



SYSTEMES EMBARQUES 2

TP06 – TRAITEMENT DES INTERRUPTIONS ET DES EXCEPTIONS

Fabio Valverde & Samuel Mertenat
Classe T2f

Fribourg, le 2 mars 2015
08h15-11h50

TABLE DES MATIERES

Buts et Objectifs du travail pratique.....	3
Analyse	3
Assembleur ARM : instructions utiles	3
<i>Registres et registres du coprocesseur.....</i>	3
<i>Sauvegarder / restaurer les registres.....</i>	4
Les interruptions	4
<i>Concept général et types d'interruptions.....</i>	4
<i>Séquence d'interruption.....</i>	5
<i>Modes de fonctionnement et état du processeur</i>	6
<i>Entrée et sortie de la routine d'interruption</i>	6
Conception.....	7
Structure du programme	7
Explication des différents fichiers	7
<i>main.c</i>	7
<i>exception.c.....</i>	7
<i>interrupt_asm.s.....</i>	8
<i>interrupt.c.....</i>	8
Tests et validation	9
Problèmes rencontrés & Solutions	10
Acquis	10
Perspectives	10
Conclusion	11
Annexe	11
Sources	11

SYSTEMES EMBARQUES 2

TP06 – TRAITEMENT DES INTERRUPTIONS ET DES EXCEPTIONS

BUTS ET OBJECTIFS DU TRAVAIL PRATIQUE

Ce travail pratique a pour but de nous enseigner le déroulement d'une interruption matérielle ou logicielle et d'en proposer une procédure de traitement.

Objectifs :

- Décrire le traitement des interruptions et des exceptions du processeur ARM
- Concevoir et développer une petite application en C et en assembleur permettant le traitement des interruptions et des exceptions de bas niveau sur processeur i.MX27
- Debugger une application mixte C / assembleur et traitant des interruptions
- Etudier les datasheets du processeur i.MX27

Cahier des charges :

- Initialiser la bibliothèque (interrupt_init)
- Attacher une routine de service des interruptions (ISR) à un vecteur donnée (interrupt_attach). Cette méthode doit permettre de spécifier un paramètre avec l'ISR. Ce paramètre sera passé, avec le numéro du vecteur de l'interruption, comme argument lors de l'appel de la ISR. Une seule ISR pourra être attachée à un vecteur d'interruption. En cas d'erreur la valeur -1 sera retournée.
- Détacher l'ISR d'un vecteur d'interruption (interrupt_detach)
- Activer/désactiver les interruptions matérielles (interrupt_enable / interrupt_disable)

ANALYSE

ASSEMBLEUR ARM : INSTRUCTIONS UTILES

REGISTRES ET REGISTRES DU COPROCESSEUR

L'instruction « MRS¹ » permet d'attribuer la valeur contenue dans un registre du coprocesseur vers un registre « normal ». A l'inverse, l'instruction « MSR² » permet d'affecter un registre du coprocesseur avec une valeur stockée dans un registre « normal ».

¹ <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/CIHGJHHH.html>

² <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489f/CIHFIDAJ.html>

Exemple :

```
mrs    r2, cpsr    // saves the current mode into r2 from cpsr
msr    cpsr_c, r0   // switches to the mode given by the first param (r0)
mov     sp, r1      // initiates the stack pointer with the second given param (r1)
msr     cpsr_c, r2   // restores to the previous mode (affects only the 8 first bits)
```

SAUVEGARDER / RESTAURER LES REGISTRES

Par convention, les registres r0-3 servent à passer des paramètres à une fonction ; le registre r0 étant destiné au retour de la fonction. Par contre, les registres r4-12 doivent être laissés intacts, il est donc nécessaire de réaliser des « sauvegardes » avant toute modification de ceux-ci.

L'assembleur ARM nous propose une solution afin de réaliser ces « sauvegardes » :

```
stmfd   sp!, {r0-r12,lr}    // saves the registers & the link register
mov      r0, lr              // saves the return address (lr)
mov      r1, #INT_UNDEFINED  // saves the vector number
bl       interrupt_process    // calls interrupt_process(r0, r1) (C function)
ldmfd   sp!, {r0-r12,pc}^    // restores back the registers, pc & cpsr
```

- ^ : permet de restaurer le CPSR depuis le SPSR
- lr : est remplacé par « pc » afin de retourner directement au programme interrompu

LES INTERRUPTIONS

CONCEPT GENERAL ET TYPES D'INTERRUPTIONS

Une interruption peut être imaginée comme un appel d'un sous-programme déclenché par un signal. Celui-ci est lui-même la conséquence d'un événement, qui peut être lié au matériel ou au logiciel.

