur	C		
Circuit de Crystal	X		
Source d'énergie	VCC		

Symboles des composants électroniques de base

1.3 Interconnexions microprocesseur, mémoire et périphériques

- **Le microprocesseur doit communiquer avec d'autres composants tel que la mémoire et les périphériques d'entrée et de sortie.**

- **Pour communiquer, le processeur leur attribue des adresses uniques.**

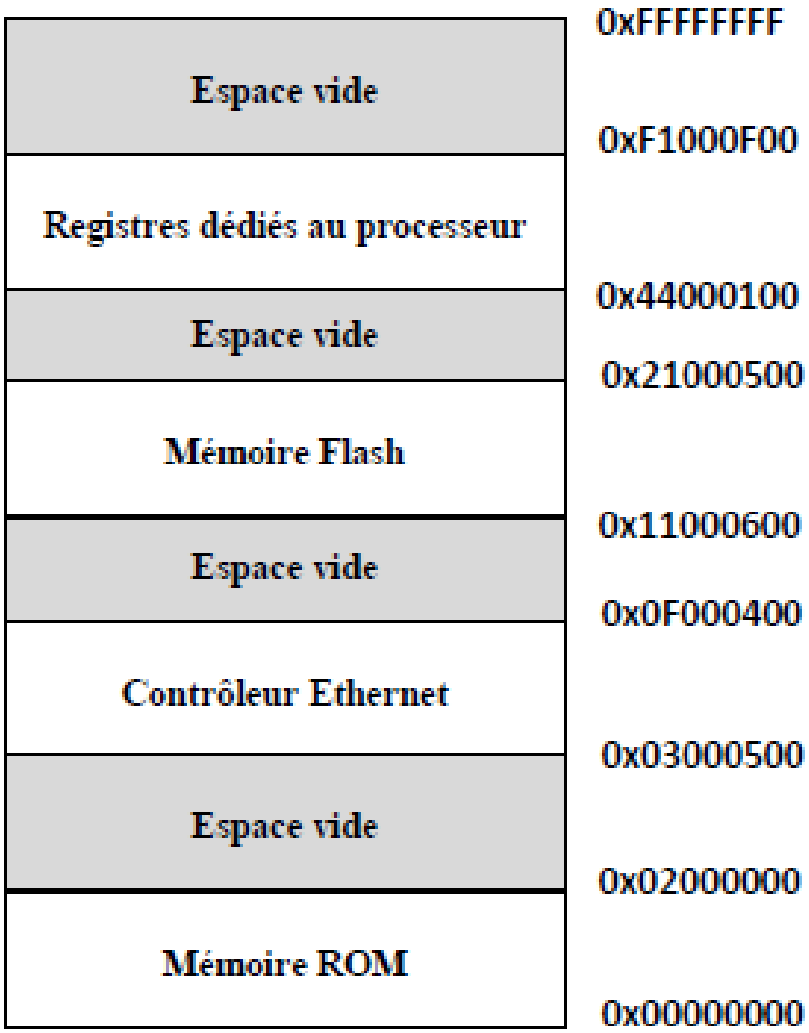
- **Ces derniers seront sauvegardés dans des registres internes au processeur ou dans un espace réservé dans la mémoire externe.**

- **Dans ce cas on parle de périphériques mappés dans la mémoire.**

- **Ce procédé permet de réduire la taille du processeur en éliminant l'espace mémoire dédié pour l'adressage des périphériques.**

- **De plus, la mémoire mappée offre plus de flexibilité au programmeur des systèmes embarqués et facilite l'interaction avec les périphériques du système.**

1.3 Interconnexions microprocesseur, mémoire et périphériques



Exemple d'une mémoire mappée

1.3 Interconnexions microprocesseur, mémoire et périphériques

- **La mémoire de stockage peut être présente sur le même circuit intégré (Ship), ou externe sur un autre support.**

- **Deux ensembles de fils électriques relient la mémoire au microprocesseur, sont connus sous le nom de bus de données et d'adresses.**

- **Pour lire ou écrire dans un emplacement spécifique de la mémoire, le processeur écrit l'adresse de ce dernier sur le bus d'adresses.**

- **Un circuit logique (décodeur) interprète l'adresse envoyée par le processeur et sélectionne l'emplacement désiré dans la mémoire.**

- **Ensuite, les données seront envoyées au processeur à travers le bus de données.**

1.4. Fichier entête pour une mémoire mappée

- **L'une des étapes essentielles dans le développent d'un logiciel embarqué est de concevoir un fichier entête pour les composants mappés dans la mémoire principale.**

- **Ce dernier consiste à définir des variables avec des valeurs constantes qui représentent les adresses des composants embarqués dans la mémoire.**

- **Le fichier entête permet de donner plus d'abstraction sur le matériel en adressant les composants électroniques à travers leurs identificateurs (nom du périphérique) plutôt que par leurs adresses.**

