Manuel de référence du langage

Exemples

Les aventures de Docteur R.

www.irai.com

1. Eléments communs *7*

1.1. Les variables **7**

1.1.1. Les variables booléennes 7

1.1.2. Les variables numériques 8

1.1.3. Les temporisations 8

1.2. Les actions **10**

1.2.1. Affectation d'une variable booléenne 10

Affectation complémentée d'une variable booléenne 11

1.2.2. Mise à un d'une variable booléenne 12

1.2.3. Mise à zéro d'une variable booléenne 13

1.2.4. Inversion d'une variable booléenne 13

1.2.5. Mise à zéro d'un compteur, d'un mot ou d'un long 14

1.2.6. Incrémentation d'un compteur, d'un mot ou d'un long 15

1.2.7. Décrémentation d'un compteur, d'un mot ou d'un long 15

1.2.8. Temporisations 16

1.2.9. Interférences entre les actions 17

1.2.10. Actions de la norme CEI 1131-3 17

1.2.11. Actions multiples 18

1.2.12. Code littéral 19

1.3. Les tests **19**

1.3.1. Forme générale 20

1.3.2. Modificateur de test 20

1.3.3. Temporisations 21

1.3.4. Priorité des opérateurs booléens 21

1.3.5. Test toujours vrai 22

1.3.6. Test sur variable numérique 22

1.3.7. Transitions sur plusieurs lignes 23

1.4. Utilisation de symboles **23**

1.4.1. Syntaxe des symboles 23

1.4.2. Symboles automatiques 24

1.4.3. Syntaxe des symboles automatiques 24

1.4.4. Comment le compilateur gère-t-il les symboles automatiques ? 24

1.4.5. Plage d'attribution des variables 25

1.5. A propos des exemples **26**

1.6. Grafcet **28**

1.6.1. Grafcet simple 28

1.6.2. Divergence et convergence en « Et » 31

1.6.3. Divergence et convergence en « Ou » 33

1.6.4. Etapes puits et sources, transitions puits et sources 36

1.6.5. Actions multiples, actions conditionnées....36
1.6.6. Synchronisation....38
1.6.7. Forçages de Grafcet....39
1.6.8. Macro-étapes....48
1.6.9. Compteurs....51

1.7. Gemma....52
1.7.1. Création d'un Gemma....54
1.7.2. Contenu des rectangles du Gemma....54
1.7.3. Obtenir un Grafcet correspondant....54
1.7.4. Annuler les espaces vides dans le Grafcet....55
1.7.5. Imprimer le Gemma....55
1.7.6. Exporter le Gemma....55
1.7.7. Exemple de Gemma....55

1.8. Ladder....58
1.8.1. Exemple de Ladder....59

1.9. Logigramme....60
1.9.1. Dessin des logigrammes....61
1.9.2. Exemple de logigramme....62

1.10. Langages littéraux....65
1.10.1. Comment utiliser le langage littéral ?....65
1.10.2. Définition d'une boîte de code....66
1.10.3. Le langage littéral bas niveau....67
1.10.4. Macro-instruction....120
1.10.5. Librairie....121
1.10.6. Macro-instructions prédéfinies....121
1.10.7. Description des macro-instructions prédéfinies....121
1.10.8. Exemple en langage littéral bas niveau....123

1.11. Langage littéral étendu....126
1.11.1. Ecriture d'équations booléennes....127
1.11.2. Ecriture d'équations numériques....128
1.11.3. Structure de type IF ... THEN ... ELSE 130
1.11.4. Structure de type WHILE ... ENDWHILE....130
1.11.5. Exemple de programme en langage littéral étendu....131

1.12. Langage littéral ST....132
1.12.1. Généralités....132
1.12.2. Equations booléennes....133
1.12.3. Equations numériques....134
1.12.4. Structures de programmation....135
1.12.5. Exemple de programme en langage littéral étendu....137

1.13. Organigramme 137
1.13.1. Dessin d'un organigramme 138
1.13.2. Contenu des rectangles 139

1.14. Illustration 139

1.15. Blocs fonctionnels 142
1.15.1. Création d'un bloc fonctionnel 142
1.15.2. Dessin du bloc et création du fichier « .ZON » 143
1.15.3. Création du fichier « .LIB » 145
1.15.4. Exemple simple de bloc fonctionnel 145
1.15.5. Illustration 146
1.15.6. Complément de syntaxe 149

1.16. Blocs fonctionnels évolués 150
1.16.1. Syntaxe 150
1.16.2. Différencier anciens et nouveaux blocs fonctionnels 150
1.16.3. Exemple 151

1.17. Blocs fonctionnels prédéfinis 152
1.17.1. Blocs de conversion 152
1.17.2. Blocs de temporisation 153
1.17.3. Blocs de manipulations de chaîne de caractères 153
1.17.4. Blocs de manipulation de table de mots 153

1.18. Techniques avancées 153
1.18.1. Code généré par le compilateur 153
1.18.2. Optimisation du code généré 154

2. Exemples 157

2.1. A propos des exemples 157
2.1.1. Grafcet simple 157
2.1.2. Grafcet avec divergence en OU 158
2.1.3. Grafcet avec divergence en ET 159
2.1.4. Grafcet et synchronisation 160
2.1.5. Forçage d'étapes 161
2.1.6. Etapes puits et sources 162
2.1.7. Etapes puits et sources 163
2.1.8. Forçage de Grafcets 164
2.1.9. Mémorisation de Grafcets 164
2.1.10. Grafcet et macro-étapes 166
2.1.11. Folios chaînés 167
2.1.12. Logigramme 169
2.1.13. Grafcet et Logigramme 170
2.1.14. Boîte de langage littéral 171

2.1.15. Organigramme 172
2.1.16. Organigramme 173
2.1.17. Bloc fonctionnel 174
2.1.18. Bloc fonctionnel 175
2.1.19. Ladder 176
2.1.20. Exemple développé sur une maquette de train 177

Manuel pédagogique à l'usage des utilisateurs d'AUTOMGEN 183

1. Eléments communs

Ce chapitre détaille les éléments communs à tous les langages utilisables dans AUTOMGEN.

Le logo  repère les nouveautés utilisables dans la version 7 d'AUTOMGEN.

1.1. Les variables

Les types de variables suivants existent :

⇨ le type booléen : la variable peut prendre la valeur vrai (1) ou faux (0).

⇨ le type numérique : la variable peut prendre une valeur numérique, différents sous types existent : variables 16 bits, 32 bits et virgule flottante.

⇨ le type temporisation : type structuré, il est une combinaison du type booléen et numérique.

A partir de la version 6 la syntaxe des noms de variables peut être celle propre à AUTOMGEN ou la syntaxe de la norme CEI 1131-3.

1.1.1. Les variables booléennes

Le tableau suivant donne la liste exhaustive des variables booléennes utilisables.

Type	Syntaxe AUTOMGEN	Syntaxe CEI 1131-3	Commentaire
Entrées	I0 à I9999	%I0 à %I9999	Peut correspondre ou non à des entrées physiques (dépend de la configuration des E/S de la cible).
Sorties	O0 à O9999	%Q0 à %Q9999	Peut correspondre ou non à des sorties physiques (dépend de la configuration des E/S de la cible).
Bits Système	U0 à U99	%M0 à %M99	Voir le manuel consacré à l'environnement pour le détail des bits Système.
Bits Utilisateur	U100 à U9999	%M100 à %M9999	Bits internes à usage général.

Etapes Grafcet	X0 à X9999	%X0 à %X9999	Bits d'étapes Grafcet.
Bits de mots	M0#0 à M9999#15	%MW0 :X0 à %MW9999 :X15	Bits de mots : le numéro du bit est précisé en décimal et est compris entre 0 (bit de poids faible) et 15 (bit de poids fort).

1.1.2. Les variables numériques

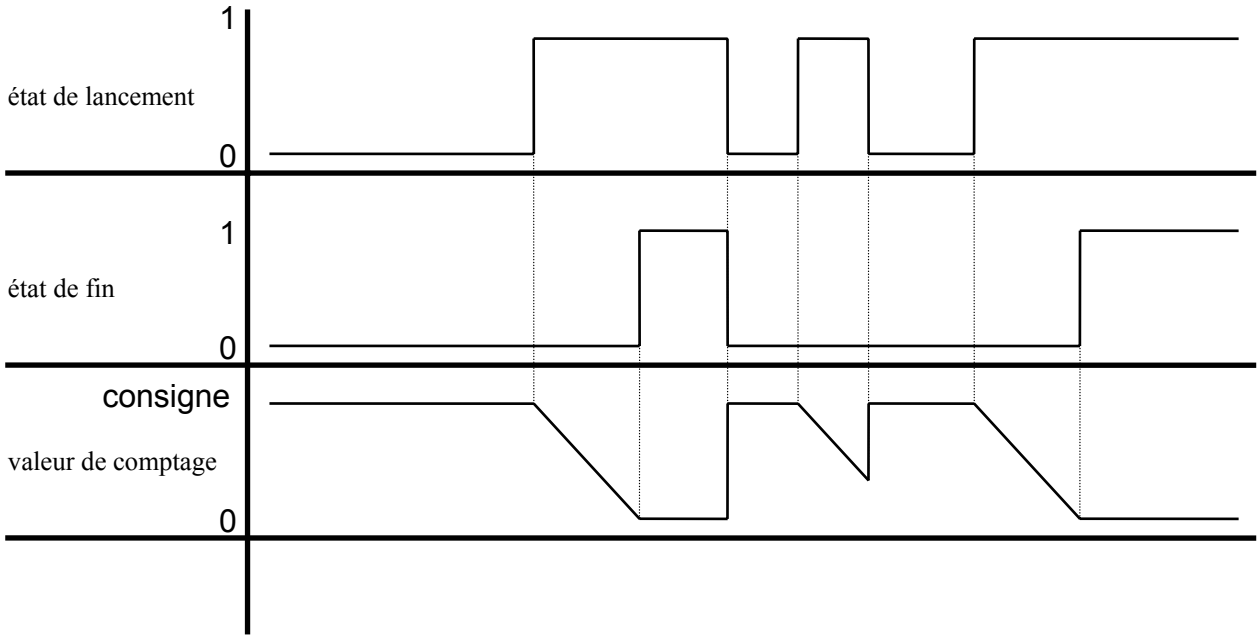
Le tableau suivant donne la liste exhaustive des variables numériques.

Type	Syntaxe AUTOMGEN	*Syntaxe CEI 1131-3*	Commentaire
Compteurs	C0 à C9999	%C0 à %C9999	Compteur de 16 bits, peut être initialisé, incrémenté, décrémenté et testé avec les langages booléens sans utiliser le langage littéral.
Mots Système	M0 à M199	%MW0 à %MW199	Voir le manuel consacré à l'environnement pour le détail des mots Système.
Mots Utilisateur	M200 à M9999	%MW200 à %MW9999	Mot de 16 bits à usage général.
Longs	L100 à L4998	%MD100 à %MD4998	Valeur entière sur 32 bits.
Flottants	F100 à F4998	%MF100 à %MF4998	Valeur réelle sur 32 bits (format IEEE).

1.1.3. Les temporisations

La temporisation est un type composé qui regroupe deux variables booléennes (état de lancement, état de fin) et deux variables numériques sur 16 bits (la consigne et le compteur).

Le schéma suivant donne le chronogramme de fonctionnement d'une temporisation :



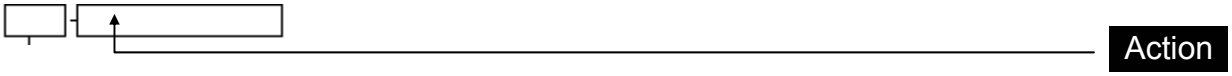
La valeur de consigne d'une temporisation est comprise entre 0 ms et 4294967295 ms (soit un peu plus de 49 jours)

La consigne de la temporisation peut être modifiée par programme, voir chapitre 1.10.3. Le langage littéral bas niveau (instruction STA).
Le compteur de la temporisation peut être lu par programme, voir chapitre 1.10.3. Le langage littéral bas niveau (instruction LDA).

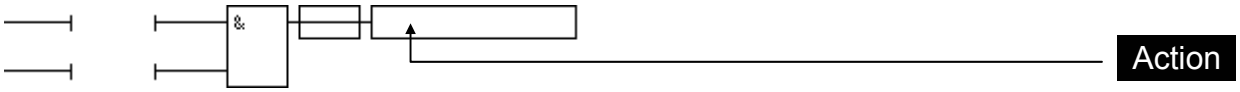
1.2. Les actions

Les actions sont utilisées dans :

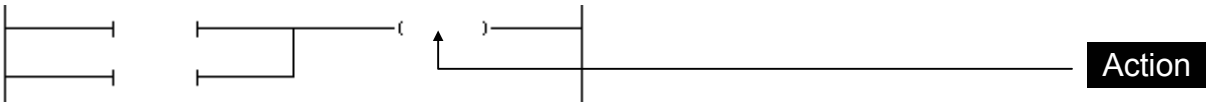
⇨ les rectangles d'action du langage Grafcet,



⇨ les rectangles d'action du langage logigramme,



⇨ les bobines du langage ladder.



1.2.1. Affectation d'une variable booléenne

La syntaxe de l'action « Affectation » est :

«variable booléenne»

Fonctionnement :

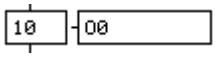
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors la variable est mise à 1 (état vrai),

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux alors la variable est mise à 0 (état faux).

Table de vérité :

Commande	Etat de la variable (résultat)
0	0
1	1

Exemple :



Si l'étape 10 est active alors O0 prend la valeur 1, sinon O0 prend la valeur 0.

Plusieurs actions « Affectation » peuvent être utilisées pour une même variable au sein d'un programme. Dans ce cas, les différentes commandes sont combinées en « Ou » logique.

Exemple :

Etat de X10	Etat de X50	Etat de O5
0	0	0
1	0	1
0	1	1
1	1	1

Affectation complémentée d'une variable booléenne

La syntaxe de l'action « Affectation complémentée » est :

«N variable booléenne»

Fonctionnement :

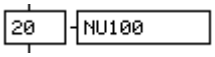
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors la variable est mise à 0 (état faux),

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux alors la variable est mise à 1 (état vrai).

Table de vérité :

Commande	Etat de la variable (résultat)
0	1
1	0

Exemple :



Si l'étape 20 est active, alors U100 prend la valeur 0, sinon U100 prend la valeur 1.

Plusieurs actions « Affectation complémentée» peuvent être utilisées pour une même variable au sein d'un programme. Dans ce cas, les différentes commandes sont combinées en « Ou » logique.

Exemple :

Etat de X100	Etat de X110	Etat de O20
0	0	1
1	0	0
0	1	0
1	1	0

1.2.2. Mise à un d'une variable booléenne

La syntaxe de l'action « Mise à un » est :

«S variable booléenne»

Fonctionnement :

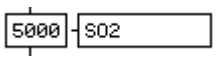
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors la variable est mise à 1 (état vrai),

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux alors l'état de la variable n'est pas modifié.

Table de vérité :

Commande	Etat de la variable (résultat)
0	inchangé
1	1

Exemple :



Si l'étape 5000 est active alors O2 prend la valeur 1, sinon O2 garde son état.

1.2.3. Mise à zéro d'une variable booléenne

La syntaxe de l'action « Mise à zéro » est :

«R variable booléenne»

Fonctionnement :

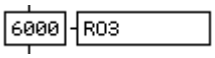
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors la variable est mise à 0 (état faux),

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux alors l'état de la variable n'est pas modifié.

Table de vérité :

Commande	Etat de la variable (résultat)
0	inchangé
1	0

Exemple :



Si l'étape 6000 est active alors O3 prend la valeur 0, sinon O3 garde son état.

1.2.4. Inversion d'une variable booléenne

La syntaxe de l'action « Inversion » est :

«I variable booléenne»

Fonctionnement :

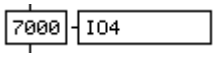
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors l'état de la variable est inversé à chaque cycle d'exécution,

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux alors l'état de la variable n'est pas modifié.

Table de vérité :

Commande	Etat de la variable (résultat)
0	inchangé
1	inversé

Exemple :



Si l'étape 7000 est active alors l'état de O4 est inversé, sinon O4 garde son état.

1.2.5. Mise à zéro d'un compteur, d'un mot ou d'un long

La syntaxe de l'action « Mise à zéro d'un compteur, d'un mot ou d'un long» est :
«R compteur ou mot»
Fonctionnement :

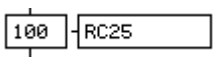
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors le compteur, le mot ou le long est remis à zéro,

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux la valeur du compteur, du mot, ou du long n'est pas modifiée.

Table de vérité :

Commande	Valeur du compteur du mot ou du long (résultat)
0	Inchangé
1	0

Exemple :



Si l'étape 100 est active alors le compteur 25 est remis à zéro, sinon C25 garde sa valeur.

1.2.6. Incrémentation d'un compteur, d'un mot ou d'un long

La syntaxe de l'action « Incrémentation d'un compteur » est :

«+ compteur, mot ou long»

Fonctionnement :

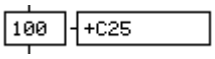
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors le compteur, le mot ou le long est incrémenté à chaque cycle d'exécution,

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux la valeur du compteur n'est pas modifiée.

Table de vérité :

Commande	Valeur du compteur, du mot ou du long (résultat)
0	Inchangé
1	valeur actuelle +1

Exemple :



Si l'étape 100 est active alors le compteur 25 est incrémenté, sinon C25 garde sa valeur.

1.2.7. Décrémentation d'un compteur, d'un mot ou d'un long

La syntaxe de l'action « Décrémentation d'un compteur » est :

«- compteur, mot ou long»

Fonctionnement :

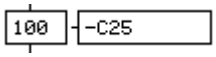
⇨ si la commande du rectangle d'action ou de la bobine est à l'état vrai alors le compteur, le mot ou le long est décrémenté à chaque cycle d'exécution,

⇨ si la commande du rectangle d'action ou de la bobine est à l'état faux la valeur du compteur n'est pas modifiée.

Table de vérité :

Commande	Valeur du compteur, du mot ou du long (résultat)
0	inchangé
1	valeur actuelle -1

Exemple :



Si l'étape 100 est active alors le compteur 25 est decrémenté, sinon C25 garde sa valeur.

1.2.8. Temporisations

Les temporisations sont considérées comme des variables booléennes et sont utilisables avec les actions « Affectation », « Affectation complémentée », « Mise à un », « Mise à zéro », et « Inversion ». La consigne de la temporisation peut être écrite à la suite de l'action. La syntaxe est :
« temporisation(durée) »
La durée est par défaut exprimée en dixièmes de seconde. Le caractère « S » placé à la fin de la durée indique qu'elle est exprimée en secondes.

Exemples :

L'étape 10 lance une temporisation de 2 secondes qui restera active tant que l'étape le sera. L'étape 20 arme une temporisation de 6 secondes qui restera active même si l'étape 20 est désactivée.
Une même temporisation peut être utilisée à plusieurs endroits avec une même consigne et à des instants différents. La consigne de la temporisation doit dans ce cas être indiquée une seule fois.

Remarque : d'autres syntaxes existent pour les temporisations. Voir chapitre 1.3.3. Temporisations

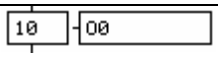
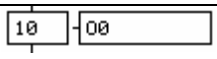
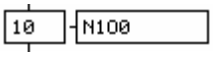
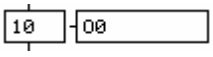
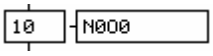
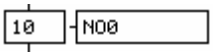
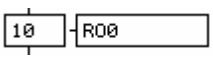
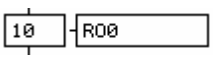
1.2.9. Interférences entre les actions

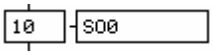
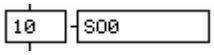
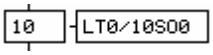
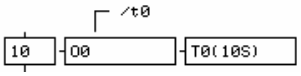
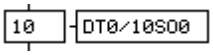
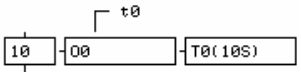
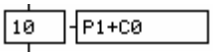
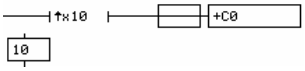
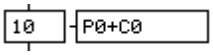
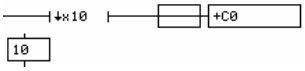
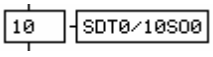
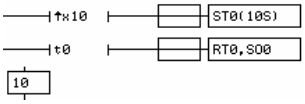
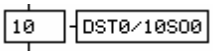
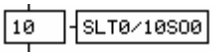
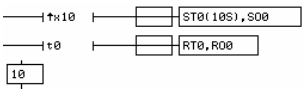
Certains types d'action ne peuvent être utilisés simultanément sur une variable. Le tableau ci-dessous résume les combinaisons interdites.

	Affectation	**Affectation complémentée**	**Mise à un**	**Mise à zéro**	**Inversion**
Affectation	OUI	NON	NON	NON	NON
Affectation complémentée	NON	OUI	NON	NON	NON
Mise à un	NON	NON	OUI	OUI	OUI
Mise à zéro	NON	NON	OUI	OUI	OUI
Inversion	NON	NON	OUI	OUI	OUI

1.2.10. Actions de la norme CEI 1131-3

Le tableau ci-dessous donne la liste des actions de la norme CEI 1131-3 utilisables dans AUTOMGEN V>=6 par rapport à la syntaxe standard d'AUTOMGEN. V5.

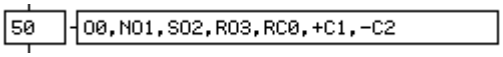
Nom	*Syntaxe AUTOMGEN V>=6*	*Syntaxe AUTOMGEN V5*	*Exemple AUTOMGEN V>=6*	*Exemple d'équivalent AUTOMGEN V5*
Nom mémorisé	Néant	Néant		
Nom mémorisé	N1	Néant		
Nom mémorisé complémenté	N0	N		
Mise à zéro	R	R		

Mise à 1	S	S		
Limité dans le temps	LTn/durée	Inexistant		
Temporisé	DTn/durée	Inexistant		
Impulsion sur front montant	P1	Inexistant		
Impulsion sur front descendant	P0	Inexistant		
Mémorisé et temporisé	SDTn/durée	Inexistant		
Temporisé et mémorisé	DSTn/durée	Inexistant		
Mémorisé et limité dans le temps	SLTn/durée	Inexistant		

1.2.11. Actions multiples

Au sein d'un même rectangle d'action ou d'une bobine, plusieurs actions peuvent être écrites en les séparant par le caractère « , » (virgule).

Exemple :



Plusieurs rectangles d'action (Grafcet et logigramme) ou bobines (ladder) peuvent être juxtaposés. Reportez-vous aux chapitres correspondant à ces langages pour plus de détails.

1.2.12. Code littéral

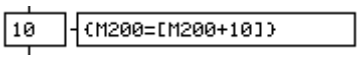
Du code littéral peut être inséré dans un rectangle d'action ou une bobine.

La syntaxe est :

« { code littéral } »

Plusieurs lignes de langage littéral peuvent être écrites entre les accolades. Le séparateur est ici aussi le caractère « , » (virgule).

Exemple :

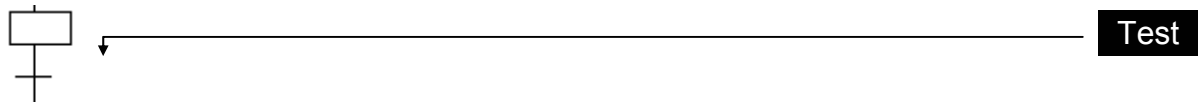


Pour plus de détails, consultez les chapitres « Langage littéral bas niveau », « Langage littéral étendu » et « Langage littéral ST ».

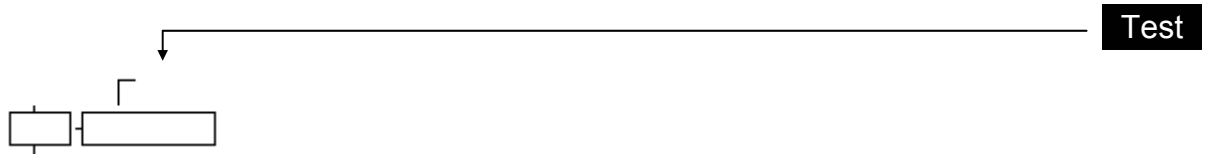
1.3. Les tests

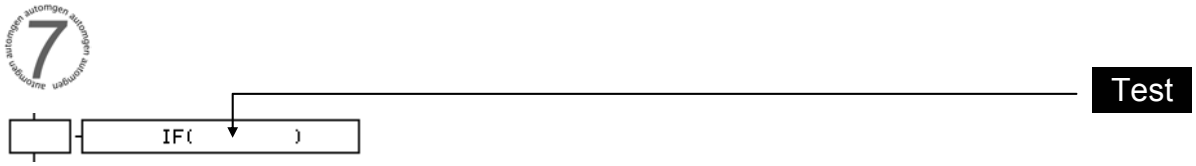
Les tests sont utilisés dans :

⇨ les transitions du langage Grafcet,

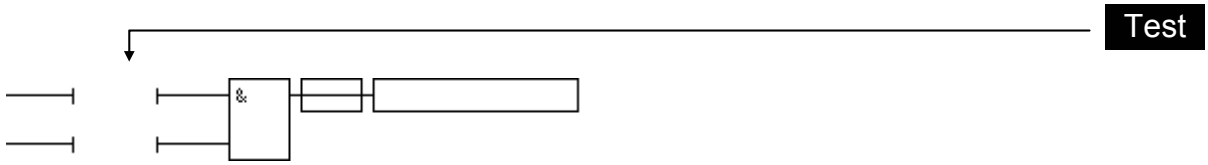


⇨ les conditions sur action du langage Grafcet,

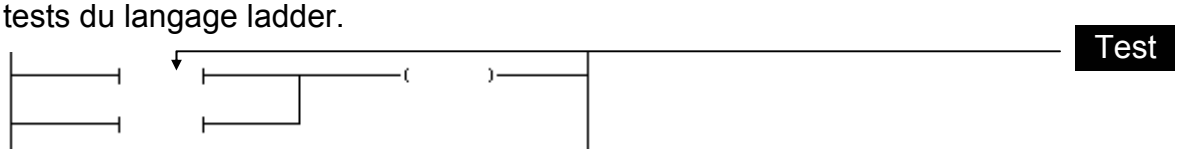




⇨ les tests du langage logigramme,



⇨ les tests du langage ladder.



1.3.1. Forme générale

Un test est une équation booléenne composée de une ou de n variables séparées par des opérateurs « + » (ou) ou « . » (et).

Exemple de test :

```
i0 (test l'entrée 0)
i0+i2 (test l'entrée 0 « ou » l'entrée 2)
i10.i11 (test l'entrée 10 « et » l'entrée 11)
```

1.3.2. Modificateur de test

Par défaut, si seul le nom d'une variable est spécifié, le test est « si égal à un» (si vrai). Des modificateurs permettent de tester l'état complémenté, le front montant et le front descendant :

⇨ le caractère « / » placé devant une variable teste l'état complémenté,

⇨ le caractère « u » ou le caractère « ↑* » placé devant une variable teste le front montant,

⇨ le caractère « d » ou le caractère « ↓** » placé devant une variable teste le front descendant.

Les modificateurs de tests peuvent s'appliquer à une variable ou à une expression entre parenthèses.

Exemples :

```
↑ i0
/i1
/(i2+i3)
↓(i2+(i4./i5))
```

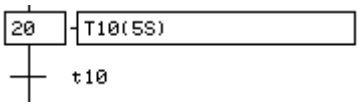
* Pour obtenir ce caractère pendant l'édition d'un test pressez sur la touche [↑].

** Pour obtenir ce caractère pendant l'édition d'un test pressez sur la touche [↓].

1.3.3. Temporisations

Quatre syntaxes sont disponibles pour les temporisations.

Dans la première, on active la temporisation dans l'action et on mentionne simplement la variable temporisation dans un test pour vérifier l'état de fin :



Pour les autres, tout est écrit dans le test. La forme générale est :

« temporisation / variable de lancement / durée »

ou

« durée / variable de lancement / temporisation »

ou

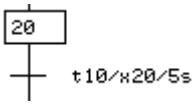

« durée / variable de lancement »

Dans ce cas, une temporisation est automatiquement attribuée. La plage d'attribution est celle des symboles automatiques voir chapitre 1.4.2. Symboles automatiques.

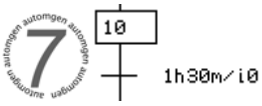
La durée est par défaut exprimée en dixièmes de seconde.


La durée peut être exprimée en jours, heures, minutes, secondes et millisecondes avec les opérateurs « d », « h », « m », « s » et « ms ». Par exemple : 1d30s = 1 jour et 30 secondes.

