



SYSTEMES EMBARQUES 2

TP07 – TRAITEMENT DES INTERRUPTIONS MATERIELLES

Fabio Valverde & Samuel Mertenat
Classe T2f

Fribourg, le 16 mars 2015
Salle C0016
08h15-11h50

TABLE DES MATIERES

1	Buts et Objectifs du travail pratique.....	3
2	Analyse	3
2.1	AITC.....	3
2.2	AITC – Registres	4
2.3	GPIO.....	5
3	Conception.....	5
3.1	aitc_init.....	6
3.2	aitc_attach.....	6
3.3	aitc_detach	7
3.4	aitc_force.....	7
4	Tests et validation	7
4.1	Phase de simulation.....	7
4.2	Phase concrète	9
5	Acquis	9
6	Problèmes.....	10
7	Perspectives	10
8	Conclusion.....	10
9	Références	10
10	Annexes	10

SYSTEMES EMBARQUES 2

TP07 – TRAITEMENT DES INTERRUPTIONS MATERIELLES

1 BUTS ET OBJECTIFS DU TRAVAIL PRATIQUE

Ce travail pratique est la suite logique du travail précédent et consiste à implémenter une petite application permettant de traiter les interruptions matérielles générées, dans notre cas, à l'aide de boutons-poussoirs, reliés au GPIO (« General Purpose Input / Output »).

Objectifs :

- Décrire le traitement des interruptions matérielles du processeur ARM
- Concevoir et développer une petite application en C et en assembleur permettant le traitement des interruptions matérielles du 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 (aitc_init)
- Attacher une routine de service des interruptions (ISR) à un vecteur donné (aitc_attach). Cette méthode doit permettre de spécifier que le type d'interruption (FIQ ou IRQ) ainsi qu'un paramètre avec l'ISR. Ce dernier sera passé 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 routine retourne la valeur -1.
- Détacher l'ISR d'un vecteur d'interruption (aitc_detach)
- Forcer/simuler une interruption matérielle (aitc_force).

2 ANALYSE

2.1 AITC

L'AITC est un contrôleur d'interruptions. Il permet de collecter les requêtes d'interruptions de différentes sources internes au processeur et est capable de prioriser ces sources et de générer un vecteur d'interruption pour celles-ci.

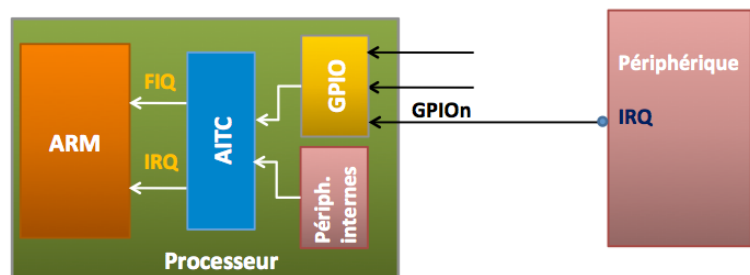


Figure 1: Système d'interruptions

Les sources d'interruptions peuvent être internes (Timer, SPI, I2C) ou externes (GPIO : boutons, capteurs, etc.).

La liste complète des sources d'interruptions pour la cible i.MX27 peut être consultée aux pages 414-416 du document "02_ARM_i.MX27_Reference_Manual.pdf" (section 10.2.12 – disponible sur Moodle).

Chaque ligne d'interruption peut être priorisée à l'aide du contrôleur AITC. Lorsque des interruptions surviennent, l'AITC peut ainsi traiter la source prioritaire en fonction du niveau de priorité attribué aux différentes sources d'interruptions.

IF (Source-Priority > NIMASK) OR (NIMASK == -1) → INTERRUPT

	NIMASK	Source Priority Level							
		0	1	2	3	...	14	15	
	-1	✓	✓	✓	✓	✓	✓	✓	Unaccepted interrupts
	0	x	✓	✓	✓	✓	✓	✓	
	1	x	x	✓	✓	✓	✓	✓	
Current Mask →	2	x	x	x	✓	✓	✓	✓	
	3	x	x	x	x	✓	✓	✓	Accepted interrupts
	...	x	x	x	x	x	✓	✓	
	14	x	x	x	x	x	x	✓	
	15	x	x	x	x	x	x	x	

Figure 2: Niveaux de priorité des interruptions

Pour les interruptions de type IRQ, l'AITC implémente un décodeur de priorité prenant en compte uniquement les sources ayant un niveau suffisant pour être traitées¹.