- **Le fichier entête permet aussi la réutilisation du code et sa portabilité, étant donné que le changement d'un composant nécessite uniquement la mise à jour d'une variable dans le fichier entête.**

1.4 Fichier entête pour une mémoire mappée

```
/*****************************************************************************
 *
 *  Exemple d'un fichier entête d'une mémoire mappée
 *
 *            Addresse      Taille  Description
 *            ------------- ----- --------------------------------
 *            0x00000000     64M   Mémoire ROM
 *            0x08000300     N/A   Controlleur Ethernet
 *            0x50000000     16M   Mémoire Flash
*
 *****************************************************************************/

#defineROM                          (0x00000000)
#define ETHERNET                     (0x03000500)
#define FLASH                     (0x11000600)
```

Exemple d'un fichier entête d'une mémoire mappée

1.5 Etude d'un processeur embarqué

Le processeur PXA255

- **Le processeur PXA255 est muni de:**

  - **Une unité de calcul Xscale (même architecture d'ARM).**
  - **Une unité pour le contrôle d'interruptions**
  - **Un contrôleur de mémoire**
  - **Quatre compteurs,**
  - **Un bus d'interface**
  - **Un contrôleur LCD,**
  - **Une horloge,**
  - **Une unité de gestion d'énergie.**

- **Pour communiquer avec les composants internes le processeur utilise plusieurs bus internes.**

1.5 Etude d'un processeur embarqué

- **Le programmeur peut contrôler les composants associés au processeur PXA255 à travers la lecture ou l'écriture dans les registres dédiés à ces composants.**

- **Dans le cas de mémoire mappée, les registres sont positionnés à des endroits spécifiques dans la mémoire centrale.**

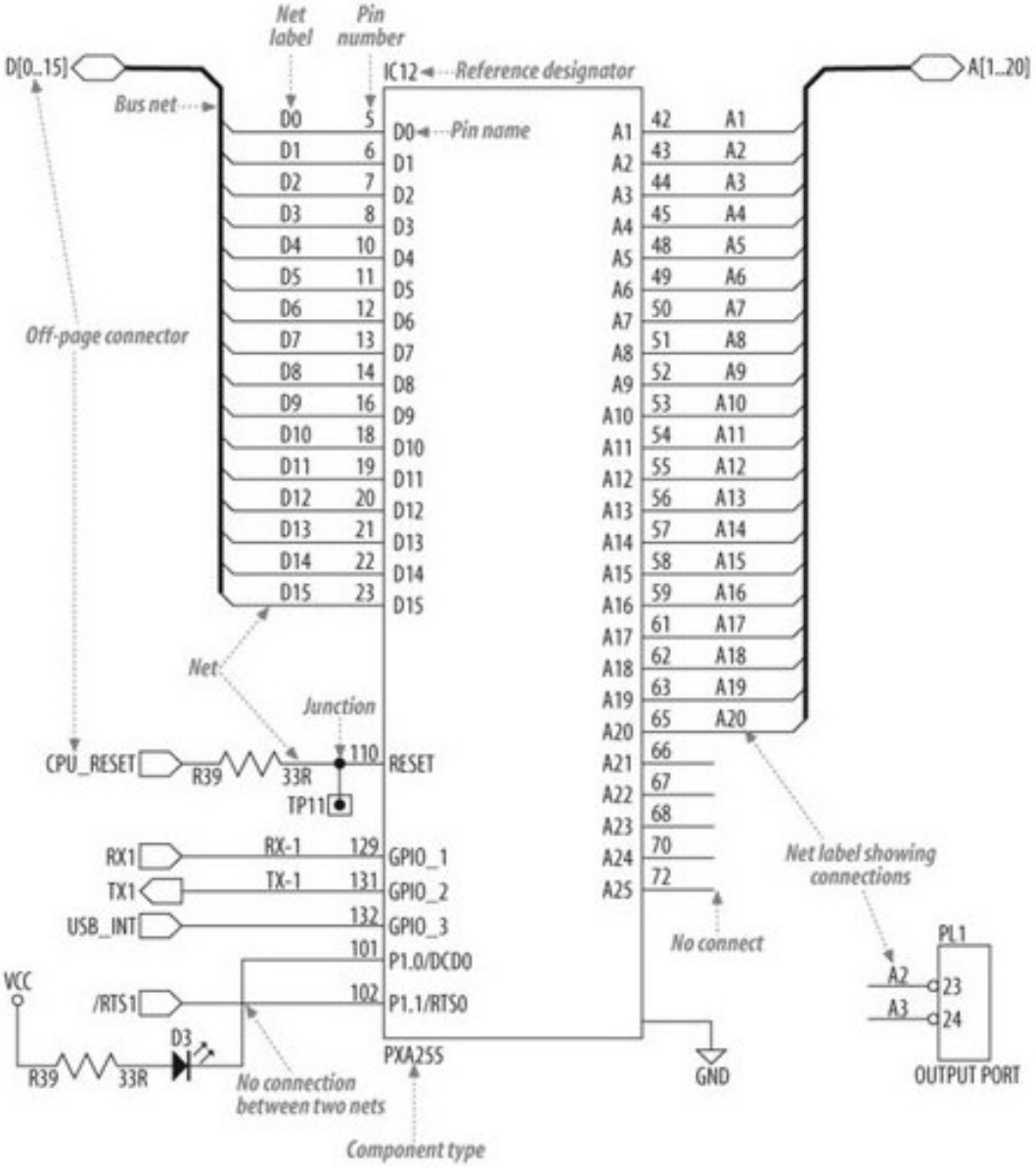
- **Pour obtenir les adresses de ces registres le programmeur doit consulter le manuel de développement fournit avec le processeur PCA255.**