Exemple utilisant la deuxième syntaxe :



Exemple utilisant la syntaxe normalisée :



1.3.4. Priorité des opérateurs booléens

Par défaut l'opérateur booléen «. » (ET) à une priorité supérieure à l'opérateur '+' (OU). Des parenthèses peuvent être utilisées pour définir une autre priorité.

Exemples :

```
i0.(i1+i2)
((i0+i1).i2)+i5
```

1.3.5. Test toujours vrai

La syntaxe du test toujours vrai est :

« » (néant) ou « =1 »

1.3.6. Test sur variable numérique

Les tests sur variable numérique doivent utiliser la syntaxe suivante :

« variable numérique » « type de test » « constante ou variable numérique »

Le type de test peut être :

⇨ « = » égal,

⇨ « ! » ou « <> » différent,

⇨ « < » inférieur (non signé),

⇨ « > » supérieur (non signé),

⇨ « << » inférieur (signé),

⇨ « >> » supérieur (signé),

⇨ « <= » inférieur ou égal (non signé),

⇨ « >= » supérieur ou égal (non signé),

⇨ « <<= » inférieur ou égal (signé),

⇨ « >>= » supérieur ou égal (signé).

Un flottant ne peut comparé qu'avec un autre flottant ou une constante réelle.

Un long ne peut être comparé qu'avec un autre long ou une constante longue.

Un mot ou un compteur ne peut être comparé qu'avec un mot, un compteur ou une constante 16 bits.

Les constantes réelles doivent être suivies du caractères « R ».

Les constantes longues (32 bits) doivent être suivies du caractère « L ».

Les constantes entières 16 ou 32 bits sont écrite en décimal par défaut. Elle peuvent être écrites en hexadécimal (suffixe « $ » ou « 16# ») ou en binaire (suffixe « % » ou « 2# »).

Les tests sur variables numériques sont utilisés dans les équations comme les tests sur variables booléennes. Ils peuvent être utilisés avec les modificateurs de test à condition d'être encadrés par des parenthèses.

Exemples :

```
m200=100
%mw1000=16#abcd
c10>20.c10<100
f200=f201
m200=m203
%md100=%md102
f200=3.14r
l200=$12345678L
m200<<-100
m200>>1000
%mw500<=12
/(m200=4)
🡻(m200=100)
/(l200=100000+l200=-100000)
```

1.3.7. Transitions sur plusieurs lignes

Le texte des transitions peut être étendu sur plusieurs lignes. La fin d'une ligne de transition doit obligatoirement être un opérateur « . » ou « + ». Les combinaisons de touche [CTRL] + [🡻] et [CTRL] + [🡹] permettent de déplacer le curseur d'une ligne à l'autre.

1.4. Utilisation de symboles

Les symboles permettent d'associer un texte à une variable.
Les symboles peuvent être utilisés avec tous les langages.
Un symbole doit être associé à une et une seule variable.

1.4.1. Syntaxe des symboles

Les symboles sont composés de :

⇨ un caractère « _ » optionnel (souligné, généralement associé à la touche [8] sur les claviers) qui marque le début du symbole,

⇨ le nom du symbole,

⇨ un caractère « _ » optionnel (souligné) qui marque la fin du symbole.

Les caractères « _ » encadrant les noms de symboles sont optionnels. Il doivent être utilisés si le symbole commence par un chiffre ou un opérateur (+,-, etc…).

1.4.2. Symboles automatiques

Il est parfois fastidieux de devoir définir l'attribution entre chaque symbole et une variable, notamment si l'attribution précise d'un numéro de variable importe peu. Les symboles automatiques sont une solution à ce problème, ils permettent de laisser le soin au compilateur de générer automatiquement l'attribution d'un symbole à un numéro de variable. Le type de variable à utiliser est, quant à lui, fourni dans le nom du symbole.

1.4.3. Syntaxe des symboles automatiques

La syntaxe des symboles automatiques est la suivante :

```
_« nom du symbole » %« type de variable »_
```

« type de variable » peut être :

I , O ou Q, U ou M, T, C, M ou MW, L ou MD, F ou MF.

Il est possible de réserver plusieurs variables pour un symbole. Ceci est utile pour définir des tableaux. Dans ce cas la syntaxe est :

```
_« nom du symbole » %« type de variable »« longueur »_
```

« longueur » représente le nombre de variables à réserver.

1.4.4. Comment le compilateur gère-t-il les symboles automatiques ?

Au début de la compilation d'une application, le compilateur efface tous les symboles automatiques qui se trouvent dans le fichier « .SYM » de l'application. A chaque fois que le compilateur rencontre un symbole automatique, il crée une attribution unique pour ce symbole en fonction du type de variable spécifié dans le nom du symbole. Le symbole ainsi généré, est écrit dans le fichier « .SYM ». Si un même symbole automatique est présent plusieurs fois dans une application, il fera référence à la même variable.

1.4.5. Plage d'attribution des variables

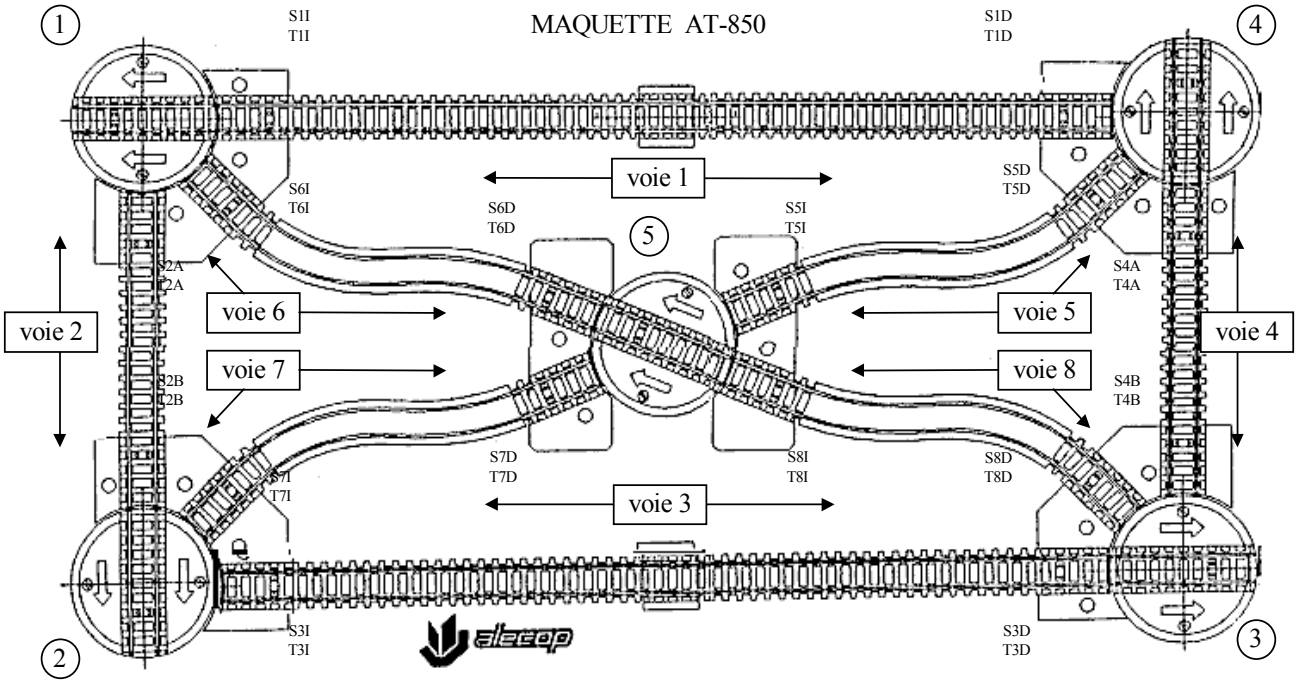
Par défaut, une plage d'attribution est définie pour chaque type de variable :

Type	Début	Fin
I ou %I	0	9999
O ou %Q	0	9999
U ou %M	100	9999
T ou %T	0	9999
C ou %C	0	9999
M ou %MW	200	9999
L ou %MD	100	4998
F ou %MF	100	4998

La plage d'attribution est modifiable pour chaque type de variable en utilisant la directive de compilation #SR« type »=« début », « fin »
« type » désigne le type de variable, début et fin, les nouvelles bornes à utiliser.
Cette directive modifie l'attribution des variables automatiques pour la totalité du folio où elle est écrite et jusqu'à une prochaine directive « #SR ».

1.5. A propos des exemples

Pour mieux illustrer ce manuel, nous avons développé des exemples fonctionnant avec une maquette de train dont voici le schéma :



Nous avons utilisé des cartes d'E/S sur PC pour piloter cette maquette. Les symboles définis par le constructeur de la maquette ont été conservés.

Le fichier de symboles suivant a été créé :

AV1	O0	alimentation voie 1
AV2	O1	alimentation voie 2
AV3	O2	alimentation voie 3
AV4	O3	alimentation voie 4
AV5	O4	alimentation voie 5
AV6	O5	alimentation voie 6
AV7	O6	alimentation voie 7
AV8	O7	alimentation voie 8
AP1	O8	alimentation plateforme 1
AP2	O9	alimentation plateforme 2
AP3	O10	alimentation plateforme 3
AP4	O11	alimentation plateforme 4
AP5	O12	alimentation plateforme 5
IP1	O13	rotation plateforme 1
IP2	O14	rotation plateforme 2
IP3	O15	rotation plateforme 3
IP4	O16	rotation plateforme 4
IP5	O17	rotation plateforme 5
ZP1	O18	initialisation plateforme 1
ZP2	O19	initialisation plateforme 2
ZP3	O20	initialisation plateforme 3
ZP4	O21	initialisation plateforme 4
ZP5	O22	initialisation plateforme 5
DV1	O23	direction voie 1
DV2	O24	direction voie 2
DV3	O25	direction voie 3
DV4	O26	direction voie 4
DV5	O27	direction voie 5
DV6	O28	direction voie 6
DV7	O29	direction voie 7
DV8	O30	direction voie 8
S1D	O31	feu droit voie 1
S1I	O32	feu gauche voie 1
S2A	O33	feu haut voie 2
S2B	O34	feu bas voie 2
S3D	O35	feu droit voie 3
S3I	O36	feu gauche voie 3
S4A	O37	feu haut voie 4
S4B	O38	feu bas voie 4
S5D	O39	feu droit voie 5
S5I	O40	feu gauche voie 5
S6D	O41	feu droit voie 6
S6I	O42	feu gauche voie 6
S7D	O43	feu droit voie 7
S7I	O44	feu gauche voie 7
S8D	O45	feu droit voie 8
S8I	O46	feu gauche voie 8
T1D	i0	train droit voie 1
T1I	i1	train gauche voie 1
T2A	i2	train haut voie 2
T2B	i3	train bas voie 2
T3D	i4	train droit voie 3
T3I	i5	train gauche voie 3
T4A	i6	train haut voie 4
T4B	i7	train bas voie 4
T5D	i8	train droit voie 5

T5I	i9	train gauche voie 5
T6D	i10	train droit voie 6
T6I	i11	train gauche voie 6
T7D	i12	train droit voie 7
T7I	i13	train gauche voie 7
T8D	i14	train droit voie 8
T8I	i15	train gauche voie 8
TP1	i16	train plateforme 1
TP2	i17	train plateforme 2
TP3	i18	train plateforme 3
TP4	i19	train plateforme 4
TP5	i20	train plateforme 5
P1P	i21	index plateforme 1
P2P	i22	index plateforme 2
P3P	i23	index plateforme 3
P4P	i24	index plateforme 4
P5P	i25	index plateforme 5
P1Z	i26	init plateforme 1
P2Z	i27	init plateforme 2
P3Z	i28	init plateforme 3
P4Z	i29	init plateforme 4
P5Z	i30	init plateforme 5
ERR	i31	court-circuit

1.6. Grafcet

AUTOMGEN supporte les éléments suivants :

- ⇨ divergences et convergences en « Et » et en « Ou »,
- ⇨ étapes puits et sources,
- ⇨ transitions puits et sources,
- ⇨ synchronisation,
  - ⇨ forçages de Grafcets,
  - ⇨ mémorisation de Grafcets,
  - ⇨ figeage,
  - ⇨ macro-étapes.

1.6.1. Grafcet simple

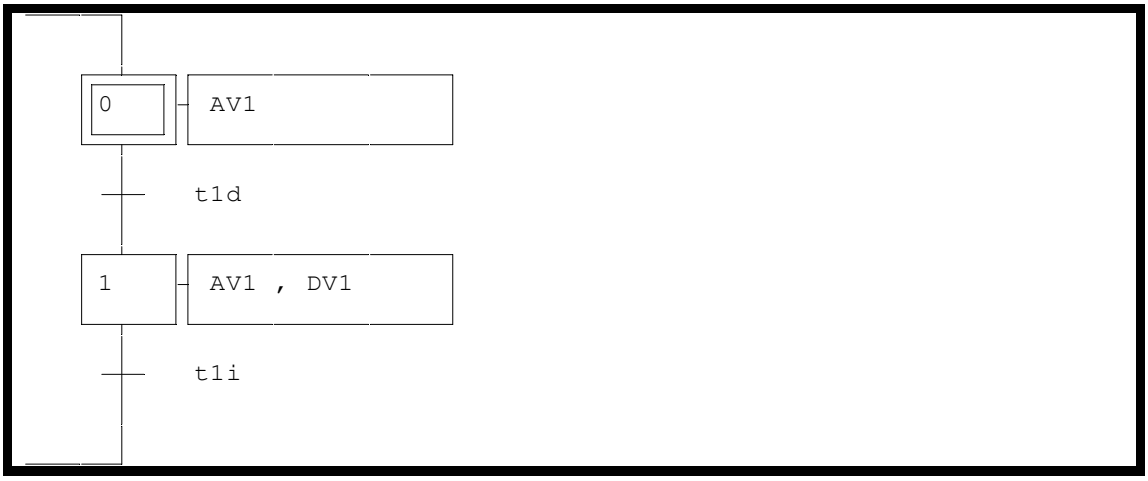
L'écriture de Grafcet en ligne se résume à la juxtaposition d'étapes et de transitions.

Illustrons un Grafcet en ligne avec l'exemple suivant :

Cahier des charges :

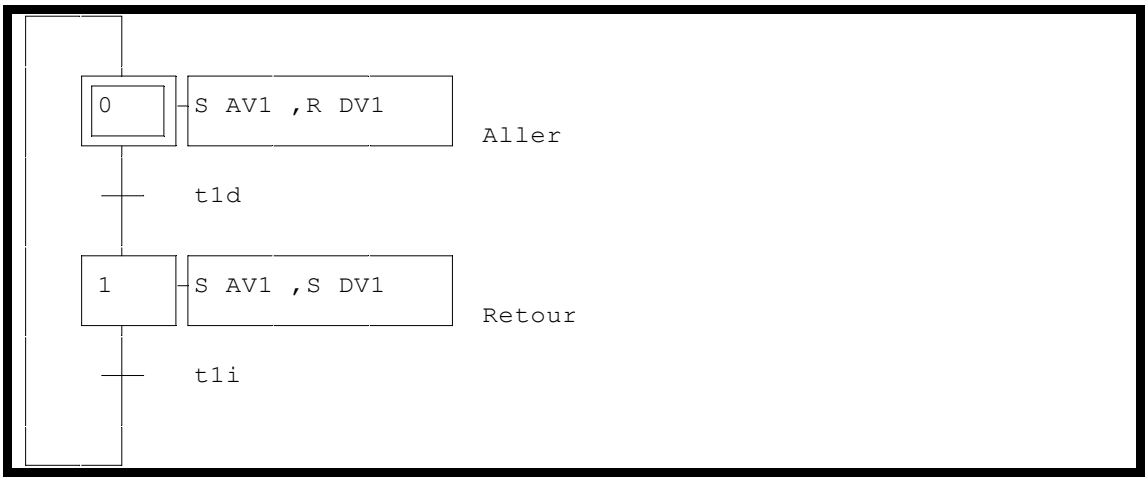
La locomotive doit partir sur la voie 3 vers la droite, jusqu'au bout de la voie. Elle revient ensuite dans le sens inverse jusqu'à l'autre bout et recommence.

Solution 1 :



exemples\grafcet\simple1.agn

Solution 2 :

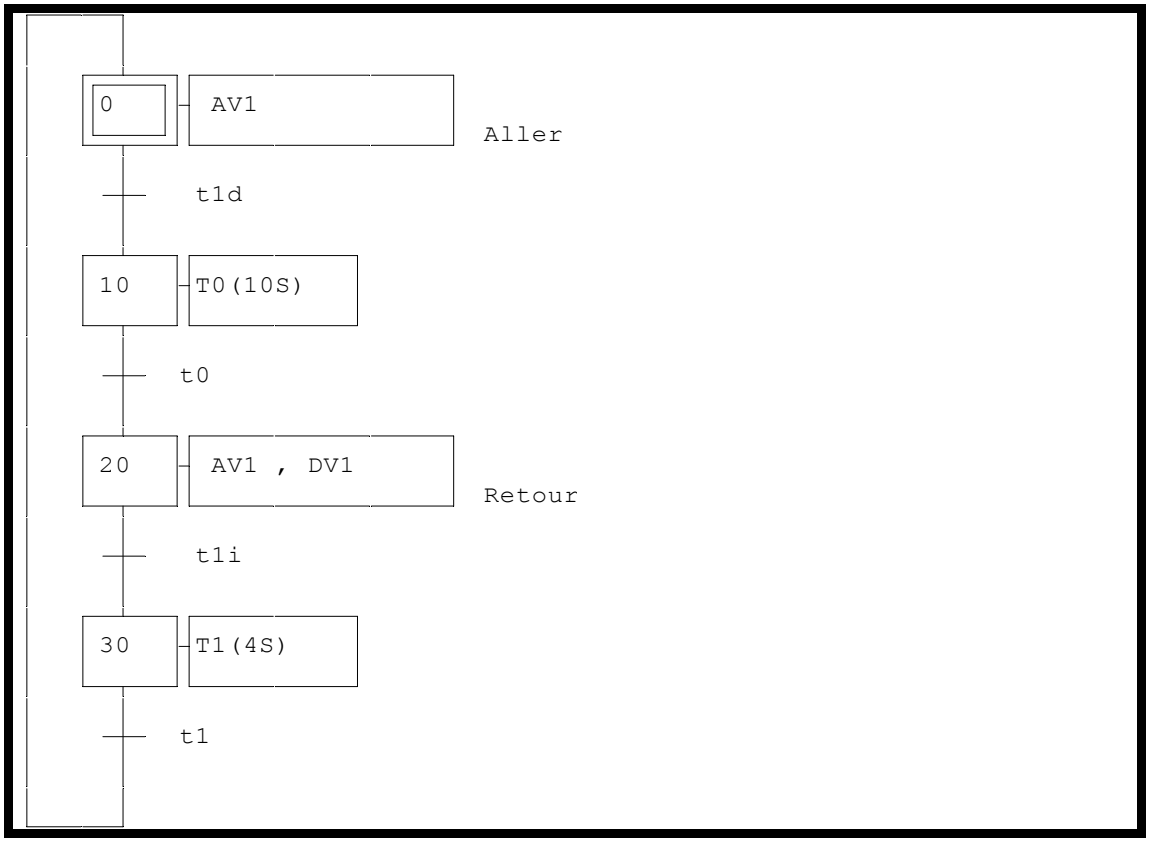


exemple\grafcet\simple2.agn

La différence entre ces deux solutions réside dans l'utilisation des actions « Affectation » pour le premier exemple et des actions « Mise à un » et « Mise à zéro » pour le deuxième.

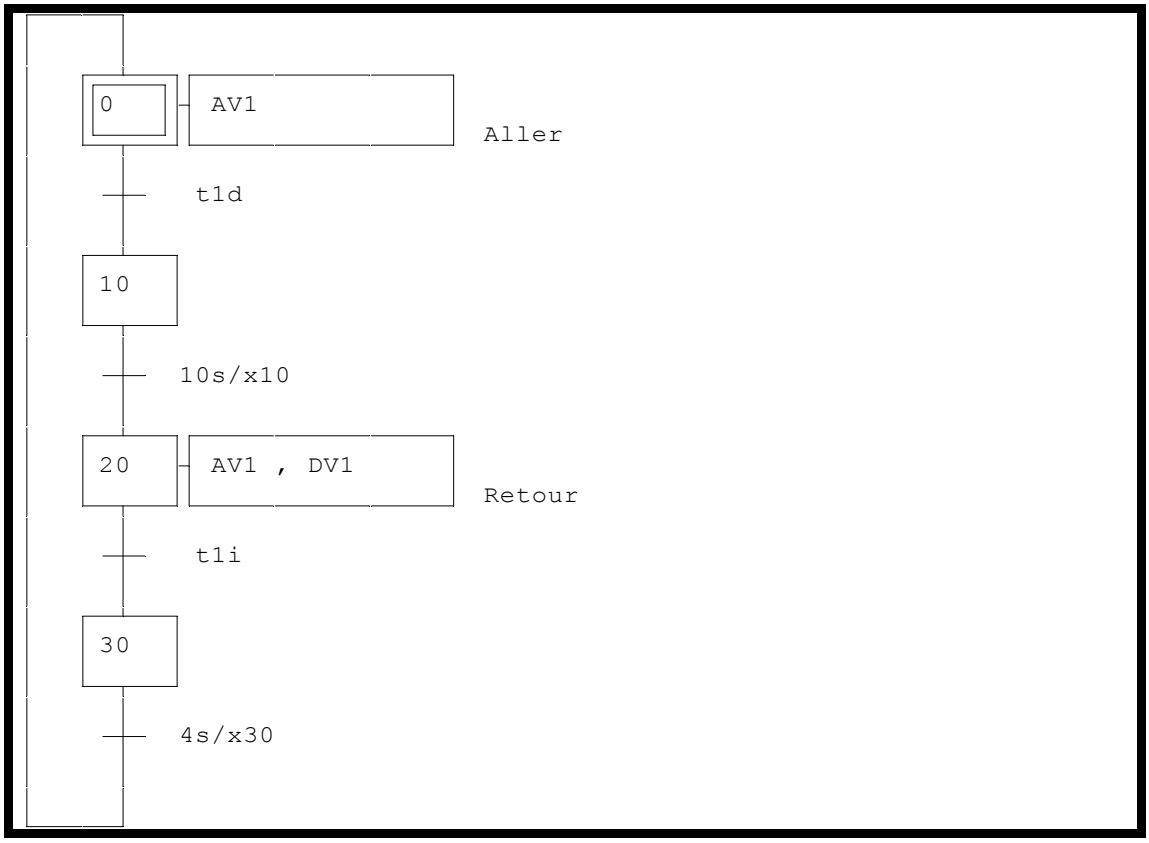
Modifions notre cahier des charges en imposant une attente de 10 secondes lorsque la locomotive arrive à droite de la voie 1 et une attente de 4 secondes lorsque la locomotive arrive à gauche de la voie 1.

Solution 1 :



exemple\grafcet\simple3.agn

Solution 2 :

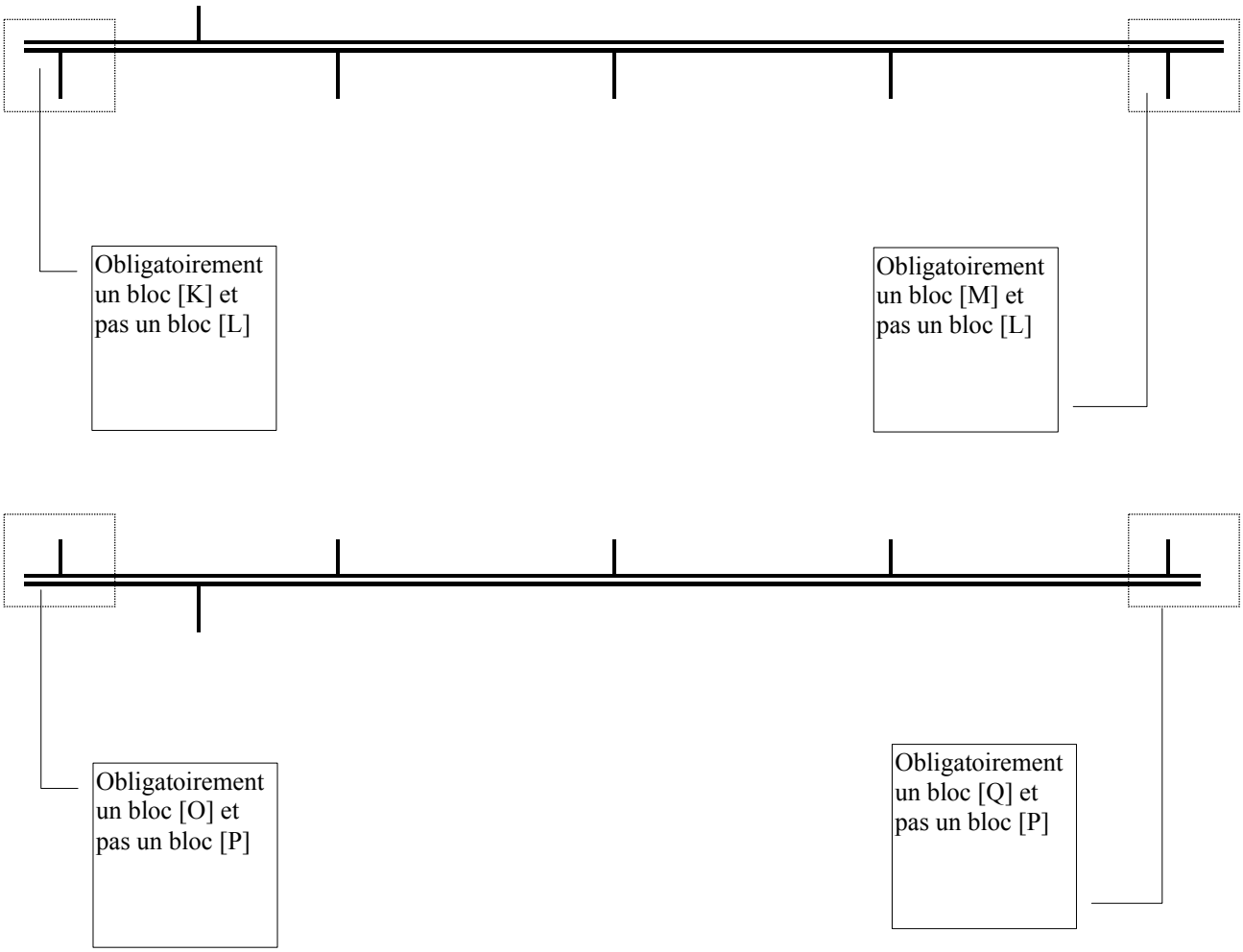


exemple\grafcet\simple4.agn

La différence entre les exemples 3 et 4 réside dans le choix de la syntaxe utilisée pour définir les temporisations. Le résultat au niveau du fonctionnement est identique.

