ARM® Cortex®-A8 / Gestion des interruptions

Cours de systèmes embarqués 2 | Jacques Supcik | Mars 2017

1 Traitement des interruptions logicielles

Ce document traite des interruptions logiciels du microprocesseur ARM :

- Undefined exception
- Software interrupt
- Prefetch abort
- Data abort

Nous préparons aussi le code pour gérer les interruptions matérielles (IRQ et FIQ). Nous traiterons ces interruptions dans un prochain travail.

2 Les exceptions

Les interruptions logicielles sont principalement des « exceptions » et nous décidons d'implémenter le traitement des ces exceptions en « C » dans le fichier « exceptions.c »

```c
#include <stdio.h>

void undef_isr() {
    printf("ARM Exception: Undefined instruction\n");
}

void svc_isr() {
    printf("ARM Exception: Software interrupt\n");
}

void prefetch_isr() {
    printf("ARM Exception: Prefetch abort\n");
    while (1)
        ;
}
```

```
void data_isr() {
    printf("ARM Exception: Data abort\n");
}

void irq_isr() {
    printf("ARM Exception: IRQ\n");
}

void fiq_isr() {
    printf("ARM Exception: FIQ\n");
}
```

Notez que l'exception « prefetch abort » implémente une boucle sans fin et la routine ne termine jamais. En effet, une exception de type « prefetch abort » indique que le processeur n'a pas réussi à lire une instruction et il n'est donc pas possible de continuer le programme principal.

3 Table des vecteurs

Pour que le processeur appelle ces routines lorsqu'une interruption logicielle est levée, nous devons appeler ces routines depuis la table des vecteurs. Cette partie doit de faire en assembleur, car avant d'appeler la routine en « C », nous devons sauver tous les registres sur la pile et le restaurer au retour de la routine.

```
.align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)
undef_handler:
stmfd   sp!, {r0-r12,lr}        // save context and return address
    bl      undef_isr           // Call C code
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)

svc_handler:
    stmfd   sp!, {r0-r12,lr}
    bl      svc_isr
    ldmfd   sp!,{r0-r12,pc}^

prefetch_handler:
    stmfd   sp!, {r0-r12,lr}
    bl      prefetch_isr
```

```
    ldmfd   sp!,{r0-r12,pc}^

data_handler:
    sub     lr, #4              // update lr
    stmfd   sp!, {r0-r12,lr}
    bl      data_isr
    ldmfd   sp!,{r0-r12,pc}^

irq_handler:
    sub     lr, #4
    stmfd   sp!, {r0-r12,lr}
    bl      irq_isr
    ldmfd   sp!,{r0-r12,pc}^

fiq_handler:
    sub     lr, #4
    stmfd   sp!, {r0-r12,lr}
    bl      fiq_isr
    ldmfd   sp!,{r0-r12,pc}^
```

4 Utilisation de « macros »

Dans le code assembleur de la section précédente, on remarque que tous les « handlers » se ressemblent. Pour simplifier leur écriture, nous pouvons utiliser une macro :

```
.macro interrupt_handler offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset            // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    bl      \isr                // call C handler
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler 0, undef_isr
svc_handler:        interrupt_handler 0, svc_isr
prefetch_handler:   interrupt_handler 0, prefetch_isr
data_handler:       interrupt_handler 4, data_isr
irq_handler:        interrupt_handler 4, irq_isr
fiq_handler:        interrupt_handler 4, fiq_isr
```

5 Initialisation

Pour que notre système fonctionne, il faut encore l'initialiser. Concrètement, ça signifie initialiser les pointeurs de pile dans les différents modes et indiquer au système où se trouve sa table de

vecteurs.

Pour les piles, nous décidons d'utiliser de l'espace dans 64 KiByte de « SRAM » que nous avons sur le « Beaglebone Black ». Nous allouons 8 KiByte par pile.

```
AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)
```

Pour l'initialisation des piles, nous faisons également appel à une macro :

```
.macro set_stack mode, address
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init
interrupt_init:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

6 Activation et désactivation des interruptions

Nous pouvons très facilement activer et désactiver les interruptions sur notre système en manipulant le registre « CPSR » :

```
.global interrupt_enable
interrupt_enable:
    mrs     r0, cpsr
    bic     r1, #0xc0
    msr     cpsr_c, r1
    bx      lr
```

```
.global interrupt_disable
interrupt_disable:
    mrs     r0, cpsr
    orr     r1, r0, #0xc0
    msr     cpsr_c, r1
    bx      lr
```

7 Implémentation d'une table de routines d'interruptions

Le code présenté jusqu'ici est simple, mais il n'est pas très flexible. En effet, les routines de traitement sont définies à l'avance et ce n'est ni possible de modifier ces routines pendant l'exécution du programme, ni même de laisser l'utilisateur choisir la routine qu'il souhaite utiliser.

Pour corriger cette faiblesse, nous décidons d'implémenter un tableau de pointeurs de fonctions vers les routines de traitement des interruptions. L'appel à ces routines se fera de manière indirecte et nous mettrons à disposition des méthodes pour gérer ce tableau de pointeurs de fonctions.

Tous les « handlers » en assembleurs appelleront la même routine en « C » qui consultera la table des routines d'interruption pour appeler la routine correspondante. Pour faire son choix, la procédure en « C » recevra le numéro de l'interruption en paramètre. Ces numéros sont déclarés dans le fichier « interrupt.h » :