- **Comme mentionné précédemment, avant de commencer de programmer il est préférable de définir un fichier entête qui contient les identificateurs des registres associés à leurs adresses.**

- **Cela permet de clarifier et simplifier les éventuels codes sources et favoriser leurs portabilités et leur réutilisation.**

1.5 Etude d'un processeur embarqué

Schéma électronique



1.5 Etude d'un processeur embarqué

```
/*************************************************************
 * PXA255 XScale ARM Processor On-Chip Peripherals
 *************************************************************/

/* Timer Registers */
#define TIMER_0_MATCH_REG           (*((uint32_t volatile *)0x40A00000))
#define TIMER_1_MATCH_REG           (*((uint32_t volatile *)0x40A00004))
#define TIMER_2_MATCH_REG           (*((uint32_t volatile *)0x40A00008))
#define TIMER_3_MATCH_REG           (*((uint32_t volatile *)0x40A0000C))
#define TIMER_COUNT_REG             (*((uint32_t volatile *)0x40A00010))
#define TIMER_STATUS_REG            (*((uint32_t volatile *)0x40A00014))
#define TIMER_INT_ENABLE_REG        (*((uint32_t volatile *)0x40A0001C))

/* Timer Interrupt Enable Register Bit Descriptions */
#define TIMER_0_INTEN               (0x01)
#define TIMER_1_INTEN               (0x02)
#define TIMER_2_INTEN               (0x04)
#define TIMER_3_INTEN               (0x08)

/* Timer Status Register Bit Descriptions */
#define TIMER_0_MATCH               (0x01)
#define TIMER_1_MATCH               (0x02)
#define TIMER_2_MATCH               (0x04)
#define TIMER_3_MATCH               (0x08)

/* Interrupt Controller Registers */
#define INTERRUPT_PENDING_REG       (*((uint32_t volatile *)0x40D00000))
#define INTERRUPT_ENABLE_REG        (*((uint32_t volatile *)0x40D00004))
#define INTERRUPT_TYPE_REG          (*((uint32_t volatile *)0x40D00008))

/* Interrupt Enable Register Bit Descriptions */
#define GPIO_0_ENABLE               (0x00000100)
#define UART_ENABLE                 (0x00400000)
#define TIMER_0_ENABLE              (0x04000000)
#define TIMER_1_ENABLE              (0x08000000)
#define TIMER_2_ENABLE              (0x10000000)
#define TIMER_3_ENABLE              (0x20000000)
```

Fichier entête du processeur PXA255

1.5 Etude d'un processeur embarqué

```
/* General Purpose I/O (GPIO) Registers */
#define GPIO_0_LEVEL_REG            (*((uint32_t volatile *)0x40E00000))
#define GPIO_1_LEVEL_REG            (*((uint32_t volatile *)0x40E00004))
#define GPIO_2_LEVEL_REG            (*((uint32_t volatile *)0x40E00008))
#define GPIO_0_DIRECTION_REG        (*((uint32_t volatile *)0x40E0000C))
#define GPIO_1_DIRECTION_REG        (*((uint32_t volatile *)0x40E00010))
#define GPIO_2_DIRECTION_REG        (*((uint32_t volatile *)0x40E00014))
#define GPIO_0_SET_REG              (*((uint32_t volatile *)0x40E00018))
#define GPIO_1_SET_REG              (*((uint32_t volatile *)0x40E0001C))
#define GPIO_2_SET_REG              (*((uint32_t volatile *)0x40E00020))
#define GPIO_0_CLEAR_REG            (*((uint32_t volatile *)0x40E00024))
#define GPIO_1_CLEAR_REG            (*((uint32_t volatile *)0x40E00028))
#define GPIO_2_CLEAR_REG            (*((uint32_t volatile *)0x40E0002C))
#define GPIO_0_FUNC_LO_REG          (*((uint32_t volatile *)0x40E00054))
#define GPIO_0_FUNC_HI_REG          (*((uint32_t volatile *)0x40E00058))
```

