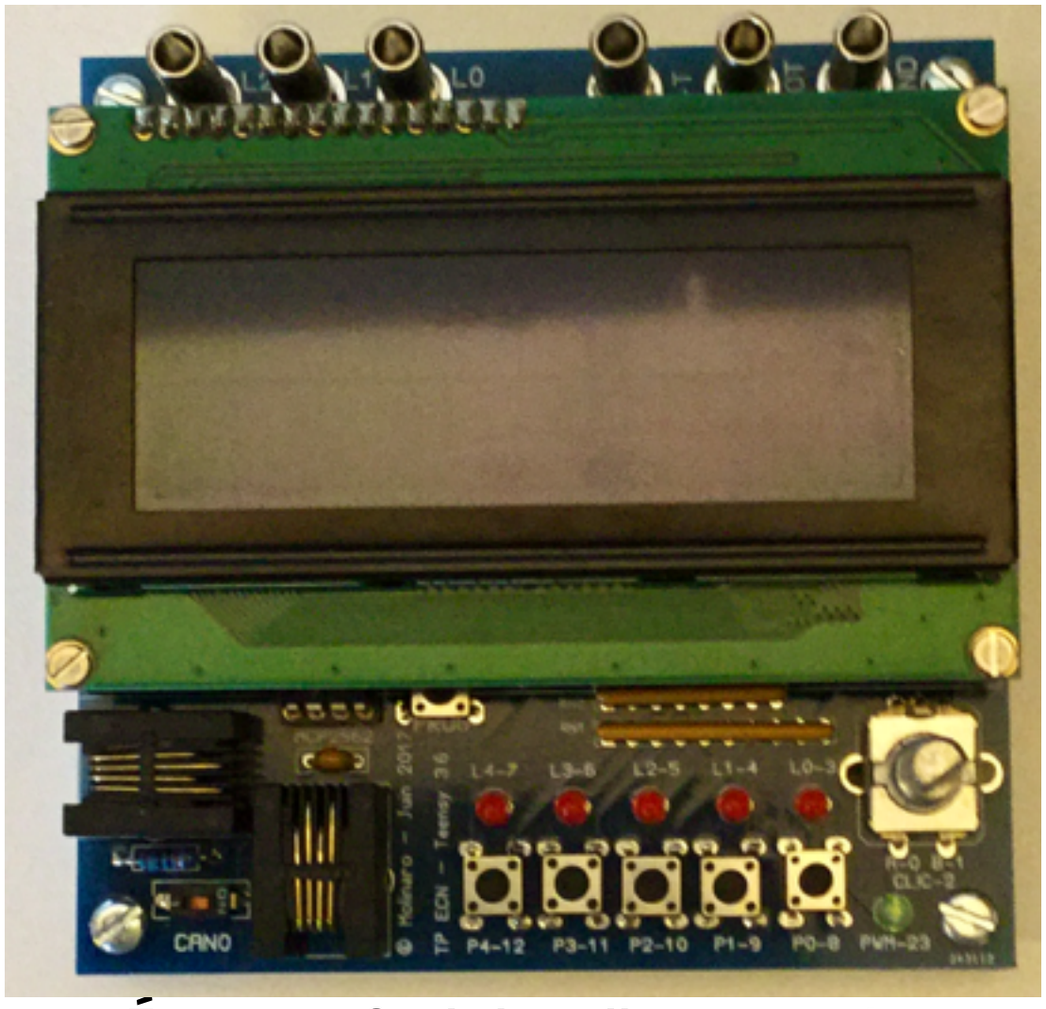
# Temps Réel




*Étape 10-fault-handler--assertion*

CENTRALE NANTES

**Pierre Molinaro**
**Juin 2018**

# Erreur, faute, défaillance

Avant d'entrer dans le détail des explications de cette étape, il convient de définir les termes ***erreur***, ***faute***, ***défaillance***.

**Une erreur (*error*) est la fourniture d'un résultat invalide.** Par exemple, un résultat négatif (alors qu'il devrait positif ou nul), un pointeur nul (alors qu'il devrait être non nul). Une erreur n'a pas de conséquence visible, tant que le résultat n'est pas exploité.

**Une faute (*fault*) est l'impossibilité de réaliser une opération** : par exemple, extraire une racine carrée d'un nombre négatif, atteindre l'objet référencé par un pointeur nul. Dans du code engendré par un compilateur C++, une faute peut provoquer une exception (dans le sens de ce langage).

**Une défaillance (*failure*) est la non réalisation du service demandé.** On observe que le fonctionnement n'est pas celui attendu (tiens, ça a planté !).

