Développer en C sur 68332 avec les outils GNU

Florent de LAMOTTE

8 janvier 2002

Ce document a pour but de présenter l'environnement GNU pour développer sur une cible 68332.

1 Pourquoi les outils GNU

Nous disposions de trois solutions pour développer sur 68332 :

- Utiliser les outils gratuits mis a disposition par Motorola as32 et bd32 respectivement un assembleur et un debugger pour 68332. Qui nous auraient permis de développer en assembleur pour notre cible
- Utiliser la chaîne de développement HIWARE, utilisée par l'école qui permettait de développer du code en C ou en assembleur.
- Utiliser les outils GNU qui permettent de développer en C et en assembleur.

Les outils GNU sont disponibles sur un grand nombre de plateformes et permettent de générer du code pour un grand nombre d'architectures dont le 68332.

Ces outils sont gratuits et téléchargeables sur Internet.

Ils sont par contre un peu plus difficiles à mettre en oeuvre et à utiliser que d'autres outils car ils nécessitent une recompilation spécifique pour la cible sur laquelle on veut travailler et ne possèdent pas d'IDE. Par contre tous les membres du club de robotique ont utilisé GCC pendant les cours de C.

2 Ce dont vous aurez besoin

Pour pouvoir développer avec les outils GNU, vous aurez besoin de recompiler ceux-ci pour votre cible. Voici les packages que vous aurez besoin de récupérer

- binutils
- GCC
- GDB ainsi que les patches pour le bdm
- newlib qui est une librairie regroupant toutes les fonctions standard

3 Installation des outils de développement

Nous allons procéder à une installation des composants nécessaires pour développer avec les outils GNU. Ces outils seront placés dans le répertoire */usr/local/m68k-coff* et un lien portant le préfixe *m68k-coff-* sera créé pour chacun des utilitaires dans le répertoire */usr/local/bin*.

3.1 Binutils

L'archive binutils contient les outils de base pour générer les fichiers binaires.
Binutils fournit :
– un assembleur : gas
– un linker : ld
– un décompilateur : objdump
– un archiveur : ar
– et d'autres outils indispensables
L'installation des binutils s'effectue en deux phases :
– la phase de compilation
– la phase d'installation
Désarchivez binutils, et entrez dans le répertoire nouvellement créé. Créez y un répertoire dans lequel vous allez pouvoir compiler binutils :

```
$ mkdir m68k-coff-obj
$ cd m68k-coff-obj
```

Configurez l'archive à l'aide du script de configuration présent dans le répertoire de base.

```
$ ../configure --target=m68k-coff
```

Il ne vous reste plus qu'à compiler et à installer à l'aide de make

```
$ make
$ su
$ make install
```

3.2 GCC

GCC est une collection de compilateurs, nous allons l'utiliser pour compiler du C pour notre cible.
Pour installer GCC, vous procédez de la même manière que pour binutils, pour dire que vous ne voulez que le compilateur C, vous devez ajouter l'option *–enable-languages=C*.

3.3 GDB

GDB est le debugger du projet GNU. En appliquant des patches à GDB, on peut lui permettre de travailler avec le BDM[1] du 68332.
Le patch doit être récupéré et doit correspondre a la version de GCC utilisée.
On applique le patch en utilisant la commande patch dans le répertoire dans lequel on a lancé la commande de décompression de l'archive de GDB.

```
$ patch -p0 < gdb-bdm-patch
```

Une fois le patch appliqué, on peut procéder à l'installation de GDB.
Le script de configuration est le suivant.

[1]Background Debugger Mode : ce mode de fonctionnement permet de debugger une carte en faisant tourner le debugger sur un PC

```
$ ../configure --target=m68k-bdm-coff
```

Ceci permet de valider le mode bdm de GDB. En ce qui concerne les autres étapes de l'installation, elles restent identiques à celles décrites pour binutils.

Par contre, une fois GDB installé, il faut encore installer le driver pour le BDM. Celui-ci est normalement livré avec le patch pour le BDM. Il suffit de suivre les instructions livrées pour l'installer.

3.4 Newlib

Newlib est une librairie permettant d'utiliser les fonctions standard du C. Son installation n'est pas obligatoire mais elle apporte un confort non négligeable.

L'installation ne pose pas de difficultés et est identique à celle des binutils, il faut quand même que binutils et GCC soient correctement installés pour qu'elle se termine sans erreur.

4 Utilisation

L'utilisation du compilateur C est en tout point identique à celle de GCC dans le cas d'une compilation pour linux par exemple. L'exécutable permettant de lancer GCC s'appelle *m68k-coff-gcc* par contre pour GDB, son exécutable s'appelle *m68k-bdm-coff-gdb*.

Outre les fichiers source du code que l'on désire compiler, on aura besoin de plusieurs fichiers, assurant la bonne compilation du code et l'implantation dans la cible.

- un fichier Makefile décrivant la compilation et les dépendances entre les fichiers.
- un script pour le linker, qui lui permettra de savoir ou placer le code en mémoire.
- le code de démarrage, s'occupant d'initialiser le programme en copiant les données présentes en ROM dans la RAM.

On peut aussi écrire des fichiers permettant de paramétrer le debugger.

Nous allons décrire comment créer ces fichiers et comment utiliser les outils de développement.

4.1 Le linker

Le but du linker est de prendre plusieurs fichiers objets dits sources et de n'en ressortir qu'un qui correspond au programme.

Le linker que nous utilisons s'appelle *ld* et est fourni dans le package binutils.

Pour paramétrer son fonctionnement, on utilise un script qui décrit comment seront intégrées dans le fichier objet final les différentes sections du programme.