```
#define INT_NB_OF_VECTORS 6
enum interrupt_vectors {
    INT_UNDEF,      // undefined instruction
    INT_SVC,        // supervisor call (software interrupt)
    INT_PREFETCH,   // prefetch abort (instruction prefetch)
    INT_DATA,       // data abort (data access)
    INT_IRQ,        // hardware interrupt request
    INT_FIQ         // hardware fast interrupt request
};
```

Nous déclarons aussi un type « pointeur de fonction » qui sera l'élément principal de la table des routines :

```
typedef int (*interrupt_handler_t)();
```

La table elle-même est implémentée dans le fichier « interrupt.c » :

```
struct isr_table_entry {
    interrupt_handler_t routine;
};
```

```
static struct isr_table_entry isr_table[INT_NB_OF_VECTORS];
```

Nous devons aussi initialiser cette table. Nous décidons de faire cette initialisation en deux phases. La première phase, en assembleur, initialise les piles et indique au processeur où se trouve sa table des vecteurs :

```
interrupt_init_asm:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

La deuxième phase, en « C », initialise la table des routines à zéro. Pour éviter d'avoir à appeler deux fonctions, nous imbriquons l'appel à la routine en assembleur dans la routine en « C » :

```
extern void interrupt_init_asm();

void interrupt_init() {
    interrupt_init_asm();
    memset(isr_table, 0, sizeof(isr_table));
}
```

Les procédures suivantes permettent de gérer cette table de routines :

```
int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
        return -1;
    handler->routine = routine;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    isr_table[vector].routine = 0;
}
```

Et voici la procédure qui sera appelée par le « handler » en assembleur et qui appellera effectivement la routine d'interruption :

```
int interrupt_call(enum interrupt_vectors vector) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine == 0)
        return -1;
    interrupt_handler_t routine = handler->routine;
    return routine();
}
```

Le code en assembleur doit lui aussi connaître le numéro des vecteurs d'interruption :

```
INT_UNDEF       = 0     // undefined instruction
INT_SVC         = 1     // software interrupt
INT_PREFETCH    = 2     // prefetch abort (instruction prefetch)
INT_DATA        = 3     // data abort (data access)
INT_IRQ         = 4     // hardware interrupt request
INT_FIQ         = 5     // hardware fast interrupt request
```

Et voici comment nous implémentons la table des vecteurs et les « handlers » :

```
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)
.macro interrupt_handler vector offset
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // set vector number into first argument
    bl      interrupt_call      // call the second level handler in C
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0
svc_handler:        interrupt_handler INT_SVC,      0
prefetch_handler:   interrupt_handler INT_PREFETCH, 0
data_handler:       interrupt_handler INT_DATA,     4
irq_handler:        interrupt_handler INT_IRQ,      4
```

```
fiq_handler:        interrupt_handler INT_FIQ,      4
```

Le numéro du vecteur est passé à la routine en « C » par le registre « R0 » qui correspond bien au premier argument.

Nous pouvons maintenant adapter le fichier « exception.c » de manière à utiliser la table :

```c
#include <stdio.h>
#include "interrupt.h"

static int undef_isr() {
    printf("ARM Exception: Undefined instruction\n");
    return 0;
}

static int svc_isr() {
    printf("ARM Exception: Software interrupt\n");
    return 0;
}

static int prefetch_isr() {
    printf("ARM Exception: Prefetch abort\n");
    while (1)
        ;
    return 0;
}

static int data_isr() {
    printf("ARM Exception: Data abort\n");
    return 0;
}

void exception_init() {
    interrupt_attach(INT_UNDEF, undef_isr);
    interrupt_attach(INT_SVC, svc_isr);
    interrupt_attach(INT_PREFETCH, prefetch_isr);
    interrupt_attach(INT_DATA, data_isr);
}
```

8 Routines d'interruptions avec paramètres

Nous pouvons gagner en flexibilité si les routines d'interruptions peuvent utiliser des paramètres. Nous décidons d'étendre notre système en donnant à la routine les paramètres suivants :

- Le numéro du vecteur d'interruption.

- Un paramètre quelconque que nous attacherons à la table en même temps que la routine. Ce paramètre sera un pointeur sur « void ».
- L'adresse de l'instruction qui a provoqué l'interruption.