2.2 AITC – REGISTRES

Plusieurs registres sont nécessaires afin de réaliser la bibliothèque « aitc ». Les principaux registres sont les suivants :

- Interrupt Control Register (INTCNTL) : registre permettant de contrôler les interruptions de l'AITC et de lui lier une table de traitement des différents vecteurs (handlers ; bits « Pointer »)
- Normal Interrupt Mask Register (NIMASK) : registre permettant de définir un « masque », qui peut définir les niveaux des interruptions (initialisé à -1 afin de ne pas utiliser cette fonctionnalité)
- Interrupt Enable Number Register (INTENNUM) : registre permettant d'activer les interruptions pour chaque type de vecteur
- Interrupt Disable Number Register (INTDISNUM) : registre permettant de désactiver les interruptions pour chaque type de vecteur
- Interrupt Enable Register High / Low (INTENABLEH/L) : registres permettant de lister les interruptions en cours pour toutes les sources disponibles (2 registres ; 64 sources). Ces registres peuvent être modifiés directement ou par l'intermédiaire des registres « INTENNUM » ou « INTDISNUM ».
- Interrupt Type Register High / Low (INTTYPEH/L) : registres permettant de définir le type d'interruptions (IRQ ou FIQ) pour chaque source d'interruptions

¹ Systèmes Embarqués 1 & 2, Interruptions, p33, D. Gachet

- Interrupt Force Register High / Low (INTFRCH/L) : registres permettant de générer de manière logicielle toutes les différentes sources d'interruptions. Ils sont généralement utilisés à des fins de débogage

Plus d'informations sur les registres relatifs à l'AITC se trouvent aux pages 396-419 (10.2.3 – 10.2.15, 02_ARM_i.MX27_Reference_Manual.pdf).

2.3 GPIO

Les « General Purpose Input / Output » (GPIO) sont des ports, qui peuvent être configurés en tant qu'entrée ou en tant que sortie, auxquels on peut associer des lignes d'interruptions².

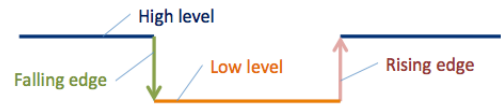


Figure 3: Détection d'une interruption

Utilisés en tant qu'entrée, ces ports permettent de détecter, si activés, les variations suivantes et ainsi générer une interruption :

- Niveau haut du signal (high level) : lorsque le signal est de niveau High
- Flanc descendant (falling edge) : lorsque le signal passe du niveau High au niveau Low
- Niveau bas du signal (low level) : lorsque le signal est de niveau Low
- Flanc montant (rising edge) : lorsque le signal passe du niveau Low à High

La cible i.MX27 comporte 6 ports GPIO de 32 broches (port A à port F) ; le schéma de liaisons est disponible sur Moodle (12_HES_fribourg_expand_schema.pdf).

3 CONCEPTION

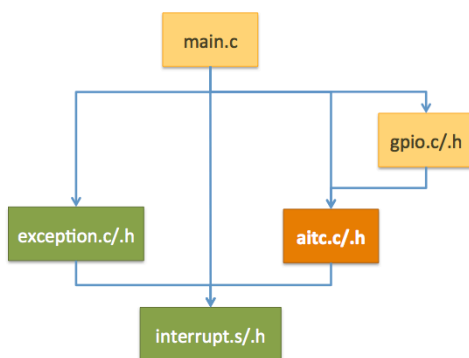


Figure 4: Squelette de l'application

Le programme est décomposé en plusieurs fichiers, dont seuls les fichiers airc.c et airc.h sont à réaliser ; les autres fichiers ont été développés lors du travail précédent ou sont disponibles directement dans le dossier « workspace/se12/apf27/source/ ».

La bibliothèque « airc » contiendra les fonctions « airc_init », initialisant la bibliothèque, « airc_attach », permettant d'attacher une routine d'interruptions à un vecteur, « airc_detach » pour détacher la routine d'un vecteur et « airc_force », permettant de simuler une interruption matérielle. Ces fonctions sont donc

déclarées dans le fichier « airc.h » et implémentées dans « airc.c ». Les prototypes de ces fonctions sont les suivants :

² Systèmes embarqués 1 & 2, TP7, D. Gachet, p35

```
// Method to initialize AITC controller
extern void aitc_init();
// Method to attach an interrupt routine to the AITC
extern int aitc_attach(enum aitc_interrupt_vectors vector, enum aitc_interrupt_types type, aitc_isr_t
routine, void* param);
// Method to detach an interrupt routine from the AITC
extern int aitc_detach(enum aitc_interrupt_vectors vector);
// Method to force an interrupt
extern void aitc_force(enum aitc_interrupt_vectors vector, bool state);
```

