PLM

Pierre Molinaro

24 février 2019

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Tutorial

À revoir

PLM est un langage destiné aux systèmes embarqués. Particularités :

- détection des débordements arithmétiques ;
- mode d'exécution d'une routine ;
- routines de démarrage, d'initialisation, de panique ;
- arithmétique sur une taille quelconque ;
- tâches ;
- synchronisations définissables par le langage ;
- commandes gardées définissables par le langage ;

1.1 Premier exemple « blinkled »

1.1.1 Programmes d'exemple embarqués

Le compilateur PLM embarque un certain nombre de fichiers d'exemple, prêts à l'emploi. Deux options (voir section 3.6 page 33) permettent de lister les fichiers d'exemple et de les extraire.

Pour afficher la liste des fichiers d'exemple, entrez :

```
plm -l
Embedded sample code:
  /teensy-3-1-it/01-blink-led.plm
  ...
```

Les exemples sont triés par cible, ainsi, pour la première ligne, la cible est `teensy-3-1-it`, et le fichier d'exemple `01-blink-led.plm`. Les noms des fichiers d'exemple commencent par des chiffres, simplement pour qu'ils apparaissent dans l'ordre souhaité. Vous pouvez nommer vos fichiers PLM comme vous le voulez.

Comme son nom le suggère, l'exemple `01-blink-led.plm` fait clignoter une led, celle embarquée sur la carte *Teensy 3.1*.

1.1.2 Extraction d'un fichier d'exemple

Pour extraire le fichier d'exemple `01-blink-led.plm`, exécuter :

```
plm -x=/teensy-3-1-it/01-blink-led.plm
```

Ceci écrit le fichier `01-blink-led.plm` dans le répertoire courant. Ce fichier est prêt à être compilé.

1.1.3 Compilation et flashage

Pour compiler, entrer[1] :

```
plm --Oz 01-blink-led.plm
Downloading compiler tool chain
............................................................
Making "objects" directory
Compiling src.c
Assembling src.s
LLVM Link sources/src.ll objects/src.c.ll
Optimizing all.ll
Compiling objects/opt.all.ll
Assembling objects/opt.all.ll.s
Stack requirements
Check stacks
Linking product/product-linker.elf
```

1. L'option « --Oz » ordonne une optimisation avec réduction de la taille du code engendré ; pour compiler et flasher, ajouter l'option « -f » (section 3.2 page 32) ; pour compiler avec des messages détaillés, ajouter l'option « -v » (section 3.1 page 31). Les options peuvent apparaître sur la ligne de commande avant ou après le nom du fichier compilé.

```
Hexing product/product.ihex
Code:        4472 bytes
ROM data:    0 bytes
RAM + STACK: 1660 bytes
```

À la première compilation, le compilateur C va être automatiquement téléchargé[2], cela peut prendre plusieurs minutes.

Ensuite, la compilation s'effectue, et range tous ses fichiers de production dans un sous répertoire de `~/plm-products/`, dont le nom est le chemin absolu du fichier source (sans son extension), dans lequel tous les « `/` » ont été remplacés par des « `+` ».

Si l'option « `-f` » était présente, la carte *Teensy 3.1* est prête à être flashée : appuyer sur le bouton *reset* de la carte pour démarrer le flashage[3]. L'affichage sur le terminal devient :

```
Found HalfKay Bootloader
Read "product/product.ihex": 1472 bytes, 1.1% usage
Programming..
Booting
Success
```

Le programme d'exemple a été flashé (« *Programming* »), puis le micro-contrôleur a été redémarré (« *Booting* ») : la led de la carte clignote.

1.1.4 Texte source de `01-blink-led.plm`

Maintenant, nous allons examiner le code source PLM du fichier `01-blink-led.plm` :

```
1  target "teensy-3-1-tp"
2
3  task T1 priority 1 stackSize 512 {
4    var compteur UInt32 = 0
5
6    while time.waitUntilMS (!deadline :self.compteur) {
7      leds.on (!LED_L0)
8      self.compteur +%= 500
9      time.waitUntilMS (!deadline :self.compteur)
10     leds.off (!LED_L0)
11     self.compteur +%= 500
12     lcd.goto (!line :0 !column :0)
```

2. Les outils de compilation C sont installés en `~/plm-tools/`.
3. Si vous ne voulez pas flasher le micro-contrôleur, entrez simplement « `ctrl C` ».

```
13      lcd.printUnsigned (!time.millis ())
14    }
15  }
```

Examinons le code ligne par ligne.

```
1  target "teensy-3-1-tp"
```

La première ligne `target "teensy-3-1-it"` nomme la cible, en référençant un fichier embarqué dans le compilateur. Un fichier de définition de cible définit types, routines, registres de contrôle, ... Un fichier de définition de cible ne peut être compilé séparement, il doit être cité dans un programme utilisateur (ici `01-blink-led.plm`).

```
3  task T1 priority 1 stackSize 512 {
```

Cette ligne déclare une tâche nommée `T1`, de priorité `1`, avec une pile de `512` octets. Ici, le nom de la tâche n'a pas d'importance, ni sa priorité (c'est la seule tâche). Les tâches sont démarrées automatiquement, après la phase d'initialisation.

```
4    var compteur UInt32 = 0
```

Déclaration d'une variable `compteur`, de type `UInt32` (entier non signé sur 32 bits), initialisée à 0. Les variables déclarées dans une tâche sont locales à la tâche, aucune autre tâche ne peut y accéder. PLM prédéfinit les types entiers de taille quelconque (jusqu'à 2048 bits), signés (`IntXX`), et non signés (`UIntXX`). Il est donc parfaitement légal d'utiliser des types comme par exemple `Int8`, `Int13`, ou encore `UInt2000`.

```
6    while time.waitUntilMS (!deadline :self.compteur) {
```

Appel de la commande gardée `waitUntilMS` du pilote `time`, avec un argument, la variable de tâche `compteur`. L'utilisation de `self` est obligatoire pour accéder aux variables de tâche[4]. Cette appel effectue l'attente jusqu'à la date exprimée par l'argument `self.compteur`. Cette date est comptée en millisecondes depuis le démarrage du micro-contrôleur.

```
7      leds.on (!LED_L0)
```

Appel de la fonction `on` du pilote `leds`, avec un argument, la constante `LED_L0`. L'effet de l'exécution de cet appel est d'allumer la led n°0.

```
8      self.compteur +%= 500
```

Incrémente la variable de tâche `compteur` de `500`. L'opérateur `+%=` est l'addition sans test de débordement. Si on voulait le tester, et exécuter un code de panique en cas de débordement, on utiliserait l'opérateur `+=`. Comme `compteur` est utilisé pour l'attente d'échéance qui s'exprimme en millisecondes, `500` représente la date de la prochaine échéance, qui est $500\ ms$ plus tard.

```
9      time.waitUntilMS (!deadline :self.compteur)
```

4. Sans `self`, ce serait un accès à une variable locale à la fonction, ou à une constante globale.

```
10      leds.off (!LED_L0)
11      self.compteur +%= 500
```

Attente de la nouvelle échéance, extinction de la led n°0, incrémentation de l'échéance de $500\ ms$.

```
12      lcd.goto (!line :0 !column :0)
```

Appel de la fonction `goto` du pilote `lcd`, avec deux arguments. L'effet de l'exécution de cet appel est de positionner le curseur de l'afficheur LCD au début de la première ligne.

```
13      lcd.printUnsigned (!time.millis ())
```

Cette ligne affiche sur l'afficheur LCD à la position du curseur le nombre de millisecondes écoulés depuis le démarrage du micro-contrôleur.

```
14    }
```

Cette ligne marque la fin de la commande gardée qui commence à la ligne 6. Comme cette commande gardée commence le mot réservé `while`, elle est re-exécutée.

1.2 Fichiers produits par la compilation

À titre d'information, on décrit l'organisation des fichiers produits par la compilation d'un fichier `fichier.plm`. Cette organisation est résumée par la figure 1.1.

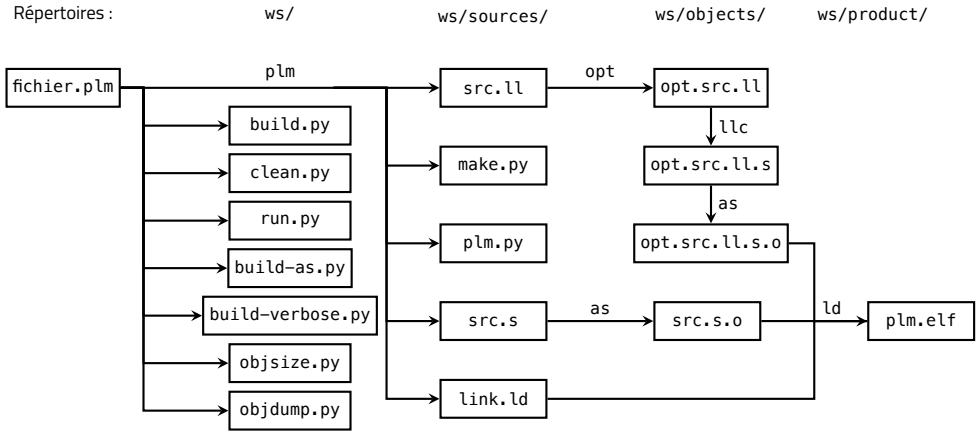


Figure 1.1 – *Fichiers produits par la compilation d'un programme PLM*

Les fichiers produits par la compilation sont regroupés dans un répertoire noté ws dans la figure 1.1, et obtenu de la façon suivante :

- on considère le chemin absolu du fichier source ; par exemple, pour `fichier.plm` situé dans le répertoire courant `/chemin/absolu` : `/chemin/absolu/fichier.plm` ;

Nom	Commentaire
`build.py`	Effectue la compilation de `plm.c`, l'assemblage de `plm.s`, et l'édition de liens afin d'obtenir l'exécutable `plm.elf`, comme décrit à la figure 1.1.
`build-as.py`	Effectue la compilation de `plm.c` jusqu'à la génération d'un fichier assembleur `plm.c.s` qui est placé dans `ws/objects`. Cette opération permet de connaître la traduction en assembleur du source C.
`build-verbose.py`	Effectue les mêmes opérations que `build.py`, mais affiche les lignes de commande.
`run.py`	Construit l'exécutable `plm.elf` (comme le fait `build.py`), et lance l'exécution sur la cible. Équivalent à l'option `-f` de la ligne de commande.
`clean.py`	Supprime les répertoires `ws/objects` et `ws/product`.
`objdump.py`	Effectue les mêmes opérations que `build.py`, puis affiche le code machine obtenu désassemblé.
`objsize.py`	Effectue les mêmes opérations que `build.py`, puis affiche la taille du code engendré.

Tableau 1.1 – *Scripts Python engendrés dans le répertoire* `ws`

Nom	Commentaire
`src.ll`	Contient tout le code LLVM produit par la compilation du fichier source `fichier.plm`.
`src.s`	Contient tout le code assembleur produit par la compilation du fichier source `fichier.plm`. En fonction de la cible, ce fichier peut être vide.
`make.py`	Makefile générique écrit en Python.
`plm.py`	Configuration du makefile générique pour la compilation PLM.
`link.ld`	Script de l'édition de liens.

Tableau 1.2 – *Fichiers engendrés dans le répertoire* `ws/sources`

- on retire l'extension du fichier source ; `/chemin/absolu/fichier` ;
- on remplace chaque « `/` » par un « `+` » : `+chemin+absolu+fichier` ;
- le chemin vers le répertoire `ws` est obtenu en préfixant par `~/plm-product/`, où ~ est le répertoire *home* de l'utilisateur : `~/plm-product/+chemin+absolu+fichier`.

La compilation place dans le répertoire `ws` les scripts *python* décrits par le tableau 1.1. Ces scripts peuvent être appelés directement à partir de la ligne de commande.

Le répertoire `ws/sources` contient les fichiers décrits dans le tableau 1.2, `ws/objects` contient les fichiers objets issus de la compilation et l'assemblage, et `ws/product` le fichier exécutable nommé `plm.elf`. Suivant la cible, il peut aussi contenir l'exécutable `plm.ihex` sous le format *hex* d'Intel.

Chapitre 2

Cible `teensy-3-1-it`

Dans l'état actuel de PLM, une seule cible est définie : `teensy-3-1-it`. Elle permet une programmation séquentielle avec routines d'interruption. L'interruption `systick` est programmée pour se déclencher chaque milliseconde. L'objet de ce chapitre est de décrire son utilisation.

Il est possible de définir sa propre cible (chapitre 28 page 142).

2.1 Organigramme d'exécution

La figure 2.1 définit l'organigramme d'exécution d'un programme.

Le micro-contrôleur démarre sur une horloge interne, la mémoire vive n'étant pas initialisée. Il est dans le mode *thread, priviliged access*, avec une seule pile. La configuration conservera cette pile unique, jusqu'au démarrage des tâches.

La première étape est de configurer les horloges internes du micro-contrôleur : c'est le rôle des routines `boot` (section 2.2 page 21). À ce stade, la mémoire vive n'est toujours pas initialisée, aussi les routines `boot` n'y accèdent pas (le compilateur l'assure).

La deuxième étape est d'initialiser les *variables globales*, c'est-à-dire mettre à zéro la zone `bss`, et de recopier à partir de la flash les valeurs initiales des variables initialisées.

La troisième étape est l'exécution des routines `init` (section 2.3 page 21). À partir de cette étape et pour les suivantes, les variables globales sont initialisées, et donc leur emploi est autorisé. Le rôle des routines `init` est de configurer les entrées/sorties du micro-contrôleur.

Ensuite, les tâches sont lancées, et exécutées en fonction de leurs priorités et synchronisations.

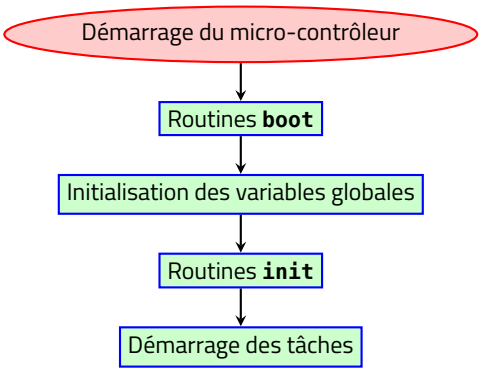


Figure 2.1 – *Organigramme d'exécution de la cible* `teensy-3-1-it`

2.2 Personalisation du démarrage

La cible définit la routine `boot 0` qui configure le micro-contrôleur.

Vous pouvez ajouter vos propres routines `boot` . À chaque routine `boot` est associée une priorité d'exécution, qui doit être unique. Les routines `boot` sont exécutées dans l'ordre croissant des priorités, c'est-à-dire que la routine `boot 0` est exécutée la première.

2.3 Personalisation de l'initialisation

La cible définit la routine `init 0` qui configure le *SysTick Timer* pour qu'il engendre une interruption toutes les millisecondes.

Vous pouvez ajouter vos propres routines `init` . À chaque routine `init` est associée une priorité d'exécution, qui doit être unique. Les routines `init` sont exécutées dans l'ordre croissant des priorités, c'est-à-dire que la routine `init 0` est exécutée la première.

2.4 API

L'API de la cible `teensy-3-1-it` regroupe les fonctions disponibles dans des pilotes et des types :

- le pilote `time` , la gestion du temps (section 2.4.1 page 22) ;
- le pilote `leds` , la gestion des leds (section 2.4.2 page 23) ;
- le pilote `lcd` , la gestion de l'afficheur LCD (section 2.4.3 page 24) ;
- le type `Semaphore` , le sémaphore de Dijkstra (section 2.4.4 page 26) ;

2.4.1 Pilote `time`

Le pilote `time` regroupe des fonctions dédiées à la gestion du temps. Pour la cible `teensy-3-1-it`, le temps est compté en nombre de milli-secondes écoulées depuis le démarrage du micro-contrôleur.

2.4.1.1 Fonction `oneMillisecondBusyWait`

```
public func panic init oneMillisecondBusyWait ()
```

Cette fonction réalise une attente active jusqu'à la prochaine milli-seconde. Elle n'est appelable que dans les modes `panic` et `init`.

2.4.1.2 Fonction `busyWaitingDuringMS`

```
public func panic init busyWaitingDuringMS (?inDelay UInt32)
```

Cette fonction effectue une attente active en appelant `inDelay` fois la fonction `oneMillisecondBusyWait`. Elle n'est appelable que dans les modes `panic` et `init`.

2.4.1.3 Primitive `waitUntilMS`

```
public primitive waitUntilMS (?deadline : inDate UInt32)
```

Cette fonction réalise une attente passive jusqu'à la date absolue `inDate`.

2.4.1.4 Primitive `waitDuringMS`

```
public primitive waitDuringMS (?delay : inDelay UInt32)
```

Cette fonction réalise une attente passive pendant `inDelay` milli-secondes.

2.4.1.5 Garde `waitUntilMS`

```
public guard waitUntilMS (?deadline :inDeadline UInt32)
```

Cette fonction exprime l'attente passive en garde pendant `inDelay` milli-secondes.

2.4.2 Pilote `leds`

2.4.2.1 Constantes

Les constantes suivantes sont de type `UInt32` et sont dédiées aux leds : `LED_L0`, `LED_L1`, `LED_L2`, `LED_L3` et `LED_L4`.

2.4.2.2 Fonction `write(?off :)`

```
public func user panic service write (?off :inLeds UInt32)
```

Cette routine éteint un ensemble de leds. Pour éteindre une led, écrire :

```
leds.write (!off :LED_L0)
```

Pour éteindre plusieurs leds, utiliser l'opérateur `|` :

```
leds.write (!off :LED_L0 | LED_L4)
```

2.4.2.3 Fonction `write(?on :)`

```
public func user panic service write (?on :inLeds UInt32)
```

Cette routine allume un ensemble de leds. Pour allumer une led, écrire :

```
leds.write (!on :LED_L0)
```

Pour allumer plusieurs leds, utiliser l'opérateur `|` :

```
leds.write (!on :LED_L0 | LED_L4)
```

2.4.2.4 Fonction `write(?toggle :)`

```
public func user panic service write (?toggle :inLeds UInt32)
```

Cette routine complémente un ensemble de leds.

Pour complémenter une led, écrire :

```
leds.write (!toggle :LED_L0)
```

Pour complémenter plusieurs leds, utiliser l'opérateur `|` :

```
leds.write (!toggle :LED_L0 | LED_L4)
```

2.4.3 Pilote `lcd`

2.4.3.1 Fonction `clearScreen`

```
public func user clearScreen ()
```

Cette fonction efface l'afficheur LCD, et place le curseur au début de la première ligne.

2.4.3.2 Fonction `goto`

```
public func user goto (?line :inLine UInt2
                       ?column :inColumn UInt8)
```

Cette fonction place le curseur à la colonne `inColumn` de la ligne `inLine`. L'afficheur possédant quatre lignes, l'argument `inLine` est de type `UInt2`.

2.4.3.3 Fonction `printSpaces`

```
public func user printSpaces (?inCount UInt32)
```

Cette fonction écrit `inCount` caractères espace à partir de la position du curseur.

2.4.3.4 Fonction `printUnsigned`

```
public func user printUnsigned (?inValue UInt32)
```

Cette fonction écrit la valeur de l'argument (un entier non signé sur 32 bits) `inValue` à partir de la position du curseur.

2.4.3.5 Fonction `printSigned`

```
public func user printSigned (?inValue Int32)
```

Cette fonction écrit la valeur de l'argument (un entier signé sur 32 bits) `inValue` à partir de la position du curseur.

2.4.3.6 Fonction `printString`

```
public func user printString (?inValue LiteralString)
```

Cette fonction écrit la valeur de l'argument (une chaîne de caractères) `inValue` à partir de la position du curseur.

2.4.3.7 Fonction `clearScreenInPanicMode`

```
public func panic clearScreenInPanicMode ()
```

Cette fonction efface l'afficheur LCD, et place le curseur au début de la première ligne. Appelable uniquement en mode `panic`.

2.4.3.8 Fonction `gotoInPanicMode`

```
public func panic gotoInPanicMode (?line :inLine UInt2
                                   ?column :inColumn UInt8)
```

Cette fonction place le curseur à la colonne `inColumn` de la ligne `inLine`. L'afficheur possédant quatre lignes, l'argument `inLine` est de type `UInt2`. Appelable uniquement en mode `panic`.

2.4.3.9 Fonction `printSpacesInPanicMode`

```
public func panic printSpacesInPanicMode (?inCount UInt32)
```

Cette fonction écrit `inCount` caractères espace à partir de la position du curseur. Appelable uniquement en mode `panic`.

2.4.3.10 Fonction `printUnsignedInPanicMode`

```
public func panic printUnsignedInPanicMode (?inValue UInt32)
```

Cette fonction écrit la valeur de l'argument (un entier non signé sur 32 bits) `inValue` à partir de la position du curseur. Appelable uniquement en mode `panic`.

2.4.3.11 Fonction `printSignedInPanicMode`

```
public func panic printSignedInPanicMode (?inValue Int32)
```

Cette fonction écrit la valeur de l'argument (un entier signé sur 32 bits) `inValue` à partir de la position du curseur. Appelable uniquement en mode `panic`.

2.4.3.12 Fonction `printStringInPanicMode`

```
public func panic printStringInPanicMode (?inValue LiteralString)
```

Cette fonction écrit la valeur de l'argument (une chaîne de caractères) `inValue` à partir de la position du curseur. Appelable uniquement en mode `panic`.

2.4.4 Type `Semaphore`

Le type `Semaphore` implémente le sémaphore de Dijstra. La liste des tâches bloquées est ordonnée selon leur priorité.

La déclaration d'un sémaphore mentionne sa valeur initiale, qui est un entier non signé de 32 bits :

```
var s = Semaphore (!value :0)
```

2.4.4.1 Service `signal()`

```
public service signal ()
```

Ce service effectue l'opération `signal` sur le sémaphore sur lequel il s'applique. C'est un service, car il est appelable à partir d'une tâche, d'une primitive, et d'une routine d'interruption.

2.4.4.2 Primitive `wait()`

```
public primitive wait ()
```

Cette primitive effectue l'opération `wait` sur le sémaphore sur lequel il s'applique. C'est une primitive, car elle n'est pas appelable à partir d'une routine d'interruption.

2.4.4.3 Garde `wait()`

```
public guard wait ()
```

Cette primitive exprime l'opération `wait` en garde.

2.5 Les routines d'interruption

Toutes les interruptions du micro-contrôleur sont accessibles en PLM, avec quelques exceptions :

Numéro	Nom routine	Numéro	Nom routine
1	*Reset, réservé par PLM*	16	`DMAChannel0TranfertComplete`
2	`NMI`	17	`DMAChannel1TranfertComplete`
3	`HardFault`, réservé par PLM	18	`DMAChannel2TranfertComplete`
4	`MemManage`	19	`DMAChannel3TranfertComplete`
5	`BusFault`	20	`DMAChannel4TranfertComplete`
6	`UsageFault`	21	`DMAChannel5TranfertComplete`
7 à 10	*réservées par ARM*	22	`DMAChannel6TranfertComplete`
11	`svc`, *réservé par PLM*	23	`DMAChannel7TranfertComplete`
12	`DebugMonitor`	24	`DMAChannel8TranfertComplete`
13	*réservée par ARM*	25	`DMAChannel9TranfertComplete`
14	`PendSV`	26	`DMAChannel10TranfertComplete`
15	`Systick`, voir section 2.5.4 page 29	27	`DMAChannel11TranfertComplete`

Tableau 2.1 – *Table des interruptions 1 à 27 de la cible* `teensy-3-1-it`

- l'interruption n°1 (*Reset*), est réservée par PLM ;
- l'interruption n°3 (*HardFault*), est réservée par PLM ;
- l'interruption n°11 (*svc*), est réservée par PLM ;
- l'interruption n°15 (*systick*), est implémentée par la cible ; toutefois, une fonction `userSystickHandler` est disponible, voir section 2.5.4 page 29.

Une routine d'interruption porte un des noms définis par la cible :

- interruptions 1 à 27 : tableau 2.1 page 27 ;
- interruptions 28 à 90 : tableau 2.2 page 28 ;
- interruptions 91 à 110 : tableau 2.3 page 28.

2.5.1 Définir une routine d'interruption

Prenons l'exemple d'une interruption que l'on appelera `MyIT`. On déclare la routine d'interruption avec l'un des trois qualificatifs suivants :

- **`section`** , section 2.5.1.1 ;
- **`service`** , section 2.5.1.2.

2.5.1.1 Routine d'interruption en mode `section`

Numéro	Nom routine	Numéro	Nom routine
28	DMAChannel12TranfertComplete	60	UART0LON
29	DMAChannel13TranfertComplete	61	UART0Status
30	DMAChannel14TranfertComplete	62	UART0Error
31	DMAChannel15TranfertComplete	63	UART1Status
32	DMAError	64	UART1Error
33	*inutilisée*	65	UART2Status
34	flashMemoryCommandComplete	66	UART2Error
35	flashMemoryReadCollision	67 à 72	*inutilisées*
36	modeController	73	ADC0
37	LLWU	74	ADC1
38	WDOGEWM	75	CMP0
39	*inutilisée*	76	CMP1
40	I2C0	77	CMP2
41	I2C1	78	FMT0
42	SPI0	79	FMT1
43	SPI1	80	FMT2
44	*inutilisée*	81	CMT
45	CAN0MessageBuffer	82	RTCAlarm
46	CAN0BusOff	83	RTCSecond
47	CAN0Error	84	PITChannel0
48	CAN0TransmitWarning	85	PITChannel1
49	CAN0ReceiveWarning	86	PITChannel2
50	CAN0WakeUp	87	PITChannel3
51	I2S0Transmit	88	PDB
52	I2S0Receive	89	USBOTG
53 à 59	*inutilisées*	90	USBChargerDetect

Tableau 2.2 – *Table des interruptions 28 à 90 de la cible* `teensy–3–1–it`

Numéro	Nom routine	Numéro	Nom routine
91 à 96	*inutilisées*	104	pinDetectPortB
97	DAC0	105	pinDetectPortC
98	*inutilisée*	106	pinDetectPortD
99	TSI	107	pinDetectPortE
100	MCG	108 et 109	*inutilisées*
101	lowPowerTimer	110	softwareInterrupt
103	pinDetectPortA		

Tableau 2.3 – *Table des interruptions 91 à 110 de la cible* `teensy–3–1–it`

```
isr section MyIT {
  ...
}
```

La routine d'interruption s'exécute en mode `section`, c'est-à-dire que les routines de l'exécutif lui sont inaccessibles.

2.5.1.2 Routine d'interruption en mode `service`

```
isr service MyIT {
  ...
}
```

La routine d'interruption s'exécute en mode `service`, les routines correspondantes de l'exécutif sont accessibles.

2.5.2 Routines d'interruption par défaut, panique activée

Quand la panique est activée, une routine par défaut est prédéfinie pour chaque interruption (sauf pour celles réservées par ARM et par PLM). Celle-ci exécute l'instruction `panic`, dont l'argument est l'opposé du numéro de l'interruption. Par exemple, pour l'interruption 84 (`PITChannel0`), la routine prédéfinie est :

```
isr section PITChannel0 {
  panic -84
}
```

2.5.3 Routines d'interruption par défaut, panique inactivée

Quand la panique est inactivée, aucune routine d'interruption pas défaut n'est engendrée ; dans la table des vecteurs d'interruption, l'entrée d'une routine d'interruption indéfinie contient une valeur par défaut qui dépend de la cible.

2.5.4 Routine associée à l'interruption `SysTick`

L'interruption `SysTick` est particulière. Elle est programmé par la routine `init 0` de façon à engendrer une interruption chaque milliseconde. Cette interruption se déclenche après la fin de l'exécution la routine `init 0`, y compris éventuellement dans les routines `init` qui suivent. La routine d'interruption associée, inaccessible à l'utilisateur :

- incrémente une variable globale de comptage du temps ;
- appelle la routine `makeTasksReadyFrom` qui contrôle les tâches bloquées en attente d'échéance ;
- appelle la routine `tickHandlerForGuardedWaitUntil` qui contrôle les tâches en attente d'échéance en garde ;
- appelle la routine `userSystickHandler`.

Cette routine `userSystickHandler` est définie par [À REVOIR] :

```
public func service userTickHandler () {
}
```

Chapitre 3

Options de la ligne de commande

Les options de la ligne de commande commencent toutes par un caractère « - ». La forme courte débute par un simple « - », la forme longue par un double « - ». Certaines options acceptent au choix les deux formes (par exemple --`verbose` ou -v). Une option ne commençant par un « - » est considérée comme un nom de fichier source. Les deux seules extensions autorisées pour les fichiers sources sont « `.plm` » (fichier source PLM) et « `.plm-target` » (fichier de description d'une cible). Par exemple, pour compiler un fichier source `source.plm` avec l'option *verbose* :

```
plm -v source.plm
```

Les options peuvent apparaître avant ou après le nom du fichier source, elles sont toujours examinées avant que la compilation ne commence.

3.1 Options générales

`--help` Affiche l'aide.

`--version` Affiche la version du compilateur PLM.

`--log-file-read` Affiche l'accès en lecture à tout fichier.

`--max-errors=`n Arrête la compilation si le nombre de n erreurs est atteint. Par défaut, la borne est égale à 100.

`--max-warnings=`n Arrête la compilation si le nombre de n alertes est atteint. Par défaut, la borne est égale à 100.

`--Werror` Transforme toute alerte en erreur.

`--verbose`, **`-v`** Affiche des messages indiquant la progression de la compilation.

`--Werror` Considère tout *warning* comme une erreur.

`--no-color` Les textes affichés sur le terminal sont sans l'enrichissement en couleur.

3.2 Options affectant le code engendré

`--no-panic-generation` Inhibe la génération du code de la panique organisée.

`--no-file-generation` Inhibe l'écriture de fichiers par le compilateur.

`--O1` Premier niveau d'optimisation (correspond à l'option –O1 de LLVM).

`--O2` Premier niveau d'optimisation (correspond à l'option –O2 de LLVM).

`--O3` Premier niveau d'optimisation (correspond à l'option –O3 de LLVM).

`--Os` Comme l'option ––O2, mais avec des optimisations supplémentaires pour réduire la taille (correspond à l'option –Os de LLVM).

`--Oz` Comme l'option ––Os, mais avec encore d'autres optimisations pour réduire la taille (correspond à l'option –Oz de LLVM).

3.3 Options de débogage

`--output-concrete-syntax-tree` Engendre un fichier au format `dot` contenant l'arbre syntaxique concret du texte source.

`--routine-invocation-graph`, **`-i`** Engendre un fichier au format `dot` pouvant être ouvert par `graphviz` contenant le graphe d'invocation des routines.

`--do-not-detect-recursive-calls`, **`-r`** N'effectue pas la détection des routines récursives. Par défaut le compilateur affiche un message d'erreur si il trouve des routines récursives.

`--display-deadcode-elimination`, **`-z`** Affiche sur le terminal les détails d'élimination du code mort.

`--control-register-map` Écrit dans un fichier HTML la tables des registres de contrôle.

`--global-constant-dependency-graph`, **`-c`** Écrit dans un fichier au format `dot` pouvant être ouvert par `graphviz` le graphe de dépendance des constantes statiques.

`--routine-invocation-graph`, **`-i`** Écrit dans un fichier au format `dot` pouvant être ouvert par `graphviz` le graphe d'invocation des routines.

`--type-dependency-graph`, **`-t`** Écrit dans un fichier au format `dot` pouvant être ouvert par `graphviz` le graphe de dépendance des types.

`--mode=`$string$, où $string$ est l'une des chaînes suivantes : `lexical-only`,`syntax-only`,`latex`.

`--output-keyword-list-file=`$string$, où $string$ est une chaîne décrivant le formatage de la sortie.

3.4 Option de flashage du code engendré

`--flash-target` , **`-f`** Après une compilation sans erreur, effectue le flashage du micro-contrôleur cible.