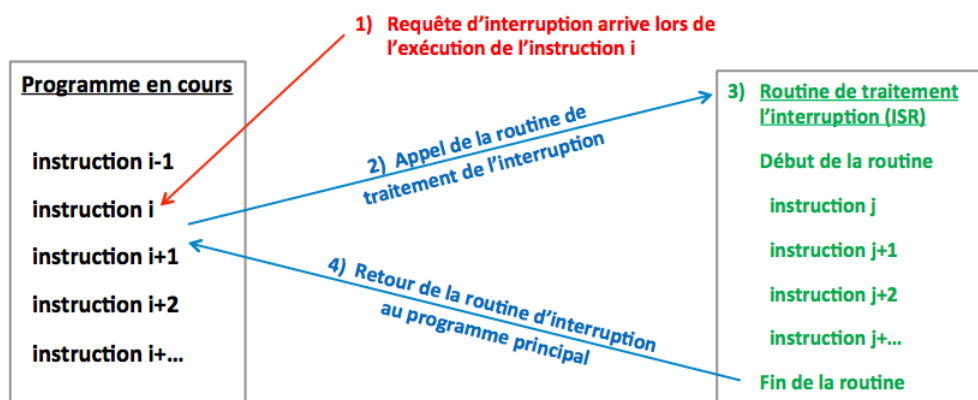


Figure 1: Fonctionnement d'une interruption

Les types d'interruption :

- Logicielles : 3 fonctions : permettre à une application de communiquer avec un système d'I/O ; permettre à une application de communiquer avec un système

opératif évolué ; permettre à des outils de développement de poser des points d'arrêt lors du débogage de l'application. Exemple « SWI n_24 ».

- Matérielles : permettent aux périphériques de prendre l'initiative de l'échange, en forçant le microprocesseur (p. ex. carte réseau, port série, ...).
- Exceptions : se propagent depuis l'appelé vers l'appelant jusqu'à ce qu'elles rencontrent un bloc de code qui s'occupe de les traiter.

SEQUENCE D'INTERRUPTION

On peut définir le cycle de vie d'une interruption ainsi:

