

Informatique embarquée

**Langages, techniques et outils
pour la programmation
embarquée**

Philippe.Plasson@obspm.fr

Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Sommaire

1. Langages pour le développement des logiciels embarqués

2. Outils pour le développement des logiciels embarqués

- a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
- b. Simulateurs (QEMU, TSIM, ...)
- c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
- d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
- e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
- f. Environnements de développement intégrés

Langages pour le développement des logiciels embarqués – L'assembleur

- Langage de bas niveau permettant de représenter le langage machine (utilisant le jeu d'instructions du processeur) sous une forme lisible (à l'aide de mnémoniques).
- L'assembleur doit être utilisé de façon parcimonieuse dans certains cas où il est vraiment indispensable (par exemple dans certaines parties du code des OS temps-réel ou dans certains drivers) :
 - initialisation de la trap table
 - initialisation des registres du processeur
 - sauvegarde / restauration de contexte
 - accès à certaines instructions du processeur non prises en par les compilateur

Langages pour le développement des logiciels embarqués – L'assembleur

- L'architecture complexe des processeurs fait qu'il est de plus en plus délicat de programmer directement en assembleur.
- Programmer en assembleur est loin d'être un gage de production de code efficace.
- Les compilateurs C/C++ produisent un code machine plus compacte et plus efficace que celui qui pourrait être produit par un humain.
- Conclusion : pour des raisons de qualité logicielle et de portabilité du code, il faut minimiser dans un projet la quantité de code assembleur.

Langages pour le développement des logiciels embarqués – L'assembleur

- Exemple d'une routine assembleur (LEON3, jeu d'instructions Sparc V8) permettant d'effacer une zone mémoire.

```
/* the start address to be cleaned is set in %g2 = 0x40000028*/
/* shall be aligned on 8 bytes due to std instruction working on double word */
set 0x40000028, %g2
/* the number of bytes to erase is set in %g3 = 256 * 1024 * 1024 - 0x28 = 268435416 */
set 268435416, %g3
clr %g4
.Lcleansdram:
    cmp %g3, %g4
    be .Lcleansdram_end
    nop
    std %g0, [%g2 + %g4]
    ba .Lcleansdram
    /* std instruction works on double word => the loop counter shall be incremented by 8 */
    add %g4, 8, %g4
.Lcleansdram_end:
```

Langages pour le développement des logiciels embarqués – Le langage C

- Langage le plus utilisé pour le développement des logiciels embarqués.
- Offre un niveau d'abstraction bien meilleur que celui de l'assembleur.
 - Meilleure lisibilité.
 - Meilleure modularité et réutilisabilité.
- Par rapport à l'assembleur, l'utilisation du C accroît la productivité des développements et la qualité des développements.
- Bon couplage avec le code écrit en assembleur.
- Le code machine généré est très compact et très efficace.

Langages pour le développement des logiciels embarqués – Le langage C

- Code facilement analysable par des outils de vérification de type PolySpace (analyse statique).
- Principale norme de programmation en C utilisée dans le monde de l'embarqué : MISRA C
 - MISRA = Motor Industry Software Reliability Association
 - <https://www.misra.org.uk/MISRAHome/tabid/55/Default.aspx>
- Nombreuses bibliothèques de traitement numérique.

Langages pour le développement des logiciels embarqués – Le langage C

- Principaux inconvénients :
 - Langage procédural → manque de concepts pour une structuration propre des applications.
 - Pas de liens évidents avec les notations graphiques de type UML.

Langages pour le développement des logiciels embarqués – Le langage C++

- Les compilateurs C++ génèrent un code machine aussi efficace que les compilateurs C.
 - Overhead CPU négligeable.
 - Overhead mémoire négligeable (virtual tables + virtual pointers).
- C++ = Langage objet → apporte toute la richesse de la programmation objets en termes de conception et de modélisation des applications
 - Modularité, réutilisabilité, couplage faible, patterns de conception.
- Par rapport au langage C, l'utilisation du C++ accroît la productivité des développements et la qualité des développements.

Langages pour le développement des logiciels embarqués – Le langage C++

- Lien direct avec UML (notation graphique pour la modélisation des applications logicielles) : facilite la migration vers une approche de type MDE (Model Driven Engineering).
- Principales normes de programmation en C++ utilisées dans le monde de l'embarqué :
 - MISRA C++
 - <https://www.misra.org.uk/Activities/MISRAC/tabid/171/Default.aspx>
 - JSF (Joint Strike Fighter Air Vehicle C++ Coding Standards)
 - <http://www.stroustrup.com/JSF-AV-rules.pdf>

Langages pour le développement des logiciels embarqués – Le langage C++

- Dans le cadre du développement des applications embarquées temps-réel, tous les mécanismes offerts par le C++ ne peuvent pas être utilisés.
 - Seul un sous-ensemble du C++ est utilisé (pour des raisons de performances, de déterminisme et de complexité du code).
- Les mécanismes suivants ne doivent pas être utilisés :
 - Exceptions : non compatible avec les exigences de déterminisme du comportement de l'application.
 - Operateur new / delete : non compatible avec les exigences de déterminisme de l'architecture mémoire.
 - RTTI (Run-Time Type Information) : services implémentés dans la bibliothèque standard du C++ permettant de déterminer le type d'une variable lors de l'exécution : trop gourmand en termes d'empreinte mémoire.

Langages pour le développement des logiciels embarqués – Le langage C++

- En revanche, des mécanismes comme le polymorphisme et les fonctions virtuelles sont particulièrement puissants pour rendre modulaire, réutilisable et reconfigurable le code produit.

Langages pour le développement des logiciels embarqués – Le langage C++

- L'approche par objets actifs =
 - Objet encapsulant son propre thread d'exécution et une FIFO d'entrée pour les messages déclenchant les services
 - Le programmeur se concentre sur l'implémentation des services
 - Masquer les spécificités des noyaux temps réel
 - Pouvoir utiliser le C++ et toute la puissance de la programmation orientée objet pour développer des applications temps-réel embarquées
 - Rendre la transition de l'activité de modélisation vers l'activité d'implémentation quasi-immédiate ➔ approche MDE

Langages pour le développement des logiciels embarqués – Le langage JAVA

- Java SE embedded API (ORACLE)
 - Java SE embedded est basé sur la plate-forme Java Standard Edition.
 - Conçu pour être utilisé sur des systèmes avec au moins 32 Mo de RAM.
 - ARMv5/ARMv6/ARMv7 Linux, X86 Linux Small Footprint.
 - <http://www.oracle.com/technetwork/java/embedded/embedded-se/overview/index.html>
- Java ME embedded API (ORACLE)
 - Conçu pour être utilisé sur des systèmes avec au moins 8 Mo de RAM.
 - Linux ARM, PowerPC, Cortex-M (ST, Freescale, NXP, ...)
 - <http://www.oracle.com/technetwork/java/embedded/javame/embed-me/overview/index.html>

Langages pour les systèmes critiques

■ ADA

- Multitâches, intègre des concepts temps-réel (tâches, objets protégés, interruption)
- Utilisé dans des systèmes temps réel et embarqués nécessitant un haut niveau de fiabilité et de sécurité
- Automobile, transports ferroviaires, aéronautique, espace

■ LUSTRE / ESTEREL

- Langages de programmation synchrones, déclaratifs et par flots.
- Possèdent une définition formelle.
- Utilisés pour la programmation des systèmes réactifs.
- Conception de logiciels critiques dans l'aéronautique (Airbus, Eurocopter, Dassault, Pratt & Whitney), le ferroviaire (Eurostar) et les centrales nucléaires (Schneider Electric).

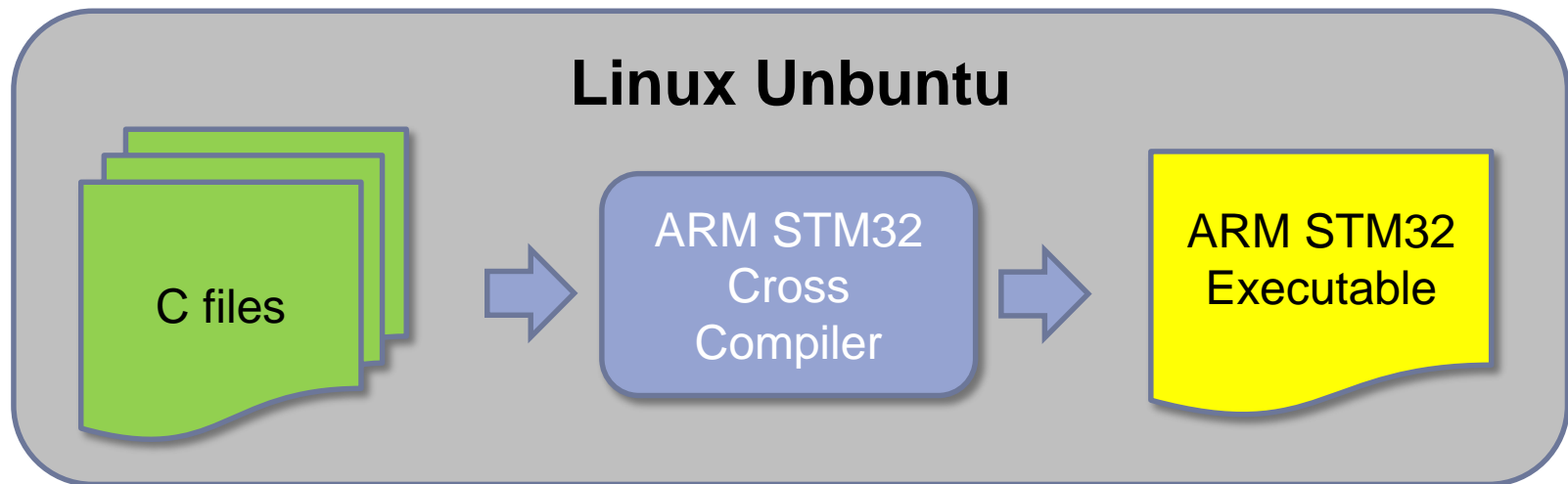
Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Chaînes de compilation croisées

Définition

- Une chaîne de compilation croisée est un ensemble d'outils capables de créer du code exécutable pour une plate-forme autre que celle sur laquelle le compilateur fonctionne, en particulier pour une cible embarquée.
- Un compilateur qui fonctionne sur Linux Ubuntu mais génère du code qui fonctionne sur un microcontrôleur ARM STM32 est un compilateur croisé.



Chaînes de compilation croisées

Définition

- Beaucoup de chaînes de compilation croisées pour les cibles embarquées sont basées sur GCC (GNU Compiler).
 - Le manuel utilisateur de GCC peut être trouvé ici :
<https://gcc.gnu.org/onlinedocs/gcc/>
- Il existe des alternatives à GCC comme les chaînes de compilation basées sur LLVM et le compilateur Clang (<https://llvm.org/>)

Chaînes de compilation croisées

Définition

- Une chaîne de compilation basée sur GCC contient généralement les éléments suivants :
 - GNU C/C++ Compiler : compilateur croisé
 - Binutils : collection d'outils permettant de manipuler / transformer des fichiers binaires
 - GDB : débogueur (<https://www.gnu.org/software/gdb/>)
 - Newlib : bibliothèque C conçue pour les systèmes embarqués et contenant des fonctions d'accès au temps, de définition de timers, d'affichage redirigés vers les ports série, de fonctions math.,
 - <https://sourceware.org/newlib/>
 - Bibliothèques propres à la cible embarquée (« Bare-C run-time library », « Board Support Package ») permettant de gérer par exemple les traps et interruptions (ex. : la bibliothèque libleonbare.a pour les processeurs LEON)
 - Un outil permettant de construire une image bootable d'un exécutable (loader / chargeur).

Chaînes de compilation croisées

Exemples

- Exemples de chaîne de compilation croisées (cibles ARM) :
 - GNU Arm Embedded Toolchain
 - Chaîne de compilation disponible pour les environnements Microsoft Windows, Linux et Mac OS X
 - Pour le développement de logiciels « bare metal » (sans OS) pour les cibles processeur ARM Cortex-M et Cortex-R.
 - <https://developer.arm.com/open-source/gnu-toolchain/gnu-rm>
 - SW4STM32
 - Environnement complet de développement pour les microcontrôleurs STM32 et carte associées comprenant une chaîne de compilation multi-OS basée sur GCC
 - Disponible pour les environnements Microsoft Windows, Linux et Mac OS X
 - <https://www.st.com/en/development-tools/sw4stm32.html>

Chaînes de compilation croisées

Exemples

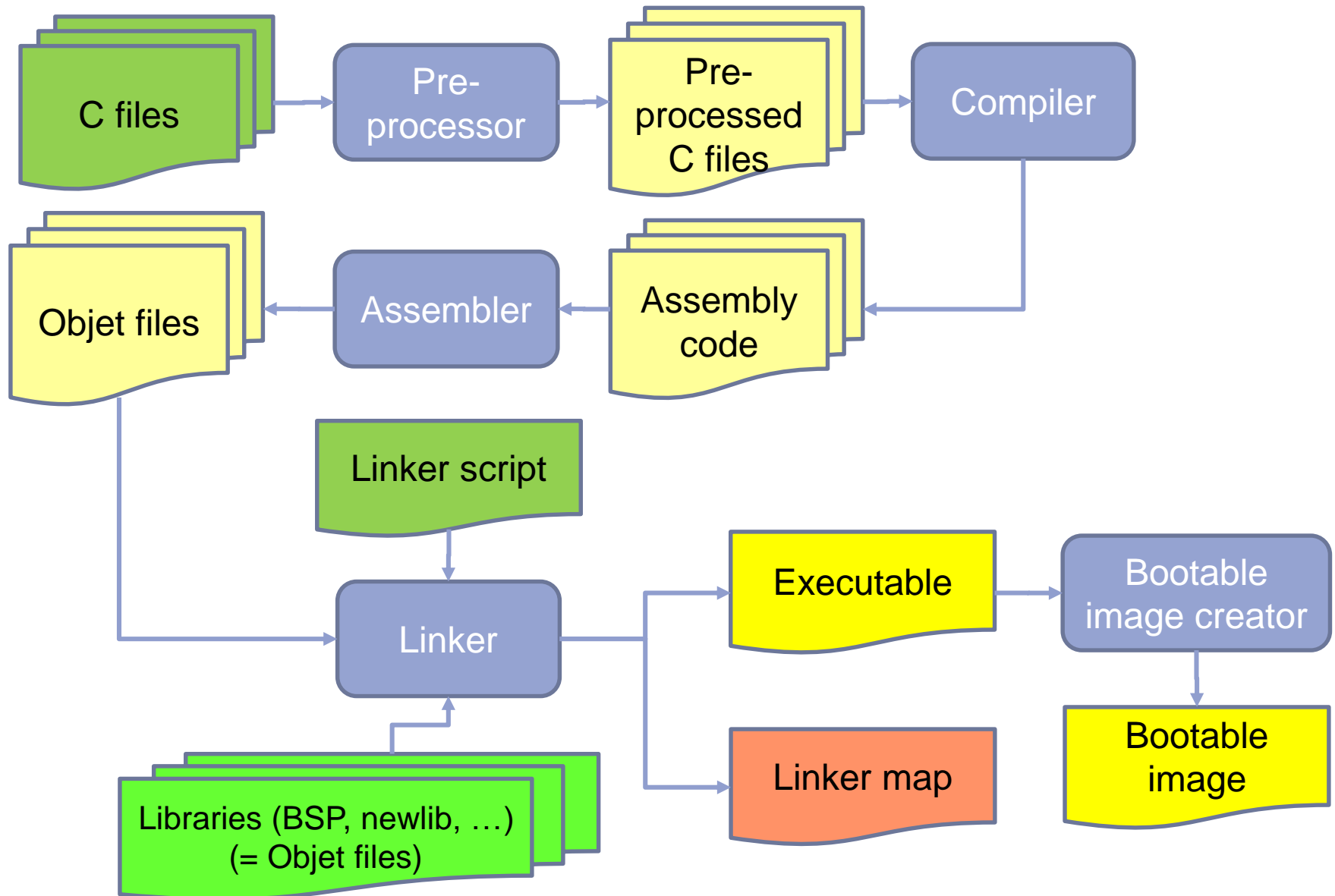
- Exemples de chaîne de compilation croisées (cibles LEON) :
 - LEON Bare-C Cross Compilation System (BCC)
 - Chaîne de compilation disponible pour les environnements Microsoft Windows et Linux
 - Pour le développement de logiciels pour les cibles processeurs LEON.
 - <https://www.gaisler.com/index.php/products/operating-systems/bcc>
 - RTEMS LEON/ERC32 Cross-Compiler System
 - Chaîne de compilation disponible pour les environnements Microsoft Windows et Linux
 - Pour le développement de logiciels pour les cibles processeurs LEON utilisant l'OS RTEMS

Les compilateurs croisés

Définition

- Un compilateur C produit à partir de fichiers sources C des fichiers objets.
 - Un fichier objet est produit pour chaque fichier source C.
 - Extension= .o
- Ces fichiers objets contiennent du code machine et des informations qui seront utilisées par le linker pour construire un exécutable.
- Le code machine est une représentation binaire des instructions qui pourront être exécutées par un processeur.
- Les fichiers objets ne peuvent pas être directement chargés dans la mémoire du processeur et exécutés : ils doivent être combinés ensemble afin de créer un exécutable → c'est le rôle du linker.