Pour le « handler » en assembleur, ça ne change pas beaucoup. Il doit juste encore passer l'adresse de l'instruction qui a provoqué l'interruption à la routine en « C ». On obtient cette adresse en soustrayant 4 au « link register ». Nous passons cet argument dans le registre. « R1 » :

```
.macro interrupt_handler vector offset
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    sub     r1, lr, #4          // load the address of interrupted instruction into r1
    bl      interrupt_call      // call the second level handler in C
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm
```

La table des routines est étendue avec un paramètre :

```
struct isr_table_entry {
    interrupt_handler_t routine;
    void* param;
};

static struct isr_table_entry isr_table[INT_NB_OF_VECTORS];
```

Nous définissons aussi un nouveau type pour le pointeur de fonction vers la routine d'interruption avec les nouveaux paramètres :

```
typedef int (*interrupt_handler_t)(
        enum interrupt_vectors vector,
        void* param,
        void* address);
```

Nous devons aussi adapter les procédures pour gérer la table des routines en donnant la possibilité de spécifier un paramètre :

```
int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
```

```
        return -1;
    handler->routine = routine;
    handler->param = param;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    isr_table[vector].routine = 0;
    isr_table[vector].param = 0;
}
```

Et l'appel sera lui aussi modifié en conséquence :

```
int interrupt_call(enum interrupt_vectors vector, void* address) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine == 0)
        return -1;
    interrupt_handler_t routine = handler->routine;
    void* param = handler->param;
    return routine(vector, param, address);
}
```

Le fichier « exception.c » peut maintenant être modifié pour profiter de ces paramètres :

```
#include <stdio.h>
#include <stdint.h>
#include "interrupt.h"

static int exception_handler(
        enum interrupt_vectors vector,
        void* param,
        void* address) {
    printf("0x%08lx : ARM Exception(%d): %s\n",
            (uint32_t) address,
            vector,
            (char*) param);
    // do not return in case of a pre-fetch error
    if (vector == INT_PREFETCH) {
        while (1) {
            ; // infinite loop
        }
    }
    return 0;
}

void exception_init() {
    interrupt_attach(INT_UNDEF, exception_handler, "undefined instruction");
    interrupt_attach(INT_SVC, exception_handler, "software interrupt");
    interrupt_attach(INT_PREFETCH, exception_handler, "prefetch abort");
```

```
    interrupt_attach(INT_DATA, exception_handler, ”data abort”);
}
```

9 Optimisation

Dans la version ci-dessus, le « handler » en assembleur appelle une procédure en « C » qui appelle à son tour la routine voulue. Si le code en assembleur avait accès à la table des interruptions, alors il pourrait appeler directement la bonne routine sans passer par le code de « C ». C'est ce que nous allons faire ici.

Nous commençons par déplacer la table du « C » vers l'assembleur. Une telle table n'est rien d'autre qu'une liste de paires de « double » ; un pour la routine et un pour le paramètre :

```
isr_table_start:
undef_isr:      .long   0, 0
svc_isr:        .long   0, 0
prefetch_isr:   .long   0, 0
data_isr:       .long   0, 0
irq_isr:        .long   0, 0
fiq_isr:        .long   0, 0
```

Dans le code en « C », la table est remplacée par un pointeur :

```
struct isr_table_entry {
    interrupt_handler_t routine;
    void* param;
};

static struct isr_table_entry* isr_table;
```

La routine d'initialisation écrite en assembleur retourne maintenant l'adresse de la table dans le registre « R0 » :

```
.global interrupt_init_asm
interrupt_init_asm:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
```

```
    mcr     p15, #0, r1, c12, c0, #0

    ldr     r0, =isr_table_start
    bx      lr
```

et la routine d'initialisation en « C » récupère cette adresse et initialise le pointeur :

```
static struct isr_table_entry* isr_table;
extern struct isr_table_entry* interrupt_init_asm();

void interrupt_init() {
    isr_table = interrupt_init_asm();
}
```

Il ne nous reste plus qu'à modifier le code des « handlers » pour utiliser la table des routines et directement appeler la bonne routine :

```
.macro interrupt_handler vector, offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    ldr     r3, \isr            // load the pointer to the routine into r3
    ldr     r1, \isr + 4        // load the pointer to the argument into r1
    sub     r2, lr, #4          // load the address of interrupted instruction into r2
    cmp     r3, #0
    blxne   r3                  // call ISR if the routine is not null
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0, undef_isr
svc_handler:        interrupt_handler INT_SVC,      0, svc_isr
prefetch_handler:   interrupt_handler INT_PREFETCH, 0, prefetch_isr
data_handler:       interrupt_handler INT_DATA,     4, data_isr
irq_handler:        interrupt_handler INT_IRQ,      4, irq_isr
fiq_handler:        interrupt_handler INT_FIQ,      4, fiq_isr

isr_table_start:
undef_isr:      .long   0, 0
svc_isr:        .long   0, 0
prefetch_isr:   .long   0, 0
data_isr:       .long   0, 0
irq_isr:        .long   0, 0
fiq_isr:        .long   0, 0
```