3.5 Options de débogage du compilateur

`--output-concrete-syntax-tree` Engendre un fichier au format `dot` pouvant être ouvert par `graphviz` contenant l'arbre syntaxique concret du texte source.

`--mode=`nom , où *nom* peut prendre pour valeur :

- « *vide* » : fonctionnement nominal, le compilateur effectue toutes les phases : analyse lexicale, analyse syntaxique, analyse sémantique, et génération de code ;
- `lexical-only` : le compilateur s'arrête après l'analyse lexicale, et affiche la séquence des symboles terminaux obtenue ;
- `syntax-only` : le compilateur s'arrête après l'analyse syntaxique, et affiche l'arbre de dérivation.
- `latex` : le compilateur s'arrête après l'analyse lexicale, et engendre un fichier latex contenant le texte source.

Écrire l'option –-`mode=` est équivalent à l'absence de cette option.

3.6 Options d'accès aux fichiers d'exemple embarqués

`--list-embedded-samples` , **`-l`** Affiche la liste des fichiers d'exemple embarqués dans le compilateur.

`--extract-embedded-sample-code=`nom , **`-x=`**nom Extrait le fichier d'exemple nom et l'écrit dans le répertoire courant.

L'utilisation de ces deux options est illustrée à la section 1.1 page 14.

3.7 Options d'accès aux cibles embarquées

`--use-target-dir=`*repertoire* , **`-T=`***repertoire* N'utilise pas les cibles embarquées, au profit des cibles définies dans le répertoire *repertoire*.

`--list-targets` , **`-L`** Affiche la liste des cibles disponibles, soit celles embarquées dans le compilateur, soit, si l'option précédente est présente, celles définies dans le répertoire *repertoire*.

`--extract-embedded-targets=`*repertoire* , **`-X=`***repertoire* Extrait les fichiers de définition des cibles embarquées dans le compilateur et les écrit dans le répertoire *repertoire*.

L'utilisation de ces options est illustrée à la section 28.2 page 145.

Chapitre 4

Élements lexicaux

4.1 Identificateurs

Un identificateur commence par une lettre Unicode, qui est suivie par zéro, un ou plusieurs lettres Unicode, chiffres décimaux, caractères « _ ».

La casse est significative pour les identificateurs. Comme toute lettre Unicode est acceptée, les lettres accentuées, les lettres grecques, cyrilliques, ... sont autorisées :

```
let entréeValidée Bool = yes
let α UInt4 = 2
let постоянная UInt8 = 200
```

4.2 Mots réservés

Les mots réservés correspondant aux éléments du langage sont listés dans le tableau 4.1.

4.3 Constante entière

Vous pouvez écrire les constantes entières en décimal, en hexadécimal ou en binaire. Une constante entière de type `LiteralInt` ; ce type est décrit au chapitre 7 page 43.

Décimal. Une constante entière décimale commence par un chiffre décimal, et est suivie par zéro, un ou plusieurs chiffres décimaux, ou caractères « _ ».

Contrairement au C, un nombre qui commence par un zéro est un nombre écrit en décimal.

`addressof`	`and`	`assert`	`boot`	`case`
`check`	`convert`	`driver`	`else`	`enum`
`event`	`exit`	`extend`	`extern`	`for`
`func`	`guard`	`if`	`import`	`in`
`interrupt`	`let`	`no`	`not`	`opaque`
`option`	`or`	`panic`	`primitive`	`public`
`registers`	`required`	`safe`	`section`	`self`
`service`	`sizeof`	`startup`	`staticArray`	`struct`
`switch`	`sync`	`target`	`task`	`truncate`
`typealias`	`user`	`var`	`while`	`xor`
`yes`				

Tableau 4.1 – *Mots réservés du langage PLM*

Hexadécimal. Un chiffre hexadécimal est soit un chiffre décimal, soit une lettre entre « `a` » et « `f` », écrite indifféremment en minuscule ou en majuscule. Une constante hexadécimale commence par la séquence « `0x` », et est suivie par un ou plusieurs chiffres hexadécimal, ou caractères « `_` ».

Binaire. Une constante entière binaire commence par la séquence « `0b` » suivie par un ou plusieurs chiffres binaires, `0` ou `1`, ou caractères « `_` ».

Le caractère « `_` ». Dans une constante entière, écrite en binaire, décimal ou en hexadécimal, le caractère « `_` » peut servir de séparateur ; on peut ainsi écrire indifféremment : `123`, `1_23`, `1_2_3`, `1___23`, `1___2__3__`, …

Taille. Une constante entière n'est pas limitée en taille : vous pouvez écrire des constantes entières avec un nombre quelconque de chiffres :

```
let a = 123_456_789_123_456_789_123_456_789_123_456_789_123_456_789
```

4.4 Délimiteurs

PLM définit les délimiteurs listés dans le tableau 4.2.

`!%`	`!%=`	`!/`	`!/=`	`•ERRLEX•$(`	`•ERRLEX•${`	`%`	`%=`	`&`	`&=`	`(`	`)`
`*`	`*%`	`*%=`	`*=`	`+`	`+%`	`+%=`	`+=`	`,`	`-`	`-%`	`-%`
`-=`	`->`	`.`	`...`	`..<`	`/`	`/=`	`:`	`: :`	`<`	`<<`	`<<`
`=`	`==`	`>`	`>>`	`>>=`	`[`	`]`	`]!`	`^`	`^=`	`_`	`{`
`|`	`|=`	`}`	`~`	`≠`	`≤`	`≥`					

Tableau 4.2 – *Délimiteurs du langage PLM*

4.5 Attributs

Un attribut est une séquence commençant par un `@` et suivi d'un ou plusieurs chiffres, lettres. Voici quelques attributs valides : `@toto`, `@truc1`, `@123`. Un attribut sert à différents usages comme par exemple (liste non exhautive) :

- `@ro` indique qu'un registre de contrôle est accessible en lecture seule (section 17.6.1 page 87) ;
- ajouter une étiquette à une instruction `if` (section 21.11 page 110), pour en augmenter la lisibilité.

4.6 Sélecteurs

Un sélecteur spécifie le mode de passage d'un argument formel et d'un paramètre effectif. Ils se présentent sous plusieurs formes :

- une forme anonyme : `?`, `!`, `?!`, `!?` ;
- `?selecteur :`, `!selecteur :`, `?!selecteur :`, `!?selecteur :`, où $selecteur$ est une séquence de lettres ou de chiffres.

Les sélecteurs font l'objet de la section 19.2 page 94.

4.7 Séparateurs

Tout caractère dont le code ASCII est compris entre `0x01` et `0x20` est considéré comme un séparateur (ceci inclut donc la tabulation horizontale HT (`0x09`), le passage à la ligne LF (`0x0A`), le retour-chariot CR (`0x0D`), et l'espace (`0x20`).

4.8 Commentaires

Un commentaire commence par deux barres obliques consécutives `//`, et s'étend jusqu'à la fin de la ligne courante.

4.9 Format

Le format est libre, la fin de ligne n'est pas significative (sauf pour les commentaires, qui commencent par deux barres obliques consécutives `//`, et s'étendent jusqu'à la fin de la ligne courante). Le compilateur accepte de manière indifférente que les fins de ligne soient codés par un caractère LF (`0x0A`), un caractère CR (`0x0D`), ou par la séquence CRLF (`0x0D`, `0x0A`).

Chapitre 5

Démarrage du micro-contrôleur

5.1 Séquence de démarrage

La séquence de démarrage du micro-contrôleur est illustrée par la figure 5.1.

La première étape est de configurer les horloges internes du micro-contrôleur : c'est le rôle des routines `boot`. À ce stade, la mémoire vive n'est toujours pas initialisée, aussi les routines `boot` n'y accèdent pas (le compilateur l'assure).

La deuxième étape est d'initialiser les *variables globales*, c'est-à-dire mettre à zéro la zone « `.bss.*` », et de recopier à partir de la flash les valeurs initiales des variables initialisées.

La troisième étape est l'exécution des routines `init`. À partir de cette étape et pour les suivantes, les variables globales sont initialisées, et donc leur emploi est autorisé. Le rôle des routines `init` est de configurer les entrées/sorties du micro-contrôleur.

Ensuite, les tâches sont lancées, et exécutées en fonction de leurs priorités et synchronisations.

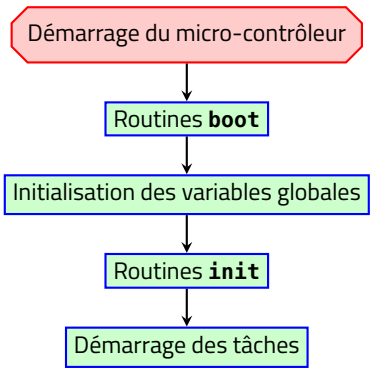


Figure 5.1 – *Organigramme d'exécution de la séquence de démarrage*

5.2 `boot` routines

Une routine `boot` est exécutée une et une seule fois, lors du démarrage du micro-contrôleur, avant que les variables globales ne soient initialisées. Elle a la syntaxe suivante :

```
boot priorité {
  liste_instructions
}
```

Où `priorité` est la priorité de la routine. C'est une constante entière statique. Les routines `boot` sont exécutées dans l'ordre des priorités croissantes. Le compilateur vérifie que deux routines `boot` n'ont pas la même priorité.

Les routines `boot` s'exécutent dans le mode `boot`.

Par contre, l'accès aux registres de contrôle est autorisé dans une routine `boot` (d'ailleurs, elle sert à cela : configurer le micro-contrôleur au démarrage).

5.3 `init` routines

Une routine `init` est exécutée une et une seule fois, lors du démarrage du micro-contrôleur, après l'initialisation des variables globales. Elle a la syntaxe suivante :

```
init priorité {
  liste_instructions
}
```

Où `priorité` est la priorité de la routine. C'est une constante entière statique. Les routines `init` sont exécutées dans l'ordre des priorités croissantes. Le compilateur vérifie que deux routines `init` n'ont pas la même priorité.

Les routines `init` s'exécutent dans le mode `init`.

Chapitre 6

Le type booléen

Le type booléen `Bool` est prédéfini par le langage.

6.1 Les mots réservés `yes` et `no`

Les mots réservés `yes` et `no` dénotent respectivement la valeur logique *vraie* et la valeur logique *fausse*.

6.2 Les opérateurs infix de comparaison

Les valeurs booléennes sont comparables, les six opérateurs `==`, `≠`, `≥`, `>`, `≤` et `<` sont acceptés, avec `no < yes`.

6.3 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.

6.4 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.

6.5 Conversion en une valeur entière

La conversion d'une valeur booléenne en une valeur entière s'effectue par l'intermédiaire d'une expression `if`. Par exemple :

```
let x Bool = ...
let result UInt8 = if x { 4 }else{ 2 }
```

6.6 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 7

Le type *entier statique*

Ce chapitre est consacré au type *entier statique* `LiteralInt`, qui est le type de toute constante entière littérale. Mais ce type est aussi applicable à des constantes *entières statiques*, c'est-à-dire dont la valeur est calculée à la compilation.

Le type `LiteralInt` n'est pas applicable à une variable : une variable entière doit être déclarée du type `UIntN` (entier non signé de N bits) ou `IntN` (entier signé de N bits), décrit au chapitre 8 page 45.

Le langage accepte des constantes littérales entières d'une valeur quelconque. Une constante littérale entière a pour type `LiteralInt`.

Il est valide de déclarer des constantes de type `LiteralInt` :

```
let N LiteralInt = 1_000_000 // Ok, N a pour type LiteralInt
```

L'annotation de type peut être omise :

```
let N = 1_000_000 // Ok, N a pour type LiteralInt
```

Il est possible d'utiliser une constante entière statique pour définir une autre constante :

```
let P = N + 1 // Ok, P a pour type LiteralInt, et vaut 1_000_001
```

Le type `LiteralInt` n'est pas acceptable pour une variable, aussi la déclaration suivante provoque une erreur de compilation :

```
var N = 1_000_000 // Erreur, LiteralInt invalide pour une variable
```

Il faut une annotation de type qui nomme un type `UIntN` (entier non signé de N bits) ou `IntN` (entier signé de N bits) :

```
var v  : UInt32 = 1_000_000 // Ok
```

Une constante entière statique de type `LiteralInt` est silencieusement convertie en un type entier `UIntN` ou `IntN`, en vérifiant si la conversion est possible ; par exemple, l'écriture suivante déclenche une erreur de compilation :

```
var w UInt8 = 1_000 // Erreur, 1_000 ne peut pas être représenté
                    // par un entier non signé de 8 bits
```

Comme les constantes entières statiques sont calculées à la compilation, l'écriture suivante est correcte :

```
let A = 1_000_000_000_000_000_000_000_000_000_000_000_000_000
let B = 1_000_000_000_000_000_000_000_000_000_000_000_000_001
var z UInt1 = B - A
```

En effet, la valeur initiale de `z` est 1, représentable par un entier non signé de 1 bit.

Chapitre 8

Les types entiers

Sont définis implicitement les types entiers signés et non signés d'une taille variant entre 1 bit et 32768 bits, et sont notés :

- `UInt1` à `UInt32768` pour les types entiers non signés de 1 à 32768 bits ;
- `Int1` à `Int32768` pour les types entiers signés de 1 à 32768 bits.

PLM définit aussi le type `LiteralInt`, qui est le type de toute constante entière littérale. Mais ce type est aussi applicable à des constantes entières *statiques*, c'est-à-dire dont la valeur est calculée à la compilation. Ce type est décrit au chapitre 7 page 43.

8.1 Constante littérale entière

Le langage accepte des constantes littérales entières d'une taille quelconque. 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
var v Int8 = -128 // Ok
```

Une constante littérale entière a pour type `LiteralInt`, or ce type n'est pas acceptable pour une variable. Par exemple, si on écrit :

```
var v = 28 // Erreur, le type LiteralInt n'est pas valide pour une variable
```

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

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

8.2 Conversion entre objets de type entier

Il y a trois types de conversion entre objets de type entier :

- les conversions toujours possibles `extend Type (exp)` (section 8.2.1 page 46) ;
- les conversions pouvant échouer `convert Type (exp)` (section 8.2.2 page 47) ;
- les troncatures `truncate Type (exp)` (section 8.2.3 page 47).

8.2.1 Conversions toujours possibles : `extend`

Les conversions qui sont toujours possibles sont exprimées par le mot réservé `extend`. Par exemple :

```
let v UInt8 = ...
let x UInt9 = extend UInt9 (v)
let y Int9 =  extend Int9 (v)
let z Int10 = extend Int10 (y)
```

D'une manière générale :

- un entier non signé peut être étendu en un entier non signé de taille strictement supérieure ;
- un entier non signé peut être étendu en un entier signé de taille strictement supérieure ;
- un entier signé peut être étendu en un entier signé de taille strictement supérieure.

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

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

L'annotation de type après le mot-clé `extend` est optionnel : par défaut, le type est inféré par le contexte. Par exemple, on peut écrire :

```
let x Int9 =  ...
let y Int10 = extend (x) // Le type Int10 est inféré du contexte
let z = extend (x) // Invalide  : aucun type ne peut être inféré.
```

8.2.2 Conversions pouvant échouer : `convert`

Les conversions pouvant échouer sont exprimées par le mot réservé `convert`. Par exemple :

```
let s Int8 = ...
let x UInt16 = convert UInt16 (s)
```

L'opérateur `convert` 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, la panique dont le code est donné dans le tableau 27.1 page 139 est déclenchée. L'opérateur `convert` est donc interdit dans les constructions où la panique ne peut être déclenchée : il faut alors utiliser l'opérateur `truncate`.

L'annotation de type après le mot-clé `convert` est optionnel : par défaut, le type est inféré par le contexte. Par exemple, on peut écrire :

```
let s Int8 = ...
let x UInt16 = convert (s) // Le type UInt16 est inféré du contexte
let y = convert (s) // Invalide  : aucun type ne peut être inféré
```

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

De plus, une erreur de compilation est déclenchée si l'opérateur `convert` est utilisé alors que la conversion est toujours possible :

```
let v UInt8 = ...
let y = convert Int16 (v) // Erreur, conversion toujours possible
```

8.2.3 Troncatures : `truncate`

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

```
let s Int8 = -10
let x UInt16 = truncate UInt16 (x)
```

L'annotation de type après le mot-clé `truncate` est optionnel : par défaut, le type est inféré par le contexte. Par exemple, on peut écrire :

```
let s Int8 = -10
let x UInt16 = truncate (s) // Le type UInt16 est inféré du contexte
```

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

```
let y = truncate (s) // Invalide  : aucun type ne peut être inféré
```

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

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

```
let v UInt8 = ...
let y = truncate Int16 (v) // Erreur, conversion toujours possible
```

8.3 Opérateurs infixes de comparaison

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

La comparaison ne peut s'effectuer qu'entre objets du même type entier, ou entre un objet de type entier et une constante littérale entière.

8.4 Opérateurs infixes arithmétiques

Les opérateurs infixes arithmétiques sont listés dans le tableau 8.1 avec leur signification. Ils ne peuvent opérer qu'entre objets du même type entier, ou entre un objet de type entier et une constante littérale entière.

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 8.1 – *Opérateurs infixes arithmétiques*

8.5 Opérateurs préfixés de négation arithmétique

8.5.1 Opérateur –

L'opérateur préfixé `-` est la négation arithmétique avec détection de débordement. Il n'est accepté 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 panique dont le code est donné dans le tableau 27.1.

8.5.2 Opérateur –%

L'opérateur préfixé `-%` est la négation arithmétique sans détection de débordement. Il n'est accepté 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. Cet opérateur ne déclenche jamais de panique.

8.6 Opérateurs infixes bit-à-bit

Les opérateurs infixes bit-à-bit acceptent les types entiers non signés (tableau 8.2).

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

Tableau 8.2 – *Opérateurs infixes bit-à-bit sur les entiers non signés*

8.7 Opérateur préfixé bit-à-bit

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

8.8 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 8.3). `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 8.3 – *Opérateurs infixes de décalage sur les entiers*

8.9 Opérateurs combinées avec une affectation

Les opérateurs suivants sont définis pour les entiers.

`a &= b` est équivalent à `a = a & b`.

`a |= b` est équivalent à `a = a | b`.

`a ∧= b` est équivalent à `a = a ∧ b`.

`a += b` est équivalent à `a = a + b`.

`a +%= b` est équivalent à `a = a +% b`.

`a -= b` est équivalent à `a = a - b`.

`a -%= b` est équivalent à `a = a -% b`.

`a *= b` est équivalent à `a = a * b`.

`a *%= b` est équivalent à `a = a *% b`.

Les opérateurs infixes `&=`, `|=` et `∧=` sont décrits à la section 8.6 page 49.

Les opérateurs infixes `+`, `+%`, `-`, `-%`, `*` et `*%` sont décrits à la section 8.4 page 48.

8.10 Accesseurs

8.10.1 Accesseur `bitReversed`

L'accesseur `bitReversed` est implémenté pour tout type entier. Pour un entier sur n bits, il retourne une valeur dont le bit d'indice i (avec $0 \leqslant i < n$) a la valeur du bit $n - i - 1$ du récepteur. Par exemple :

```
let x UInt3 = 0x_3
let y = x.bitReversed () // 0x_6
let z UInt16 = 0x_1234
let t = z.bitReversed () // 0x_2C48
```

8.10.2 Accesseur `byteSwapped`

L'accesseur `byteSwapped` est implémenté pour tout type entier dont la taille est un multiple de 16 bits, qu'il soit signé ou non (`Int16` , `UInt16` , `Int32` , `UInt32` , `Int48` , `UInt48` , `Int64` , `UInt64` , ...). Il réalise la conversion *big endian* ↔ *little endian*. Par exemple :

```
let x  : UInt32 = 0x_1234_5678
let y = x.byteSwapped () // 0x_8756_3412
let z UInt48 = 0x_1234_5678_ABCD
let t = z.byteSwapped () // 0x_CDAB_8756_3412
```

8.10.3 Accesseur `leadingZeroCount`

L'accesseur `leadingZeroCount` est implémenté pour tout type entier. Il retourne le nombre de bits à zéro à partir du bit le plus significatif. Par exemple :

```
let x  : UInt32 = 0x_1234_5678
let y = x.leadingZeroCount () // 3
```

8.10.4 Accesseur `setBitCount`

L'accesseur `setBitCount` est implémenté pour tout type entier. Il retourne le nombre de bits à un dans la valeur du récepteur. Par exemple :

```
let x UInt16 = 0x_1234
let y = x.setBitCount () // 5
```

8.10.5 Accesseur `trainingZeroCount`

L'accesseur `trainingZeroCount` est implémenté pour tout type entier. Il retourne le nombre de bits à zéro à partir du bit le moins significatif. Par exemple :

```
let x UInt16 = 0x_1234
let y = x.trainingZeroCount () // 2
```

8.11 Construction d'un entier non signé par tranches

Cette expression permet de construire un entier non signé à partir d'entiers non signés ou de booléens. Par exemple :

```
let x = {UInt8 !1 :1 !1 :0 !6 :12} // 0x8C
```

Dans l'exemple ci-dessus, le bit n°7 est à 1, le bit n°6 à 0, et les 6 bits de poids faibles à 12.

Les tranches sont désignés par des sélecteurs (`!1 :` , `!6 :`) qui indiquent le nombre de bits de la tranche. La description commence à partir du bit de poids fort. L'expression qui suit le sélecteur doit être du type entier non signé correspondant : par exemple, `!1 :` → `UInt1` , `!6 :` → `UInt6` .

Attention, une tranche `!1 :` signifie que l'expression doit être un entier non signé sur un bit, et non pas un booléen. Si l'on veut initialiser une tranche de 1 bit à partir d'un booléen, il faut utiliser le sélecteur particulier `!b :` . Par exemple :

```
let x = {UInt8 !b :yes !1 :0 !6 :12} // 0x8C
```

Si toutes les expressions associées aux sélecteurs sont statiques, alors l'expression de construction d'un entier non signé par tranches est aussi une expression statique : on peut donc utiliser cette construction pour définir des constantes globales.

Chapitre 9

Les types flottants

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

Chapitre 10

Le type caractère

Non pris en charge actuellement.

Chapitre 11

Les types chaîne de caractères

La prise en charge des chaînes de caractères est très partielle : uniquement le type `LiteralString` est défini.

11.1 Constante littérale chaîne de caractères

Une constante littérale chaîne de caractères est délimitée par des « " ». Comme les sources PLM sont des textes UTF-8, la constante chaîne de caractères accepte tout caractère UTF-8. Par Exemple : `"Hello !"`.

Une constante littérale chaîne de caractères a pour type `LiteralString`.

11.2 Le type `LiteralString`

11.2.1 Déclaration d'un objet de type `LiteralString`

Le type `LiteralString` représente une référence vers une chaîne statique. La sémantique du langage garantit que cette référence toujours valide[1]. Une utilisation typique est la déclaration d'une constante :

```
let x LiteralString = "Hello !"
```

L'annotation de type peut être omise :

1. On peut faire un parallèle avec le langage C, où un pointeur vers une chaîne littérale est du type « `const char *` » ; cependant, en C, ce pointeur peut recevoir la valeur `NULL`. En PLM, la sémantique interdit cette situation : une référence est toujours valide.

```
let x = "Hello !"
```

La chaîne référencée ne peut pas être modifiée, cependant une référence déclarée par `var` peut figurer comme cible d'une affectation :

```
var x = "Hello !"
x = "Bonjour !"
```

11.2.2 Énumération d'une chaîne

L'instruction `for` permet d'énumérer un objet de type `LiteralString` :

```
for c in x {
  ...
}
```

`c` est une constante locale à l'instruction `for`, de type `UInt8`. Pour le moment, la seule opération que l'on peut faire à l'intérieur de la boucle est d'afficher `c` sous la forme d'un caractère sur l'afficheur LCD :

```
for c in x {
  lcd.writeData_inUserMode (!c)
}
```

Chapitre 12

Les types énumérés

12.1 Déclaration d'un type énuméré

La déclaration d'un type énuméré est introduite par le mot réservé `enum` :

```
enum Feu {
  case vert
  case orange
  case rouge
}
```

12.2 Utilisation d'un type énuméré

12.2.1 Constructeurs

La déclaration d'un type énuméré définit les constructeurs associés au type énuméré, ici `Feu.vert`, `Feu.orange` et `Feu.rouge`.

12.2.2 Constante globale et variable globale

Les constructeurs d'un type énuméré sont *statiques*, c'est-à-dire qu'ils permettent d'initialiser des variables globales et des constantes globales :

```
let ROUGE  : Feu = Feu.rouge
```

L'annotation de type peut être omis une fois, c'est-à-dire que l'on peut aussi écrire :

```
let ROUGE  : Feu = .rouge
```

Ou encore :

```
let ROUGE = Feu.rouge
```

De même, on peut déclarer une variable globale appartenant à un type énuméré :

```
var unFeu  : Feu = Feu.vert
```

L'annotation de type peut être omis une fois, c'est-à-dire que l'on peut aussi écrire :

```
var unFeu  : Feu = .vert
```

Ou encore :

```
var unFeu = Feu.vert
```

12.2.3 Comparaison

Les opérateurs `==`, `≠`, `<`, `≤`, `>` et `≥` permettent de comparer les valeurs d'un type énuméré ; la relation d'ordre est donnée par l'ordre de déclaration des constantes, c'est-à-dire que `Feu.vert < Feu.orange` et `Feu.orange < Feu.rouge`.

12.3 Accesseur

12.3.1 Accesseur `uint`N

Tout type énuméré implémente un accesseur qui retourne la valeur entière non signée associée à la valeur du récepteur. Le type de la valeur retournée est `UIntN`, où N est le nombre de bits nécessaires pour coder la valeur de ce type.

Par exemple, avec le type énuméré :

```
enum Feu {
  case vert
  case orange
  case rouge
}
```

L'accesseur est `uint3`, et :

```
let x = Feu.orange
let y UInt3 = x.uint3 () // 1
```

12.4 Représentation d'un type énuméré

Un type énuméré est représenté dans le code engendré par une valeur codée sur le plus petit nombre de bits nécessaire. Par exemple, le type énuméré suivant code trois valeurs.

```
enum Feu {
  case vert
  case orange
  case rouge
}
```

Un objet de ce type est donc codé sur deux bits, et `vert` est représenté par 0, `orange` par 1 et `rouge` par 2.

Chapitre 13

Le type structure

Le type structure est fondamental en PLM : comme dans bien d'autres langages, il permet de décrire des données structurées. Mais en PLM, une structure permet de décrire les outils de synchronisation tels que le sémaphore, le rendez-vous…

Un type structure a une sémantique de *valeur*, c'est-à-dire qu'une affectation entre instances de structure effectue une copie de l'objet source vers l'objet destination. De plus, il est possible d'interdire la copie d'un type structure, ce qui est indispensable dans le cas où ce type décrit un outil de synchronisation.

Un type structure peut contenir les déclarations de :

- propriétés, section « *Déclaration des propriétés* » page 61 ;
- fonctions, section « *Fonctions* » page 63 ;
- services, section « *Services* » page 67 ;
- sections, section « *Sections* » page 66 ;
- primitives, section « *Primitives* » page 68 ;
- gardes, section « *Gardes* » page 68.

De plus, des attributs paramètrent le comportement de ses instances.

13.1 Déclaration d'un type structure

La déclaration d'un type structure est introduite par le mot réservé `struct`.

```
struct Point {
  // Déclaration de propriétés
  // Déclaration de fonctions
  // Déclaration de services
  // Déclaration de sections
  // Déclaration de primitives
  // Déclaration de gardes
}
```

L'ordre des déclarations est sans importance.

13.2 Déclaration des propriétés

Un type structure peut définir, zéro, un ou plusieurs propriétés. Tout type est acceptable pour une propriété.

Les propriétés déclarées soit présente une valeur initiale par défaut, soit sont initialisées par l'initialiseur engendré implicitement. De cette façon, quand un type structure est instancié, toutes ses propriétés sont initialisées.

13.2.1 Propriétés toutes initialisées par défaut

Voici un exemple qui déclare une structure contenant deux propriétés initialisées par défaut :

```
struct Point {
  var x Int32 = 0
  var y Int32 = 0
}
```

Un type structure dont toutes les propriétés sont initialisées par défaut présente un initialiseur sans argument. On pourra donc écrire :

```
var pt  : Point = Point ()
```

Et l'annotation de type peut être omise :

```
var pt = Point ()
```

13.2.2 Propriétés non initialisées

Pour chaque propriété non initialisée par défaut, l'initialiseur synthétisé présente un argument. Par exemple :

```
struct Point {
  var x  : Int32
  var y  : Int32 = 0
}
```

La propriété `x` n'étant pas initialisée par défaut, l'initialiseur synthétisé présente un argument dont la sélecteur porte le nom de la propriété non initialisée :

```
var pt = Point (!x :4)
```

Si plusieurs propriétés ne sont pas initialisées par défaut, l'initialiseur synthétisé présente des arguments dans l'ordre de déclaration des propriétés non initialisées. Par exemple :

```
struct Point {
  var x  : Int32
  var y  : Int32
}
```

Alors :

```
var pt = Point (!x :4 !y :8)
```

13.2.3 Propriété d'un type structure

Tout type acceptable pour une propriété. Par exemple, le type `Point3` a une propriété de type `Point` :

```
struct Point {
  var x  : Int32 = 0
  var y  : Int32 = 0
}

struct Point3 {
  var p  : Point = Point ()
  var z Int32 = 0
}
```

Toutes les propriétés du type `Point3` étant initialisées par défaut, l'initialiseur est `Point3 ()`. Si des propriétés ne ne sont pas initialisées par défaut, l'initialiseur synthétisé comporte des arguments en conséquences. Par exemple :

```
struct Point {
  var x  : Int32
  var y  : Int32
}

struct Point3 {
  var p  : Point
  var z  : Int32
}
```

L'initialisation s'effectue par :

```
var pt3 = Point3 (!p : Point (!x :4 !y :8) !z :6)
```

13.3 Fonctions

Les fonctions définies dans une structure sont particulières dans le sens où deux attributs permettent de contrôler l'accès aux propriétés :

- l'attribut `@userAccess` autorise la fonction à accéder aux propriétés en mode `user` ;
- l'attribut `@mutating` autorise la fonction à modifier *directement* ou *indirectement* les propriétés.

L'attribut `@userAccess` peut paraître surprenant, mais il permet de garantir l'absence d'accès en parallèle aux propriétés par plusieurs tâches : la fonction d'une variable globale de type structure ne peut pas être appelée en mode `user` si la fonction a été définie avec l'attribut `@userAccess`.

13.3.1 Attribut `@userAccess`

L'attribut `@userAccess` autorise la fonction à lire *directement* les propriétés en mode `user`. Par *directement*, on entend que les instructions de la fonction nomment les propriétés (notation `self.`), ou appellent des fonctions qui déclarent l'attribut `@userAccess`.

Ainsi, l'attribut `@userAccess` est indispensable pour lire les propriétés en mode `user`. Par contre, si la fonction ne s'exécute pas en mode `user`, cet attribut est inutile. Il est aussi inutile si la fonction appelle des sections qui lisent des propriétés.

Si l'on veut l'accès en écriture, ajouter l'attribut `@mutating` est indispensable. Comme les services, primitives et gardes ont par défaut un accès en écriture aux propriétés, cet attribut est indispensable pour que leur appel soit valide.

Voici un code d'exemple qui illustre les différentes possibilités :

```
struct Exemple {
  var x  : Int32

  func user getNok () -> Int32 {
    result = self.x // Erreur  : @userAccess est nécessaire
  }

  func user getOk @userAccess () -> Int32 {
    result = self.x // Ok  : @userAccess autorise l'accès
  }

  func user getNok2  () -> Int32 {
    result = self.getOk () // Erreur  : @userAccess est nécessaire
  }

  func user getOk2 @userAccess () -> Int32 {
    result = self.getOk () // Ok  : @userAccess autorise l'accès
  }

  func section sGet () -> Int32 {
    result = self.x // Ok  : @userAccess est inutile
  }

  section sectionGet () -> Int32 {
    result = self.x
  }

  func user getOk3 () -> Int32 {
    result = self.sectionGet ()
  }
}
```

Dans le code d'exemple ci-dessus :

- la fonction `getNok` s'exécute en mode `user`, ne nomme pas l'attribut `@userAccess`, donc son instruction qui nomme une propriété déclenche une erreur de compilation ;
- la fonction `getOk` s'exécute en mode `user`, nomme l'attribut `@userAccess`, donc son instruction qui nomme une propriété est acceptée par le compilateur ;
- la fonction `getNok2` s'exécute en mode `user`, ne nomme pas l'attribut `@userAccess`, donc son instruction qui nomme une fonction déclarée avec cet attribut déclenche une erreur de com-

pilation ;

- la fonction `getOk2` s'exécute en mode `user`, nomme l'attribut `@userAccess`, donc son instruction qui nomme une fonction déclarée avec cet attribut est acceptée par le compilateur ;
- la fonction `sGet` s'exécute en mode `section`, l'attribut `@userAccess` est inutile pour que son instruction qui nomme une propriété soit acceptée par le compilateur ;
- la section `sectionGet` s'exécute en mode `section`, son instruction qui nomme une propriété est acceptée par le compilateur ;
- la fonction `getOk3` s'exécute en mode `user`, l'attribut `@userAccess` est inutile pour que l'appel à une section soit acceptée par le compilateur.

13.3.2 Attribut `@mutating`

L'attribut `@mutating` autorise une fonction de structure à modifier *directement* les propriétés de la structure, du moment que l'accès ait été autorisé avec `@userAccess`.

Voici un code d'exemple qui reprend le code d'exemple de la section précédente et illustre les différentes possibilités :

```
struct Exemple {
  var x  : Int32

  func user setNok @userAccess (?inX Int32) {
    self.x = inX // Erreur  : @mutating est nécessaire
  }

  func user setOk @userAccess @mutating (?inX Int32) {
    self.x = inX // Ok  : @mutating autorise l'écriture
  }

  func user setNok2 @userAccess (?inX Int32) {
    self.setOk (!inX) // Erreur  : @mutating est nécessaire
  }

  func user setOk2 @userAccess @mutating (?inX Int32) {
    self.setOk (!inX) // Ok  : @mutating autorise l'accès
  }

  section sectionSet @mutating (?inX Int32) {
    self.x = inX
```

```
  }

  func user setOk3 @userAccess @mutating (?inX Int32) {
    self.sectionSet (!inX)
  }

}
```

Dans le code d'exemple ci-dessus :

- la fonction `setNok` ne nomme pas l'attribut `@mutating`, donc son instruction qui écrit une propriété déclenche une erreur de compilation ;
- la fonction `setOk` nomme l'attribut `@mutating`, donc son instruction qui écrit une propriété est acceptée par le compilateur ;
- la fonction `setNok2` ne nomme pas l'attribut `@mutating`, donc son instruction qui écrit une fonction déclarée avec l'attribut `@mutating` déclenche une erreur de compilation ;
- la fonction `setOk2` nomme l'attribut `@mutating`, donc son instruction qui nomme une fonction déclarée avec l'attribut `@mutating` est acceptée par le compilateur ;
- la section `sectionGet` nomme l'attribut `@mutating`, son instruction qui écrit une propriété est acceptée par le compilateur ;
- la fonction `setOk3` s'exécute en mode `user`, l'attribut `@mutating` est nécessaire pour que l'appel à une section déclarée avec l'attribut `@mutating` soit acceptée par le compilateur.

13.4 Sections

Une structure peut définir des *sections*. Une section s'exécute en mode `section`, c'est-à-dire interruptions matérielles masquées ; aucune primitive de l'exécutif n'est accessible dans ce mode. On peut donc utiliser une section pour accéder en exclusion mutuelle aux propriétés de la structure.

Par défaut, une section définie dans une structure a accès aux propriétés en lecture ; pour pouvoir accéder en écriture, il faut ajouter l'attribut `@mutating`.

