PLM

Pierre Molinaro

25 mai 2015

Table des matières

Table des matières **2**

Liste des tableaux **5**

Liste des figures **7**

1 Introduction **8**
1.1 Cible 8

2 Le type booléen **9**
2.1 Le type `Bool` 9
2.2 Les mots réservés `true` et `false` 9
2.3 Les opérateurs infix de comparaison 9
2.4 Les opérateurs infixes `and`, `or` et `xor` 9
2.5 L'opérateur préfixé `not` 10
2.6 Conversion en une valeur entière 10
2.7 Conversion d'une valeur entière en booléen 10

3 Les types entiers **11**
3.1 Constante litérale entière 11
3.2 Conversion entre valeurs entières 12
3.2.1 Conversions implicites silencieuses 12
3.2.2 L'opérateur `\` 12
3.2.3 L'opérateur `&\` 13
3.3 Opérateurs infixes et conversions implicites 13
3.4 Opérateurs infix de comparaison 13
3.5 Opérateurs infixes arithmétiques 13
3.6 Opérateurs préfixés de négation arithmétique 14
3.6.1 Opérateur `-` 14
3.6.2 Opérateur `&-` 14
3.7 Opérateurs infixes bit-à-bit 14

3.8 Opérateur préfixé bit-à-bit 15
3.9 Opérateurs infixes de décalage 15

4 Les types flottants 16

5 Déclaration des variables globales 17

6 Les registres de contrôle 18
6.1 Simple déclaration d'un registre 18
6.2 Déclaration d'un registre et de ses champs 19
6.2.1 Accès en lecture aux champs booléens 20
6.2.2 Accès en lecture aux champs entiers 20
6.2.3 Constantes associées aux champs booléens 21
6.2.4 Expressions associées aux champs entiers 21
6.2.5 Masques associés aux champs entiers 22
6.3 Attribut `@ro` 22

7 Directives 23
7.1 Directive `import` 23
7.2 Directive `check` 23

8 Exceptions 24
8.1 L'instruction `assert` 24
8.2 L'instruction `throw` 25
8.3 Définition des types liés aux exceptions 25
8.4 Routines exécutées lors de l'occurrence d'une exception 26
8.4.1 Routines d'exception `setup` et `loop` 26

9 Configuration d'une cible 28

Liste des tableaux

3.1 Types entiers définis par la cible `target-teensy-sequential-systick` . . . 11
3.2 Opérateurs infixes arithmétiques . . . 14
3.3 Opérateurs infixes bit-à-bit sur les entiers non signés . . . 14
3.4 Opérateurs infixes de décalage sur les entiers . . . 15

8.1 Code des exceptions . . . 24

Liste des figures

6.1 Registre de contrôle `ICSR` intégré dans l'ARMv7 19

Chapitre 1

Introduction

1.1 Cible

Chapitre 2

Le type booléen

2.1 Le type `Bool`

L'identificateur `Bool` dénote le type booléen. Sa taille est fixée par la définition de la cible.

2.2 Les mots réservés `true` et `false`

Les mots réservés `true` et `false` dénotent respectivement la valeur logique *vraie* et la valeur logique *fausse*.

2.3 Les opérateurs infix de comparaison

Les valeurs booléennes sont comparables, les six opérateurs `==`, `!=`, `>=`, `>`, `<=` et `<` sont acceptés, avec `false` < `true`.

2.4 Les opérateurs infixes `and`, `or` et `xor`

Les opérateurs infixes `and`, `or` et `xor` implémentent respectivement le *et* logique, *ou* logique, *ou exclusif* logique. Les deux premiers évaluent les opérandes en *court-circuit*, c'est-à-dire que si la valeur de l'opérande de gauche détermine la valeur de l'expression, alors l'opérande de droite n'est pas évalué.

Noter que les opérateurs infixes `&`, `|` et `^` sont des opérateurs bit-à-bit sur les entiers non signés, et ne peuvent pas être appliqués à des valeurs booléennes.

2.5 L'opérateur préfixé `not`

L'opérateur préfixé `not` est la complémentation booléenne. Noter que l'opérateur préfixé `~` effectue la complémentation bit-à-bit d'un entier non signé et ne peut pas être appliqué à une valeur booléenne.