10 Tout en SRAM

Nous utilisons déjà la SRAM pour les piles, mais nous pouvons encore améliorer les performances de notre système en mettant la table des vecteurs, les « handlers » ainsi que la table des routines dans la SRAM. Pour ceci, il nous suffit de copier la DRAM vers la SRAM avec le routine « memcpy ». La plage que nous allons copier se trouve entre les labels « vector_table_start » et « vector_table_end » :

```
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)

.macro interrupt_handler vector, offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    ldr     r3, \isr            // load the pointer to the routine into r3
    ldr     r1, \isr + 4        // load the pointer to the argument into r1
    sub     r2, lr, #4          // load the address of interrupted instruction into r2
    cmp     r3, #0
    blxne   r3                  // call ISR if the routine is not null
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0, undef_isr
svc_handler:        interrupt_handler INT_SVC,      0, svc_isr
prefetch_handler:   interrupt_handler INT_PREFETCH, 0, prefetch_isr
data_handler:       interrupt_handler INT_DATA,     4, data_isr
irq_handler:        interrupt_handler INT_IRQ,      4, irq_isr
fiq_handler:        interrupt_handler INT_FIQ,      4, fiq_isr

isr_table_start:
undef_isr:      .long   0, 0
svc_isr:        .long   0, 0
prefetch_isr:   .long   0, 0
data_isr:       .long   0, 0
irq_isr:        .long   0, 0
fiq_isr:        .long   0, 0
vector_table_end:
```

La copie de fait dans la routine d'initialisation :

```
interrupt_init_asm:
    push    {lr}
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =VECTOR_BASE_ADDR
    mcr     p15, #0, r1, c12, c0, #0

    // copy low level interrupt handling code into vector table
    ldr     r0, =VECTOR_BASE_ADDR
    ldr     r1, =vector_table_start
    ldr     r2, =(vector_table_end - vector_table_start)
    bl      memcpy
    // return the address of the isr table in SRAM
    ldr     r0, =(VECTOR_BASE_ADDR + (isr_table_start - vector_table_start))

    pop     {pc}
```

Nous observons que l'adresse de la table des vecteurs correspond au début de la SRAM et l'adresse de la table des routines retournée par la fonction est corrigée pour correspondre à l'adresse dans la SRAM.

Cette dernière modification achève notre implémentation optimisée des interruptions sur notre processeur ARM Cortex-A8.

11 Codes Sources

Cette dernière section donne les codes sources complets pour ce projet.

11.1 Version 1 (simple)

11.1.1 exception.c

```
// exception.c / version 1 / GAC 27.02.2016 / SUP 20.03.2017

#include <stdio.h>

void undef_isr() {
    printf("ARM Exception: Undefined instruction\n");
}

void svc_isr() {
    printf("ARM Exception: Software interrupt\n");
}

void prefetch_isr() {
    printf("ARM Exception: Prefetch abort\n");
    while (1)
        ;
}

void data_isr() {
    printf("ARM Exception: Data abort\n");
}

void irq_isr() {
    printf("ARM Exception: IRQ\n");
}

void fiq_isr() {
    printf("ARM Exception: FIQ\n");
}
```

11.1.2 interrupt.h

```
// interrupt.h / version 1 / GAC 4.3.2017 / SUP 20.3.2017

#pragma once
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include <stdint.h>

extern void interrupt_init();
extern uint32_t interrupt_enable();
extern uint32_t interrupt_disable();

#endif
```

11.1.3 interrupt_asm.S

```
// interrupt_asm.S / version 1 / GAC 3.4.2017 / SUP 20.3.2017

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
.align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)
undef_handler:
stmfd   sp!, {r0-r12,lr}        // save context and return address
    bl      undef_isr           // Call C code
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)

svc_handler:
    stmfd   sp!, {r0-r12,lr}
```

```
    bl      svc_isr
    ldmfd   sp!,{r0-r12,pc}^

prefetch_handler:
    stmfd   sp!, {r0-r12,lr}
    bl      prefetch_isr
    ldmfd   sp!,{r0-r12,pc}^

data_handler:
    sub     lr, #4              // update lr
    stmfd   sp!, {r0-r12,lr}
    bl      data_isr
    ldmfd   sp!,{r0-r12,pc}^

irq_handler:
    sub     lr, #4
    stmfd   sp!, {r0-r12,lr}
    bl      irq_isr
    ldmfd   sp!,{r0-r12,pc}^

fiq_handler:
    sub     lr, #4
    stmfd   sp!, {r0-r12,lr}
    bl      fiq_isr
    ldmfd   sp!,{r0-r12,pc}^

.global interrupt_init
interrupt_init:
    // initialize all stack pointers
    mrs     r1, cpsr                // save mode
    msr     cpsr_c, #0xd1           // switch to fiq mode
    ldr     sp, =FIQ_STACK_TOP
    msr     cpsr_c, #0xd2           // switch to irq mode
    ldr     sp, =IRQ_STACK_TOP
    msr     cpsr_c, #0xd7           // switch to abort mode
    ldr     sp, =ABORT_STACK_TOP
    msr     cpsr_c, #0xdb           // switch to undef mode
    ldr     sp, =UNDEF_STACK_TOP
    msr     cpsr_c, r1              // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

11.1.4 interrupt_enabling_asm.S