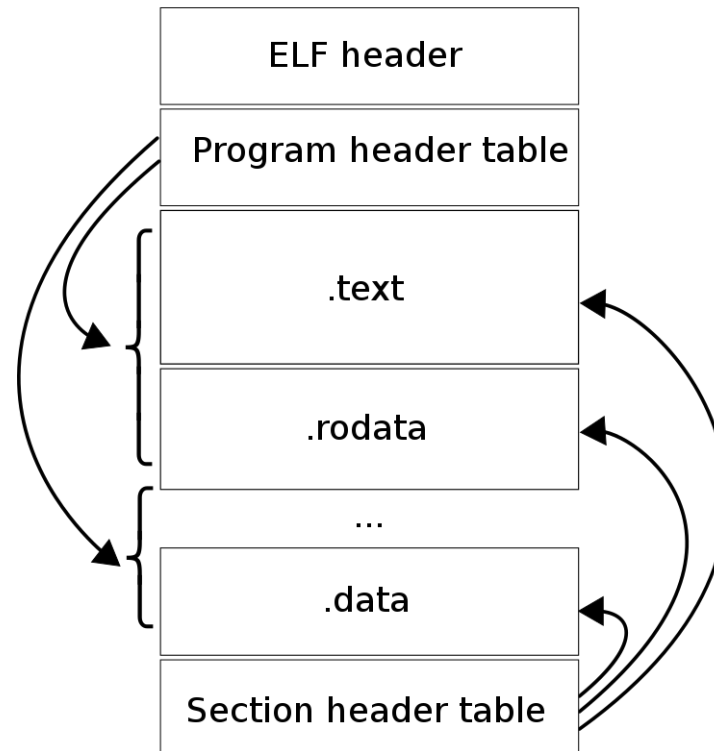
Les différents éléments d'une chaîne de compilation



Les compilateurs croisés

Le format ELF

- Les fichiers objets sont généralement au format ELF (Executable and Linkable Format).
 - ELF = format de fichier standard pour les exécutables, les fichiers objets, les bibliothèques partagées et les « core dumps ».



Les linkers

Définition

- Le linker (ou éditeur de liens) a pour but de combiner l'ensemble des fichiers objets produits afin de créer un exécutable (au format ELF).
- Le linker a pour rôle de positionner chaque fichier objet en mémoire à une adresse précise et à compléter les instructions machines contenant des adresses de fonctions non résolues au moment de la compilation.

Les linkers

Définition

- Quand dans un fichier source C1, une fonction f1() appelle une autre fonction f2() située dans un fichier source C2, alors le fichier objet O1 produit en compilant le fichier C1 contient un symbole non résolu (adresse non résolue), celui de la fonction f2().
- Le linker lie les fichiers objets entre eux en résolvant les symboles.

Chaîne de compilation

Exemple avec sparc-elf-gcc

- On souhaite créer un exécutable à partir de 5 fichiers sources C et pouvant fonctionner sur un processeur LEON.

Compilation des 5 fichiers C de l'application avec production à chaque fois d'un fichier objet. L'appel au préprocesseur, au compilateur et à l'assembleur est fait en une seule fois

```
sparc-elf-gcc -O0 -c -o display_factorial.o display_factorial.c
sparc-elf-gcc -O0 -c -o main.o main.c
sparc-elf-gcc -O0 -c -o factorial.o factorial.c
sparc-elf-gcc -O0 -c -o square.o square.c
sparc-elf-gcc -O0 -c -o store_factorial.o store_factorial.c
sparc-elf-gcc -o factorial store_factorial.o square.o main.o factorial.o display_factorial.o
```

Edition de liens (par le linker) des 5 fichiers objets et création de l'exécutable **factorial**

L'option `-c` indique au compilateur que les fichiers sources doivent être compilés et assemblés mais pas liés

L'option `-o` indique au compilateur dans quel fichier la sortie doit être stockée.

Les compilateurs croisés

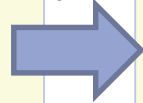
Exemple de fichiers générés

```
sparc-elf-gcc -O2 -g3 -Wall -mcpu=v8 -c -fmessage-length=0 -o "src\\factorial.o"  
"..\src\\factorial.c"
```

```
#include "factorial.h"  
  
uint32_t factorial( uint32_t n)  
{  
    uint32_t result = 1;  
    uint32_t i;  
    for (i = 1; i <= n; i++)  
    {  
        result = result * i;  
    }  
    return result;  
}
```

Fichier source factorial.c

Compilation



```
mov %o0, %o5  
mov 1, %o0  
cmp %o0, %o5  
bgu .LL7  
mov 1, %g1  
smul %o0, %g1, %o0  
.LL9:  
add %g1, 1, %g1  
cmp %g1, %o5  
bleu,a .LL9  
smul %o0, %g1, %o0  
.LL7:  
retl  
nop
```

Sortie intermédiaire non
stockée, peut être produite
avec l'option -S

Assemblage



```
00000000 <factorial>:  
0: 9a 10 00 08 mov %o0, %o5  
4: 90 10 20 01 mov 1, %o0  
8: 80 a2 00 0d cmp %o0, %o5  
c: 18 80 00 07 bgu 28 <factorial+0x28>  
10: 82 10 20 01 mov 1, %g1  
14: 90 5a 00 01 smul %o0, %g1, %o0  
18: 82 00 60 01 inc %g1  
1c: 80 a0 40 0d cmp %g1, %o5  
20: 28 bf ff fe bleu,a 18 <factorial+0x18>  
24: 90 5a 00 01 smul %o0, %g1, %o0  
28: 81 c3 e0 08 retl  
2c: 01 00 00 00 nop
```

Fichier objet factorial.o,
contenu obtenu avec objdump

Exemple simple dans le
quel le fichier objet généré
ne contient pas de
symboles non résolus

Les compilateurs croisés

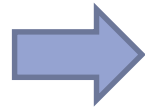
Exemple de fichiers générés

```
sparc-elf-gcc -O2 -g3 -Wall -mcpu=v8 -c -fmessage-length=0 -o  
"src\\store_factorial.o" "..\\src\\store_factorial.c"
```

```
#include "factorial.h"  
#include "store_factorial.h"  
  
void store_factorial( uint32_t *  
array, uint32_t n)  
{  
    uint32_t i = 0;  
    for ( i=0 ; i < n ; i++)  
    {  
        array[i] = factorial(i);  
    }  
}
```

Fichier source store_factorial.c

Compilation



```
save %sp, -104, %sp  
mov 0, %l0  
cmp %l0, %i1  
bgeu .LL7  
nop  
.LL5:  
call factorial, 0  
mov %l0, %o0  
sll %l0, 2, %g1  
add %l0, 1, %l0  
cmp %l0, %i1  
bgeu .LL7  
st %o0, [%i0+%g1]  
call factorial, 0  
mov %l0, %o0  
sll %l0, 2, %g1  
add %l0, 1, %l0  
cmp %l0, %i1  
blu .LL5  
st %o0, [%i0+%g1]  
.LL7:  
ret  
restore
```

Symbole non
résolu (externe)

Symbole non
résolu (externe)

Sortie intermédiaire non
stockée, peut être produite
avec l'option -S

Exemple dans le quel le fichier objet
généré contient des symboles non
résolus au moment de la compilation.
La fonction factorial() n'est pas
définie dans le fichier
store_factorial() : elle est externe →
le symbole factorial ne peut pas être
résolue au moment de la compilation

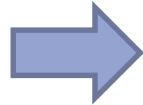
Les compilateurs croisés

Exemple de fichiers générés

```
sparc-elf-gcc -O2 -g3 -Wall -mcpu=v8 -c -fmessage-length=0 -o  
"src\\store_factorial.o" "../src\\store_factorial.c"
```

```
save%sp, -104, %sp  
mov 0, %l0  
cmp %l0, %i1  
bgeu .LL7  
nop  
.LL5:  
call factorial, 0  
mov %l0, %o0  
sll %l0, 2, %g1  
add %l0, 1, %l0  
cmp %l0, %i1  
bgeu .LL7  
st %o0, [%i0+%g1]  
call factorial, 0  
mov %l0, %o0  
sll %l0, 2, %g1  
add %l0, 1, %l0  
cmp %l0, %i1  
blu .LL5  
st %o0, [%i0+%g1]  
.LL7:  
ret  
restore
```

Assemblage



```
00000000 <store_factorial>:  
0: 9d e3 bf 98 save %sp, -104, %sp  
4: a0 10 20 00 clr %l0  
8: 80 a4 00 19 cmp %l0, %i1  
c: 1a 80 00 10 bcc 4c <store_factorial+0x4c>  
10: 01 00 00 00 nop  
14: 40 00 00 00 call 14 <store_factorial+0x14>  
18: 90 10 00 10 mov %l0, %o0  
1c: 83 2c 20 02 sll %l0, 2, %g1  
20: a0 04 20 01 inc %l0  
24: 80 a4 00 19 cmp %l0, %i1  
28: 1a 80 00 09 bcc 4c <store_factorial+0x4c>  
2c: d0 26 00 01 st %o0, [ %i0 + %g1 ]  
30: 40 00 00 00 call 30 <store_factorial+0x30>  
34: 90 10 00 10 mov %l0, %o0  
38: 83 2c 20 02 sll %l0, 2, %g1  
3c: a0 04 20 01 inc %l0  
40: 80 a4 00 19 cmp %l0, %i1  
44: 0a bf ff f4 bcs 14 <store_factorial+0x14>  
48: d0 26 00 01 st %o0, [ %i0 + %g1 ]  
4c: 81 c7 e0 08 ret  
50: 81 e8 00 00 restore
```

Code machine incomplet

Code machine incomplet

Fichier objet store_factorial.o,
contenu obtenu avec objdump

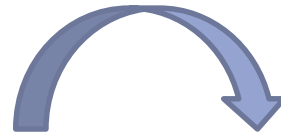
Le code machine généré est incomplet car, au moment de la compilation, l'adresse de la fonction factorial() n'est pas connue. Ce sera le linker qui va créer le lien entre store_factorial() et factorial().

Les compilateurs croisés

Avant linkage / Après linkage

```
sparc-elf-gcc -o factorial store_factorial.o square.o main.o factorial.o  
display_factorial.o
```

Edition des liens par le linker



Adresses

Code machine

```
00000000 <store_factorial>:  
 0: 9d e3 bf 98      save %sp, -104, %sp  
 4: a0 10 20 00      clr %l0  
 8: 80 a4 00 19      cmp %l0, %i1  
 c: 1a 80 00 10      bcc 4c <store_factorial+0x4c>  
10: 01 00 00 00      nop  
14: 40 00 00 00      call 14 <store_factorial+0x14>  
18: 90 10 00 10      mov %l0, %o0  
1c: 83 2c 20 02      sll %l0, 2, %g1  
20: a0 04 20 01      inc %l0  
24: 80 a4 00 19      cmp %l0, %i1  
28: 1a 80 00 09      bcc 4c <store_factorial+0x4c>  
2c: d0 26 00 01      st %o0, [ %i0 + %g1 ]  
30: 40 00 00 00      call 30 <store_factorial+0x30>  
34: 90 10 00 10      mov %l0, %o0  
38: 83 2c 20 02      sll %l0, 2, %g1  
3c: a0 04 20 01      inc %l0  
40: 80 a4 00 19      cmp %l0, %i1  
44: 0a bf ff f4      bcs 14 <store_factorial+0x14>  
48: d0 26 00 01      st %o0, [ %i0 + %g1 ]  
4c: 81 c7 e0 08      ret  
50: 81 e8 00 00      restore
```

Fichier objet
store_factorial.o

```
400019ac <store_factorial>:  
400019ac: 9d e3 bf 98      save %sp, -104, %sp  
400019b0: a0 10 20 00      clr %l0  
400019b4: 80 a4 00 19      cmp %l0, %i1  
400019b8: 1a 80 00 10      bcc 400019f8 <store_factorial+0x4c>  
400019bc: 01 00 00 00      nop  
400019c0: 40 00 27 34      call 4000b690 <etext>  
400019c4: 90 10 00 10      mov %l0, %o0  
400019c8: 83 2c 20 02      sll %l0, 2, %g1  
400019cc: a0 04 20 01      inc %l0  
400019d0: 80 a4 00 19      cmp %l0, %i1  
400019d4: 1a 80 00 09      bcc 400019f8 <store_factorial+0x4c>  
400019d8: d0 26 00 01      st %o0, [ %i0 + %g1 ]  
400019dc: 40 00 27 2d      call 4000b690 <etext>  
400019e0: 90 10 00 10      mov %l0, %o0  
400019e4: 83 2c 20 02      sll %l0, 2, %g1  
400019e8: a0 04 20 01      inc %l0  
400019ec: 80 a4 00 19      cmp %l0, %i1  
400019f0: 0a bf ff f4      bcs 400019c0 <store_factorial+0x14>  
400019f4: d0 26 00 01      st %o0, [ %i0 + %g1 ]  
400019f8: 81 c7 e0 08      ret  
400019fc: 81 e8 00 00      restore
```

Extrait de l'exécutable

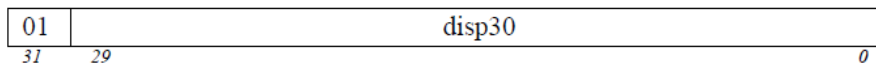
Les compilateurs croisés

Avant linkage / Après linkage

- [illegible]

<i>opcode</i>	<i>op</i>	<i>operation</i>
CALL	01	Call and Link

Format (1):



<i>Suggested Assembly Language Syntax</i>
call <i>label</i>

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address “PC + (4 × *disp30*)”. Since the word displacement (*disp30*) field is 30 bits wide, the target address can be arbitrarily distant.