2.6 Conversion en une valeur entière

Lors d'une conversion valeur booléenne vers valeur entière, `false` est converti en la valeur entière 0, et `true` en la valeur entière 1. Comme tous les types entiers peuvent représenter 0 et 1, cette conversion est toujours acceptée silencieusement. Par exemple :

```
let result : UInt8 = true // result a pour valeur 1
```

2.7 Conversion d'une valeur entière en booléen

Il n'y a pas d'opérateur dédié à la conversion d'une valeur entière vers un booléen. Il suffit d'utiliser des opérateurs entre entiers comme `==` ou `!=` pour réaliser une conversion :

```
let result : Bool = x != 0 // x est une expression entière
```

Chapitre 3

Les types entiers

Les types entiers ne sont pas prédéfinis dans le langage. C'est la configuration de la cible qui définit les types entiers disponibles (voir chapitre 9 page 28).

À titre d'exemple, le tableau 3.1 liste les types entiers définis par la cible `target-teensy-sequential-systick`.

Nature	8 bits	16 bits	32 bits	64 bits
Signé	`Int8`	`Int16`	`Int32`	`Int64`
Non signé	`UInt8`	`UInt16`	`UInt32`	`UInt64`

Tableau 3.1 – *Types entiers définis par la cible* `target-teensy-sequential-systick`

3.1 Constante litérale entière

Le langage accepte des constantes litérales non signés de 64 bits. Une constante est convertie dans le type entier requis par le contexte sémantique, et une erreur est déclenchée à la compilation en cas d'impossibilité. Par exemple :

```
var v : Int8 = 128 // Erreur de compilation :
                   // 128 non représentable par un entier signé 8 bits
```

Erreur à corriger

```
var v : Int8 = -128 // Devrait être accepté, erreur actuellement
```

Une constante litérale entière n'a pas de type défini, aussi seul l'inférence de type peut lui affecter un type. Par exemple, si on écrit :

```
var v = 28 // Erreur, le type ne peut pas être inféré
```

Dans ce cas, il faut que la déclaration contienne l'annotation de type :

```
var v : Int32 = 28 // Ok
```

3.2 Conversion entre valeurs entières

3.2.1 Conversions implicites silencieuses

Les conversions qui sont toujours possibles sans débordement sont acceptées silencieusement. Par exemple :

```
let v : UInt8 = ...
let x : UInt16 = v
let y : Int16 = v
```

Par contre, une conversion pouvant provoquer un débordement est rejetée à la compilation :

```
let s : Int8 = ...
let x : UInt16 = x // Erreur de compilation
```

3.2.2 L'opérateur \

L'opérateur `\` permet de spécifier une conversion explicite.

```
let s : Int8 = ...
let x : UInt16 = x \ UInt16
```

L'opérateur `\` engendre un code qui vérifie à l'exécution que l'expression source (ici `x`) peut être convertie dans le type cible (ici `UInt16`) sans débordement. En cas de débordement détecté à l'exécution, une exception dont le code est donné dans le tableau 8.1 page 24 est levée. L'opérateur `\` est donc interdit dans les constructions où les exceptions sont interdites : il faut alors utiliser l'opérateur `&\`.

L'opérateur `\` ne peut pas apparaître dans une expression statique.

De plus, une erreur de compilation est déclenchée si l'opérateur `\` est utilisé alors qu'une conversion implicite est possible :

```
let v : UInt8 = ...
let y : Int16 = v \ Int16 // Erreur, la conversion implicite est possible
```

3.2.3 L'opérateur &\

L'opérateur `&\` permet de spécifier une conversion explicite silencieuse, qui ne lève aucune exception. La valeur de l'expression source est tronquée en cas de débordement[1]. Par exemple :

```
let s : Int8 = -10
let x : UInt16 = x &\ UInt16
```

L'opérateur `&\` ne peut pas apparaître dans une expression statique.

De plus, une erreur de compilation est déclenchée si l'opérateur `&\` est utilisé alors qu'une conversion implicite est possible :

```
let v : UInt8 = ...
let y : Int16 = v &\ Int16 // Erreur, la conversion implicite est pos-
sible
```

3.3 Opérateurs infixes et conversions implicites

3.4 Opérateurs infix de comparaison

Les valeurs entières sont comparables, les six opérateurs `==`, `!=`, `>=`, `>`, `<=` et `<` sont acceptés.

La comparaison ne peut s'effectuer qu'entre valeurs de même type entier.

3.5 Opérateurs infixes arithmétiques

Les opérateurs infixes arithmétiques sont listés dans le tableau 3.2 avec leur signification.