```
// interrupt_enabling_asm.S / version 1 / GAC 3.4.2017 / SUP 20.3.2017

.text
.global interrupt_enable
interrupt_enable:
    mrs     r0, cpsr
    bic     r1, #0xc0
    msr     cpsr_c, r1
    bx      lr

.global interrupt_disable
interrupt_disable:
    mrs     r0, cpsr
    orr     r1, r0, #0xc0
    msr     cpsr_c, r1
    bx      lr
```

11.2 Version 2 (avec les macros)

11.2.1 interrupt_asm.S

```
// interrupt_asm.S / version 2 / GAC 3.4.2017 / SUP 20.3.2017

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
.align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)
.macro interrupt_handler offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset            // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    bl      \isr                // call C handler
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler 0, undef_isr
svc_handler:        interrupt_handler 0, svc_isr
prefetch_handler:   interrupt_handler 0, prefetch_isr
data_handler:       interrupt_handler 4, data_isr
irq_handler:        interrupt_handler 4, irq_isr
fiq_handler:        interrupt_handler 4, fiq_isr

.macro set_stack mode, address
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init
interrupt_init:
    // initialize all stack pointers
```

```
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

11.3 Version 3 (table des routines)

11.3.1 exception.h

```
// exception.h / version 3 / GAC 27.2.2016 / SUP 13.3.2017

#pragma once
#ifndef EXCEPTION_H
#define EXCEPTION_H

extern void exception_init();

#endif
```

11.3.2 exception.c

```
// exception.c / version 3 / GAC 27.2.2016 / SUP 13.3.2017

#include <stdio.h>
#include "interrupt.h"

static int undef_isr() {
    printf("ARM Exception: Undefined instruction\n");
    return 0;
}

static int svc_isr() {
    printf("ARM Exception: Software interrupt\n");
    return 0;
}

static int prefetch_isr() {
    printf("ARM Exception: Prefetch abort\n");
    while (1)
        ;
    return 0;
}

static int data_isr() {
    printf("ARM Exception: Data abort\n");
    return 0;
}

void exception_init() {
    interrupt_attach(INT_UNDEF, undef_isr);
    interrupt_attach(INT_SVC, svc_isr);
    interrupt_attach(INT_PREFETCH, prefetch_isr);
```

```
    interrupt_attach(INT_DATA, data_isr);
}
```

11.3.3 interrupt.h

```
// interrupt.h / version 3 / GAC 4.3.2017 / SUP 13.3.2017

#pragma once
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include <stdint.h>

#define INT_NB_OF_VECTORS 6
enum interrupt_vectors {
    INT_UNDEF,      // undefined instruction
    INT_SVC,        // supervisor call (software interrupt)
    INT_PREFETCH,   // prefetch abort (instruction prefetch)
    INT_DATA,       // data abort (data access)
    INT_IRQ,        // hardware interrupt request
    INT_FIQ         // hardware fast interrupt request
};

typedef int (*interrupt_handler_t)();

extern void interrupt_init();

extern int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine);
extern void interrupt_detach(enum interrupt_vectors vector);

extern uint32_t interrupt_enable();
extern uint32_t interrupt_disable();

#endif
```

11.3.4 interrupt.c

```c
// interrupt_c / version 3 / GAC 3.4.2017 / SUP 13.3.2017

#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "interrupt.h"

struct isr_table_entry {
    interrupt_handler_t routine;
};

static struct isr_table_entry isr_table[INT_NB_OF_VECTORS];
extern void interrupt_init_asm();

void interrupt_init() {
    interrupt_init_asm();
    memset(isr_table, 0, sizeof(isr_table));
}

int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
        return -1;
    handler->routine = routine;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    isr_table[vector].routine = 0;
}

int interrupt_call(enum interrupt_vectors vector) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine == 0)
        return -1;
    interrupt_handler_t routine = handler->routine;
    return routine();
}
```

11.3.5 interrupt_asm.S

```
// interrupt_asm.S / version 3 / GAC 3.4.2017 / SUP 13.3.2017

INT_UNDEF       = 0     // undefined instruction
INT_SVC         = 1     // software interrupt
INT_PREFETCH    = 2     // prefetch abort (instruction prefetch)
INT_DATA        = 3     // data abort (data access)
INT_IRQ         = 4     // hardware interrupt request
INT_FIQ         = 5     // hardware fast interrupt request

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
        .align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)
.macro interrupt_handler vector offset
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // set vector number into first argument
    bl      interrupt_call      // call the second level handler in C
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0
svc_handler:        interrupt_handler INT_SVC,      0
prefetch_handler:   interrupt_handler INT_PREFETCH, 0
data_handler:       interrupt_handler INT_DATA,     4
irq_handler:        interrupt_handler INT_IRQ,      4
fiq_handler:        interrupt_handler INT_FIQ,      4

.align 8
.macro set_stack mode, address
```

```
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init_asm
interrupt_init_asm:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

11.4 Version 4 (paramètres)

11.4.1 exception.c