Extrait de « The SPARC Architecture Manual V8 »
<https://www.gaisler.com/doc/sparcv8.pdf>

Les compilateurs croisés

Options d'optimisation du compilateur

- La performance du code peut être améliorée en jouant sur les options d'optimisation de GCC
- Il y a un grand nombre d'options possibles qui peuvent être activées séparément ou par groupe.
 - Voir <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options>
- 4 groupes sont définis :
 - -O0 : pas d'optimisation ; à utiliser essentiellement pour le débogage ou la couverture de code (car le mapping code machine / code C est plus facile à faire).
 - -O1 : premier niveau d'optimisation
 - -O2 : second niveau d'optimisation
 - -O3 : troisième niveau (le plus élevé)

Les compilateurs croisés

Options d'optimisation du compilateur

```
sparc-elf-gcc -O0 -g3 -Wall -mcpu=v8 -c  
-fmessage-length=0 -o "src\\factorial.o"  
"..\src\\factorial.c"
```

-O0

C:\workspace\factorial\Debug_O0\src\factorial.o:
file format elf32-sparc

Disassembly of section .text_factorial:

```
00000000 <factorial>:  
 0:  9d e3 bf 90      save %sp, -112, %sp  
 4:  f0 27 a0 44      st %i0, [ %fp + 0x44 ]  
 8:  82 10 20 01      mov 1, %g1  
 c:  c2 27 bf f4      st %g1, [ %fp + -12 ]  
10:  82 10 20 01      mov 1, %g1  
14:  c2 27 bf f0      st %g1, [ %fp + -16 ]  
18:  fa 07 bf f0      ld [ %fp + -16 ], %i5  
1c:  c2 07 a0 44      ld [ %fp + 0x44 ], %g1  
20:  80 a7 40 01      cmp %i5, %g1  
24:  18 80 00 0b      bgu 50 <factorial+0x50>  
28:  01 00 00 00      nop  
2c:  fa 07 bf f4      ld [ %fp + -12 ], %i5  
30:  c2 07 bf f0      ld [ %fp + -16 ], %g1  
34:  82 5f 40 01      smul %i5, %g1, %g1  
38:  c2 27 bf f4      st %g1, [ %fp + -12 ]  
3c:  c2 07 bf f0      ld [ %fp + -16 ], %g1  
40:  82 00 60 01      inc %g1  
44:  c2 27 bf f0      st %g1, [ %fp + -16 ]  
48:  10 bf ff f4      b 18 <factorial+0x18>  
4c:  01 00 00 00      nop  
50:  c2 07 bf f4      ld [ %fp + -12 ], %g1  
54:  b0 10 00 01      mov %g1, %i0  
58:  81 c7 e0 08      ret  
5c:  81 e8 00 00      restore
```

```
sparc-elf-gcc -O2 -g3 -Wall -mcpu=v8  
-c -fmessage-length=0 -o  
"src\\factorial.o"  
"..\src\\factorial.c"
```

-O2

C:\workspace\factorial\Debug_O2\src\factorial.o:
file format elf32-sparc

Disassembly of section .text_factorial:

```
00000000 <factorial>:  
 0:  9a 10 00 08      mov %o0, %o5  
 4:  90 10 20 01      mov 1, %o0  
 8:  80 a2 00 0d      cmp %o0, %o5  
 c:  18 80 00 07      bgu 28 <factorial+0x28>  
10:  82 10 20 01      mov 1, %g1  
14:  90 5a 00 01      smul %o0, %g1, %o0  
18:  82 00 60 01      inc %g1  
1c:  80 a0 40 0d      cmp %g1, %o5  
20:  28 bf ff fe      bleu,a 18 <factorial+0x18>  
24:  90 5a 00 01      smul %o0, %g1, %o0  
28:  81 c3 e0 08      retl  
2c:  01 00 00 00      nop
```

Les compilateurs croisés

Prise en charge des FPU

- Certains processeurs sont équipés de FPU = Floating Point Unit, c'est-à-dire d'une unité capable de réaliser des opérations flottantes, sans avoir besoin de recourir à une bibliothèque mathématique d'émulation des calculs flottants basée sur des calculs entiers.
- L'utilisation d'un FPU accroît considérablement les performances des routines de calcul utilisant des opérations flottantes.
- Pour les processeurs intégrant un FPU, l'option GCC `-msoft-float` permet de désactiver l'utilisation du FPU et de forcer l'utilisation d'une bibliothèque émulant les calculs flottants.

Les compilateurs croisés Avec FPU

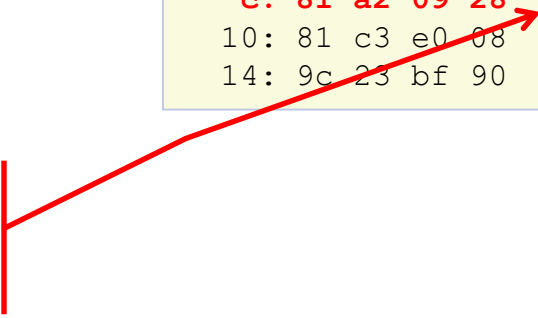
```
sparc-elf-gcc -O2 -g3 -mcpu=v8 -Wall -c -fmessage-length=0 -o "src\\square.o"  
"..\src\square.c"
```

```
float square(float x) {  
    float result = x * x;  
    return result;  
}
```

```
C:\workspace\factorial\Debug_O2_FPU\src\square.o:  
      file format elf32-sparc
```

Disassembly of section .text:

```
00000000 <square>:  
    0: 9c 03 bf 90    add %sp, -112, %sp  
    4: d0 23 a0 64    st  %o0, [ %sp + 0x64 ]  
    8: d1 03 a0 64    ld  [ %sp + 0x64 ], %f8  
   c: 81 a2 09 28    fmuls %f8, %f8, %f0  
  10: 81 c3 e0 08    retl  
  14: 9c 23 bf 90    sub %sp, -112, %sp
```



fmuls est une instruction
du FPU

Les compilateurs croisés

Sans FPU

```
sparc-elf-gcc -O2 -g3 -Wall -msoft-float -c -fmessage-length=0 -o "src\\square.o"  
"..\src\square.c"
```

```
float square(float x) {  
    float result = x * x;  
    return result;  
}
```

```
40001a00 <square>:  
40001a00: 9d e3 bf 98    save %sp, -104, %sp  
40001a04: 90 10 00 18    mov %i0, %o0  
40001a08: 40 00 00 73    call 40001bd4 <__mulsf3>  
40001a0c: 92 10 00 18    mov %i0, %o1  
40001a10: 81 c7 e0 08    ret  
40001a14: 91 e8 00 08    restore %g0, %o0, %o0
```

L'appel à `fmuls` est
remplacé par l'appel à une
fonction `<__mulsf3>`:

```
40001bd4 <__mulsf3>:  
40001bd4: 9d e3 bf 60    save %sp, -160, %sp  
40001bd8: a0 07 bf e8    add %fp, -24, %l0  
40001bdc: f0 27 bf c4    st %i0, [ %fp + -60 ]  
40001be0: f2 27 bf c0    st %i1, [ %fp + -64 ]  
40001be4: 90 07 bf c4    add %fp, -60, %o0  
40001be8: 40 00 01 1a    call 40002050 <__unpack_f>  
40001bec: 92 10 00 10    mov %l0, %o1  
40001bf0: b2 07 bf d8    add %fp, -40, %i1  
40001bf4: 90 07 bf c0    add %fp, -64, %o0  
40001bf8: 40 00 01 16    call 40002050 <__unpack_f>  
40001bfc: 92 10 00 19    mov %i1, %o1  
40001c00: c2 07 bf e8    ld [ %fp + -24 ], %g1  
40001c04: 80 a0 60 01    cmp %g1, 1  
40001c08: 08 80 00 1a    bleu 40001c70 <__mulsf3+0x9c>  
40001c0c: b0 07 bf c8    add %fp, -56, %i0  
40001c10: da 07 bf d8    ld [ %fp + -40 ], %o5  
40001c14: 80 a3 60 01    cmp %o5, 1  
40001c18: 28 80 00 0c    bleu,a 40001c48 <__mulsf3+0x74>  
40001c1c: da 07 bf dc    ld [ %fp + -36 ], %o5  
40001c20: 80 a0 60 04    cmp %g1, 4  
40001c24: 22 80 00 10    be,a 40001c64 <__mulsf3+0x90>  
40001c28: 03 10 00 2e    sethi %hi(0x4000b800), %g1  
40001c2c: 80 a3 60 04    cmp %o5, 4  
40001c30: 12 80 00 18    bne 40001c90 <__mulsf3+0xbc>  
40001c34: 80 a0 60 02    cmp %g1, 2  
40001c38: 03 10 00 2e    sethi %hi(0x4000b800), %g1  
40001c3c: 02 80 00 5c    be 40001dac <__mulsf3+0x1d8>  
40001c40: 90 10 63 50    or %g1, 0x350, %o0 ! 4000bb50 <__t  
40001c44: da 07 bf dc    ld [ %fp + -36 ], %o5  
40001c48: c2 07 bf ec    ld [ %fp + -20 ], %g1  
40001c4c: 82 18 40 0d    xor %g1, %o5, %g1
```

Les linkers

Les linker scripts

- Le travail d'édition de liens du linker est défini par un script (linker script).
- Ce script est écrit dans le langage de commande du linker.
- Le but principal du linker script est de décrire comment les sections des fichiers d'entrée (.text, .data, .bss, ...) doivent être mappées dans le fichier de sortie et de contrôler la disposition de la mémoire du fichier de sortie.
- Le linker utilise toujours un script de linker. Si vous n'en fournissez pas vous-même, l'éditeur de liens utilisera un script par défaut qui est intégré dans l'exécutable de l'éditeur de liens.
 - L'option verbose de GCC permet d'afficher sur la console le contenu du linker script.

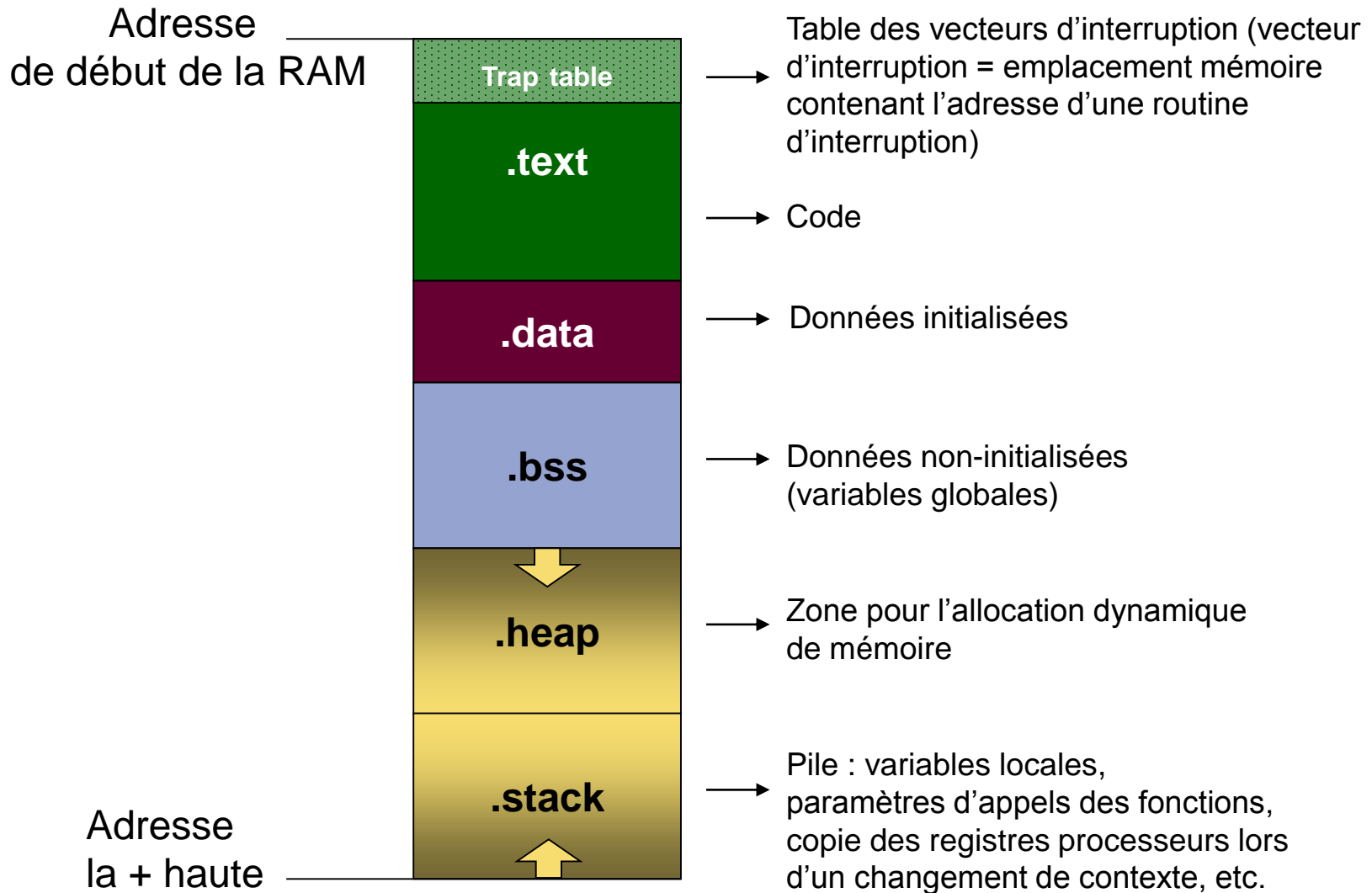
Les linkers

Les linker scripts

- Dans le domaine de l'embarqué, il est crucial de maîtriser parfaitement l'organisation mémoire des applications développées et donc de maîtriser l'écriture des linker scripts.
- L'option `-Xlinker -T ../linkerscript/improved_linker_script.txt` permet de spécifier un linker script spécifique
- <https://sourceware.org/binutils/docs/ld/Scripts.html>

Les linkers

Organisation générale de la mémoire



Les linkers

Organisation générale de la mémoire

- Par défaut, il est défini :
 - Une section `.text` pour le code
 - Une section `.data` pour les données initialisées
 - Une section `.bss` pour les données non-initialisées (variables globale)
- Quand on développe un logiciel embarqué, il est souvent utile de définir plusieurs sections différentes pour le code et les données afin de rendre modulaire et plus facilement reprogrammable l'application.
- C'est le rôle du linker script de définir les différentes sections en les nommant et en spécifiant leur positionnement mémoire (de façon absolue ou relative).

Les linkers

Les linker scripts : exemple 1

```
SECTIONS
{
    . = 0x40000000;
    .text : {
        *(.text)
    }
    . = 0x40100000;
    .data : {
        *(.data)
    }
    .bss : {
        *(.bss)
    }
}
```

Le code est chargé à
l'adresse 0x40000000

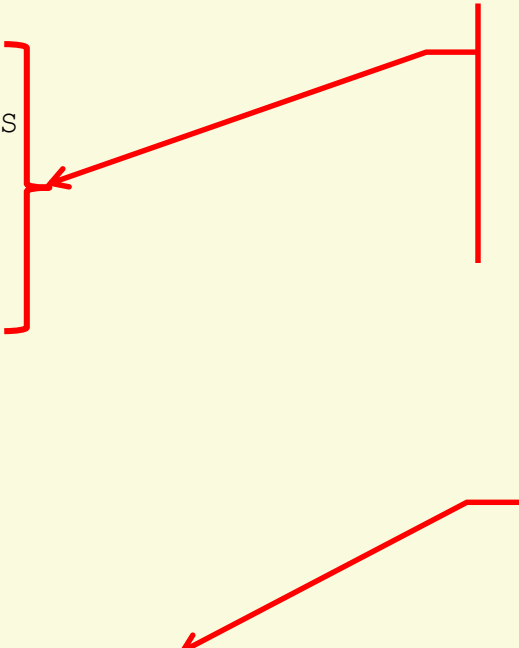
Les données sont
chargées à l'adresse
0x40100000

Les linkers

Les linker scripts : exemple 2

```
MEMORY
{
    ram      : ORIGIN = 0x40000000, LENGTH = 4096K
}

SECTIONS
{
    .text    : {
        CREATE_OBJECT_SYMBOLS
        *(.text)
    } > ram
    .text_factorial : {
        *(.text)
    } > ram
    .data    : {
        *(.data)
    } > ram
    .bss     :
    {
        . = ALIGN(0x8);
        *(.bss)
    } > ram
    .bss_factorial_array 0x40100000 : {
        . = ALIGN(0x8);
        *(.bss)
    } > ram
}
```



Le code sera chargé dans 2 sections distinctes : la section `.text` ou la section `.text_factorial`

Une section `.bss_factorial_array` est ajoutée en plus de la section `.bss` par défaut. Elle commence à l'adresse `0x40100000`

Définition de sections spécifiques via le mot clé `__attribute__`

- Comment va-t-on pouvoir spécifier que telle ou telle fonction devra être localisée dans telle ou telle section ?
- Comment va-t-on pouvoir spécifier que telle ou telle variable globale devra être localisée dans telle ou telle section ?
- Par défaut, si l'on ne fait rien, toutes les fonctions seront localisées dans la section `.text` et toute les variables dans la section `.bss`
- Le mécanisme consiste à définir des attributs de fonctions et de variables en employant dans le code C le mot clé `__attribute__`

Définition de sections spécifiques via le mot clé __attribute__

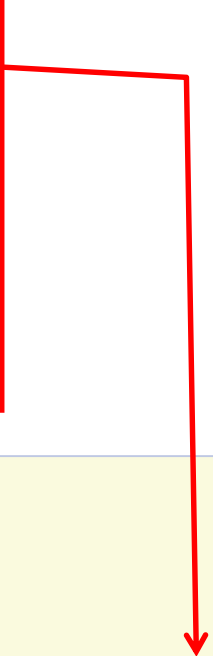
La fonction factorial() sera placée dans la section .text_factorial.
L'édition de lien échouera si cette section n'est pas définie dans le linker script.

```
#ifndef FACTORIAL_H_
#define FACTORIAL_H_

#include <stdint.h>

uint32_t factorial( uint32_t n) __attribute__ ((section (".text_factorial")));

#endif /* FACTORIAL_H_ */
```



Définition de sections spécifiques via le mot clé __attribute__

Le tableau factorial_array[] sera placé dans la section .bss_factorial_array. L'édition de lien échouera si cette section n'est pas définie dans le linker script.



```
#define FACTORIAL_ARRAY_SIZE 10

uint32_t factorial_array[FACTORIAL_ARRAY_SIZE] __attribute__((section(\".bss_factorial_array\")));
packet_header packet_header1;
alt_packet_header alt_packet_header1;

int main( void )
{
    store_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
    display_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
    printf("5.67 * 5.67 = %f\\n", square(5.67));
    packet_header1.node_id = 0xFE;
    packet_header1.protocol_id = 0x1234;
    packet_header1.spare = 0x00;

    printf("sizeof(packet_header) = %d\\n", sizeof(packet_header));
    printf("sizeof(alt_packet_header) = %d\\n", sizeof(alt_packet_header));
    return 0;
}
```

Les linkers

Les linker scripts : exemple 3

```
.text : {  
    *(EXCLUDE_FILE(*Gsc*.o *librtemsbsp.a:*.*.o) .text*)  
} > sdram  
  
.text_rtems : {  
    *librtemsbsp.a:*.*(.text*)  
} > sdram  
  
.text_GSC : {  
    *Gsc*.o(EXCLUDE_FILE(*Gsc*Inst.o) .text*)  
} > sdram
```

Tous les fichiers objets de la bibliothèque librtemsbsp.a seront localisés dans la section .text_rtems.