1. L'opérateur &\ est équivalent au *type cast* entre entiers du langage C.

Opérateur	Signification
`+`	Addition avec détection de débordement
`-`	Soustraction avec détection de débordement
`*`	Multiplication avec détection de débordement
`/`	Division avec détection de débordement
`%`	Modulo avec détection de division par zéro
`&+`	Addition sans détection de débordement
`&-`	Soustraction sans détection de débordement
`&*`	Multiplication sans détection de débordement
`&/`	Division sans détection de débordement
`&%`	Modulo sans détection de division par zéro

Tableau 3.2 – *Opérateurs infixes arithmétiques*

3.6 Opérateurs préfixés de négation arithmétique

3.6.1 Opérateur -

L'opérateur préfixé `-` est la négation arithmétique avec détection de débordement. Elle n'est acceptée que sur les types signés. La négation de la borne inférieure d'un type signé (`-128` pour `Int8` , `-32768` pour `Int16` , ...) entraîne un débordement arithmétique qui déclenche une exception dont le code est donné dans le tableau 8.1.

3.6.2 Opérateur &-

L'opérateur préfixé `&-` est la négation arithmétique sans détection de débordement. Elle n'est acceptée que sur les types signés. La négation de la borne inférieure d'un type signé (`-128` pour `Int8` , `-32768` pour `Int16` , ...) retourne cette même valeur.

3.7 Opérateurs infixes bit-à-bit

Les opérateurs infixes bit-à-bit acceptent les types entiers non signés (tableau 3.3).

Opérateur	Signification
`|`	*ou* bit-à-bit
`&`	*et* bit-à-bit
`∧`	*ou exclusif* bit-à-bit

Tableau 3.3 – *Opérateurs infixes bit-à-bit sur les entiers non signés*

3.8 Opérateur préfixé bit-à-bit

L'opérateur préfixé `~` retourne la complémentation bit-à-bit d'une valeur entière non signée.

3.9 Opérateurs infixes de décalage

Les opérateurs infixes `<<` et `>>` réalisent respectivement le décalage à gauche et à droite de l'opérande de gauche. L'amplitude du décalage est spécifiée par la valeur de l'opérande droite (tableau 3.4). `a` est une expression entière signée ou non signée, et l'expression renvoie une valeur de même type que `a`. L'expression `b` est une expression entière non signée.

Expression	Signification
`a << b`	Décalage à gauche de `a` d'une amplitude de `b` bits
`a >> b`	Décalage à droite de `a` d'une amplitude de `b` bits

Tableau 3.4 – *Opérateurs infixes de décalage sur les entiers*

Chapitre 4

Les types flottants

Les types flottants ne sont pas pris en charge dans la version actuelle.

Chapitre 5

Déclaration des variables globales

Chapitre 6

Les registres de contrôle

La déclaration d'un registre de contrôle obéit à une syntaxe particulière, ne serait-ce que parce que son adresse absolue doit y être spécifiée. Pour de nombreux registres, un bit ou un groupe de bits ont une signification particulière, et obtenir la valeur d'un champ ou modifier sa valeur est une opération courante.

À titre d'exemple, nous allons nous intéresser au registre `ICSR` du processeur ARMv7-M. Le *manuel de référence de l'architecture ARMv7-M* [1] décrit ce registre comme indiqué à la figure 6.1, et indique que son adresse est `0xE000ED04`.

6.1 Simple déclaration d'un registre

Pour déclarer le registre `ICSR` (figure 6.1), on écrira simplement :

```
register ICSR at 0xE000_ED04 : UInt32
```

Le type `UInt32` qui est mentionné signifie que les valeurs écrites et lues de ce registre sont des entiers non signés de 32 bits. Tout type entier, signé ou non signé est autorisé.