```
struct Exemple {
  var x  : Int32

  section sectionGet () -> Int32 {
    result = self.x // Ok, accès en lecture par défaut
  }
```

```
  section sectionSet @mutating (?inX Int32) {
    self.x = inX // Ok, @mutating  autorise l'accès en écriture
  }

}
```

Par défaut, une section définie dans une structure est privée, c'est-à-dire appelable uniquement par les autres routines (fonctions, sections, services, primitives, gardes) de la structure. Pour la rendre publique, il faut la déclarer avec le mot réservé `public` :

```
struct Exemple {
  var x  : Int32

  public section sectionGet () -> Int32 {
    result = self.x // Ok, accès en lecture par défaut
  }

  public section sectionSet @mutating (?inX Int32) {
    self.x = inX // Ok, @mutating  autorise l'accès en écriture
  }

}
```

13.5 Services

Une structure peut définir des *services*. Un service s'exécute en mode `service`, c'est-à-dire interruptions matérielles masquées ; seules les fonctions de l'exécutif pouvant libérer des tâches sont accessibles dans ce mode, qui est celui des routines d'interruption. Un exemple de service est la primitive `V` du sémaphore de Dijkstra (section 24.3 page 124).

Par défaut, un service défini dans une structure a accès aux propriétés en lecture et en écriture (l'attribut `@mutating` est inutile).

Par défaut, un service défini dans une structure est privé. Pour le rendre publique, il faut le déclarer avec le mot réservé `public`.

13.6 Primitives

Une structure peut définir des *primitives*. Une primitive s'exécute en mode `primitive`, c'est-à-dire interruptions matérielles masquées ; les fonctions de l'exécutif pouvant bloquer ou libérer des tâches sont accessibles dans ce mode. Un exemple de service est la primitive `P` du sémaphore de Dijkstra (section 24.3 page 124).

Par défaut, une primitive définie dans une structure a accès aux propriétés en lecture et en écriture (l'attribut `@mutating` est inutile).

Par défaut, une primitive définie dans une structure est privée. Pour la rendre publique, il faut la déclarer avec le mot réservé `public`.

13.7 Gardes

Une structure peut définir des *gardes*. Une garde permet d'exprimer une attente élémentaire dans une instruction `sync` (section 24.7 page 133). Un exemple est la garde `P` du sémaphore de Dijkstra (section 24.3 page 124).

Par défaut, une garde définie dans une structure a accès aux propriétés en lecture et en écriture (l'attribut `@mutating` est inutile).

Par défaut, une garde définie dans une structure est privée. Pour la rendre publique, il faut la déclarer avec le mot réservé `public`.

13.8 Extensions

Pour tout type de structure, un nombre quelconque d'extensions peut être défini. Une extension peut déclarer propriétés, fonctions, sections, services, primitives et gardes.

Voici des exemples d'extension :

```
struct Exemple {
  var x  : Int32
}

extension Exemple {
  func user setOk @userAccess @mutating (?inX Int32) {
    self.x = inX
  }
```

```
}

extension Exemple {
  func user setOk2 @userAccess @mutating (?inX Int32) {
    self.setOk (!inX)
  }

  section sectionSet @mutating (?inX Int32) {
    self.x = inX
  }
}

extension E xemple {
  func user setOk3 @userAccess @mutating (?inX Int32) {
    self.sectionSet (!inX)
  }
}
```

13.9 Visibilité des propriétés et des méthodes

Par défaut, les propriétés et les routines définies dans une structure sont privées, c'est-à-dire que seules les routines définies dans la même structure peuvent y accéder. Pour rendre publique une propriété ou une routine, il faut la déclarer avec le mot réservé `public`.

Chapitre 14

Les types opaque

Un *type opaque* est un type dont la définition est externe à PLM, dans le code C associé. Par défaut, les objets de ce type ne peuvent pas être copiés, ni comparés, ni instanciés et leur contenu est inaccessible. La déclaration d'attributs associés modifie ces propriétés par défaut (section 14.2 page 70).

14.1 Déclaration d'un type opaque

La déclaration d'un type opaque est introduite par le mot réservé `newtype` :

```
newtype MonTypeOpaque  : [[32]]
```

La séquence `[[expression]]` caractérise la déclaration d'un type opaque. Les deux niveaux de parenthèses sont obligatoires. Le nombre associé (ici `32`) est le nombre de bits pour représenter un objet de ce type.

L'`expression` doit être une expression statique (c'est-à-dire calculée à la compilation), de type entier. On peut donc faire appel à des constantes globales, comme par exemple :

```
let TAILLE = 32
newtype MonTypeOpaque  : [[TAILLE]] // 32 bits
newtype AutreTypeOpaque  : [[TAILLE * 2]] // 64 bits
```

14.2 Attributs d'un type opaque

La déclaration d'un type opaque accepte deux attributs :

- `@instantiable`, qui rend instanciable un objet d'un type opaque (par défaut, il n'est pas instanciable) ;
- `@copyable`, qui rend copiable un objet d'un type opaque (par défaut, il n'est pas copiable).

Ces deux attributs sont cumulables.

14.2.1 Attribut `@instantiable`

L'attribut `@instantiable` est spécifié après l'indication de taille du type :

```
newtype MonTypeOpaque  : [[32]] @instantiable
```

Par défaut, un type opaque n'est pas instanciable ; l'attribut `@instantiable` le rend instanciable, c'est-à-dire que l'on peut écrire :

```
var t MonTypeOpaque = MonTypeOpaque ()
```

Ou encore, en supprimant l'annotation de type :

```
var t = MonTypeOpaque ()
```

L'expression `MonTypeOpaque ()` est à une instance de `MonTypeOpaque` dont la valeur correspond à des zéros binaires.

14.2.2 Attribut `@copyable`

L'attribut `@copyable` est spécifié après l'indication de taille du type :

```
newtype MonTypeOpaque  : [[32]] @copyable
```

Par défaut, un type opaque n'est pas copiable ; l'attribut `@copyable` le rend copiable, c'est-à-dire que l'on peut écrire :

```
var t = MonTypeOpaque ()
var u = t // Copie d'une instance de MonTypeOpaque
```

Chapitre 15

Les types tableau

PLM implémente les tableaux de taille fixe. La déclaration d'un type tableau utilise obligatoirement la construction `type` (section 15.1 page 72), il n'est pas possible de déclarer un type tableau anonyme.

15.1 Déclaration d'un type tableau

La déclaration d'un type tableau est réalisée par la construction `newtype` :

```
newtype MonTypeTableau  : UInt32 [20]
```

Cette construction déclare le type `MonTypeTableau` comme tableau de 20 instances de `UInt32` .

Tout type est acceptable comme élément de tableau (ici `UInt32`), du moment qu'il est instanciable et copiable. Essayer de définir un type tableau avec un type non instanciable et/ou non copiable entraîne une erreur de compilation.

La taille du tableau est une expression statique, de type entier. Il est donc possible de faire référence à des constantes globales, comme par exemple :

```
let SIZE = 20
typealias MonTypeTableau = [SIZE @x UInt32]
```

15.2 Construction d'un tableau

Un tableau statique implémente deux constructeurs :

- un constructeur qui initialise tous les éléments du tableau à la même valeur (section 15.2.1 page 73) ;
- un constructeur qui initialise chaque élément du tableau à une valeur particulière (section 15.2.2 page 73).

15.2.1 Constructeur (!repeated)

L'expression `MonTypeTableau (!repeated :exp)` est une instance du type `MonTypeTableau`, dont tous les éléments sont initialisés avec la valeur de `exp`. Par exemple :

```
let v  : UInt32 = 0
var t  : UInt32 [10] = UInt32 [10] (!repeated :v)
```

15.2.2 Constructeur (!!...)

Ce constructeur possède autant d'arguments que d'éléments dans le tableau. Le premier est affecté à l'élément d'indice 0, le deuxième à celui d'indice 1, ... Par exemple :

```
var t UInt32 [3] = UInt32 [3] (!0 !1 !2)
```

15.3 Déclaration d'une instance de tableau

La déclaration d'une instance de tableau s'effectue en nommant le type tableau et l'expression de construction de ce tableau, où `v` est une valeur du type de l'élément de tableau (ici, `UInt32`) :

```
let v UInt32 = 0
var t  : MonTypeTableau = MonTypeTableau (!repeated :v)
```

Et on peut omettre l'annotation de type :

```
var t = MonTypeTableau (!repeated :v)
```

15.4 Obtention de la taille d'un tableau

L'expression `MonTypeTableau.count` renvoie la taille du tableau, sous la forme d'un entier statique (c'est-à-dire de type `LiteralInt`, voir chapitre 7 page 43). Ou peut aussi appliquer `count` à une instance de tableau.

On peut donc utiliser les expressions `$t.count` et `$monTypeTableau.count` pour itérer sur tous les élements d'un tableau :

```
var t = MonTypeTableau (!repeated :0)
for i UInt32 in 0 ..< t.count {
  t [i] = i
}
for i UInt32 in 0 ..< MonTypeTableau.count {
  t [i] = i
}
```

Une erreur de compilation est déclenchée si on écrit :

```
var s = MonTypeTableau.count // ERREUR
```

En effet, `MonTypeTableau.count` est un entier statique et le type *entier statique* ne peut pas être attribué à une variable. Il faut préciser obligatoirement un type d'entier :

```
var s UInt32 = MonTypeTableau.count // Ok
```

Ce type peut être signé ou non signé, du moment que sa plage de valeur permet de représenter la valeur statique.

15.5 Accès à un élément d'un tableau

L'accès à un élément d'un tableau s'effectue par la notation classique `[expression_indice]`. Les indices valides commencent à 0 jusqu'à la taille du tableau moins 1.

L'accès à un élément de tableau est valide dans les constructions suivantes :

- expression : `var x = t [1]` ;
- cible d'une affectation : `t [1] = x` ;
- cible d'un opérateur combiné à une affectation : `t [1] += x` ;

L'`expression_indice` doit être une expression entière, signée ou non signée.

15.5.1 Expression indice statique

Si l'`expression_indice` est statique, c'est-à-dire dont la valeur est calculée à la compilation, la vérification de sa validité est effectuée à la compilation. Par exemple, pour un tableau `t` de 20 éléments :

```
t [0] // Ok
t [19] // Ok
t [-1] // Erreur de compilation, indice négatif
t [20] // Erreur de compilation, indice trop grand
```

Comme la validité est effectuée à la compilation, aucune vérification de validité n'est effectuée à l'exécution.

15.5.2 Expression indice non signée

Si l'`expression_indice` est une instance d'un type entier non signé, il y a deux possibilités.

Soit ce type entier non signé présente une valeur maximum supérieure ou égale à la taille du tableau : alors le code engendré vérifie à l'exécution que l'`expression_indice` a une valeur valide. Dans l'exemple ci-dessous, le tableau `t` contient 20 éléments, l'`expression_indice` doit donc être inférieure ou égale à 19 ; or la plage de valeurs de `UInt32` dépasse cette borne, aussi la validité est vérifiée à l'exécution. En cas d'échec, la panique est déclenchée (tableau 27.1 page 139).

```
var i UInt32 = ...
var x = t [i] // Vérification à l'exécution que i < 20
```

La seconde possibilité est que ce type entier non signé présente une valeur maximum strictement inférieure à la taille du tableau : la compilation garantit que l'indice sera toujours valide, aucune vérification n'est effectuée à l'exécution. Par exemple :

```
var i UInt4 = ... // Donc 0 ≤ i ≤ 15
var x = t [i] // Aucune vérification à l'exécution car toujours i < 20
```

15.5.3 Expression indice signée

Quand l'`expression_indice` est une expression entière signée, il faut vérifier :

- qu'elle est positive ou nulle ;
- qu'elle est strictement inférieure à la taille du tableau.

La première vérification est toujours réalisée à l'exécution ; la seconde dépend de la valeur maximum du type entier, de manière analogue à ce qui est fait pour un indice entier non signé. Donc, pour un tableau `t` de 20 éléments :

```
var i Int32 = ...
var x = t [i] // Vérification à l'exécution que 0 ≤ i < 20
```

```
var i Int4 = ... // Donc −16 ≤ i < 15
var x = t [i] // Vérification à l'exécution que 0 ≤ i
```

En cas d'échec de la vérification à l'exécution, la panique est déclenchée (tableau 27.1 page 139).

15.5.4 Accès à un élément en mode panique

Dans une liste d'instructions devant être exécutée en mode panique, les instructions pouvant engendrer une panique sont interdites et leur présence entraîne une erreur de compilation.

Ainsi l'accès à un élément de tableau est donc accepté par le compilateur si l'`expression_indice` est :

- une expression statique ;
- ou une expression entière non signée, dont la valeur maximum est strictement inférieure à la taille du tableau.

Donc l'accès à un élément de tableau est donc rejeté par le compilateur si l'`expression_indice` est :

- une expression entière signée ;
- ou une expression entière non signée, dont la valeur maximum est supérieure ou égale à la taille du tableau.

Chapitre 16

Tableaux statiques constants

PLM permet de construire des tableaux statiques constants, en séparant déclaration du tableau et constitution. Cette caractéristique s'appuie sur les trois constructions suivantes :

- déclaration du tableau statique (section 16.1) ;
- ajout d'un élément au tableau statique (section 16.2) ;
- parcours d'un tableau statique (section 16.4).

16.1 Déclaration

La déclaration d'un tableau statique est réalisée par la construction `staticArray` :

```
staticArray maListeStatique {
  let a UInt32
  let b UInt32
}
```

Cette construction déclare la constante `maListeStatique` comme tableau constant vide. Pour le remplir, utiliser la construction `extend staticArray` (section 16.2).

La composition de chaque élément est spécifiée par la liste des propriétés, chacune d'elles étant définie par son nom et son type.

16.2 Ajout d'un élément au tableau

La construction `extend staticArray` permet d'ajouter un élément en fin de tableau statique :

```
extend staticArray maListeStatique (5, 9)
```

Toutes les propriétés de l'élément ajouté doivent être initialisées par une expression statique.

Pour ajouter plusieurs éléments, on peut écrire :

```
extend staticArray maListeStatique (5, 9)
extend staticArray maListeStatique (15, 8)
```

Ou utiliser le délimiteur `;` pour séparer les éléments apparaissant dans une même déclaration :

```
extend staticArray maListeStatique (5, 9 ; 15, 8)
```

16.3 Ordre des éléments

Le compilateur regroupe les constructions `extend staticArray` d'un même fichier de façon à ce que les éléments soient dans le tableau dans l'ordre d'apparition dans le fichier. Si les constructions `extend staticArray` apparaissent dans plusieurs fichiers, l'ordre des groupes formés par fichier est imprévisible.

16.4 Parcours d'un tableau statique

L'instruction `for` (section 21.13 page 111) est la seule qui accède à un tableau statique. Elle permet de parcourir tous les éléments du tableau :

```
var total UInt32 = 0
for élément in maListeStatique {
  total += élément.a
  total += élément.b
}
```

L'élément courant est désigné par la constante `élément`, et on accède aux propriétés par la notation pointée habituelle.

16.5 Fonctions

Il est possible de définir des propriétés désignant une routine. Celle-ci peut être une *vraie* fonction (c'est-à-dire appelable dans une expression et qui renvoie une valeur), ou bien une *procédure*, appelable dans une instruction et qui ne renvoie aucune valeur. Par exemple :

```
staticArray maListeStatique {
  let propriétéFonction func user () -> UInt32
  let propriétéProcédure func user ()
}

func user maFonction () -> UInt32 {
  result = 10
}

func user maProcédure () -> UInt32 {
  ...
}

extend staticArray maListeStatique (func maFonction (), func maProcédure ())
```

Et l'appel de ces fonctions :

```
var total UInt32 = 0
for élément in maListeStatique {
  total += élément.propriétéFonction ()
  élément.maProcédure ()
}
```

Chapitre 17

Les registres de contrôle

Dans ce chapitre, nous allons décrire :

- comment déclarer un registre de contrôle, comment lui affecter une valeur, et le lire (section 17.2 page 80) ;
- comment déclarer les champs d'un registre de contrôle, et comment les utiliser (section 17.3 page 82) ;
- comment déclarer plusieurs registres de contrôle ayant la même composition de champs (section 17.4 page 85) ;
- comment déclarer et utiliser un tableau de registres de contrôle (section 17.5 page 85) ;
- les attributs applicables aux registres de contrôle (section 17.6 page 87) ;
- les restrictions d'usage des registres de contrôle (section 17.7 page 88).

À titre d'exemple, nous allons nous intéresser aux registres `PORTx_PCRn` du micro-contrôleur `MK20DX256VLH7` qui équipe les cartes *Teensy 3.x* (figure 17.1). La documentation de ce micro-contrôleur indique que l'un des registres de cette famille, `PORTA_PCR0` est à l'adresse `0x4004_9000`.

17.1 Groupe de registres

17.2 Simple déclaration d'un registre

Pour déclarer le registre `PORTA_PCR0` (figure 17.1), situé à l'adresse `0x4004_9000`, on écrit :

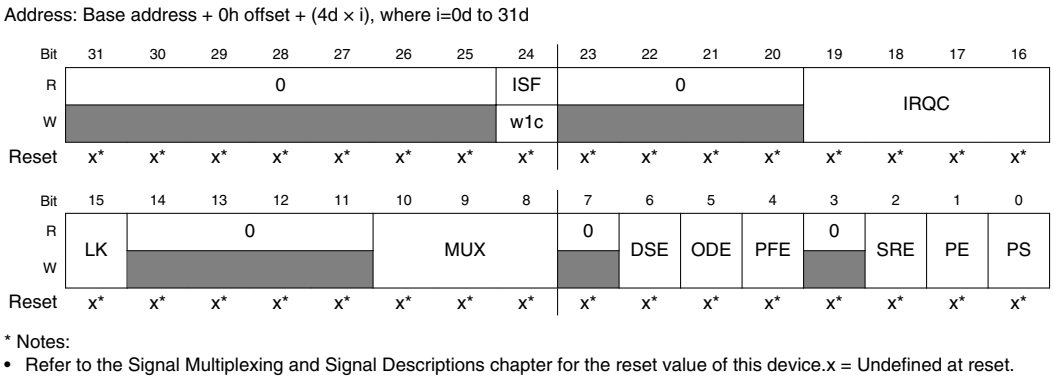


Figure 17.1 – *Registres de contrôle PORTx_PCRn intégré dans le MK20DX256VLH7*

```
registers #PORTA 0x4004_9000 {
  PCR0 0x0 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 configurer le bit 0 du port A en entrée ou en sortie logique, il faut écrire 1 dans le champ MUX et zéro dans les autres champs. Comme le champ MUX commence au 8^e bit, on écrit :

```
PORTA.PCR0 = 1 << 8
```

Si l'on veut que ce port soit un *collecteur ouvert* si il est programmé en sortie, il faut mettre le champ ODE à 1. On écrit donc :

```
PORTA.PCR0 = (1 << 8) | (1 << 5)
```

Lire le contenu du registre est réalisé par une instruction d'affectation :

```
let x = PORTA.PCR0
```

Le type de la constante `x` est `UInt32`, déduit du type du registre de contrôle PORTA_PCR0.

Pour savoir si le bit ODE est activé, on réalise un masquage (l'annotation du type `Bool` est facultative) :

```
let ODEactivé Bool = (PORTA.PCR0 & (1 << 5)) ≠ 0
```

Pour obtenir la valeur du champ MUX, on effectue un décalage, suivi d'un masquage :

```
let champMUX UInt32 = (PORTA.PCR0 >> 8) & 7
```

Toutes ces formulations peuvent être rendues plus intelligibles en précisant la composition du registre `PORTA_PCR0` dans sa déclaration. C'est ce qui va être réalisé dans la section suivante.

17.3 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. Par exemple, pour le registre `PORTA_PCR0` et en s'appuyant par sa description dans la figure 17.1 page 81 :

```
registers PORTA {
  PCR0 0x4004_9000 UInt32 {
    7, ISF, 4, IRQC :4, LK, 4, MUX :3, 1, DSE, ODE, PFE, 1, SRE, PE, PS
  }
}
```

Entre accolades, trois définitions différentes peuvent apparaître :

- un nombre indique le nombre de bits consécutifs inutilisés ;
- un identificateur (par exemple `ISF`) nomme un champ booléen ;
- un identificateur suivi du délimteur `:` et d'un nombre (par exemple `IRQC :4`) 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 `ISF` porte le n°24, `IRQC` s'étend sur 4 bits à partir du n°16, ...

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`.

Définir la composition des champs d'un registre permet d'utiliser des constructions qui simplifient :

- l'obtention de leur valeur (section 17.3.1 page 83) ;
- la construction d'une valeur à affecter à un registre de contrôle (section 17.3.2 page 83).

17.3.1 Accès en lecture aux champs

À la section 17.2, pour obtenir la valeur du champ MUX est activé, on réalisait un décalage suivi d'un masquage :

```
let champMUX UInt32 = (PORTA.PCR0 >> 8) & 7 // 0, 1, 2, ..., 7
```

Plusieurs formulations nommant le champ MUX sont possibles.

La première renvoie la valeur du champ non décalée :

```
let résultatNonDécalé UInt32 = PORTA.PCR0.MUX
  // 0, 0x100, 0x200, ..., 0x700
```

Pour obtenir la valeur d'un champ justifiée à droite, on utilise l'accesseur `shifted` :

```
let champMUX UInt32 = PORTA.PCR0.MUX.shifted // 0, 1, 2, ..., 7
```

L'expression `PORTA.PCR0.MUX.shifted` est équivalente à `(PORTA.PCR0 >> 8) & 7`.

Pour le champ booléen ODE, on écrivait à la section 17.2 :

```
let ODEactivé Bool = (PORTA.PCR0 & (1 << 5)) ≠ 0
```

On peut maintenant écrire (noter que le type du résultat est `UInt32`) :

```
let champODEnonDécalé UInt32 = PORTA.PCR0.ODE // 0 ou 2**5
```

De même, on peut obtenir la valeur justifiée à droite (noter que le type du résultat est toujours `UInt32`) :

```
let champODEdécalé UInt32 = PORTA.PCR0.ODE.shifted // 0 ou 1
```

Pour obtenir la valeur valeur booléenne, on utilise l'accesseur `bool` (noter que le type du résultat est maintenant `Bool`) :

```
let ODEactivé Bool = PORTA.PCR0.ODE.bool // no ou yes
```

L'expression `PORTA.PCR0.ODE.bool` est équivalente à `(PORTA.PCR0 & (1 << 5)) ≠ 0`.

17.3.2 Construction à partir des valeurs de champs d'un registre de contrôle

La construction particulière `$registre {champ :expression, ...}` permet de définir facilement la valeur à affecter à un registre de contrôle. Prenons toujours l'exemple du registre PORTA_PCR0 dont la composition est décrite à la figure 17.1 page 81.

À la section 17.2, pour écrire 1 dans le champ MUX et zéro dans les autres champs on écrivait :

```
PORTA.PCR0 = 1 << 8
```

En utilisant la notation dédiée, on écrit maintenant :

```
PORTA.PCR0 = {PORTA.PCR0 !MUX :1}
```

L'expression `{PORTA.PCR0 !MUX :1}` est équivalente à `1 << 8`.

Si l'on veut que ce port soit un *collecteur ouvert* si il est programmé en sortie, il faut mettre le champ `ODE` à 1, et on écrivait :

```
PORTA.PCR0 = (1 << 8) | (1 << 5)
```

On peut maintenant écrire :

```
PORTA.PCR0 = {PORTA.PCR0 !MUX :1 !ODE :1}
```

17.3.3 Vérifications sémantiques

Examinons maintenant les conditions de validité de l'`expression` dans la construction (décrite à la section 17.3.2 page 83) `$registre {champ :expression, ...}`.

Expression entière statique. Le compilateur vérifie qu'elle est comprise entre 0 et $2^n - 1$, n étant le nombre de bits du champ : par exemple, pour le champ `MUX` de 3 bits, une valeur entre 0 et 7. Ainsi :

```
PORTA.PCR0 = {PORTA.PCR0 !MUX :1-2} // Erreur de compilation, exp. < 0
PORTA.PCR0 = {PORTA.PCR0 !MUX :8} // Erreur de compilation, exp. > 7
```

Expression entière non statique signée. Le compilateur considère que c'est une erreur : uniquement une expression entière non statique non signée est acceptable.

Expression entière non statique non signée. Il y a plusieurs sous cas à examiner.

Si l'`expression` est d'un type entier non signé dont le nombre de bits est inférieur ou égal au nombre de bits du champ, alors toute valeur de l'`expression` est acceptable : le code engendré se borne à faire le décalage à gauche de la valeur de l'`expression`.

Par exemple, le champ `MUX` s'étendant sur 3 bits, une expression de type `UInt3`, `UInt2` ou `UInt1` est toujours acceptée :

```
let x UInt2 = 1
PORTA.PCR0 = {PORTA.PCR0 !MUX :x} // Ok
```

Dans le cas contraire, c'est-à-dire si l'`expression` est d'un type entier non signé dont le nombre de bits est strictement supérieur au nombre de bits du champ, une vérification de la valeur à l'exécution est

effectuée : si la valeur de l'`expression` est trop grande, la panique (code : voir tableau 27.1 page 139) est déclenchée. Si la génération de code panique n'est pas activée, le débordement est silencieusement ignoré. Par exemple :

```
let x UInt8 = ...
PORTA.PCR0 = {PORTA.PCR0 !MUX :x} // Vérification à l'exécution
```

Si le code ci-dessus apparaît dans une routine où la génération de panique est interdite (par exemple, dans une routine `boot`), alors il déclenche une erreur de compilation. Il faut ajouter une troncature explicite pour le code soit accepté :

```
let x UInt8 = ...
PORTA.PCR0 = {PORTA.PCR0 !MUX :truncate UInt3 (x)}
```

Ce code n'effectue aucune vérification à l'exécution.

17.4 Déclaration de plusieurs registres

Il est possible de regrouper les déclarations de registres partageant la même décomposition de leur champs. Par exemple, pour les registres `PORT`n`_PCR`m du `mk20dx256` (seule la déclaration de deux premiers registres est montrée) :

```
registers PORTA {
  PCR0 0x4004_9000
  PCR1 0x4004_9004
  UInt32 {
    7, ISF, 4, IRQC :4, LK, 4, MUX :3, 1, DSE, ODE, PFE, 1, SRE, PE, PS
  }
}
```

17.5 Déclaration d'un tableau de registres

Le micro-contrôleur LPC2294 de NXP possède 4 modules CAN.

> La documentation du LPC2294 numérote ces modules de 1 à 4. Dans ce document, ils sont numérotés de 0 à 3, ce qui s'avère beaucoup plus pratique à l'usage.