**Lien :**

https://www.nasa.gov/pdf/636745main_day_3-algirdas_avizienis.pdf

# Description de cette étape

Les fichiers qui vont être ajoutés dans cette étape (**fault-handlers--assertion.cpp** et **fault-handlers--assertion.h**) vont permettre d'afficher trois sortes de fautes :

- celles détectées par le processeur lui-même ;
- le déclenchement d'une interruption sans que son *handler* soit spécifié ;
- le non respect d'une assertion.

Dans les trois cas, l'occurrence d'une provoque :

- le clignotement synchrone des cinq leds ;
- l'affichage d'un message d'erreur.

L'exécution sera bloquée sur cet affichage.

Le fichier **apnt209.pdf** est un document qui décrit les registres du Cortex-M4 concernés par les interruptions ***Fault*** (voir plus loin, division entière par zéro).

**Travaux à faire :**

- à titre d'exemple, on écrira trois programmes mettant en évidence chaque type de faute.

# Erreurs détectées par le processeur

Un processeur Cortex-M4 est capable d'engendrer une interruption quand certaines fautes surviennent :

- tentative d'accès à une adresse mémoire non autorisée ;
- exécution d'une instruction invalide ;
- …

Certains comportements sont paramétrables. Par exemple, la division entière par zéro :

- par défault, la division par zéro retourne silencieusement un quotient nul : c'est le comportement dans toutes les étapes précédentes ;
- à partir de cette étape et dans toutes les suivantes, le fichier **fault-handlers--assertion.cpp** change ce comportement par défaut, une interruption est déclenchée lors d'une division entière par zéro.

CENTRALE NANTES

# Interruption sans *handler* associé

Jusqu'à présent, cette exception n'était pas exploitée, si bien qu'une faute provoquait l'entrée dans une boucle sans fin (comme toutes les interruptions inutilisées, dans **unused-interrupt.s**).

Les fichiers qui vont être ajoutés dans cette étape (**fault-handlers--assertion.cpp** et **fault-handlers--assertion.h**) vont intercepter cette exception.

Le déclenchement d'une interruption non utilisée (c'est-à-dire sans que son *handler* soit spécifié) provoquera l'affichage d'un numéro caractéristique. On retrouvera l'interruption correspondante en examinant le fichier engendrée **zSOURCES/interrupt-handlers.s**.

CENTRALE NANTES

# Assertion

Une assertion est une expression booléenne qui doit être évaluée vraie à l'exécution. Si elle est fausse, un message d'erreur est affiché.

Dans le fichier **fault-handlers--assertion.h**, la fonction **assertion** a pour prototype :

```
void assertion (const bool inAssertion,
                const uint32_t inMessageValue,
                const char * inFileName,
                const int inLine) ;
```

Le deuxième argument **inMessageValue** est simplement une valeur qui sera affichée, et qui *peut aider* à retrouver la cause de l'erreur.

Un exemple d'appel :

```
assertion (expression_booléenne, valeur, __FILE__, __LINE__) ;
```

Les macros `__FILE__` et `__LINE__` contiennent respectivement le nom du fichier source et le numéro de la ligne courante. Ainsi, le message d'erreur indique précisément où l'erreur a eu lieu.

# Travail à faire

Dupliquez le projet de l'étape précédente et renommez-le en par exemple **10-fault-handler--assertion**.

Ajoutez aux sources les fichiers **fault-handlers--assertion.h** et **fault-handlers--assertion.cpp** de l'archive **10-files.tar.bz2**. Remplacer les fichiers **lcd.h**, **lcd.cpp**, **dev-board-io.h** et **dev-board-io.cpp** par ceux de cette archive.

Supprimez des sources les fichiers **lcd-wo-fault-mode.h**, l**cd-wo-fault-mode.cpp** et **unused-interrupt.s**. En effet, jusqu'à l'étape précédente, ce fichier prenait en charge les interruptions inutilisées, ce qui est maintenant fait par le fichier C++ que l'on vient d'ajouter.

Trois programmes différents sont à écrire :

- un programme qui effectue une violation d'une assertion ;
- un programme qui effectue une division entière par zéro ;
- un programme qui déclenche une interruption sans que son handler soit défini.

Chacun de ses programme est décrit dans les pages suivantes.

# Violation d’une assertion

Écrire un programme réalisant la violation d’une assertion ; vérifier que l’affichage nomme le fichier source et la ligne source de cette assertion.

# La division entière par zéro

Écrire un code effectuant une division entière par zéro.

Dans la fonction **configureFaultRegisters** du fichier **fault-handlers--assertion.cpp** que l'on vient d'ajouter, le bit 4 du registre **SCB_CCR** est mis à 1 (il est à zéro par défaut), ce qui a pour effet de déclencher l'exception **HardFault** lors d'une division par zéro.

Il n'est pas simple de retrouver la ligne source où la division par zéro a eu lieu.