3.1 AITC_INIT

Paramètre(s) : -

Retour : -

Descriptif :

- Initialisation du registre de contrôle (INTCNTL)
- Initialisation du registre de priorité (NIMASK) à -1, ce qui n'affecte pas le niveau de priorité par défaut des différents types d'interruptions.
- Initialisation des registres « INTENABLEH » et « INTENABLEL » à 0, afin de réinitialiser toutes les interruptions
- Initialisation du registre des priorités (NIPRIORITY), en définissant une priorité de 0 à toutes les interruptions possibles.
- Attachement des interruptions « FIQ » et « IRQ » avec la méthode « interrupt_attach() »

3.2 AITC_ATTACH

Paramètres : Source d'interruption, type d'interruption, routine d'interruption, paramètre

Retour : « 0 » ou « -1 » en cas d'erreur

Descriptif :

- Vérifie que le vecteur passé en paramètre soit valide et qu'aucune routine n'ait déjà été assignée au « Handler » pour ce vecteur
- Si ok :
 - Attribue au « Handler » de ce vecteur la routine et le paramètre passés en arguments
 - Modifie les registres « INTTYPEH » / « INTYPEL » en fonction du type de vecteur transmis (AITC_FIQ / AITC_IRQ)
 - Active les interruptions pour ce type de vecteur en modifiant le registre « INTENUM »
 - Retourne 0
- Si pas ok :
 - Retourne -1

3.3 AITC_DETACH

Paramètre : Source d'interruption

Retour : « 0 » ou « -1 » en cas d'erreur

Descriptif :

- Vérifie que le vecteur à détacher passé en paramètre soit valide
- Si ok :
 - Désactive les interruptions pour ce type de vecteur en modifiant le registre « INTDISNUM »
 - Initialise les paramètres attribués au « Handler » pour ce vecteur
 - Retourne 0
- Si pas ok :
 - Retourne -1

3.4 AITC_FORCE

Paramètre(s) : Source d'interruption, « true » ou « false »

Retour : -

Descriptif :

- Vérifie que le vecteur à simuler, passé en paramètre, soit valide
- Si ok :
 - Génère une FIQ ou une IRQ pour le vecteur passé en paramètre, selon la valeur du booléen, en affectant les registres « INTFRCH/L ».

4 TESTS ET VALIDATION

La validation du travail réalisé est séparée en deux phases distinctes : la phase relative à la simulation d'une interruption matérielle à l'aide de la méthode « aitc_force » et la phase plus « concrète », faisant interagir boutons-poussoirs et affichage 7-segments afin de matérialiser un compteur.

4.1 PHASE DE SIMULATION

Le premier test consiste à tester la fonction « aitc_force() », permettant de simuler une interruption matérielle, afin de vérifier le bon fonctionnement de la bibliothèque « aitc ». Cette fonction permet de générer de façon « software » une interruption de n'importe quelle source ; généralement utilisée à des fins de débogage, elle permet d'atteindre directement les registres « Interrupt Force Register High (INTFRCH) and Low (INTFRCL) » et


de définir le bit correspondant au vecteur d'interruption à 1 ou à 0, selon le paramètre passé à la fonction³.

```
void airc_force(enum airc_interrupt_vectors vector, bool force) {  
    if (vector < AIRC_NB_OF_VECTORS) {  
        if (force)  
            airc->intfrfc[1 - (vector / 32)] |= (1 << (vector % 32));  
        else  
            airc->intfrfc[1 - (vector / 32)] &= ~(1 << (vector % 32));  
    }  
}
```

Cette fonction est appelée au sein de la méthode « airc_test() », qui fait référence aux autres fonctions de la librairie que sont « airc_force() » et « airc_detach () ».

```
void airc_test(void* param) {  
    printf ("AIRC interrupt simulation on GPT6 occurred\n");  
    airc_force(AIRC_GPT6, false);  
    airc_detach(AIRC_GPT6);  
    *(bool*)param = true;  
}
```

Suite à l'appel de cette fonction, nous pouvons constater que tout se déroule convenablement ; le programme poursuit son exécution jusqu'au prochain test, qui consiste à ce que l'utilisateur presse sur le bouton relié à la patte 4 du GPIO (cf. figure 4 ci-dessous).



```

Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Jan  1 2014, 17:13:19.
Port /dev/ttyM0, 21:14:03

Press CTRL-A Z for help on special keys

EIA-FR - Embedded Systems 2 Laboratory
TP7: i.MX27 Hardware Interrupt Handling Test Program
-----
AITC interrupt simulation on GPT6 occurred ← airc_force() + airc_test()
--> press gpio switch 1 to continue...
GPIO interrupt test on port_e pin_4 occurred ← gpio_e4_test()

```

Figure 5: Sortie sur la console lors de l'exécution du programme

Ce second test consiste à attendre que le bouton-poussoir soit pressé avant de pouvoir passer à la partie plus « concrète » du programme, abordée dans le point suivant (4.2).