Les registres de ces modules sont aux adresses :

```
0xE004_4000 + (canal << 14) + register_offset
```

où `canal` vaut 0 pour le module 0, ..., 3 pour le module 3 ; `register_offset` est une valeur propre à chaque type de registre. Par exemple, pour les registres CANCMR, l'offset est égal à 4. Les quatre registres CANCMR sont donc aux adresses :

`CANCMR0` : `0xE004_4000 + (0 << 14) + 4 = 0xE004_4004`

`CANCMR1` : `0xE004_4000 + (1 << 14) + 4 = 0xE004_8004`

`CANCMR2` : `0xE004_4000 + (2 << 14) + 4 = 0xE004_C004`

`CANCMR3` : `0xE004_4000 + (3 << 14) + 4 = 0xE005_0004`

Il est possible de déclarer ces registres individuellement :

```
register
  CANCMR0 0xE004_4004
  CANCMR1 0xE004_8004
  CANCMR2 0xE004_C004
  CANCMR3 0xE005_0004
UInt32 {
  STB3, STB2, STB1, SRR, CDO, RRB, AT, TR
}
```

Mais on n'a pas de solution simple pour sélectionner un de ces registres en fonction du numéro de module. Si on veut écrire une valeur `v` dans le registre désigné par la variable `n` (dont la valeur est comprise entre 0 et 3), il faut écrire :

```
if n == 0 {
  CANCMR0 = v
}else if n == 1 {
  CANCMR1 = v
}else if n == 2 {
  CANCMR2 = v
}else{
  CANCMR3 = v
}
```

On peut simplifier l'accès en déclarant les registres CANCMRn comme un tableau de registres de contrôle :

```
register
  CANCMR[4] 0xE004_4004  : 1 << 14
UInt8 {
  STB3, STB2, STB1, SRR, CDO, RRB, AT, TR
}
```

D'une manière générale, la déclaration d'un tableau de registres de contrôle est de la forme :

```
nom_registre [taille] adresse_base~ : multiplicateur
$type { ... }
```

La `taille` doit toujours être égale à une puissance de 2. L'adresse du registre d'indice i est égale à `adresse_base + i * multiplicateur`. Ici :

Adresse de `CANCMR[0]` : `0xE004_4004 + 0 * (1 << 14) = 0xE004_4004`

Adresse de `CANCMR[1]` : `0xE004_4004 + 1 * (1 << 14) = 0xE004_8004`

Adresse de `CANCMR[2]` : `0xE004_4004 + 2 * (1 << 14) = 0xE004_C004`

Adresse de `CANCMR[3]` : `0xE004_4004 + 3 * (1 << 14) = 0xE005_0004`

Si vous voulez confirmer le calcul des adresses des registres de contrôle, utilisez l'option de la ligne de commande **`--control-register-map`** (section 3.3 page 32) qui affiche le détail de la définition des registres de contrôle dans un fichier HTML.

L'accès aux registres de contrôle s'effectue alors en utilisant la notation `[...]` habituelle de l'accès à un élément de tableau. Par exemple, en reprenant l'exemple précédent, écrire une valeur `v` dans le registre désigné par la variable `n` (dont la valeur est comprise entre 0 et 3) s'exprime simplement par :

```
CANCMR[n] = v
```

L'indice d'un tableau de registre peut-être une expression entière statique, ou une expression dynamique signée ou non signée. Les vérifications à la compilation et à l'exécution sont les même que pour l'accès à un élément de tableau (voir la section 15.5 page 74).

En particulier, si l'indice est une expression de type non signée dont la valeur maximum est strictement inférieure à la taille du tableau, aucune vérification n'est faite à l'exécution, puisque l'indice sera toujours valide :

```
let n UInt2 = ...
CANCMR[n] = v // Aucune vérification, indice toujours valide
```

17.6 Attributs d'un registre de contrôle

17.6.1 Attribut `@ro`

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

```
registers {
```

```
    CALIB @ro 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.

17.6.2 Attribut `@user`

La déclaration d'un registre accepte l'attribut `@user`, qui signifie qu'il est accessible en mode `user`. Par défaut, un registre de contrôle n'est pas accessible en mode `user`. Par exemple :

```
registers GPIOE {
  PSOR @user 0x400F_F104 UInt32
}
```

Évidemment, il faut que le matériel accepte effectivement que le registre soit accessible quand le processeur en mode *utilisateur*.

17.7 Restrictions d'usage des registres

Un registre ne peut pas :

- apparaître comme paramètre effectif en entrée d'une procédure ;
- apparaître comme paramètre effectif en sortie/entrée d'une procédure.

Prenons un exemple ; la procédure `uneProcedure` présente un argument formel en sortie, et on suppose que REGISTRE est un registre de type `UInt32` :

```
func user uneProcedure (!outValue UInt32) {
  outValue = 5
}
```

L'écriture suivante est rejetée par le compilateur (passage d'un registre comme paramètre effectif en entrée) :

```
func user autreProcedure () {
  uneProcedure (?REGISTRE) // Erreur
}
```

Par contre, l'écriture suivante est correcte (écriture du registre) :

```
func user autreProcedure () {
  REGISTRE = 5 // Ok
}
```

Chapitre 18

Déclaration des constantes globales

Les constantes peuvent être déclarées en deux endroits :

- en dehors de toute routine : c'est une constante globale (voir ci-après) ;
- parmi les instructions d'une routine : c'est une constante locale à la routine (voir section 21.2 page 106).

18.1 Déclaration d'une constante globale

La déclaration d'une constante globale est la suivante :

```
let nom  : Type = expression_statique
```

Où :

- `nom` est le nom de la constante globale ;
- `Type` est le nom du type de la constante globale ;
- `expression_statique` est l'expression qui fournit la valeur de cette constante ; cette expression est calculée lors de la compilation.

La portée d'une constante globale est le programme dans son intégralité : peut importe le fichier et la ligne où elle est déclarée.

L'`expression_statique` ne peut pas nommer une autre constante globale, ni une variable globale.

Chapitre 19

Routines

PLM définit plusieurs natures de routines :

- les *procédures*, dont l'appel est une instruction (section 19.3.2 page 96) ;
- les *fonctions*, dont l'appel est une instruction ou apparaît dans une expression (section 19.3 page 95) ;
- les *boot routines*, exécutées une fois avant l'initialisation des variables globales (section 5.2 page 40) ;
- les *init routines*, exécutées une fois après l'initialisation des variables globales (section 5.3 page 40) ;
- les *routines de panique*, exécutées lors d'une panique (section 27.3 page 139) ;
- les routines *système*, qui sont les points d'entrée des routines privilégiées à partir des routines utilisateur (section 19.4 page 99) ;
- les *routines d'interruption*, qui sont des primitives de l'exécutif (section 19.5 page 100).

19.1 Modes logiques

À chaque routine est attaché statiquement un mode d'exécution.

La plupart des processeurs définit plusieurs modes d'exécution, typiquement un mode *système* ou *privilégié* (où toutes les ressources sont accessibles), et un mode *utilisateur*, où certaines instructions, espace mémoire, registres de contrôles sont inaccessibles.

Cependant, un mode *système* unique est trop sommaire pour refléter toutes les natures de routines. C'est pourquoi PLM définit neuf modes de fonctionnement, `user`, `primitive`, `service`, `section`, `safe`, `guard`, `panic`, `boot` et `init`, qui sont décrits ci-après.

19.1.1 Définition des modes

`user`. Ce mode est le mode d'exécution des tâches. C'est le seul mode qui correspond au mode *user* du processeur. Seuls les registres de contrôle déclarés avec l'attribut `@user` sont accessibles. Les instructions qui peuvent engendrer une panique sont autorisées.

`primitive`. Dans ce mode, les opérations de l'exécutif qui peuvent bloquer ou rendre prête une tâche sont appelables. Tous les registres de contrôle sont accessibles, et les instructions qui peuvent engendrer une panique sont autorisées.

`service`. Seule différence avec le mode précédent, les opérations de l'exécutif qui peuvent bloquer une tâche ne sont pas appelables, uniquement celles qui peuvent rendre prête une tâche le sont.

`section`. Dans ce mode, on ne peut ni bloquer ni rendre prête une tâche. Ce mode permet d'exécuter des opérations de manière indivisible. Les instructions qui peuvent engendrer une panique sont autorisées.

`safe`. Par rapport au mode précédent, les instructions qui peuvent engendrer une panique ne sont pas autorisées.

`guard`. C'est le mode d'évaluation des commandes gardées. Les seules opérations de l'exécutif autorisées sont celles relatives aux gardes. Les instructions qui peuvent engendrer une panique sont autorisées.

`panic`. Lorsqu'une instruction engendre une panique, l'exécution est déroutée vers les routines dédiées qui s'exécutent dans ce mode. Les instructions qui peuvent engendrer une panique n'y sont pas autorisées.

`boot`. C'est le mode après le démarrage du micro-contrôleur, pour le configurer. L'accès aux variables globales est interdit (elles ne sont pas encore initialisées), seul l'accès aux registres de contrôle est autorisé. Les instructions qui peuvent engendrer une panique n'y sont pas autorisées.

`init`. C'est le mode d'initialisation. L'initialisation est exécutée après le `boot`, l'accès aux variables globales est autorisé, ainsi que l'accès aux registres de contrôle. Les instructions qui peuvent engendrer une panique n'y sont pas autorisées.

19.1.2 Changement de mode

Les possibilités de changements de mode sont illustrés par la figure 19.1.

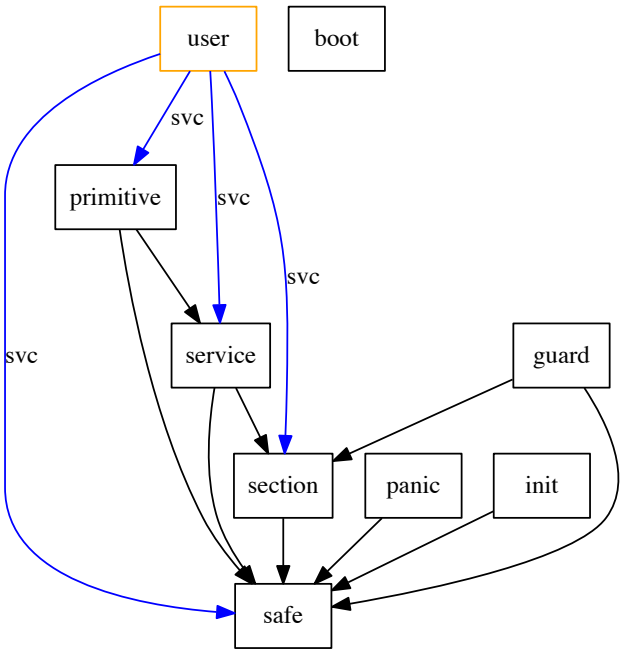


Figure 19.1 – *Graphe des changements de mode*

Paramètre formel	Transfert d'information	Sélecteur
Entrée	Lors de l'appel, de l'appelant vers l'appelé	`?` ou `?selecteur :`
Sortie	Lors du retour, de l'appelé vers l'appelant	`!` ou `!selecteur :`
Entrée / sortie	Lors de l'appel, de l'appelant vers l'appelé, et lors du retour, de l'appelé vers l'appelant	`?!` ou `?!selecteur :`

Tableau 19.1 – *Paramètres formels*

En mode `user`, on peut appeler une routine `system` déclarée avec le mode `primitive`, `service`, `section` ou `safe`. Appeler une routine `system` est la seule façon de changer de mode d'exécution. L'implémentation de l'appel est réalisé par une instruction de type *Supervisor Call*.

Tous les changements de modes décrits ci-après sont implémentés au moyen d'une instruction classique d'appel de sous-programme.

Les modes `primitive`, `service`, `section` et `safe` sont des modes avec des privilèges décroissants. Aussi, les appels `primitive` -> `service` -> `section` -> `safe` sont valides.

Le mode `boot` est le mode des routines exécutés au démarrage du processeur ; ce mode est isolé.

Le mode `init` est le mode des routines d'initialisation ; le seul changement de mode autorisé est vers le mode `safe`.

Le mode `panic` est le mode des routines exécutées suite à une panique ; le seul changement de mode autorisé est vers le mode `safe`, car les instructions pouvant engendrer la panique n'y sont pas autorisées.

Enfin, le mode `guard` est le mode d'évaluation des commandes gardées. On peut y appeler les routines `section` et `safe`.

19.2 Paramètres formels, arguments effectifs, sélecteurs

19.2.1 Paramètres formels

Il existe trois natures de paramètres formels : *entrée*, *sortie* et *entrée / sortie*, décrits dans le tableau 19.1. Un sélecteur peut être *anonyme* (par exemple `?` pour un paramètre formel en entrée), ou comporter un nom (le nom « `nom` » pour `?nom :`).

Une procédure déclare zéro, un ou plusieurs paramètres formels qui peuvent être en *entrée*, en *sortie* ou en *entrée/sortie*. Une fonction déclare zéro, un ou plusieurs paramètres formels en *entrée*.

Paramètre formel	Sélecteur	Argument effectif	Sélecteur
Entrée	`?`	Sortie	`!expression`
	`?selecteur :`		`!selecteur :expression`
Sortie	`!`	Entrée	`?variable`
	`!selecteur :`		`?selecteur :variable`
Entrée/sortie	`?!`	Sortie/entrée	`!?variable`
	`?!selecteur :`		`!?selecteur :variable`

Tableau 19.2 – *Paramètre formel et argument effectif*

19.2.2 Arguments effectifs

La syntaxe des différents paramètres formels et de leur argument effectif est résumée dans le tableau 19.2.

19.2.3 Signature d'une routine

En PLM, une routine est identifiée par son nom et la liste de ses sélecteurs. Il est donc possible de déclarer des routine de mêmes noms, du moment qu'elles se distinguent par le nombre, la nature et le nom des sélecteurs.

[DES EXEMPLES POUR COMPLÉTER]

19.3 Fonctions

Sous le terme *fonction* sont en fait définies les *procédures* et les « vraies » *fonctions*.

Une procédure est appelable dans une instruction et ne peut renvoyer des valeurs que par ses paramètres formels de sortie et d'entrée/sortie.

Une fonction est appelable dans une expression et renvoie une valeur.

19.3.1 Déclaration d'une « vraie » fonction

La déclaration d'une fonction est la suivante :

```
func mode nom @attribut (arguments_formels) -> Type {
  liste_instructions
}
```

Où :

- `nom` est le nom de la fonction ;
- `mode` est le mode associé à la fonction, c'est l'un des mots réservés `primitive`, `service`, `section` ou `safe` ;
- `@attribut` est une liste éventuellement vide d'attributs associés à la fonction ;
- `arguments_formels` est la liste (éventuellement vide) des paramètres formels ;
- `Type` est le type de la valeur renvoyée.

Contrairement à beaucoup de langages, PLM n'a pas d'instruction `return`, qui permettrait de nommer la valeur de retour. En PLM, une variable nommée `result` est implicitement déclarée du type `Type`, et est non valuée initialement. La liste d'instructions de la fonction **doit** valuer cette variable. Sa valeur à l'issue de l'exécution de la liste d'instructions est la valeur renvoyée par la fonction. Par exemple :

```
func user maFonction @noUnusedWarning () -> UInt32 {
  result = 6
}
```

Ceci définit la fonction `maFonction`, sans argument, appelable uniquement en mode `user` ; l'attribut `@noUnusedWarning` signifie qu'aucune alerte n'est émise si la fonction n'est pas utilisée.

```
func user autreFonction (?arg :a UInt27 ?b UInt27) -> UInt27 {
  result = a +% b
}
```

Ceci définit la fonction `autreFonction`, avec deux arguments, sans attribut, appelable uniquement en mode `user`.

19.3.2 Déclaration d'une « procédure »

La déclaration d'une « procédure » est la suivante :

```
func mode nom @attribut (arguments_formels) {
  liste_instructions
}
```

Où :

- `nom` est le nom de la fonction ;
- `mode` est le mode associé à la fonction, c'est l'un des mots réservés `primitive`, `service`, `section` ou `safe` ;

- `@attribut` est une liste éventuellement vide d'attributs associés à la fonction ;
- `arguments_formels` est la liste (éventuellement vide) des paramètres formels.

Aucun type de retour n'est mentionné, aussi aucune variable nommée `result` n'est pré-définie. Une procédure est appelable dans une instruction.

19.3.3 Fonctions requises

La déclaration `required func` permet de signifier au compilateur qu'une procédure doit être définie, soit par la cible, soit par le programme utilisateur.

Cette déclaration est la suivante :

```
required func mode nom @attribut (arguments_formels) -> Type
```

ou :

```
required func mode nom @attribut (arguments_formels)
```

Elle consiste en la déclaration de l'en-tête d'une fonction, précédée par le mot réservé `required` .

19.3.4 Fonctions externes

Une fonction externe est une procédure définie en C ou en assembleur et appelable en PLM.

La déclaration d'une fonction externe est la suivante :

```
extern func mode nom (arguments_formels) -> Type  : "nom-assembleur"
```

Où :

- `nom` est le nom de la fonction externe ;
- `mode` est la liste non vide de l'ensemble des modes associés à la fonction ;
- `@type` est le type de la valeur renvoyée ;
- `"nom-assembleur"` est le nom qui sera engendré en assembleur, elle correspond à la directive `asm` de CLANG (et de GCC).

Si la fonction externe est en fait une *procédure*, sa déclaration est la suivante :

```
extern func mode nom (arguments_formels)  : "nom-assembleur"
```

19.3.4.1 Exemple de fonction externe

Prenons l'exemple de cette fonction, qui implémente le blocage de la tâche appelante dans une primitive :

```
extern func
primitive block (?!inList :ioWaitingList  : TaskList)  : "blockInList"
```

La signature PLM de cette fonction est `block(?!inList :)`.

Elle est appelable en mode `primitive`, c'est-à-dire dans une primitive. Dans l'instruction d'appel engendré en assembleur, le nom utilisé sera `func.blockInList`. Le préfixe `func.` est automatiquement ajouté par le compilateur.

Dans le code C qui implémente cette fonction, la déclaration du prototype est :

```
void block_in_list (TaskList * ioWaitingList)
asm ("!FUNC!blockInList") ;
```

Le nom `block_in_list` est interne au code C.

Le type C `TaskList` correspond au type PLM `TaskList` ; ces deux types sont définis indépendemment, c'est au programmeur de s'assurer qu'ils ont des tailles identiques (ici, 32 bits).

Le mode de passage de l'argument est spécifié par le sélecteur `?!inList :`, c'est le mode *entrée/sortie*, qui correspond au passage d'un pointeur en C. C'est au programmeur de s'assurer que les mode de passage spécifiés en PLM et C correspondent.

Enfin, le nom assembleur de la routine engendrée par le compilateur CLANG est spécifié par la deuxième ligne « `asm("!FUNC!blockInList")` ». Le nom « `blockInList` » doit être celui déclaré comme « *nom-assembleur* » dans la déclaration PLM, et « `!FUNC!` » est remplacé par « `func.` » par le compilateur PLM.

19.3.5 Attribut `@noUnusedWarning`

L'attribut `@noUnusedWarning` signifie qu'aucune alerte n'est émise si la fonction n'est pas utilisée.

19.3.6 Attribut `@exported`

L'attribut `@exported` signifie que la procédure sera visible par le code C et assembleur du projet. Ainsi une procédure écrite en PLM pourra être appelée à partir du code C et du code assembleur.

19.3.7 Fonctions « universelles »

Les fonctions universelles peuvent être appelées à partir de n'importe quel mode. Leur déclaration est la suivante, la notation de mode est simplement absente :

```
// Fonction universelle renvoyant un résultat
func nom @attribut (arguments_formels) -> Type {
  liste_instructions
}
// Procédure universelle
func nom @attribut (arguments_formels) {
  liste_instructions
}
```

Où :

- `nom` est le nom de la fonction ;
- `@attribut` est une liste éventuellement vide d'attributs associés à la fonction ;
- `arguments_formels` est la liste (éventuellement vide) des paramètres formels ;
- `@type` est le type de la valeur renvoyée.

En conséquence, les contraintes sur l'écriture de leurs instructions sont importantes :

- comme elles peuvent s'exécuter en mode **`user`**, seuls les registres de contrôles accessibles dans ce mode sont autorisés ;
- comme elles peuvent s'exécuter en mode `safe`, les instructions pouvant engendrer la panique ne sont pas autorisées.

19.3.8 Fonctions et registres de contrôle

[À REVOIR]

Si la fonction est déclarée avec le mode **`user`**, seuls les registres de contrôle déclarés `@user` sont accessibles. Dans les autres modes, tous les registres de contrôle sont accessibles.

19.4 Routines système

Une *routine système* est une routine qui s'exécute dans un des modes **`primitive`**, **`service`**, **`section`** ou `safe`. Elle est appelable à partir des modes :

- `user`, l'appel s'effectue alors à travers une instruction *Supervisor Call* ;
- d'un autre mode privilégié, en respectant le graphe de la figure 19.1 page 93 (l'appel est alors une instruction d'appel de sous-programme).

Comme une *routine système* est exécutée par le processeur en mode privilégié, les registres de contrôles accessibles.

19.4.1 Déclaration d'une routine système

La déclaration d'une routine système est la suivante :

```
system mode nom @attribut1 @attribut2 (arguments_formels) {
  liste_instructions
}
```

Où :

- `nom` est le nom de la routine système ;
- `mode` est le mode associé à la routine, c'est l'un des mots réservés `primitive`, `service`, `section` ou `safe` ;
- `@attribut1` `@attribut2` est une liste éventuellement vide d'attributs associés à la routine système ; actuellement, seul l'attribut `@noUnusedWarning` est défini (section 19.4.2 page 100).

19.4.2 Attribut `@noUnusedWarning`

L'attribut `@noUnusedWarning` signifie qu'aucune alerte n'est émise si la section n'est pas utilisée.

19.5 Routines d'interruption

La déclaration d'une routine d'interruption est la suivante :

```
system mode nom {
  liste_instructions
}
```

Où :

- `nom` est le nom de la routine ;

- `mode` est le mode associé à la routine, c'est l'un des mots réservés **`service`**, **`section`** ou `safe`.

Le nom de la routine doit être un des noms de routines d'interruption déclarés par la déclaration **`target`** (voir §§).

Le mode d'exécution associé est l'un des trois modes suivants :

- **`service`**, les routines de l'exécutif rendant des tâches prêtes peuvent être appelées ;
- **`section`**, la routine d'interruption n'interfère pas avec l'exécutif ;
- `safe`, comme pour **`section`**, et de plus les instructions pouvant engendrer la panique y sont interdites.

19.6 Routines utiles

Le compilateur élimine les routines qui ne sont jamais appelées, en calculant le graphe des appels. Les racines de ce graphe sont les procédures requises (section 19.3.3 page 97), les routines **`boot`**, **`init`** et **`panic`**. Les routines inatteignables sont éliminées, sans message d'alerte pour les routines déclarées avec l'attribut `@noUnusedWarning`.

19.7 Récursivité

Par défaut, le compilateur émet un message d'erreur si une ou plusieurs routines récursives sont détectées. L'option **`--do-not-detect-recursive-calls`** (section 3.2 page 32) permet d'inhiber cette recherche.

L'option **`--routine-invocation-graph`** permet d'obtenir un fichier contenant le graphe d'invocation, qui peut être affiché par le logiciel `graphviz`. Si le fichier source est `source.plm`, le fichier engendré s'appelle `source.subprogramInvocation.dot`.

Chapitre 20

Expressions

Priorité	Opérateur	Commentaire
0	`-`, `-%`	*moins* unaire
0	`~`, `not`	*complémentation* binaire et *non* logique
1	`convert`	Conversion
2	`*`, `*%`, `/`, `!/`, `%`, `!%`	Multiplication, division, modulo
3	`+`, `+%`, `-`, `-%`	Addition, soustraction
4	`<<`, `>>`	Décalage à gauche et à droite
5	`≤`, `<`, `>=`, `>`	Comparaison
6	`==`, `≠`	Test d'égalité, d'inégalité
7	`&`	*et* binaire
8	`^`	*ou exclusif* binaire
9	`|`	*ou* binaire
10	`and`	*et* logique
11	`xor`	*ou exclusif* logique
12	`or`	*ou* logique

Tableau 20.1 – *Priorité des opérateurs*

20.1 Opérateur ~

L'opérateur ~ renvoie la complémentation bit-à-bit d'un entier non signé. Une erreur de compilation est déclenchée si l'opérateur est appliqué à un entier signé :

```
let x Int8 = 3
let y = ~ x // Erreur, x est signé
```

Le nombre de bits complémentés dépend du nombre de bits du type entier non signé :

```
let x UInt8 = 1
let y = ~ x // y est égal à 0xFE
let z UInt16 = 1
let t = ~ z // t est égal à 0xFFFE
```

L'opérateur ~ ne peut s'appliquer à une constante entière statique uniquement si le type du résultat peut être inféré, et que ce type est un entier non signé :

```
let x = ~ 1 // Erreur, le type du résultat ne peut pas être inféré
let y Int8 = ~ 1 // Erreur, le type inféré est signé
let z UInt8 = ~ 1 // Ok, z = 0xFE
let t UInt16 = ~ 1 // Ok, z = 0xFFFE
```

20.2 Expression `if`

```
let x = if expression_1 { expression_2 } else { expression_3 }
```

L'expression `if` fonctionne comme suit :

- l' `expression_1` est une expression booléenne ;
- si l' `expression_1` est vraie, l' `expression_2` est calculée et sa valeur est celle renvoyée par l'expression `if` ;
- si l' `expression_1` est fausse, l' `expression_3` est calculée et sa valeur est celle renvoyée par l'expression `if` .

Les expressions `if` peuvent se succéder, dans tous les cas il faut terminer par une clause `else` :

```
let x =
  if expression_1 {
    expression_2
  }else if expression_3 {
    expression_4
  }else{
    expression_5
  }
```

20.3 Expression `addressof`

L'expression `addressof` permet d'obtenir l'adresse de toute *lvalue*. La valeur retournée a pour type l'entier non signé de la taille d'un pointeur (sur Cortex, c'est donc `UInt32`).

Par exemple :

```
var x = yes // x est booléen
let adresse UInt32 = addressof (x)
let adresse_registre_controle UInt32 = addressof (PORTA_PCR0)
```

20.4 Expression `sizeof`

L'expression `sizeof` permet d'obtenir la taille (en nombre d'octets) de toute *lvalue*, ou de tout type. La valeur retournée a pour type l'entier non signé de la taille d'un pointeur (sur Cortex, c'est donc `UInt32`).

Par exemple :

```
var x = yes // x est booléen
let s1 UInt32 = sizeof (x) // 1
let s2 UInt32 = sizeof (Bool) // 1
```

Chapitre 21

Instructions

21.1 Déclaration d'une variable locale

La déclaration d'une variable locale peut prendre plusieurs formes, suivant que la variable est initialisée ou non.

21.1.1 Déclaration d'une variable locale initialisée

```
var nom  : Type = expression
```

Où :

- `nom` est le nom de la variable locale ;
- `Type` est le nom du type de la variable globale ;
- `expression` est l'expression qui fournit la valeur initiale de cette variable ; cette expression peut être statique ou non.

L'annotation de type peut être omise ; la variable a alors le type de l'expression qui l'initialise :

```
var nom = expression
```

21.1.2 Déclaration d'une variable locale non initialisée

```
var nom  : Type
```

Où :

- `nom` est le nom de la variable locale ;
- `Type` est le nom du type de la variable globale.

Le compilateur garantit qu'aucune lecture n'est faite avant que la variable reçoive une valeur.

21.2 Déclaration d'une constante locale

La déclaration d'une constante locale apparaît dans une liste d'instructions et sa syntaxe est la suivante :

```
let nom  : Type = expression
```

Où :

- `nom` est le nom de la constante globale ;
- `Type` est le nom du type de la constante globale ;
- `expression` est l'expression qui fournit la valeur de cette constante ; cette expression est soit calculable statiquement, soit à l'exécution.

Il n'y a aucune restriction pour l'`expression` : elle peut nommer constantes locales et globales, variables locales et globales.

21.3 Instruction d'affectation

L'instruction d'affectation se présente sous plusieurs formes ; la plus simple est la suivante :

```
nom = expression
```

Où :

- `nom` est le nom d'une variable globale, d'une variable locale, ou d'un registre de contrôle ;
- `expression` est l'expression qui fournit la valeur.

On peut aussi accéder à une propriété d'une variable instance de structure :

```
nom.propriété = expression
```

Ainsi qu'à un élément de tableau :

```
nom [expression_indice] = expression
```

Ou encore toute combinaison de propriétés et d'éléments de tableau.

21.4 Opérateurs combinés avec l'affectation

Le tableau 21.1 liste les opérateurs combinés avec une affectation.

Opérateur combiné	Écriture équivalente	Lien
`a &= b`	`a = a & b`	section 8.9 page 50
`a |= b`	`a = a | b`	section 8.9 page 50
`a ∧= b`	`a = a ∧ b`	section 8.9 page 50
`a += b`	`a = a + b`	
`a +%= b`	`a = a +% b`	
`a -= b`	`a = a - b`	
`a -%= b`	`a = a -% b`	
`a *= b`	`a = a * b`	
`a *%= b`	`a = a *% b`	
`a /= b`	`a = a / b`	
`a !/= b`	`a = a !/ b`	
`a %= b`	`a = a % b`	
`a !%= b`	`a = a !% b`	
`a <<= b`	`a = a << b`	
`a >>= b`	`a = a >> b`	

Tableau 21.1 – *Opérateurs combinés avec l'affectation*

21.5 Instruction d'affectation « Bit banding »

La technique de « Bit banding » est implémentée sur certains processeurs pour pouvoir mettre à 0 ou à 1 un bit d'un registre, et ce de manière atomique. C'est par exemple le cas des Cortex-M4.

Considérons l'instruction suivante :

```
GPIOB.PDDR |= 1 << 16
```

Son exécution met à 1 le bit n°16 du registre `GPIOB.PDDR`. Mais cette opération n'est pas atomique sur un Cortex-M4. Si elle apparaît dans une fonction s'exécutant en mode utilisateur, elle peut être interrompue.

Pour la rendre atomique, on pourrait l'enfermer dans une `section`. En PLM, une autre possibilité est d'utiliser l'instruction de « Bit banding » :

```
{GPIOB.PDDR  : 16} = 1
```

La forme générale de cette instruction est :

```
{registre_contrôle  : numéro_bit} = expression_source
```

PLM limite le bit-banding à une zone de registres, aussi le premier argument `registre_contrôle` doit nommer un registre de contrôle. L'argument `numéro_bit` désigne le bit du registre par son numéro, c'est une expression du type `UInt5` pour un registre 32-bits, `UInt4` pour un registre 16-bits et `UInt3` pour un registre 8-bits. L'`expression_source` est de type `UInt1`, et donne la valeur à affecter au bit du registre.

21.6 Instruction de décomposition d'un entier non signé en tranches

Cette forme particulière d'affectation permet de décomposer une valeur entière non signée en tranches. Par exemple :

```
{UInt8 ?1 :var b1 ?2 :let b2 ?5 :let b3} = 0xCC // b1 <- 1, b3 <- 2, b3 <- 12
```

21.7 Instruction 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 21.8 page 109) 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 la panique est interdite.

Exemples :

```
check yes  // Ok
check no // Erreur, expression fausse
```

21.8 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 la panique activée, 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 panique (dont le code est donné par le tableau 27.1) est déclenchée.

Si le programme est compilé avec l'option `--no-panic-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 21.7 page 108. Par exemple, le code suivant provoque une erreur de compilation :

```
assert yes // Erreur de compilation, l'expression est calculable statiquement
```

21.9 L'instruction `panic`

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

```
panic expression
```

L'`expression` est une expression entière, calculée statiquement. Son type est défini pour chaque cible (section 28.2.4 page 147), c'est un type entier non signé.

Si le programme est compilé avec la panique activée, alors l'exécution de l'instruction **`panic`** déclenche la panique avec le code est la valeur de l'`expression`. Pour éviter un conflit de code avec les codes prédéfinis dans PLM, consulter le tableau 27.1 page 139.

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

21.10 Instruction d'appel de procédure

À définir

21.11 Instruction `if`

L'instruction `if` a une structure classique, où `condition` est une expression booléenne :

```
if condition {
  instructions_then
}else{
  instructions_else
}
```

Le compilateur vérifie que la `condition` n'est pas une expression statique : une erreur de compilation est émise si elle l'est.

La branche `else` est optionnelle :

```
if condition {
  instructions_then
}
```

Une ou plusieurs branches `else if` peuvent être ajoutées, avec ou sans branche `else` :

```
if condition {
  instructions_then
}else if condition2 {
  instructions_elsif
}else if condition3 {
  instructions_elsif_3
}else{
  instructions_else
}
```

21.12 Instruction `while`

L'instruction `while` permet d'exprimer une répétition, où la `condition` est testée avant l'exécution des instructions de la boucle :