Tous les fichiers objets dont le nom répond au pattern « *Gsc*.o » sauf ceux répondant au pattern *Gsc*Inst.o seront localisés dans la section .text_GSC.

Cette technique de placement agit au niveau des fichiers objets ou des archives contrairement à la technique utilisant le mot clé `__attribute__` qui permet d'agir à un niveau plus fin (celui de la fonction ou de la variable).

Les linkers

Les linker maps

- L'option `-Xlinker -Map=map.txt` du linker permet de générer une cartographie (« map ») mémoire de l'application construite.
- Elle permet de retrouver l'adresse et la taille de chaque section définie dans le linker script et de chaque fonction.
- Cette map est très utile pour les développeurs de logiciels embarqués.
- Elle permet d'effectuer un certain nombre de vérifications sur l'organisation mémoire de l'application.

Les linkers

Les linker maps

- Elle facilite le débogage de l'application.
- En cas de crash de l'application, il est possible de récupérer l'adresse contenue dans le registre Program Counter (PC), c'est-à-dire l'adresse de la dernière instruction exécutée.
- Connaissant cette adresse, on pourra utiliser la linker map pour identifier la fonction défaillante (même en dehors de l'utilisation d'un debugger)

Les linkers

Les linker maps : exemple

```
...  
.text          0x40000000      0xb690  
CREATE_OBJECT_SYMBOLS  
*(.text .text.*)  
.text          0x40000000      0x1000 /sparc-elf/lib/soft/v8/locore_mvt.o  
                0x40000000      _trap_table  
                0x40000000      start  
.text          0x40001000      0x88 /sparc-elf/lib/soft/v8/crt0.o  
                0x40001000      _start  
.text          0x40001088      0xec /lib/gcc/sparc-elf/3.4.4/soft/v8/crtbegin.o  
.text          0x40001174      0x838 /lib/gcc/sparc-elf/3.4.4/soft/v8/pnpinit.o  
                0x40001174      ambapp_addr_from  
                0x400018ec      pnpinit  
                0x40001870      find_apbslv  
.text          0x400019ac      0x54 src\store_factorial.o  
                0x400019ac      store_factorial  
.text          0x40001a00      0x18 src\square.o  
                0x40001a00      square  
.text          0x40001a18      0x94 src\main.o  
                0x40001a18      main  
.text          0x40001aac      0x40 src\display_factorial.o  
                0x40001aac      display_factorial  
...  
.text_factorial 0x4000b690      0x30  
*(.text .text.*)  
.text_factorial  
                0x4000b690      0x30 src\factorial.o  
                0x4000b690      factorial
```

La fonction factorial()
a bien été placée
dans la section
.text_factorial.

Les linkers

Les linker maps : exemple (suite)

```
.bss_factorial_array
0x40100000      0x28 src\main.o
0x40100000      factorial_array
```

Le tableau factorial_array a bien été placée dans la section .bss_factorial_array.

La taille de 0x28 = 40 octets correspond à la taille du tableau (10 x 4), un uint32_t étant codé sur 4 octets (32 bits) uint32_t

Automatisation de la construction des exécutables et bibliothèques

- La compilation d'un exécutable peut impliquer un grand nombre de fichiers source et donc des temps de compilation assez long.
- Généralement, on utilise des outils permettant d'automatiser le processus de construction des exécutables ou des bibliothèques.
- Ces outils gèrent des dépendances entre les fichiers sources et ne recompilent que les fichiers qui ont besoin de l'être :
 - Fichiers sources modifiés depuis la dernière compilation
 - Fichiers sources qui dépendent de fichiers qui ont été modifiés depuis la dernière compilation

Automatisation de la construction des exécutables et bibliothèques

- Plusieurs solutions existent :
 - L'utilitaire make et les fichiers makefile :
 - Solution intégrée dans la suite GCC
 - Le développeur doit maintenir pour chaque projet un fichier makefile contenant des règles de type

```
cible : dépendance1 dépendance2...  
    <tabulation>commande1  
    <tabulation>commande2  
    <tabulation>...
```
 - Dans les environnements de développement intégrés, la production des fichiers makefile est automatisée → le développeur ne s'en occupe plus.
 - Scons
 - Alternative à l'utilitaire make basée sur des scripts pythons
 - <https://scons.org/>

Aperçu de binutils

- Principaux outils de la suite binutils :
 - ld : linker
 - as : assembleur
 - ar : utilitaire pour créer, modifier et extraire des archives (bibliothèques)
 - objdump : affiche les informations contenues dans les fichiers objets
 - objcopy : copie et transforme les fichiers objets
 - nm : liste les symboles contenus dans des fichiers objets
 - readelf : affiche les informations contenues dans des fichiers objets au format ELF
 - size : liste les tailles des différentes sections contenues dans un fichier objet
- Les outils ld, as et ar peuvent être appelés séparément ou de façon intégrée à GCC.

objdump

- Objdump est l'outil de la suite binutils permettant d'afficher les informations contenues dans les fichiers objets
 - <https://sourceware.org/binutils/docs/binutils/objdump.html>
 - L'option -d permet de produire la sortie désassemblée du fichier objet.
 - L'option -S, combinée à l'option -d, permet de produire une sortie dans laquelle le code C est affiché de façon entrelacée avec le code assembleur.

objdump

Exemple 1

```
sparc-elf-objdump.exe -d factorial.o > factorial.txt
```

```
C:\workspace\factorial\Debug_00\src\factorial.o:      file format elf32-sparc
```

```
Disassembly of section .text_factorial:
```

```
00000000 <factorial>:
```

```
  0:   9d e3 bf 90      save  %sp, -112, %sp
  4:   f0 27 a0 44      st   %i0, [ %fp + 0x44 ]
  8:   82 10 20 01      mov  1, %g1
  c:   c2 27 bf f4      st   %g1, [ %fp + -12 ]
 10:   82 10 20 01      mov  1, %g1
 14:   c2 27 bf f0      st   %g1, [ %fp + -16 ]
 18:   fa 07 bf f0      ld   [ %fp + -16 ], %i5
 1c:   c2 07 a0 44      ld   [ %fp + 0x44 ], %g1
 20:   80 a7 40 01      cmp  %i5, %g1
 24:   18 80 00 0b      bgu  50 <factorial+0x50>
 28:   01 00 00 00      nop
 2c:   fa 07 bf f4      ld   [ %fp + -12 ], %i5
 30:   c2 07 bf f0      ld   [ %fp + -16 ], %g1
 34:   82 5f 40 01      smul %i5, %g1, %g1
 38:   c2 27 bf f4      st   %g1, [ %fp + -12 ]
 3c:   c2 07 bf f0      ld   [ %fp + -16 ], %g1
 40:   82 00 60 01      inc  %g1
 44:   c2 27 bf f0      st   %g1, [ %fp + -16 ]
 48:   10 bf ff f4      b   18 <factorial+0x18>
 4c:   01 00 00 00      nop
 50:   c2 07 bf f4      ld   [ %fp + -12 ], %g1
 54:   b0 10 00 01      mov  %g1, %i0
 58:   81 c7 e0 08      ret
 5c:   81 e8 00 00      restore
```

objdump

Exemple 2

```
sparc-elf-objdump.exe -d -S factorial.o > factorial.txt
```

```
uint32_t factorial( uint32_t n)
{
    0:      9d e3 bf 90      save %sp, -112, %sp
    4:      f0 27 a0 44      st %i0, [ %fp + 0x44 ]
        uint32_t result = 1;
    8:      82 10 20 01      mov 1, %g1
    c:      c2 27 bf f4      st %g1, [ %fp + -12 ]
        uint32_t i;
        for (i = 1; i <= n; i++)
    10:      82 10 20 01      mov 1, %g1
    14:      c2 27 bf f0      st %g1, [ %fp + -16 ]
    18:      fa 07 bf f0      ld [ %fp + -16 ], %i5
    1c:      c2 07 a0 44      ld [ %fp + 0x44 ], %g1
    20:      80 a7 40 01      cmp %i5, %g1
    24:      18 80 00 0b      bgu 50 <factorial+0x50>
    28:      01 00 00 00      nop
        {
            result = result * i;
    2c:      fa 07 bf f4      ld [ %fp + -12 ], %i5
    30:      c2 07 bf f0      ld [ %fp + -16 ], %g1
    34:      82 5f 40 01      smul %i5, %g1, %g1
    38:      c2 27 bf f4      st %g1, [ %fp + -12 ]
    3c:      c2 07 bf f0      ld [ %fp + -16 ], %g1
    40:      82 00 60 01      inc %g1
    44:      c2 27 bf f0      st %g1, [ %fp + -16 ]
    48:      10 bf ff f4      b 18 <factorial+0x18>
    4c:      01 00 00 00      nop
        }
        return result;
    50:      c2 07 bf f4      ld [ %fp + -12 ], %g1
}
    54:      b0 10 00 01      mov %g1, %i0
    58:      81 c7 e0 08      ret
    5c:      81 e8 00 00      restore
```

objcopy

- objcopy est l'outil de la suite binutils permettant de copier et transformer les fichiers objets
 - Extraction de sections
 - Production de fichiers SREC
 - <https://sourceware.org/binutils/docs/binutils/objcopy.html>
 - L'option [-O bfdname] permet de produire une sortie au format binaire [-O bin] ou au format SREC [-O srec]
 - L'option [-R sectionpattern] permet d'enlever les sections spécifiées
 - L'option [-j sectionpattern] permet de ne garder que les sections spécifiées

Le format SREC

- Le format SREC est un format de fichier couramment utilisé dans le mode de l'embarqué.
- C'est le format habituellement utilisé pour programmer les PROM, EEPROM et autres mémoires FLASH.
- Le format SREC est un format de type ASCII lisible via un éditeur de texte standard.
- Un fichier SREC contient un ensemble d'enregistrements du type (un enregistrement = une ligne) :


S	Type	Byte Count	Address	Data	Checksum
---	------	------------	---------	------	----------

- [https://en.wikipedia.org/wiki/SREC_\(file_format\)](https://en.wikipedia.org/wiki/SREC_(file_format))

objcopy et le format SREC

```
sparc-elf-objcopy.exe -O srec factorial factorial.srec
```

```
S02B0000433A5C776F726B73706163655C666163746F7269616C5C6F626A636F70795C4F325C666163746F7211
S31540000010A1480000A750000010802BB1AC10200171
S3154000002091D0200001000000010000000100000006
S3154000003091D02000010000000100000001000000F6
S31540000040A14800002910002B81C522300100000084
S31540000050A14800002910002881C523300100000076
S31540000060A14800002910002881C5239C01000000FA
S3154000007091D02000010000000100000001000000B6
S3154000008091D02000010000000100000001000000A6
S31540000090A1480000A750000010802B91AC10200909
S315400000A091D0200001000000010000000100000086
S315400000B091D0200001000000010000000100000076
S315400000C091D0200001000000010000000100000066
S315400000D091D0200001000000010000000100000056
S315400000E091D0200001000000010000000100000046
S315400000F091D0200001000000010000000100000036
S3154000010091D0200001000000010000000100000025
S31540000110AE102001A14800001080294DA7500000D4
S31540000120AE102002A148000010802949A7500000C7
S31540000130AE102003A148000010802945A7500000BA
...
```



Chaque enregistrement (ligne) contient 21 octets (21 = 0x15) soit 4 octets d'adresse, 16 octets de code machine (c'est-à-dire 4 instructions 32 bits) et un checksum de 1 octet

Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)**
 - c. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - d. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Les simulateurs

- Les exécutables produits à partir d'une chaîne de compilation croisée ne peuvent pas s'exécuter directement sur le système d'exploitation hébergeant cette chaîne de compilation.
- Pour tester ces exécutables, on a 2 possibilités :
 - Utiliser une vraie cible matérielle (carte électronique)
 - Utiliser un simulateur
- Les simulateurs sont des applications qui tournent sur le système d'exploitation hébergeant la chaîne de compilation et qui sont capables d'émuler le fonctionnement d'un processeur embarqué.
- L'émulation se fait généralement au niveau de l'instruction machine.

Les simulateurs

- Les avantages des simulateurs sont nombreux :
 - Simulateur = plate-forme virtuelle
 - Indépendance vis-à-vis de la cible matérielle réelle => cela permet d'anticiper les développements
 - Réplication facilitée de l'environnement de test
 - Diminution des coûts de développement
 - Certains simulateurs de processeur peuvent être étendus à l'aide de modules simulant les périphériques
 - Débogage temps réel simplifié => un point d'arrêt entraîne le gel de la time-line d'exécution et n'a donc pas d'impact sur les timing de l'application.

Les simulateurs

- Fonctionnalités permettant d'analyser plus facilement le fonctionnement de l'application :
 - Profiling
 - Mesure de temps d'exécution non intrusive (cycles machine)
 - Mesure de la couverture du code
 - Taux d'occupation du bus mémoire
 - Statistiques sur l'utilisation du cache
 - Gestion des points d'arrêts (break point) et des watchpoints
 - Manipulation de la mémoire
 - Visibilité des registres internes
- Les simulateurs peuvent être utilisés de façon standalone ou via un débogueur de type GDB.

Les simulateurs

Exemple de produits disponibles sur le marché

■ Processeurs ARM

■ Fixed Virtual Platforms

- <https://developer.arm.com/products/system-design/fixed-virtual-platforms>
- <https://www.arm.com/products/development-tools/simulation/fixed-virtual-platforms>
- Cibles simulées : Armv8-A, Armv7-A, Arm Cortex-R, Arm Cortex-M

■ Crossware ARM simulator

- <https://www.crossware.com/arm/simulator>
- Cibles simulées :
 - STMicroelectronic (famille STM32, ...)
 - NXP
 - ATMEL (famille AT91SAM, ...)
 - Silicon Labs (famille EFM32, ...)
 - Kinetis (K22, K70)
 - Nuvoton M0516
 - Analog device (famille ADuC, famille ADSP, ...)

Les simulateurs

Exemple de produits disponibles sur le marché

■ Processeurs ARM (suite)

■ QEMU

- Emulateur multi-plate-forme, open source
- Pas d'émulation au niveau instruction => limitations dans la représentativité des timings et temps d'exécution
- <https://wiki.qemu.org/Documentation/Platforms/ARM>

■ GNU MCU Eclipse QEMU

- <https://gnu-mcu-eclipse.github.io/qemu/>
- Fork QEMU
- Cibles supportées :
 - Cortex-M (famille STM32F, ...)