Dès que le bouton est pressé, la fonction « gpio_e4_test() » est appelée et affiche un message à l'utilisateur (cf. figure 4 ci-dessus).

```
void gpio_e4_test(void* param) {  
    printf ("GPIO interrupt test on port_e pin_4 occurred\n");  
    imx27_gpio_detach (IMX27_GPIO_PORT_E, 4);  
    *(bool*)param = true;  
}
```

³ 02_ARM_i.MX27_Reference_Manual.pdf (Moodle), p416-417

Cette « procédure » est possible, car la fonction avait été précédemment liée au bouton-poussoir attaché à la patte 4 du GPIO, par les deux premières lignes de code ci-dessous.

```
imx27_gpio_attach (IMX27_GPIO_PORT_E, 4, IMX27_GPIO_IRQ_FALLING, gpio_e4_test, &gpio_has_been_pressed);  
imx27_gpio_enable (IMX27_GPIO_PORT_E, 4);  
printf (" --> press gpio switch 1 to continue...\n");  
while (!gpio_has_been_pressed);
```

4.2 PHASE CONCRETE

Dès lors qu'une interruption matérielle est parfaitement détectable, nous pouvons passer à la vérification du bon fonctionnement du compteur de « clics ».

Ce compteur de « clics » utilise l'affichage 7-segments pour présenter le nombre de clics et les boutons reliés aux pattes 3 (incrémenter le compteur) ou 6 (réinitialiser le compteur) du GPIO.

```
/* application initialization */  
int counter = 0;  
imx27_gpio_attach (IMX27_GPIO_PORT_E, 3, IMX27_GPIO_IRQ_FALLING, gpio_count, &counter);  
imx27_gpio_enable (IMX27_GPIO_PORT_E, 3);  
imx27_gpio_attach (IMX27_GPIO_PORT_E, 6, IMX27_GPIO_IRQ_FALLING, gpio_reset, &counter);  
imx27_gpio_enable (IMX27_GPIO_PORT_E, 6);
```

Aucune information n'est transmise sur la console afin d'attester du bon fonctionnement du programme, mais la figure ci-dessous est la preuve « vivante » de la bonne implémentation de la bibliothèque du contrôleur de l'AITC.

Le comportement des deux boutons-poussoirs sont conformes à la donnée et l'affichage 7-segments retransmet un nombre tout à fait plausible du nombre de clics réalisés à l'aide du bouton de gauche.



Figure 6: Programme en cours d'exécution

5 ACQUIS

Ce travail pratique nous a permis de renforcer notre compréhension du fonctionnement des interruptions, appréhendé au travail précédent et de rajouter une pièce à notre puzzle : les interruptions liées au matériel.

6 PROBLEMES

Nous avons rencontré quelques problèmes durant ce travail avec les fichiers « airc.h », dont l'énumération des vecteurs n'était pas correcte ou dans « airc.c » où un oubli s'était glissé dans la fonction « airc_attach() » (activer les interruptions pour le vecteur ; registre « INTENNUM »). D'autre part, le fonctionnement du programme lors de phases de débogages était parfois aléatoire ; un « clean » du projet a, nous semble-t-il, pallié à ce problème.

7 PERSPECTIVES

Les possibilités offertes par l'utilisation du GPIO et des interruptions sont maintenant énormes. On peut maintenant imaginer des programmes « plus intelligent » que ceux réalisés précédemment, en utilisant par exemple, non plus des boucles afin de détecter des changements liés aux boutons, mais des interruptions.

8 CONCLUSION

Ce travail pratique nous a permis d'approfondir nos connaissances dans le domaine des interruptions, en particulier au niveau des interruptions liées au matériel. D'autre part, ce travail nous a permis de survoler le fonctionnement du GPIO, ainsi que de pratiquer toujours un peu plus le langage C.