1.6.2. Divergence et convergence en « Et »

Les divergences en « Et » peuvent avoir n branches. L'important est de respecter l'utilisation des blocs de fonction :

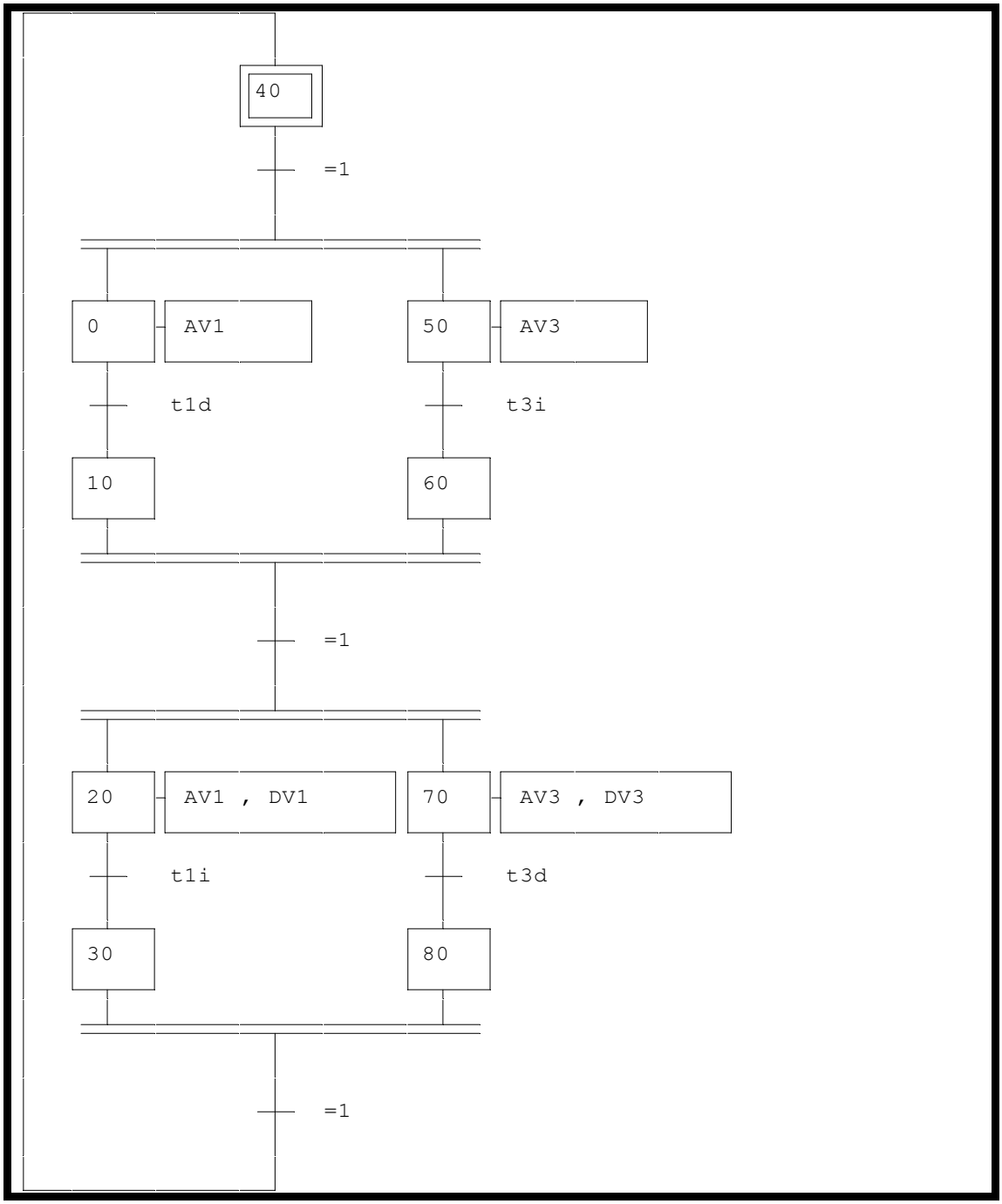


Illustrons l'utilisation des divergences et convergences en « Et ».

Cahier des charges :

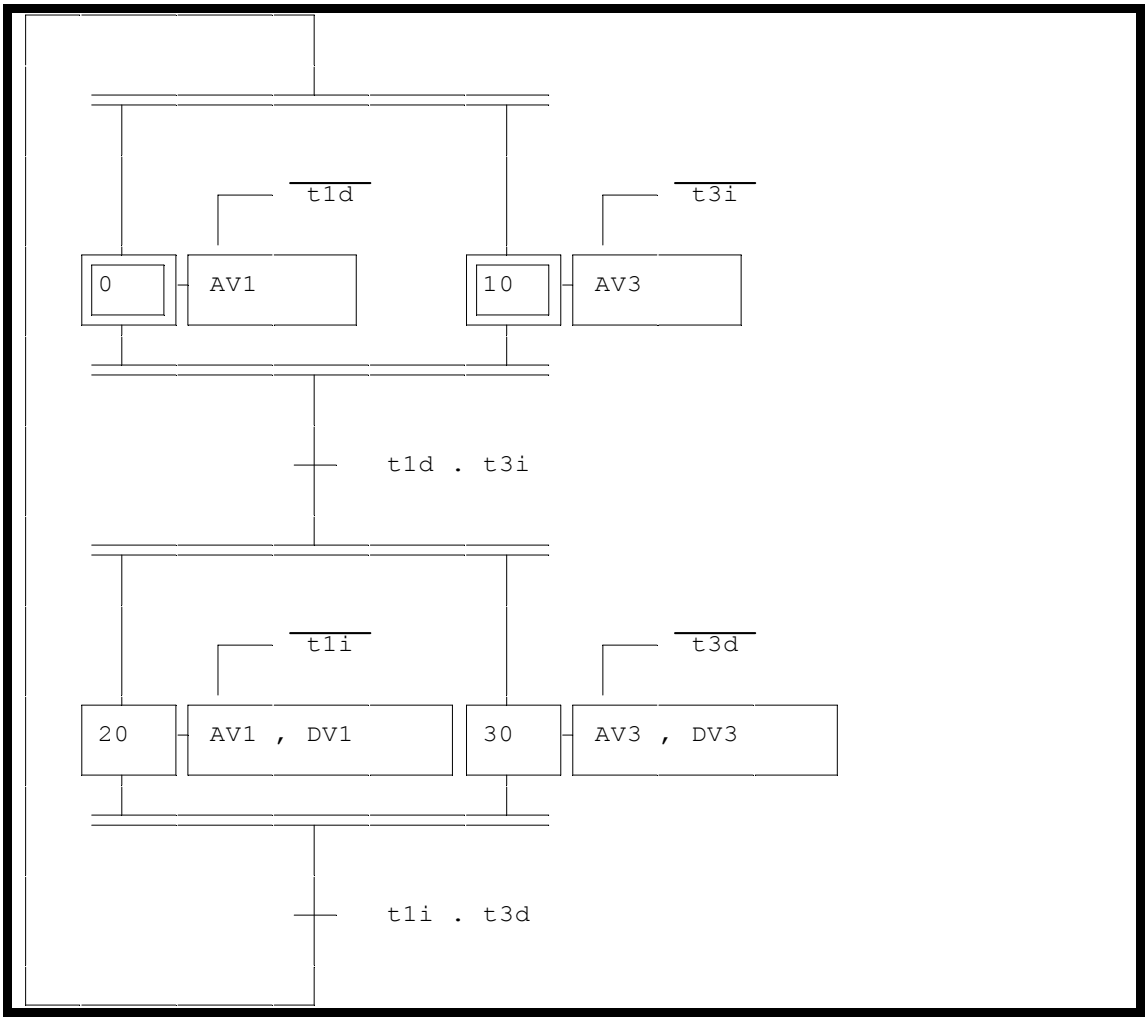
Nous allons utiliser deux locomotives : la première effectuera des allers et retours sur la voie 1, la seconde sur la voie 3. Les deux locomotives seront synchronisées (elles s'attendront en bout de voie).

Solution 1 :



exemple\grafcet\divergence et 1.agn

Solution 2 :

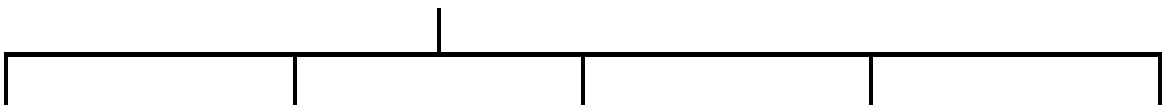


exemple\grafcet\divergence et 2.agn

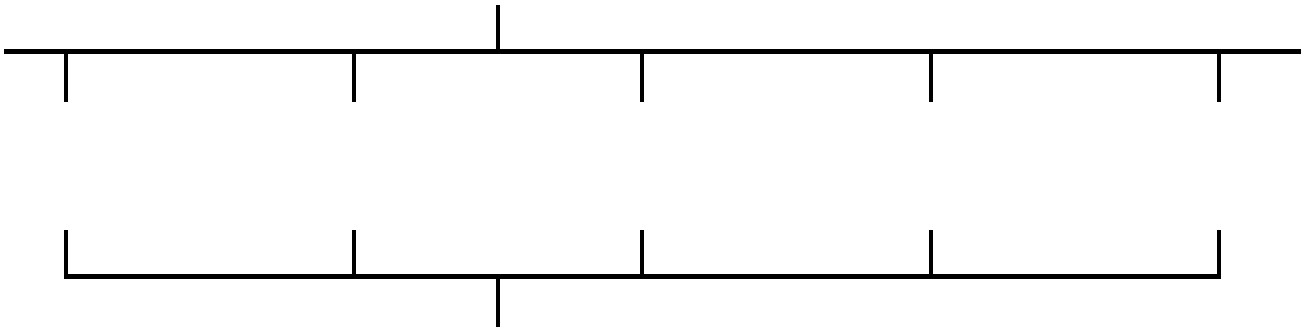
Ces deux solutions sont équivalentes au niveau du fonctionnement. La deuxième est une version plus compacte qui utilise des actions conditionnées.

1.6.3. Divergence et convergence en « Ou »

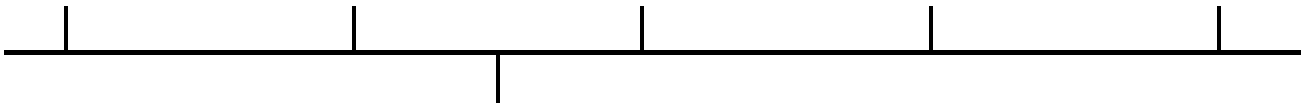
Les divergences en « Ou » peuvent avoir n branches. L'important est de respecter l'utilisation des blocs de fonction :



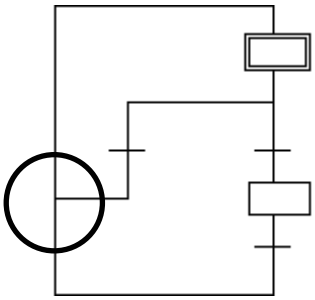
ou



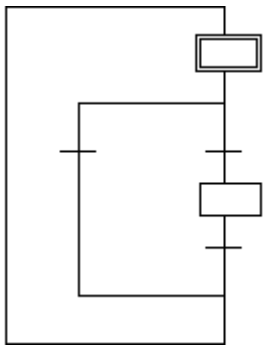
ou



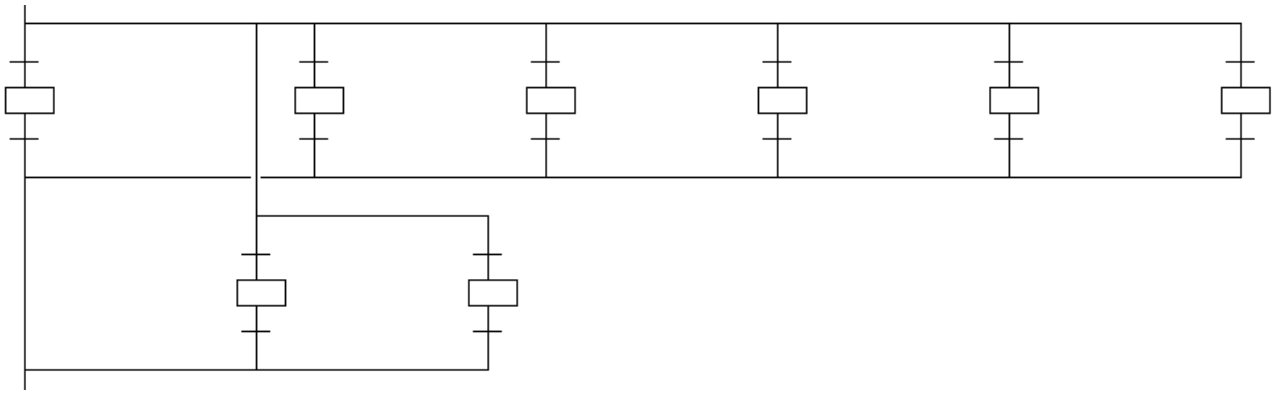
Les divergences en « Ou » doivent obligatoirement se brancher sur des liaisons descendantes. Par exemple :



incorrect, le bon dessin est :



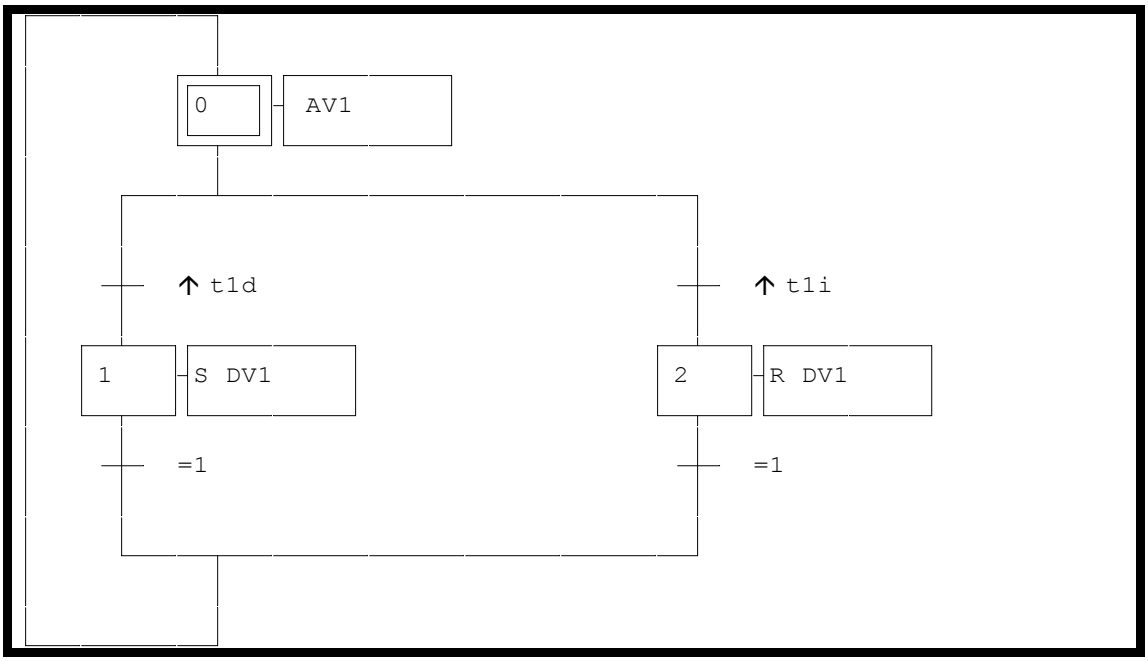
Si la largeur de la page vous interdit l'écriture d'un grand nombre de divergences, vous pouvez adopter une structure du type:



Voyons un exemple pour illustrer l'utilisation des divergences et convergences en « Ou » :

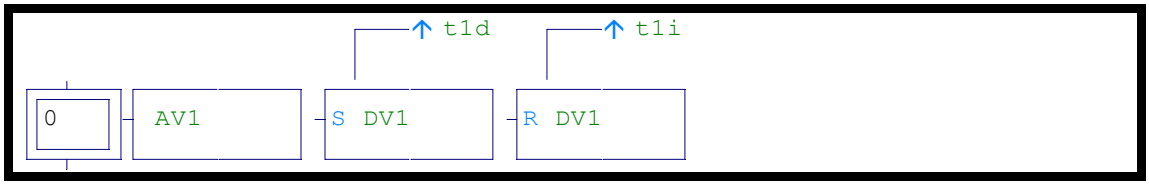
Cahier des charges :
Reprenons le cahier des charges du premier exemple de chapitre : aller et retour d'une locomotive sur la voie 1.

Solution :



exemple\grafcet\divergence ou.agn

Ce Grafcet pourrait se résumer à une étape en utilisant des actions conditionnées, comme dans cet exemple :



exemple\grafcet\action conditionnée.agn

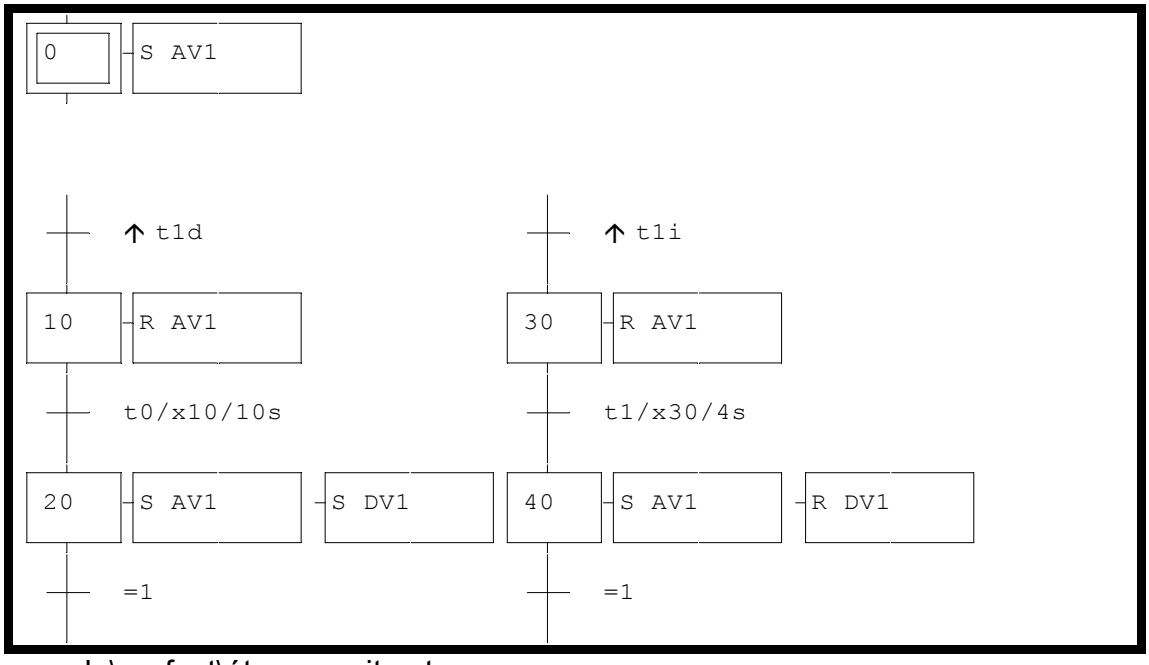
1.6.4. Etapes puits et sources, transitions puits et sources

Illustrons ces principes par des exemples :

Cahier des charges :

Traitons à nouveau le second exemple de ce chapitre : aller et retour d'une locomotive sur la voie 1 avec attente en fin de voie.

Solution :



exemple\grafcet\étapes puits et sources.agn

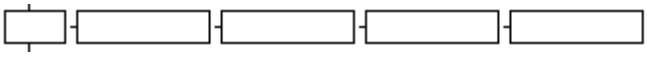
1.6.5. Actions multiples, actions conditionnées

Nous avons déjà utilisé dans ce chapitre des actions multiples et des actions conditionnées. Détaillons ici ces deux principes.

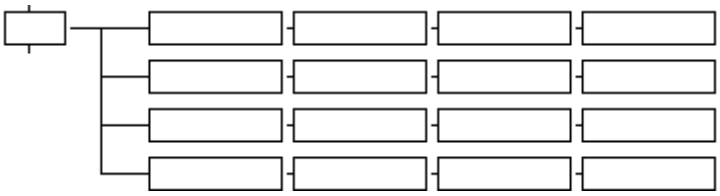
Comme il est dit dans le chapitre consacré au compilateur, plusieurs actions peuvent être écrites dans un même rectangle, le caractère « , » (virgule) sert de délimiteur dans ce cas.

Lorsqu'une condition est ajoutée sur un rectangle d'action, c'est l'ensemble des actions contenues dans ce rectangle qui sont conditionnées.

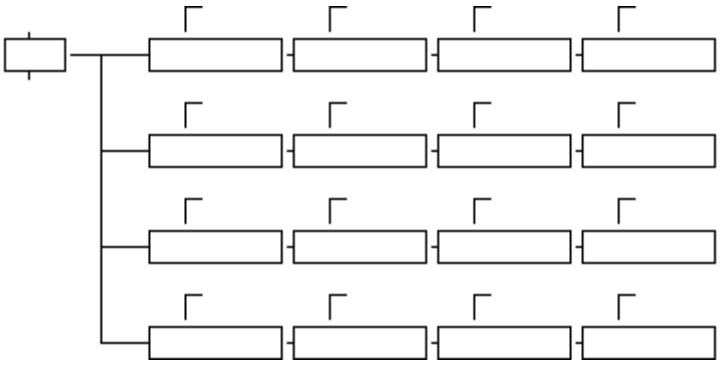
Plusieurs rectangles d'action peuvent être associés à une étape :



autre possibilité :



Chacun des rectangles peut recevoir une condition différente :



Pour dessiner une action conditionnée, placez le curseur sur le rectangle d'action, cliquez sur le bouton droit de la souris et choisissez « Action conditionnelle » dans le menu. Pour documenter la condition sur action, cliquez sur l'élément .

La syntaxe IF(condition) permet d'écrire une condition sur action dans le rectangle d'action.

1.6.6. Synchronisation

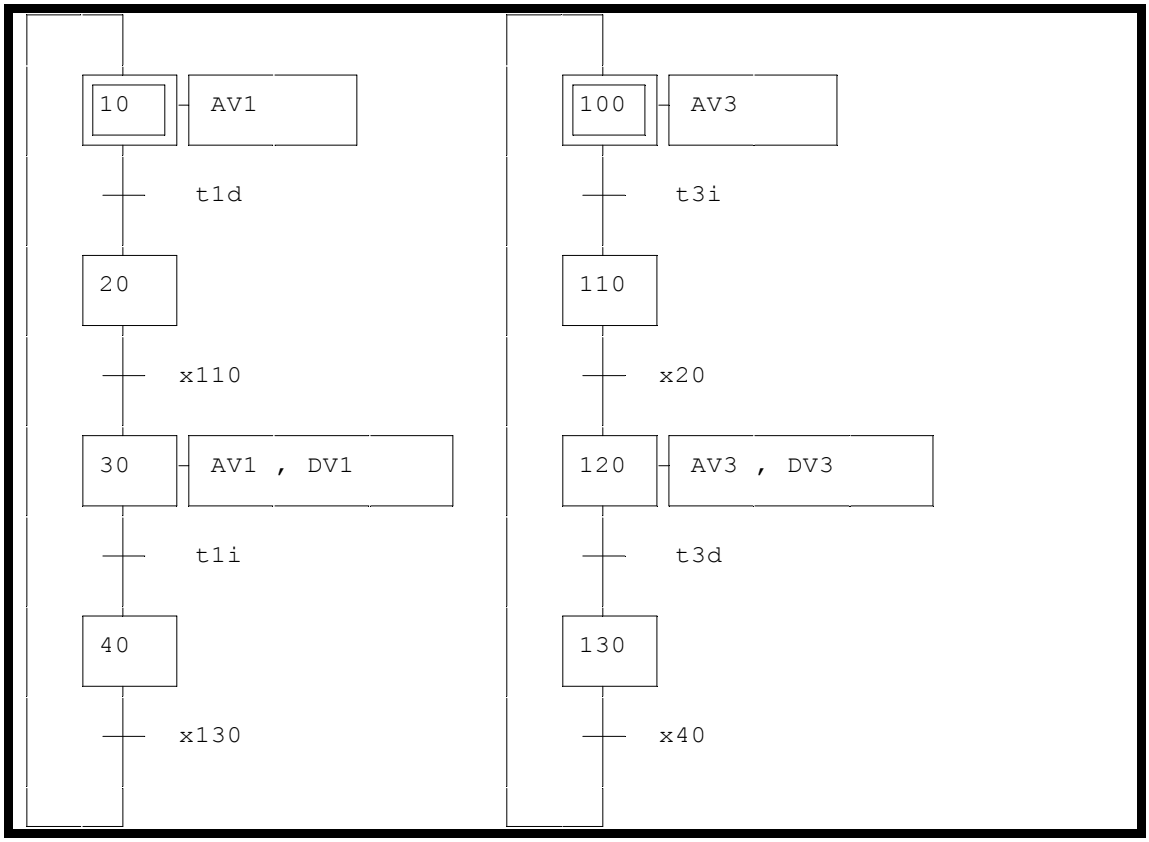
Reprenons un exemple déjà traité pour illustrer la synchronisation de Grafcets.

Cahier des charges:

Aller et retour de deux locomotives sur les voies 1 et 3 avec attente entre les locomotives en bout de voie.

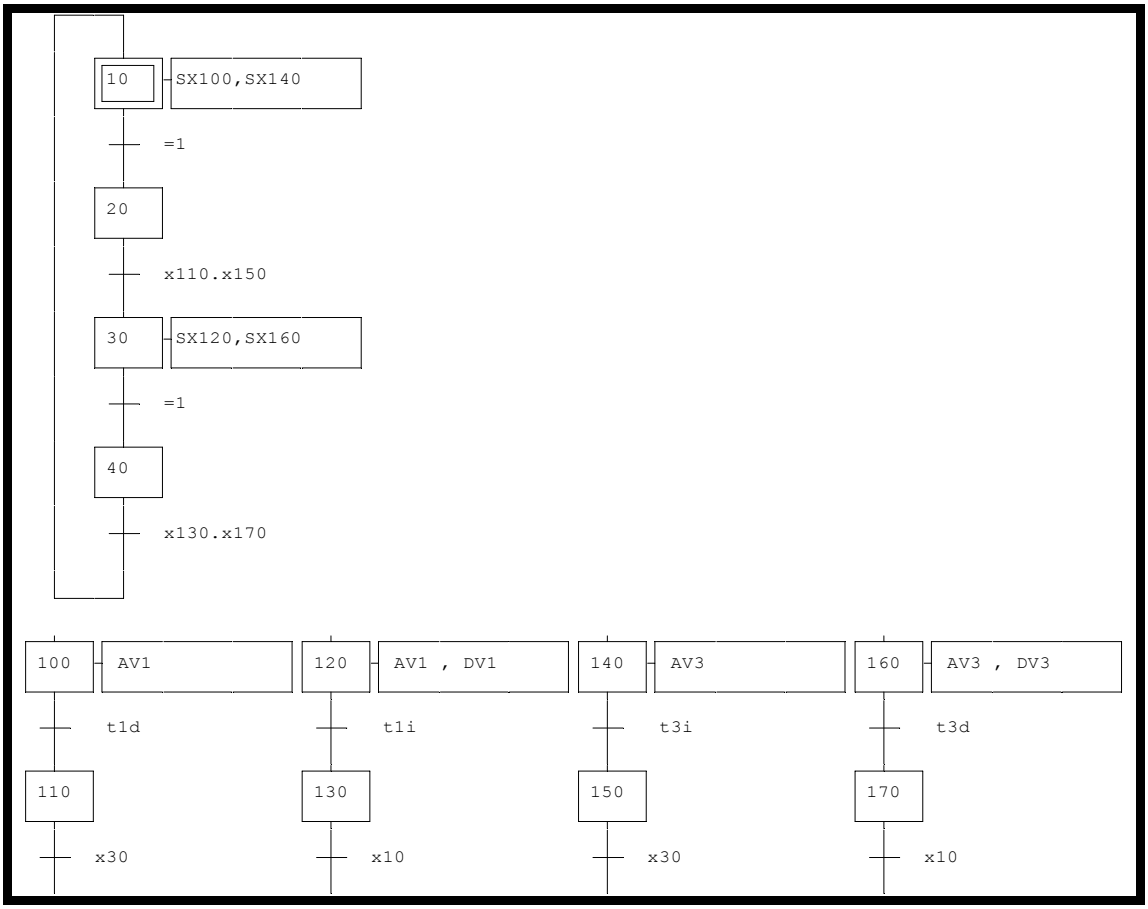
Cet exemple avait été traité avec une divergence en « Et ».

Solution 1 :



exemple\grafcet\synchro1.agn

Solution 2 :



exemple\grafcet\synchro2.agn

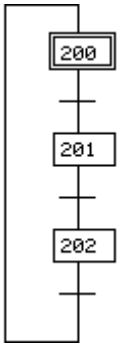
Cette deuxième solution est un excellent exemple illustrant l'art de compliquer les choses les plus simples à des fins pédagogiques.

1.6.7. Forçages de Grafcet

Le compilateur regroupe les étapes en fonction des liens qui sont établis entre elles. Pour désigner un Grafcet, il suffit de faire référence à une des étapes composant ce Grafcet.

On peut également désigner l'ensemble des Grafcets présents sur un folio en mentionnant le nom du folio.

Par exemple:



Pour désigner ce Grafcet nous parlerons du Grafcet 200, du Grafcet 201 ou encore du Grafcet 202.

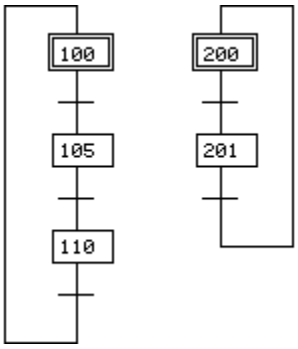
Le Grafcet en tant qu'ensemble d'étapes devient donc une variable de type structurée, composée de n étapes, chacune de ces étapes, étant, soit active, soit inactive.

Comme nous l'avons vu, AUTOMGEN divise les étapes en ensembles indépendants. Ces ensembles peuvent être regroupés, ceci permet donc de les considérer comme un seul Grafcet. Pour regrouper plusieurs Grafcets, il faut utiliser la directive de compilation « `#G:g1,g2` » (commande à inclure dans un commentaire). Cette commande regroupe les Grafcets g1 et g2. Rappelons que la désignation d'un Grafcet s'effectue en invoquant le numéro d'une de ses étapes.