Les simulateurs

Exemple de produits disponibles sur le marché

■ Processeurs LEON

■ TSIM

- <https://www.gaisler.com/index.php/products/simulators/tsim>
- Simulateur de niveau instruction capable d'émuler les processeurs LEON
- Processeurs simulés : AT697, GR712RC, UT699/E, UT700, AT7913E
- Simulation mono-core seulement

Les simulateurs

Exemple avec TSIM pour cible LEON

■ Chargement d'un exécutable embarqué

```
C:\Program Files (x86)\GRTools>tsim-leon3
```

```
This TSIM evaluation version will expire 2018-12-18
```

```
TSIM/LEON3 SPARC simulator, version 2.0.61 (evaluation version)
```

```
Copyright (C) 2018, Cobham Gaisler - all rights reserved.
```

```
This software may only be used with a valid license.
```

```
For latest updates, go to http://www.gaisler.com/
```

```
Comments or bug-reports to support@gaisler.com
```

```
system frequency: 50.000 MHz
```

```
serial port A on stdin/stdout
```

```
allocated 4096 KiB SRAM memory, in 1 bank
```

```
allocated 32 MiB SDRAM memory, in 1 bank
```

```
allocated 2048 KiB ROM memory
```

```
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
```

```
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
```

```
tsim> load C:\workspace\factorial\Debug_O2\factorial
```

```
section: .text, addr: 0x40000000, size 46736 bytes
```

```
section: .data, addr: 0x4000b6c0, size 2784 bytes
```

```
section: .bss_factorial_array, addr: 0x40100000, size 40 bytes
```

```
section: .text_factorial, addr: 0x4000b690, size 48 bytes
```

```
read 446 symbols
```

```
tsim>
```

Les simulateurs

Exemple avec TSIM pour cible LEON

- Mise à 0 des statistiques d'exécution et démarrage de l'exécution de l'application embarquée

```
tsim> reset perf
tsim> run
starting at 0x40000000
fact(0) = 1
fact(1) = 1
fact(2) = 2
fact(3) = 6
fact(4) = 24
fact(5) = 120
fact(6) = 720
fact(7) = 5040
fact(8) = 40320
fact(9) = 362880
5.67 * 5.67 = 32.148899
sizeof(packet_header) = 6
sizeof(alt_packet_header) = 4

Program exited normally.
tsim>
```

Les simulateurs

Exemple avec TSIM pour cible LEON

■ Affichage des statistiques d'exécution

```
tsim> perf

Cycles      : 1249455
Instructions : 680086
Overall CPI  : 1.84

CPU performance (50.0 MHz) : 27.22 MOPS (27.22 MIPS, 0.00 MFLOPS)
Cache hit rate             : 99.5 % (inst: 99.6, data: 95.6)
AHB bandwidth utilisation  : 85.8 % (inst: 2.4, data: 83.4)
Simulated time            : 24.99 ms
Processor utilisation      : 100.00 %
Real-time performance     : 4.42 %
Simulator performance     : 1.20 MIPS
Used time (sys + user)    : 0.56 s

tsim>
```

Les simulateurs

Exemple avec TSIM pour cible LEON

- Affichage de contenu des registres du processeur (registres internes)

```
tsim> reg
```

	INS	LOCALS	OUTS	GLOBALS
0:	00000000	40100000	0000001E	00000000
1:	4000B67C	4000A940	00000004	00000001
2:	00000000	4000A944	20000000	4000C1A0
3:	00000000	00000000	4000C5F0	FFFFFFF8
4:	80000310	00000000	4000C400	4000AC00
5:	80000200	00000000	FFFFFFFE	00000003
6:	403FFE00	00000000	403FFD98	4000AE08
7:	40001078	00000000	40001A98	00000000


```
psr: F34010C6    wim: 00000001    tbr: 40000800    y: 00000000
```



```
pc: 40000800    91d02000    ta    0x0
```

```
npc: 40000804    01000000    nop
```


Les simulateurs

Exemple avec TSIM pour cible LEON

■ Affichage des registres des périphériques

```
tsim> leon
0x80000000    Memory configuration register 1    0x00000233
0x80000004    Memory configuration register 2    0x81805220
0x80000008    Memory configuration register 3    0x00000000
0x80000104    UART 1 status register            0x00000006
0x80000108    UART 1 control register           0x00000003
0x8000010c    UART 1 scaler reload register      0x00000000
0x80000200    Interrupt level register          0x00000000
0x80000204    Interrupt pending register        0x00000000
0x80000208    Interrupt force register          0x00000000
0x80000240    Interrupt mask register           0x00000000
0x80000300    Timer scaler counter register      0x0000002c
0x80000304    Timer scaler reload register       0x00000031
0x80000308    Timer configuration register       0x00000142
0x80000310    Timer 1 counter register          0xffff9e62
0x80000314    Timer 1 reload register           0xffffffff
0x80000318    Timer 1 control register          0x00000003
0x80000320    Timer 2 counter register          0x00000000
0x80000324    Timer 2 reload register           0x00000000
0x80000328    Timer 2 control register          0x00000000
0x80000904    UART 2 status register            0x00000006
0x80000908    UART 2 control register           0x00000003
0x8000090c    UART 2 scaler reload register      0x00000000
CTRL         Cache control register            0x0081000f
ICCFG        Icache config register            0x10220000
DCCFG        Dcache config register            0x18220000
ASR17        Processor config register         0x00000f07
ASR18        MAC lsb register                  0x00000000
```

Les simulateurs

Exemple avec TSIM pour cible LEON

- Ajout d'un breakpoint à l'adresse 0x400b690 sur la fonction factorial()
- Activation du buffer de trace des instructions
- Affichage de l'historique après l'arrêt sur le breakpoint

```
tsim> break 0x4000b690
breakpoint 1 at 0x4000b690:
tsim> hist 10
trace history length = 10
tsim> run
starting at 0x40000000

breakpoint 1
tsim> hist
1146887 40001a20 90142000 or      %l0, %o0
1146898 40001a24 7fffffff02 call   0x400019ac
1146899 40001a28 9210200a mov    10, %o1
1146900 400019ac 9de3bf98 save   %sp, -104, %sp
1146901 400019b0 a0102000 mov    0, %l0
1146912 400019b4 80a40019 cmp    %l0, %i1
1146913 400019b8 1a800010 bcc    0x400019f8
1146916 400019bc 01000000 nop
1146917 400019c0 40002734 call   0x4000b690
1146928 400019c4 90100010 mov    %l0, %o0
tsim>
```

Les simulateurs

Exemple avec TSIM pour cible LEON

- Affichage de la backtrace après l'arrêt sur un breakpoint

```
tsim> run
starting at 0x40000000

breakpoint 1
tsim> bt
      %pc          %sp
#0    0x4000b690    0x403ffd30    + 0x0
#1    0x400019c0    0x403ffd30    store_factorial + 0x10
#2    0x40001a24    0x403ffd98    main + 0x8
#3    0x40001070    0x403ffe00    zerobss + 0x50
#4    0x4000ae1c    0x403ffe40    slavego + 0x4
tsim>
```

Les simulateurs

Exemple avec TSIM pour cible LEON

■ Désassemblage de la fonction factorial()

```
tsim> dis 0x4000b690
```

4000b690	9a100008	mov	%o0, %o5
4000b694	90102001	mov	1, %o0
4000b698	80a2000d	cmp	%o0, %o5
4000b69c	18800007	bgu	0x4000b6b8
4000b6a0	82102001	mov	1, %g1
4000b6a4	905a0001	smul	%o0, %g1, %o0
4000b6a8	82006001	add	%g1, 1, %g1
4000b6ac	80a0400d	cmp	%g1, %o5
4000b6b0	28bffffe	blue,a	0x4000b6a8
4000b6b4	905a0001	smul	%o0, %g1, %o0
4000b6b8	81c3e008	retl	
4000b6bc	01000000	nop	

Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Les débogueurs

- Les débogueurs comme GDB permettent de réaliser les tâches suivantes afin d'investiguer les bugs logiciels ou analyser le comportement du code :
 1. Démarrer le programme en réalisant un certain nombre d'initialisation au démarrage qui pourront influencer le comportement du programme (ex. : initialisation de la mémoire)
 2. Mettre en pause l'exécution du programme en définissant des points d'arrêts (qui se déclenchent systématiquement ou sur condition)
 3. Examiner l'état de l'application quand le programme s'est arrêté sur un point d'arrêt (état de la mémoire, contenu des variables, contenu des registres du processeur et des registres des périphériques, pile des appels, backtrace, etc..)
 4. Modifier l'état de l'application (écriture dans la mémoire, modification de la valeurs de certaines variables ou registres, etc.

Les débogueurs

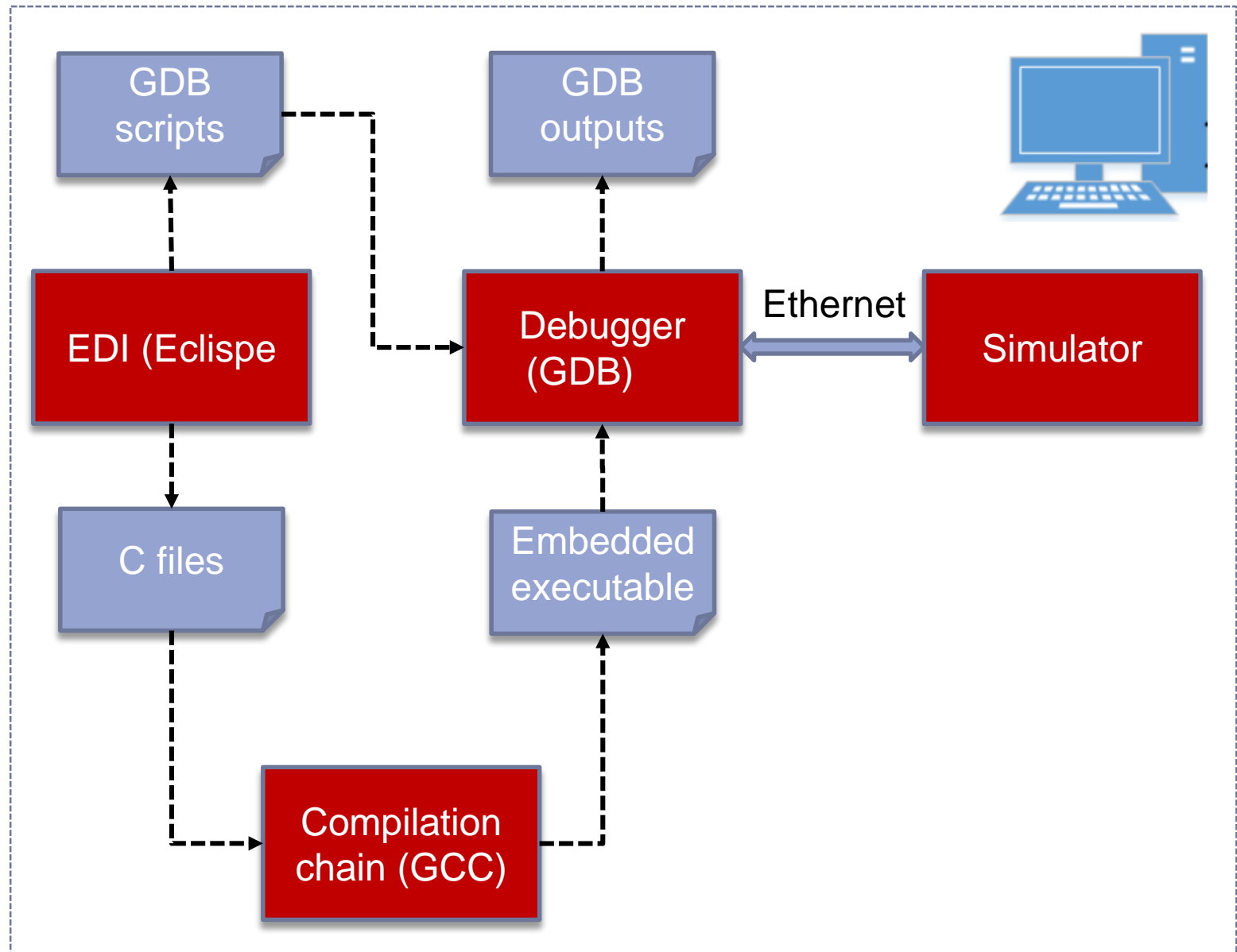
- Un des débogueurs les plus utilisés dans le monde du logiciel embarqué est GDB :
 - <https://www.gnu.org/software/gdb/>
- GDB peut se connecter aussi bien aux simulateurs de processeurs qu'aux moniteurs interfacés à des vraies cartes processeur
 - La connexion se fait via une socket TCP.
 - Les simulateurs et les moniteurs embarquent un serveur GDB.

Les débogueurs

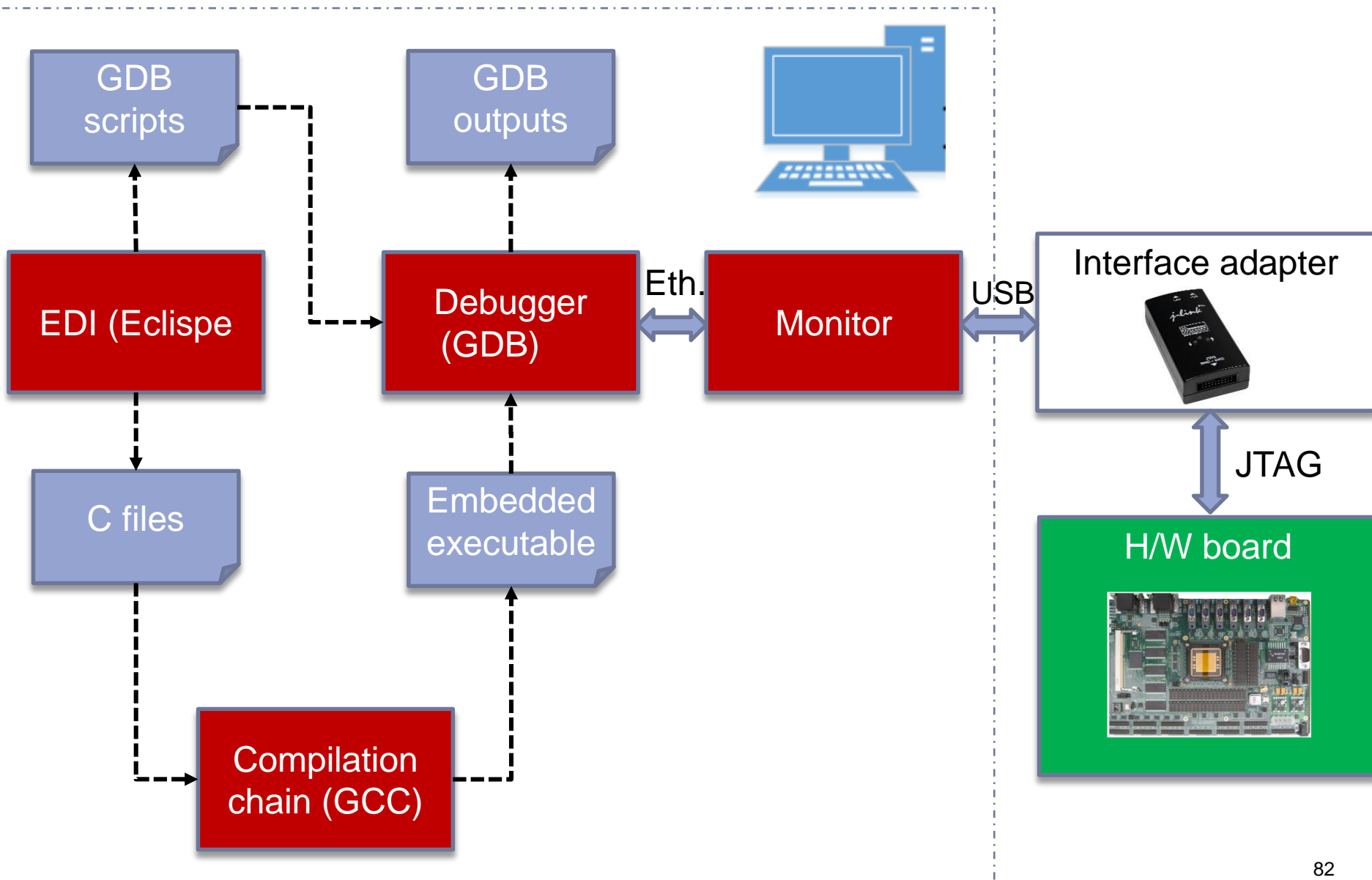
Utilisation interactive versus utilisation scriptée

- GDB peut s'utiliser en mode interactif à travers une console ou à travers une surcouche graphique
 - Surcouche graphique généralement présente dans les environnements de développement intégré
- GDB peut aussi s'utiliser à travers un script programmé à l'avance.
 - Cette approche-là est très puissante car elle permet d'automatiser des séquences complètes de débogage.