L'affichage associé à l'exception ***HardFault*** comprend quatre pages (numéro en haut à droite), que l'on peut parcourir en tournant l'encodeur numérique. Retrouver l'erreur n'est pas toujours simple, pour une vision générale voir le document **apnt209.pdf**. La démarche pour la division entière par zéro est présentée pages suivantes.

# La division entière par zéro : retrouver l'erreur (1/2)

Pour retrouver où la division entière par zéro a été exécutée, procéder comme suit.

① D'abord regarder le registre UFSR (page 2 de l'affichage ***HardFault***) : il a pour valeur 0x0200, qui caractérise la division entière par zéro (voir le document **apnt209.pdf**, page 11) ;

② Ensuite, noter la valeur de PC (page 3 de l'affichage ***HardFault***) : `0x7B2` dans le programme solution, cela est sans doute différent pour vous ; cette valeur est l'adresse de l'instruction fautive ;

③ Il faut maintenant trouver la fonction qui contient cette instruction ; pour cela, ouvrez le fichier texte **zPRODUCTS/product.map** et rechercher la fonction par son adresse :

```
.text.loop.function
                0x000000000000077c       0x84 zBUILDS/setup-loop.cpp.o
                0x000000000000077c                loop.function
.text.start.function
                0x0000000000000800      0x1ec zBUILDS/start-teensy-3-6.cpp.o
```

Ci-dessus, la fonction **loop.function** commence à l'adresse `0x77C` et la suivante commence en `0x800`. L'adresse `0x7B2` est dans cette fonction. L'adresse relative de l'instruction fautive par rapport au début de l'instruction est `0x7B2 - 0x77C = 0x36`.

# La division entière par zéro : retrouver l'erreur (2/2)

④ Exécuter le fichier python **zBUILDS/setup-loop.cpp.objdump.py**. Celui-ci exécute ***objdump***, qui affiche le code binaire désassemblé et le code source :

```
void loop (USER_MODE) {
   0:  b570          push   {r4, r5, r6, lr}
  digitalWrite (L4_LED, !digitalRead (P4_PUSH_BUTTON)) ;
   2:  200c          movs   r0, #12
   ..................................
    printUnsigned (MODE_ millis () / gDownCounter) ;
  2e:  f7ff fffe     bl  0 <_Z6millisv>
  32:  4d13          ldr r5, [pc, #76]; (80 <loop.function+0x80>)
  34:  682b          ldr r3, [r5, #0]
  36:  fbb0 f0f3     udiv   r0, r0, r3
  3a:  f7ff fffe     bl  0 <_Z13printUnsignedm>
    gDownCounter -- ;
  3e:  682b          ldr r3, [r5, #0]
```

L'instruction à l'adresse `0x36` est une division.

CENTRALE NANTES

# Interruption sans *handler* associé

Nous allons écrire un programme qui engendre une interruption quand on appuie sur le CLIC de l'encodeur. Cette entrée est la n°2 (notation Arduino), et pour le micro-contrôleur le port `PTD0` (voir étape 05).

Rappel (étape 05) : la routine **configurePorts** du fichier **dev-board-io.cpp** programme le port n°2 en entrée. Dans la fonction **setup**, il y a deux opérations à faire pour activer l'interruption :

- activer l'interruption sur front descendant du `PTD0` :

```
    PORTD_PCR (0) |= PORT_PCR_IRQC (10) ;
```

- activer l'interruption correspondante (`ISRSlot::PORTD`) sur le processeur :

```
    NVIC_ENABLE_IRQ (ISRSlot::PORTD) ;
```

Dans la suite nous allons voir comment :

- retrouver le nom de l'interruption qui s'est déclenchée ;
- installer la routine d'interruption associée, qui à titre d'exemple, comptera le nombre d'appuis du bouton.

# Interruption sans *handler* associé : l'affichage

Quand on appuie sur `CLIC`, l'interruption `PORTD` est déclenchée, et l'affichage devient :


78 est le numéro du vecteur d'interruption associé à l'interruption `PORTD` : on peut retrouver cette information dans le fichier **teensy-3-6-interrupt-vectors.s** :

```
  .word interrupt.PORTC @ 77
  .word interrupt.PORTD @ 78
  .word interrupt.PORTE @ 79
  .word interrupt.SWINT @ 80
```

# Implémenter la routine d'interruption

Écrire la routine d'interruption :

- ne pas oublier de déclarer cette routine avec l'annotation `//$interrupt-section PORTD` dans le fichier d'en-tête **setup-loop.h** (comme cela a été présenté dans l'étape 07) ;

Note : il faut acquitter l'interruption dans la routine d'interruption :

```
    PORTD_PCR (0) |= PORT_PCR_ISF ;
    PORTD_PCR ;
```

La seconde instruction est indispensable, elle permet d'attendre que l'écriture de la ligne précédente soit réalisée, le retard possible étant dû au *write buffer*.