```
// exception.c / version 4 / GAC 27.2.2016 / SUP 13.3.2017

#include <stdio.h>
#include <stdint.h>
#include "interrupt.h"

static int exception_handler(
        enum interrupt_vectors vector,
        void* param,
        void* address) {
    printf("0x%08lx : ARM Exception(%d): %s\n",
            (uint32_t) address,
            vector,
            (char*) param);
    // do not return in case of a pre-fetch error
    if (vector == INT_PREFETCH) {
        while (1) {
            ; // infinite loop
        }
    }
    return 0;
}

void exception_init() {
    interrupt_attach(INT_UNDEF, exception_handler, "undefined instruction");
    interrupt_attach(INT_SVC, exception_handler, "software interrupt");
    interrupt_attach(INT_PREFETCH, exception_handler, "prefetch abort");
    interrupt_attach(INT_DATA, exception_handler, "data abort");
}
```

11.4.2 interrupt.h

```
// interrupt.h / version 4 / GAC 4.3.2017 / SUP 13.3.2017

#pragma once
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include <stdint.h>

#define INT_NB_OF_VECTORS 6
enum interrupt_vectors {
    INT_UNDEF,      // undefined instruction
    INT_SVC,        // supervisor call (software interrupt)
    INT_PREFETCH,   // prefetch abort (instruction prefetch)
    INT_DATA,       // data abort (data access)
    INT_IRQ,        // hardware interrupt request
    INT_FIQ         // hardware fast interrupt request
};

typedef int (*interrupt_handler_t)(
        enum interrupt_vectors vector,
        void* param,
        void* address);

extern void interrupt_init();

extern int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param);
extern void interrupt_detach(enum interrupt_vectors vector);

extern uint32_t interrupt_enable();
extern uint32_t interrupt_disable();

#endif
```

11.4.3 interrupt.c

```c
// interrupt_c version 4 / GAC 3.4.2017 / SUP 13.3.2017

#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "interrupt.h"

struct isr_table_entry {
    interrupt_handler_t routine;
    void* param;
};

static struct isr_table_entry isr_table[INT_NB_OF_VECTORS];
extern void interrupt_init_asm();

void interrupt_init() {
    interrupt_init_asm();
    memset(isr_table, 0, sizeof(isr_table));
}

int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
        return -1;
    handler->routine = routine;
    handler->param = param;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    isr_table[vector].routine = 0;
    isr_table[vector].param = 0;
}

int interrupt_call(enum interrupt_vectors vector, void* address) {
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine == 0)
        return -1;
    interrupt_handler_t routine = handler->routine;
    void* param = handler->param;
    return routine(vector, param, address);
}
```

11.4.4 interrupt_asm.S

```
// interrupt_asm.S / version 4 / GAC 3.4.2017 / SUP 13.3.2017

INT_UNDEF       = 0     // undefined instruction
INT_SVC         = 1     // software interrupt
INT_PREFETCH    = 2     // prefetch abort (instruction prefetch)
INT_DATA        = 3     // data abort (data access)
INT_IRQ         = 4     // hardware interrupt request
INT_FIQ         = 5     // hardware fast interrupt request

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
        .align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)

.macro interrupt_handler vector offset
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    sub     r1, lr, #4          // load the address of interrupted instruction into r1
    bl      interrupt_call      // call the second level handler in C
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0
svc_handler:        interrupt_handler INT_SVC,      0
prefetch_handler:   interrupt_handler INT_PREFETCH, 0
data_handler:       interrupt_handler INT_DATA,     4
irq_handler:        interrupt_handler INT_IRQ,      4
fiq_handler:        interrupt_handler INT_FIQ,      4
```

```
.align 8
.macro set_stack mode, address
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init_asm
interrupt_init_asm:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    bx      lr
```

11.5 Version 5 (appel directe)

11.5.1 interrupt.c

```
// interrupt_c / version 5 / GAC 3.4.2017 / SUP 13.3.2017

#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "interrupt.h"

struct isr_table_entry {
    interrupt_handler_t routine;
    void* param;
};

static struct isr_table_entry* isr_table;
extern struct isr_table_entry* interrupt_init_asm();

void interrupt_init() {
    isr_table = interrupt_init_asm();
}

int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param) {
    if (vector >= INT_NB_OF_VECTORS) return -1;
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
        return -1;
    handler->routine = routine;
    handler->param = param;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    if (vector >= INT_NB_OF_VECTORS) return;
    isr_table[vector].routine = 0;
    isr_table[vector].param = 0;
}
```

11.5.2 interrupt_asm.S