Voyons un exemple :

#G:105,200

cette directive de compilation regroupe ces deux Grafcets :



Remarque : plusieurs directives « #G » peuvent être utilisées afin de regrouper plus de deux Grafcets.

Nous allons maintenant détailler les ordres de forçage utilisables. Ils seront simplement écrits dans des rectangles d'action comme des affectations classiques. Ils supporteront également les opérateurs S(mise à un), R(mise à

zéro), N(affectation complémentée) et I(inversion) ainsi que les actions conditionnelles.

1.6.7.1. Forçage d'un Grafcet selon une liste d'étapes actives

Syntaxe:

« F<Grafcet>:{<liste d'étapes actives>} »

ou

 « F/<nom de folio>:{<liste d'étapes actives>} »

Le ou les Grafcets ainsi désignés seront forcés à l'état défini par la liste des étapes actives se trouvant entre accolades. Si plusieurs étapes doivent être actives alors il faut les séparer par le caractère « , » (virgule). Si le ou les Grafcets doivent être forcés à l'état vide (aucune étape active) alors aucune étape ne doit être précisée entre les deux accolades.

 Le numéro des étapes peut être précédé de « X ». On peut ainsi associer un symbole au nom d'une étape.

Exemples :

« F10:{0} »

force toutes les étapes du Grafcet 10 à 0 sauf l'étape 0 qui sera forcée à 1.

« F0:{4,8,9,15} »

force toutes les étapes du Grafcet 0 à 0 sauf les étapes 4,8,9 et 15 qui seront forcées à 1.

« F/marche normale :{} »

force tous les Grafcets se trouvant sur le folio « marche normale » à l'état vide.

1.6.7.2. Mémorisation de l'état d'un Grafcet

Etat actuel d'un Grafcet:

Syntaxe:

« G<Grafcet>:<N° de bit> »

ou

 « G/<nom de folio>:<N° de bit> »

Cette commande mémorise l'état d'un ou plusieurs Grafcets dans une série de bits. Il est nécessaire de réserver un espace au stockage de l'état du ou des

Grafcets désignés (un bit par étape). Ces bits de stockage doivent être consécutifs. Vous devez utiliser une commande #B pour réserver un espace linéaire de bit.

 Le numéro de l'étape désignant le Grafcet peut être précédé de « X » . On peut ainsi associer un symbole au nom d'une étape. Le numéro du bit peut être précédé de « U » ou de « B ». On peut ainsi associer un symbole au premier bit de la zone de stockage d'état.

Etat particulier d'un Grafcet :
Syntaxe:
« G<Grafcet>:<N° de bit> {liste d'étapes actives} »
ou

 « G/<nom de folio> :<N° de bit> {liste d'étapes actives} »

Cette commande mémorise l'état défini par la liste d'étapes actives appliquées aux Grafcets spécifiés à partir du bit indiqué. Il est nécessaire ici aussi de réserver un nombre suffisant de bits. Si une situation vide doit être mémorisée alors aucune étape ne doit apparaître entre les deux accolades.

 Le numéro des étapes peut être précédé de « X ». On peut ainsi associer un symbole au nom d'une étape. Le numéro du bit peut être précédé de « U » ou de « B ». On peut ainsi associer un symbole au premier bit de la zone de stockage d'état.

Exemples:
« G0:100 »
mémorise l'état actuel du Grafcet 0 à partir de U100.
« G0:U200 »
mémorise l'état vide du Grafcet 0 à partir de U200.
« G10:150{1,2} »
mémorise l'état du Grafcet 10, dans lequel seules les étapes 1 et 2 sont actives, à partir de U150.
« G/PRODUCTION :_SAUVE ETAT PRODUCTION_ »
mémorisé l'état des Grafcets se trouvant sur le folio « PRODUCTION » dans la variable _SAUVE ETAT PRODUCTION_.

1.6.7.3. Forçage d'un Grafcet à partir d'un état mémorisé

Syntaxe:

« F<Grafcet>:<N° de bit> »

ou

 « F/<Nom de folio>:<N° de bit> »

Force le ou les Grafcets avec l'état mémorisé à partir du bit précisé.

Le numéro de l'étape désignant le Grafcet peut être précédé de 'X'. On peut ainsi associer un symbole au nom d'une étape. Le numéro du bit peut être précédé de « U » ou de « B ». On peut ainsi associer un symbole au premier bit de la zone de stockage d'état.

Exemple:

« G0:100 »

mémorise l'état actuel du Grafcet 0

« F0:100 »

et restaure cet état

1.6.7.4. Figeage d'un Grafcet

Syntaxe:

« F<Grafcet> »

ou

 « F/<Nom de folio> »

Fige un ou des Grafcets : interdit toute évolution de ceux-ci.

Exemple :

« F100 »

fige le Grafcet 100

« F/production »

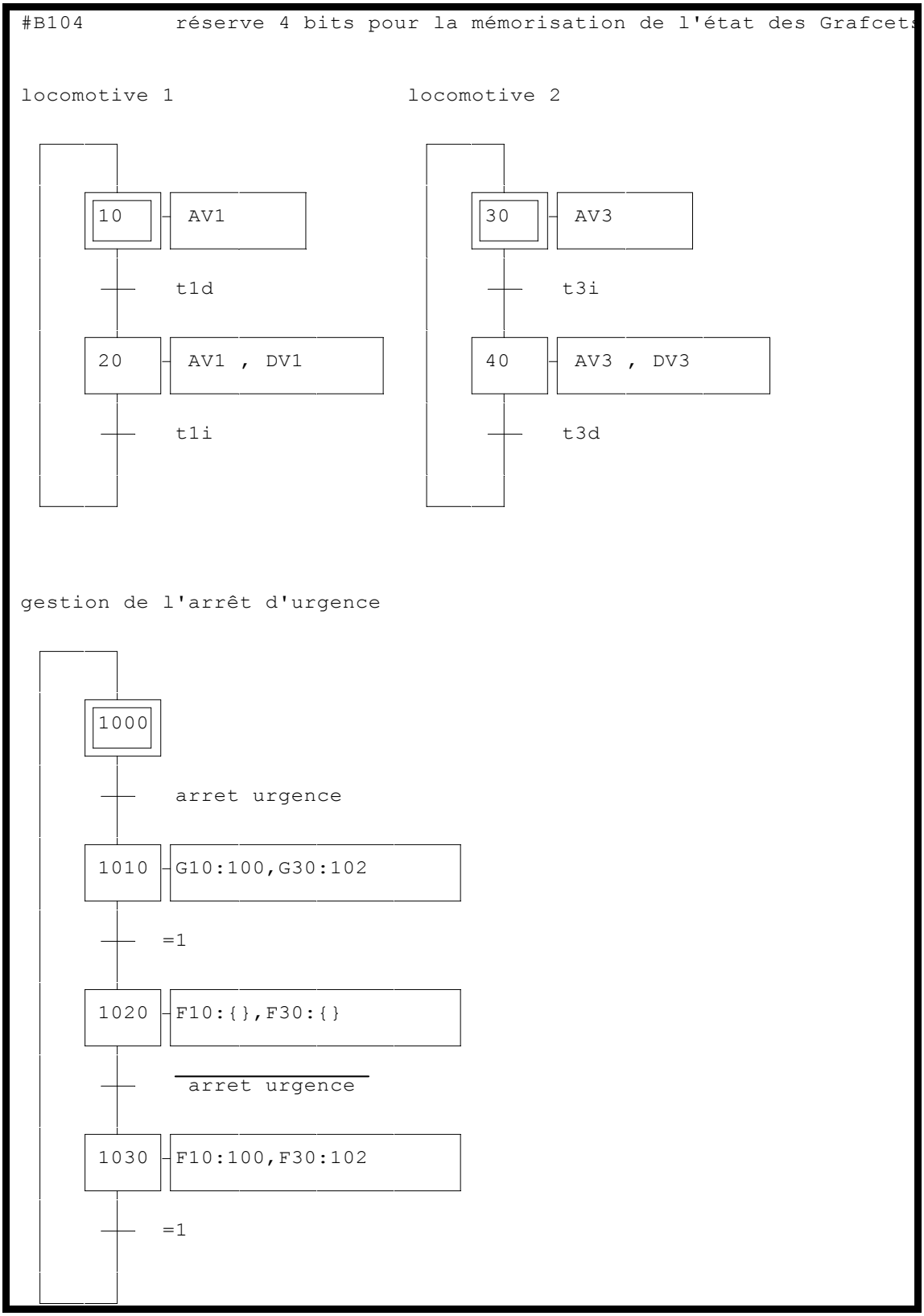
fige les Grafcets contenus dans le folio « production »

Illustrons les forçages par un exemple.

Cahier des charges :

Reprenons un exemple déjà traité : aller et retour de deux locomotives sur les voies 1 et 3 (cette fois sans attente entre les locomotives) et ajoutons le traitement d'un arrêt d'urgence. Lorsque l'arrêt d'urgence est détecté toutes les sorties sont remises à zéro. A la disparition de l'arrêt d'urgence le programme doit reprendre là ou il s'était arrêté.

Solution 1 :

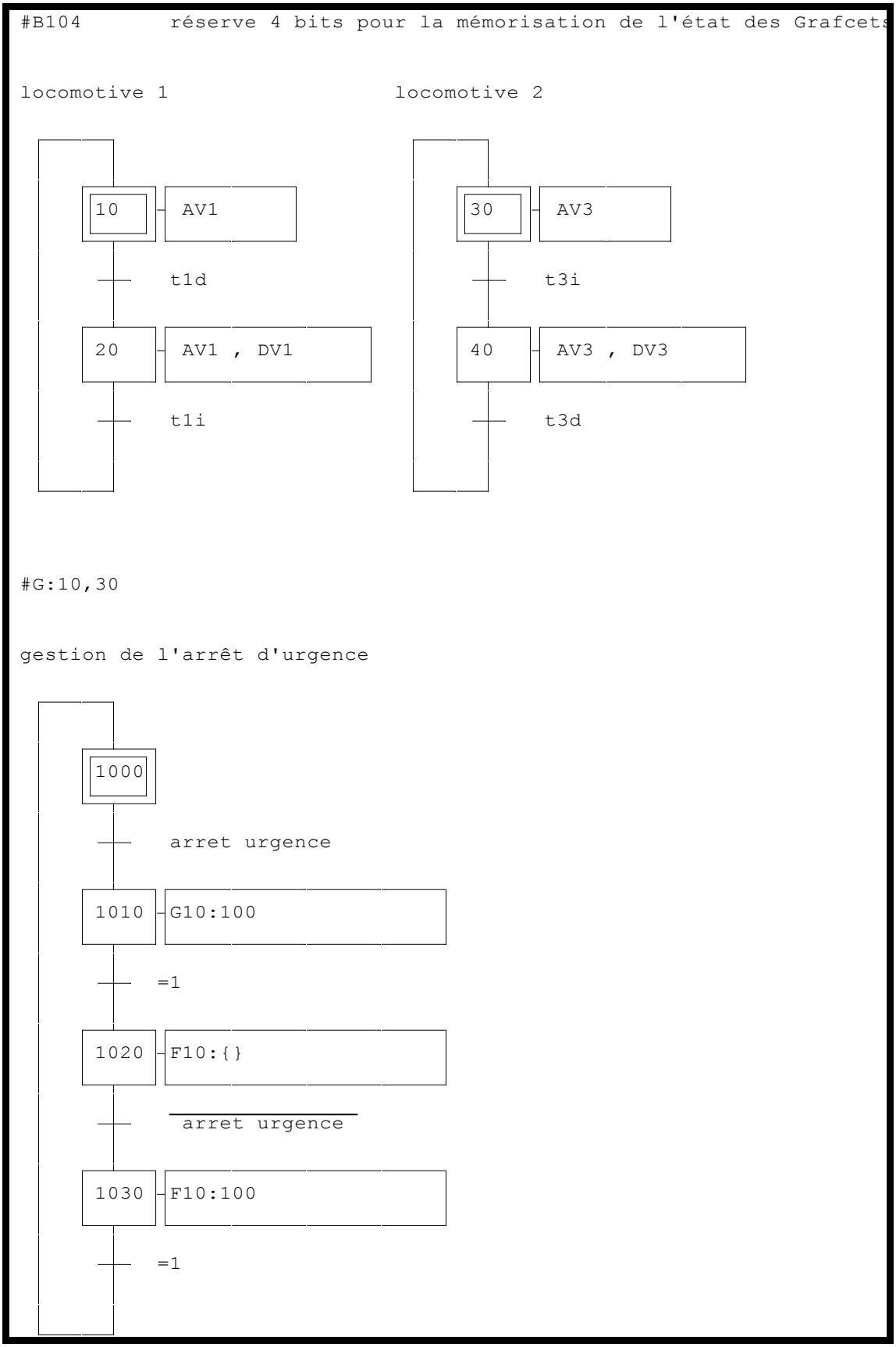


exemple\grafcet\forçage1.agn

Notez l'utilisation de la directive #B104 qui permet de réserver quatre bits consécutifs (U100 à U103) pour la mémorisation de l'état des deux Grafcets.

« _arret urgence_ » a été associé à un bit (U1000). Son état peut donc être modifié à partir de l'environnement en cliquant dessus lorsque la visualisation dynamique est activée.

Solution 2 :



exemple\grafcet\forçage2.agn

Cette deuxième solution montre l'utilisation de la directive de compilation « #G » qui permet de regrouper les Grafcets pour les commandes de forçages.

1.6.8. Macro-étapes

AUTOMGEN implémente les macro-étapes.
Donnons quelques rappels à ce sujet :
Une macro-étape ME est l'unique représentation d'un ensemble unique d'étapes et de transitions nommé « expansion de ME ».
Une macro-étape obéit aux règles suivantes :

- ⇨ l'expansion de ME comporte une étape particulière dite **étape d'entrée** et une étape particulière dite **étape de sortie**.
- ⇨ l'étape d'entrée a la propriété suivante : tout franchissement d'une transition amont de la macro-étape, active l'étape d'entrée de son expansion.
- ⇨ l'étape de sortie a la propriété suivante : elle participe à la validation des transitions aval de la macro-étape.
- ⇨ en dehors des transitions amont et aval de ME, il n'existe aucune liaison structurale entre, d'une part une étape ou une transition de l'expansion ME, et d'autre part, une étape ou une transition n'appartenant pas à ME.

L'utilisation des macro-étapes sous AUTOMGEN a été définie comme suit :

- ⇨ l'expansion d'une macro-étape est un Grafcet se trouvant dans un folio distinct,
- ⇨ l'étape d'entrée de l'expansion d'une macro-étape devra porter le numéro 0 ou le repère Exxx, (avec xxx = un numéro quelconque),
- ⇨ l'étape de sortie de l'expansion d'une macro-étape devra porter le numéro 9999 ou le repère Sxxx, avec xxx = un numéro quelconque,
- ⇨ en dehors de ces deux dernières obligations, l'expansion d'une macro-étape peut être un Grafcet quelconque et pourra à ce titre contenir des macro-étapes (l'imbrication de macro-étapes est possible).

1.6.8.1. Comment définir une macro-étape ?

Le symbole  doît être utilisé. Pour poser ce symbole, cliquez sur un emplacement vide du folio et choisissez « Plus …/Macro-étape » dans le menu contextuel. Pour ouvrir le menu contextuel, cliquez avec le bouton droit de la souris sur le fond du folio.

Pour définir l'expansion de la macro-étape, créez un folio, dessinez l'expansion et modifier les propriétés du folio (en cliquant avec le bouton droit de la souris sur le nom du folio dans le navigateur). Réglez le type du folio sur « Expansion de macro-étapes » ainsi que le numéro de la macro-étape.

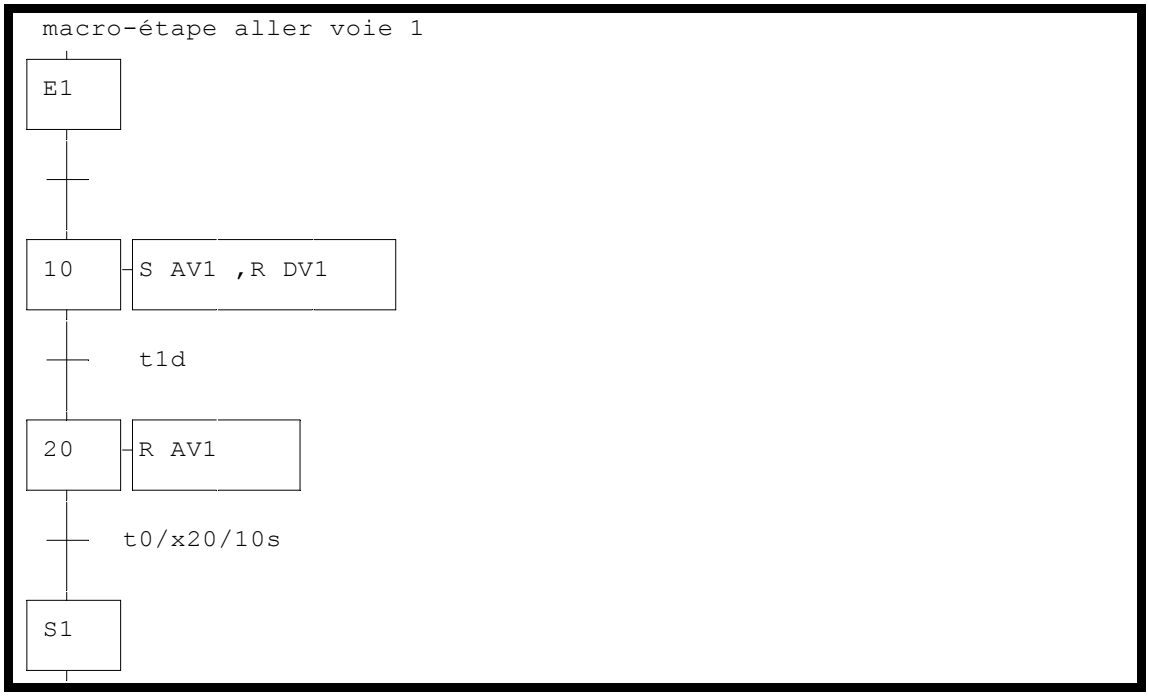
En mode exécution, il est possible de visualiser une expansion de macro-étape. Pour cela, il faut placer le curseur sur la macro-étape et cliquer sur le bouton gauche de la souris.

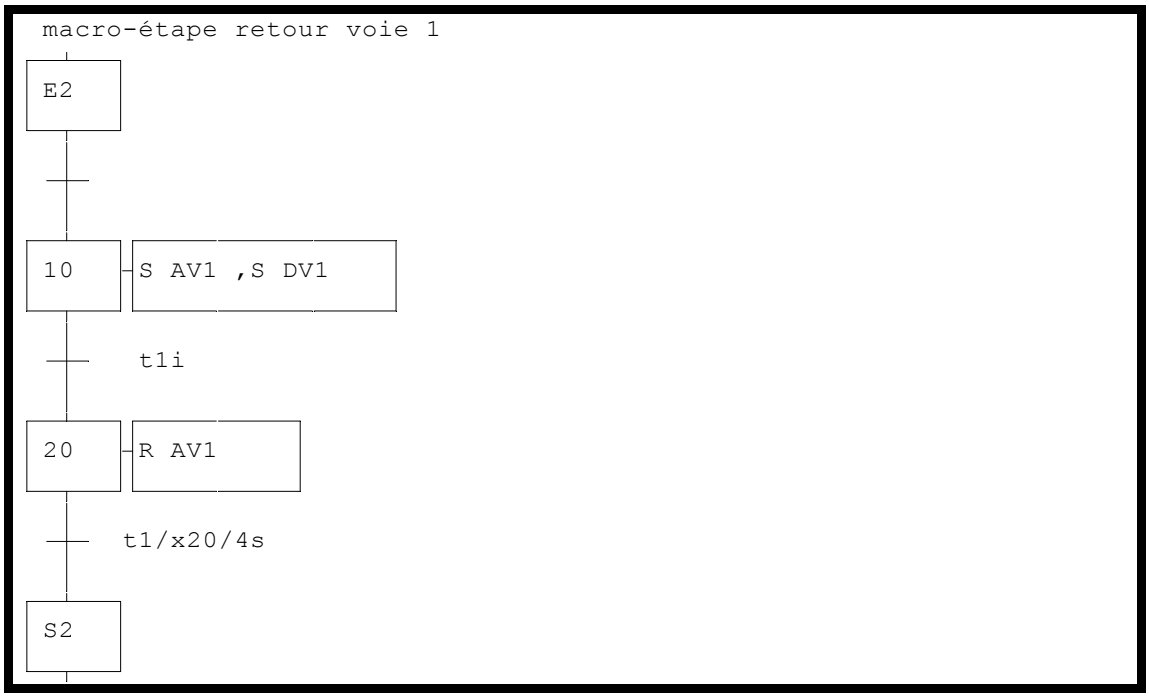
Remarques :

- ⇨ les étapes et les bits Utilisateur utilisés dans une expansion de macro-étape sont locaux, c'est à dire qu'ils n'ont aucun rapport avec les étapes et les bits d'autres Grafcets. Tous les autres types de variables n'ont pas cette caractéristique : ils sont communs à tous les niveaux.
- ⇨ si une zone de bits doit être utilisée de façon globale alors il faut la déclarer avec la directive de compilation « #B ».
- ⇨ l'affectation de variables non locales par différents niveaux ou différentes expansions n'est pas gérée par le système. En d'autres termes, il faut utiliser les affectations « S » « R » ou « I » afin de garantir un fonctionnement cohérent du système.

Illustrons l'utilisation des macro-étapes par un exemple déjà traité : aller et retour d'une locomotive sur la voie 1 avec attente en bout de voie. Nous décomposerons l'aller et le retour en deux macro-étapes distinctes.

Solution :


macro-étape aller voie 1
E1
10
S AV1 ,R DV1
t1d
20
R AV1
t0/x20/10s
S1


macro-étape retour voie 1
E2
10
S AV1 ,S DV1
t1i
20
R AV1
t1/x20/4s
S2

exemple\grafcet\macro-étape.agn

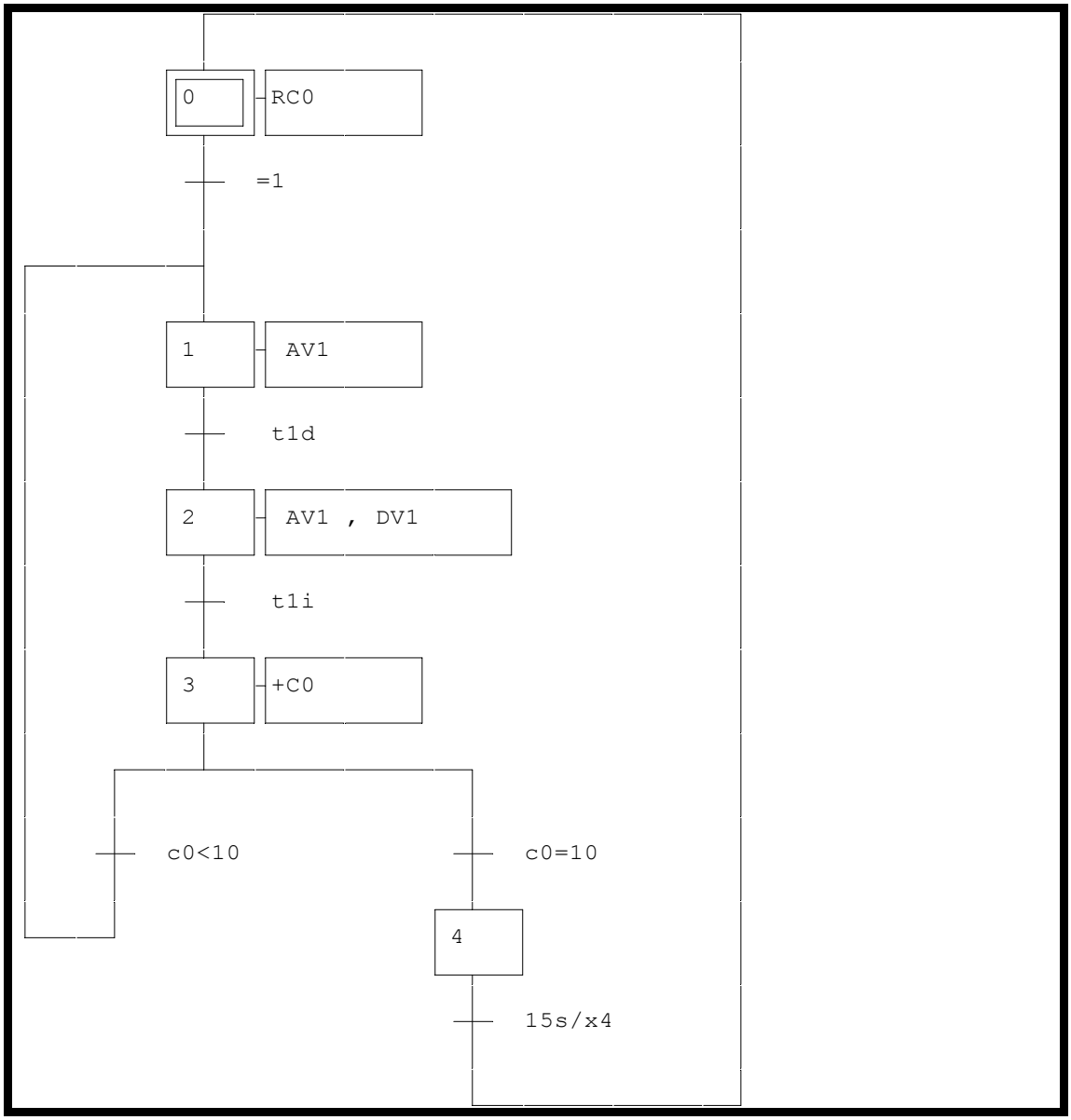
1.6.9. Compteurs

Illustrons l'utilisation des compteurs par un exemple.

Cahier des charges :

Une locomotive doit effectuer 10 allers et retours sur la voie 1, s'immobiliser pendant quinze secondes et recommencer.