Fichier entête du processeur PXA255

1.5 Etude d'un processeur embarqué

- **Le code source suivant est un exemple de l'utilisation du fichier entête dans un programme en langage C.**

```
if (bLedEnable == TRUE)
{
    GPIO_0_SET_REG = 0x00400000;
}
```

- **l'identificateur GPIO_0_SET_REG désigne un périphérique quelconque relié au pin d'entrée sortie numéro zéro.**

- **L'identificateur bLedEnable désigne le registre relié au contrôleur d'un Led.**

1.6 Initialisation du hardware

- **L'initialisation du hardware est une étape nécessaire pour faciliter la tâche de la programmation des systèmes embarqués.**

- **Cette étape consiste à développer un petit programme pour initialiser les composants embarqués et sera appelé par la suite au début des softwares implémentés pour le système embarqué.**

- **La première phase dans l'écriture d'un programme d'initialisation est d'activer la fonctionnalité d'initialisation du processeur à travers un appel à une routine qui va initialiser ses composants.**

- **Pour cela, le programme d'initialisation affecte l'adresse de la première instruction de la routine d'initialisation au compteur ordinal du processeur.**

- **Dans notre exemple typique (PXA255) l'adresse d'initialisation est égale à 0x00000000.**

1.6 Initialisation du hardware

- **Après l'initialisation du processeur, on doit initialiser les autres composants embarqués.**

- **En effet, la plupart des systèmes embarqués intègrent une routine dédiée pour l'initialisation du hardware.**

- **Par conséquence, le programme d'initialisation doit faire appel à cette routine à travers son adresse dans la mémoire.**

- **Enfin, la dernière étape d'initialisation consiste à appeler le programme de démarrage qui représente le software développé pour le système embarqué.**

- **Cela peut être effectué par un simple appel de la fonction principale du software (fonction main dans le langage C).**

2. Premier programme embarqué avec le langage C

- **Les systèmes embarqués sont très divers et il est difficile de proposer un exemple de programme embarqué adapté à tous ces derniers.**

- **Cependant, la plupart des systèmes embarqués partagent un composant commun qui peut être contrôlé avec un simple programme embarqué.**

- **Ce dernier est un diode lumineux généralement connu sous le nom Led.**

- **En effet, ce composant est habituellement connecté au processeur du système embarqué à travers un pin d'entrée sortie d'usage général nommé GIOP (General purpos In Out Pin).**

- **Par conséquence, notre programme exemple consiste à contrôler un Led dans un système embarqué en l'allumant et l'éteignant plusieurs fois selon une fréquence donnée.**

2. Premier programme embarqué avec le langage C

- **La première étape avant d'écrire notre programme est d'identifier d'abord le Led à contrôler pour connaitre comment il est connecté au processeur.**

- **Par exemple, dans une carte mère Arcom Viper IO il existe trois Led de couleurs différentes nommés respectivement LED0, LED1 et LED2.**

- **Le manuel fournit avec la carte mère Arcom Viper IO décrit comment les Leds sont connectés au processeur.**

- **Ainsi, on constate dans le manuel que le LED2 est contrôlé par un signal inversé (NON logique) dont le préfixe est OUT2, ce qui signifie que le led s'allumera si le signal est à un niveau bas et s'éteindra dans le cas contraire.**

- **On constate aussi que le signal OUT2 est contrôlé par le pin numéro 22 du processeur qui s'identifie par le préfixe GIOP22.**

- **Par conséquence, dans notre programme exemple on doit modifier la valeur du sortie du pin GIOP22 pour faire clignoter le Led numéro 2.**

2. Premier programme embarqué avec le langage C

Carte mère ArcomViper IO

2. Premier programme embarqué avec le langage C

```
#include "led.h"

/*****************************************************************
 * Fonction principale pour le contrôle du clignotement du led LED2
 * sur une carte mère Arcom Viper IO
*****************************************************************/
int main(void)
{
Initialisation();/* Initialisation de l'état du led. */

while (1)
    {
Clignoter();/* Faire clignoter le LED 2. */

Pause(500);/* Faire un pause de 500 millisecondes. */
    }
return 0;
}
```

Fonction principale pour le contrôle du clignotement d'un Led

2. Premier programme embarqué avec le langage C

- **Avant de détailler les trois fonctions appelées par notre fonction principale 'main', il faut bien comprendre l'interconnexion entre le led et le processeur et comment ce dernier peut contrôler le diode lumineux.**

- **Comme décrit précédemment, le pin d'usage général numéro 22 du processeur permet de contrôler le Led de couleur verte (troisième led sur la carte mère ArcomViper IO).**

- **En effet, tous les pins GIOP du processeur PXA255 (processeur intégré dans la carte mère ArcomViper IO) sont gérés par plusieurs registres internes au processeur.**

- **Ces derniers ont 32 bits de taille dont chaque bit est dédié pour un pin spécifique du processeur.**