```
while condition {
  instructions_while
}
```

`condition` est une expression booléenne, qui ne doit pas être statique : une erreur de compilation est émise si elle l'est.

21.13 Instruction `for`

21.13.1 Énumération entière

```
for nom  : Type in lower_bound ..< upper_bound {
  instructions_for
}
```

Si il n'est pas lu par les `instructions_for`, `nom` peut être remplacé par le joker « `_` ».

21.13.2 Énumération d'un tableau ou d'une chaîne de caractères

```
for nom in objet while expression {
  instructions_for
}
```

21.14 Instruction d'appel de routine

21.15 Instruction `switch`

21.16 Instruction `sync`

L'instruction **`sync`** est décrite à la section 24.7 page 133.

21.17 Instruction `nop`

```
nop
```

Le mot réservé **`nop`** engendre une instruction assembleur *nop*, telle que défini par le paramètre `NOP` dans la définition de la cible (section 28.2.7 page 148).

Chapitre 22

Pilotes

Un *pilote* PLM est un singleton, c'est-à-dire un type structure qui n'est être instancié qu'une fois et qui est visible globalement. C'est l'outil qui permet d'implémenter des pilotes matériels.

Chapitre 23

Tâches

En PLM, la tâche est l'unité d'exécution. Une tâche est déclarée statiquement, de priorité fixe. Une tâche peut déclarer des variables privées et du code à exécuter.

23.1 Un exemple de tâche

```
task T1 priority 1 stackSize 512 {
  var compteur UInt32 = 0

  setup 0 {
    lcd.print (!string :"Hello")
  }

  on time.waitUntilMS (!deadline :self.compteur) continue {
    digitalWrite (!port :LED_L0 !yes)
    self.compteur +%= 500
    time.waitUntilMS (!deadline :self.compteur)
    digitalWrite (!port :LED_L0 !no)
    self.compteur +%= 500
  }
}
```

23.2 Déclaration d'une tâche

L'en-tête de la déclaration d'une tâche définit :

- son nom ;
- sa priorité ;
- la taille de sa pile.

Toutes les priorités doivent être différentes ; 0 est la plus forte priorité.

Par exemple :

```
task T priority 12 stackSize 512 {
  ...
}
```

Peuvent être déclarés dans le corps d'une tâche :

- des variables privées ;
- des routines d'initialisation (`setup`) ;
- des fonctions ;
- des commandes gardées.

La déclaration d'une variable privée doit comporter une expression qui fixe sa valeur initiale (ainsi toutes les variables privées d'une tâche sont initialisées lorsqu'elle démarre) ; cette expression doit être calculable statiquement. Ces variables sont privées, c'est-à-dire qu'aucune autre entité extérieure (comme par exemple une autre tâche) ne peut y accéder. L'accès aux variables privées doit être obligatoirement préfixé par `self.` .

Une tâche peut déclarer zéro, une ou plusieurs routines d'initialisation. Chacune présente une priorité (exprimée sous la forme d'un nombre positif ou nul). Deux routines d'initialisation d'une même tâche ne peuvent pas avoir la même priorité. Les routines d'initialisation sont exécutées dans l'ordre croissant de leur priorité, et en mode utilisateur (section 19.1 page 91). Une routine d'initialisation peut servir par exemple à donner une valeur initiale calculée dynamiquement à une variable privée, ou encore à réaliser des initialisations matérielles.

Une tâche peut déclarer zéro, une ou plusieurs gardes. Une garde exprime l'attente qu'une condition de synchronisation se réalise. Si une tâche ne définit aucune garde, alors son exécution se termine à la fin de l'exécution des routines d'initialisation `setup` .

Enfin, une tâche peut déclarer des fonctions privées.

23.3 Extensions

23.4 Exécution des tâches

Les tâches sont toutes démarrées à la fin de la phase d'initialisation (figure 5.1 page 39)[1]. Après avoir exécuté ses routines `setup` (dans l'ordre croissant des valeurs associées), la tâche se met en attente des gardes tant qu'aucune n'est vraie. Dès que l'une d'elles devient vraie, sa liste d'instructions est exécutée ; par défaut, la tâche se remet en attente de l'évolution des gardes. Si la garde nomme le qualificatif `exit`, l'exécution est terminée.

La figure 23.1 illustre l'exécution de la tâche `T` suivante :

```
task T priority 1 stackSize 512 {
  setup 1 { ... }
  on G1 exit { L1 }
  on G2 { L2 }
}
```

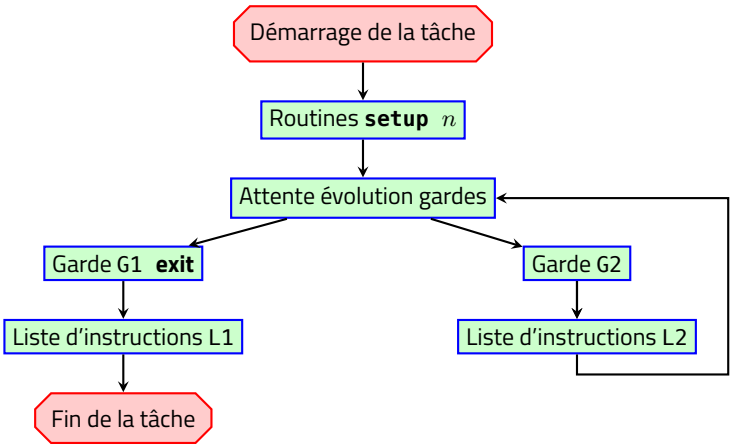


Figure 23.1 – *Organigramme d'exécution d'une tâche*

1. Si aucune tâche n'est déclarée, l'exécution s'arrête à la fin de l'exécution des routines **init**.

Chapitre 24

Synchronisation et communication

PLM ne définit pas d'outils de synchronisation particulier, mais propose des briques permettant de les construire. Ces briques sont des types opaques et des fonctions.

Ce chapitre présente :

- les types et fonctions constituant ces briques élémentaires (section 24.1) ;
- comment écrire un pilote de gestion du temps sur un processeur Cortex-M4 (section 24.2 page 120) ;
- l'implémentation du sémaphore de Dijkstra (section 24.3 page 124) ;
- l'implémentation d'un rendez-vous (section 24.4 page 127) ;
- l'implémentation d'un chronomètre permettant d'exprimer simplement une exécution périodique (section 24.5 page 130) ;
- l'implémentation d'une porte de synchronization (section 24.6 page 131).

24.1 Types et fonctions prédéfinis

PLM définit deux types opaques et neuf fonctions externes[1] qui permettent d'écrire outils de synchronisation et gardes.

1. Externes à PLM, car écrites en C.

24.1.1 Type opaque `TaskList`

Le type opaque `TaskList` est utilisé pour maintenir la liste des tâches bloquées sur un outil de synchronisation.

24.1.2 Type opaque `GuardList`

Le type opaque `GuardList` est utilisé pour maintenir la liste des tâches qui ont invoqué une garde d'un outil de synchronisation.

24.1.3 Fonction `block(?!inList :)`

```
func primitive block (?!inList :ioWaitingList  : TaskList)
```

Cette fonction bloque la tâche en cours dans la liste `ioWaitingList` passée en argument. Elle est appelable en mode `primitive`.

24.1.4 Fonction `block(?onDeadline :)`

```
func primitive block (?onDeadline :deadline UInt32)
```

Cette fonction bloque la tâche en cours jusqu'à la date `deadline`. Elle est appelable en mode `primitive`.

24.1.5 Fonction `block(?!inList :?onDeadline :)`

```
func primitive block (?!inList :waitingList  : TaskList
                      ?onDeadline :deadline  : UInt32
                      !result : outResult  : Bool)
```

Cette fonction bloque la tâche en cours dans la liste `waitingList` jusqu'à la date `deadline`. Elle est appelable en mode `primitive`. L'argument de retour `outResult` vaut `yes` si la tâche a été rendue prête via la liste `waitingList`, et `no` si si elle a été rendue prête parce que son échéance `deadline` a été atteinte.

24.1.6 Fonction `makeTaskReady(?!fromList :)`

```
func service makeTaskReady (?!fromList :ioWaitingList TaskList
                            !found : outFound Bool)
```

Si la liste `waitingList` est vide, la fonction renvoie `outFound` à `no`. Si la liste n'est pas vide, cette fonction retire de la liste la tâche de plus forte priorité, la rend prête, et la fonction renvoie `outFound` à `yes`.

Cette fonction est appelable en mode `primitive`, c'est-à-dire par les primitives, et en mode `service`, c'est-à-dire par les routines d'interruption.

24.1.7 Fonction `makeTasksReady(?fromCurrentDate :)`

```
func service makeTasksReady (?fromCurrentDate :inCurrentDate UInt32)
```

Cette fonction rend prête toutes les tâches en attente d'échéance dont l'échéance est atteinte.

Cette fonction est appelable mode `service`, c'est-à-dire par les routines d'interruption.

24.1.8 Fonction `handleGuardedCommand`

```
func guard handle (?!guard :ioGuard GuardList)
```

Cette fonction enregistre la tâche courante dans la liste `ioGuard`. Cette fonction est appelable mode `guard`, c'est-à-dire par les gardes.

24.1.9 Fonction `handle (?guardedDeadline :)`

```
func guard handle (?guardedDeadline :deadline UInt32)
```

Cette fonction enregistre la tâche courante pour une attente en garde jusqu'à la date `deadline`. Cette fonction est appelable mode `guard`, c'est-à-dire par les gardes.

24.1.10 Fonction `notifyChange`

```
func service notifyChange (?!forGuard :ioGuard GuardList)
```

Cette fonction signifie aux tâches enregistrées dans `ioGuard` que les gardes doivent être re-évaluées. Cette fonction est appelable en mode `primitive` et en mode `service`.

24.1.11 Fonction `notifyChangeForGuardedWaitUntil(?withCurrentDate :)`

```
func service
notifyChangeForGuardedWaitUntil (?withCurrentDate :inCurrentDate UInt32)
```

Cette fonction signifie aux tâches en attente d'échéance en garde que l'instant `inCurrentDate` est atteint, et qu'elles doivent re-évaluer leur gardes. Cette fonction est appelable en mode **`service`** (et aussi en mode **`primitive`**, bien qu'en pratique c'est la routine d'interruption périodique qui l'appelle).

24.2 Le pilote `time` pour un Cortex-M4

Le pilote `time` décrit ici permet une gestion élémentaire du temps, c'est-à-dire :

- maintenir une date courante, incrémentée sur interruption ;
- une fonction permettant d'acquérir la date courante ;
- une primitive d'attente d'échéance de la date courante ;
- la définition d'une garde exprimant l'attente d'échéance en garde.

Les Cortex-M4 implémentent le timer *Systick* qui permet de programmer très simplement des interruptions périodiques. La période choisie est $1\ ms$. Le comptage du temps est initialisé à 0 au démarrage du micro-contrôleur. Il est mémorisé dans une variable de type `UInt32`, ce qui signifie qu'il retombe à 0 au bout de $2^{32}\ ms$, c'est-à-dire un peu de 49 jours[2].

24.2.1 En-tête

L'en-tête du pilote `time` déclare la variable privée `mUptime` qui est utilisée pour détenir la date courante.

```
driver time {
  var mUptime UInt32 = 0
```

24.2.2 Initialisation

La routine **`init`** `0` est exécutée durant la phase d'initialisation, c'est-à-dire après les routines de **`boot`** et l'initialisation des variables globales (section 5.1 page 39). Elle programme simplement le timer *SysTick* de façon qu'il engendre une interruption toutes les millisecondes. La constante `F_CPU_MHZ` contient

2. Il n'y a pas d'impossiblité technique de passer à une variable plus large, par exemple un entier 64 bits. Il faut uniquement que les routines concernées de l'exécutif, écrites en C, prennent en charge ce type. Ces fonctions sont : `makeTasksReady(?fromCurrentDate :)`, `handle (?guardedDeadline :)` et `notifyChangeForGuardedWaitUntil(?withCurrentDate :)`.

la fréquence du processeur, exprimée en nombre de MHz : l'horloge du processeur est aussi l'horloge du timer *SysTick*. Il faut noter que les interruptions sont toujours masquées durant la phase d'initialisation, aussi, bien que le timer tourne, la première interruption sera prise en compte au moment du démarrage des tâches.

```
init 0 { // Configure Systick interrupt every ms
  SYST_RVR = F_CPU_MHZ * 1000 - 1 // Interrupt every ms
  SYST_CVR = 0
  SYST_CSR = {SYST_CSR !CLKSOURCE :1 !ENABLE :1 !TICKINT :1}
}
```

24.2.3 La routine d'interruption

La routine d'interruption est `systick`. Son nom est défini par le fichier de configuration de la cible (section 28.2.15 page 154). Le mot réservé **`service`** dans son en-tête signifie qu'elle peut appeler les fonctions déclarées dans ce mode. L'appel de la fonction `noteCurrentTaskFreeStackSize` permet de maintenir une information de remplissable de la tâche en cours d'exécution au moment où l'interruption est survenue. Ensuite, la variable **`self`**`.mUptime` est incrémentée. Noter l'utilisation de l'opérateur `+%`, qui réalise une incrémentation sans détection de débordement. La fonction `makeTasksReady (!fromCurrentDate :)` libère toutes les tâches bloquées en attente d'échéance, dont l'échéance est atteinte (c'est-à-dire dont l'échéance est inférieure ou égale à la valeur de `now`). La fonction `notifyChangeForGuardedWaitUntil (!withCurrentDate :)` fait de même pour les tâches en attente d'échéance en garde.

```
isr service systick {
  noteCurrentTaskFreeStackSize ()
  let now = self.mUptime +% 1
  self.mUptime = now
  makeTasksReady (!fromCurrentDate :now)
  notifyChangeForGuardedWaitUntil (!withCurrentDate :now)
}
```

24.2.4 Obtenir la date courante

L'appel système `now` retourne la date courante.

```
public system safe now @noUnusedWarning () -> UInt32 {
  result = self.mUptime
}
```

24.2.5 Attente d'échéance

Cette primitive système exprime l'attente d'échéance :

- `public` signifie qu'elle peut être appelée en dehors du pilote ;
- `system` qu'elle est appelable à partir d'une tâche, et s'exécute interruptions masquées via un *system call* ;
- `primitive` que la tâche appelante peut être bloquée ;
- `@noUnusedWarning` qu'aucune alerte ne sera engendrée par la compilation si cette fonction n'est pas appelée.

Si la date fournie par la valeur `inDate` est postérieure à la date courante, la tâche appelante est bloquée.

```
public system primitive
wait @noUnusedWarning (?untilDeadline : inDate UInt32) {
  if inDate > self.mUptime {
    block (!onDeadline :inDate)
  }
}
```

L'appel de la primitive a pour syntaxe :

```
var échéance UInt32 = ...
time.wait (!untilDeadline :échéance)
```

24.2.6 Attente de délai

L'implémentation de l'attente de délai est similaire à l'attente d'échéance : le délai est ajouté à la date courante pour former une date absolue.

```
public system primitive
wait @noUnusedWarning (?duringDelay : inDelay UInt32) {
  if inDelay > 0 {
    block (!onDeadline :self.mUptime +% inDelay)
  }
}
```

24.2.7 Commande gardée d'attente d'échéance

L'implémentation d'une commande gardée prédéfinit la variable booléenne `accept`. Celle-ci doit être valuée par la liste d'instructions, et à l'issue de l'exécution de celle-ci :

- si la valeur de `accept` est `yes`, la condition d'attente est considérée comme satisfaite (c'est-à-dire ici, que la date d'échéance `deadline` est postérieure à la date courante) ;
- si la valeur de `accept` est `no`, la condition d'attente est considérée comme non satisfaite (c'est-à-dire ici, que la date d'échéance `deadline` est antérieure à la date courante), et que la tâche appelante a été enregistrée par un appel à la fonction `handle (!guardedDeadline :)`.

```
public guard wait @noUnusedWarning (?untilDeadline :deadline UInt32) {
  noteCurrentTaskFreeStackSize ()
  accept = (deadline) ≤ self.mUptime
  if not accept {
    handle (!guardedDeadline :deadline)
  }
}
```

24.2.8 Attente en mode `init`

Le pilote `time` est configuré par la routine `init` de priorité 0, elle est donc la première à s'exécuter dans ce mode. Les autres routines `init` s'exécutent donc ensuite, dans le pilote `time` est configuré. Dans le mode `init`, l'exécutif n'est pas démarré, on ne peut pas appeler les routines d'attente d'échéance et de délai, mais uniquement exécuter que des attentes actives.

Les fonctions suivantes réalisent des attentes actives, et, comme elles sont déclarées dans le mode `init`, elles ne peuvent pas être appelées par les tâches.

```
public func init oneMillisecondBusyWait @noUnusedWarning () {
  while not SYST_CSR.COUNTFLAG.bool {}
}

public func panic panicOneMillisecondBusyWait @noUnusedWarning () {
  while not SYST_CSR.COUNTFLAG.bool {}
}

public func init busyWaitingDuringMS @noUnusedWarning (?inDelay UInt32) {
  for _ UInt32 in 0 ..< inDelay {
    self.oneMillisecondBusyWait ()
```

```
  }
}
```

24.2.9 Attente en mode `panic`

En mode `panic`, l'exécutif n'est plus actif, on ne peut qu'exécuter des attentes actives.

```
public func panic
panicBusyWaitingDuringMS @noUnusedWarning (?inDelay UInt32) {
  for _ UInt32 in 0 ..< inDelay {
    self.panicOneMillisecondBusyWait ()
  }
}
```

24.3 Sémaphore de Dijkstra

Le sémaphore de Dijkstra est implémenté comme une structure et la synchronisation est réalisée en appelant les fonctions de l'exécutif.

24.3.1 Version sans primitive d'attente en garde

```
struct Semaphore {
  var value UInt32
  var list = TaskList ()

  public system service signal @noUnusedWarning @mutating () {
    makeTaskReady (!?fromList :self.list ?found :let found)
    if not found {
      self.value += 1
    }
  }

  public system primitive wait @noUnusedWarning @mutating () {
    if self.value > 0 {
      self.value -= 1
    }else{
      block (!?inList :self.list)
    }
```

```
  }
}
```

Un sémaphore est constitué de deux propriétés :

- `value`, sa valeur qui est un entier positif ou nul ;
- `list`, la liste des tâches bloquées sur ce sémaphore.

Instanciation. L'instanciation d'un sémaphore doit fournir sa valeur initiale ; par exemple :

```
var s = Semaphore (!value :1)
```

Routine `signal`. Elle est déclarée en mode `service`, c'est-à-dire qu'elle peut être appelée à partir des tâches, et des routines d'interruption. Elle appelle la routine `makeTaskReady(!?fromList :?found :)` qui fait l'opération suivante :

- si la liste `self.list` est vide, alors le booléen `found` est retourné à `no` ;
- dans le cas contraire, la tâche la plus prioritaire est retirée de la liste `self.list` et est rendue *prête*, et le booléen `found` est retourné à `yes`.

Ensuite, la valeur du sémaphore est incrémentée si la liste `self.list` était initialement vide.

Routine `wait`. Elle est déclarée en mode `primitive`, c'est-à-dire seules les tâches peuvent l'appeler. Elle commence par tester la valeur du sémaphore ; si il est strictement positif, il est décrémenté, et aucun blocage n'a lieu. Si il est nul, la tâche appelante est bloquée dans la liste `self.list` par l'appel de `block(!?inList :)`.

24.3.2 Ajout de l'attente jusqu'à une échéance

```
public system primitive wait
@noUnusedWarning @mutating (?untilDeadline : deadline UInt32
                            !result : outResult Bool) {
  outResult = self.value > 0
  if outResult {
    self.value -= 1
  }else if deadline > time.now () {
    block (!?inList :self.list !onDeadline :deadline !?result :outResult)
  }
}
```

Cette primitive permet d'attendre sur un sémaphore jusqu'à ce qu'une échéance soit atteinte. Son appel renvoie dans `outResult` :

- `yes` si le sémaphore a permis le passage ;
- `no` si l'échéance a été atteinte.

L'exécution de la primitive teste successivement :

- la valeur du sémaphore ; s'il est strictement positif, il est décrémenté et la primitive retourne immédiatement avec `outResult` à `yes` ;
- la valeur de l'argument `deadline` ; si l'échéance est atteinte, la primitive retourne immédiatement avec `outResult` à `no`.

Si la valeur du sémaphore est nulle et que l'échéance n'est pas atteinte, la tâche est bloquée par l'appel à `block(!?inList :!onDeadline :!?result :)`.

Attention. L'appel à `block(!?inList :!onDeadline :!?result :)` ne modifie pas la valeur de l'argument `outResult` : seule son adresse est notée, de façon à pouvoir lui affecter ultérieurement la valeur `yes` quand le déblocage intervient via le sémaphore, ou la valeur `no` quand l'échéance est atteinte. La variable `outResult` **doit** être un alias d'une variable de la tâche, et non pas une variable locale de la primitive.

24.3.3 Version avec primitive d'attente en garde

Pour implémenter une primitive d'attente en garde, deux modifications de l'existant sont nécessaires :

- ajouter une propriété `guardList` qui va détenir l'état du sémaphore par rapport aux commandes gardées ;
- modifier la primitive `signal()` de façon à signaler aux commandes gardées l'évolution de la valeur du sémaphore.

La déclaration des propriétés des sémaphores devient donc :

```
struct Semaphore {
  var value UInt32
  var list = TaskList ()
  var guardList = GuardList ()
  ...
}
```

Et la primitive `signal()` appelle `notifyChange(!?forGuard :)` si le sémaphore est incrémenté :

```
public system service signal @noUnusedWarning @mutating () {
    makeTaskReady (!?fromList :self.list ?found :let found)
    if not found {
      self.value += 1
      notifyChange (!?forGuard :self.guardList)
    }
  }
```

On peut maintenant écrire la garde `wait()`. D'une manière générale, la variable `accept` est implicitement déclarée, de type `Bool`, et doit être valuée par la liste d'instructions, à `yes` si la condition d'acceptation de la garde est statisfaite, et `no` s'il ne l'est pas.

Ici, la condition d'acceptation est que le sémaphore soit strictement positif. Si il l'est, il est décrémenté, sinon, le refus est noté par l'appel de `handle(!?guard :)`.

```
public guard wait @noUnusedWarning () {
  accept = self.value > 0
  if accept {
    self.value -= 1
  }else{
    handle (!?guard :self.guardList)
  }
}
```

Les gardes ont leur propre espace de nommage. Aussi il n'y a pas d'ambiguïté entre la primitive `wait()` et la garde `wait()`.

24.4 Rendez-vous

Comme pour le sémaphore de Dijkstra, nous allons d'abord présenter une implémentation sans les attentes temporisées ni les gardes. Comme il n'y a pas de transmission de données, les noms `input` et `output` sont arbitraires et traduisent seulement la dissymétrie des primitives.

24.4.1 Primitives de base

```
struct RendezVous {
  var inputWaitList = TaskList ()
  var outputWaitList = TaskList ()
```

```
  public system primitive input @mutating () {
    makeTaskReady (!?fromList :self.outputWaitList ?found :let found)
    if not found {
      block (!?inList :self.inputWaitList)
    }
  }

  public system primitive output @mutating () {
    makeTaskReady (!?fromList :self.inputWaitList ?found :let found)
    if not found {
      block (!?inList :self.outputWaitList)
    }
  }
```

24.4.2 Ajout des attentes temporisées

```
public system primitive
inputUntil @noUnusedWarning @mutating (?deadline :deadline UInt32
                                       !result : result Bool) {
  makeTaskReady (!?fromList :self.outputWaitList ?found :result)
  if (not result) and (deadline > time.now ()) {
    block (!?inList :self.inputWaitList !onDeadline :deadline !?result :result)
  }
}

public system primitive
outputUntil @noUnusedWarning @mutating (?deadline :deadline UInt32
                                        !result : result Bool) {
  makeTaskReady (!?fromList :self.inputWaitList ?found :result)
  if (not result) and (deadline > time.now ()) {
    block (!?inList :self.outputWaitList !onDeadline :deadline !?result :result)
  }
}
```

24.4.3 Ajout des gardes

```
struct RendezVous {
  var inputWaitList = TaskList ()
  var outputWaitList = TaskList ()
  var inputGuardList = GuardList ()
```

```
var outputGuardList = GuardList ()

public system primitive input @mutating () {
  makeTaskReady (!?fromList :self.outputWaitList ?found :let found)
  if not found {
    notifyChange (!?forGuard :self.outputGuardList)
    block (!?inList :self.inputWaitList)
  }
}

public system primitive output @mutating () {
  makeTaskReady (!?fromList :self.inputWaitList ?found :let found)
  if not found {
    notifyChange (!?forGuard :self.inputGuardList)
    block (!?inList :self.outputWaitList)
  }
}

public system primitive
inputUntil @noUnusedWarning @mutating (?deadline :deadline UInt32
                                       !result : result Bool) {
  makeTaskReady (!?fromList :self.outputWaitList ?found :result)
  if (not result) and (deadline > time.now ()) {
    block (!?inList :self.inputWaitList !onDeadline :deadline !?result :result)
  }
}

public system primitive
outputUntil @noUnusedWarning @mutating (?deadline :deadline UInt32
                                        !result : result Bool) {
  makeTaskReady (!?fromList :self.inputWaitList ?found :result)
  if (not result) and (deadline > time.now ()) {
    block (!?inList :self.outputWaitList !onDeadline :deadline !?result :result)
  }
}

public guard input @noUnusedWarning () {
  makeTaskReady (!?fromList :self.outputWaitList ?found :accept)
  if not accept {
    handle (!?guard :self.inputGuardList)
  }
```

```
  }

  guard output @noUnusedWarning () {
    makeTaskReady (!?fromList :self.inputWaitList ?found :accept)
    if not accept {
      handle (!?guard :self.outputGuardList)
    }
  }

}
```

24.5 Chronomètre périodique

La structure `PeriodicTimer` est un outil de synchronisation qui permet d'exprimer simplement des exécutions périodiques.

24.5.1 Implémentation

La structure `PeriodicTimer` définit trois propriétés :

- `deadline`, qui détient la date de la prochaine échéance ;
- `period`, la période du chronomètre, qui est une constante fixée lors de l'initialisation ;
- `guardList`, pour gérer l'attente en garde.

La primitive `wait` teste l'échéance avec la date courante, et effectue le blocage de la tâche appelante si l'échéance n'est pas atteinte. L'échéance est toujours incrémentée de la valeur de la période dans les deux situations – date courant atteinte ou non.

De même, la garde `wait` teste l'échéance avec la date courante. Si elle est atteinte (`accept` est à `yes`), l'échéance est incrémentée de la valeur de la période ; si elle n'est pas atteinte, l'appel de la fonction `handle(!?guard :)` enregistre la garde.

```
struct PeriodicTimer {
  var deadline UInt32 = 0
  let period UInt32
  var guardList = GuardList ()

  public system primitive wait @noUnusedWarning @mutating () {
    if self.deadline ≤ time.now () {
```

```
      block (!onDeadline :self.deadline)
    }
    self.deadline += self.period
  }

  public guard wait @noUnusedWarning () {
    accept = self.deadline ≤ time.now ()
    if accept {
      self.deadline += self.period
    }else{
      handle (!?guard :self.guardList)
    }
  }

  public system section deadline @noUnusedWarning () -> UInt32 {
    result = self.deadline
  }
}
```

24.5.2 Exemple d'utilisation

```
task Tâche priority 1 stackSize 512 {
  var deadline = PeriodicTimer (!period :500)


  on self.deadline.wait () {
    ...
  }
}
```

24.6 Porte de synchronisation

Une *porte de synchronisation* est soit ouverte, soit fermée. Quand le porte est ouverte, la primitive `wait` est non bloquante. Quand elle est fermée, la primitive `wait` est systématiquement bloquante. Ouvrir la porte (service `open`) libère toutes les tâches bloquées. Fermer la porte (service `close`) ne provoque pas d'action sur la liste de tâches bloquées (celle-ci est forcément vide), ni sur la liste des gardes.