Solution :

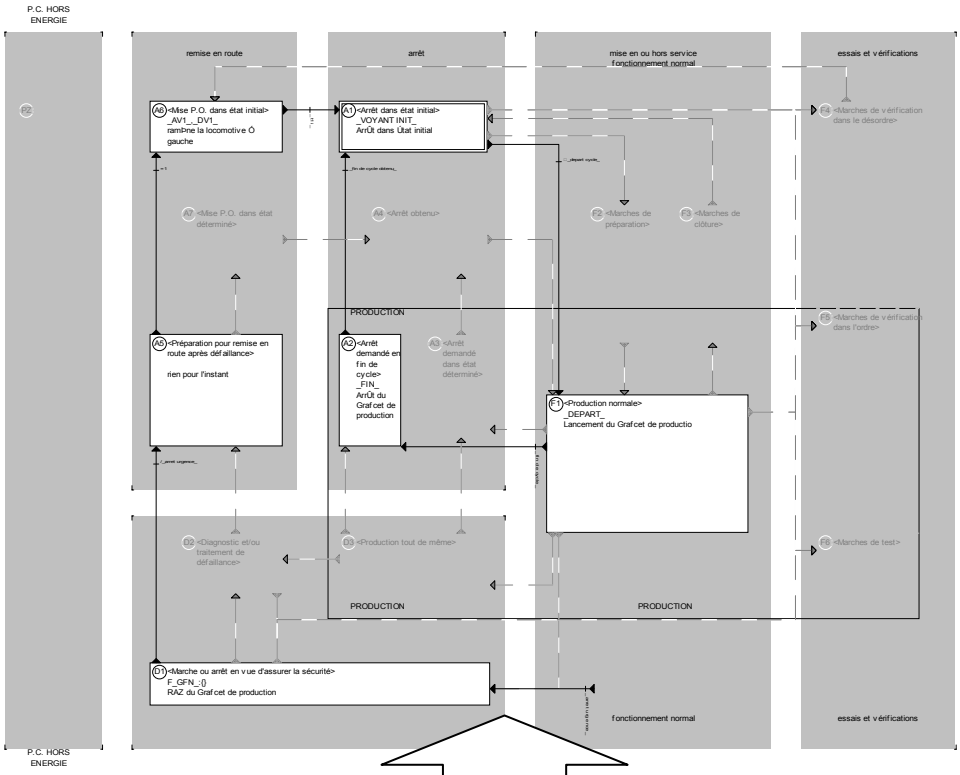


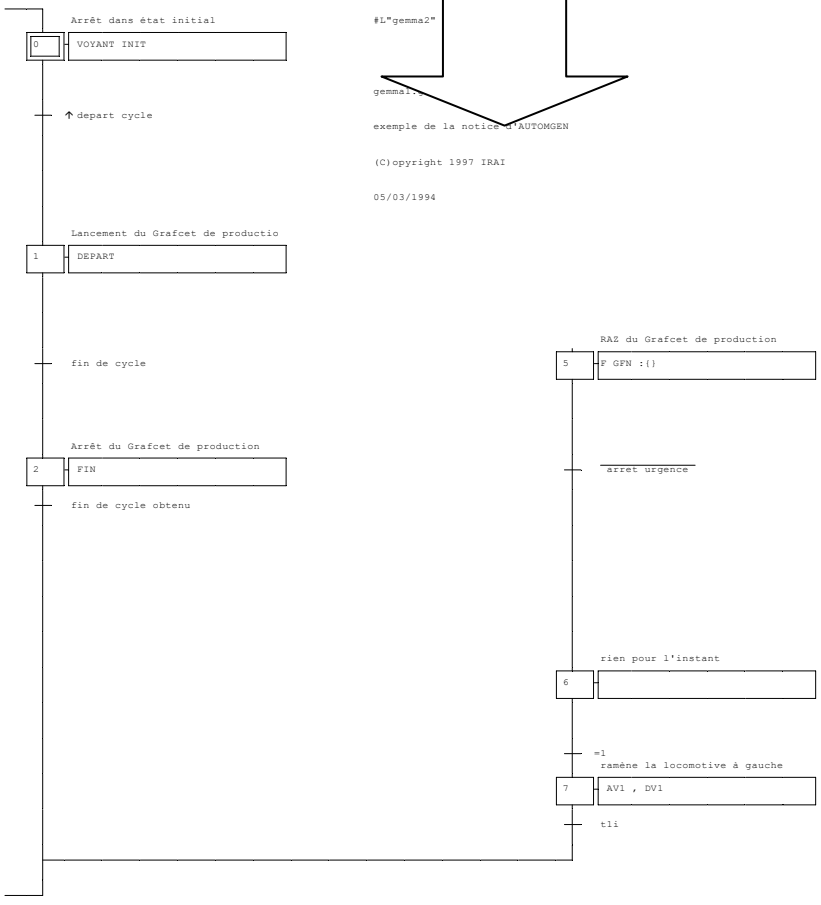
exemple\grafcet\compteur.agn

1.7. Gemma

AUTOMGEN implémente la description de Grafcet de gestion des modes de marche sous forme de Gemma. Le principe retenu est un mode d'édition transparent au mode Grafcet. Il est possible de passer du mode d'édition Grafcet au mode d'édition Gemma. La traduction d'un Gemma en Grafcet de gestion des modes de marche est donc automatique et immédiate.

C'est la commande « Editer sous la forme d'un Gemma » du menu « Boîte à outils » qui permet de passer d'un mode à l'autre.


P.C. HORS ENERGIE
remise en route
arrêt
mise en ou hors service
fonctionnement normal
essais et vérifications
A6 <Mise P.O. dans état initial>
_AV1_,_DV1_
ramène la locomotive Ô gauche
A1 <Arrêt dans état initial>
VOYANT INIT
ArrÛt dans Ùtat initial
E4 <Marches de vérification dans le désordre>
A7 <Mise P.O. dans état déterminé>
A8 <Arrêt obtenu>
F2 <Marches de préparation>
F3 <Marches de clôture>
PRODUCTION
E5 <Marches de vérification dans l'ordre>
A5 <Préparation pour remise en route après défaillance>
rien pour l'instant
A2 <Arrêt demandé en fin de cycle>
FIN
ArrÛt du Grafcet de production
A3 <Arrêt demandé dans état déterminé>
F1 <Production normale>
DEPART
Lancement du Grafcet de productio
D2 <Diagnostic et/ou traitement de défaillance>
D3 <Production tout de même>
E6 <Marches de test>
PRODUCTION
PRODUCTION
D1 <Marche ou arrêt en vue d'assurer la sécurité>
F_GFN_:()
RAZ du Grafcet de production
fonctionnement normal
essais et vérifications
P.C. HORS ENERGIE


Arrêt dans état initial
0
VOYANT INIT
depart cycle
Lancement du Grafcet de productio
1
DEPART
fin de cycle
Arrêt du Grafcet de production
2
FIN
fin de cycle obtenu
#L"gemma2"
gemma1.g
exemple de la notice d'AUTOMGEN
(C)opyright 1997 IRAI
05/03/1994
RAZ du Grafcet de production
5
F_GFN :{}
arret urgence
rien pour l'instant
6
=1
ramène la locomotive à gauche
7
AV1 , DV1
tli

1.7.1. Création d'un Gemma

Pour créer un Gemma il faut :

⇨ cliquez sur l'élément « Folio » du navigateur avec le bouton droit de la souris sélectionner la commande « Ajouter un nouveau folio »,

⇨ dans la liste des tailles choisir « Gemma »,

⇨ cliquer sur le bouton poussoir « OK »,

⇨ cliquez avec le bouton droit de la souris sur le nom du folio ainsi créé dans le navigateur,

⇨ choisissez « Propriétés » dans le menu,

⇨ cochez la case « Afficher sous la forme d'un Gemma ».

La fenêtre contient alors un Gemma dont tous les rectangles et tous les liens sont grisés. Pour valider un rectangle ou une liaison il faut cliquer dessus avec le bouton droit de la souris.

Pour modifier le contenu d'un rectangle ou la nature d'une liaison il faut cliquer dessus avec le bouton gauche de la souris.

Le contenu des rectangles du Gemma sera placé dans des rectangles d'action du Grafcet. La nature des liaisons sera placée dans les transitions du Grafcet.

Un commentaire peut être associé à chaque rectangle du Gemma, il apparaîtra à proximité du rectangle d'action correspondant dans le Grafcet.

1.7.2. Contenu des rectangles du Gemma

Les rectangles du Gemma peuvent recevoir n'importe quelle action utilisable dans le Grafcet. Comme il s'agit de définir une structure de gestion des modes d'arrêt et de marche, il parait judicieux d'utiliser des ordres de forçage vers des Grafcets de plus bas niveau voir le chapitre 1.6.7. Forçages de Grafcet.

1.7.3. Obtenir un Grafcet correspondant

C'est encore la case à cocher « Afficher sous la forme d'un Gemma » dans les propriétés du folio qui permet de revenir à une représentation Grafcet. Il est possible de revenir à tout moment à une représentation Gemma tant que la structure du Grafcet n'a pas été modifiée. Les transitions, le contenu des

rectangles d'action et les commentaires peuvent être modifiés avec une mise à jour automatique du Gemma.

1.7.4. Annuler les espaces vides dans le Grafcet

Il est possible que le Grafcet obtenu occupe plus d'espace que nécessaire sur la page. La commande « Réorganiser la page » du menu « Outils » permet de supprimer tous les espaces non utilisés.

1.7.5. Imprimer le Gemma

Quand l'édition est en mode Gemma c'est la commande « Imprimer » qui permet d'imprimer le Gemma.

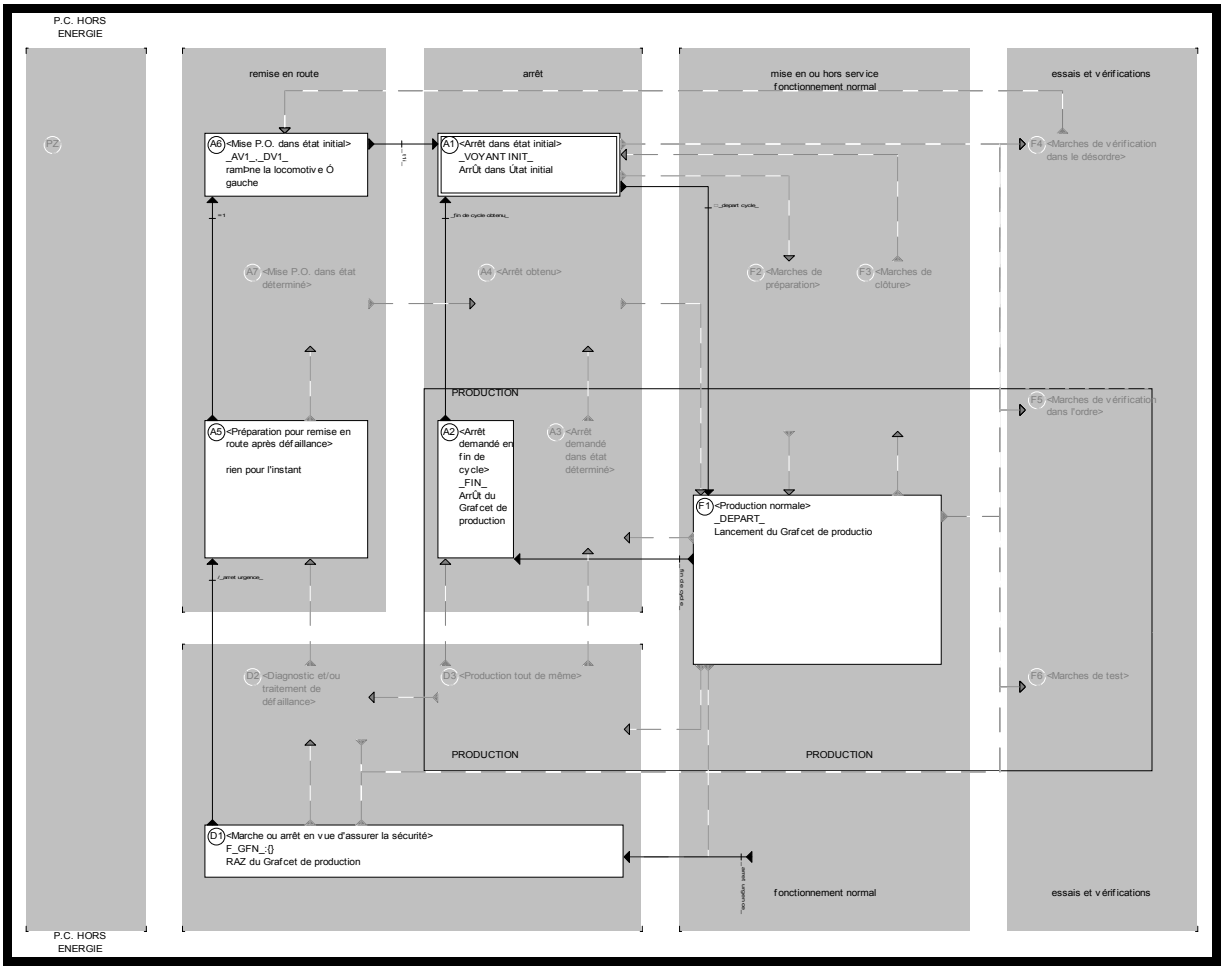
1.7.6. Exporter le Gemma

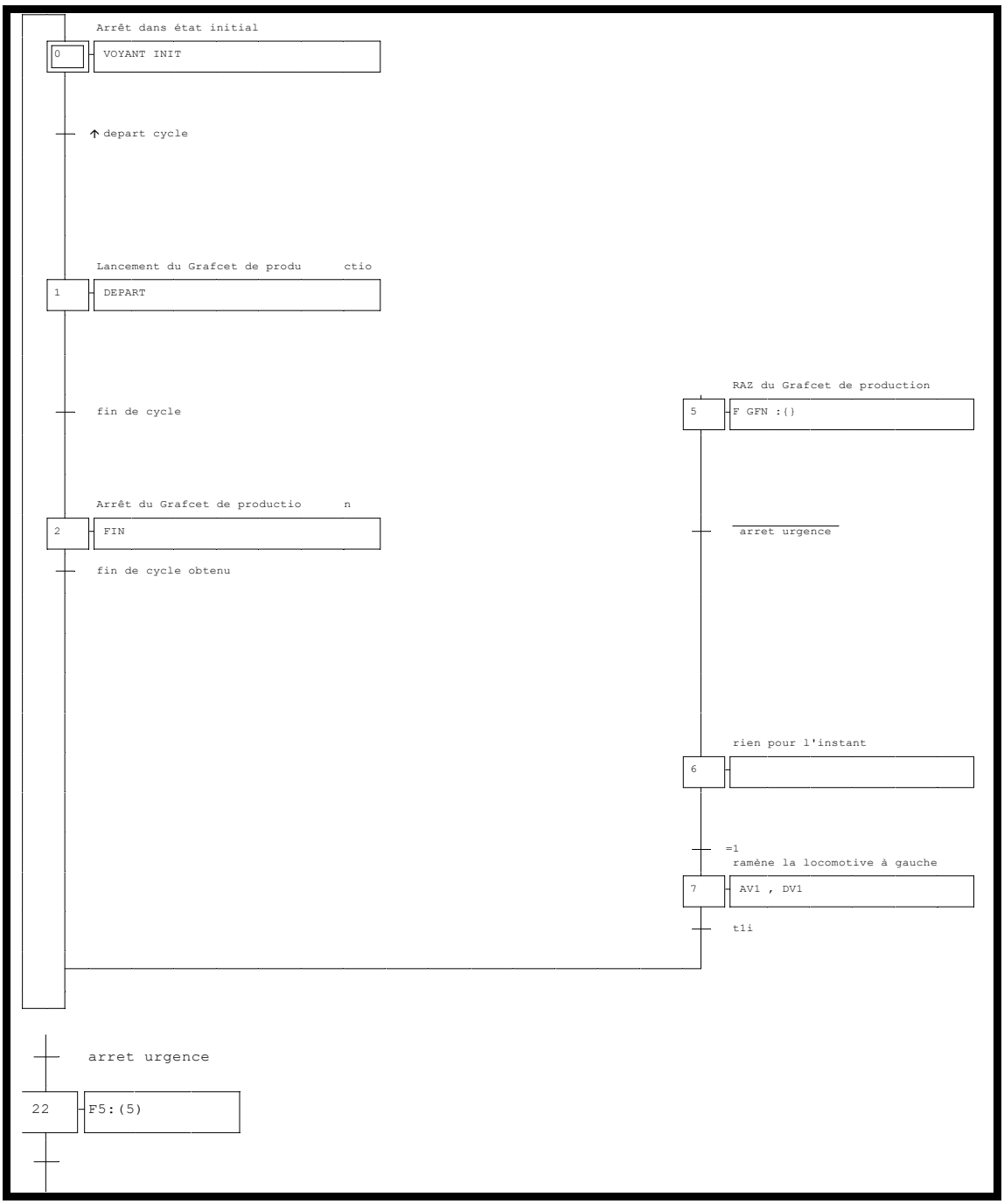
La commande « Copier au format EMF » du menu « Edition » permet d'exporter un Gemma au format vectoriel.

1.7.7. Exemple de Gemma

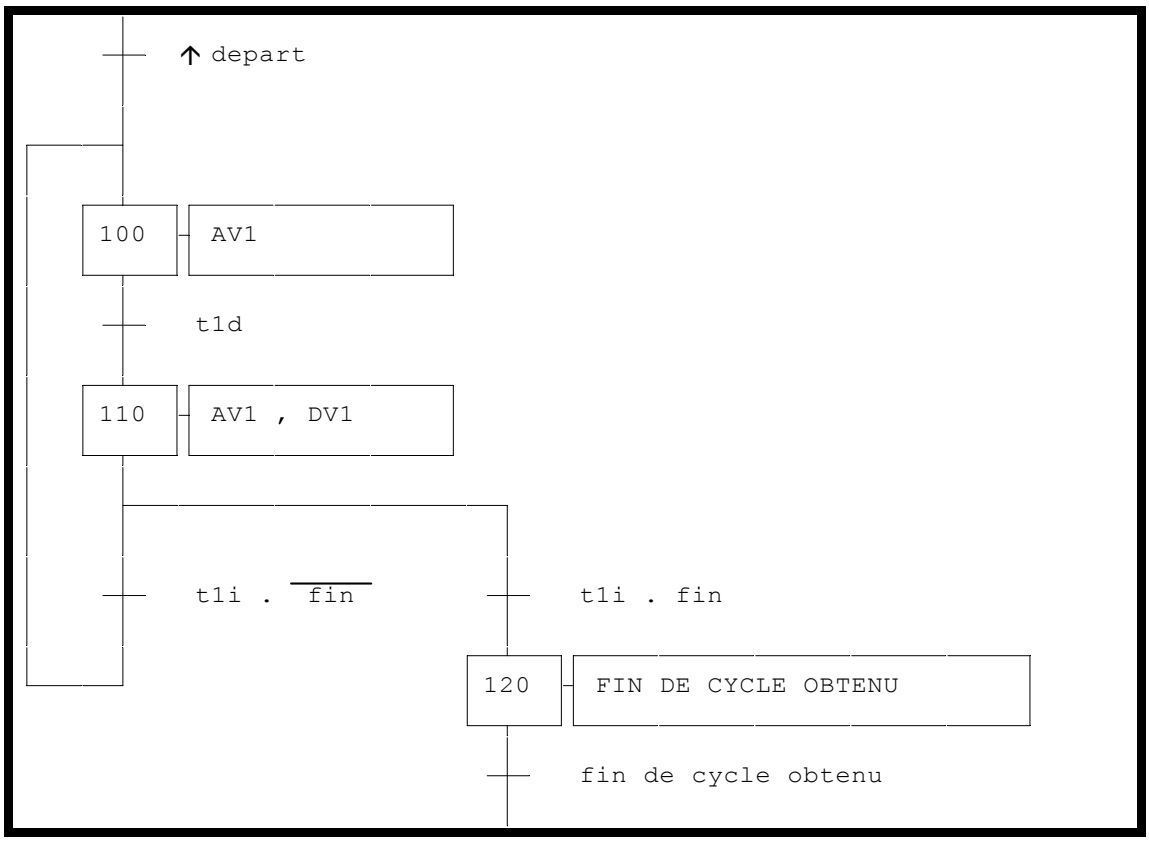
Illustrons l'utilisation du Gemma.
Cahier des charges :
Imaginons un pupitre composé des boutons poussoirs suivants : « départ cycle », « fin de cycle » et « arrêt d'urgence » et d'un voyant « INIT ».
Le programme principal consistera à faire effectuer des allers et retours à une locomotive sur la voie 1.

Solution :


P.C. HORS ENERGIE
remise en route
arrêt
mise en ou hors service fonctionnement normal
essais et vérifications
P2
A6 <Mise P.O. dans état initial> _AV1_..._DV1_ ramÞne la locomotive Ó gauche
A1 <Arrêt dans état initial> _VOYANT INIT_ ArrÛt dans Ûtat initial
F4 <Marches de vérification dans le désordre>
A7 <Mise P.O. dans état déterminé>
A4 <Arrêt obtenu>
F2 <Marches de préparation>
F3 <Marches de clôture>
PRODUCTION
F5 <Marches de vérification dans l'ordre>
A5 <Préparation pour remise en route après défaillance> rien pour l'instant
A2 <Arrêt demandé en fin de cycle> _FIN_ ArrÛt du Grafcet de production
A3 <Arrêt demandé dans état déterminé>
F1 <Production normale> _DEPART_ Lancement du Grafcet de productio
D2 <Diagnostic et/ou traitement de défaillance>
D3 <Production tout de même>
F6 <Marches de test>
PRODUCTION
PRODUCTION
D1 <Marche ou arrêt en vue d'assurer la sécurité> F_GFN_:{} RAZ du Grafcet de production
fonctionnement normal
essais et vérifications
P.C. HORS ENERGIE



(édité sous la forme d'un Grafcet)



exemple\gemma\gemma.agn

1.8. Ladder

Le langage Ladder, également appelé « schéma à contact », permet de décrire graphiquement des équations booléennes. Pour réaliser une fonction logique « Et », il faut écrire des contacts en série. Pour écrire une fonction « Ou », il faut écrire des contacts en parallèle.

Fonction « Et »

Fonction « Ou »

Le contenu des contacts doit respecter la syntaxe définie pour les tests et détaillée dans le chapitre « Eléments communs » de ce manuel.

Le contenu des bobines doit respecter la syntaxe définie pour les actions, elle aussi détaillée dans le chapitre « Eléments communs » de ce manuel.

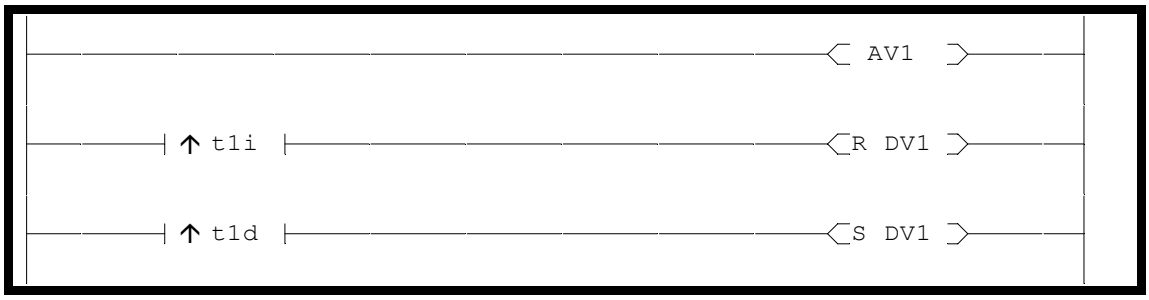
1.8.1. Exemple de Ladder

Commençons par l'exemple le plus simple.

Cahier des charges :

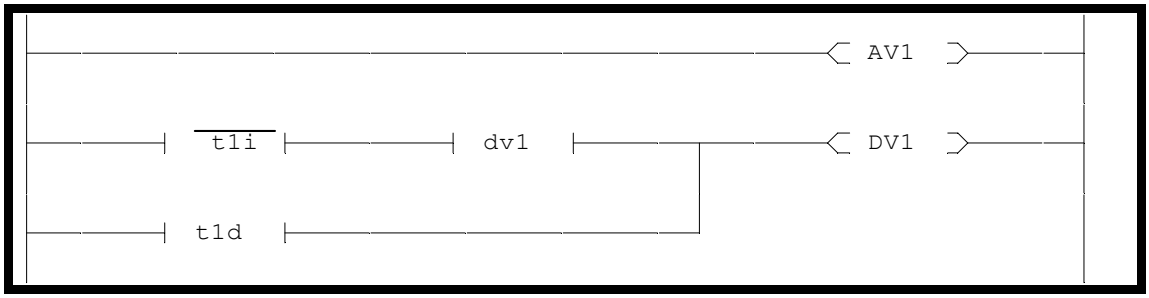
Aller et retour d'une locomotive sur la voie 1.

Solution 1 :



exemple\ladder\ladder1.agn

Solution 2 :



exemple\ladder\ladder2.agn

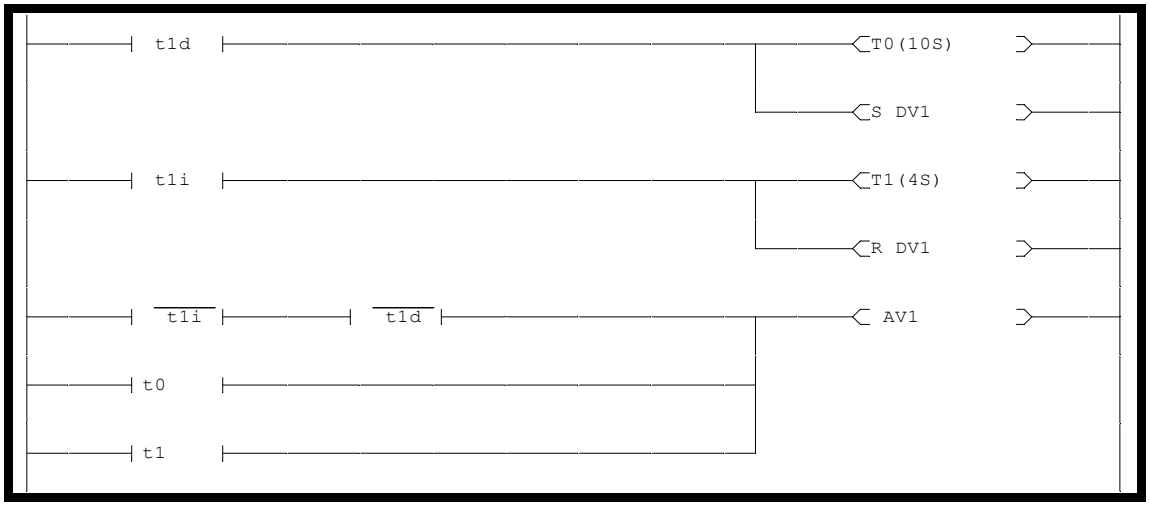
Cette deuxième solution est identique d'un point de vue fonctionnel. Son intérêt est de montrer l'utilisation d'une variable en auto maintien.

Enrichissons notre exemple.

Cahier des charges :

La locomotive devra s'arrêter 10 secondes à droite de la voie 1 et 4 secondes à gauche.

Solution :



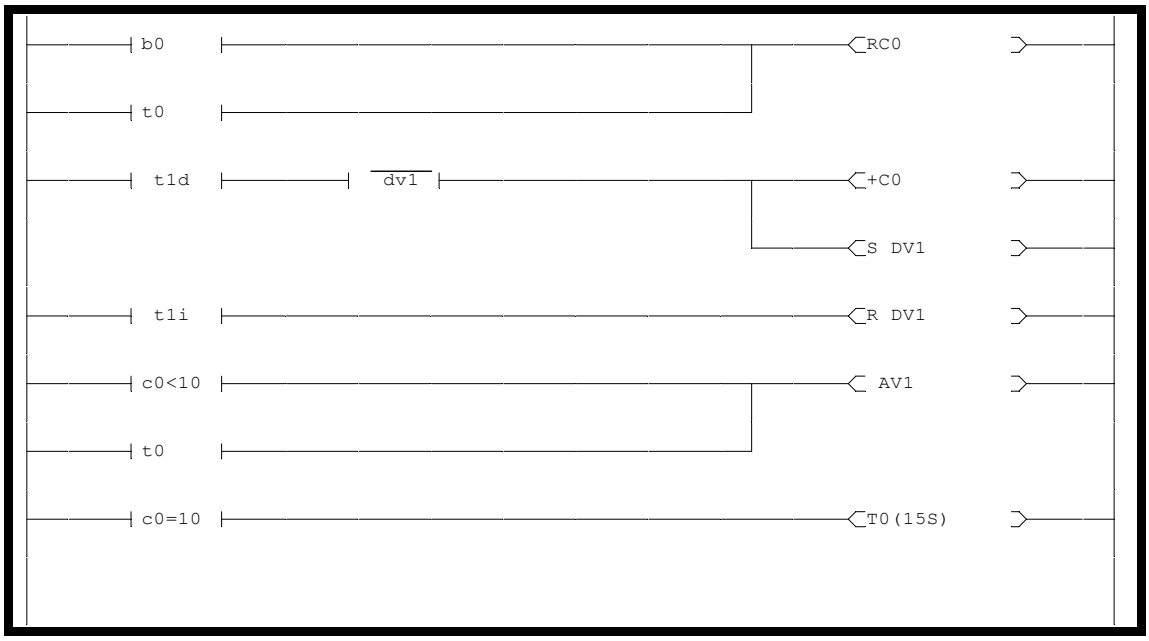
exemple\ladder\ladder3.agn

Un dernier exemple un peu plus complexe.

Cahier des charges :

Toujours une locomotive qui fait des allers et retours sur la voie 1. Tous les 10 allers et retours elle devra marquer un temps d'arrêt de 15 secondes.

Solution :



exemple\ladder\ladder4.agn

1.9. Logigramme

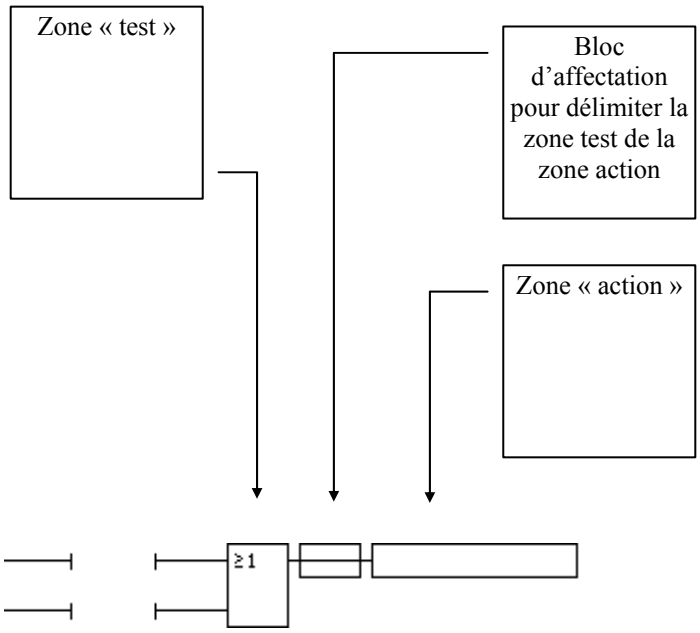
AUTOMGEN implémente le langage logigramme de la façon suivante :

⇨ utilisation d'un bloc spécial nommé « bloc d'affectation », ce bloc sépare la zone d'action de la zone test, il a la forme suivante  et est associé à la touche [0] (zéro),

⇨ utilisation des fonctions « Pas », « Et » et « Ou »,

⇨ utilisation de rectangles d'action à droite du bloc d'action.