2. Premier programme embarqué avec le langage C

Préfixe	Adresse	Lecture/Ecriture	Fonction	Valeur du bit
GPDR0	0x40E0000C	Lecture/Ecriture	Permet de modifier le type du pin pour le rendre comme entrée ou sortie.	0 = Entrée 1 = Sortie
GPLR0	0x40E00000	Lecture seule	Affiche l'état du pin	0 = Signal faible 1 = Signal élevé
GPSR0	0x40E00018	Ecriture seule	Modifier la valeur du signal sortant en se basant sur la valeur du registre GPDR0	Si GPDR0 = 1 1 = Signal élevé 0 = Aucune action
GPCR0	0x40E00024	Ecritureseule	Modifier la valeur du signal sortant en se basant sur la valeur du registre GPDR0	Si GPDR0 = 1 1 = Signal faible 0 = Aucune action
GAFR0_U	0x40E00058	Lecture/Ecriture	Modifier la fonctionnalité des pines GIOP	0 = Entrée/Sortie 1 = Fonction alternative 1

Registres du processeur PXA255 dédiés aux pins GIOP

2. Premier programme embarqué avec le langage C

- **En consultant le manuel de programmation fournit avec le processeur PXA255, on peut identifier le numéro de bit dédié pour le contrôle du led vert (LED2) dans les registres présentés dans le tableau précédent.**

- **Ce dernier est le bit numéro 22 qui correspond au même numéro de pin dans le processeur PXA255.**

- **Les registres GIOP sont mappés dans la mémoire intégrée avec la carte mère ArcomViper IO, les adresses exposées dans le tableau précédent présentent leur emplacement dans la mémoire.**

- **Cela permet au programmeur avec langage de haut niveau d'accéder facilement aux registres et contrôler directement les périphériques reliés à ces derniers.**

- **Dans le cas où les registres de contrôle ne sont pas mappés, on ne peut les contrôler qu'avec des instructions écrites en langage assembleur (c'est le cas de registre RESET dédié pour l'initialisation de l'état du processeur).**

2.1 Initialisation des registres de contrôle

- **Avant d'écrire la fonction qui contrôle le clignotement du led vert, il faut d'abord écrire une fonction qui va initialiser l'état des registres qui contrôlent ce dernier.**

- **A sa configuration par défaut tous les pins GIOP du processeur PXA255 sont configurés comme des entrées.**

- **Dans notre cas nous devons modifier le type pin d'entrée sortie numéro 22 pour le rendre une sortie, à travers le bit 22 du registre GPDR0.**

- **De plus, on doit s'assurer qu'il sera dédié pour la gestion d'une entrée sortie à travers le même numéro de bit dans le registre GAFR0_U.**

- **Par conséquence, le bit 22 du registre GPDR0 doit être mis à 1, tandis que le bit 22 du registre GAFR0_U doit être mis à 0.**

- **L'instruction suivante décrit le masque à utiliser pour initialiser le bit 22 du registre GPFR0. Le masque a été sauvegardé dans une variable constante nommée LED_VERT.**

2.1 Initialisation des registres de contrôle

#define LED_VERT (0x00200000)

- **Afin de mettre à un le 22ème bit (partant de zéro) du registre GPDR0, on doit faire un OU logique avec le masque définit précédemment.**

- **Pour cela, on peut utiliser l'opérateur de manipulation de bits OU logique '|' dans le langage C.**

- **Le '&' (ET logique) sera aussi utilisé pour mettre à zéro le bit correspondant dans le registre GAFR0_U.**

2.1 Initialisation des registres de contrôle

```
#define PIN22_FUNC_GENERAL          (0xFFDFFFFF)
#define LED_VERT                    (0x00200000)

/*****************************************************************

 *Fonction d'initialisation du LED2 sur une carte mère ArcomViperIO

*****************************************************************/
void Initialisation(void)
{
    /* Modifier la fonctionnalité du pine pour le rendre dédié à une
fonctionnalité générale */

GPIO_0_FUNC_HI_REG &= PIN22_FUNC_GENERAL;

/* Modifier le type du pine correspondant au LED2 afin de fonctionner comme
une sortie. */
    GPIO_0_DIRECTION_REG |= LED_VERT;
}
return 0;
}
```

Fonction d'initialisation du LED2

2.2 Contrôler le clignotement du led

- **Comme présenté dans le tableau précédent, les registres GPSR0 et GPCR0 permettent respectivement de modifier l'intensité du signal sortant du pin numéro 22.**

- **Ce dernier permet d'étendre le Led vert si le bit correspondant dans le registre GPSR0 est mis à 1 (signal élevé) et d'allumer le led vert dans le cas où le bit correspondant dans le registre GPCR0 est mis à 1 (signal faible).**

- **Par conséquence, la fonction Clignoter va vérifier l'état du led à travers la lecture du registre d'état GPLR0, et modifier en conséquence le contenu des registres GPSR0 et GPCR0.**