Chaîne de test : GDB + simulateur



Chaîne de test : GDB + Moniteur + carte H/W

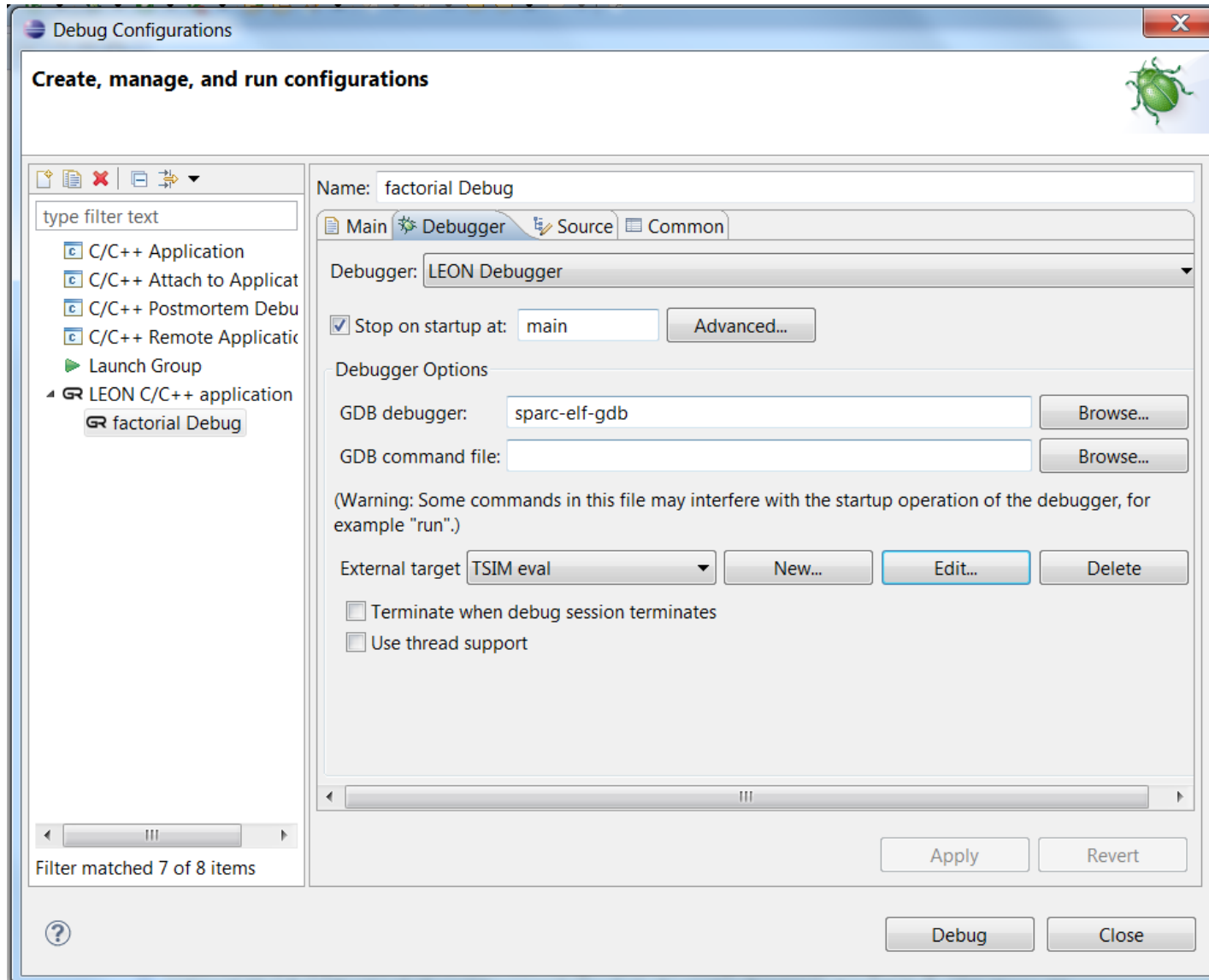


Les débogueurs

Limitations

- Influence des options de compilation
 - Un programme compilé en -O2 sera plus difficile à déboguer qu'un programme non optimisé (compilé en -O0)
 - Le code généré est plus compacte
 - Il est plus difficile de positionner des points d'arrêt au sein des fonctions
- L'approche « stop and go » est fortement intrusive quand on travaille avec une carte processeur réelle car chaque point d'arrêt brise la timeline d'exécution et fausse les timings d'exécution
 - Les approches de type « streaming trace » avec la technologie ARM CoreSight permet de contourner ces limites.

Les débogueurs - Utilisation interactive via une surcouche graphique



Les débogueurs - Utilisation interactive via une surcouche graphique

The screenshot shows the Eclipse IDE with a GDB debug session. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar contains various icons for debugging and development. The main window is divided into several panes:

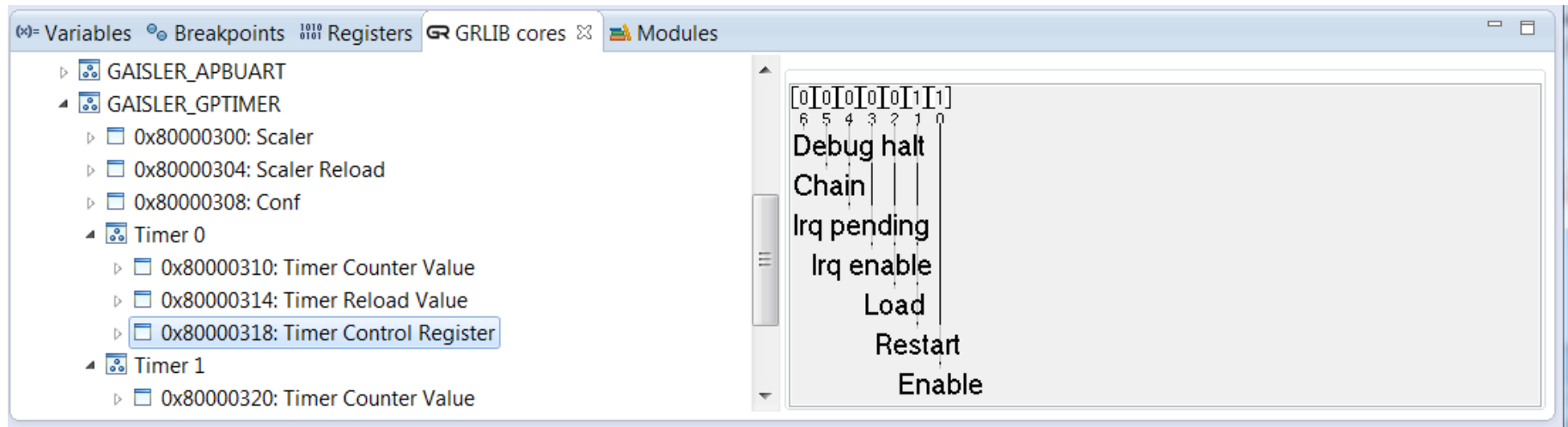
- Debug Console:** Shows the debug session for 'factorial Debug [LEON C/C++ application]'. It lists the LEON Debugger (Suspended), Thread [0] (Suspended), and the current execution point at line 27 of main.c:27 0x40001a48.
- Variables Window:** Displays the current state of variables. The table below shows the values of several variables.
- Source Editor:** Shows the source code of main.c. The current line is 27, which is highlighted. The code includes declarations for factorial_array, packet_header, and alt_packet_header, followed by the main function.
- Outline View:** Shows the project structure, including stdio.h, stdlib.h, stdint.h, store_factorial.h, display_factorial.h, packet_header.h, square.h, and the main function.

Name	Value
f30	0.0
f31	0.0
y	0
psr	-213905178
wim	2
tbr	1073741824

```
20
21 uint32_t factorial_array[FACTORIAL_ARRAY_SIZE] __attribute__((section(".bss_factorial_array")));
22 packet_header packet_header1;
23 alt_packet_header alt_packet_header1;
24
25 int main( void )
26 {
27     store_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
28     display_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
29     printf("5.67 * 5.67 = %f\n", square(5.67));
30     packet_header1.node_id = 0xFE;
31     packet_header1.protocol_id = 0x1234;
32     packet_header1.square = 0x00;
```

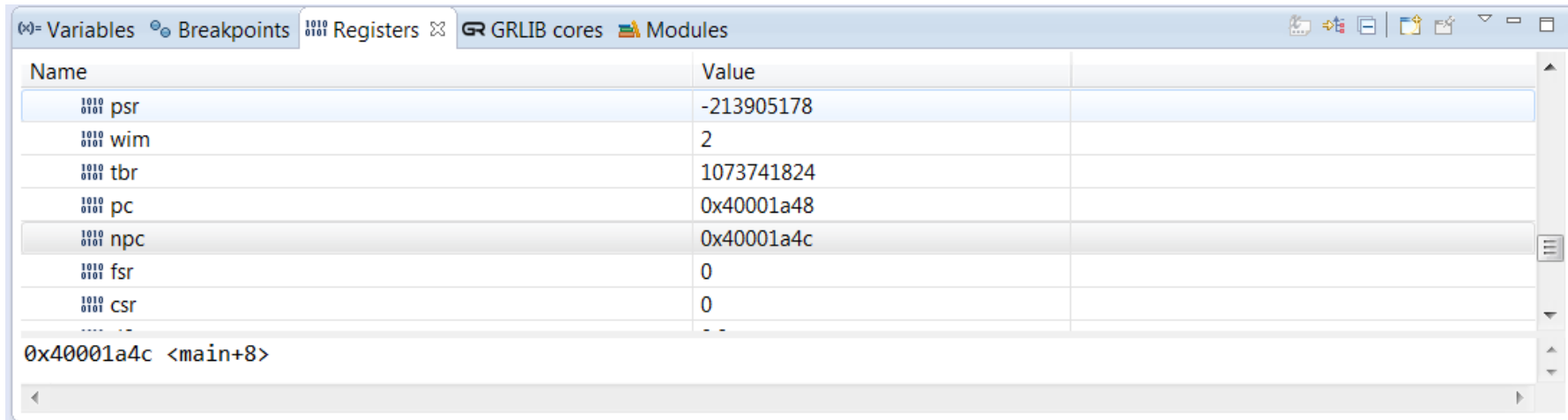
Les débogueurs - Utilisation interactive via une surcouche graphique

- Visualisation des registres des périphériques :



Les débogueurs - Utilisation interactive via une surcouche graphique

- Visualisation des registres du processeur :



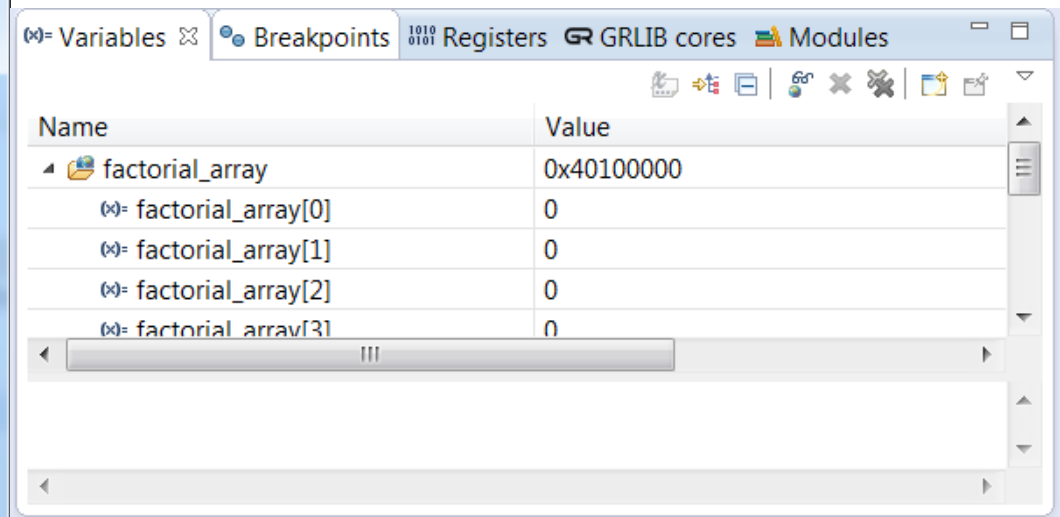
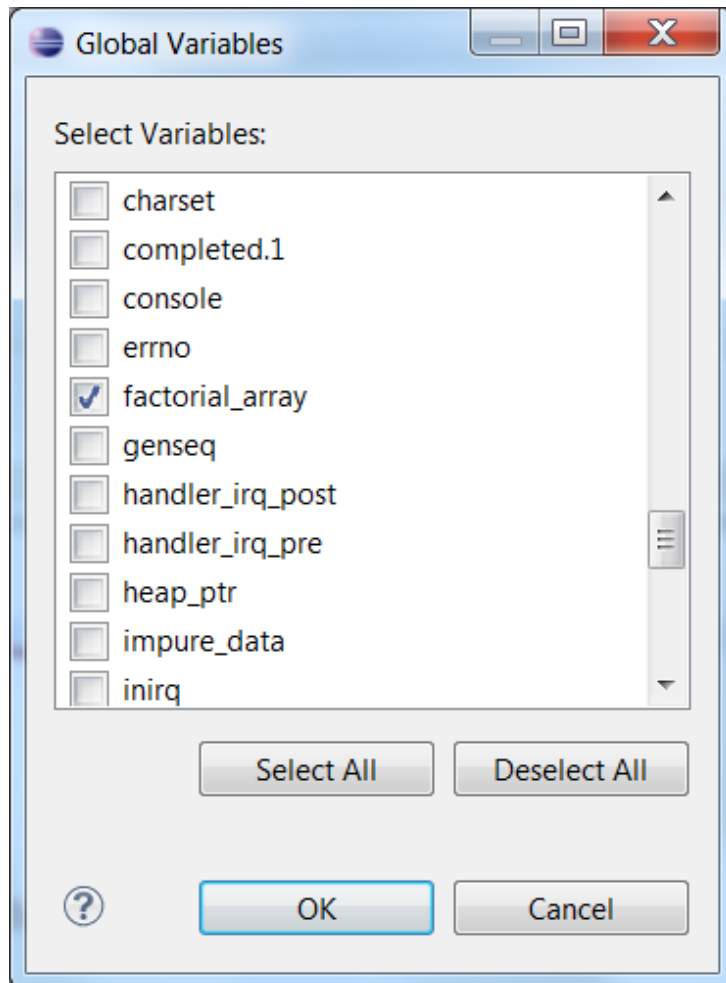
The screenshot shows a debugger window with the 'Registers' tab selected. The window has a toolbar with icons for refreshing, adding, and deleting registers. The main area displays a table of registers with their names, values, and a column for comments. The registers listed are psr, wim, tbr, pc, npc, fsr, and csr. The 'npc' register is highlighted. Below the table, the current instruction is shown as '0x40001a4c <main+8>'.

Name	Value	
psr	-213905178	
wim	2	
tbr	1073741824	
pc	0x40001a48	
npc	0x40001a4c	
fsr	0	
csr	0	

0x40001a4c <main+8>

Les débogueurs - Utilisation interactive via une surcouche graphique

- Visualisation des variables du programme :



Les débogueurs - Utilisation interactive via une surcouche graphique

The screenshot displays the Eclipse IDE interface during a debug session. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar contains various icons for file operations, editing, and debugging.