Le langage logigramme permet d'écrire graphiquement des équations booléennes.
Le contenu des tests doit respecter la syntaxe définie dans le chapitre « Eléments communs » de ce manuel.
Le contenu des rectangles d'action doit respecter la syntaxe définie pour les actions et détaillée dans le chapitre « Eléments communs » de ce manuel.



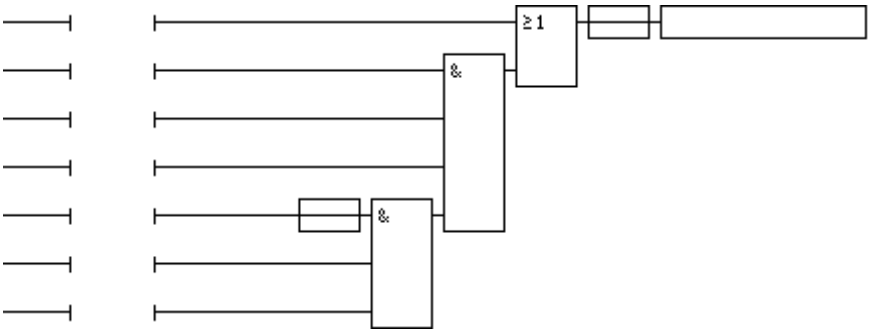
1.9.1. Dessin des logigrammes

1.9.1.1. Nombre d'entrées des fonctions « Et » et « Ou »

Les fonctions « Et » et « Ou » se composent respectivement d'un bloc  (touche [2]) ou d'un bloc  (touche [3]), éventuellement de blocs  (touche [4]) pour ajouter des entrées aux blocs et en fin d'un bloc  (touche [5]).
Les fonctions « Et » et « Ou » comportent donc un minimum de deux entrées.

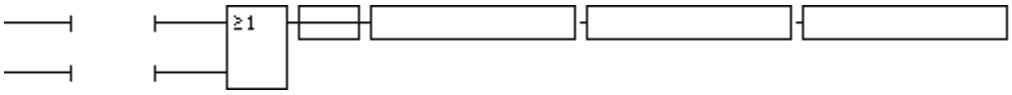
1.9.1.2. Enchaînement des fonctions

Les fonctions peuvent être chaînées.

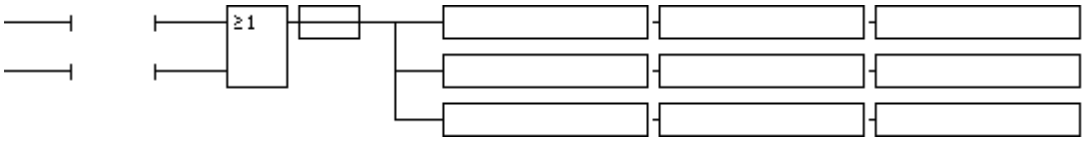


1.9.1.3. Actions multiples

Plusieurs rectangles d'action peuvent être associés à un logigramme après le bloc d'affectation.



ou



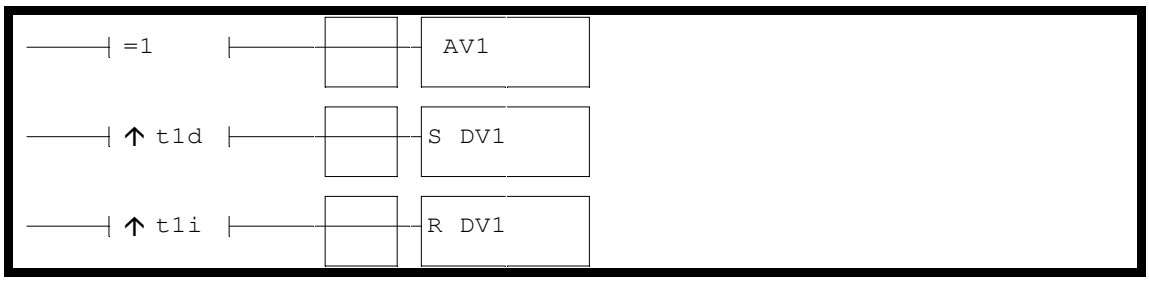
1.9.2. Exemple de logigramme

Commençons par l'exemple le plus simple.

Cahier des charges :

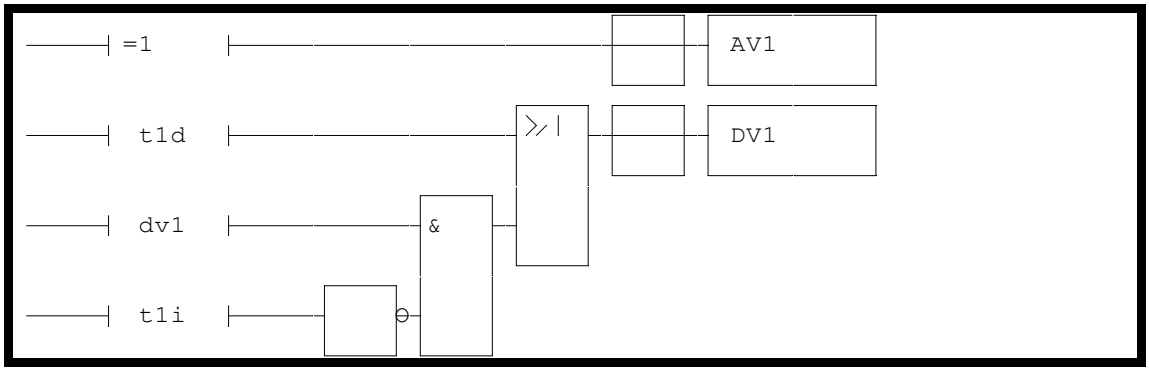
Aller et retour d'une locomotive sur la voie 1.

Solution 1 :



exemple\logigramme\logigramme1.agn

Solution 2 :



exemple\logigramme\logigramme2.agn

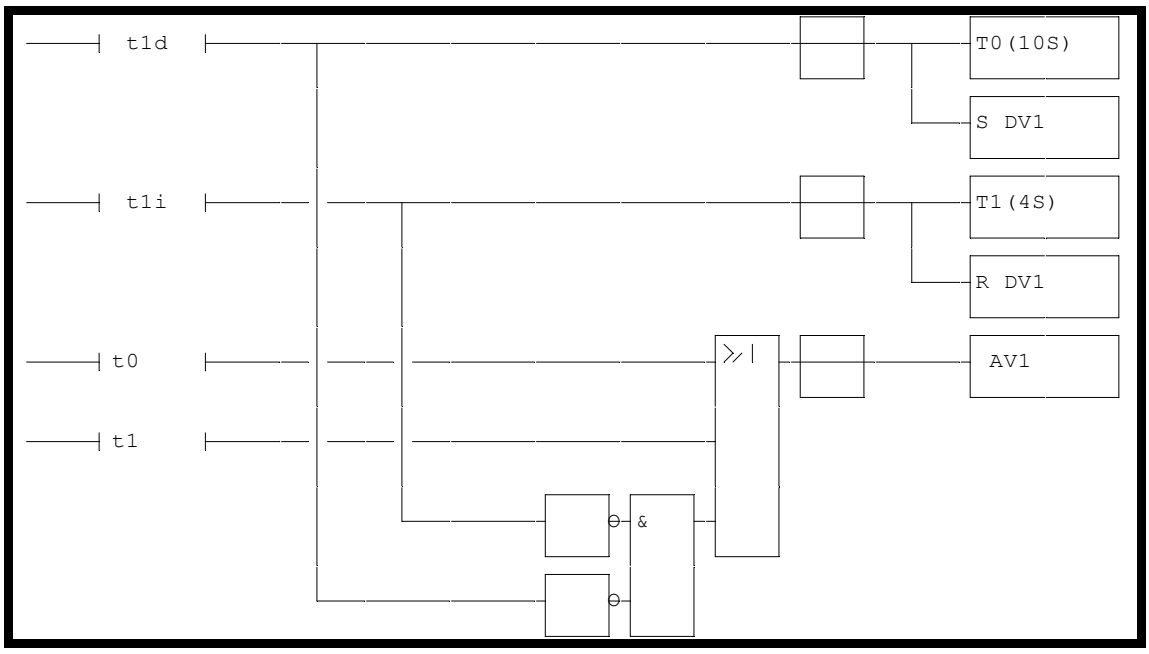
Cette deuxième solution est identique d'un point de vue fonctionnel. Son intérêt consiste à montrer l'utilisation d'une variable en auto maintien.

Enrichissons notre exemple.

Cahier des charges :

La locomotive devra s'arrêter 10 secondes à droite de la voie 1et 4 secondes à gauche.

Solution :



exemple\logigramme\logigramme3.agn

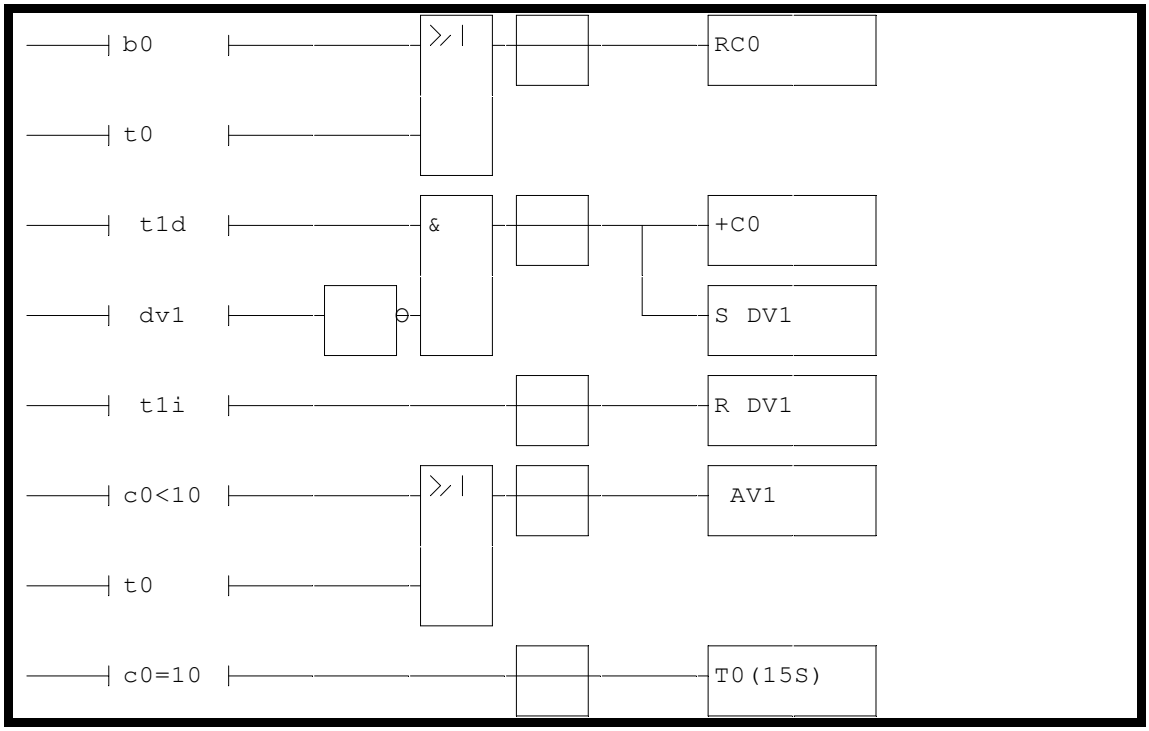
Notez le repiquage du bloc « Et » du bas de l'exemple vers les entrées « _t1d_ » et « _t1i_ ». Cela évite d'avoir à écrire une deuxième fois ces deux tests.

Un dernier exemple un peu plus complexe.

Cahier des charges :

Toujours une locomotive qui fait des allers et retours sur la voie 1. Tous les 10 allers et retours, elle devra marquer un temps d'arrêt de 15 secondes.

Solution :



exemple\logigramme\logigramme4.agn

1.10. Langages littéraux

Ce chapitre décrit l'utilisation les trois formes de langages littéraux disponibles dans AUTOMGEN :

⇨ le langage littéral bas niveau,

⇨ le langage littéral étendu,

⇨ le langage littéral ST de la norme CEI 1131-3.

1.10.1. Comment utiliser le langage littéral ?

Le langage littéral peut être utilisé sous la forme suivante :

⇨ fichier de code associé à une action (Grafcet, Ladder, logigramme),

⇨ boîte de code associée à une action (Grafcet, logigramme),

⇨ code littéral dans un rectangle d'action ou une bobine (Grafcet, Ladder, logigramme),

⇨ boîte de code utilisée sous forme d'organigramme (consultez le chapitre « Organigramme »),

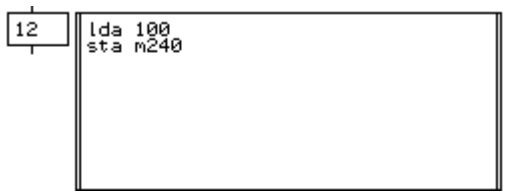
⇨ fichier de code régissant le fonctionnement d'un bloc fonctionnel (consultez le chapitre « Blocs fonctionnels »),

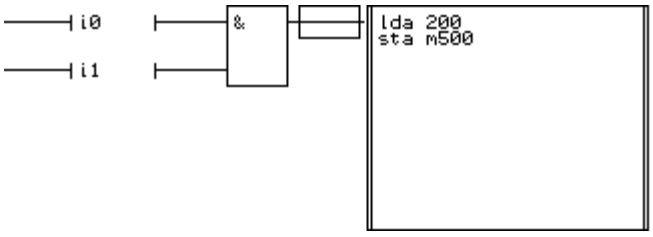
⇨ fichier de code régissant le fonctionnement d'une macro-instruction voir chapitre 1.10.4. Macro-instruction.

1.10.1.1. Boîte de code associée à une étape ou un logigramme

Une boîte de code associée à une action permet d'écrire quelques lignes de langage littéral au sein d'une page de l'application.

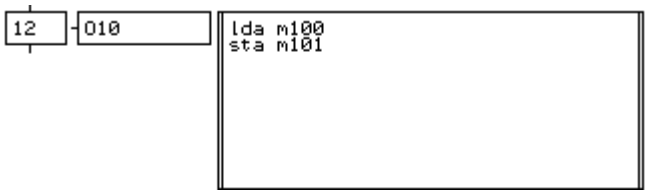
Exemples :





Le code ainsi utilisé est scruté tant que l'action est vraie.
Il est possible d'utiliser conjointement des rectangles d'action et des boîtes de code.

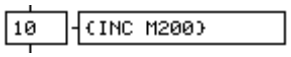
Exemple :

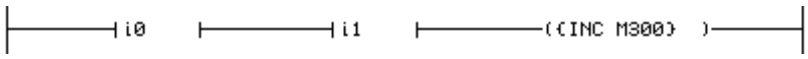


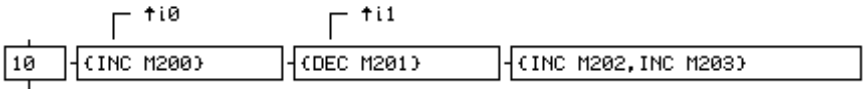
1.10.1.2. Code littéral dans un rectangle d'action ou une bobine

Les caractères « { » et « } » permettent d'insérer directement des instructions en langage littéral dans un rectangle d'action (langages Grafcet et logigramme) ou une bobine (langage ladder). Le caractère « , » (virgule) est utilisé comme séparateur si plusieurs instructions sont présentes entre « { » et « } ».
Ce type d'insertion peut être utilisé avec des ordres conditionnés.

Exemples :







1.10.2. Définition d'une boîte de code

Pour dessiner une boîte de code, suivez les étapes suivantes :

⇨ cliquez avec le bouton droit de la souris sur un emplacement vide du folio,

⇨ choisissez dans le menu « Plus … / Boîte de code »,

⇨ cliquez sur le bord de la boîte de code pour modifier son contenu.

Pour sortir de la boîte après modification, utilisez la touche [Enter] ou cliquez à l'extérieur.

1.10.3. Le langage littéral bas niveau

Ce chapitre détaille l'utilisation du langage littéral bas niveau. Ce langage est un code intermédiaire entre les langages évolués Grafcet, logigramme, ladder, organigramme, blocs fonctionnels, littéral étendu, littéral ST et les langages exécutables. Il est également connu sous le nom de code pivot. Ce sont les post-processeurs qui traduisent le langage littéral bas niveau en code exécutable pour PC, automate ou carte à microprocesseur.
Le langage littéral peut également être utilisé au sein d'une application pour effectuer divers traitements booléens, numériques ou algorithmiques.
Le langage littéral bas niveau est un langage de type assembleur. Il utilise une notion d'accumulateur pour les traitements numériques.
Le langage littéral étendu et le langage littéral ST décrits dans les chapitres suivants, offrent une alternative simplifiée et de plus haut niveau pour l'écriture de programmes en langage littéral.
Syntaxe générale d'une ligne de langage littéral bas niveau :
«action » [[[« Test »] « Test »]...]
Les actions et les tests du langage littéral bas niveau sont représentés par des mnémoniques formés de trois caractères alphabétiques. Une instruction est toujours suivie d'une expression : variable, constante, etc...
Une ligne est composée d'une seule action et éventuellement d'un test. Si une ligne comporte uniquement une action, alors par convention l'instruction sera toujours exécutée.

1.10.3.1. Les variables

Les variables utilisables sont les mêmes que celles définies dans le chapitre « Eléments communs ».

1.10.3.2. Les accumulateurs

Certaines instructions utilisent la notion d'accumulateur. Les accumulateurs sont des registres internes au système qui exécute le programme final et permettent d'accueillir temporairement des valeurs.

Il existe trois accumulateurs : un accumulateur entier 16 bits noté AAA, un accumulateur entier 32 bits noté AAL et un accumulateur flottant noté AAF.

1.10.3.3. Les drapeaux

Les drapeaux sont des variables booléennes positionnées en fonction du résultat des opérations numériques.

Il y a quatre drapeaux permettant de tester le résultat d'un calcul. Ces quatre indicateurs sont :

- ⇨ indicateur de retenue C : il indique si une opération a engendré une retenue (1) ou n'a pas engendré de retenue (0),
- ⇨ indicateur de zéro Z : il indique si une opération a engendré un résultat nul (1) ou non nul (0),
- ⇨ indicateur de signe S : il indique si une opération a engendré un résultat négatif (1) ou positif (0),
- ⇨ indicateur de débordement O : il indique si une opération a généré un dépassement de capacité (1).

1.10.3.4. Modes d'adressage

Le langage littéral bas niveau possède 5 modes d'adressage. Un mode d'adressage est une caractéristique associée à chacune des instructions du langage littéral.

Les modes d'adressage utilisables sont :

TYPE	SYNTAXE	EXEMPLE
Immédiat 16 bits	{constante}	100
Immédiat 32 bits	{constante}L	100000L
Immédiat flottant	{constante}R	3.14R
Absolu	{variable} {repère de variable}	O540
Accumulateur 16 bits	AAA	AAA
Accumulateur 32 bits	AAL	AAL

Accumulateur flottant	AAF	AAF
Indirect	{variable}{(repère de mot)}	O(220)
Label	:{nom de label} :	:boucle:

Une instruction possède donc deux caractéristiques : le type de variable et le mode d'adressage. Certaines instructions supportent ou ne supportent pas certains modes d'adressage ou certains types de variables. Par exemple une instruction pourra s'appliquer uniquement à des mots et pas aux autres types de variables.

Remarque : Les variables X et U ne peuvent être associées à un adressage indirect du fait de la non linéarité de leurs affectations. Si toutefois il est nécessaire d'accéder à un tableau de variables U, alors il faut utiliser une directive de compilation #B pour déclarer une table de bits linéaires.

1.10.3.5. Les tests

Les tests pouvant être associés aux instructions sont composés d'un mnémonique, d'un type de test et d'une variable.
Les mnémoniques de tests permettent de définir des tests combinatoires sur plusieurs variables (et, ou). Si un test est composé d'une seule variable, un opérateur AND doit quand même lui être associé.
Il existe seulement trois mnémoniques de test :
AND et
ORR ou
EOR fin de ou
Voici quelques exemples d'équivalences entre des équations booléennes et le langage littéral bas niveau :

```
o0=i1                          : and i1
o0=i1.i2                       : and i1 and i2
o0=i1+i2                       : orr i1 eor i2
o0=i1+i2+i3+i4                 : orr i1 orr i2 orr i3 eor i4
o0=(i1+i2).(i3+i4)             : orr i1 eor i2 orr i3 eor i4
o0=i1.(i2+i3+i4)               : and i1 orr i2 orr i3 eor i4
o0=(i1.i2)+(i3.i4)             ; impossible à traduire directement,
                               ; il faut utiliser des
                               ; variables intermédiaires :
```

```
equ u100 and i1 and i2
equ u101 and i3 and i4
equ o0 orr u100 eor u101
```

Les modificateurs de tests permettent de tester autre chose que l'état vrai d'une variable :

⇨ / pas

⇨ # front montant

⇨ * front descendant

⇨ @ état immédiat

Remarques :

⇨ les variables booléennes sont rafraîchies après chaque cycle d'exécution. En d'autres termes, si une variable binaire est positionnée à un état au cours d'un cycle, alors son nouvel état sera réellement reconnu au cycle suivant. Le modificateur de test @ permet d'obtenir l'état réel d'une variable booléenne sans attendre le cycle suivant.

⇨ les modificateurs de tests ne sont pas utilisables avec les tests numériques.

Exemples:

```
set o100
equ o0 and @o100          ; test vrai dès le premier cycle
equ o1 and o100           ; test vrai au deuxième cycle
```

Pour les tests, seuls deux modes d'adressage sont disponibles : le mode absolu et le mode indirect.
Un test sur compteurs, sur mots, sur longs et sur flottants est disponible :

Syntaxe :

```
« {variable} {=, !, <, >, << , >>} {constante ou variable} »
```

= signifie égal,
! signifie différent,
< signifie inférieur non signé,
> signifie supérieur non signé,

<< signifie inférieur signé,

>> signifie supérieur signé,

Les constantes sont écrites par défaut en décimal. Les suffixes « $ » et « % » permettent de les écrire en hexadécimal ou en binaire. Les guillemets permettent de les écrire en ASCII.

Les constantes 32 bits doivent être suivies du caractère « L ».

Les constantes réelles doivent être suivies du caractère « R ».

Un mot ou un compteur peut être comparé avec un mot, un compteur ou une constante 16 bits.

Un long peut être comparé avec un long ou une constante 32 bits.

Un flottant peut être comparé avec un flottant ou une constante réelle.

Exemples :

```
and c0>100 and m225=10
orr m200=m201 eor m202=m203 and f100=f101 and f200<f203
orr m200<<-100 eor m200>>200
and f200=3.14r
and l200=$12345678L
and m200=%1111111100000000
```

1.10.3.6. Commentaires

Les commentaires doivent débuter par le caractère « ; » (point virgule), tous les caractères venant à la suite sont ignorés.

1.10.3.7. Base de numérotation

Les valeurs (repères de variables ou constantes) peuvent être écrites en décimal, hexadécimal, binaire ou en ASCII.

La syntaxe suivante doit être appliquée pour les constantes 16 bits :

⇨ décimal : éventuellement le caractère « - » puis 1 à 5 digits « 0123456789 »,

⇨ hexadécimal : le préfixe « $ » ou « 16# » suivi de 1 à 4 digits « 0123456789ABCDEF »,

⇨ binaire : le préfixe « % » ou « 2# » suivi de 1 à 16 digits « 01 »,

⇨ ASCII : le caractère « " » suivi de 1 ou 2 caractères suivis de « " ».

La syntaxe suivante doit être appliquée pour les constantes 32 bits :

⇨ Décimal : éventuellement le caractère « - » puis 1 à 10 digits « 0123456789 »,

⇨ Hexadécimal : le préfixe « $ » ou « 16# » suivi de 1 à 8 digits « 0123456789ABCDEF »,

⇨ Binaire : le préfixe « % » ou « 2# » suivi de 1 à 32 digits « 01 »,

⇨ ASCII : le caractère « " » suivi de 1 à 4 caractères suivis de « " ».

La syntaxe suivante doit être appliquée pour les constantes réelles :
[-] i [[.d] Esx]
i est la partie entière
d une éventuelle partie décimale
s éventuellement le signe de l'exposant
x éventuellement l'exposant

1.10.3.8. Prédispositions

Une prédisposition permet de fixer la valeur d'une variable au démarrage de l'application.
Les variables T ou %T, M ou %MW, L ou %MD et F ou %F peuvent être prédisposées.

La syntaxe est la suivante :

```
« $(variable)=constante{,constante{,constante...}} »
```

Pour les temporisations la valeur doit être écrite en décimal et comprise entre 0 et 65535.
Pour les mots la syntaxe suivante doit être utilisée :

⇨ Décimal : éventuellement le caractère « - » puis 1 à 5 digits « 0123456789 »,

⇨ Hexadécimal : le préfixe « $ » ou « 16# » suivi de 1 à 4 digits « 0123456789ABCDEF »,

⇨ Binaire : le préfixe « % » ou « 2# » suivi de 1 à 16 digits « 01 »,

- ⇨ ASCII : (deux caractères par mot) le caractère « " » suivi de n caractères suivis de « " »,
- ⇨ ASCII : (un caractère par mot) le caractère « ’ » suivi de n caractères suivis de « ’ ».

Pour les longs la syntaxe suivante doit être utilisée :

- ⇨ Décimal : éventuellement le caractère « - » puis 1 à 10 digits « 0123456789 »,
- ⇨ Hexadécimal : le préfixe « $ » ou « 16# » suivi de 1 à 8 digits « 0123456789ABCDEF »,
- ⇨ Binaire : le caractère « % » ou « 2# » suivi de 1 à 32 digits « 01 »,
- ⇨ ASCII : (quatre caractères par long) le caractère « " » suivi de n caractères suivis de « " »,
- ⇨ ASCII : (un caractère par long) le caractère « ’ » suivi de n caractères suivis de « ’ »

Pour les flottants, la valeur doit être écrite sous la forme :
[-] i [[.d] Esx]
i est la partie entière
d une éventuelle partie décimale
s éventuellement le signe de l’exposant
x éventuellement l’exposant

Exemples :

```
$t25=100
```

fixe la consigne de la temporisation 25 à 10 s

```
$MW200=100,200,300,400
```

place les valeurs 100,200,300,400 dans les mots 200, 201, 202, 203

```
$m200="ABCDEF"
```

place la chaîne de caractères « ABCDEF » à partir de m200 (« AB » dans m200, « CD » dans m201, « EF » dans m202)

```
$m200='ABCDEF'
```

place la chaîne de caractères « ABCDEF » à partir de m200, chaque mot reçoit un caractère.

```
$f1000=3.14
```

place la valeur 3,14 dans f1000

```
$%mf100=5.1E-15
```

place la valeur 5,1 * 10 exposant -15 dans %mf100

```
$l200=16#12345678
```

place la valeur 12345678 (hexa) dans le long l200

Il est possible d'écrire plus facilement du texte dans les prédispositions.

Exemple :

```
$m200=" Arrêt de la vanne N°10 "
```

Place le message à partir du mot 200 en plaçant deux caractères dans chaque mot.

```
$m400=‘ Défaut moteur ‘
```

Place le message à partir du mot 400 en plaçant un caractère dans l'octet de poids faibles de chaque mot, l'octet de poids fort contient 0.

La syntaxe « $...= » permet de poursuivre une table de prédispositions à la suite de la précédente.

Par exemple :

```
#$m200=1,2,3,4,5
#$...=6,7,8,9
```

Place les valeurs de 1 à 9 dans les mots m200 à m208.

Les prédispositions peuvent être écrites de la même façon que le langage littéral bas niveau ou dans une directive de compilation dans un folio. Dans ce cas, la prédisposition commence par le caractère « # ».

Exemple de prédisposition écrite dans une boîte de code :

Exemple de prédisposition écrite dans une directive de compilation :

```
#$m200=12,13
```

1.10.3.9. Adressage indirect

L'adressage indirect permet d'effectuer une opération sur une variable pointée par un index.

Ce sont les variables M (les mots) qui servent d'index.