```
// interrupt_asm.S / version 5 / GAC 3.4.2017 / SUP 13.3.2017

INT_UNDEF       = 0     // undefined instruction
INT_SVC         = 1     // software interrupt
INT_PREFETCH    = 2     // prefetch abort (instruction prefetch)
INT_DATA        = 3     // data abort (data access)
INT_IRQ         = 4     // hardware interrupt request
INT_FIQ         = 5     // hardware fast interrupt request

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
.align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)

.macro interrupt_handler vector, offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    ldr     r3, \isr            // load the pointer to the routine into r3
    ldr     r1, \isr + 4        // load the pointer to the argument into r1
    sub     r2, lr, #4          // load the address of interrupted instruction into r2
    cmp     r3, #0
    blxne   r3                  // call ISR if the routine is not null
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0, undef_isr
svc_handler:        interrupt_handler INT_SVC,      0, svc_isr
prefetch_handler:   interrupt_handler INT_PREFETCH, 0, prefetch_isr
data_handler:       interrupt_handler INT_DATA,     4, data_isr
```

```
irq_handler:        interrupt_handler INT_IRQ,      4, irq_isr
fiq_handler:        interrupt_handler INT_FIQ,      4, fiq_isr

isr_table_start:
undef_isr:      .long   0, 0
svc_isr:        .long   0, 0
prefetch_isr:   .long   0, 0
data_isr:       .long   0, 0
irq_isr:        .long   0, 0
fiq_isr:        .long   0, 0

.align 8
.macro set_stack mode, address
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init_asm
interrupt_init_asm:
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =vector_table_start
    mcr     p15, #0, r1, c12, c0, #0

    ldr     r0, =isr_table_start
    bx      lr
```

11.6 Version 6 (copie en SRAM)

Voici encore la version complète du code final.

11.6.1 exception.h

```
// exception.h / version 6 / GAC 27.2.2016 / SUP 13.3.2017

#pragma once
#ifndef EXCEPTION_H
#define EXCEPTION_H

extern void exception_init();

#endif
```

11.6.2 exception.c

```
// exception.c / version 6 / GAC 27.2.2016 / SUP 13.3.2017

#include <stdio.h>
#include <stdint.h>
#include "interrupt.h"

static int exception_handler(
        enum interrupt_vectors vector,
        void* param,
        void* address) {
    printf("0x%08lx : ARM Exception(%d): %s\n",
            (uint32_t) address,
            vector,
            (char*) param);
    // do not return in case of a pre-fetch error
    if (vector == INT_PREFETCH) {
        while (1) {
            ; // infinite loop
        }
    }
    return 0;
}

void exception_init() {
    interrupt_attach(INT_UNDEF, exception_handler, "undefined instruction");
    interrupt_attach(INT_SVC, exception_handler, "software interrupt");
    interrupt_attach(INT_PREFETCH, exception_handler, "prefetch abort");
```

```
    interrupt_attach(INT_DATA, exception_handler, "data abort");
}
```

11.6.3 interrupt.h

```
// interrupt.h / version 6 / GAC 4.3.2017 / SUP 13.3.2017

#pragma once
#ifndef INTERRUPT_H
#define INTERRUPT_H

#include <stdint.h>

#define INT_NB_OF_VECTORS 6
enum interrupt_vectors {
    INT_UNDEF,      // undefined instruction
    INT_SVC,        // supervisor call (software interrupt)
    INT_PREFETCH,   // prefetch abort (instruction prefetch)
    INT_DATA,       // data abort (data access)
    INT_IRQ,        // hardware interrupt request
    INT_FIQ         // hardware fast interrupt request
};

typedef int (*interrupt_handler_t)(
        enum interrupt_vectors vector,
        void* param,
        void* address);

extern void interrupt_init();

extern int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param);
extern void interrupt_detach(enum interrupt_vectors vector);

extern uint32_t interrupt_enable();
extern uint32_t interrupt_disable();

#endif
```

11.6.4 interrupt.c

```c
// interrupt_c / version 6 / GAC 3.4.2017 / SUP 13.3.2017

#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "interrupt.h"

struct isr_table_entry {
    interrupt_handler_t routine;
    void* param;
};

static struct isr_table_entry* isr_table;
extern struct isr_table_entry* interrupt_init_asm();

void interrupt_init() {
    isr_table = interrupt_init_asm();
}

int interrupt_attach(
        enum interrupt_vectors vector,
        interrupt_handler_t routine,
        void* param) {
    if (vector >= INT_NB_OF_VECTORS) return -1;
    struct isr_table_entry* handler = &isr_table[vector];
    if (handler->routine != 0)
        return -1;
    handler->routine = routine;
    handler->param = param;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    if (vector >= INT_NB_OF_VECTORS) return;
    isr_table[vector].routine = 0;
    isr_table[vector].param = 0;
}
```

11.6.5 interrupt_asm.S