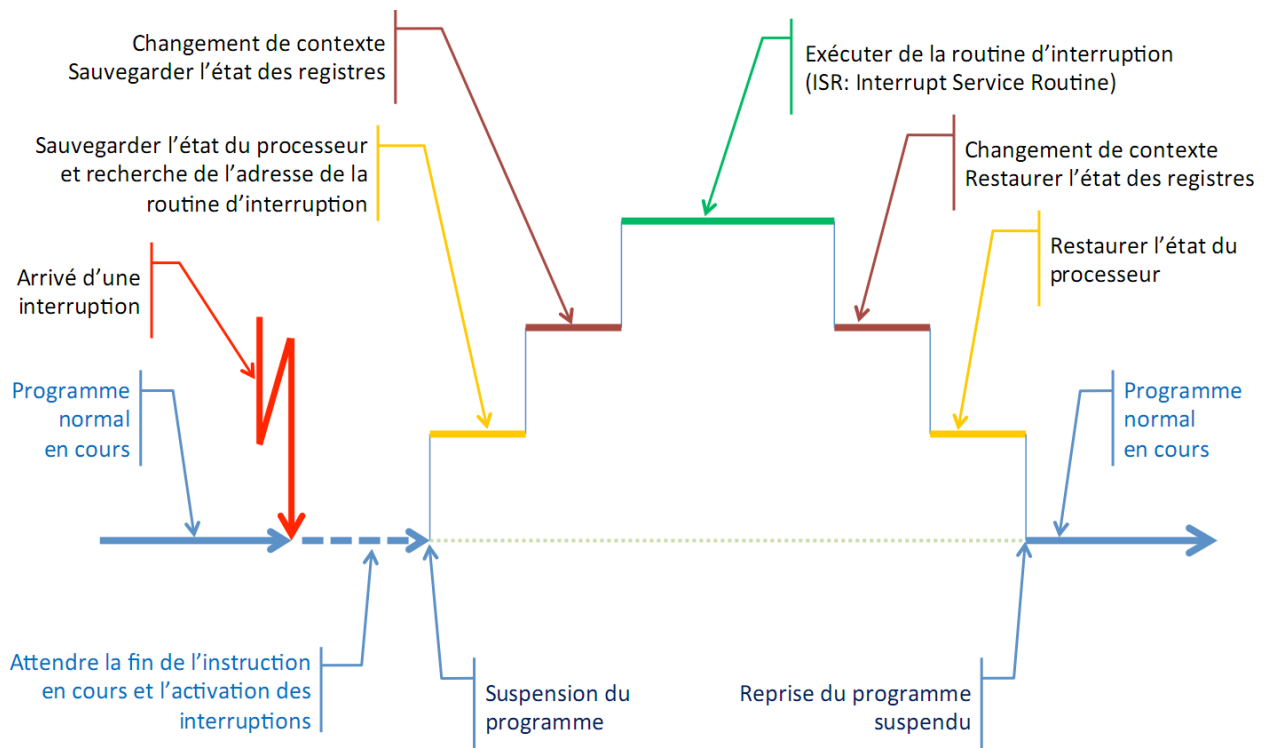


Figure 2 - Processus d'interruption

Les étapes en jaune et en marrons, aussi appelées "changement de contexte" ont une explication dans le chapitre "SAUVEGARDER / RESTAURER LES REGISTRES".

MODES DE FONCTIONNEMENT ET ETAT DU PROCESSEUR

Il existe plusieurs modes de fonctionnement pour le processeur:

Mode	Abrv	Code	Description	Accès
User	usr	0x10	Mode d'exécution normal de programme par les utilisateurs (OS)	Limité
FIQ	fiq	0x11	Mode actif lors du traitement d'interruptions demandant un traitement rapide	Illimité
IRQ	irq	0x12	Mode actif lors du traitement d'interruptions normales	Illimité
Supervisor	svc	0x13	Mode utilisé lors d'exécution de code propre au système d'exploitation (OS)	Illimité
Abort	abt	0x17	Implémente les mécanisme de mémoire virtuelle	Illimité
Undifined	und	0x1b	Software émulation	Illimité
System	sys	0x1f	Tâches mode privilégié du système d'exploitation	Illimité

Figure 3 - Modes d'exécution du processeur

Le mode User a un accès limité, mais il faut noter que le mode System utilise les mêmes registres.

Le registre de statut du processeur se compose ainsi:

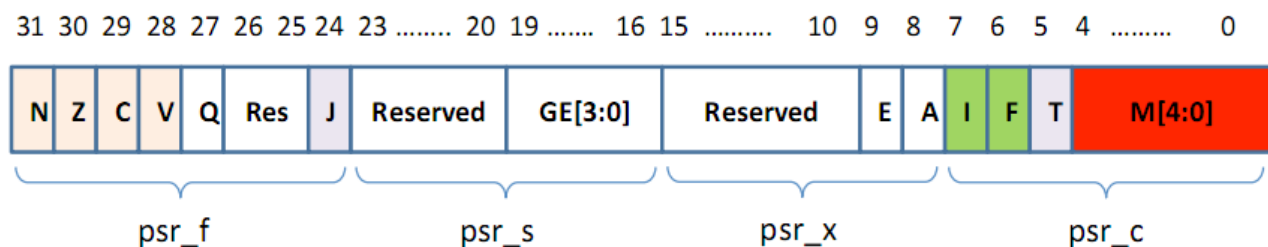


Figure 4 - PSR

M: Désigne le mode utilisé par le processeur

I F: Permet de désactiver les IRQ resp. les FIQ

ENTREE ET SORTIE DE LA ROUTINE D'INTERRUPTION

Entrée : Lorsqu'une interruption survient, le processeur, avant d'exécuter la routine d'interruption, sauve ses registres (cf. Sauvegarder / Restaurer les registres) puis appelle la routine d'interruption.

Sortie : Avant de retourner au programme interrompu par l'interruption, il est nécessaire de restaurer l'état du processeur et du microprocesseur. De plus, l'adresse de retour (lr), doit être corrigée d'un offset, dont on peut en voir les valeurs dans la table ci-dessous, en fonction du type d'interruption.

Event	Offset	Return from handler
Reset	n/a	n/a
Data Abort	-4	subs pc, lr, #4
FIQ	-4	subs pc, lr, #4
IRQ	-4	subs pc, lr, #4
Pre-fetch Abort	-4	subs pc, lr, #4
SWI	0	movs pc, lr
Undefined Instruction	0	movs pc, lr

Figure 5 - Retour d'interruption

CONCEPTION

STRUCTURE DU PROGRAMME

Le programme se décompose en 6 fichiers :