Pour lire ou écrire ce registre, on le nomme comme s'il s'agissait d'une simple variable. Par exemple, pour activer l'interruption `PendSV`, il faut mettre à 1 le bit `PENDSVSET`. On écrit donc :

```
ICSR = 1 << 28
```

Si on voulait activer simultanément les interruptions `PendSV` et `SysTick`, il faut mettre à 1 les bits `PENDSVSET` et `PENDSTSET`. On écrit donc :

1. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html

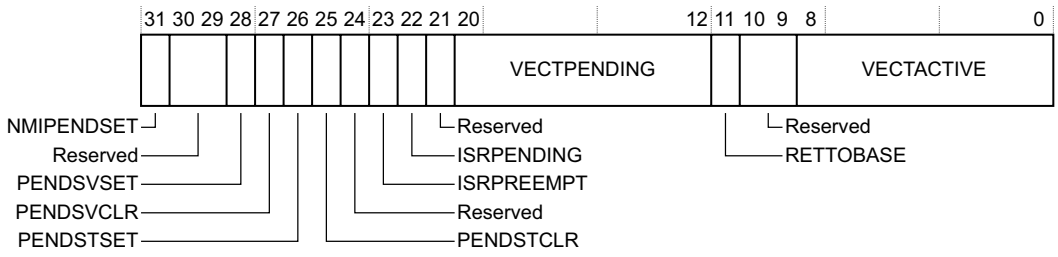

Figure 6.1 – *Registre de contrôle* `ICSR` *intégré dans l'ARMv7*

```
ICSR = (1 << 28) | (1 << 26)
```

Pour savoir si le bit `RETTOBASE` est activé, on écrit :

```
let RETTOBASEActif : Bool = (ICSR & (1 << 11)) != 0
```

Pour accéder à la valeur du champ `VECTPENDING`, on réalise un masquage :

```
let vectPending : UInt32 = ICSR & 0x1F_F000
```

Et si on veut la valeur de champ justifiée à droite :

```
let vectPending : UInt32 = (ICSR & 0x1F_F000) >> 12
```

Ces écritures peuvent être rendues plus intelligibles en précisant la composition du registre `ICSR` dans sa déclaration. C'est ce qui va être réalisé dans la section suivante.

6.2 Déclaration d'un registre et de ses champs

Lors de la déclaration d'un registre, il est possible de préciser la composition de ses champs entiers et booléens :

```
register ICSR at 0xE000_ED04 : UInt32 {
  NMIPENDSET, 2, PENDSVSET, PENDSVCLR, PENDSTSET, PENDSTCLR, 1,
  ISRPREEMPT, ISRPENDING, 1, VECTPENDING[9], RETTOBASE, 2, VECTACTIVE[9]
}
```

Entre accolades, trois définitions différentes peuvent apparaître :

— un nombre indique le nombre de bits consécutifs inutilisés ;

— un identificateur (par exemple `NMIPENDSET`) nomme un champ booléen ;

— un identificateur suivi d'un nombre entre crochets (par exemple `VECTPENDING[9]`) nomme un champ entier constitué du nombre indiqué de bits consécutifs.

La description commence par le bit le plus significatif : comme le type du registre est `UInt32` (entier non signé sur 32 bits), le premier bit nommé `NMIPENDSET` porte le n°31, `PENDSVSET` le n°28, ...

Cette écriture n'est autorisée que si le type nommé (ici `UInt32`) est une type entier non signé. Les types signés (`Int32`, ...) sont interdits. Le compilateur vérifie que la description des champs définit exactement le nombre de bits du type nommé, ici les 32 bits du type `UInt32`.

6.2.1 Accès en lecture aux champs booléens

Pour accéder à la valeur d'un champ, on utilise la notation pointée en nommant ce champ :

```
let x : UInt32 = ICSR.ISRPENDING // 0 ou 2**22
```

Ceci effectue simplement un masquage de façon à isoler le bit demandé. Aucun décalage n'est réalisé. Comme le bit `ISRPENDING` est le 22^{e}, le résultat est 0 ou 2^{22}.

Si l'on veut obtenir un résultat justifié à droite, on utilise en plus l'accesseur `.shift` :

```
let x : UInt32 = ICSR.ISRPENDING.shift // 0 ou 1
```

L'accesseur `.bool` permet d'obtenir une valeur booléenne correspondant à la valeur d'un bit d'un registre :

```
let x : Bool = ICSR.ISRPENDING.bool // false ou true
```

6.2.2 Accès en lecture aux champs entiers

Comme pour un champ booléen, l'accès à un champ entier s'effectue par la notation pointée `.`. Par exemple :

```
let x : UInt32 = ICSR.VECTPENDING
```

`ICSR.VECTPENDING` applique un masquage de la valeur de `ICSR` pour ne conserver que les bits correspondants au champ `VECTPENDING`. Aucun décalage n'est effectué.

Pour obtenir une valeur justifiée à droite, on ajoute l'accesseur `.shift` :

```
let x : UInt32 = ICSR.VECTPENDING.shift
```

La valeur renvoyée est alors comprise en 0 et $2^9 - 1$.