The **Debug** panel on the left shows the following structure:

- factorial Debug [LEON C/C++ application]
 - LEON Debugger (28/08/18 09:22) (Suspended)
 - Thread [0] (Suspended: Breakpoint hit)
 - 1 main() main.c:28 0x40001a5c
 - sparc-elf-gdb (28/08/18 09:22)
 - C:\workspace\factorial\Debug\factorial (28/08/18 09:22)
- TSIM eval [Gaisler Tools]
 - tsim-leon3

The **Variables** panel on the right shows the following data:

Name	Value
factorial_array	0x40100000
factorial_array[0]	1
factorial_array[1]	1
factorial_array[2]	2
factorial_array[3]	6
factorial_array[4]	24
factorial_array[5]	120

The **main.c** editor shows the following code:

```
21 uint32_t factorial_array[FACTORIAL_ARRAY_SIZE] __attribute__((section (".bss_factorial_array")));
22 packet_header packet_header1;
23 alt_packet_header alt_packet_header1;
24
25 int main( void )
26 {
27     store_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
28     display_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
29     printf("5.67 * 5.67 = %f\n", square(5.67));
30     packet_header1.node_id = 0xFE;
31     packet_header1.protocol_id = 0x1234;
32     packet_header1.square = 0x00;
```

The **Outline** panel on the right shows the following structure:

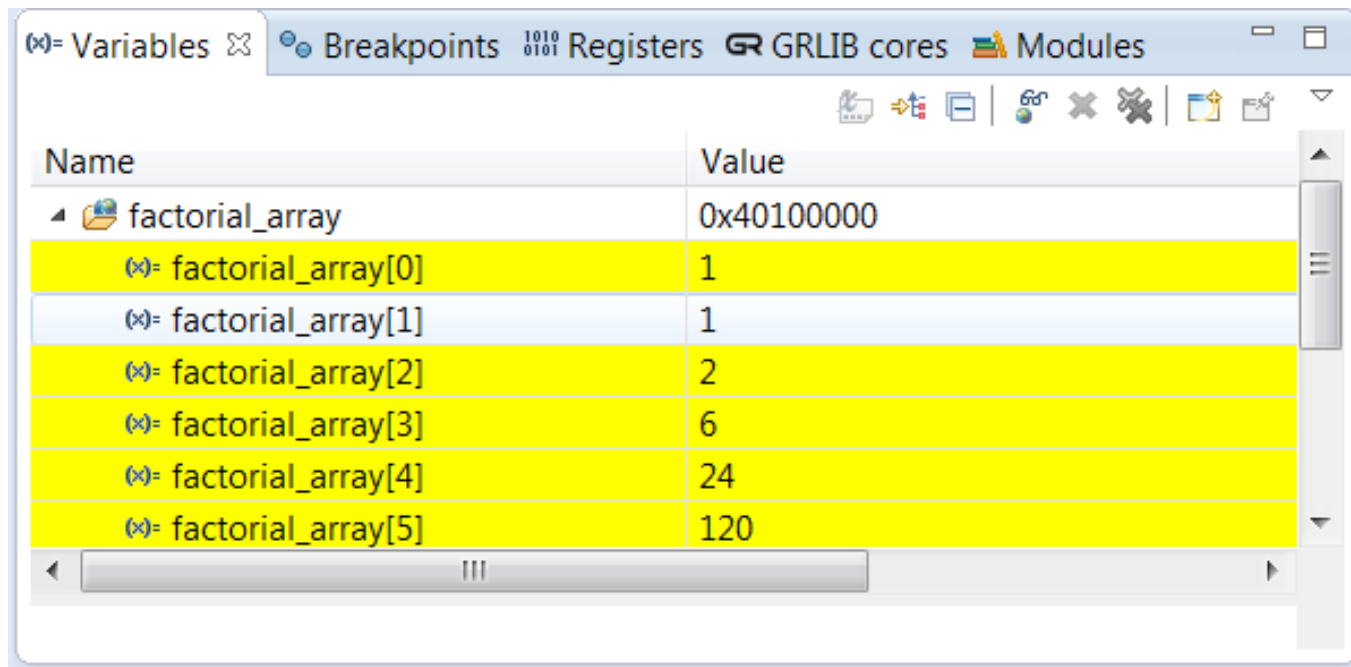
- stdio.h
- stdlib.h
- stdint.h
- store_factorial.h
- display_factorial.h
- packet_header.h
- square.h
- # FACTORIAL_ARRAY_SIZE
- factorial_array : uint32_t[]
- packet_header1 : packet_header
- alt_packet_header1 : alt_packet_header
- main(void) : int

The **Console** panel at the bottom shows the following output:

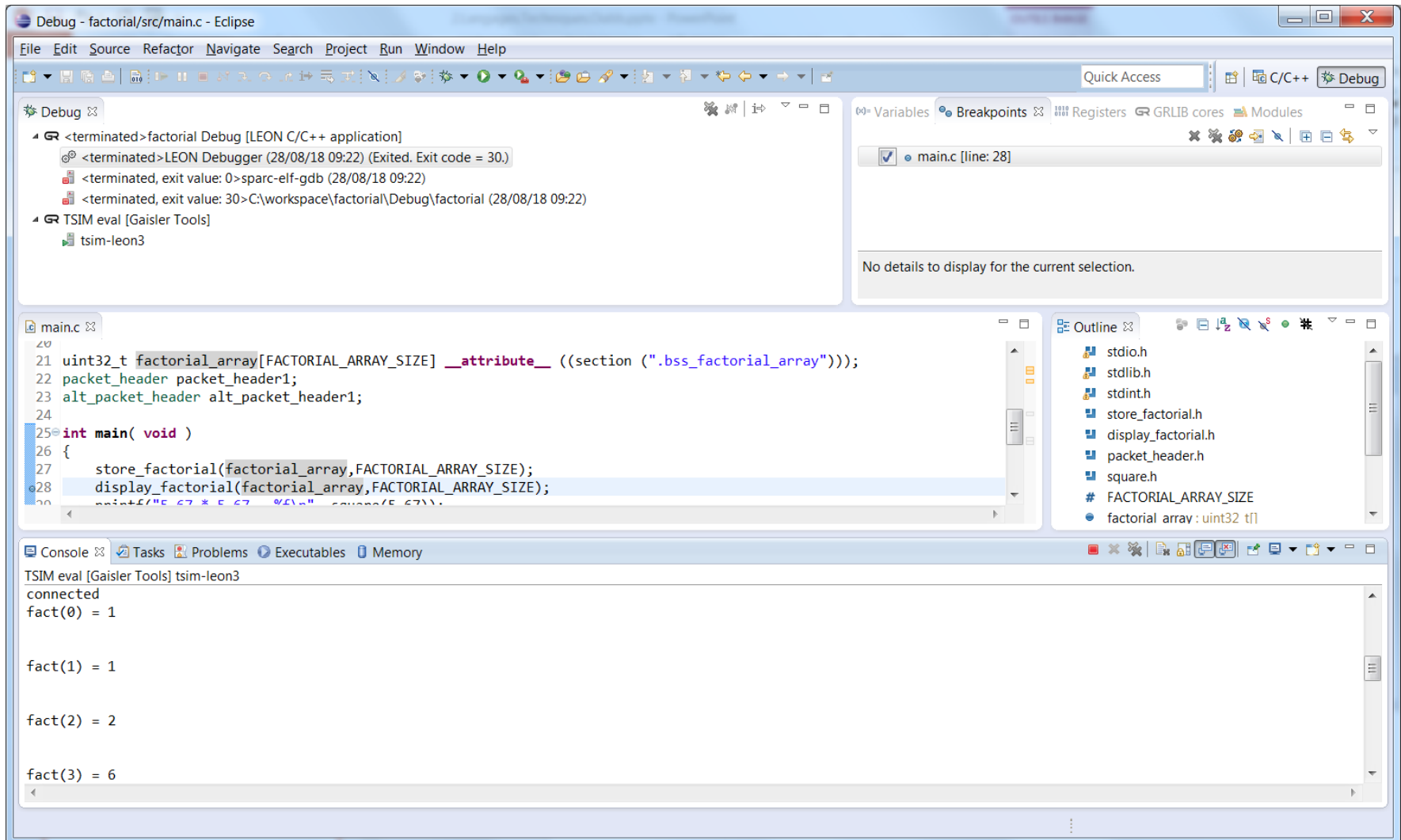
```
factorial Debug [LEON C/C++ application] sparc-elf-gdb (28/08/18 09:22)
Previous frame identical to this frame (corrupt stack?)
The target endianness is set automatically (currently big endian)
Attempt to use a type name as an expression
```

Les débogueurs - Utilisation interactive via une surcouche graphique

- Visualisation des variables du programme :



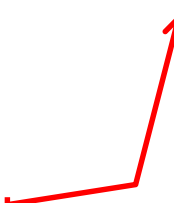
Les débogueurs - Utilisation interactive via une surcouche graphique



Les débogueurs – Utilisation de GDB à partir d'un script

- GDB peut être démarré avec un script de commande :

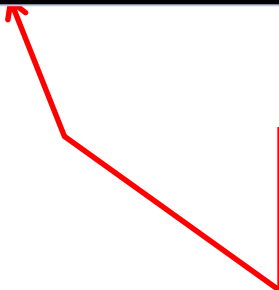
```
sparc-elf-gdb C:\workspace\factorial\Debug\factorial  
-d C:\workspace\factorial\src  
-batch -x C:\workspace\factorial\tests\gdb_batch_001.txt
```

- L'option -d indique dans quel répertoire se trouve les sources de l'application.
 - L'option -batch -x permet de spécifier un script de commandes gdb qui sera exécuté au démarrage.
- 

Les débogueurs – Utilisation de GDB à partir d'un script

- Si l'on veut connecter GDB à un simulateur de processeur ou à un moniteur, il faut d'abord lancer le simulateur ou le moniteur en activant le serveur GDB embarqué dans ces outils.
- Exemple avec le simulateur de processeur LEON TSIM :

```
tsim-leon3 -gdb
```

- 
- L'option -gdb de TSIM permet d'activer le serveur GDB embarqué dans TSIM
→ TSIM est piloté par GDB.
 - TSIM ouvre une socket TCP sur le port 1234 (par défaut).

Les débogueurs – Utilisation de GDB à partir d'un script

■ Code source de l'application testée :

```
#include "factorial.h"
#include "store_factorial.h"

void store_factorial( uint32_t *
array, uint32_t n)
{
    uint32_t i = 0;
    for ( i=0 ; i < n ; i++)
    {
        array[i] = factorial(i);
    }
}
```

```
#include "factorial.h"

uint32_t factorial( uint32_t n)
{
    uint32_t result = 1;
    uint32_t i;
    for (i = 1; i <= n; i++)
    {
        result = result * i;
    }
    return result;
}
```

```
#include "display_factorial.h"
#include <stdio.h>

void display_factorial( uint32_t * array,
uint32_t n)
{
    uint32_t i = 0;

    for ( i = 0; i < n; i++)
    {
        printf("fact(%d) = %d\n", i, array[i]);
    }
}
```

```
#define FACTORIAL_ARRAY_SIZE 10

uint32_t factorial_array[FACTORIAL_ARRAY_SIZE] __attribute__ ((section (".bss_factorial_array")));

int main( void )
{
    store_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
    display_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);

    return 0;
}
```

Les débogueurs – Utilisation de GDB à partir d'un script

■ Exemple de script GDB :

```
set remotetimeout 10000

set logging file gdb_result_001.txt
set logging overwrite on
set logging on
set height 0
set print pretty on
set print array on

tar extended-remote localhost:1234

load

mon perf reset
```

- Déclaration de 2 points d'arrêt via la commande hbreak

```
hbreak factorial
commands
    silent
    printf "\n**** factorial() ****\n"
    cont
end

hbreak display_factorial
commands
    silent
    printf "\n**** display_factorial() ****\n"
    print factorial_array
    cont
end

start
cont

detach
```

- La commande GDB mon (monitor) permet de passer des commandes au simulateur ou au moniteur

Les débogueurs – Utilisation de GDB à partir d'un script

■ Résultat produit par GDB quand on exécute le programme factorial

```
0x00000000 in ?? ()
Loading section .text, size 0xbf00 lma 0x40000000
Loading section .data, size 0xae0 lma 0x4000bf68
Loading section .bss_factorial_array, size 0x28 lma 0x40100000
Loading section .text_factorial, size 0x68 lma 0x4000bf00
Start address 0x40000000, load size 51824
Transfer rate: 414592 bits in <1 sec, 498 bytes/write.
Hardware assisted breakpoint 1 at 0x4000bf08: file ..\src\factorial.c, line 5.
Hardware assisted breakpoint 2 at 0x40001b2c: file ..\src\display_factorial.c, line 13.
Breakpoint 3 at 0x40001a48: file ..\src\main.c, line 27.
main () at ..\src\main.c:27
27      store_factorial(factorial_array, FACTORIAL_ARRAY_SIZE);

**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
**** factorial() ****
```


Les débogueurs – Utilisation de GDB à partir d'un script

- Résultat produit par GDB quand on exécute le programme factorial (suite)

```
**** display_factorial() ****
```

```
$1 = {1,  
1,  
2,  
6,  
24,  
120,  
720,  
5040,  
40320,  
362880}
```

- Affichage du tableau factorial_array[]

```
Program received signal SIGTERM, Terminated.
```

Sommaire

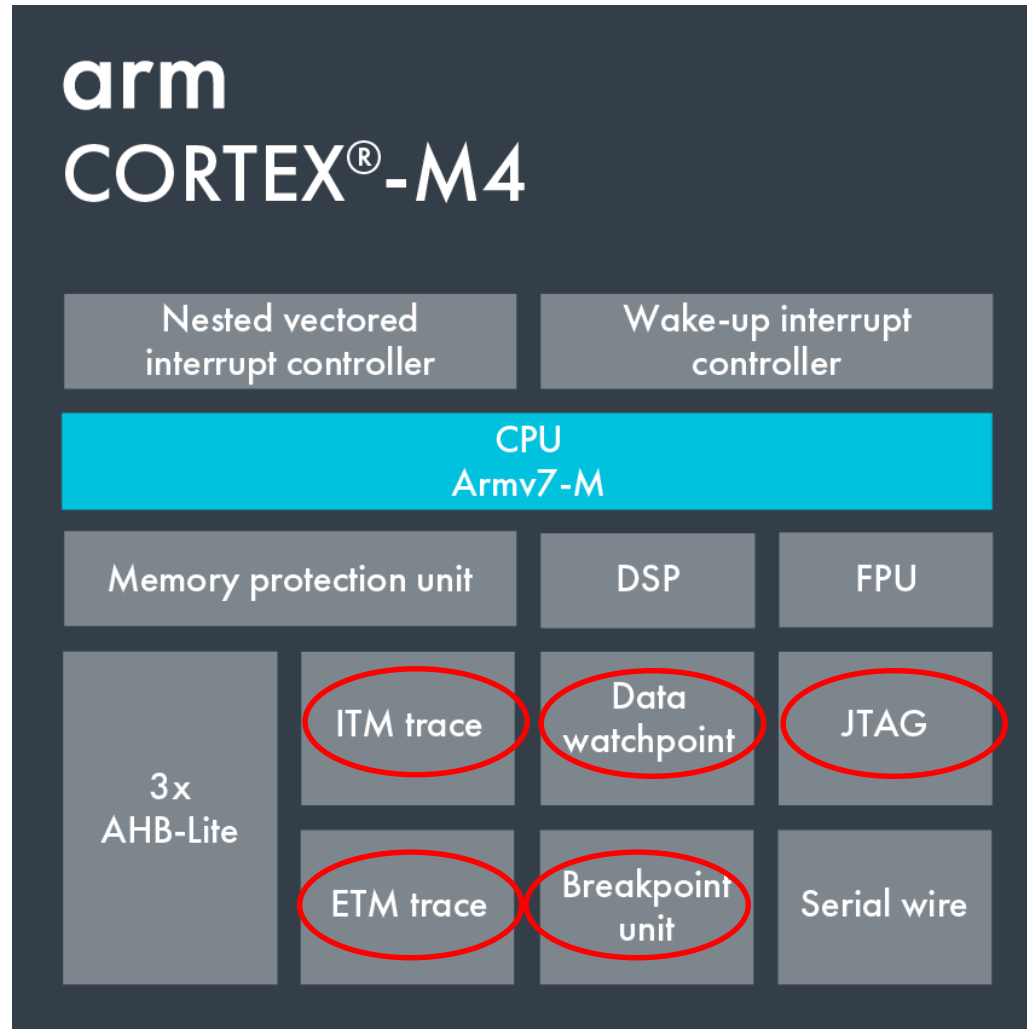
1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Fonctions de débogage intégrées aux processeurs

- Les processeurs embarqués contiennent des fonctions de débogage intégrées qui peuvent être contrôlées par un outil logiciel s'exécutant sur la machine de développement (moniteur)
 - Accès direct à l'intérieur du processeur (points d'arrêt, lecture et écriture des registres internes, des mémoires internes et externes ...) sans perturber ses interactions avec l'extérieur
 - DSU = Debug Support Unit
 - ICE = In-Circuit Emulator
 - ICD In-Circuite Debugger
- Ces fonctions de débogage intégrées sont accessibles via une interface de débogage et un protocole de débogage (JTAG, SWD, ...).