- main.c : Module principale comportant différentes instructions (C ou ARM) permettant de déclencher des interruptions.
- exception.h : Entêtes du fichier exception.c
- exception.c : Module lié au traitement des exceptions
- interrupt_asm.s : Module comportant les différentes sous-routines permettant de lever les interruptions
- interrupt.h : Entêtes du fichier interrupt.h
- interrupt.c : Module permettant d'initialiser, d'attacher ou de détacher des vecteurs d'interruption

EXPLICATION DES DIFFERENTS FICHIERS

MAIN.C

main() :

- Initialisations des interruptions et des exceptions (interrupt_init() & exception_init())
- Test 1 : accès non aligné de données
- Test 2 : déclenchement d'une interruption logicielle
- Test 3 : déclenchement d'une interruption non identifiée
- Test 4 : déclenchement d'une interruption « prefetch abort »
- Boucle sans fin (while(1))

EXCEPTION.C

exception_handler() :

- Affiche le type de l'interruption traitée

- Attend indéfiniment, si le vecteur correspond à une interruption de type « prefetch_abort »)

exception_init() :

- Attribue, à l'aide de la méthode « interrupt_attach(...) », type d'interruption, « exception_handler » et le nom de l'interruption (texte affiché)

INTERRUPT_ASM.S

interrupt_init_sp :

- Sauvegarde le mode courant
- Rendre dans le mode spécifié en paramètre (r0)
- Initialise le pointeur de pile de ce présent mode (sp) avec le paramètre passé à la fonction (r1)
- Retourne dans le mode précédant

interrupt_undefined_handler, interrupt_swi_handler, interrupt_prefetch_abort_handler, interrupt_data_abort_handler, interrupt_irq_handler, interrupt_fiq_handler :

- Corrige le « link register » (lr) d'un offset de 0 ou de -4 (cf. Interruptions, p20)
- Sauvegarde l'état des registres et du lr
- Précise l'adresse de retour (lr) de la fonction dans r0§
- Précise le type d'interruption (constante définie plus haut dans le fichier) dans r1
- Appel de la fonction « interrupt_process() » (Params : r0-1)
- Restauration des registres, du pc et du cpsr

interrupt_enable, interrupt_disable :

- Sauvegarde le mode courant
- Modifie les bits « I » et « F » : activation (bits à 0, « bic » avec la valeur « 0xc0 ») ou désactivation (bits à 1, « orr » avec la valeur « 0xc0 »)
- Retourne dans le mode précédant

INTERRUPT.C

interrupt_init() :

- Appel de la sous-routine « interrupt_init_sp » (ARM) afin d'initialiser les vecteurs d'interruptions avec leur pile de registres dédiés
- Place en mémoire la table des vecteurs
- Attribue aux « attributs » de la structure « vram » (instance) les adresses des différentes sous-routines d'interruptions

interrupt_attach(...) :

- Crée un pointeur vers l'adresse du vecteur dans la table des vecteurs
- Vérifie que le vecteur soit présent dans table
 - Si oui : on définit la sous-routine à utiliser

- Si non : on retourne -1

interrupt_detach(...) :

- Vérifie que le vecteur soit présent dans la table des vecteurs
 - Si oui : on attribue 0 à l'adresse de la sous-routine normalement employée pour traiter ce type d'interruption

TESTS ET VALIDATION

Afin de valider le bon fonctionnement de notre programme, développé en partie en C et en assembleur ARM, il devra implémenter une routine pour un traitement générique des exceptions et des interruptions logicielles. Lorsqu'une exception est levée, un message approprié devra être affiché sur le terminal de la cible.

Nous vérifions tout d'abord le cas d'une interruption « data abort » en produisant un non-alignement des données. En effet, en déclarant une variable sur 32 bits et en créant un pointeur sur celle-ci, incrémenté de 1, on crée une erreur d'alignement (adresse de la variable 1 additionnée de 1, grâce à un « cast » en char).

Le deuxième test consiste à créer une exception logicielle, en utilisant l'instruction « SWI » afin de changer de mode (passer en supervisor).

Le troisième test permet gérer les instructions non-définies.

Et le dernier test permet de détecter les interruptions dites « prefetch abort », lorsqu'une adresse quelconque est passée au « pc ».