2.2 Contrôler le clignotement du led

```
/*****************************************************************
*

 *Fonction de clignotement du LED2 sur une carte mère ArcomViperIO

*****************************************************************/
void Clignoter(void)
{
    /* verifier l'état courant du led vert */

if (GPIO_0_LEVEL_REG & LED_VERT)
        GPIO_0_CLEAR_REG = LED_VERT;
else
        GPIO_0_SET_REG = LED_VERT;
}
```

Fonction de clignotement du LED2

- **Les identificateurs GPIO_0_LEVEL_REG, GPIO_0_CLEAR_REG et GPIO_0_SET_REG correspondent respectivement aux registres GPLR0, GPSR0 et GPCR0.**

2.3 Temporiser l'appel de la fonction Clignoter

- **L'allumage et l'extension du led doivent être séparés par une courte période.**

- **Pour cela, on doit temporiser un petit instant avant d'appeler à nouveau la fonction Clignoter.**

- **Par conséquence, la fonction pause est appelée avec un paramètre qui représente la durée de la période de temporisation.**

- **Afin d'implémenter la fonction de temporisation, on va utiliser une simple boucle, dont le nombre d'itération correspond à la période de temporisation.**

- **Le paramètre d'appel sera multiplié par la valeur d'une constante (MS_CYCLES) qui permet d'ajuster le temps de la pose en microsecondes.**

2.3 Temporiser l'appel de la fonction Clignoter

```
#define MS_CYCLES                (9000)

/*****************************************************************
* Fonction de temporisation.
*****************************************************************/

void Pause(int duree)
{

  long volatile boucles = (duree * MS_CYCLES);

  while (boucles != 0)

  boucles--;

}
```

Fonction de temporisation

3. Processus de compilation d'un programme embarqué

- **La compilation d'un programme destiné pour un système embarqué génère plusieurs problèmes.**

- **En effet, vu les capacités limitées du dispositif embarqué il est non envisageable dans la plupart des cas de compiler le programme directement sur ce dernier.**

- **Par conséquence, les programmeurs des systèmes embarqués ont tendance à compiler le programme embarqué sur une machine hôte (PC), et de charger ensuite le résultat sur le système ciblé.**

- **Ce processus est connu sous le nom de compilation croisée.**

- **Contrairement à la compilation classique, la compilation croisée exige au programme de bien connaitre l'architecture du matériel cible afin de générer un programme compatible à ce dernier.**

3. Processus de compilation d'un programme embarqué

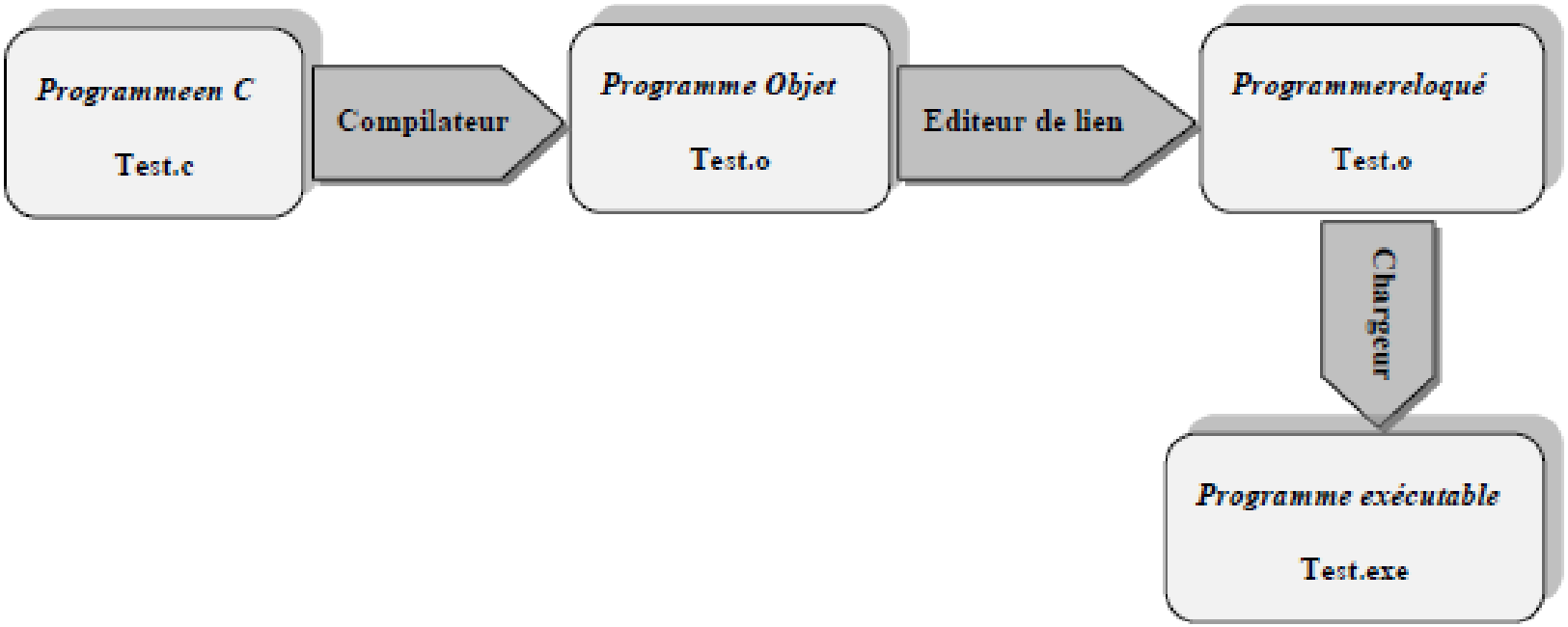
- **Le processus de compilation croisée génère en premier temps un fichier objet**