```
struct SynchronizationGate {
  var isOpen Bool
```

```
  var taskList = TaskList ()
  var guardList = GuardList ()

  public system service close @noUnusedWarning @mutating () {
    self.isOpen = no
  }

  public system service open @noUnusedWarning @mutating () {
    if not self.isOpen {
      var continue = yes
      while continue {
        makeTaskReady (!?fromList :self.taskList ?found :continue)
      }
      notifyChange (!?forGuard :self.guardList)
      self.isOpen = yes
    }
  }

  public system primitive wait @noUnusedWarning @mutating () {
    if not self.isOpen {
      block (!?inList :self.taskList)
    }
  }

  public system primitive wait
  @noUnusedWarning @mutating (?untilDeadline :deadline UInt32
                              !result : result Bool) {
    result = self.isOpen
    if result {
    }else if deadline > time.now () {
      block (!?inList :self.taskList !onDeadline :deadline !?result :result)
    }
  }

  public guard wait @noUnusedWarning () {
    accept = self.isOpen
    if not accept {
      handle (!?guard :self.guardList)
    }
  }
}
```

24.7 Instruction sync

24.8 Implémentation des commandes gardées

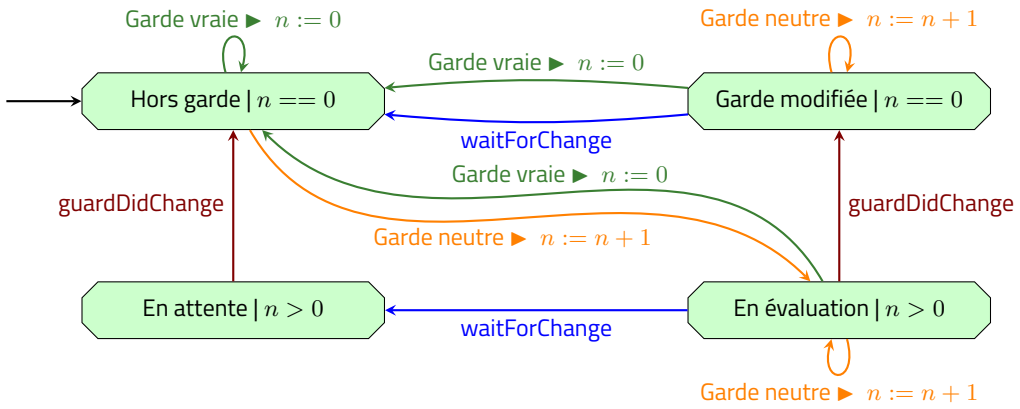


Figure 24.1 – *Graphe d'état des gardes d'une tâche*

Chapitre 25

Contrôle d'accès

25.1 Routines boot

Les routines `boot` n'ont pas le droit d'appeler des pilotes ou de solliciter les points d'entrée d'une tâche.

En effet, les routines `boot` sont exécutées avant l'initialisation des propriétés des pilotes, et les tâches ne sont pas démarrées.

25.2 Propriétés

Les propriétés des structures sont par défaut privées ; le qualificatif `public` les rend publiques.

Les propriétés des pilotes et des tâches sont toujours privées.

25.3 Méthodes

Le contrôle d'accès suivant s'applique aux :

- méthodes de structure ;
- méthodes de pilote ;
- méthode de tâche.

Une méthode déclarée sans mode ou avec le mode `user` ne peut accéder qu'aux propriétés constantes.

25.4 Registres de contrôles

Si un registre de contrôle est déclaré avec l'attribut `@ro`, il ne peut pas être modifié.

Si un registre de contrôle est déclaré avec l'attribut `@user`, il peut être accédé en mode `user`.

Chapitre 26

Directives

26.1 Directive `target`

La directive `target` fixe la cible pour laquelle le code source est compilé. Sa syntaxe est la suivante :

```
target "cible"
```

Où `"cible"` est le nom du descriptif de la cible.

26.2 Directive `import`

La directive `import` permet d'ajouter les définitions contenues dans le fichier texte nommé. Sa syntaxe est la suivante :

```
import "chemin.plm"
```

Où `"chemin.plm"` est un chemin (absolu, relatif) vers le fichier à importer.

Importer plusieurs fois le même fichier n'est pas une erreur. Le compilateur garde trace des importations déjà effectuées et ignore les importations des fichiers déjà importés.

L'extension `.plm` doit être mentionnée dans le chemin.

26.3 La directive `check target`

La directive `check target` permet de vérifier l'identité de la cible. Elle est utile pour s'assurer qu'un fichier est acceptable par une cible. À la compilation, toute directive `check target` incorrecte entraîne une erreur.

Par exemple, compiler pour la cible `"teensy-3-1"` est spécifié par :

```
target "teensy-3-1"
```

Si la définition du projet implique d'importer le fichier `"monFichier.plm"` :

```
target "teensy-3-1"
import "monFichier.plm"
```

Si le contenu de ce fichier commence par :

```
check target "lpc2294"
```

Alors une erreur de compilation apparaît. La directive `check target` doit être :

```
check target "teensy-3-1"
```

Chapitre 27

Panique organisée

Une panique peut survenir dans deux situations :

- exécution d'une opération provoquant la panique ;
- déclenchement d'une interruption pour laquelle aucune routine n'a été définie.

Une panique est caractérisée par trois informations :

- son code ;
- le nom du fichier source de l'opération qui a levé la panique ;
- le numéro de ligne du fichier source de l'opération qui a levé la panique.

Les types du code et du numéro de ligne sont définies dans la configuration de la cible (section « *PANIC* » page 147).

27.1 Exécution d'une opération provoquant la panique

Le code correspondant est défini dans le tableau 27.1.

27.2 Occurrence d'une interruption sans routine associée

Le déclenchement d'une interruption pour laquelle aucune routine n'a été définie provoque la panique :

Code	Signification	Lien
1	Échec de l'instruction `assert`	section 21.8 page 109
2	Dépassement de capacité de l'addition non signée (`+`)	section 8.4 page 48
3	Dépassement de capacité de l'addition signée (`+`)	section 8.4 page 48
4	Dépassement de capacité de la soustraction non signée (`-`)	section 8.4 page 48
5	Dépassement de capacité de la soustraction signée (`-`)	section 8.4 page 48
6	Dépassement de capacité de la multiplication non signée (`*`)	section 8.4 page 48
7	Dépassement de capacité de la multiplication signée (`*`)	section 8.4 page 48
8	Division non signée par zéro (`/`)	section 8.4 page 48
9	Division signée par zéro (`/`)	section 8.4 page 48
10	Modulo non signé par zéro (`%`)	section 8.4 page 48
11	Modulo signé par zéro (`%`)	section 8.4 page 48
12	Dépassement de capacité d'une conversion entre entiers (`convert`)	section 8.2.2 page 47
13	Dépassement de capacité d'un champ de registre de contrôle	
14	Indice de tableau négatif	section 15.5.3 page 75
15	Indice de tableau supérieur ou égal à la taille du tableau	section 15.5.2 page 75, section 15.5.3 page 75
16	Instruction `sync` invalide	

Tableau 27.1 – *Code des paniques*

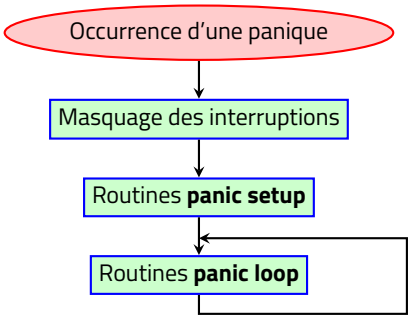


Figure 27.1 – *Organigramme de la réponse à une panique*

- son code est le numéro associé à l'interruption, comme définie à la section « *INTERRUPTS* » page 154 ;
- son numéro de ligne est 0 ;
- le nom de fichier source associée est la chaîne vide.

27.3 Routines exécutées lors de l'occurrence d'une panique

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

- les interruptions sont masquées, si elles ne le sont pas déjà ;

- les routines `panic setup` sont exécutées une fois ;
- les routines `panic loop` sont exécutées indéfiniment.

Ce fonctionnement est illustré à figure 27.1.

Si plusieurs routines de panique `panic setup` sont définies, celles-ci sont exécutées dans l'ordre de leurs priorités relatives. Ces routines offrent l'opportunité d'agir sur les sorties du micro-contrôleur, et d'afficher les caractéristiques de la panique.

Si plusieurs routines de panique `panic loop` sont définies, celles-ci sont exécutées dans l'ordre de leurs priorités relatives. Ces routines permettent de signaler d'une manière répétitive l'occurrence d'une panique.

27.3.1 Routines de panique `setup` et `loop`

Leur syntaxe est la suivante :

```
panic nom priorite {
  liste_instructions
}
```

`nom` est soit `setup`, soit `loop`. `priorite` est une constante entière, comprise entre 0 et $2^{64}-1$. Si il y a plusieurs routines de panique de même nom, elles sont exécutées dans l'ordre des priorités croissantes. Le compilateur vérifie que deux routines de panique de même nom n'ont pas la même priorité.

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

Trois constantes sont prédéfinies :

- `CODE`, qui contient le code de panique, et dont le type est défini par la construction `panic :` (section 28.2.4 page 147) ;
- `FILE`, qui contient le nom du fichier source de l'instruction qui a déclenché la panique, et dont le type est `LiteralString` ;
- `LINE`, qui contient le numéro de ligne du fichier source de l'instruction qui a déclenché la panique, et dont le type est défini par la construction `PANIC :` (section 28.2.4 page 147).

Les trois constantes `CODE`, `FILE` et `LINE` permettent de signaler les caractéristiques de la panique.

27.4 Exemples

Pour redémarrer un processeur ARMv7 lors d'une panique, on peut écrire la routine `panic setup` suivante :

```
panic setup 255 {
  AIRCR = {AIRCR !VECTKEY :0x5FA !SYSRESETREQ :1}
}
```

Chapitre 28

Définition d'une cible

Dans ce chapitre, nous allons voir comment est définie une cible, en prenant pour exemple la cible `teensy-3-1/unprivileged`.

Si vous voulez définir votre propre cible, la lecture de ce chapitre est indispensable.

Le compilateur recherche la cible citée :

- par défaut, parmi les fichiers embarqués dans le compilateur PLM ;
- si l'option **-T=***repertoire* (voir section 3.7 page 34) est présente dans la ligne de commande, dans les fichiers du répertoire $cibles$.

Pour explorer les fichiers indispensables à la définition d'une cible, il est nécessaire de les placer dans le système de fichiers. La première section de ce chapitre (section 28.1 page 142) va donc indiquer comment extraire de l'exécutable les fichiers de définition des cibles, et comment les exploiter.

28.1 Utilisation d'une cible dans le système de fichiers

Une cible est définie par une arborescence de fichiers, soit embarquée dans le compilateur, soit située dans le système de fichiers. On va voir dans cette section comment est configurée une cible dans le système de fichiers, permettant ainsi de créer ses propres cibles. Pour cela, on commence par extraire de l'exécutable du compilateur les cibles qu'il embarque.

28.1.1 Liste des cibles embarquées

À titre d'information, on peut appeler l'option **-L** pour obtenir la liste des cibles embarquées :

```
plm -L
Embedded targets:
  LPC-2294
  teensy-3-1/privileged
  teensy-3-1/unprivileged
  ...
```

28.1.2 Extraction des cibles embarquées

L'option `-X=cibles` permet d'extraire de l'exécutable du compilateur tous les fichiers de définition des cibles et de les placer dans le répertoire `cibles`[1] :

```
plm -X=cibles
  ...
```

28.1.3 Liste des fichiers d'exemple embarqués dans l'exécutable

L'option `-l` permet d'afficher la liste de tous les fichiers d'exemple embarqués dans l'exécutable du compilateur :

```
plm -l
  LPC-L2294/01-blinkleds.plm
  LPC-L2294/02-control-register-array.plm
  teensy-3-1/00-structure-example.plm
  teensy-3-1/01-blink-led.plm
  ...
```

28.1.4 Extraction d'un fichier d'exemple

On extrait maintenant un fichier d'exemple `teensy-3-1/01-blink-led.plm` :

```
plm -x=teensy-3-1/01-blink-led.plm
```

Le fichier extrait est recopié dans le répertoire courant.

28.1.5 Compilation du fichier exemple

Par défaut, le compilateur utilise les cibles qu'il embarque :

1. Ici, le répertoire destination `cibles` est un répertoire relatif au répertoire courant, mais un répertoire absolu peut aussi être utilisé.

```
plm 01-blink-led.plm
```

Si on veut utiliser les cibles définies dans le répertoire $cibles$, on utilise l'option `-T` :

```
plm -T=cibles 01-blink-led.plm
```

28.1.6 Changement du nom de la cible dans le système de fichiers

Pour se convaincre que la cible nommée dans le fichier est bien celle située dans le répertoire $cibles$, nous allons changer son nom. Pour cela, nous devons modifier :

- la référence à la cible dans le fichier `01-blink-led.plm` ;
- le nom d'un fichier et d'un répertoire dans le répertoire `cibles`.

Commençons par le fichier `01-blink-led.plm`. Sa première ligne est :

```
target "teensy-3-1/unprivileged"
```

Cette ligne signifie que la cible s'appelle `teensy-3-1/unprivileged`. Nous allons la renommer en `teensy-3-1/custom`. La première ligne du fichier `01-blink-led.plm` devient donc :

```
target "teensy-3-1/custom"
```

Maintenant, si on essaie de compiler le fichier `01-blink-led.plm`, une erreur se déclenche : la cible `teensy-3-1/custom` n'est pas définie.

```
plm -T=cibles 01-blink-led.plm
semantic error #1: This target is not defined in 'cibles' directory
target "teensy-3-1/custom"
_______^^^^^^^^^^^^^^^^^^
```

Pour définir la cible, il faut d'abord comprendre comment le répertoire `cibles` est organisé. Affichons la liste de ses sous répertoires :

```
find cibles -type d
cibles
cibles/LPC-L2294
cibles/teensy-3-1
cibles/teensy-3-1/privileged
cibles/teensy-3-1/unprivileged
```

Chaque sous répertoire peut définir une cible[2]. Il suffit donc de dupliquer le répertoire `teensy–3–1/unprivileged` en le renommant `teensy–3–1/custom` :

```
find cibles -type d
cibles
cibles/LPC–L2294
cibles/teensy–3–1
cibles/teensy–3–1/custom
cibles/teensy–3–1/privileged
cibles/teensy–3–1/unprivileged
```

Maintenant, la compilation s'effectue avec succès :

```
plm -T=cibles 01–blink–led.plm
```

Si on essaie de compiler en utilisant les cibles embarquées dans le compilateur :

```
plm 01–blink–led.plm
semantic error #1: This target is not defined in embedded targets
target "teensy–3–1/custom"
_______^^^^^^^^^^^^^^^^^^^
```

Une erreur se déclenche, la cible `teensy–3–1/custom` n'étant pas définie dans l'exécutable du compilateur.

28.2 Définition d'une cible : fichier +config.plm–target

Nous allons décrire comment est définie une cible dans le système de fichiers. Nous prenons comme exemple la cible `teensy–3–1/custom` du répertoire `cibles` (voir la section 28.1 page 142).

Un répertoire définit une cible si il contient un fichier nommé `+config.plm–target`. Le contenu de ce fichier configure la cible, en voici le début :

```
//--- Python tool list
PYTHON_UTILITIES :
   "../../py–toolpath.txt" -> "sources/toolpath.py",
   "../../py–makefile.txt" -> "sources/makefile.py",
   ...
```

Le contenu est une séquence de définitions, chacune d'elles ayant la syntaxe suivante :

2. Pour cela, il doit en fait contenir un fichier `+config.plm–target`.

- son nom (dans l'exemple ci-dessus, `PYTHON_UTILITIES`) ;
- le délimiteur `:` ;
- une liste de paramètres séparés par un virgule, ce qui signifie qu'un nombre quelconque d'items est accepté ;
- **ou** une liste de paramètres séparés par un point-virgule, ce qui signifie qu'un nombre fixe d'items est accepté, et que chacun d'eux a une signification particulière ;
- **ou** un simple paramètre ;
- quand un paramètre désigne un fichier par un chemin relatif, celui-ci est pris à partir du répertoire qui contient le fichier de configuration `+config.plm-target`.

L'ordre des séquences est imposée.

Les sections qui suivent détaillent chaque définition, dans leur ordre d'apparition dans le fichier `+config.plm-tar`

28.2.1 PYTHON_UTILITIES

```
PYTHON_UTILITIES :
   "../../py-toolpath.txt" -> "sources/toolpath.py",
   "../../py-makefile.txt" -> "sources/makefile.py",
   "../../py-check-stacks.txt" -> "sources/check-stacks.py",
   "../../py-plm.txt" -> "sources/plm.py",
   "../../py-build-verbose.txt" -> "build-verbose.py",
   "../../py-clean.txt" -> "clean.py",
   "../../py-objdump.txt" -> "objdump.py",
   "../../py-objsize.txt" -> "objsize.py",
   "../py-run.txt" -> "run.py"
```

Cette déclaration liste une liste de fichiers à recopier dans le répertoire engendré pour chaque projet :

- le fichier source (par exemple `../../py-toolpath.txt`) est désigné par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target` ;
- le fichier destination (par exemple `sources/toolpath.py`) est désigné par un chemin relatif par rapport au répertoire du projet ;
- le fichier destination est rendu exécutable.

28.2.2 PYTHON_BUILD

```
PYTHON_BUILD :
   "../py-build.txt"
```

Le fichier source (ici `../py-build.txt`) est désigné par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source :

- `<<OPT_OPTIMIZATION_OPTION>>` est remplacé par l'option d'optimisation choisie (`O0`, `O1`, `O2`, `O3`, `Os` ou `Oz`) ;
- `<<LLC_OPTIMIZATION_OPTION>>` est remplacé par l'option d'optimisation correspondante pour l'utilitaire `llc` (`O0`, `O1`, `O2` ou `O3`).

La chaîne résultat est copiée dans le fichier `build.py` situé dans le répertoire du projet. Ce fichier est rendu exécutable.

28.2.3 LINKER_SCRIPT

```
LINKER_SCRIPT :
  "../ld-linker.txt"
```

Le fichier source (ici `../ld-linker.txt`) est désigné par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Une substitution est effectuée dans la chaîne obtenue par la lecture du fichier source :

- `!SYSTEMSTACKSIZE !` est remplacé par la taille allouée (en nombre d'octets) à la pile système ; cette valeur est fixée par la définition `SYSTEM_STACK_SIZE`, voir section 28.2.6 page 148.

La chaîne résultat est copiée dans le fichier `sources/linker.ld` situé dans le répertoire du projet.

28.2.4 PANIC

```
PANIC :
  Int32 ; UInt32 ; "../target-panic.ll"
```

Trois arguments (séparés par un point-virgule) sont définis ici :

- le premier (ici `Int32`) est le type PLM utilisé pour le code de panique ;
- le deuxième (ici `UInt32`) est le type PLM utilisé pour coder le numéro de la ligne source du fichier contenant l'instruction qui peut engendrer une panique ;

- le dernier (ici `"../target-panic.ll"`) désigne un fichier source LLVM par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `../target-panic.ll`

- `!PANICCODE!` est remplacé par le type LLVM correspondant type PLM utilisé pour le code de panique (donc, ici `i32` qui est le type LLVM image de `Int32`) ;
- `!PANICLINE!` est remplacé par le type PLM utilisé pour coder le numéro de la ligne source (donc, ici `i32` qui est aussi le type LLVM image de `UInt32`).

La chaîne résultat est ensuite ajoutée au fichier source LLVM `src.ll` du projet.

28.2.5 POINTER_BIT_COUNT

```
POINTER_BIT_COUNT :
  32
```

Le nombre de bits d'un pointeur. Le Cortex-M4 étant une machine 32 bits, sa valeur est 32.

28.2.6 SYSTEM_STACK_SIZE

```
SYSTEM_STACK_SIZE :
  1024
```

La taille en octets allouée à la pile système.

28.2.7 NOP

```
NOP :
  "call void asm sideeffect \"nop\", \"\"() nounwind"
```

Cette entrée définit l'instruction processeur nop, telle qu'elle doit être formulée en LLVM. Pour insérer cette instruction en PLM, utiser le mot réservé **`nop`** (section 21.17 page 111).

28.2.8 SERVICE

```
SERVICE :
  "service-handler.s" ;
  12 ; // as_svc_handler saves 3 registers on system stack
  "service-dispatcher-header.s" ;
```

```
    "service-dispatcher-entry.s" ;
    "service-entry.s"
```

Le premier argument (ici `"service-handler.s"`) est un fichier assembleur désigné par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`. Son contenu est simplement ajouté au fichier source assembleur `src.s` du projet. Ce fichier contient l'implémentation du *svc handler*.

Le deuxième argument (ici `12`) est le nombre d'octets de la pile système que l'exécution du *svc handler* utilise.

Les troisième et quatrième arguments (ici les chaînes `"service-dispatcher-header.s"` et `"service-dispatcher-en` désignent des fichiers assembleur par un chemin relatif au répertoire qui contient le fichier de configuration `+config.plm-target`. Ces deux fichiers permettent de construire la table des appels système. D'abord, le contenu du fichier `"service-dispatcher-header.s"` est simplement ajouté au fichier source assembleur `src.s` du projet. Ensuite, l'opération suivante est effectuée pour chaque service défini dans le projet :

- deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `service-dispatcher-entr`
  - `!ENTRY!` est remplacé par le nom assembleur de la routine réalisant l'implémentation du service ;
  - `!IDX!` est remplacé par l'indice du service (PLM numérote les services, à partir de 0) ;
- la chaîne résultat est simplement ajoutée au fichier source assembleur `src.s` du projet.

Le dernier argument (ici `"service-entry.s"`) permet de construire la routine appelée par les tâches. Il désigne un fichier assembleur par un chemin relatif au répertoire qui contient le fichier de configuration `+config.plm-target`. Ensuite, l'opération suivante est effectuée pour chaque service défini dans le projet :

- deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `service-entry.s` :
  - `!ENTRY!` est remplacé par le nom assembleur de la routine appelée par les tâches ;
  - `!IDX!` est remplacé par l'indice du service ;
- la chaîne résultat est simplement ajoutée au fichier source assembleur `src.s` du projet.

28.2.9 SECTION

Il y a deux possibilités – selon les cibles – pour paramétrer l'implémentation des sections :

- par un appel système (section 28.2.9.1) ;
- par un masquage temporaire des interruptions (section 28.2.9.2 page 151).

28.2.9.1 Implémentation via un appel système

```
SECTION :
  "udfcoded-section-handler.s" ;
  8 ; // saves 2 registers on system stack
  "udfcoded-section-dispatcher-header.s" ;
  "udfcoded-section-dispatcher-entry.s" ;
  "udfcoded-section-invocation.s"
```

L'implémentation des sections via des appels système est détaillée à la section 28.3 page 154.

Le premier argument (ici `"udfcoded-section-handler.s"`) est un fichier assembleur désigné par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`. Son contenu est simplement ajouté au fichier source assembleur `src.s` du projet. Ce fichier contient l'implémentation du *handler* associé.

Le deuxième argument (ici `8`) est le nombre d'octets de la pile système que l'exécution du *handler* utilise.

Les troisième et quatrième arguments (ici les chaînes `"udfcoded-dispatcher-header.s"` et `"service-di` désignent des fichiers assembleur par un chemin relatif au répertoire qui contient le fichier de configuration `+config.plm-target`. Ces deux fichiers permettent de construire la table des appels système. D'abord, le contenu du fichier `"udfcoded-dispatcher-header.s"` est simplement ajouté au fichier source assembleur `src.s` du projet. Ensuite, l'opération suivante est effectuée pour chaque section définie dans le projet :

- deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `udfcoded-dispa`
  - `!ENTRY!` est remplacé par le nom assembleur de la routine réalisant l'implémentation de la section ;
  - `!IDX!` est remplacé par l'indice de la section (PLM numérote les sections, à partir de 0) ;
- la chaîne résultat est simplement ajoutée au fichier source assembleur `src.s` du projet.

Le dernier argument (ici `"udfcoded-entry.s"`) permet de construire la routine appelée par les tâches. Il désigne un fichier assembleur par un chemin relatif au répertoire qui contient le fichier de configuration `+config.plm-target`. Ensuite, l'opération suivante est effectuée pour chaque section définie dans le projet :

- deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `udfcoded-entry`
  - `!ENTRY!` est remplacé par le nom assembleur de la routine appelée par les tâches ;
  - `!IDX!` est remplacé par l'indice de la section ;
- la chaîne résultat est simplement ajoutée au fichier source assembleur `src.s` du projet.

28.2.9.2 Implémentation via un masquage temporaire des interruptions

```
SECTION :
"call void asm sideeffect \"cpsid i\", \"\"() nounwind";// Disable interrupt
"call void asm sideeffect \"cpsie i\", \"\"() nounwind" // Enable interrupt
```

Cette implémentation exige que les tâches puissent masquer les interruptions, ce qui impose qu'elles s'exécutent en mode privilégié. C'est par exemple le cas de la cible `teensy-3-1/privileged`.

Dans ce cas, l'implémentation est complètement réalisée en LLVM, sans avoir besoin de fichier assembleur. Il faut simplement fournir les instructions LLVM de *masquage* et *démasquage* des interruptions. C'est le rôle des deux arguments. Ils expriment les instructions LLVM qui embarquent les instructions assembleur correspondantes.

28.2.10 C_FILES

```
C_FILES :
  "../c-cortex-m4-context.c",
  "../../c-real-time-kernel.c",
  "../c-countTrainingZeros.c"
```

Cette entrée liste tous les fichiers C qui composent le fichier `src.c` d'un projet. Il sont tous désignés par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Les opérations effectuées sont les suivantes :

- les fichiers sont lus et concaténés dans leur ordre d'apparition dans l'entrée `C_FILES` :
- deux substitutions sont effectuées dans la chaîne obtenue par la lecture du fichier source `udfcoded-entry.s` :
  - `!TASKCOUNT !` est remplacé par le nombre de tâches du projet ;
  - `!GUARDCOUNT !` est remplacé par le nombre maximum de commandes gardées des tâches du projet ;
- la chaîne résultat constitue le fichier source C `src.c` du projet.

28.2.11 S_FILES

```
S_FILES :
  "../s-cortex-m4-header.s",
  "s-interrupt-vectors.s",
  "s-reset-handler.s"
```

Cette entrée liste tous les fichiers assembleur qui composent le début du fichier `src.s` d'un projet. Il sont tous désignés par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Les opérations effectuées sont les suivantes :

- les fichiers sont lus et concaténés dans leur ordre d'apparition dans l'entrée `S_FILES` :
- la chaîne résultat constitue le début du fichier source assembleur `src.s` du projet ; PLM y ajoute ensuite le texte assembleur provenant de l'implémentation des services, des sections et des interruptions.

28.2.12 LL_FILES

```
LL_FILES :
  "../ll-cortex-m4.ll",
  "../../ll-clear-bss.ll",
  "../../ll-copy-data-section.ll",
  "../../ll-configuration-on-boot.ll",
  "../../ll-create-task.ll"
```

Cette entrée liste tous les fichiers LLVM qui composent le début du fichier `src.ll` d'un projet. Il sont tous désignés par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

Les opérations effectuées sont les suivantes :

- les fichiers sont lus et concaténés dans leur ordre d'apparition dans l'entrée `LL_FILES` :
- la chaîne résultat constitue le début du fichier source assembleur `src.ll` du projet ; PLM y ajoute ensuite le texte LLVM provenant de la compilation des sources PLM.

28.2.13 PLM_FILES

```
PLM_FILES :
  "../plm-registers-mk20dx256.plm",
  "../plm-teensy-3-1-boot.plm",
  "../plm-teensy-3-1-xtr.plm",
  "../plm-teensy-3-1-digital-io.plm",
  "../plm-teensy-3-1-lcd.plm",
  "../plm-teensy-3-1-panic.plm",
  "../../plm-semaphore.plm"
```

Cette entrée liste tous les fichiers PLM qui sont automatiquement inclus lors de la compilation d'un projet. Il sont tous désignés par un chemin relatif par rapport au répertoire qui contient le fichier de configuration `+config.plm-target`.

28.2.14 INTERRUPT_HANDLER

```
INTERRUPT_HANDLER :
  "../xtr-interrupt-handler.s" ;
  32 ; // Cortex M4 saves 8 regs in user stack on interrupt
  "../undefined-interrupt.s"
```

Le premier argument définit le handler d'interruption, qui est engendré pour chaque interruption qui s'exécute en mode `service` [3] (c'est-à-dire déclarée avec le qualificatif `service`, section 2.5.1 page 27). Les opérations effectuées sont les suivantes :

- pour chaque interruption qui s'exécute en mode `service`, le fichier est lu :
- deux substitutions sont effectuées dans la chaîne obtenue :
  - `!ISR !` est remplacé par le nom assembleur du handler assembleur ;
  - `!HANDLER !` est remplacé par le nom assembleur de la routine issue de la compilation PLM de la routine d'interruption ;
- la chaîne résultat est ajoutée au fichier source assembleur `src.s` du projet.

Le deuxième argument (ici `32`) est le nombre d'octets empilés dans le pile d'une tâche lorsqu'une interruption survient.

Le dernier paramètre est utilisé uniquement quand le projet est compilé sans engendrer le code de panique. Il définit le handler pour toute interruption non définie en PLM[4]. Les opérations effectuées sont les suivantes :

- pour chaque interruption indéfinie, le fichier `../undefined-interrupt.s` est lu :
- une substitution est effectuée dans la chaîne obtenue :
  - `!ISR !` est remplacé par le nom assembleur du handler assembleur ;
- la chaîne résultat est ajoutée au fichier source assembleur `src.s` du projet.

3. Le compilateur PLM prend complètement en charge la compilation d'une routine d'interruption déclarée avec le qualificatif `section` ou `safe`.
4. Quand un projet est compilé avec le code de panique, le compilateur PLM prend en charge les interruptions non définies.

28.2.15 INTERRUPTS

```
INTERRUPTS :
  NMI -> 2,
  MemManage -> 4,
  BusFault -> 5,
  UsageFault -> 6,
  ...
  pinDetectPortD -> 106,
  pinDetectPortE -> 107,
  softwareInterrupt -> 110
```

La liste de toutes les interruptions, chacune d'elle étant accompagnée de son numéro. Celui-ci est utilisé lors de la panique, si une interruption se déclenche alors qu'aucune routine de réponse à cette interruption n'est installée.

28.3 Schéma d'appel des sections

Est présenté ici comment sont implémentées les sections, dont l'appel à partir des tâches s'effectue via un *system call*[5]. c'est le cas de la cible `teensy-3-1/unprivileged`, que nous prenons comme exemple tout au long des explications.

L'implémentation de cette opération est paramètrable grâce à quatre fichiers assembleurs, comme expliqué à la section « *Implémentation via un appel système* » page 150 :

- `udfcoded-section-dispatcher-code.s` : implémentation du code exécuté par l'appel système invoqué par l'appel à une section ;
- `udfcoded-section-dispatcher-entry.s` : entrée du tableau définissant le nom de la fonction implémentant la section ;
- `udfcoded-section-dispatcher-header.s` : en-tête du tableau contenant les noms de la fonction implémentant la section ;
- `udfcoded-section-invocation.s` : implémentation de l'appel à une section.

Il suffit donc de changer ces fichiers assembleur pour modifier le schéma d'appel. En fait, trois schémas sont possibles pour la cible `teensy-3-1/unprivileged` : `udfcoded` (celui utilisé), `r12idx` et `r12direct`, et sont décrits dans les sections suivantes. Le tableau 28.1 page 155 résume les caractéristiques de chacun. Les durées sont estimées grâce au projet `04-section-service-duration.plm`.

5. Quand une section est appelée à partir d'un mode privilégié, l'appel est un simple appel de routine.

Nom du schéma	**Par entrée** (octets)	**`as_section_handler`** (octets)	**Durée estimée** (en cycles d'horloge)
`udfcoded`	4 + 4	40	64
`r12idx`	8 + 4	36	65
`r12direct`	12 + 0	24	61

Tableau 28.1 – *Caractéristiques de différents schémas d'appel de section*

28.3.1 Schéma `udfcoded` d'appel des sections

Dans ce schéma, l'instruction indéfinie `UDF` est utilisée. Le code de cette instruction est `0xDE`xx, où xx est une valeur que le Cortex ignore. On utilise ce champ pour coder l'indice du service appelé.

Pour coder l'appel relatif à une section, PLM va utiliser le fichier `udfcoded-section-invocation.s`. Le contenu de fichier est :

```
  .section ".text.!USER_ROUTINE!","ax",%progbits
  .global !USER_ROUTINE!
  .type !USER_ROUTINE!,%function
  .align 1
  .code 16
  .thumb_func

!USER_ROUTINE!:
  .fnstart
  udf !IDX!
  bx lr

.Lfunc_end_!USER_ROUTINE!:
  .size !USER_ROUTINE!, .Lfunc_end_!USER_ROUTINE! - !USER_ROUTINE!
  .cantunwind
  .fnend
```

Trois séquences sont remplacées par le compilateur PLM :

- `!USER_ROUTINE!`, par le nom assembleur de la fonction qui implémente l'appel de la section en mode utilisateur ;
- `!IMPLEMENTATION_ROUTINE!`, par le nom assembleur de la fonction qui implémente la section (inutilisée dans ce shéma) ;
- `!IDX!`, par l'indice de la section[6].

6. Lorsqu'il analyse un projet, PLM numérote les sections qu'il rencontre (à partir de 0). C'est ce numéro qui remplace la séquence `!IDX!`.

PLM effectue donc cette opération pour chaque section, par ordre croissant de l'indice, et accumule le texte produit dans le fichier assembleur `src.s`.

Donc, quand une instruction `UDF` est rencontrée, le Cortex exécute l'exception n°3, qui renvoie vers `as_section_handler`[7], défini par le fichier `udfcoded-section-dispatcher-code.s` :

```
  .section ".text.as_section_handler","ax",%progbits
  .global as_section_handler
  .type as_section_handler, %function

as_section_handler:
@---------------------- Save preserved registers
  push r5, lr
@---------------------- R5 <- thread SP
  mrs r5, psp @ r5 <- thread SP
@---------------------- LR <- Address of UDF instruction
  ldr lr, [r5, #24] @ 24 : 6 stacked registers before saved PC
@---------------------- Set return address to instruction following UDF
  adds lr, #2
  str lr, [r5, #24]
@---------------------- R12 <- address of dispatcher
  ldr r12, =__udf_dispatcher_table
@---------------------- LR <- bits 0-7 of UDF instruction
  ldrb lr, [lr, #-2] @ LR is service call index
@---------------------- r12 <- address of routine to call
  ldr r12, [r12, lr, lsl #2] @ Address R12 + LR << 2
@---------------------- Call service routine
  blx r12 @ R5: thread PSP
@---------------------- Set return code (from R0 to R3) in stacked registers
  stmia r5!, r0, r1, r2, r3 @ R5 is thread SP
@---------------------- Restore preserved registers, return from interrupt
  pop r5, pc
```

Ce code va rechercher dans l'instruction `UDF` la valeur xx de l'octet de poids faible, et l'utilise pour construire l'adresse d'une entrée du tableau `__udf_dispatcher_table`. Cette entrée contient l'adresse de la routine à exécuter.

Remarquer ce code empile deux registres (soit 8 octets) dans la pile système ; il faut s'assurer que cette valeur correspond à la valeur du paramètre *Section handler system stack byte count footprint* déclaré dans la `configuration`.

Remarquer aussi que lorsque l'instruction `UDF` est exécutée, l'adresse de retour empilée est l'adresse de l'instruction `UDF`, et non pas l'adresse de l'instruction suivante. Il faut donc manuellement ajouter 2

7. Ce qui est défini dans le fichier `s-interrupt-vectors.s` de la cible qui décrit la table des interruptions (voir section « *S_FILES* » page 151).

à l'adresse de retour.

Il faut donc construire ce tableau. Le fichier `udfcoded-section-dispatcher-header.s` définit son en-tête :