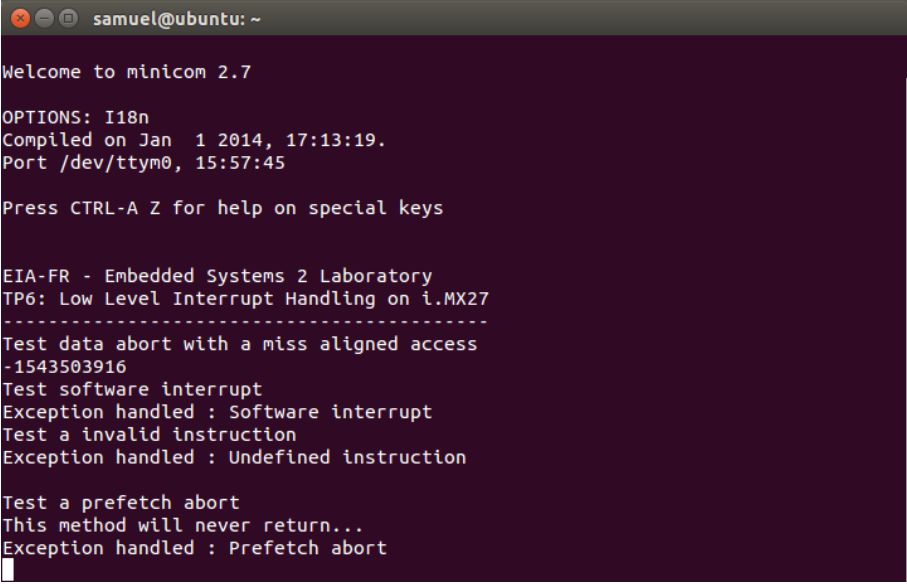
```
printf ("Test data abort with a miss aligned access\n");
uint32_t l = 0;
uint32_t * p1 = (uint32_t*)(char*) &l + 1;
printf("%d", *p1);

printf ("\nTest software interrupt\n");
__asm__ ("SWI #1;"); // generate SWI

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

printf ("\nTest a prefetch abort\nThis method will never return...\n");
__asm__ ("mov pc, #0x0f000000;"); // prefetch abort interrupt
```

En exécutant le programme au moyen du « JTagkey2 » et du debugger, nous pouvons constater que la levée des interruptions se passe correctement ; la capture ci-dessous l'atteste, fruit des tests présentés ci-dessus.



```
samuel@ubuntu: ~  
Welcome to minicom 2.7  
  
OPTIONS: I18n  
Compiled on Jan  1 2014, 17:13:19.  
Port /dev/tty0, 15:57:45  
  
Press CTRL-A Z for help on special keys  
  
EIA-FR - Embedded Systems 2 Laboratory  
TP6: Low Level Interrupt Handling on i.MX27  
-----  
Test data abort with a miss aligned access  
-1543503916  
Test software interrupt  
Exception handled : Software interrupt  
Test a invalid instruction  
Exception handled : Undefined instruction  
  
Test a prefetch abort  
This method will never return...  
Exception handled : Prefetch abort
```

Figure 6: Sortie sur la console lors de l'exécution du programme

PROBLEMES RENCONTRES & SOLUTIONS

Ce travail pratique n'est pas facile à aborder, dû au peu de code présent dans le template du Travail. Ne serait-il pas judicieux de proposer du pseudo-code, comme lors du TP précédant ? Le fait d'afficher le code durant la séance du laboratoire nous a permis de démarrer dans la bonne voie, ce qui n'aurait peut-être pas été le cas dans le cas contraire.

D'autre part, en testant notre code uniquement sur le « QEMU » nous avons des erreurs inopinées, qui faisaient planter tout le programme. En l'exécutant normalement sur la cible, tout est rentré dans l'ordre.

ACQUIS

Nous avons acquis les notions importantes concernant les interruptions ainsi que leur mise en place. Nous avons également compris que le passage d'un mode à l'autre implique un changement de contexte (tel qu'expliqué dans la partie d'analyse) et que chaque mode utilise des registres bien particuliers pour la sauvegarde de l'état du processeur.

PERSPECTIVES

Nous espérons pouvoir effectuer un débogage plus poussé pour nos programmes futurs. La priorisation d'une interruption pourrait éventuellement nous amener à nous demander si une interruption pourrait elle aussi être interrompue suivant un degré d'importance. Le traitement des interruptions prendra tout son sens avec les « GPIO », avec par exemples des actionneurs ou des capteurs pouvant déclencher des interruptions (thermomètre, TP4).

CONCLUSION

De prime abord, ce TP nous a paru très compliqué. Mais grâce aux différentes explications et à l'aide apportée par le professeur durant la séance de TP, nous avons été à même de comprendre le fonctionnement des interruptions en C.

ANNEXE

Le code source du programme se trouve sur Git, à l'adresse : <https://forge.tic.eia-fr.ch/git/samuel.mertenat/se12-tp/tree/master/tp6>.