En effet, un programme est normalement divisé en au moins trois sections qui sont :

- *.text* : qui contient le code du programme.
- *.data* : qui contient toutes les variables globales initialisées du programme.
- *.bss* : qui contient toutes les variables non initialisées du programme.

Le script permet aussi de définir des symboles qui seront ensuite utilisés par le programme de démarrage pour connaître l'emplacement en mémoire des différentes sections du programme

```
/*
 * Script permettant de linker pour les rack de l'ESEO
 */
STARTUP(crt0.o)
```

```
OUTPUT_ARCH(m68k)
OUTPUT(rom.gdb) /* nom par defaut */
SEARCH_DIR(/usr/local/m68k-coff/lib/mcpu32);
SEARCH_DIR(/usr/local/lib/gcc-lib/m68k-coff/3.0.2/mcpu32);

/*
 * Point d'entree pour gdb
 */
ENTRY (gdb_startup)

SECTIONS
{
        /*
         * Section .text (code)
         * Elle est placee dans la Flash et y reste
         */
        .text   0x00000000 : AT (0)
        {
                __s_text = . ;  /* definition de symboles (. <=> addr courante) */
                *     (.text)
                CONSTRUCTORS
                __e_text = . ;  /*  utilises dans le code d'initialisation*/
        } > rom

        /*
         * Section .data
         * Placee juste apres .text dans la FLASH
         * pour etre copiee au debut de la RAM 0x20000
         * lors de l'initialisation
         */
        .data   0x00020000 : AT (SIZEOF(.text))
        {
                __s_data = . ; /* Ces symboles permettent de savoir */
                *(.data)
                __e_data = . ; /* ou copier .data dans la RAM */
        } > ram

        /*
         * Section .bss
         * Elle est placee dans la ram, juste apres .data
         */
        .bss    0x00020000 + SIZEOF(.data) :
        {
                __s_bss = . ; /* Ces symboles permettent */
                *(.bss)
```

```
                *(COMMON)
                __e_bss = . ; /* d'effacer le bss */
        } > ram
}

/*
 * Un symbole pour definir l'emplacement de la pile
 * Ce symbole est utilisee par la routine sbrk,
 * chargee d'agrandir le tas
 */
__s_stack = 0x3F800;

/*
 * Definit la configuration memoire
 * Permet a ld de faire quelques petites verifications
 */
MEMORY {
  rom : ORIGIN = 0x00000000, LENGTH = 128K
  ram : ORIGIN = 0x00020000, LENGTH = 128k
}
```

4.2 Le programme de démarrage *crt0*

Le programme de démarrage aura donc la charge de copier la section *.data* du programme, présente uniquement dans la ROM au démarrage vers la RAM, d'initialiser la section *.bss* à zéro et de lancer le programme en appelant la fonction *main*.

Dans notre cas, il faut d'abord qu'il initialise les fonctions minimales du 68332 comme par exemple les *chip-selects* et le *watchdog*.

4.3 Le Makefile

Le Makefile décrit comment doit être compilé le programme.

4.4 Newlib

Newlib est une librairie d'utilitaires offrant les fonctions de la librairie standard du C comme par exemple :

- printf
- putchar
- malloc, free
- les fonctions mathématiques (cos, sin ...)

Pour certaines fonctions, il faut écrire des *stubs* qui permettent d'adapter la librairie à la cible que l'on est en train d'utiliser. Par exemple, pour pouvoir utiliser la fonction *malloc*, il faut fournir la fonction *sbrk* qui est dépendante du système, c'est cette fonction qui s'occupe d'agrandir le tas et qui indique quand il n'y a plus de mémoire.

Il n'est pas nécessaire d'implémenter tous les stubs, certains ne sont jamais appelés. En fait il ne faut implémenter un stub que lorsqu'il est demandé par le linker.

4.5 Debugger avec GDB et DDD

GDB est le debugger par défaut du GNU. Mais son interface texte est plutôt austère. On peut lui associer un debugger graphique appelé DDD.

Pour lancer DDD en lui spécifiant d'utiliser notre GDB comme debugger, il faut l'invoquer ainsi :

```
$ ddd --debugger m68k-bdm-coff-gdb programme
```

Il nous faut ensuite initialiser le mode BDM à l'aide des commandes suivantes :

```
(GDB) target bdm /dev/pd_bdm0
(GDB) bdm_reset
(GDB) bdm_autoreset off
(GDB) break main
(GDB) run
```

Une fois le programme lancé à l'aide de la commande *run*, tout se passe comme si le programme était exécuté sur le PC.

Pour les fainéants, il est possible d'écrire un script exécutant les instructions d'initialisation du BDM et de le faire exécuter au démarrage de GDB en ajoutant *-x script* a la ligne de commande.

Lors de l'exécution de la commande *run*, il vous sera demandé si vous voulez charger votre programme en mémoire. GDB écrit le programme dans la RAM de la cible, c'est la seule manière de pouvoir debugger un programme car lors de l'insertion d'un point d'arrêt, le 68332 positionne un flag dans le code !

Lors de l'initialisation du programme en RAM, GDB place lui-même les différentes sections du programme à leur place finale. Il ne faut surtout pas que crt0 copie la section *.data* puisque a l'emplacement de la source il n'y a rien.

La solution que j'ai adopté a été d'écrire deux points d'entrée au programme. Le premier est le point d'entrée spécifié dans la table d'exception, celui-ci est utilisé lors d'un démarrage *normal* du système. L'autre est spécifié dans le script ld par la directive *ENTRY* et sera pris en compte par GDB, ce dernier ne s'occupe que d'initialiser les chip-selects et d'appeler *main*.