- **Ensuite les différents fichiers d'un même programme seront assemblés à travers l'éditeur de lien pour construire un seul fichier objet connu sous le nom de programme reloqué (relocatable program).**

- **Enfin, les adresses physiques de la mémoire seront insérées dans le fichier objet à travers un processeur de réallocation (chargeur ou relocateur).**

- **Le résultat final représente un fichier exécutable binaire prêt à s'exécuter sur le système embarqué cible.**

3. Processus de compilation d'un programme embarqué



Cycle de compilation croisée

3.1 Compilation croisée

- **La compilation croisée est la première étape dans la génération d'un programme exécutable embarqué.**

- **Le résultat de cette étape est un fichier objet contenant un ensemble d'opérations primaires compatibles avec le jeu d'instructions du processeur cible.**

- **Pour cela, un compilateur croisé et adapté au jeu d'instructions du processeur embarqué doit être utilisé.**

- **En effet, l'environnement de développement GNU offre un grand nombre de compilateurs croisés pour les langages C (compilateur gcc) et assembleur (compilateur as).**

- **Ces compilateurs supportent plusieurs marques de processeurs embarqués tels que : PowerPC, Intel x86, ARM, AVR, SPARC, MIPS etc….**

3.1 Compilation croisée

- **Comme mentionné dans la section précédente, le fichier généré par le compilateur est un fichier objet.**

- **Ce dernier est formaté dans l'un des formats standards COFF (Common Object File Format) ou ELF (Executable and Linkable Format).**

- **La plupart des fichiers objets commencent par une entête qui décrit le contenu des sections qui vont suivre.**

- **Le compilateur gcc regroupe le contenu du fichier objet dans plusieurs sections dans lesquelles :**

  - **Le code source est groupé dans une section nommée 'text',**
  - **L'initialisation des variables globales est groupée dans une section nommée 'data' ,**
  - **Les variables globales non initialisées dans une section nommée 'bss'.**

- **Après que le fichier objet soit généré, l'éditeur de lien sera utilisé afin d'ajouter les références qui manquent dans ce dernier.**

3.2 L'édition de lien

- **L'édition de liens est une étape très nécessaire dans la génération d'un programme embaqué.**

- **Celle-ci consiste à combiner les différents fichiers objets générés par le compilateur croisé, et résoudre toutes les références manquantes.**

- **L'environnement de développement GNU dispose d'un éditeur de liens (ld) qui permet d'éditer le lien des fichiers objets générés par le compilateur GNU C (gcc).**

- **Durant le processus d'édition de liens, le programmeur doit inclure en plus de ses programmes le programme de démarrage des composants électroniques du système embarqué (startup program).**

- **Ce dernier représente un ensemble d'instructions assembleur qui permettent de préparer l'environnement d'exécution pour le programme embarqué (pile d'exécution).**

3.2 L'édition de lien

- **La plupart des compilateurs croisés offrent un tel programme tel que:**
  - **Le compilateur croisé du langage C 'gcc' où le programme de démarrage est connu sous le nom de 'crt0.s'**
  - **Et le compilateur croisé assembleur 'as' où le programme de démarrage est connu sous le nom de 'startup.asm'.**
- **Le programme de démarrage permet aussi de donner la main à un programme embarqué à travers un appel à sa fonction principale.**
- **Dans des cas particuliers, le dispositif embarqué dispose d'un programme de débogage qui permet d'initialiser le hardware, d'afficher les éventuelles erreurs et de donner la main au utilisateur afin d'effectuer certaines tâches de contrôle.**

3.2 L'édition de lien

- **Par conséquence, les programmes embarqués n'ont pas besoin d'être liés à un programme de démarrage dans le cas où un programme débogueur est présent sur le matériel embarqué.**

- **Dans le cas de notre exemple, la carte mère ArcomViper IO dispose d'un programme débogueur appelé RedBoot (RedHat's Embedded Debug and Bootstrap program).**