6.2.3 Constantes associées aux champs booléens

Le délimiteur `::` permet de définir des constantes correspondant aux bits d'un registre. Par exemple, pour activer l'interruption PendSV, il faut mettre à 1 le bit PENDSVSET. On écrit donc :

```
ICSR = ICSR::PENDSVSET
```

La constante `ICSR::PENDSVSET` a le type du registre `ICSR`, c'est-à-dire `UInt32`.

De façon analogue, si on voulait activer simultanément les interruptions PendSV et SysTick, il faut mettre à 1 les bits PENDSVSET et PENDSTSET. On écrit donc :

```
ICSR = ICSR::PENDSVSET | ICSR::PENDSTSET
```

6.2.4 Expressions associées aux champs entiers

Le délimiteur `::` permet de simplifier la composition d'une valeur correspondant à un champ entier. Par exemple :

```
let y : UInt32 = ICSR::VECTPENDING (x) // x : expression entière non si-
gnée
```

Le champ VECTPENDING est un champ de 9 bits, il peut donc accepter une valeur entière entre 0 et 511. L'expression `x` doit être de type entier non signé. Plusieurs cas sont à considérer :

- si `x` est une expression statique, alors le compilateur vérifie que sa valeur est comprise entre 0 et 511 (un message d'erreur de compilation est émis dans le cas contraire) ; l'expression `ICSR::VECTPENDING (x)` est alors aussi une expression statique ;
- si `x` est une expression non calculable statiquement :
  - si `x` est du type `UInt8`, alors toute valeur de `x` est acceptable : le code engendré se borne à faire le décalage à gauche de la valeur de `x` ;
  - sinon, une assertion (code : voir tableau 8.1 page 24) est engendrée ; la valeur de `x` est ensuite masquée pour pallier un éventuel débordement, puis décalée.

D'une manière générale, si `x` n'est pas calculable statiquement, le code engendré ne comprendra pas d'assertion si le nombre de bits du champ est supérieur ou égal au nombre de bits du type entier de l'expression.

6.2.5 Masques associés aux champs entiers

Le délimiteur `::` permet de simplifier la composition d'un masque correspondant à un champ entier. Par exemple :

```
let y : UInt32 = ICSR::VECTPENDING // y vaut 0x1F_F000
```

Le champ `VECTPENDING` est un champ de 9 bits commençant au bit 12. La valeur du masque `ICSR::VECTPENDING` est donc $(2^9 - 1)^{12}$, soit `0x1F_F000`.

Les masques ainsi obtenus sont des expressions statiques.

6.3 Attribut `@ro`

La déclaration d'un registre accepte l'attribut `@ro`, qui signifie qu'il est en lecture seule. Par exemple :

```
register SYST_CALIB @ro at 0xE000_E01C : UInt32
```

Toute tentative de faire figurer ce registre dans une construction qui provoque une écriture de celui-ci entraîne l'apparition d'une erreur de compilation.

Chapitre 7

Directives

7.1 Directive `import`

7.2 Directive `check`

La directive `check` apparaît dans une liste d'instructions et a la syntaxe suivante :

```
check expression
```

L'`expression` est une expression booléenne calculée statiquement.

Contrairement à l'instruction `assert` (section 8.1 page 24) qui évalue l'expression booléenne à l'exécution, la directive `check` est toujours évaluée à la compilation. Elle permet d'exprimer des assertions qui sont évaluées lors de la compilation.

Aucun code n'est engendré. La directive `check` peut donc apparaître dans des listes d'instructions où les exceptions sont interdites.

Exemples :

```
check true  // Ok
check false // Erreur, expression fausse
```

Chapitre 8

Exceptions

Numéros	Signification	Lien
< 0	Exceptions liées aux vecteurs d'interruption	
1	Dépassement de capacité de l'incrémentation (`++`)	
2	Dépassement de capacité de l'incrémentation (`--`)	
3	Dépassement de capacité de la négation (`-`)	section 3.6.1 page 14
4	Dépassement de la construction d'un champ entier d'un registre (`registre::champ (...)`)	section 6.2.4 page 21
5	Dépassement de capacité d'une conversion entre entiers (`\`)	
10	Dépassement de capacité de l'addition (`+`)	
11	Dépassement de capacité de la soustraction (`-`)	
12	Dépassement de capacité de la multiplication (`*`)	
13	Dépassement de capacité de la division (`/`)	
14	Modulo par zéro (`%`)	
20	Échec de l'instruction `assert`	section 8.1 page 24

Tableau 8.1 – *Code des exceptions*

8.1 L'instruction `assert`

L'instruction `assert` a la syntaxe suivante :

```
assert expression
```

L'`expression` est une expression booléenne non calculable statiquement.