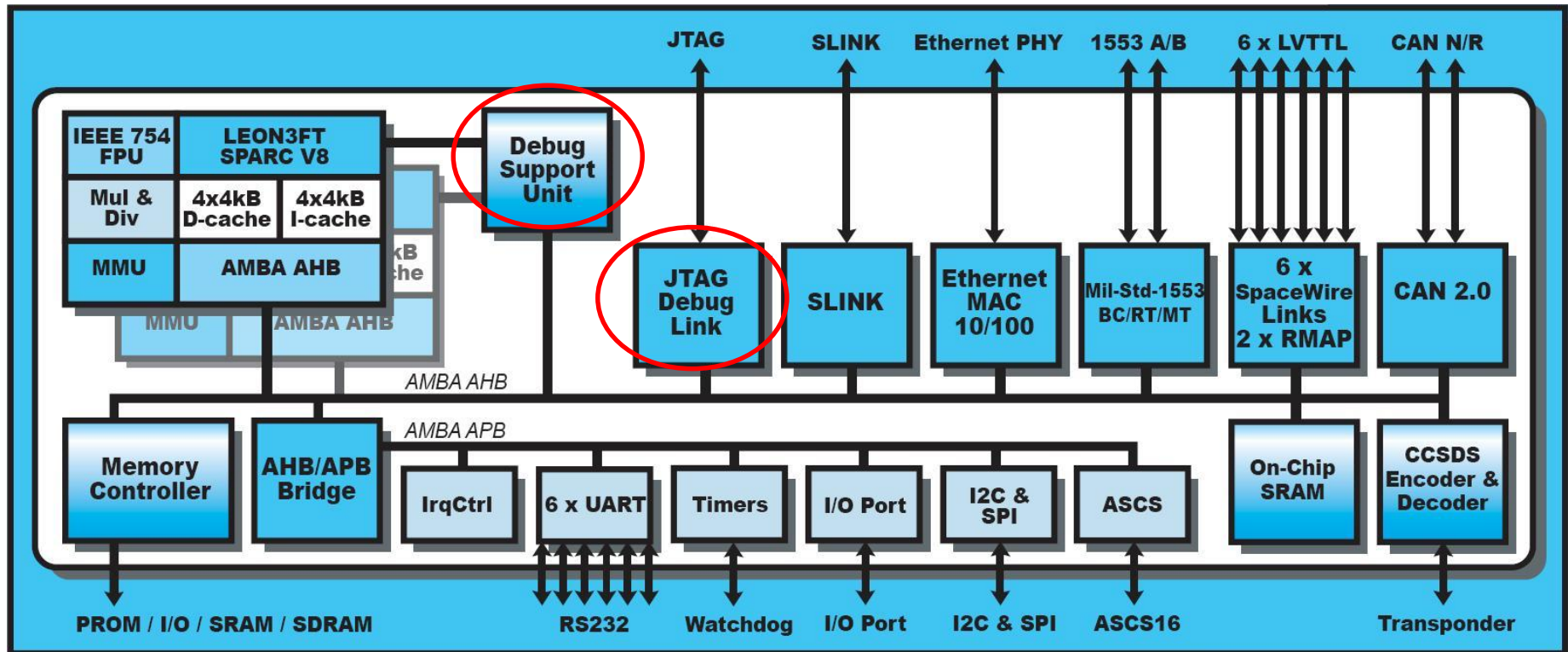
Fonctions de débogage intégrées aux processeurs

■ Exemple : ARM CORTEX-M4



Fonctions de débogage intégrées aux processeurs

■ Exemple : LEON3 GR712RC



Les moniteurs

- Un moniteur est outil logiciel s'exécutant sur la machine de développement et permettant de communiquer avec une cible embarquée (carte matérielle) à travers une interface de débogage dédiée (interface JTAG, interface SWD, interface série, ...).
 - JTAG = Joint Test Action Group = norme IEEE 1149.1
 - SWD = Serial Wire Debug interface
 - DAP = Debug Access Port
- Les moniteurs permettent d'accès aux fonctions de débogage intégrées aux processeurs.
- Les moniteurs peuvent être utilisés de façon standalone ou via un débogueur de type GDB.

Les moniteurs

- Il est souvent nécessaire d'utiliser un boîtier d'adaptation de l'interface.



Les moniteurs

- Un moniteur permet de :
 - Charger l'exécutable sur la cible embarqué
 - Démarrer l'exécution
 - Manipuler la mémoire (load / dump)
 - Inspecter / modifier les registres du processeur et les registres des périphériques
 - Accéder à l'historique des instructions exécutées
 - Interagir avec le l'unité DSU (Debug Support Unit) intégrée dans le processeur
 - Positionner des points d'arrêt et des watchpoints
 - Accéder au buffer d'instructions
 - Etc.

Les moniteurs

- Certaines technologies permettent d'accéder en temps réel aux données du processeur sans avoir à interrompre par un point d'arrêt la time-line d'exécution
 - = « Streaming trace »
 - La technologie CoreSight déployée sur les processeurs ARM Cortex-M permet une telle approche du débogage
 - https://www.arm.com/files/pdf/AT_-_Advanced_Debug_of_Cortex-M_Systems.pdf

Les moniteurs

Exemple

■ GRMON

- Moniteur développé par la société Gaisler pour les processeurs de la famille LEON
- <https://www.gaisler.com/index.php/products/debug-tools/grmon3>

■ OpenOCD

- Open On-Chip Debugger
- Moniteur open-source
- Cibles matérielles = processeurs ARM
- <http://openocd.org/>

■ Atmel-ICE Debugger

- Cibles matérielles = processeurs ARM Atmel

Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. **Création d'un exécutable bootable à partir d'une mémoire non-volatile**
 - f. Environnements de développement intégrés

Création d'un exécutable bootable à partir d'une mémoire non volatile

- Les cartes processeurs sont équipées :
 - d'une mémoire de travail de type SRAM ou SDRAM dans laquelle s'exécute le logiciel.
 - d'une mémoire non volatile de type PROM, EEPROM, FLASH, MRAM qui sert à stocker le logiciel quand la carte n'est pas sous-tension
- A la mise sous-tension de la carte, le processeur démarre à l'adresse 0x00000000 qui est l'adresse généralement de la mémoire non-volatile.

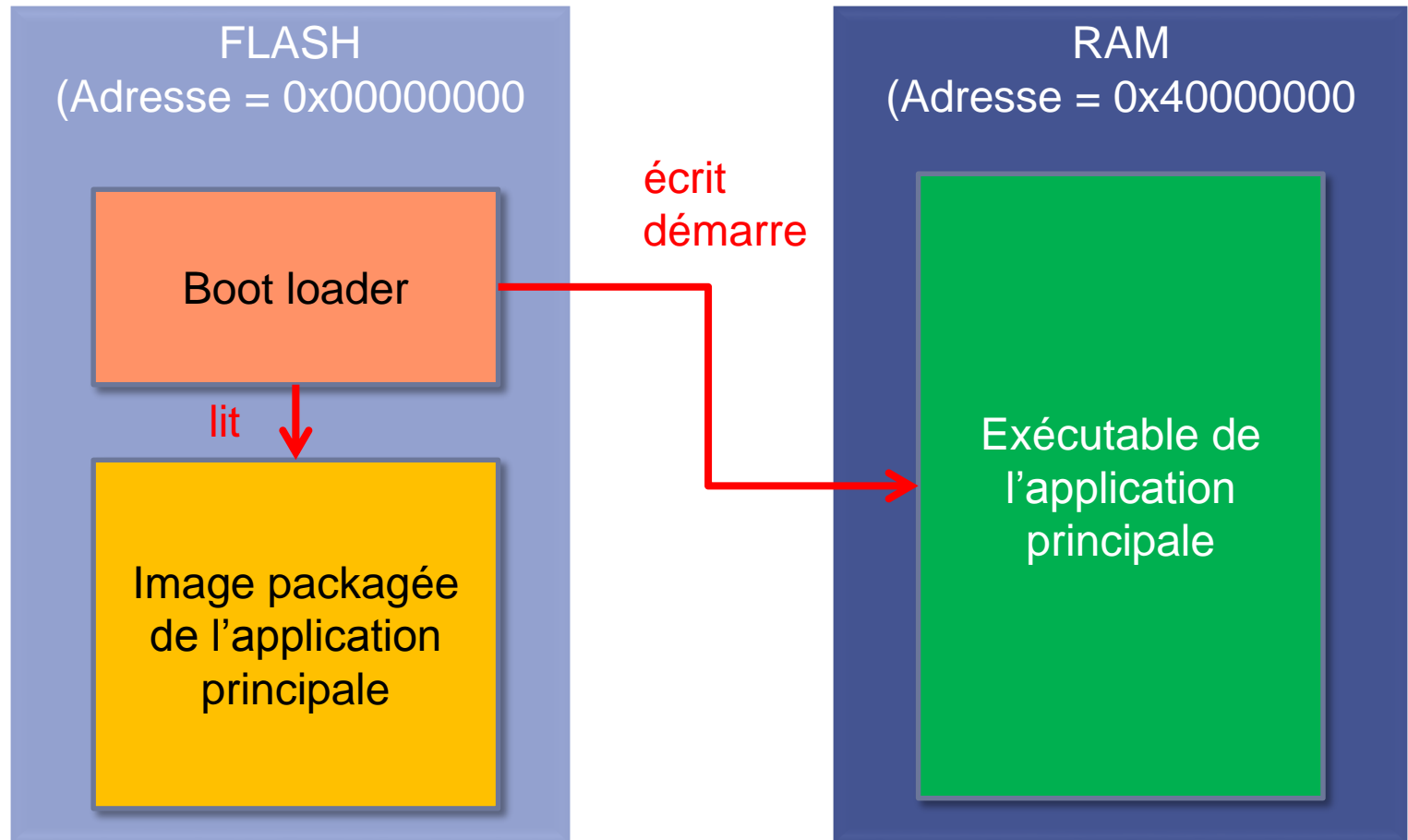
Création d'un exécutable bootable à partir d'une mémoire non volatile

- Les chaînes de compilation croisées permettent de produire une image de l'application dont le code peut s'exécuter dans une mémoire non-volatile de type FLASH et dont les données (y compris la pile) peuvent être stockées en RAM.
- Quand la mémoire non-volatile est de type read-only (PROM, EEPROM), les compilateurs ne permettent pas de créer facilement une image s'exécutant à partir de la mémoire non-volatile (problèmes de la section .data contenant les données initialisées).
- Par ailleurs, pour des raisons de performance (les mémoires non-volatiles sont généralement plus lentes) ou de maintenance du logiciel, il est souvent préférable que le code de l'application s'exécute directement dans la RAM et pas dans la mémoire non-volatile.

Création d'un exécutable bootable à partir d'une mémoire non volatile

- Le principe consiste à créer donc :
 - Une image de l'application principale qui va s'exécuter en RAM, en utilisant donc l'espace d'adressage de la RAM (les sections .text, .bss et .data sont localisées en RAM).
 - Une seconde application (bootloader, chargeur, ...) qui va :
 - S'exécuter directement dans la mémoire non-volatile au moment de la mise sous tension de la carte
 - Contenir une image packagée de l'application principale (compressée ou non)
 - Déployer dans la RAM l'application principale
 - Démarrer l'application principale
- On peut adjoindre au bootloader des fonctionnalités de maintenance (patch, remplacement) de l'application principale.

Création d'un exécutable bootable à partir d'une mémoire non volatile



A la mise sous tension de la carte, le bootloader s'exécute et déploie dans la RAM l'exécutable

Création d'un exécutable bootable à partir d'une mémoire non volatile

- Les bootloaders sont générés à partir d'utilitaires dédiés.
- Exemple : pour l'environnement LEON, l'utilitaire mkprom joue ce rôle.
 - <https://www.gaisler.com/doc/mkprom.pdf>

Création d'un exécutable bootable à partir d'une mémoire non volatile

```
mkprom2 -v -rmw -ramsize 4096 -sdram 32  
C:\workspace\factorial\Debug_O2\factorial -o  
C:\workspace\factorial\mkprom\O2\factorial.prom
```



```
MKPROM v2.0.63 - boot image generator for LEON applications  
Copyright Cobham Gaisler AB 2004-2017, all rights reserved.
```

```
phead0: type: 1, off: 65536, vaddr: 40000000, paddr: 40000000, fsize: 49568, msize: 51640  
phead1: type: 1, off: 131072, vaddr: 40100000, paddr: 40100000, fsize: 40, msize: 44  
section: .text at 0x40000000, size 46736 bytes
```

```
Uncoded stream length: 46736 bytes
```

```
Coded stream length: 27584 bytes
```

```
Compression Ratio: 1.694
```

```
section: .data at 0x4000b6c0, size 2784 bytes
```

```
Uncoded stream length: 2784 bytes
```

```
Coded stream length: 833 bytes
```

```
Compression Ratio: 3.342
```

```
section: .bss_factorial_array at 0x40100000, size 40 bytes
```

```
Uncoded stream length: 40 bytes
```

```
Coded stream length: 8 bytes
```

```
Compression Ratio: 5.000
```

```
section: .text_factorial at 0x4000b690, size 48 bytes
```

```
Uncoded stream length: 48 bytes
```

```
Coded stream length: 51 bytes
```

```
Compression Ratio: 0.941
```

```
Creating LEON3 boot prom: C:\workspace\factorial\mkprom\O2\factorial.prom
```

```
sparc-elf-gcc.exe -O2 -g -N -Tc:/opt/mkprom2/linkprom -Ttext=0x0 c:/opt/mkprom2/lib/ut699/promcore.o
```

```
c:/opt/mkprom2/lib/ut699/prominit.o c:/opt/mkprom2/lib/ut699/prominit_leon3.o
```

```
c:/opt/mkprom2/lib/ut699/promcrt0.o c:/opt/mkprom2/lib/ut699/promload.o c:/opt/mkprom2/lib/ut699/promdecomp.o
```

```
-nostdlib c:/opt/mkprom2/lib/ut699/prombdinit.o dump.s -o C:\workspace\factorial\mkprom\O2\factorial.prom
```

```
multidir:/lib/ut699
```

```
Success!
```

Création d'un exécutable bootable à partir d'une mémoire non volatile

■ Chargement de l'image PROM dans le simulateur :

```
C:\Program Files (x86)\GRTools>tsim-leon3
```

```
This TSIM evaluation version will expire 2018-12-18
```

```
TSIM/LEON3 SPARC simulator, version 2.0.61 (evaluation version)
```

```
Copyright (C) 2018, Cobham Gaisler - all rights reserved.
```

```
This software may only be used with a valid license.
```

```
For latest updates, go to http://www.gaisler.com/
```

```
Comments or bug-reports to support@gaisler.com
```

```
system frequency: 50.000 MHz
```

```
serial port A on stdin/stdout
```

```
allocated 4096 KiB SRAM memory, in 1 bank
```

```
allocated 32 MiB SDRAM memory, in 1 bank
```

```
allocated 2048 KiB ROM memory
```

```
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
```

```
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
```

```
tsim> load C:\workspace\factorial\mkprom\02\factorial.prom
```

```
section: .text, addr: 0x0, size 35520 bytes
```

```
read 216 symbols
```

Création d'un exécutable bootable à partir d'une mémoire non volatile

■ Exécution de la PROM de boot :

```
tsim> run
starting at 0x00000000

MKPROM2 boot loader v2.0.63
Copyright Cobham Gaisler AB - all rights reserved

system clock      : 50.0 MHz
baud rate         : 19171 baud
prom              : 512 K, (2/2) ws (r/w)
sram              : 4096 K, 1 bank(s), 0/0 ws (r/w)

decompressing .text to 0x40000000
decompressing .data to 0x4000b6c0
decompressing .bss_factori... to 0x40100000
decompressing .text_factor... to 0x4000b690

starting C:\workspace\actorialDebug_O2\actorial

fact(0) = 1
fact(1) = 1
fact(2) = 2
fact(3) = 6
fact(4) = 24
fact(5) = 120
fact(6) = 720
fact(7) = 5040
fact(8) = 40320
fact(9) = 362880
5.67 * 5.67 = 32.148899
sizeof(packet_header) = 6
sizeof(alt_packet_header) = 4
Program exited normally.
tsim>
```

Sommaire

1. Langages pour le développement des logiciels embarqués
2. Outils pour le développement des logiciels embarqués
 - a. Chaînes de compilation croisées (compilateur croisé, format ELF, linkers, architecture mémoire, options d'optimisation, prise en charge des FPU, linker scripts, utilisation des sections dans le code C, linker maps, makefile, binutils, objdump, objcopy, format SREC)
 - b. Simulateurs (QEMU, TSIM, ...)
 - c. Les débogueurs (GDB, utilisation interactive, utilisation scriptée)
 - d. Moniteurs et accès aux fonctions de débogage intégrées aux processeurs (OpenOCD, GRMON, ...)
 - e. Création d'un exécutable bootable à partir d'une mémoire non-volatile
 - f. Environnements de développement intégrés

Environnements de développement intégrés

- Un environnement de développement intégré est un outil logiciel qui centralise et fédère tous les outils dont on a besoin pour construire et tester une application embarquée (= point d'entrée unique pour le développement)
 - Editeur de code
 - Gestionnaire de code sous forme de projet
 - Connexion aux serveurs de configuration (subversion, git, ...)
 - Appel intégré aux outils de la chaîne de compilation
 - Création automatique des makefiles
 - Couche graphique au-dessus de GDB avec visualisation des registres internes du processeur et des registres des périphériques
 - Appel intégré au simulateur et moniteur
 - Etc.

Environnements de développement intégrés

- La plate-forme Eclipse est souvent utilisé pour élaborer ce genre d'environnement de développement intégré.
 - <http://www.eclipse.org/cdt/>