Syntaxe :

« variable (index) »

Exemple :

```
lda 10       ; charge 10 dans l'accumulateur
sta m200     ; le place dans le mot 200
set o(200)   ; mise à un de la sortie pointée par le mot 200 (o10)
```

1.10.3.10. Adresse d'une variable

Le caractère « ? » permet de spécifier l'adresse d'une variable.

Exemple :

```
lda ?o10            ; place la valeur 10 dans l'accumulateur
```

Cette syntaxe est surtout intéressante si des symboles sont utilisés.

Exemple :

```
lda ?_vanne_        ; place dans l'accumulateur le numéro de la variable
                    ; associée au symbole « _vanne_ »
```

Cette syntaxe peut également être utilisée dans les prédispositions pour créer des tables d'adresses de variables.

Exemple :

```
$m200=?_vanne1_,?_vanne2_,?_vanne3_
```

1.10.3.11. Sauts et labels

Les sauts doivent faire référence à un label. La syntaxe d'un label est :

« :nom du label: »

Exemple :

```
jmp :suite:
...
:suite:
```

1.10.3.12. Liste des fonctions par type

1.10.3.12.1. Fonctions booléennes

SET	mise à un
RES	mise à zéro
INV	inversion
EQU	équivalence
NEQ	non-équivalence

1.10.3.12.2. Fonctions de chargement et de stockage sur entiers et flottants

LDA	chargement
STA	stockage

1.10.3.12.3. Fonctions arithmétiques sur entiers et flottants

ADA	addition
SBA	soustraction
MLA	multiplication
DVA	division
CPA	comparaison

1.10.3.12.4. Fonctions arithmétiques sur flottants

ABS	valeur absolue
SQR	racine carrée

1.10.3.12.5. Fonctions d'accès aux ports d'entrées/sorties sur PC

AIN	lecture d'un port
AOU	écriture d'un port

1.10.3.12.6. Fonctions d'accès à la mémoire sur PC

ATM	lecture d'une adresse mémoire
MTA	écriture d'une adresse mémoire

1.10.3.12.7. Fonctions binaires sur entiers

ANA	et bit à bit
ORA	ou bit à bit
XRA	ou exclusif bit à bit
TSA	test bit à bit

SET	mise à un de tous les bits
RES	mise à zéro de tous les bits
RRA	décalage à droite
RLA	décalage à gauche

1.10.3.12.8. Autres fonctions sur entiers

INC	incrémentation
DEC	décrémentation

1.10.3.12.9. Fonctions de conversion

ATB	entier vers booléens
BTA	booléens vers entier
FTI	flottant vers entier
ITF	entier vers flottant
LTI	entier 32 bits vers entier 16 bits
ITL	entier 16 bits vers entier 32 bits

1.10.3.12.10. Fonctions de branchement

JMP saut

JSR saut à un sous-programme

RET retour de sous-programme

1.10.3.12.11. Fonctions de test

RFZ flag de résultat nul

RFS flag de signe

RFO flag de débordement

RFC flag de retenue

1.10.3.12.12. Fonctions d'accès asynchrones aux entrées sorties

RIN lecture des entrées

WOU écriture des sorties

1.10.3.12.13. Informations contenues dans la liste des fonctions

Pour chaque instruction sont donnés :

⇨ Nom : le mnémonique.

⇨ Fonction : une description de la fonction réalisée par l'instruction.

⇨ Variables : les types de variables utilisables avec l'instruction.

⇨ Adressage : les types d'adressages utilisables.

⇨ Voir aussi : les autres instructions ayant un rapport avec ce mnémonique.

⇨ Exemple : un exemple d'utilisation.

Les post-processeurs qui génèrent des langages constructeurs sont soumis à certaines restrictions. Consultez les notices relatives à ces post-processeurs pour avoir le détail de ces restrictions.

ABS

Nom	:	ABS - abs accumulator
Fonction	:	calcule la valeur absolue de l'accumulateur flottant
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	SQR
Exemple	:	

```
lda f200
abs aaf
sta f201
; laisse dans f201 la valeur absolue de f200
```


ADA

Nom	:	ADA - adds accumulator
Fonction	:	ajoute une valeur à l'accumulateur
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	SBA
Exemple	:	

```
ada 200
; ajoute 200 à l'accumulateur 16 bits

ada f124
; ajoute le contenu de f124 à l'accumulateur flottant

ada l200
; ajoute le contenu de l200 à l'accumulateur 32 bits

ada 200L
; ajoute 200 à l'accumulateur 32 bits

ada 3.14R
; ajoute 3.14 à l'accumulateur flottant
```

AIN

Nom	:	AIN - accumulator input
Fonction	:	lecture d'un port d'entrée (8 bits) et stockage dans la partie basse de l'accumulateur 16 bits ;
		 lecture d'un port d'entrée 16 bits et stockage dans l'accumulateur 16 bits (dans ce cas l'adresse du port doit être écrite sous la forme d'une constante 32 bits)
		utilisable seulement avec l'exécuteur PC
Variables	:	M ou %MW
Adressage	:	indirect, immédiat
Voir aussi	:	AOU
Exemple	:	

```
ain $3f8
; lecture du port $3f8 (8 bits)

ain $3f8l
; lecture du port $3f8 (16 bits)
```

ANA

Nom	:	ANA - and accumulator
Fonction	:	effectue un ET logique entre l'accumulateur 16 bits et un mot ou une constante ou l'accumulateur 32 bits et un long ou une constante
Variables	:	M ou %MW, L ou %MD
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	ORA, XRA
Exemple	:	

```
ana %1111111100000000
; masque les 8 bits de poids faible de
; l'accumulateur 16 bits

ana $ffff0000L
; masque les 16 bits de poids faibles de l'accumulateur 32 bits
```

AOU

Nom	:	AOU - accumulator output
Fonction	:	transfère la partie basse (8 bits) du contenu de l'accumulateur 16 bits sur un port de sortie ;  transfère les 16 bits de l'accumulateur 16 bits sur un port de sortie (sans ce cas l'adresse du port doit être écrite sous la forme d'une constante 32 bits) utilisable seulement avec l'exécuteur PC
Variables	:	M ou %MW
Adressage	:	indirect, immédiat
Voir aussi	:	AIN
Exemple	:	

```
lda "A"
aou $3f8
; place le caractère « A » sur le port de sortie $3f8

lda $3f8
sta m200
lda "z"
aou m(200)
; place le caractère « z » sur le port de sortie $3f8

lda $1234
aou $300l
; place la valeur 16 bits 1234 sur le port de sortie $300
```

ATB

Nom	:	ATB - accumulator to bit
Fonction	:	transfère les 16 bits de l'accumulateur 16 bits vers 16 variables booléennes successives ; le bit de poids faible correspond à la première variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, U*
Adressage	:	absolu
Voir aussi	:	BTA
Exemple	:	

```
lda m200
atb o0
; recopie les 16 bits de m200 dans les variables
; o0 à o15
```

* Attention : pour pouvoir utiliser les bits U avec cette fonction il faut réaliser une table linéaire de bits avec la directive de compilation #B.

ATM

Nom	:	ATM - accumulator to memory
Fonction	:	transfère l'accumulateur 16 bits à une adresse mémoire ; le mot ou la constante spécifié défini l'offset de l'adresse mémoire à atteindre, le mot m0 doit être chargé avec la valeur du segment de l'adresse mémoire à atteindre ; utilisable seulement avec l'exécuteur PC
Variables	:	M ou %MW
Adressage	:	indirect, immédiat
Voir aussi	:	MTA
Exemple	:	

```
lda $b800
sta m0
lda 64258
atm $10
; place la valeur 64258 à l'adresse $b800:$0010
```


BTA

Nom	:	BTA - bit to accumulator
Fonction	:	transfère 16 variables booléennes successives vers les 16 bits de l'accumulateur 16 bits ; le bit de poids faible correspond à la première variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, U*
Adressage	:	absolu
Voir aussi	:	ATB
Exemple	:	

```
bta i0
sta m200
; recopie les 16 entrées i0 à i15 dans le mot m200
```

* Attention : pour pouvoir utiliser les bits U avec cette fonction il faut réaliser une table linéaire de bits avec la directive de compilation #B.

CPA

Nom	:	CPA - compares accumulator
Fonction	:	compare une valeur à l'accumulateur 16 bits ou 32 bits ou flottant ; effectue la même opération que SBA mais sans modifier le contenu de l'accumulateur
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	SBA
Exemple	:	

```
lda m200
cpa 4
rfz o0
; met o0 à 1 si m200 est égal à 4, autrement o0
; est mis à 0

lda f200
cpa f201
rfz o1
; met o1 à 1 si f200 est égal à f201, autrement o1
; est mis à 0
```

DEC

Nom	:	DEC – decrement
Fonction	:	décrémente un mot, un compteur, un long, l'accumulateur 16 bits ou 32 bits
Variables	:	M ou %MW, C ou %C, L ou %MD
Adressage	:	absolu, indirect, accumulateur
Voir aussi	:	INC
Exemple	:	

```
dec m200
; décrémente m200

dec aal
; décrémente l'accumulateur 32 bits

dec m200
dec m201 and m200=-1
; décrémente une valeur 32 bits composée de
; m200 (poids faibles)
; et m201 (poids forts)
```

DVA

Nom	:	DVA - divides accumulator
Fonction	:	division de l'accumulateur 16 bits par un mot ou une constante ; division de l'accumulateur flottant par un flottant ou une constante ; division de l'accumulateur 32 bits par un long ou une constante, pour l'accumulateur 16 bits le reste est placé dans le mot m0 ; en cas de division par 0 le bit système 56 passe à 1
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	MLA
Exemple	:	

```
lda m200
dva 10
sta m201
; m201 est égal à m200 divisé par 10, m0 contient le
; reste de la division

lda l200
dva $10000L
sta l201
```

EQU

Nom	:	EQU - equal
Fonction	:	force une variable à 1 si le test est vrai, dans le cas contraire la variable est forcée à 0
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu, indirect (sauf sur les variables X)
Voir aussi	:	NEQ, SET, RES, INV
Exemple	:	

```
equ o0 and i10
; force la sortie o0 au même état que l'entrée i10

lda 10
sta m200
equ o(200) and i0
; force o10 au même état que l'entrée i0

$t0=100
equ t0 and i0
equ o0 and t0
; force o0 à l'état de i0 avec un retard à l'activation
; de 10 secondes
```


FTI

Nom	:	FTI - float to integer
Fonction	:	transfère l'accumulateur flottant vers l'accumulateur 16 bits
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	ITF
Exemple	:	

```
lda f200
fti aaa
sta m1000
; laisse la partie entière de f200 dans m1000
```

INC

Nom	:	INC - increment
Fonction	:	incrémente un mot, un compteur, un long, l'accumulateur 16 bits ou 32 bits
Variables	:	M ou %MW, C ou %C, L ou %MD
Adressage	:	absolu, indirect, accumulateur
Voir aussi	:	DEC
Exemple	:	

```
inc m200
; ajoute 1 à m200

inc m200
inc m201 and m201=0
; incrémente une valeur sur 32 bits, m200
; représente les
; poids faibles, et m201 les poids forts

inc l200
; incrémente le long l200
```


INV

Nom	:	INV - inverse
Fonction	:	inverse l'état d'une variable booléenne, ou inverse tous les bits d'un mot, d'un long ou de l'accumulateur 16 bits ou 32 bits
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U, M ou %MW, L ou %MD
Adressage	:	absolu, indirect, accumulateur
Voir aussi	:	EQU, NEQ, SET, RES
Exemple	:	

```
inv o0
; inverse l'état de la sortie 0

inv aaa
; inverse tous les bits de l'accumulateur 16 bits

inv m200 and i0
; inverse tous les bits de m200 si i0 est à l'état 1
```

ITF

Nom	:	ITF - integer to float
Fonction	:	transfère l'accumulateur 16 bits vers l'accumulateur flottant
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	FTI
Exemple	:	

```
lda 1000
itf aaa
sta f200
; laisse la constante 1000 dans f200
```


ITL

Nom	:	ITL - integer to long
Fonction	:	transfère l'accumulateur 16 bits vers l'accumulateur 32 bits
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	LTI
Exemple	:	

```
lda 1000
itl aaa
sta f200
; laisse la constante 1000 dans l200
```

JMP

Nom	:	JMP - jump
Fonction	:	saut à un label
Variables	:	label
Adressage	:	label
Voir aussi	:	JSR
Exemple	:	

```
jmp :fin de programme:
; branchement inconditionnel au label :fin
; de programme:

jmp :suite: and i0
set o0
set o1
:suite:
; branchement conditionnel au label :suite:
; suivant l'état de i0
```

JSR

Nom	:	JSR - jump sub routine
Fonction	:	effectue un branchement à un sous-programme
Variables	:	label
Adressage	:	label
Voir aussi	:	RET
Exemple	:	

```
        lda m200
        jsr :carre:
        sta m201
        jmp :fin:

:carre:
        sta m53
        mla m53
        sta m53
        ret m53

:fin:

; le sous-programme « carre » élève le contenu de
; l'accumulateur au carré
```


LDA

Nom	:	LDA - load accumulator
Fonction	:	charge dans l'accumulateur 16 bits une constante, un mot ou un compteur ; charge dans l'accumulateur 32 bits un long ou une constante, charge dans l'accumulateur flottant un flottant ou une constante, charge un compteur de temporisation dans l'accumulateur 16 bits
Variables	:	M ou %MW, C ou %C, L ou %MD, F ou %MF, T ou %T
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	STA
Exemple	:	

```
lda 200
; charge la constante 200 dans l'accumulateur 16 bits

lda 0.01R
; charge la constante réelle 0.01 dans l'accumulateur flottant

lda t10
; charge le compteur de la temporisation 10 dans
; l'accumulateur
```


LTI

Nom	:	LTI - long to integer
Fonction	:	transfère l'accumulateur 32 bits vers l'accumulateur 16 bits
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	ITL
Exemple	:	

```
lda l200
lti aaa
sta m1000
; laisse les 16 bits de poids faibles de l200 dans m1000
```


MLA

Nom	:	MLA - multiples accumulator
Fonction	:	multiplication de l'accumulateur 16 bits par un mot ou une constante ; multiplication de l'accumulateur 32 bits par un long ou une constante, multiplication de l'accumulateur flottant par un flottant ou une constante ; pour l'accumulateur 16 bits, les 16 bits de poids forts du résultat de la multiplication sont transférés dans m0
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	DVA
Exemple	:	

```
lda m200
mla 10
sta m201
; multiplie m200 par 10, m201 est chargé avec les
; 16 bits de poids faibles, et m0 avec les 16 bits de
; poids forts
```

MTA

Nom	:	MTA - memory to accumulator
Fonction	:	transfère le contenu d'une adresse mémoire dans l'accumulateur 16 bits ; le mot ou la constante spécifié défini l'offset de l'adresse mémoire à atteindre, le mot m0 doit être chargé avec la valeur du segment de l'adresse mémoire à atteindre ; utilisable seulement avec l'exécuteur PC
Variables	:	M ou %MW
Adressage	:	indirect, immédiat
Voir aussi	:	ATM
Exemple	:	

```
lda $b800
sta m0
mta $10
; place la valeur contenue à l'adresse $b800:$0010
; dans l'accumulateur 16 bits
```


NEQ

Nom	:	NEQ - not equal
Fonction	:	force une variable à 0 si le test est vrai, dans le cas contraire la variable est forcée à 1
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu, indirect (sauf sur les variables X)
Voir aussi	:	EQU, SET, RES, INV
Exemple	:	

```
neq o0 and i00
; force la sortie o0 à l'état complémenté de l'entrée
; i10

lda 10
sta m200
neq o(200) and i0
; force o10 à l'état complémenté de l'entrée i0

$t0=100
neq t0 and i0
neq o0 and t0
; force o0 à l'état de i0 avec un retard à la
; désactivation de 10 secondes
```

ORA

Nom	:	ORA - or accumulator
Fonction	:	effectue un OU logique entre l'accumulateur 16 bits et un mot ou une constante, ou entre l'accumulateur 32 bits et un long ou une constante
Variables	:	M ou %M, L ou %MD
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	ANA, XRA
Exemple	:	

```
ora %1111111100000000
; force les 8 bits de poids forts de
; l'accumulateur 16 bits à 1

ora $ffff0000L
; force les 16 bits de poids forts de l'accumulateur 32 bits
; à 1
```

RES

Nom	:	RES - reset
Fonction	:	force une variable booléenne, un mot, un compteur, un long, l'accumulateur 16 bits ou l'accumulateur 32 bits à 0
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U, M ou %MW, C ou %C, L ou %MD
Adressage	:	absolu, indirect (sauf sur les variables X), accumulateur
Voir aussi	:	NEQ, SET, EQU, INV
Exemple	:	

```
res o0
; force la sortie o0 à 0

lda 10
sta m200
res o(200) and i0
; force o10 à 0 si l'entrée i0 est à 1

res c0
; force le compteur 0 à 0
```


RET

Nom	:	RET - return
Fonction	:	marque le retour d'un sous-programme et place dans l'accumulateur 16 bits un mot ou une constante ; ou place dans l'accumulateur 32 bits un long ou une constante, ou place dans l'accumulateur flottant un flottant ou une constante
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	JSR
Exemple	:	

```
ret 0
; retour de sous-programme en plaçant 0 dans
; l'accumulateur 16 bits

ret f200
; retour de sous-programme en plaçant le contenu
; de f200 dans l'accumulateur flottant
```

RFC

Nom	:	RFC - read flag : carry
Fonction	:	transfère le contenu de l'indicateur de retenue dans une variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu
Voir aussi	:	RFZ, RFS, RFO
Exemple	:	

```
rfc o0
; transfère l'indicateur de retenue dans o0

lda m200
ada m300
sta m400
rfc b99
lda m201
ada m301
sta m401
inc m401 and b99
; effectue une addition sur 32 bits
; (m400,401)=(m200,201)+(m300,301)
; m200, m300 et m400 sont les poids faibles
; m201, m301 et m401 sont les poids forts
```

RFO

Nom	:	RFO - read flag : overflow
Fonction	:	transfère le contenu de l'indicateur de débordement dans une variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu
Voir aussi	:	RFZ, RFS, RFC
Exemple	:	

```
rfo o0
; transfère l'indicateur de débordement dans o0
```

RFS

Nom	:	RFS - read flag : signe
Fonction	:	transfère le contenu de l'indicateur de signe dans une variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu
Voir aussi	:	RFZ, RFC, RFO
Exemple	:	

```
rfs o0
; transfère l'indicateur de signe dans o0
```


RFZ

Nom	:	RFZ - read flag : zero
Fonction	:	transfère le contenu de l'indicateur de résultat nul dans une variable booléenne
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U
Adressage	:	absolu
Voir aussi	:	RFC, RFS, RFO
Exemple	:	

```
rfz o0
; transfère l'indicateur de résultat nul dans o0

lda m200
cpa m201
rfz o0
; positionne o0 à 1 si m200 est égal à m201
; ou à 0 dans le cas contraire
```


RIN

Nom	:	RIN - read input
Fonction	:	effectue une lecture des entrées physiques. Cette fonction est uniquement implémentée sur cibles Z et varie suivant la cible. Consultez la documentation relative à chaque éxécuteur pour plus de détails.
Variables	:	aucune
Adressage	:	immédiat
Voir aussi	:	WOU

RLA

Nom	:	RLA - rotate left accumulator
Fonction	:	effectue une rotation à gauche des bits de l'accumulateur 16 bits ou 32 bits ; le bit évacué à gauche entre à droite ; l'argument de cette fonction est une constante qui précise le nombre de décalages à effectuer, la taille de l'argument (16 ou 32 bits) détermine quel est l'accumulateur qui doit subir la rotation
Variables	:	aucune
Adressage	:	immédiat
Voir aussi	:	RRA
Exemple	:	

```
ana $f000
; isole le digit de poids fort de l'accumulateur 16 bits
rla 4
; et le ramène à droite

rla 8L
; effectue 8 rotations à gauche des bits de l'accumlateur
; 32 bits
```


RRA

Nom	:	RRA - rotate right accumulator
Fonction	:	effectue une rotation à droite des bits de l'accumulateur 16 bits ou 32 bits ; le bit évacué à droite entre à gauche ; l'argument de cette fonction est une constante qui précise le nombre de décalages à effectuer, la taille de l'argument (16 ou 32 bits) détermine si c'est l'accumulateur 16 ou 32 bits qui doit subir la rotation
Variables	:	aucune
Adressage	:	immédiat
Voir aussi	:	RLA
Exemple	:	

```
ana $f000
; isole le digit de poids fort de l'accumulateur 16 bits
rra 12
; et le ramène à droite

rra 1L
; effectue une rotation des bits de l'accumulateur 32 bits
; d'une position vers la droite
```


SBA

Nom	:	SBA - substacts accumulator
Fonction	:	enlève le contenu d'un mot ou une constante à l'accumulateur 16 bits ; enlève le contenu d'un long ou d'une constante à l'accumulateur 32 bits, enlève le contenu d'un flottant ou d'une constante à l'accumulateur flottant
Variables	:	M ou %MW, L ou %MD, F ou %MF
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	ADA
Exemple	:	

```
sba 200
; enlève 200 à l'accumulateur 16 bits

sba f(421)
; enlève le contenu du flottant dont le numéro est
; contenu dans le mot 421 à l'accumulateur flottant
```

SET

Nom	:	SET - set
Fonction	:	force une variable booléenne à 1; force tous les bits d'un mot, d'un compteur, d'un long, de l'accumulateur 16 bits ou de l'accumulateur 32 bits à 1
Variables	:	I ou %I, O ou %Q, B ou %M, T ou %T, X ou %X, U, M ou %MW, C ou %C, L ou %MD
Adressage	:	absolu, indirect (sauf sur les variables X), accumulateur
Voir aussi	:	NEQ, RES, EQU, INV
Exemple	:	

```
set o0
; force la sortie o0 à 1

lda 10
sta m200
set o(200) and i0
; force o10 à 1 si l'entrée i0 est à 1

set m200
; force m200 à la valeur -1

set aal
; force tous les bits de l'accumulateur 32 bits à 1
```

SQR

Nom	:	SQR - square root
Fonction	:	calcule la racine carrée de l'accumulateur flottant
Variables	:	aucune
Adressage	:	accumulateur
Voir aussi	:	ABS
Exemple	:	

```
lda 9
itf aaa
sqr aaf
fti aaa
sta m200
; laisse la valeur 3 dans m200
```

STA

Nom	:	STA - store accumulator
Fonction	:	stocke l'accumulateur 16 bits dans un compteur ou un mot; stocke l'accumulateur 32 bits dans un long, stocke l'accumulateur flottant dans un flottant, stocke l'accumulateur 16 bits dans une consigne de temporisation
Variables	:	M ou %MW, C ou %C, L ou %MD, F ou %MF, T ou %T
Adressage	:	absolu, indirect
Voir aussi	:	LDA
Exemple	:	

```
sta m200
; transfère le contenu de l'accumulateur 16 bits
; dans le mot 200

sta f200
; transfère le contenu de l'accumulateur flottant
; dans le flottant 200

sta l200
; transfère l'accumulateur 32 bits dans le long l200
```


TSA

Nom	:	TSA - test accumulator
Fonction	:	effectue un ET logique entre l'accumulateur 16 bits et un mot ou une constante ; effectue un ET logique entre l'accumulateur 32 bits et un long ou une constante, opère de façon similaire à l'instruction ANA, mais sans modifier le contenu de l'accumulateur
Variables	:	M ou %MW, L ou %MD
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	ANA
Exemple	:	

```
tsa %10
rfz b99
jmp :suite: and b99
; branchement au label :suite: si le bit 1
; de l'accumulateur 16 bits est à 0
```

WOU

Nom	:	WOU - write output
Fonction	:	effectue une écriture des sorties physiques. Cette fonction est uniquement implémentée sur cibles Z (et varie suivant la cible). Consultez la documentation relative à chaque exécuteur pour plus de détails.
Variables	:	aucune
Adressage	:	immédiat
Voir aussi	:	RIN

XRA

Nom	:	XRA - xor accumulator
Fonction	:	effectue un OU EXCLUSIF entre l'accumulateur 16 bits et un mot ou une constante, effectue un OU EXCLUSIF entre l'accumulateur 32 bits et un long ou une constante
Variables	:	M ou %MW, L ou %MD
Adressage	:	absolu, indirect, immédiat
Voir aussi	:	ORA, ANA,
Exemple	:	

```
xra %1111111100000000
; inverse les 8 bits de poids fort de l'accumulateur 16 bits

xra 1L
; inverse le bit de poids faible de l'accumulateur 32 bits
```

1.10.4. Macro-instruction

Les macro-instructions sont des nouvelles instructions du langage littéral derrière lesquelles se cache un ensemble d'instructions de base.
Syntaxe d'appel d'une macro-instruction :
« %<nom de la macro-instruction*> {paramètres ...} »
Syntaxe de déclaration d'une macro-instruction :

```
#MACRO
<programme>
#ENDM
```

Cette déclaration se trouve dans un fichier portant le nom de la macro-instruction et possédant l'extension « .M ».

Le fichier .M peut être placé dans sous-répertoire « lib » du répertoire d'installation d'AUTOMGEN ou dans les ressources du projet.

Dix paramètres peuvent être passés à la macro-instruction. A l'appel, ces paramètres seront placés sur la même ligne que la macro-instruction et seront séparés par un espace.
Dans le programme de la macro-instruction la syntaxe « {?n} » fait référence au paramètre n.

Exemple :
Réalisons la macro-instruction « carre » qui élève le premier paramètre de la macro-instruction au carré et range le résultat dans le deuxième paramètre.
Appel de la macro-instruction :

```
lda 3
sta m200
%carre m200 m201
; m201 contiendra 9 ici
```

* Le nom de la macro-instruction peut être un chemin d'accès complet vers le fichier « .M », il peut contenir une désignation de lecteur et de répertoire.

Fichier « CARRE.M » :

```
#MACRO
lda {?0}
mla {?0}
sta {?1}
#ENDM
```

1.10.5. Librairie

La notion de librairie permet la définition de ressources qui seront compilées une seule fois dans une application, quel que soit le nombre d'appels à ces ressources.

Syntaxe de définition d'une librairie :

```
#LIBRARY <nom de la librairie>
<programme>
#ENDL
```

<nom de la librairie> est le nom de la fonction qui sera appelée par une instruction jsr :<nom de la librairie>:
Au premier appel rencontré par le compilateur, le code de la librairie est compilé. Pour les suivants, l'appel est simplement dirigé vers la routine existante.
Ce mécanisme est particulièrement adapté à l'utilisation des blocs fonctionnels et des macro-instructions pour limiter la génération de code dans le cas de l'utilisation multiple de mêmes ressources programmes.
Les mots m120 à m129 sont réservés aux librairies et peuvent être utilisés pour le passage des paramètres.

1.10.6. Macro-instructions prédéfinies

Des macro-instructions de conversion se trouvent dans le sous répertoire « LIB » du répertoire où a été installé AUTOMGEN.
Les équivalents en blocs fonctionnels sont également présents.

1.10.7. Description des macro-instructions prédéfinies

1.10.7.1. Conversions

```
%ASCTOBIN <deux premiers digits> <deux derniers digits> <résultat en binaire>
```

Effectue une conversion ASCII hexadécimal (deux premiers paramètres) vers binaire (troisième paramètre), en sortie l'accumulateur contient $FFFF si les deux premiers paramètres ne sont pas des nombres ASCII valides, 0 autrement. Tous les paramètres sont des mots de 16 bits.

`%BCDTOBIN <valeur en BCD> <valeur en binaire>`

Effectue une conversion BCD vers binaire. En sortie l'accumulateur contient $FFFF si le premier paramètre n'est pas un nombre bcd valide, 0 autrement. Les deux paramètres sont des mots de 16 bits.

`%BINTOASC <valeur en binaire> <résultat partie haute> <résultat partie basse>`

Effectue une conversion binaire (premier paramètre) vers ASCII hexadécimal (deuxième et troisième paramètres). Tous les paramètres sont des mots de 16 bits.

`%BINTOBCD <valeur en binaire> <valeur en BCD>`

Effectue une conversion BCD (premier paramètre) vers binaire (deuxième paramètre). En sortie l'accumulateur contient $FFFF si le nombre binaire ne peut être converti en BCD, 0 autrement.

`%GRAYTOB <valeur en code GRAY> <valeur en binaire>`

Effectue une conversion code Gray (premier paramètre) vers binaire (deuxième paramètre).

1.10.7.2. Traitement sur table de mots

`%COPY <premier mot table source> <premier mot table destination> <nombre de mots>`

Copie une table de mots source vers une table de mots destination. La longueur est donnée en nombre de mots.