Si le programme est compilé avec les exceptions activées, alors le compilateur engendre le

code de calcul de l'expression booléenne. Celle-ci sera calculée à l'exécution. Si le résultat est faux, une exception (dont le code est donné par le tableau 8.1) est levée.

Si le programme est compilé avec l'option `--no-exception-generation`, alors aucun code n'est engendré.

Noter que `expression` ne doit pas être calculable statiquement. Si elle est calculable statiquement, il faut utiliser la directive **`check`**, section 7.2 page 23. Par exemple, le code suivant provoque une erreur de compilation :

```
assert true // Erreur, l'expression est calculable statiquement
```

8.2 L'instruction `throw`

L'instruction **`throw`** a la syntaxe suivante :

```
throw expression
```

L'`expression` est une expression entière, calculée statiquement. Son type est défini pour chaque cible (section 8.3 page 25) et peut être signé (les valeurs négatives sont alors acceptées) ou non signé.

Si le programme est compilé avec les exceptions activées, alors l'exécution de l'instruction **`throw`** lève une exception dont le code est la valeur de l'`expression`.

Si le programme est compilé avec l'option `--no-exception-generation`, alors aucun code n'est engendré.

8.3 Définition des types liés aux exceptions

Une exception est caractérisée par trois informations :

- son code (tableau 8.1 page 24) ;
- le nom du fichier source de l'instruction qui a levé l'exception ;
- le numéro de ligne du fichier source de l'instruction qui a levé l'exception.

Le type du code et du numéro de ligne ne sont pas prédéfinis par le langage. La construction suivante définit ces types :

```
exception : nomDuTypeDuCode nomDuTypeDuNumeroDeLigne
```

Par exemple, les numéros de code sont des entiers signés 32 bits, et les numéros de ligne des entiers non signés 32 bits :

```
exception : Int32 UInt32
```

Cette construction doit apparaître exactement une fois. Normalement, c'est le fichier de définition de la cible qui la contient.

8.4 Routines exécutées lors de l'occurrence d'une exception

Lors de l'occurrence d'une exception, l'exécution séquentielle des instructions est abandonnée, et :

- les interruptions sont masquées, si elles ne le sont pas déjà ;
- les routines d'exception `setup` sont exécutées une fois ;
- les routines d'exception `loop` sont exécutées indéfiniment.

Si plusieurs routines d'exception `setup` sont définies, celles-ci sont exécutées dans un ordre quelconque. Les routines d'exception `setup` offrent l'opportunité d'agir sur les sorties du micro-contrôleur, et d'afficher les caractéristiques de l'exception.

Si plusieurs routines d'exception `loop` sont définies, celles-ci sont exécutées dans un ordre quelconque. Les routines d'exception `loop` permettent de signaler d'une manière répétitive l'occurrence d'une exception.

8.4.1 Routines d'exception `setup` et `loop`

Leur syntaxe est la suivante :

```
exception nom {
  liste_instructions
}
```

`nom` est soit `setup`, soit `loop`.

`liste_instructions` est une liste d'instructions qui n'a pas le droit d'engendrer d'exception. Toutes les opérations susceptibles de le faire sont donc interdites, et leur usage est détecté par le compilateur. Par exemple, l'addition `+` est interdite, il faut utiliser `&+` à la place.

Trois constantes sont prédéfinies :

- `CODE`, qui contient le code de l'exception, et dont le type est défini par la construction `exception:` (section 8.3 page 25) ;
- `FILE`, qui contient le nom du fichier source de l'instruction qui a levé l'exception, et dont le type est `StaticString` ;

— `LINE`, qui contient le numéro de ligne du fichier source de l'instruction qui a levé l'exception, et dont le type est défini par la construction `exception:` (section 8.3 page 25).

Les trois constantes `CODE`, `FILE` et `LINE` permettent de signaler les caractéristiques de l'exception.

Actuellement, `FILE` n'est pas exploitable.

Chapitre 9

Configuration d’une cible