- **Le programme débogueur permet aussi de charger le logiciel embarqué sur le dispositif cible.**

3.3 La résolution des adresses physiques

- **Après l'édition de liens du programme embarqué, il faut ensuite attribuer des adresses physiques aux différentes sections de code du fichier objet.**

- **En effet, c'est au programmeur d'introduire les informations sur la mémoire du système embarqué.**

- **Il existe plusieurs outils qui permettent l'affectation des adresses physiques. Cependant, dans l'environnement GNU l'éditeur de liens se charge de cette procédure.**

- **Ce dernier se base sur un script qui définit les principales adresses physiques à utiliser.**

- **Dans la figure suivante un exemple de script utilisé par l'éditeur de liens afin de résoudre les adresses physiques du programme 'Blink' présenté précédemment.**

- **Le script de résolution d'adresses informe l'éditeur de liens des périphériques de stockage disponibles ainsi que leurs tailles et les adresses de démarrage.**

3.3 La résolution des adresses physiques

```
ENTRY (main)

MEMORY
{   ram : ORIGIN = 0x00400000, LENGTH = 64M
rom : ORIGIN = 0x60000000, LENGTH = 16M}

SECTIONS
{  data :                 /* Section des variables globales initialisés */
{
        _DataStart= . ;
        *(.data)
        _DataEnd= . ;

    } >ram

bss :              /* Section des variables globales non initialisés */
    {
_BssStart= . ;
        *(.bss)
        _BssEnd= . ;
    } >ram

text :              /* section du code source */
{
        *(.text)
    } >ram }
```

Script pour la résolution des adresses physiques

3.4 Exemple de compilation d'un programme embarqué

- **Toutes les étapes de génération du code binaire seront effectuées sur l'environnement de développement GNU sous le système d'exploitation Linux.**

3.4.1 Compilation des fichiers sources.

- **La carte mère ArcomViper IO dispose d'un processeur embarqué de type ARM. Par conséquence, le compilateur croisé 'arm-elf-gcc' sera utilisé.**

- **La structure de base de l'instruction de compilation est comme suit :**

arm-elf-gcc [*options*] *Fichier_source*...

- **Le mot option définit l'option de compilation qui peut être :**

  - **-c : pour une compilation sans édition de liens,**
  - **-g : pour une compilation par défaut,**
  - **-I : pour prendre en compte les fichiers entêtes**
  - **-Wall : pour permettre la signalisation des messages d'avertissement**

3.4 Exemple de compilation d'un programme embarqué

- **Dans le cas de notre exemple, on a deux fichiers source à savoir 'Blink.c' et 'Led.c'.**

- **Par conséquence, les instructions de compilation à entrer dans l'invité de commande de l'environnement GNU seront comme suit :**

```
# arm-elf-gcc -g -c Wall -I../include led.c

# arm-elf-gcc -g c -Wall -I../include blink.c
```

3.4.2 L'édition de liens et la résolution d'adresses physiques

- **La structure générale de l'instruction d'édition de liens est comme suit :**

arm-elf-ld [options] fichier_objet...

- **-Map : pour générer un fichier de sortie map**
- **-T : pour affecter un script d'édition de liens**
- **-N : pour permettre la lecture et l'écriture des sections 'text' et 'data'**
- **-o : pour définir le nom du fichier exécutable**

3.4 Exemple de compilation d'un programme embarqué

- **Dans le cas de notre exemple, la commande d'édition de liens et de la résolution d'adresses est comme suit :**

 # arm-elf-ld -Map blink.map -T viperlite.ld -N -o blink.exe led.o blink.o

- **Le fichier viperlite.ld représente le fichier script qui contient des informations sur les composants de stockage de la carte mère ArcomViper IO.**

- **Ce dernier sera utilisé par l'éditeur de liens afin d'insérer les adresses physiques dans le fichier binaire.**

3.4.3 Formatage du fichier binaire

- **Le fichier exécutable qui résulte de l'étape précédente est prêt à être téléchargé sur le dispositif embarqué.**

- **Cependant, il est parfois nécessaire de formater ce dernier pour qu'il soit compatible avec le système cible.**

3.4 Exemple de compilation d'un programme embarqué

- **L'environnement GNU dispose d'un outil de formatage connu sous le nom 'strip' qui est fourni avec le package 'binutils'.**

- **L'outil de formatage permet de supprimer des sections précises du code objet. La structure générale de l'instruction de formatage est comme suit :**

arm-elf-strip [options] Fichier_entrée... [-o Fichier_sortie]

- **Dans le cas de notre exemple, l'instruction de formatage suivante sera utilisée pour supprimer la section du programme du démarrage dans le cas où le dispositif embarqué dispose d'un programme de débogage.**

- **Le code suivant présente l'instruction de formatage a utilisée :**

 # arm-elf-strip --remove-section=.comment blinkdbg.exe -o blink.exe .