SOURCES

- <http://infocenter.arm.com/help/index.jsp>
- Interruptions (cours, Moodle)
- Code source présenté au beamer durant le TP (TP06, M. Gachet)

```

/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
   Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: EIA-FR / Embedded Systems 2 Laboratory
 *
 * Abstract:   TP6 - Interrupt handling demo and test program
 *
 * Purpose: Main module to demonstrate and to test the i.MX27
 *           interrupt handling.
 *
 * Author:   Fabio Valverde & Samuel Mertenat
 * Date:     09.03.15
 */

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

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

/* -----
 */

int main () {
    printf ("\n");
    printf ("EIA-FR - Embedded Systems 2 Laboratory\n");
    printf ("TP6: Low Level Interrupt Handling on i.MX27\n");
    printf ("-----\n");

    interrupt_init();
    exception_init();

    printf ("Test data abort with a miss aligned access\n");
    uint32_t l = 0;
    uint32_t * pl = (uint32_t*)(char*) &l + 1;
    printf ("%d", *pl);

    printf ("\nTest software interrupt\n");
    __asm__ ("SWI #1;"); // generate SWI

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

    printf ("\nTest a prefetch abort\nThis method will never return...\n");
    __asm__ ("mov pc, #0x0f000000;"); // prefetch abort interrupt

    while(1);
    return 0;
}

```

```

#pragma once
#ifndef INTERRUPT_H
#define INTERRUPT_H
/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
 * Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: HEIA-FR/ Embedded Systems 2 Laboratory
 *
 * Abstract: ARM Interrupt Handling - Low Level Interface
 *
 * Purpose: Module to deal with the low level ARM9/i.MX27
 * microprocessor interrupt logic
 *
 * Author: Fabio Valverde & Samuel Mertenat
 * Date: 07.03.15
 */

/* interrupt vectors enumeration */
enum interrupt_vectors {
    INT_UNDEFINED, // Undefined instruction
    INT_SWI, // Software interrupt (SWI)
    INT_PREFETCH_ABORT, // Prefetch abort (instruction prefetch)
    INT_DATA_ABORT, // Data abort (data access)
    INT_IRQ, // (interrupt)
    INT_FIQ, // (fast interrupt)
    INT_NB_VECTORS // number of vectors, 6
};

/**
 * Prototype of the interrupt service routines
 *
 * @param addr return address
 * @param vector i.MX27 interrupt vector
 * @param param parameter specified while attaching the interrupt
 * service routine
 */
typedef void (*interrupt_isr_t) (void* addr,
                                enum interrupt_vectors vector,
                                void* param);

/**
 * Method to initialize low level resources of the ARM9 microprocessor
 * A 16KB of memory will be allocated for each interrupt vector
 */
extern void interrupt_init();

extern int interrupt_attach (enum interrupt_vectors vector, interrupt_isr_t

```

```

    routine, void* param);
extern void interrupt_detach (enum interrupt_vectors vector);
extern void interrupt_enable();
extern void interrupt_disable();

#endif

```

```

/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
 * Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: HEIA-FR/ Embedded Systems 2 Laboratory
 *
 * Abstract: ARM Interrupt Handling - Low Level Interface
 *
 * Author: Fabio Valverde & Samuel Mertenat
 * Date: 06.03.15
 *
 * Note: Part of the code is directly inspired by the code presented at the
 * time of the TP
 */

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

/* Low level interrupt handling -----
 */
/* declare low level routines implemented in assembler file interrupt_asm.s */
extern void interrupt_init_sp ();
extern void interrupt_init_sp (uint32_t mode, void* sp);
extern void interrupt_undefined_handler();
extern void interrupt_swi_handler();
extern void interrupt_prefetch_abort_handler();
extern void interrupt_data_abort_handler();
extern void interrupt_irq_handler();
extern void interrupt_fiq_handler();

/* Interrupt vector table contained in vram */
struct arm_vram {
    void (*undefined)();
    void (*swi)();
    void (*prefetch_abort)();
    void (*data_abort)();
    void (*irq)();
    void (*fiq)();
};
static volatile struct arm_vram* vram = (struct arm_vram*) 0xfffffef0;

/* defines stack for each processor mode */
/* 16k / 4 (.long) = 4096 -> 0x1000 */
static uint32_t undefined_stack[0x1000];
static uint32_t abort_stack[0x1000];
static uint32_t irq_stack[0x1000];