```
__udf_dispatcher_table:
```

Ce texte est ajouté à la fin du fichier assembleur `src.s`.

Ensuite, PLM va utiliser le fichier `udfcoded-section-dispatcher-entry.s` pour définir chaque entrée du tableau :

```
   .word !IMPLEMENTATION_ROUTINE! @ !IDX!, user routine !USER_ROUTINE!
```

PLM effectue pour chaque section les substitutions de `!IMPLEMENTATION_ROUTINE!`, de `!USER_ROUTINE!` et de `!IDX!`, par ordre croissant de l'indice, en ajoutant le texte produit à la fin du fichier assembleur `src.s`.

28.3.2 Schéma `r12idx` d'appel des sections

Dans ce schéma, l'instruction indéfinie `UDF` est utilisée. Le code de cette instruction est `0xDE`xx, où xx est une valeur que le Cortex ignore. Contrairement au shéma précédent, le champ xx est toujours à 0, l'indice de la section est entré dans le registre `r12`[8].

Pour coder l'appel relatif à une section, PLM va utiliser le fichier `r12idx-section-invocation.s`. Le contenu de fichier est :

```
   .section ".text.!USER_ROUTINE!","ax",%progbits
   .global !USER_ROUTINE!
   .type !USER_ROUTINE!,%function
   .align 1
   .code 16
   .thumb_func

 !USER_ROUTINE!:
   .fnstart
   mov r12, !IDX!
   udf 0
   bx lr

 .Lfunc_end_!USER_ROUTINE!:
   .size !USER_ROUTINE!, .Lfunc_end_!USER_ROUTINE! - !USER_ROUTINE!
   .cantunwind
```

8. L'ABI considère le registre `r12` comme l'*Intra-Procedure-call scratch register* : *register r12 (IP) may be used by a linker as a scratch register between a routine and any subroutine it calls.*

```
    .fnend
```

Trois séquences sont remplacées par le compilateur PLM :

- `!USER_ROUTINE!`, par le nom assembleur de la fonction qui implémente l'appel de la section en mode utilisateur ;
- `!IMPLEMENTATION_ROUTINE!`, par le nom assembleur de la fonction qui implémente la section (inutilisée dans ce shéma) ;
- `!IDX!`, par l'indice de la section[9].

PLM effectue donc cette opération pour chaque section, par ordre croissant de l'indice, et accumule le texte produit dans le fichier assembleur `src.s`.

Donc, quand une instruction `UDF` est rencontrée, le Cortex exécute l'exception n°3, qui renvoie vers `as_section_handler`, défini par le fichier `r12idx-section-dispatcher-code.s` :

```
  .section ".text.as_section_handler","ax",%progbits
  .global as_section_handler
  .type as_section_handler, %function

as_section_handler:
@---------------------- Save preserved registers
  push r5, lr
@---------------------- R5 <- thread SP
  mrs r5, psp @ r5 <- thread SP
@---------------------- LR <- Address of UDF instruction
  ldr lr, [r5, #24] @ 24 : 6 stacked registers before saved PC
@---------------------- Set return address to instruction following UDF
  adds lr, #2
  str lr, [r5, #24]
@---------------------- LR <- address of dispatcher table
  ldr lr, =__udf_dispatcher_table
@---------------------- r12 <- address of routine to call
  ldr r12, [lr, r12, lsl #2] @ Address : LR + R12 << 2
@---------------------- Call service routine
  blx r12 @ R5: thread PSP
@---------------------- Set return code (from R0 to R3) in stacked registers
  stmia r5!, r0, r1, r2, r3 @ R5 is thread SP
@---------------------- Restore preserved registers, return from interrupt
  pop r5, pc
```

9. Lorsqu'il analyse un projet, PLM numérote les sections qu'il rencontre (à partir de 0). C'est ce numéro qui remplace la séquence `!IDX!`.

Ce code va utiliser la valeur r12 pour construire l'adresse d'une entrée du tableau __udf_dispatcher_table. Cette entrée contient l'adresse de la routine à exécuter.

Remarquer ce code empile deux registres (soit 8 octets) dans la pile système ; il faut s'assurer que cette valeur correspond à la valeur du paramètre *Section handler system stack byte count footprint* déclaré dans la `configuration`.

Il faut donc construire ce tableau[10]. Le fichier `r12idx-section-dispatcher-header.s` définit son en-tête :

```
__udf_dispatcher_table:
```

Ce texte est ajouté à la fin du fichier assembleur `src.s`.

Ensuite, PLM va utiliser le fichier `r12idx-section-dispatcher-entry.s` pour définir chaque entrée du tableau :

```
   .word !IMPLEMENTATION_ROUTINE! @ !IDX!, user routine !USER_ROUTINE!
```

PLM effectue pour chaque section les substitutions de `!IMPLEMENTATION_ROUTINE!`, de `!USER_ROUTINE!` et de `!IDX!`, par ordre croissant de l'indice, en ajoutant le texte produit à la fin du fichier assembleur `src.s`.

28.3.3 Schéma `r12direct` d'appel des sections

Dans ce schéma, l'instruction indéfinie `UDF` est utilisée. L'adresse de la routine implémentant la section est entrée dans le registre r12.

Pour coder l'appel relatif à une section, PLM va utiliser le fichier `r12direct-section-invocation.s`. Le contenu de fichier est :

```
   .section ".text.!USER_ROUTINE!","ax",%progbits
   .global !USER_ROUTINE!
   .type !USER_ROUTINE!,%function
   .align 1
   .code 16
   .thumb_func

 !USER_ROUTINE!:
   .fnstart
   movw r12, :lower16:!IMPLEMENTATION_ROUTINE!
   movt r12, :upper16:!IMPLEMENTATION_ROUTINE!
   udf 0
```

10. Les fichiers `r12idx-section-dispatcher-header.s` et `r12idx-section-dispatcher-entry.s` sont identiques à ceux du schéma précédent `"udfcoded"`.

```
  bx lr

.Lfunc_end_!USER_ROUTINE!:
  .size !USER_ROUTINE!, .Lfunc_end_!USER_ROUTINE! - !USER_ROUTINE!
  .cantunwind
  .fnend
```

Trois séquences sont remplacées par le compilateur PLM :

- `!USER_ROUTINE!`, par le nom assembleur de la fonction qui implémente l'appel de la section en mode utilisateur ;
- `!IMPLEMENTATION_ROUTINE!`, par le nom assembleur de la fonction qui implémente la section ;
- `!IDX!`, par l'indice de la section (inutilisé dans ce shéma).

PLM effectue donc cette opération pour chaque section, par ordre croissant de l'indice, et accumule le texte produit dans le fichier assembleur `src.s`.

Donc, quand une instruction `UDF` est rencontrée, le Cortex exécute l'exception n°3, qui renvoie vers `as_section_handler`, défini par le fichier `r12direct-section-dispatcher-code.s` :

```
  .section ".text.as_section_handler","ax",%progbits
  .global as_section_handler
  .type as_section_handler, %function