9 REFERENCES

- Systèmes embarqués 1 & 2, TP7, D. Gachet
- 02_ARM_i.MX27_Reference_Manual.pdf (Moodle), p396-419

10 ANNEXES

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/tp7>.

```

/**
 * Copyright 2015 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:      TP7 – Hardware Interrupt Handling
 *
 * Purpose:       Main module to demonstrate and to test the i.MX27
 *                interrupt handling.
 *
 * Authors:       Fabio Valverde & Samuel Mertenat
 * Date:          19.03.2015
 */

```

```

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

```

```

#include <imx27_gpio.h>

```

```

#include "interrupt.h"
#include "exception.h"
#include "display.h"
#include "aitc.h"

```

```

/* -----
 */

```

```

void airc_test(void* param) {
    printf ("AIRC interrupt simulation on GPT6 occurred\n");
    airc_force(AIRC_GPT6, false);
    airc_detach(AIRC_GPT6);
    *(bool*)param = true;
}

```

```

void gpio_e4_test(void* param) {
    printf ("GPIO interrupt test on port_e pin_4 occurred\n");
    imx27_gpio_detach (IMX27_GPIO_PORT_E, 4);
    *(bool*)param = true;
}

```

```

void gpio_count(void* param) {
    (*(int*)param)++;
}

```

```

void gpio_reset(void* param) {

```

```

    *(int*)param = 0;
}

/* -----
*/

int main () {
    printf ("\n");
    printf ("EIA-FR - Embedded Systems 2 Laboratory\n");
    printf ("TP7: i.MX27 Hardware Interrupt Handling Test Program\n");
    printf ("-----\n");

    /* modules initialization */
    interrupt_init();
    exception_init();
    airc_init();
    imx27_gpio_init();
    airc_attach(AIRC_GPIO, AIRC_IRQ, imx27_gpio_isr, 0);
    display_init();
    interrupt_enable();

    /* test airc interrupt processing */
    bool airc_simul_has_occured = false;
    airc_attach(AIRC_GPT6, AIRC_IRQ, airc_test, &airc_simul_has_occured);
    airc_force(AIRC_GPT6, true);
    while (!airc_simul_has_occured);

    /* test gpio interrupt processing */
    bool gpio_has_been_pressed = false;
    imx27_gpio_attach (IMX27_GPIO_PORT_E, 4, IMX27_GPIO_IRQ_FALLING,
        gpio_e4_test, &gpio_has_been_pressed);
    imx27_gpio_enable (IMX27_GPIO_PORT_E, 4);
    printf (" --> press gpio switch 1 to continue...\n");
    while (!gpio_has_been_pressed);

    /* application initialization */
    int counter = 0;
    imx27_gpio_attach (IMX27_GPIO_PORT_E, 3, IMX27_GPIO_IRQ_FALLING,
        gpio_count, &counter);
    imx27_gpio_enable (IMX27_GPIO_PORT_E, 3);
    imx27_gpio_attach (IMX27_GPIO_PORT_E, 6, IMX27_GPIO_IRQ_FALLING,
        gpio_reset, &counter);
    imx27_gpio_enable (IMX27_GPIO_PORT_E, 6);

    /* main loop */
    while(1) {
        display_value (counter > 99 ? -99 : counter);
    }
    return 0;
}

```

```

#pragma once
#ifndef AITC_H
#define AITC_H
/**
 * Copyright 2015 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:     AITC Driver
 *
 * Purpose:      Module designed to manage the AITC controller
 *
 * Authors:      Fabio Valverde & Samuel Mertenat
 * Date:         19.03.2015
 */
#include <stdbool.h>
#include <stdint.h>

// List of the used AITC vectors (64 available)
// Documentation: 02_ARM....pdf – Interrupt Assignments Low – 10.2.12.2 – p415
enum aitc_interrupt_vectors {
    AITC_Reserved,
    AITC_GPT6,
    AITC_GPIO,
    AITC_NB_OF_VECTORS
};

// Type of AITC interruptions
enum aitc_interrupt_types {
    AITC_IRQ,
    AITC_FIQ
};

// Interrupts routine
typedef void(*aitc_isr_t)(void*param);

// Method to initialize AITC controller
extern void aitc_init();
// Method to attach an interrupt routine to the AITC
extern int aitc_attach(enum aitc_interrupt_vectors vector, enum
    aitc_interrupt_types type, aitc_isr_t routine, void* param);
// Method to detach an interrupt routine from the AITC
extern int aitc_detach(enum aitc_interrupt_vectors vector);
// Method to force an interrupt
extern void aitc_force(enum aitc_interrupt_vectors vector, bool state);

#endif