```

```

static uint32_t fiq_stack[0x1000];

/* interrupt service routines */
struct int_vector_entry {
    interrupt_isr_t handler;
    void* param;
};
static struct int_vector_entry int_vector_table[INT_NB_VECTORS];

/* interrupt handler called from assembler code */
void interrupt_process(void* address, enum interrupt_vectors vector) {
    // makes a pointer with the address of the given vector
    struct int_vector_entry* node = &int_vector_table[vector];
    if (node->handler != 0) {
        node->handler (address, vector, node->param);
    } else {
        printf("No interrupt handler defined for vector %d.", vector);
        while(1);
    }
}

/* Public methods -----
 */
/* Documentation: Interruptions (cours) - Modes de fonctionnement du
 * processeur - p15 */
/* Documentation: Interruptions (cours) - Etat du processeur - p17 */
void interrupt_init() {
    // defines the code & sp for each interruption vector
    // Ex: FIQ: 0xd1 = 0b.1 (I) 1 (F) 0 1 0 0 0 1 (17) = 209.
    interrupt_init_sp (0xd1, fiq_stack + sizeof(fiq_stack));
    interrupt_init_sp (0xd2, irq_stack + sizeof(irq_stack));
    interrupt_init_sp (0xd7, abort_stack + sizeof(abort_stack));
    interrupt_init_sp (0xdb, undefined_stack + sizeof(undefined_stack));

    memset (int_vector_table, 0, sizeof(int_vector_table));

    vram->undefined = interrupt_undefined_handler;
    vram->swi = interrupt_swi_handler;
    vram->prefetch_abort = interrupt_prefetch_abort_handler;
    vram->data_abort = interrupt_data_abort_handler;
    vram->irq = interrupt_irq_handler;
    vram->fiq = interrupt_fiq_handler;
}

int interrupt_attach(enum interrupt_vectors vector, interrupt_isr_t routine,
void* param) {
    // makes a pointer with the address of the given vector
    struct int_vector_entry* node = &int_vector_table[vector];
    if ((vector >= INT_NB_VECTORS) && (node->handler != 0))
        return -1; // returns -1 if an error has occurred (not in the table)
    node->handler = routine; // attaches the vector to the interruption
    node->param = param;
    return 0;
}

void interrupt_detach(enum interrupt_vectors vector) {
    // if the vector is present into the table, sets the vector's handler to 0
    if (vector < INT_NB_VECTORS)
        int_vector_table[vector].handler = 0;
}

```

```

/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
   Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: HEIA-FR/ Embedded Systems 2 Laboratory
 *
 * Abstract:   ARM Interrupt Handling - Low Level Interface
 *
 * Author:   Fabio Valverde & Samuel Mertenat
 * Date:     06.03.15
 */

/*-- exports -----
*/
.global interrupt_init_sp
.global interrupt_undefined_handler, interrupt_swi_handler
.global interrupt_prefetch_abort_handler, interrupt_data_abort_handler
.global interrupt_irq_handler, interrupt_fiq_handler
.global interrupt_enable, interrupt_disable

/*-- local variable -----
*/
/* Documentation: Interruptions (cours) - Table des vecteurs d'interruptions -
p13 */
.bss
INT_UNDEFINED      = 0 // Undefined instruction
INT_SWI            = 1 // Software interrupt (SWI)
INT_PREFETCH_ABORT = 2 // Prefetch abort (instruction prefetch)
INT_DATA_ABORT     = 3 // Data abort (data access)
INT_IRQ            = 4 // (interrupt)
INT_FIQ            = 5 // (fast interrupt)

/*-- public methods -----
*/
.text

/* MRS: Move to ARM register from system coprocessor register */
/* MSR: Move to system coprocessor register from ARM register */
interrupt_init_sp:
    nop
    mrs r2, cpsr // saves the current mode into r2 from cpsr
    msr cpsr_c, r0 // switches to the mode given by the first param (r0)
    mov sp, r1 // initiates the stack pointer with the second given param
                  (r1)
    msr cpsr_c, r2 // restores to the previous mode (affects only the 8 first
                  bits)
    bx lr