as_section_handler:
@---------------------- Save preserved registers
  push r5, lr
@---------------------- R5 <- thread SP
  mrs r5, psp @ r5 <- thread SP
@---------------------- LR <- Address of UDF instruction
  ldr lr, [r5, #24] @ 24 : 6 stacked registers before saved PC
@---------------------- Set return address to instruction following UDF
  adds lr, #2
  str lr, [r5, #24]
@---------------------- Call service routine
  blx r12 @ R5: thread PSP
@---------------------- Set return code (from R0 to R3) in stacked registers
  stmia r5!, r0, r1, r2, r3 @ R5 is thread SP
@---------------------- Restore preserved registers, return from interrupt
  pop r5, pc
```

Ce code utilise directement la valeur `r12`, c'est l'adresse de la routine à exécuter. Remarquer ce code empile deux registres (soit 8 octets) dans la pile système ; il faut s'assurer que cette valeur correspond à la valeur du paramètre *Section handler system stack byte count footprint* déclaré dans la `configuration`.

Les fichiers `r12direct-section-dispatcher-header.s` et `r12direct-section-dispatcher-entry.s` sont inutiles pour ce schéma, et sont donc vides.

Chapitre 29

Grammaires

Le langage PLM définit deux grammaires :

- la grammaire des sources des programmes PLM (section 29.1 page 162) ;
- la grammaire de description d'un cible (section 29.2 page 181).

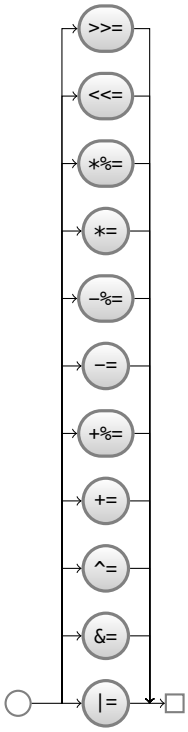
Ce chapitre liste l'ensemble des règles de production de ces deux grammaires.

29.1 Grammaire du langage PLM

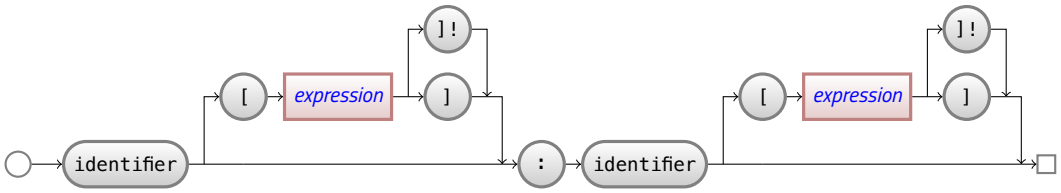
L'axiome de la grammaire est start_symbol.

Voici la liste alphabétique des non terminaux: assignment_combined_with_operator, control_register_lvalue, declaration, effective_parameters, expression, expression_access_list, expression_addition, expression_bitwise_and, expression_bitwise_or, expression_bitwise_xor, expression_comparison, expression_equality, expression_if, expression_logical_and, expression_logical_xor, expression_product, expression_shift, function, function_header, guard, guarded_command, if_instruction, import_file, instruction, instructionList, isr, lvalue, lvalue_operand, mode, primary, private_or_public_struct_property_declara private_struct_property_declaration, procedure_call, procedure_formal_arguments, procedure_input_formal propertyGetterSetter, registerDeclaration, start_symbol, staticArrayProperty, staticArray_exp, struct_property structure_function, system_routine, type_definition, type_definition_enclosed_in_square_breackets.

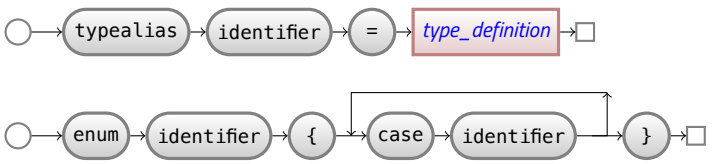
29.1.1 Non terminal *assignment_combined_with_operator*

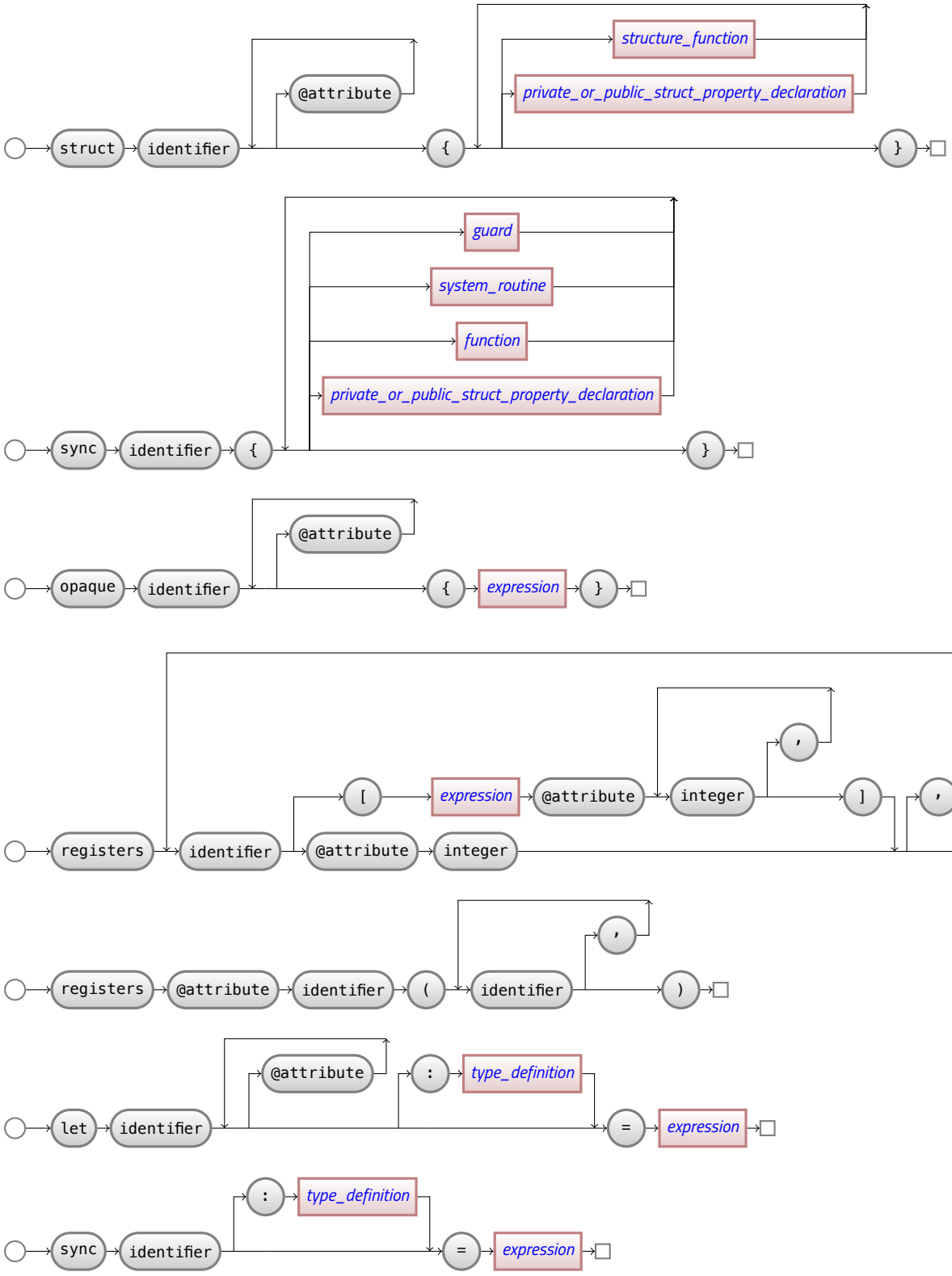


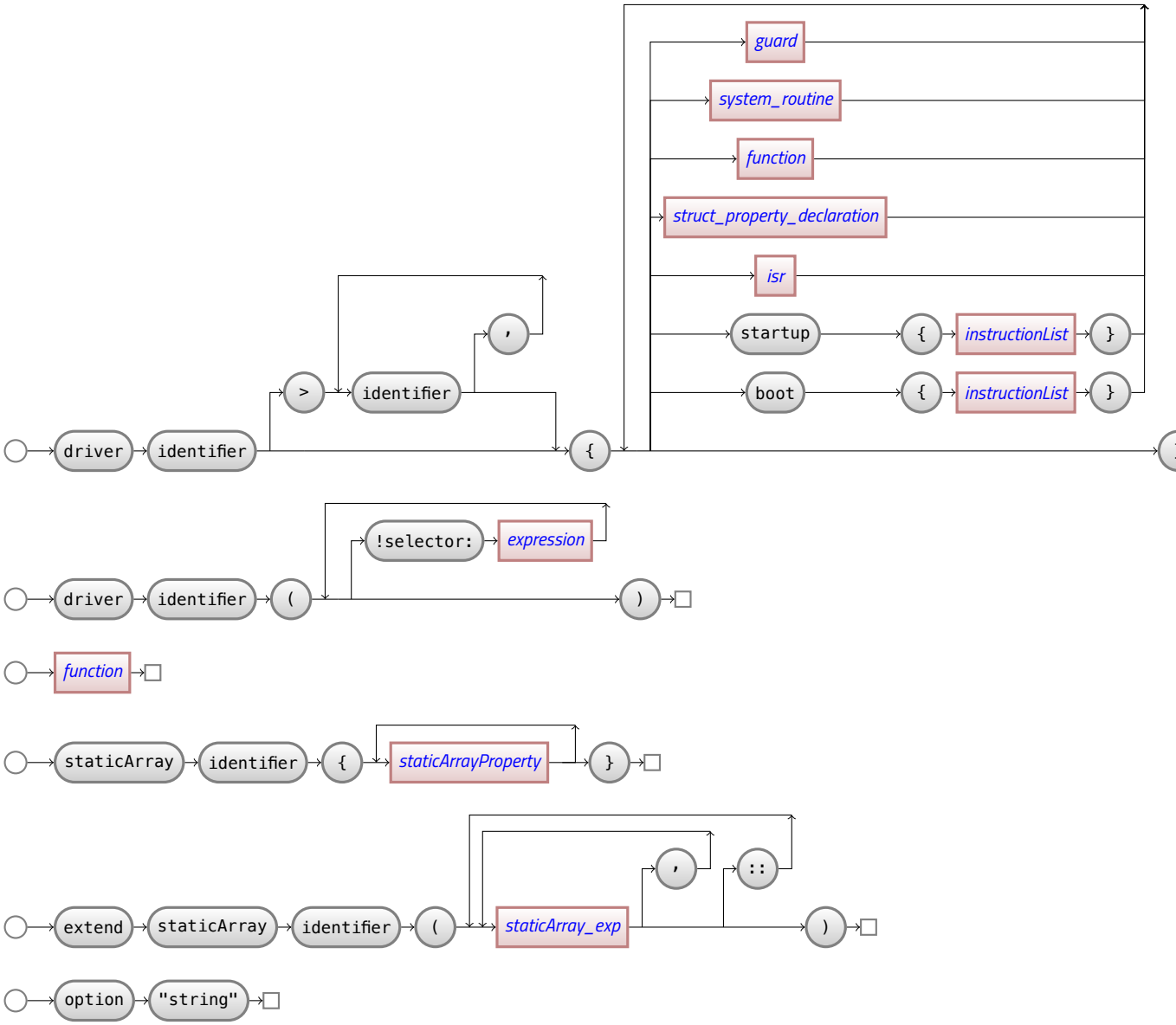
29.1.2 Non terminal *control_register_lvalue*

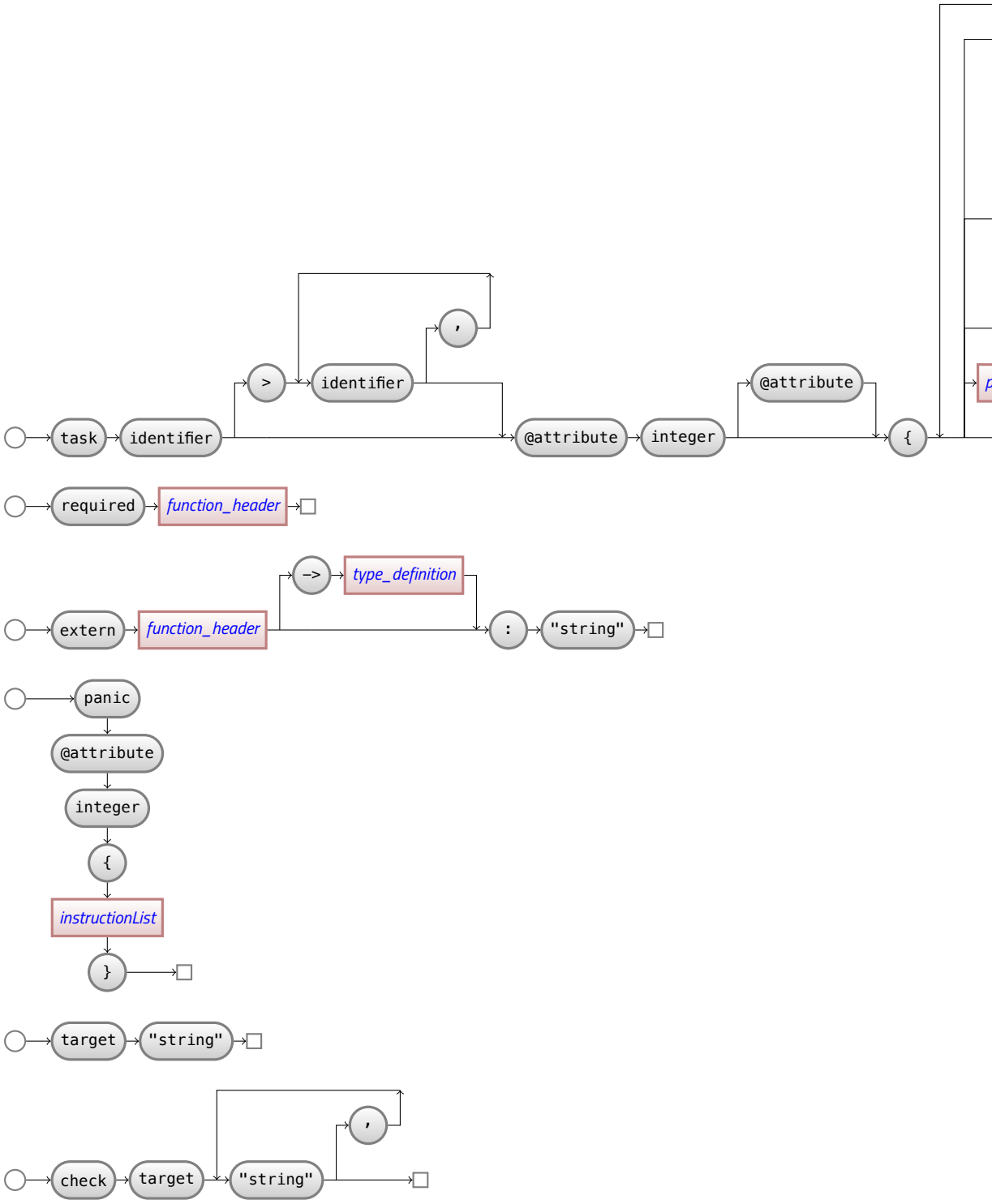


29.1.3 Non terminal *declaration*

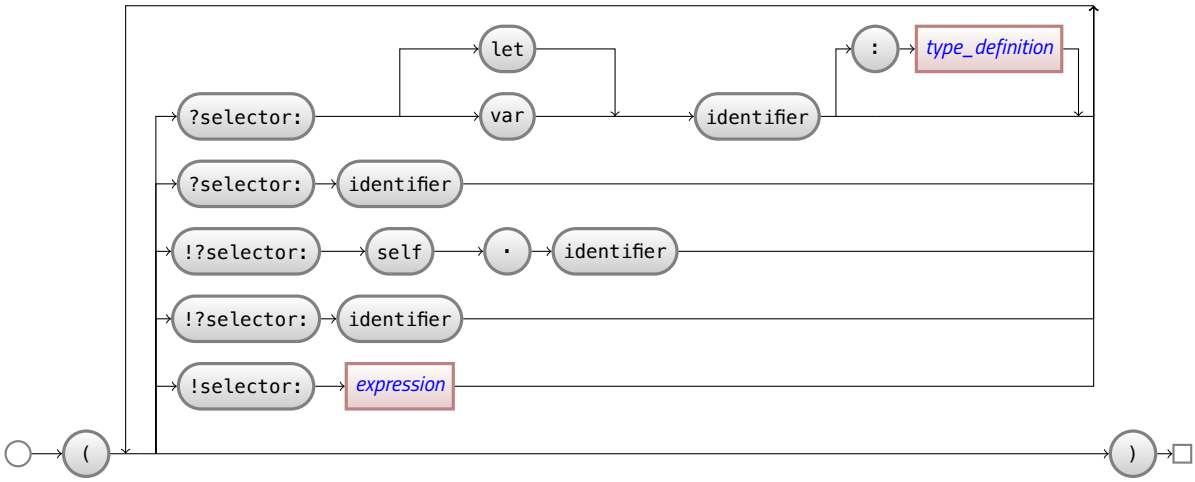



struct
identifier
@attribute
{
structure_function
private_or_public_struct_property_declaration
}
sync
identifier
{
guard
system_routine
function
private_or_public_struct_property_declaration
}
opaque
identifier
@attribute
{
expression
}
registers
identifier
[
expression
@attribute
integer
,
]
,
@attribute
integer
registers
@attribute
identifier
(
identifier
,
)
let
identifier
@attribute
:
type_definition
=
expression
sync
identifier
:
type_definition
=
expression

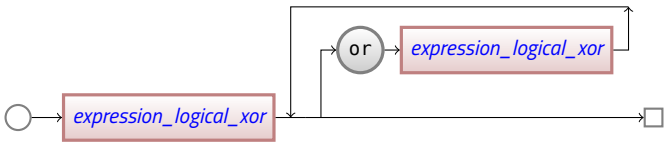

driver
identifier
>
identifier
,
{
guard
system_routine
function
struct_property_declaration
isr
startup
{
instructionList
}
boot
{
instructionList
}
driver
identifier
(
!selector:
expression
)
function
staticArray
identifier
{
staticArrayProperty
}
extend
staticArray
identifier
(
staticArray_exp
,
::
)
option
"string"


task
identifier
>
identifier
,
@attribute
integer
@attribute
{
required
function_header
extern
function_header
->
type_definition
:
"string"
panic
@attribute
integer
{
instructionList
}
target
"string"
check
target
"string"
,

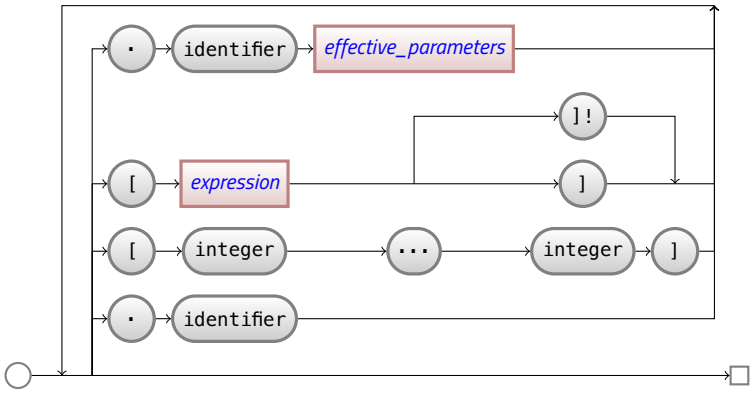
29.1.4 Non terminal *effective_parameters*



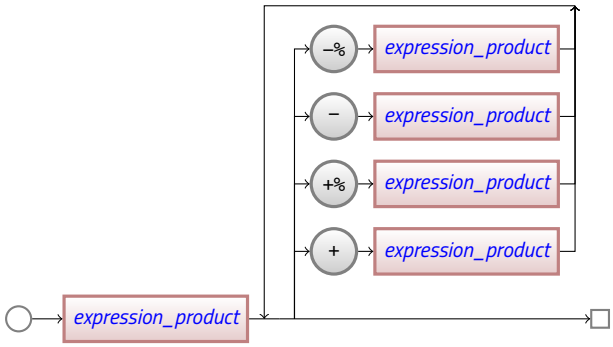
29.1.5 Non terminal *expression*



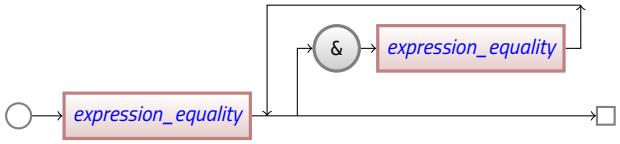
29.1.6 Non terminal *expression_access_list*



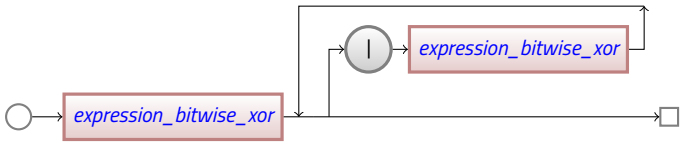
29.1.7 Non terminal *expression_addition*



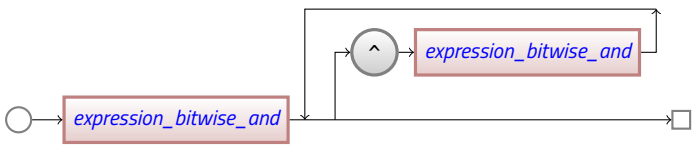
29.1.8 Non terminal *expression_bitwise_and*



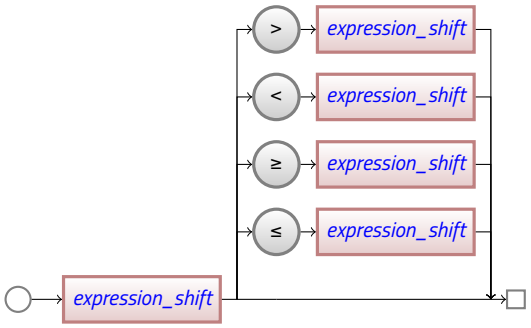
29.1.9 Non terminal *expression_bitwise_or*



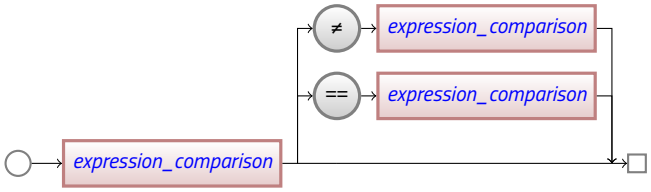
29.1.10 Non terminal *expression_bitwise_xor*



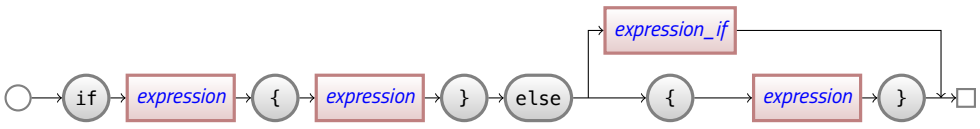
29.1.11 Non terminal *expression_comparison*



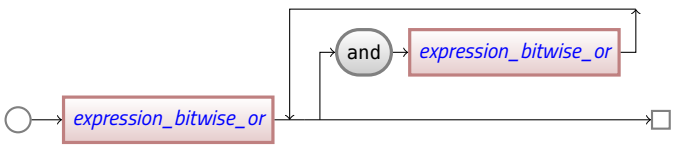
29.1.12 Non terminal *expression_equality*



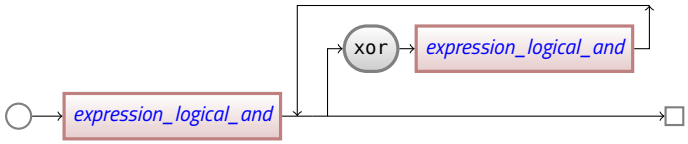
29.1.13 Non terminal *expression_if*



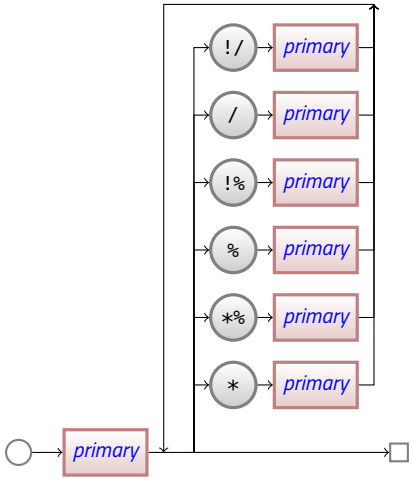
29.1.14 Non terminal *expression_logical_and*



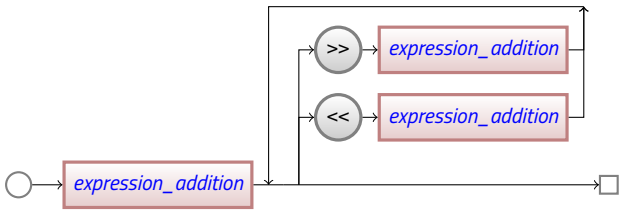
29.1.15 Non terminal *expression_logical_xor*



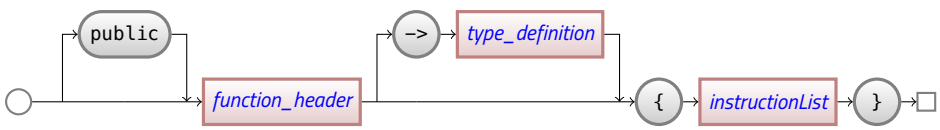
29.1.16 Non terminal *expression_product*



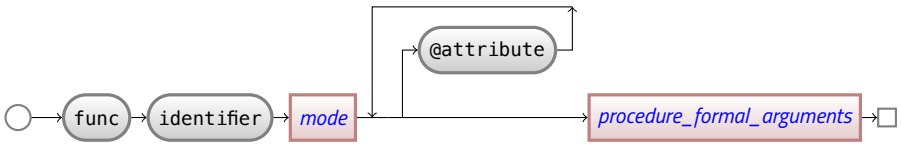
29.1.17 Non terminal *expression_shift*



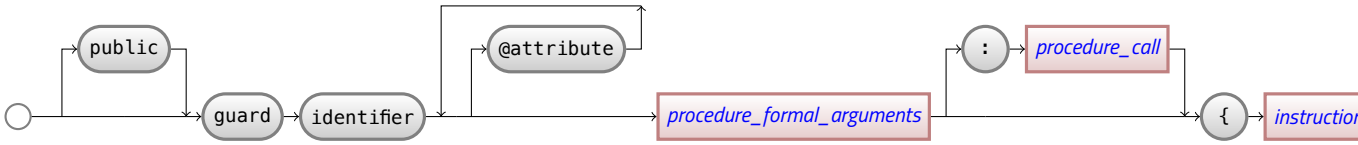
29.1.18 Non terminal *function*



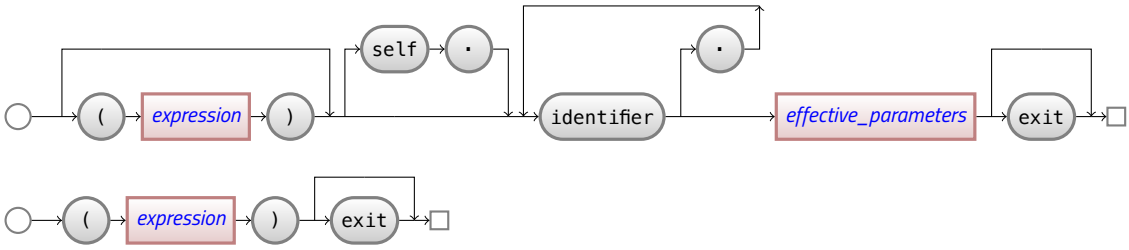
29.1.19 Non terminal *function_header*



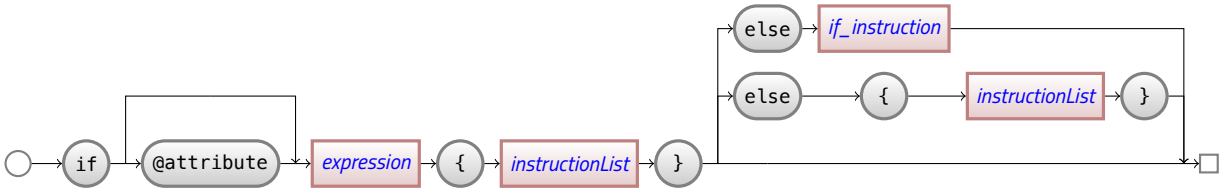
29.1.20 Non terminal *guard*



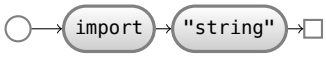
29.1.21 Non terminal *guarded_command*



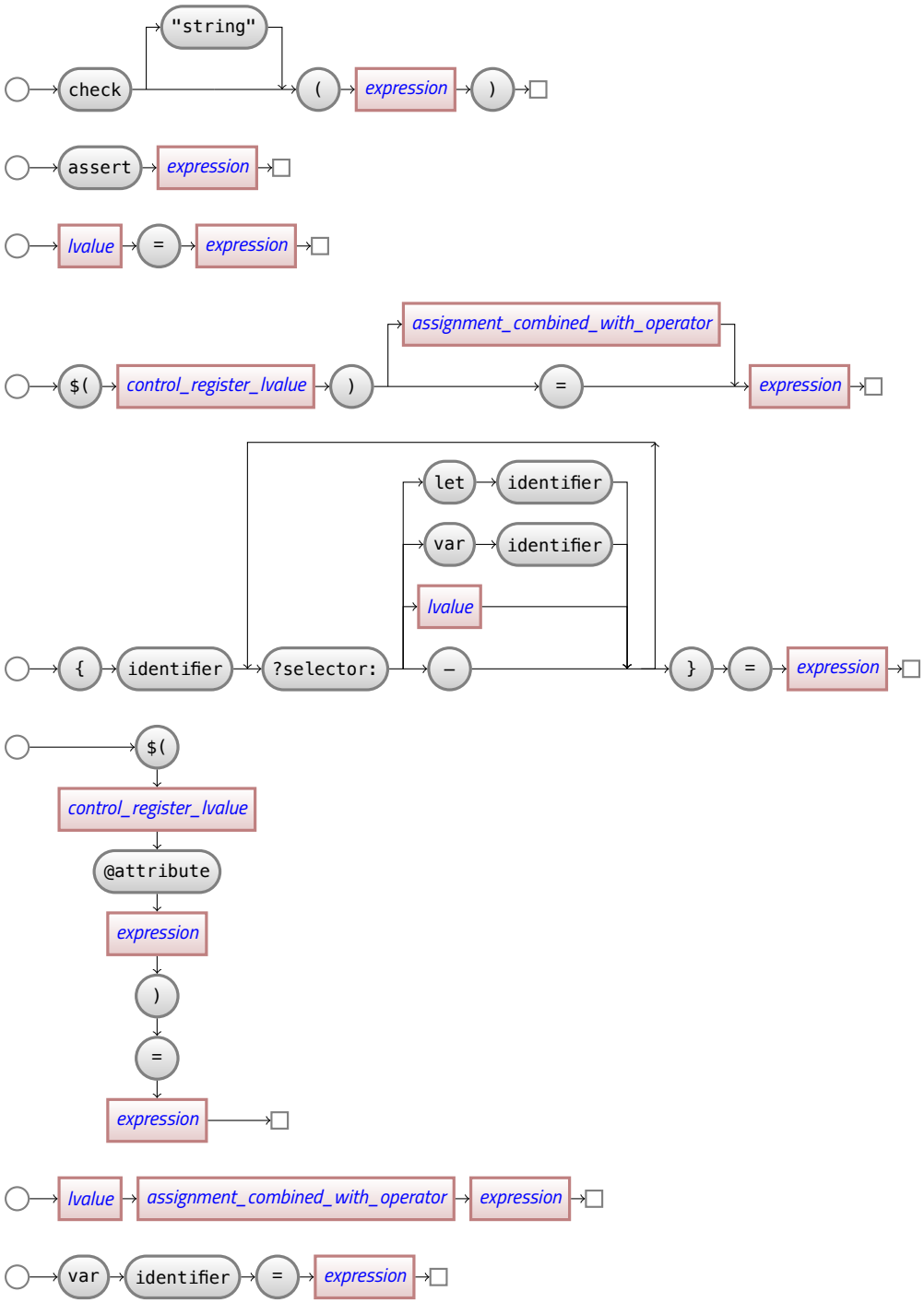
29.1.22 Non terminal *if_instruction*

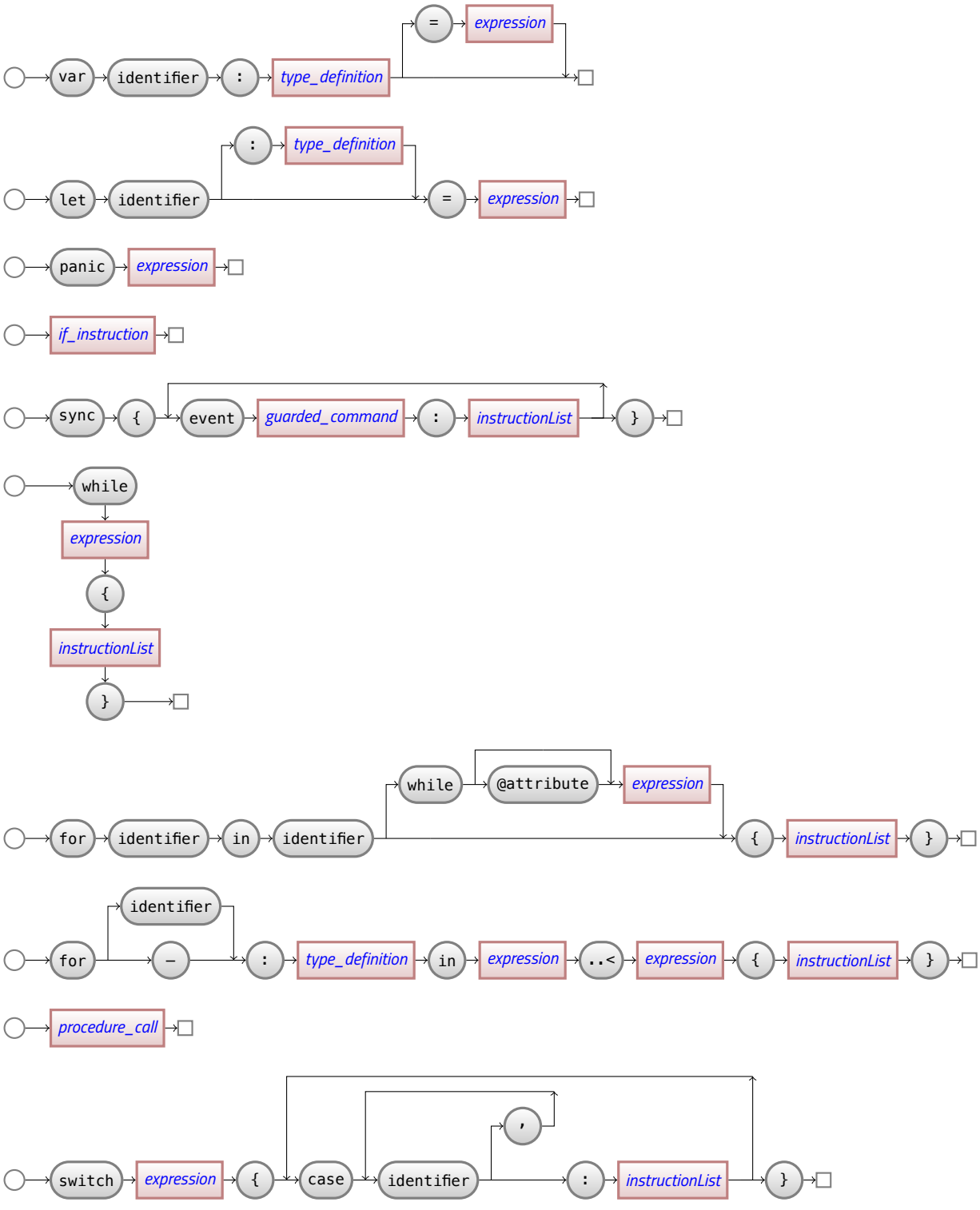


29.1.23 Non terminal *import_file*

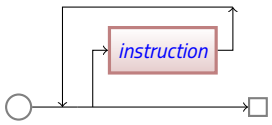


29.1.24 Non terminal *instruction*

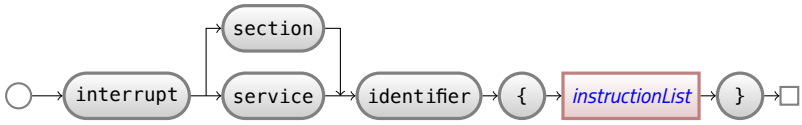



var
identifier
:
type_definition
=
expression
let
identifier
:
type_definition
=
expression
panic
expression
if_instruction
sync
{
event
guarded_command
:
instructionList
}
while
expression
{
instructionList
}
for
identifier
in
identifier
while
@attribute
expression
{
instructionList
}
for
identifier
–
:
type_definition
in
expression
..<
expression
{
instructionList
}
procedure_call
switch
expression
{
case
identifier
,
:
instructionList
}

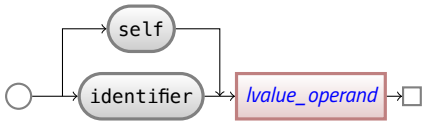
29.1.25 Non terminal *instructionList*



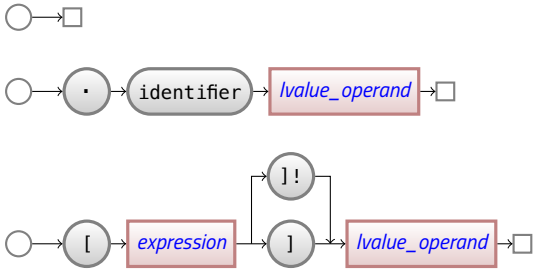
29.1.26 Non terminal *isr*



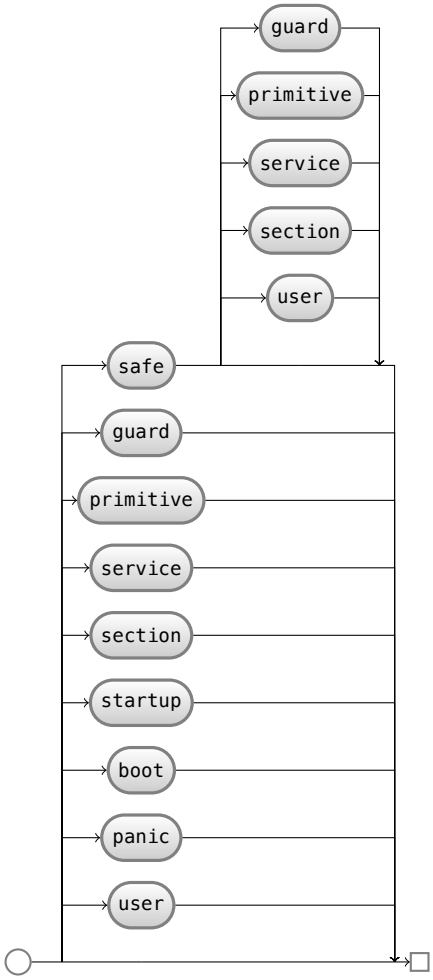
29.1.27 Non terminal *lvalue*



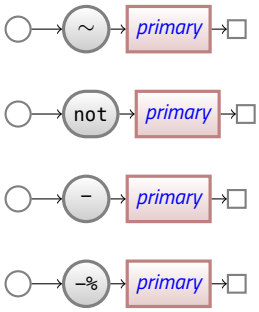
29.1.28 Non terminal *lvalue_operand*

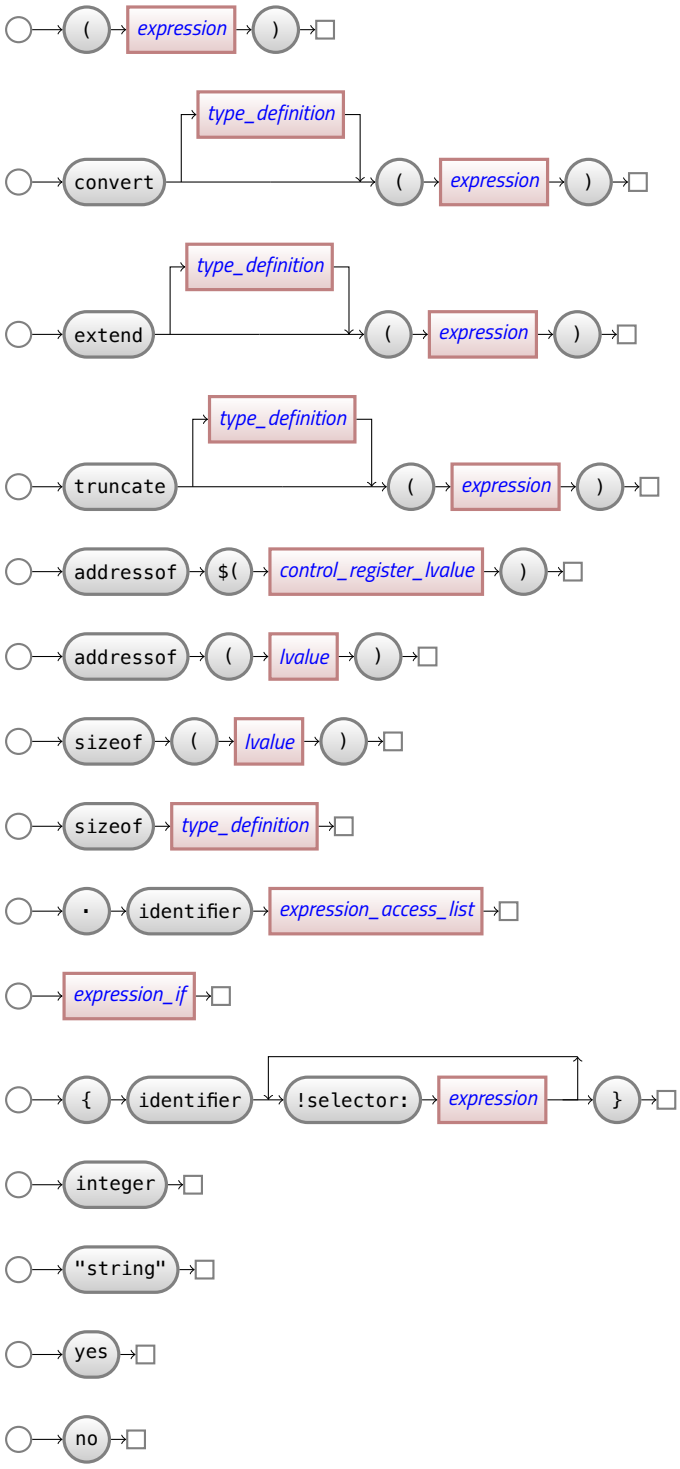


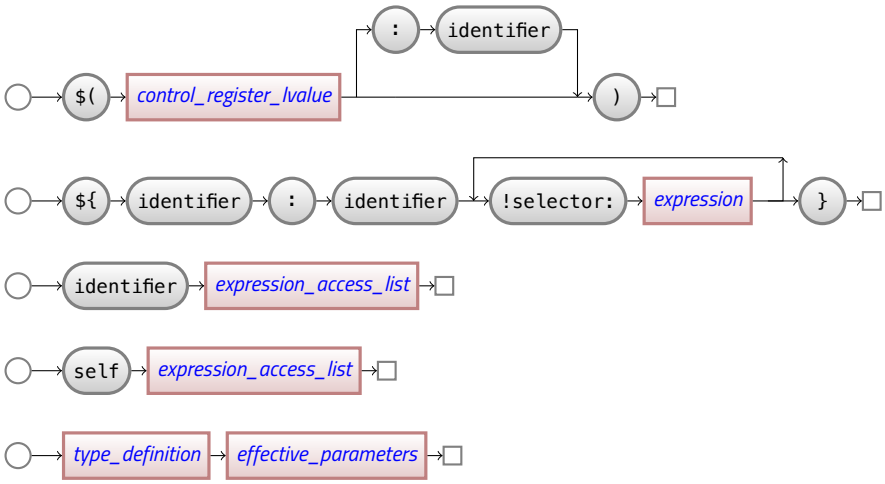
29.1.29 Non terminal *mode*



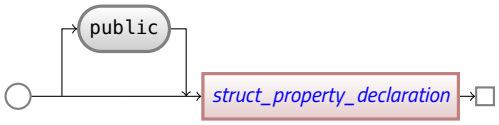
29.1.30 Non terminal *primary*




(
expression
)
type_definition
convert
(
expression
)
type_definition
extend
(
expression
)
type_definition
truncate
(
expression
)
addressof
$(
control_register_lvalue
)
addressof
(
lvalue
)
sizeof
(
lvalue
)
sizeof
type_definition
.
identifier
expression_access_list
expression_if
{
identifier
!selector:
expression
}
integer
"string"
yes
no



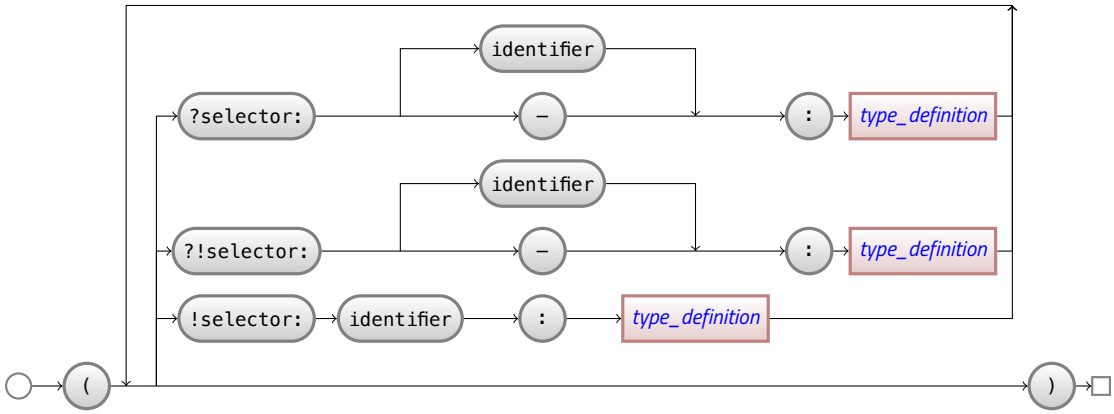
29.1.31 Non terminal *private_or_public_struct_property_declaration*



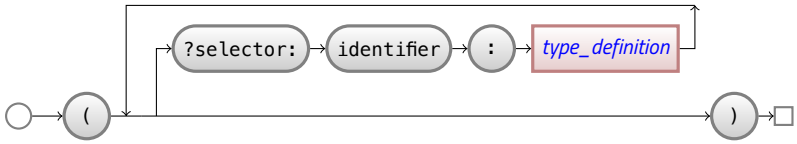
29.1.32 Non terminal *private_struct_property_declaration*

29.1.33 Non terminal *procedure_call*

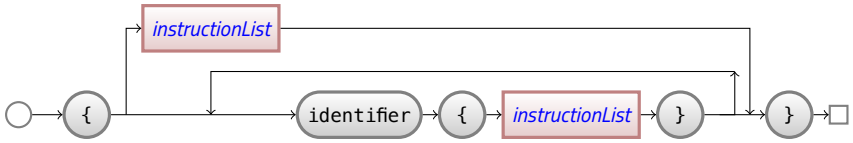
29.1.34 Non terminal *procedure_formal_arguments*



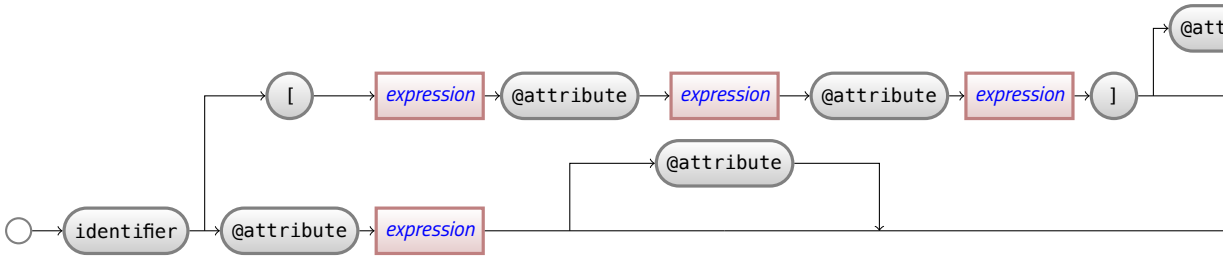
29.1.35 Non terminal *procedure_input_formal_arguments*



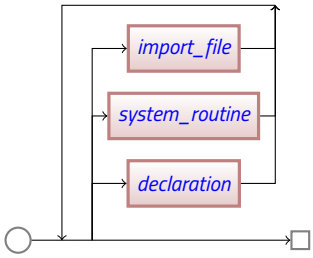
29.1.36 Non terminal *propertyGetterSetter*



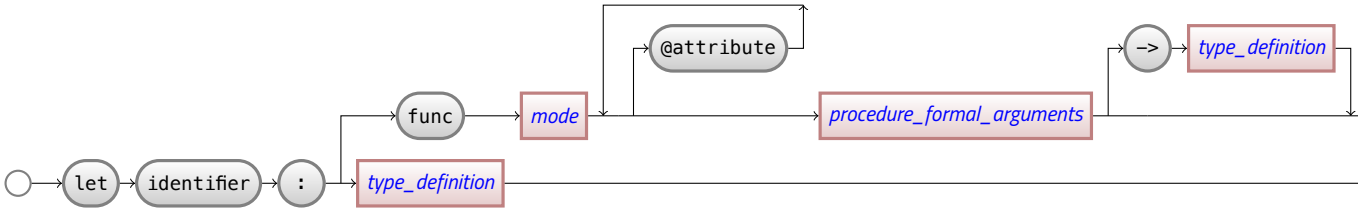
29.1.37 Non terminal *registerDeclaration*



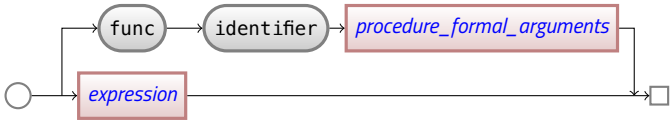
29.1.38 Non terminal *start_symbol*



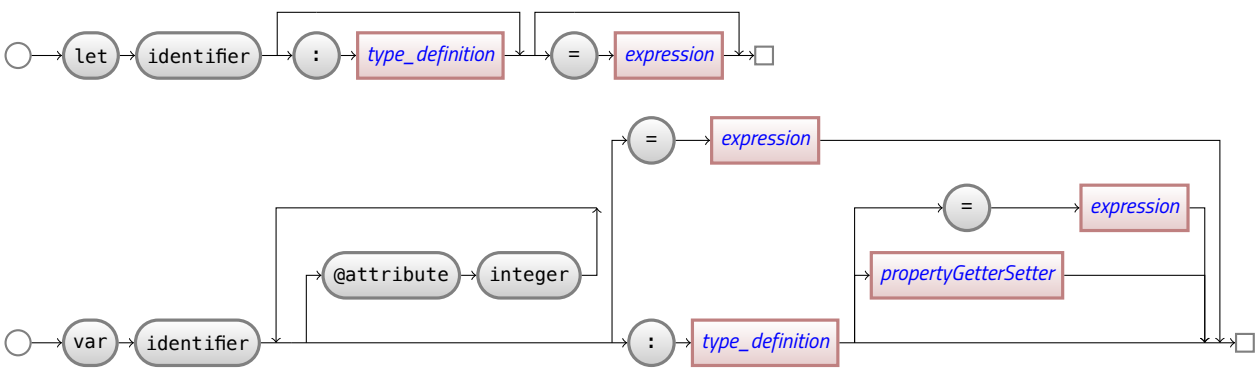
29.1.39 Non terminal *staticArrayProperty*



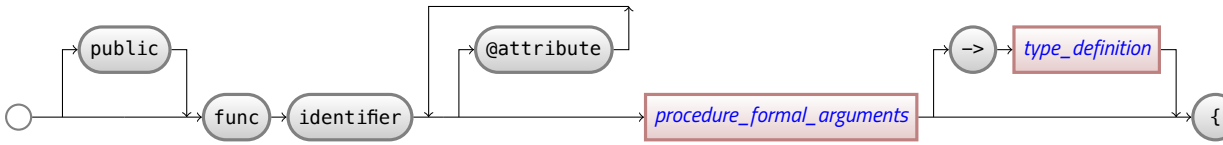
29.1.40 Non terminal *staticArray_exp*



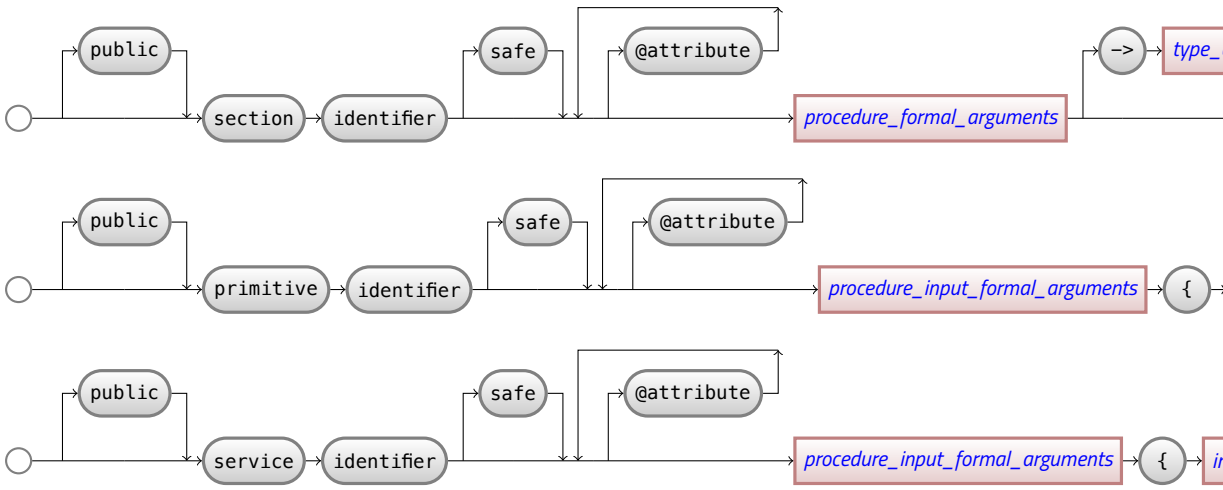
29.1.41 Non terminal *struct_property_declaration*



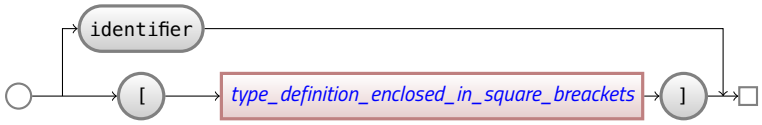
29.1.42 Non terminal *structure_function*



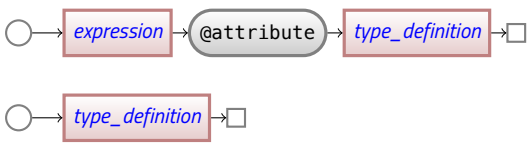
29.1.43 Non terminal *system_routine*



29.1.44 Non terminal *type_definition*



29.1.45 Non terminal *type_definition_enclosed_in_square_breackets*



29.2 Grammaire du langage de description de cible

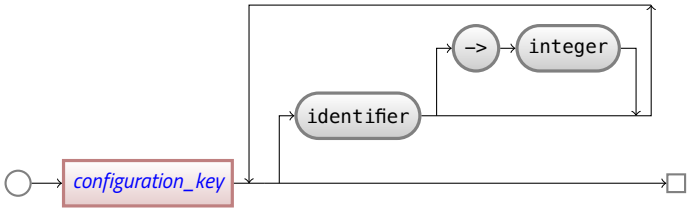
L'axiome de la grammaire est configuration_start_symbol.

Voici la liste alphabétique des non terminaux: configuration_key, configuration_start_symbol, interruptConfigList, python_utility_tool_list.

29.2.1 Non terminal *configuration_key*

29.2.2 Non terminal *configuration_start_symbol*

29.2.3 Non terminal *interruptConfigList*



29.2.4 Non terminal *python_utility_tool_list*