```

```

/**
 * Copyright 2015 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:     AITC Driver
 *
 * Purpose:      Module designed to manage the AITC·controller
 *
 * Authors:      Fabio Valverde & Samuel Mertenat
 * Date:         19.03.2015
 */

```

```

#include <stdbool.h>
#include "interrupt.h"
#include "aitc.h"
#include <stdint.h>

```

```

// AITC registers
// Documentation: 02_ARM....pdf – Register Summary – 10.2.2 – p392–6
struct aitc_regs {
    uint32_t intcntl;
    uint32_t nimask;
    uint32_t intennum;           // enables a line
    uint32_t intdisnum;         // disables a line
    uint32_t intenable[2];
    uint32_t inttype[2];        // interrupt type (IRQ or FIQ)
    uint32_t nipriority[8];
    uint32_t nivecsr;
    uint32_t fivecsr;
    uint32_t intsrc[2];
    uint32_t intfrc[2];         // register to force interruptions
    uint32_t nipnd[2];
    uint32_t fipnd[2];
};

```

```

static volatile struct aitc_regs* aitc = (struct aitc_regs*)0x10040000;

```

```

//·AITC routine parameters

```

```

struct aitc_routine_parameters {
    aitc_isr_t routine;
    void * param;
};

```

```

//·cf. enum aitc_interrupt_vectors; aitc.h
struct aitc_routine_parameters handlers[64];

```

```

// Method to handle interrupts
// Documentation: 02_ARM....pdf – NIVECSR +·FIVECSR – 10.2.10–1 – p411–2

```

```

static void airc_handler(void* addr, enum airc_interrupt_vectors vector, void*
    param) {
    uint32_t vect = 0;
    if (vector == INT_IRQ)
        // if it's an IRQ, reads the register, shifted by 16 bits
        vect = airc -> nivecscr >> 16;
    else
        // if it's a FIQ, reads the register
        vect = airc -> fivecsr;
    handlers[vect].routine(handlers[vect].param);
}

void airc_init() {
    airc->intcntl = 0;           // initializes the Interrupt Control Register
    airc->nimask = -1;           // does not disable any normal interrupts priority
                                // levels (-1)
    airc->intenable[0] = 0;      // resets all pending interrupt requests
    airc->intenable[1] = 0;
    // sets the priority level to 0 (default; highest: 1 - 7 : lowest)
    // Documentation: Interrupts (cours) - Priorité des interruptions - p36
    for (uint8_t i = 0; i < 8; i++) {
        airc->nipriority[i] = 0;
    }
    airc->intfrc[0] = 0;         // initializes the Interrupt Control Register
    airc->intfrc[1] = 0;
    // attaches the interrupt's vectors for IRQ & FIQ to the handler
    interrupt_attach(INT_IRQ, airc_handler, 0);
    interrupt_attach(INT_FIQ, airc_handler, 0);
}

int airc_attach(enum airc_interrupt_vectors vector, enum airc_interrupt_types
    type, airc_isr_t routine, void* param) {
    int8_t status = -1;
    if((vector < AITC_NB_OF_VECTORS) && (handlers[vector].routine == 0)) {
        handlers[vector].routine = routine;
        handlers[vector].param = param;
        // retrieve the type of the interrupt
        if(type == AITC_FIQ) {
            airc->inttype[1-(vector/32)] |= (1<<(vector % 32));    // FIQ :
                                bit = 1
        } else {
            airc->inttype[1-(vector/32)] &= ~(1<<(vector % 32));    // IRQ :
                                bit = 0
        }
        airc->intenum = vector;    // enables interruptions for this vector
        status = 0;
    }
    return status;
}

int airc_detach(enum airc_interrupt_vectors vector) {
    int8_t status = -1;
    if(vector < AITC_NB_OF_VECTORS) {
        airc->intdisnum = vector;    // disables the interrupts for this
                                vector
        handlers[vector].routine = 0;    // resets the handler for this vector
        handlers[vector].param = 0;
        status = 0;
    }
    return status;
}

```

```
}  
  
void airc_force(enum airc_interrupt_vectors vector, bool force) {  
    if (vector < AIRC_NB_OF_VECTORS) {  
        if (force)  
            airc->intfrc[1 - (vector / 32)] |= (1 << (vector % 32));  
        else  
            airc->intfrc[1 - (vector / 32)] &= ~(1 << (vector % 32));  
    }  
}
```