```

```

/* STMFD SP!, {<registers>, LR}: to save the registers */
/* LDMFD SP!, {<registers>, LR}: to restore the registers */
/* Documentation: Interruptions (cours) - Sortie de la routine d'interruption
- p20 */
/* Offset of -4: Data A., FIQ, IRQ, Pre-fetch Abort. The others: 0 */
interrupt_undefined_handler:
    nop
    //sub    lr, #0 // adjusts the returned value with the offset
    stmfd    sp!, {r0-r12,lr} // saves the registers & the link register
    mov      r0, lr // saves the return address (lr)
    mov      r1, #INT_UNDEFINED // saves the vector number
    bl       interrupt_process // calls interrupt_process(r0, r1) (C
                                function)
    ldmfd     sp!,{r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx     lr

interrupt_swi_handler:
    nop
    //sub    lr, #0 // adjusts the returned value with the offset
    stmfd    sp!, {r0-r12,lr} // saves the registers & the link register
    mov      r0, lr // saves the return address (lr)
    mov      r1, #INT_SWI // saves the vector number
    bl       interrupt_process // calls interrupt_process(r0, r1) (C
                                function)
    ldmfd     sp!,{r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx     lr

interrupt_prefetch_abort_handler:
    nop
    sub      lr, #4 // adjusts the returned value with the
                    offset
    stmfd    sp!, {r0-r12,lr} // saves the registers & the link register
    mov      r0, lr // saves the return address (lr)
    mov      r1, #INT_PREFETCH_ABORT // saves the vector number
    bl       interrupt_process // calls interrupt_process(r0, r1) (C
                                function)
    ldmfd     sp!,{r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx     lr

interrupt_data_abort_handler:
    nop
    sub      lr, #4 // adjusts the returned value with the offset
    stmfd    sp!, {r0-r12,lr} // saves the registers & the link register
    mov      r0, lr // saves the return address (lr)
    mov      r1, #INT_DATA_ABORT // saves the vector number
    bl       interrupt_process // calls interrupt_process(r0, r1) (C
                                function)
    ldmfd     sp!,{r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx     lr

interrupt_irq_handler:
    nop
    sub      lr, #4 // adjusts the returned value with the offset
    stmfd    sp!, {r0-r12,lr} // saves the registers & the link register
    mov      r0, lr // saves the return address (lr)
    mov      r1, #INT_IRQ // saves the vector number
    bl       interrupt_process // calls interrupt_process(r0, r1) (C
                                function)
    ldmfd     sp!,{r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx     lr

```

```

interrupt_fiq_handler:
    nop
    sub    lr, #4           // adjusts the returned value with the offset
    stmfd  sp!, {r0-r12,lr} // saves the registers & the link register
    mov    r0, lr           // saves the return address (lr)
    mov    r1, #INT_FIQ     // saves the vector number
    bl     interrupt_process // calls interrupt_process(r0, r1) (C
                             // function)
    ldmdf  sp!, {r0-r12,pc}^ // restores back the registers, pc & cpsr
    //bx   lr

/* BIC: Bit clear (Rd AND NOT Rm) */
/* Source code: Ex1 - P4 - Série 1 */
/* Documentation: Interruptions (cours) - Acceptation des interruptions - p30
   */
interrupt_enable:
    nop
    mrs    r0, cpsr         // saves the current mode into r0
    bic    r0, #0xc0        // sets the bits I 6 F to 0 (interruptions enabled)
    msr    cpsr_c, r0       // restores to the previous mode
    bx     lr

/* ORR: bitwise OR operation */
interrupt_disable:
    nop
    mrs    r0, cpsr         // saves the current mode into r0
    orr    r0, #0xc0        // sets the bits I 6 F to 1 (interruptions disabled)
    msr    cpsr_c, r0       // restores to the previous mode
    bx     lr

```



```
#pragma once
#ifndef EXCEPTION_H
#define EXCEPTION_H
/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
    Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: EIA-FR./ Embedded Systems 2 Laboratory
 *
 * Abstract:    ARM Exception Handling
 *
 * Purpose: Module do deal with ARM exception handling
 *
 * Author:    Fabio Valverde & Samuel Mertenat
 * Date:      07.03.15
 */

/**
 * initialization method
 */
extern void exception_init();

#endif
```

```

/**
 * Copyright 2014 University of Applied Sciences Western Switzerland /
 * Fribourg
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Project: EIA-FR·/ Embedded Systems 2 Laboratory
 *
 * Abstract:    ARM Exception Handling
 *
 * Author:    Fabio Valverde & Samuel Mertenat
 * Date:      07.03.15
 */

#include <stdio.h>

#include "interrupt.h"

static void exception_handler(void* addr, enum interrupt_vectors vector, void*
    param) {
    printf("Exception handled : %s\n", (char*)param);
    while(vector == INT_PREFETCH_ABORT) {
    }
}

void exception_init() {
    interrupt_attach(INT_UNDEFINED, exception_handler, "Undefined instruction"
    );
    interrupt_attach(INT_PREFETCH_ABORT, exception_handler, "Prefetch abort");
    interrupt_attach(INT_DATA_ABORT, exception_handler, "Data abort");
    interrupt_attach(INT_SWI, exception_handler, "Software interrupt");
}

```