```
// interrupt_asm.S / version 6 / GAC 3.4.2017 / SUP 13.3.2017

INT_UNDEF       = 0     // undefined instruction
INT_SVC         = 1     // software interrupt
INT_PREFETCH    = 2     // prefetch abort (instruction prefetch)
INT_DATA        = 3     // data abort (data access)
INT_IRQ         = 4     // hardware interrupt request
INT_FIQ         = 5     // hardware fast interrupt request

AM335X_OCMC_SRAM_BASE_ADDR = 0x40300000
VECTOR_BASE_ADDR           = (AM335X_OCMC_SRAM_BASE_ADDR)
FIQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0A000)
IRQ_STACK_TOP              = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0C000)
ABORT_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x0E000)
UNDEF_STACK_TOP            = (AM335X_OCMC_SRAM_BASE_ADDR + 0x10000)

.text

// Vector table
.align 8
vector_table_start:
1:  b       1b  // reset
    b       undef_handler
    b       svc_handler
    b       prefetch_handler
    b       data_handler
1:  b       1b  // reserved
    b       irq_handler
    b       fiq_handler

// Handlers (called through the vector table)

.macro interrupt_handler vector, offset, isr
    .if \offset != 0            // adjust return address
    sub     lr, #\offset        // only if necessary
    .endif
    stmfd   sp!, {r0-r12,lr}    // save context and return address
    mov     r0, #\vector        // load vector number into r0
    ldr     r3, \isr            // load the pointer to the routine into r3
    ldr     r1, \isr + 4        // load the pointer to the argument into r1
    sub     r2, lr, #4          // load the address of interrupted instruction into r2
    cmp     r3, #0
    blxne   r3                  // call ISR if the routine is not null
    ldmfd   sp!,{r0-r12,pc}^    // restore the context (pc & cpsr)
.endm

undef_handler:      interrupt_handler INT_UNDEF,    0, undef_isr
svc_handler:        interrupt_handler INT_SVC,      0, svc_isr
prefetch_handler:   interrupt_handler INT_PREFETCH, 0, prefetch_isr
```

```
data_handler:       interrupt_handler INT_DATA,     4, data_isr
irq_handler:        interrupt_handler INT_IRQ,      4, irq_isr
fiq_handler:        interrupt_handler INT_FIQ,      4, fiq_isr

isr_table_start:
undef_isr:      .long   0, 0
svc_isr:        .long   0, 0
prefetch_isr:   .long   0, 0
data_isr:       .long   0, 0
irq_isr:        .long   0, 0
fiq_isr:        .long   0, 0
vector_table_end:

.align 8
.macro set_stack mode, address
    msr     cpsr_c, #\mode
    ldr     sp, =\address
.endm

.global interrupt_init_asm
interrupt_init_asm:
    push    {lr}
    // initialize all stack pointers
    mrs     r1, cpsr  // save mode
    set_stack 0xd1, FIQ_STACK_TOP
    set_stack 0xd2, IRQ_STACK_TOP
    set_stack 0xd7, ABORT_STACK_TOP
    set_stack 0xdb, UNDEF_STACK_TOP
    msr     cpsr_c, r1  // restore mode

    // cp15 set vector base address
    ldr     r1, =VECTOR_BASE_ADDR
    mcr     p15, #0, r1, c12, c0, #0

    // copy low level interrupt handling code into vector table
    ldr     r0, =VECTOR_BASE_ADDR
    ldr     r1, =vector_table_start
    ldr     r2, =(vector_table_end - vector_table_start)
    bl      memcpy
    // return the address of the isr table in SRAM
    ldr     r0, =(VECTOR_BASE_ADDR + (isr_table_start - vector_table_start))

    pop     {pc}
```

11.6.6 interrupt_enabling_asm.S

```
// interrupt_enabling_asm.S / version 6 / GAC 3.4.2017 / SUP 20.3.2017

.text
.global interrupt_enable
interrupt_enable:
    mrs     r0, cpsr
    bic     r1, #0xc0
    msr     cpsr_c, r1
    bx      lr

.global interrupt_disable
interrupt_disable:
    mrs     r0, cpsr
    orr     r1, r0, #0xc0
    msr     cpsr_c, r1
    bx      lr
```

11.6.7 main.c

```
// main.c / version 6 / GAC 4.3.2017 / SUP 20.3.2017

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

#include "interrupt.h"
#include "exception.h"

int main() {
    printf("\n");
    printf("HEIA-FR - Embedded Systems 2 Laboratory\n");
    printf("Low Level Interrupt Handling on ARM Cortex-A8\n");
    printf("-----------------------------------------\n");

    interrupt_init();
    exception_init();

    printf("Test data abort with a miss aligned access\n");
    long l = 0;
    long* pl = (long*) ((char*) &l + 1);
    *pl = 2;

    printf("Test supervisor call instruction / software interrupt\n");
    __asm__ ("svc #1;");

    printf("Test a invalid instruction\n");
    __asm__ (".word 0xffffffff;");

    printf("Test a prefetch abort\nThis method will never return...\n");
    __asm__ ("mov pc,#0x00000000;");

    while (1)
        ;
    return 0;
}
```