`%COMP <premier mot table 1> <premier mot table 2> <nombre de mots> <résultat>`

Compare deux tables de mots. Le résultat est une variable binaire qui prend la valeur 1 si tous les éléments de la table 1 sont identiques à la table 2.

`%FILL <premier mot table> <valeur> <nombre de mots>`

Remplit une table de mots avec une valeur.

1.10.7.3. Traitement sur chaîne de caractères

Le codage des chaînes de caractères est le suivant : un caractère par mot, un mot contenant la valeur 0 marque la fin de la chaîne. Dans les macro-instructions, les chaînes sont passées en paramètres en désignant le premier mot qui les compose.

`%STRCPY <chaîne source> <chaîne destination>`

Copie une chaîne vers une autre.

`%STRCAT <chaîne source> <chaîne destination>`

Ajoute la chaîne source à la fin de la chaîne destination.

`%STRCMP <chaîne 1> <chaîne 2> <résultat>`

Compare deux chaînes. Le résultat est une variable booléenne qui passe à 1 si les deux chaînes sont identiques.

`%STRLEN <chaîne> <résultat>`

Place la longueur de la chaîne dans le mot résultat.

`%STRUPR <chaîne>`

Transforme tous les caractères de la chaîne en majuscules.

`%STRLWR <chaîne>`

Transforme tous les caractères de la chaîne en minuscules.

Exemple:

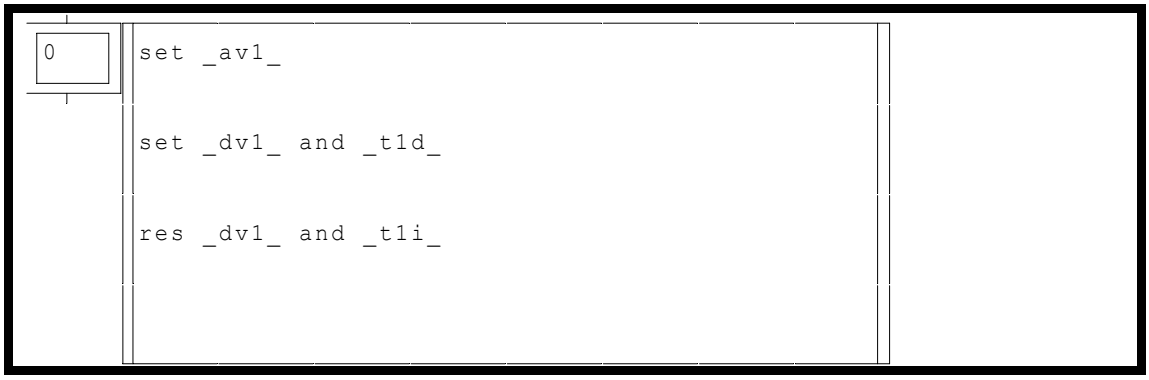
Conversion de m200 (binaire) vers m202, m203 en 4 digits (ASCII bcd)

```
%bintobcd m200 m201
%bintoasc m201 m202 m203
```

1.10.8. Exemple en langage littéral bas niveau

Cahier des charges : commençons par l'exemple le plus simple : aller et retour d'une locomotive sur la voie 1.

Solution :



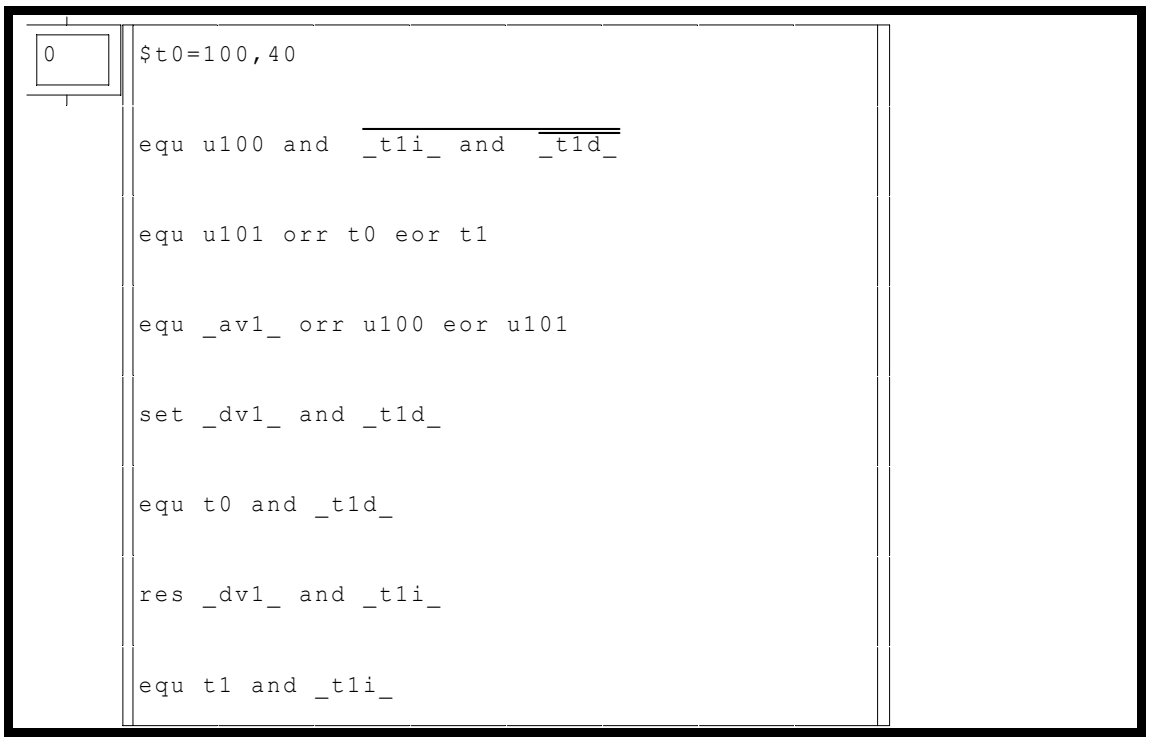
exemple\lit\littéral bas niveau1.agn

Un exemple un peu plus évolué.

Cahier des charges :

La locomotive devra maintenant marquer une attente de 10 secondes à l'extrémité droite de la voie et une attente de 4 secondes à l'extrémité gauche.

Solution :



exemple\lit\littéral bas niveau 2.agn

Autre exemple :

Cahier des charges :

Faire clignoter tous les feux de la maquette.

Solution :

0

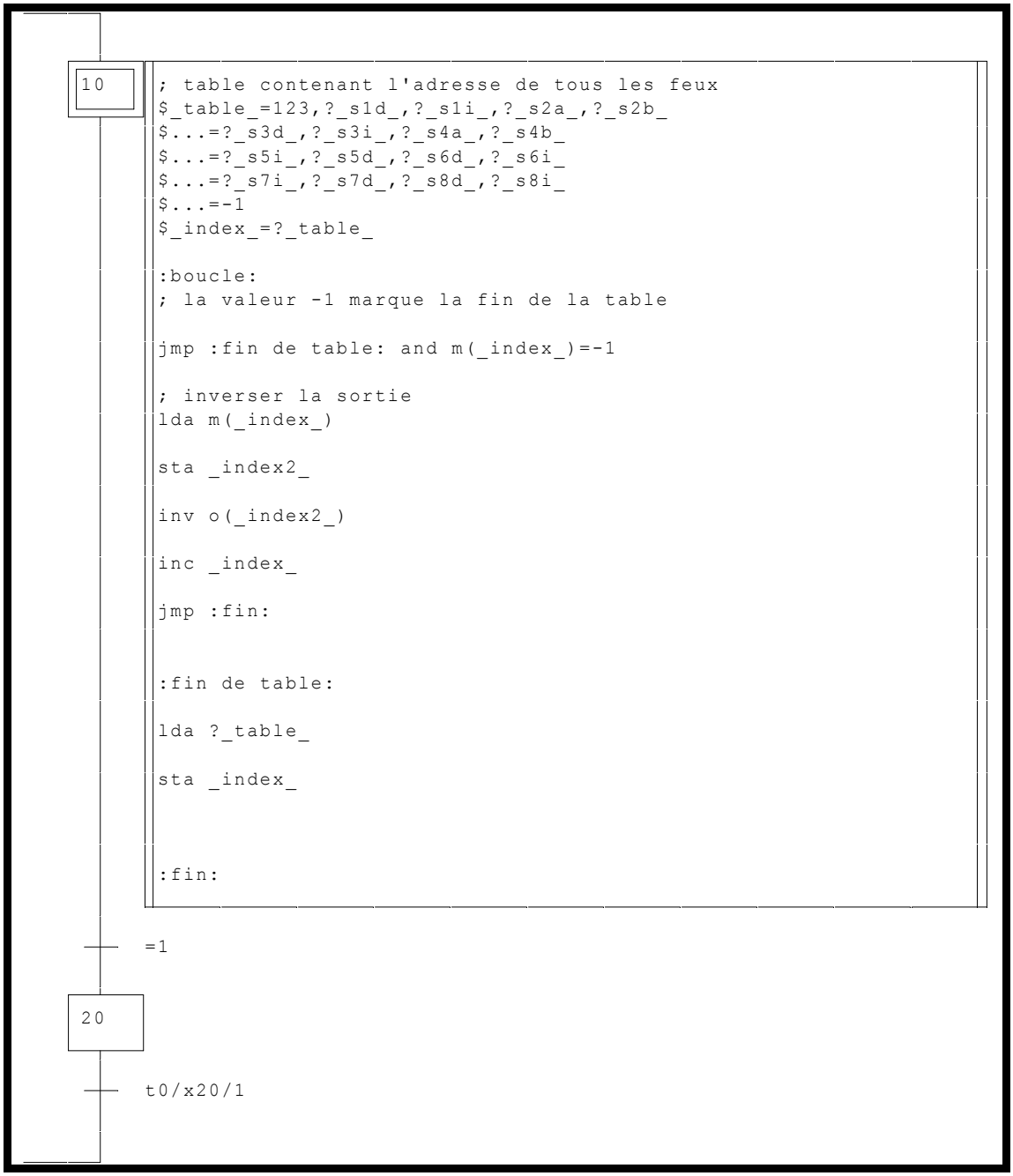
```
; table contenant l'adresse de tous les feux
$_table_=123,?_s1d_,?_s1i_,?_s2a_,?_s2b_
$...=?_s3d_,?_s3i_,?_s4a_,?_s4b_
$...=?_s5i_,?_s5d_,?_s6d_,?_s6i_
$...=?_s7i_,?_s7d_,?_s8d_,?_s8i_
$...=-1

; initialise l'index sur le debut de la table
lda ?_table_
sta _index_

:boucle:
; la valeur -1 marque la fin de la table
jmp :fin: and m(_index_)=-1

; inverser la sortie
lda m(_index_)

sta _index2_

inv o(_index2_)

inc _index_

jmp :boucle:









:fin:
```

exemple\lit\littéral bas niveau 3.agn

Cet exemple montre l'utilisation des prédispositions. Elles sont utilisées ici pour créer une table d'adresses de variables. La table contient l'adresse de toutes les sorties qui pilotent les feux de la maquette.
A chaque cycle d'exécution, l'état de tous les feux est inversé.

Un problème se pose, les feux clignotent très vite et l'on ne voit pas grand chose. Modifions notre exemple.
Cahier des charges :
Il faut maintenant inverser un par un l'état des feux tous les dixièmes de seconde.

Solution :



exemple\lit\littéral bas niveau 4.agn

1.11. Langage littéral étendu

Le langage littéral étendu est un « sur ensemble » du langage littéral bas niveau.
Il permet d'écrire plus simplement et sous une forme plus concise des équations booléennes et numériques.
Il permet également d'écrire des structures de type IF ... THEN ... ELSE et WHILE ... ENDWHILE (boucle).

L'utilisation du langage littéral étendu est soumise aux mêmes règles que le langage littéral bas niveau, il utilise la même syntaxe pour les variables, les mnémoniques, les types de test (fronts, état complémenté, état immédiat) et les modes d'adressage.
Il est possible de « mixer » le langage littéral bas niveau et le langage littéral étendu.
Lorsque le compilateur de langage littéral détecte une ligne écrite en langage littéral étendu, il la décompose en instructions du langage littéral bas niveau, puis la compile.

1.11.1. Ecriture d'équations booléennes

Syntaxe générale :

```
« variable bool.=(type d'affectation) (variable bool. 2 opérateur 1 variable bool. 3... opérateur n -1 variable bool. n ) »
```

Le type d'affectation doit être précisé s'il est autre que « Affectation ».
Il peut être :

⇨ « (/) » : affectation complémentée,

⇨ « (0) » : mise à zéro,

⇨ « (1) » : mise à un.

Les opérateurs peuvent être :

⇨ « . » : et,

⇨ « + » : ou.

Les équations peuvent contenir plusieurs niveaux de parenthèses pour préciser l'ordre d'évaluation. Par défaut, les équations sont évaluées de la gauche vers la droite.

Exemples et équivalences avec le langage littéral bas niveau:

```
o0=(i0)                    equ o0 and i0
o0=(i0.i1)                 equ o0 and i0 and i1
o0=(i0+i1)                 equ o0 orr i0 eor i1
o0=(1)                     set o0
o0=(0)                     res o0
```

o0=(1)(i0)	set o0 and i0
o0=(0)(i0)	res o0 and i0
o0=(1)(i0.i1)	set o0 and i0 and i1
o0=(0)(i0+i1)	res o0 orr o0 eor i1
o0=(/)(i0)	neq o0 and i0
o0=(/)(i0.i1)	neq o0 and i0 and i1
o0=(/i0)	equ o0 and /i0
o0=(/i0./i1)	equ o0 and /i0 and /i1
o0=(c0=10)	equ o0 and c0=10
o0=(m200<100+m200>200)	equ o0 orr m200<100 eor m200>200

1.11.2. Ecriture d'équations numériques

Syntaxe générale pour les entiers :

« variable num.1=[variable num.2 opérateur 1 ... opérateur n-1 variable num.n] »

Les équations peuvent contenir plusieurs niveaux de crochets pour préciser l'ordre d'évaluation. Par défaut, les équations sont évaluées de la gauche vers la droite.

Les opérateurs pour les entiers 16 et 32 bits peuvent être :

« + » : addition (équivalent à l'instruction ADA),
« - » : soustraction (équivalent à l'instruction SBA),
« * » : multiplication (équivalent à l'instruction MLA),
« / » : division (équivalent à l'instruction DVA),
« < » : décalage à gauche (équivalent à l'instruction RLA),
« > » : décalage à droite (équivalent à l'instruction RRA),
« & » : « Et » binaire (équivalent à l'instruction ANA),
« | »* : « Ou » binaire (équivalent à l'instruction ORA),
« ^ » : « Ou exclusif » binaire (équivalent à l'instruction XRA).

Les opérateurs pour les flottants peuvent être :

⇨ « + » : addition (équivalent à l'instruction ADA),

⇨ « - » : soustraction (équivalent à l'instruction SBA),

⇨ « * » : multiplication (équivalent à l'instruction MLA),

⇨ « / » : division (équivalent à l'instruction DVA).

On ne peut pas préciser de constante dans les équations sur les flottants. Si cela est nécessaire, il faut utiliser des prédispositions sur des flottants.

* Ce caractère est généralement associé à la combinaison de touches [ALT] + [6] sur les claviers.

Les équations sur les flottants peuvent faire appeler les fonctions « SQR » et « ABS ».

Remarque : suivant la complexité des équations, le compilateur peut utiliser des variables intermédiaires. Ces variables sont les mots m53 à m59 pour les entiers 16 bits, les longs l53 à l59 pour les entiers 32 bits et les flottants f53 à f59.

Exemples et équivalences avec le langage littéral bas niveau :

```
m200=[10]                          lda 10
                                   sta m200
m200=[m201]                        lda m201
                                   sta m200
m200=[m201+100]                    lda m201
                                   ada 100
                                   sta m200
m200=[m200+m201-m202]              lda m200
                                   ada m201
                                   sba m202
                                   sta m200
m200=[m200&$ff00]                  lda m200
                                   ana $ff00
                                   sta m200
f200=[f201]                        lda f201
                                   sta f200
f200=[f201+f202]                   lda f201
                                   ada f202
                                   sta f200
f200=[sqr[f201]]                   lda f201
                                   sqr aaa
                                   sta f200
f200=[sqr[abs[f201*100R]]]         lda f201
                                   mla 100R
                                   abs aaa
                                   sqr aaa
                                   sta f200

l200=[l201+$12345678L]             lda l201
                                   ada $12345678L
                                   sta l200
```

1.11.3. Structure de type IF ... THEN ... ELSE ...

Syntaxe générale :

```
IF(test)
      THEN
      action si test vrai
      ENDIF
      ELSE
      action si test faux
      ENDIF
```

Le test doit respecter la syntaxe décrite au chapitre consacré aux équations booléennes dans ce chapitre.
Seule une action si test vrai ou une action si test faux peut figurer.
Il est possible d'imbriquer plusieurs structures de ce type.
Les bits Système u90 à u99 sont utilisés comme variables temporaires pour la gestion de ce type de structure.

Exemples :

```
IF(i0)
      THEN
      inc m200            ; incrémenter le mot 200 si i0
      ENDIF

IF(i1+i2)
      THEN
      m200=[m200+10]      ; ajouter 10 au mot 200 si i1 ou i2
      ENDIF
      ELSE
      res m200            ; sinon effacer m200
      ENDIF
```

1.11.4. Structure de type WHILE ... ENDWHILE

Syntaxe générale :

```
WHILE(test)
      action à répéter tant que le test est vrai
ENDWHILE
```

Le test doit respecter la syntaxe décrite au chapitre consacré aux équations booléennes dans ce chapitre.

Il est possible d'imbriquer plusieurs structures de ce type.

Les bits Système u90 à u99 sont utilisés comme variables temporaires pour la gestion de ce type de structure.

Exemples :

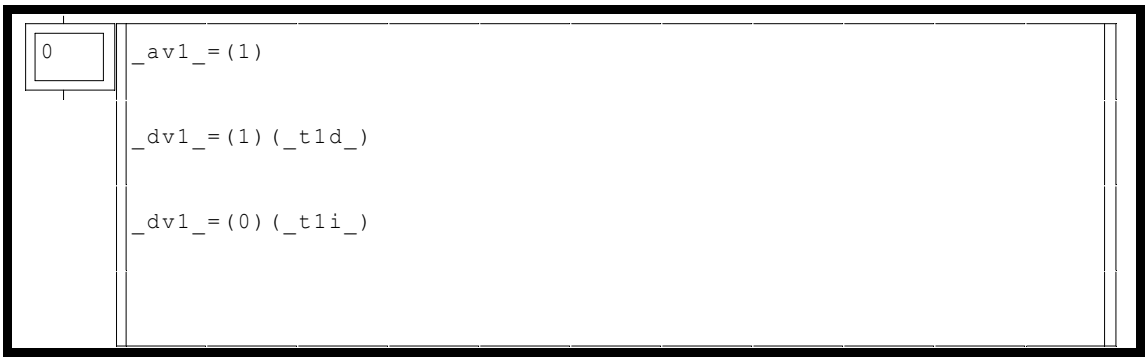
```
m200=[0]
WHILE(m200<10)
       set o(200)
       inc m200            ; incrémenter le mot 200
ENDWHILE
```

Cet exemple effectue une mise à un des sorties o0 à o9.

1.11.5. Exemple de programme en langage littéral étendu

Reprenons le premier exemple du chapitre précédent.

Solution :



exemple\lit\littéral étendu 1.agn

Enrichissons notre exemple avec des calculs.

Cahier des charges :

Calculer la vitesse en millimètres par seconde et en mètres par heure de la locomotive sur le trajet gauche vers droite.

Solution :

exemple\lit\littéral étendu 2.agn

Le mot 32 est utilisé pour lire le temps Système. La valeur est ensuite transférée dans des flottants pour pouvoir effectuer les calculs sans perte de précision.

1.12. Langage littéral ST

Le langage littéral ST est le langage littéral structuré défini par la norme CEI1131-3. Ce langage permet d'écrire des équations booléennes et numériques ainsi que des structures de programmation.

1.12.1. Généralités

Le langage littéral ST s'utilise aux mêmes endroits que le langage littéral bas niveau et le langage littéral étendu.

Des directives permettent de définir des sections en langage littéral ST :
« #BEGIN_ST » marque le début d'une section en langage ST.
« #END_ST » marque la fin d'une section en langage ST.
Exemple :

```
m200=[50]           ; langage littéral étendu
#BEGIN_ST
m201:=4;            (* langage ST *)
#END_ST
```

Il est également possible de choisir l'utilisation du langage ST pour tout un folio. Le choix s'effectue dans la boîte de dialogue de propriétés de chaque folio.

Dans un folio ou le langage ST est le langage par défaut il est possible d'insérer du langage littéral bas niveau et étendu en encadrant les lignes par deux directives « #END_ST » et « #BEGIN_ST ».

Pour le langage ST les commentaires doivent débuter par « (* » et se terminer par « *) ».

Les instructions du langage ST sont terminées par le caractère « ; ». Plusieurs instructions peuvent être écrites sur une même ligne.

Exemple :

```
o0:=1; m200:=m200+1;
```

1.12.2. Equations booléennes

La syntaxe générale est :

```
variable := équation booléenne;
```

L'équation booléenne peut être composée d'une constante, d'une variable ou de plusieurs variables séparées par des opérateurs.

Les constantes peuvent être : 0, 1, FALSE ou TRUE.

Exemples :

```
o0:=1;
o1:=FALSE;
```

Les opérateurs permettant de séparer plusieurs variables sont : + (ou), . (et), OR ou AND.

Le « Et » est prioritaire sur le « Ou ».

Exemple :

```
o0:=i0+i1.i2+i3;
```

Sera traité comme :

```
o0:=i0+(i1.i2)+i3;
```

Les parenthèses peuvent être utilisées dans les équations pour spécifier les priorités.

Exemple :

```
o0:=(i0+i1).(i2+i3);
```

Des tests numériques peuvent êtres utilisés.

Exemple :

```
o0:=m200>5.m200<100;
```

1.12.3. Equations numériques

La syntaxe générale est :

```
variable := équation numérique;
```

L'équation numérique peut être composée d'une constante, d'une variable ou de plusieurs variables et constantes séparées par des opérateurs.

Les constantes peuvent être des valeurs exprimées en décimal, hexadécimal (préfixe 16#) ou binaire (préfixe 2#).

Exemples :

```
m200:=1234;
m201:=16#aa55;
m202:=2#100000011101;
```

Les opérateurs permettant de séparer plusieurs variables ou constantes sont dans l'ordre de leurs priorités:
* (multiplication),/ (division), + (addition), - (soustraction), & ou AND (et binaire), XOR (ou exclusif binaire), OR (ou binaire).

Exemples :

```
m200:=1000*m201;
m200:=m202-m204*m203;            (* équivalent à m200:=m202-(m204*m203) *)
```

Les parenthèses peuvent être utilisées dans les équations pour spécifier les priorités.

Exemple :

```
m200:=(m202-m204)*m203;
```

1.12.4. Structures de programmation

1.12.4.1. Test SI ALORS SINON

Syntaxe :

```
IF condition THEN action ENDIF;
```

et

```
IF condition THEN action ELSE action ENDIF;
```

Exemple :

```
if i0
      then o0:=TRUE;
      else
      o0:=FALSE;
      if i1 then m200:=4; endif;
```

```
endif ;
```

1.12.4.2. Boucle TANT QUE

Syntaxe :

```
WHILE condition DO action ENDWHILE;
```

Exemple :

```
while m200<1000
       do
       m200:=m200+1;
endwhile;
```

1.12.4.3. Boucle JUSQU'A CE QUE

Syntaxe :

```
REPEAT action UNTIL condition; ENDREPEAT;
```

Exemple :

```
repeat
       m200:=m200+1;
until m200=500
endrepeat;
```

1.12.4.4. Boucle DEPUIS JUSQU'A

Syntaxe :

```
FOR variable:=valeur de départ TO valeur de fin DO action ENDFOR;
```

ou

```
FOR variable:=valeur de départ TO valeur de fin BY pas DO action ENDFOR;
```

Exemple :

```
for m200:=0 to 100 by 2
       do
       m201:=m202*m201;
       endfor;
```

1.12.4.5. Sortie de boucle

Le mot clé « EXIT » permet de sortir d'une boucle.

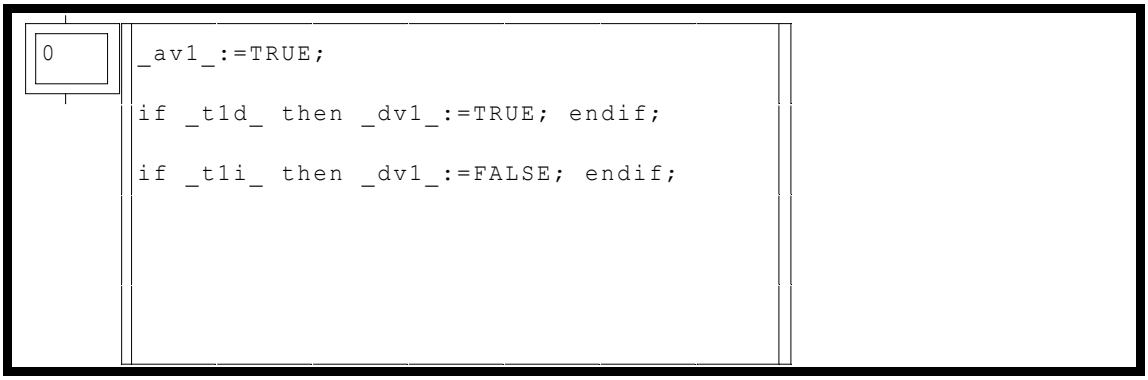
Exemple :

```
while i0
       m200:=m200+1;
       if m200>1000 then exit; endif;
endwhile;
```

1.12.5. Exemple de programme en langage littéral étendu

Reprenons le premier exemple du chapitre précédent.

Solution :



exemple\lit\littéral ST 1.agn

1.13. Organigramme

AUTOMGEN implémente une programmation de type « organigramme ».
Pour utiliser ce type de programmation il faut utiliser les langages littéraux Veuillez consulter les chapitres précédents pour apprendre à utiliser ces langages.
L'intérêt de la programmation sous forme d'organigramme est la représentation graphique d'un traitement algorithmique.
Contrairement au langage Grafcet, la programmation sous forme d'organigramme génère un code qui sera exécuté une fois par cycle de scrutation. Cela signifie que l'on ne peut rester en attente dans un rectangle d'organigramme, il faut obligatoirement que l'exécution sorte de l'organigramme pour pouvoir continuer à exécuter la suite du programme.

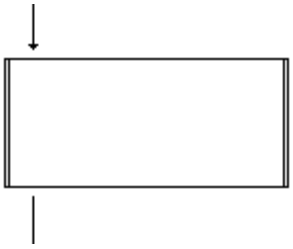
C'est un point très important à ne pas oublier lorsqu'on choisit ce langage.
Seuls des rectangles peuvent être dessinés. C'est le contenu des rectangles et les liaisons qui en partent qui déterminent si le rectangle est une action ou un test.

1.13.1. Dessin d'un organigramme

Les rectangles sont dessinés en choisissant la commande « Plus … / Boîte de code » du menu contextuel (cliquez sur le bouton droit de la souris sur le fond du folio pour ouvrir le menu contextuel).

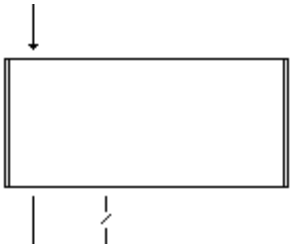
Il faut placer un bloc ↓ (touche [<]) à l'entrée de chacun des rectangles, cette entrée doit être placée sur la partie supérieure du rectangle.
Si le rectangle représente une action, il y aura une seule sortie matérialisée par un bloc | (touche [E]) en bas et à gauche du rectangle.

Un rectangle d'action :



Si le rectangle représente un test, il y aura obligatoirement deux sorties. La première, matérialisée par un bloc | (touche [E]) en bas et à gauche représente la sortie si le test est vrai, la deuxième, matérialisée par un bloc ⁞ (touche [=]) se trouvant immédiatement à droite de l'autre sortie représente la sortie si le test est faux.

Un rectangle de test :



Les branches d'organigrammes doivent toujours se terminer par un rectangle sans sortie qui peut éventuellement rester vide.

1.13.2. Contenu des rectangles

1.13.2.1. Contenu des rectangles d'action

Les rectangles d'action peuvent contenir n'importe quelles instructions du langage littéral.

1.13.2.2. Contenu des rectangles de test

Les rectangles de test doivent contenir un test respectant la syntaxe de la partie test de la structure de type IF...THEN...ELSE... du langage littéral étendu.
Par exemple :

```
IF (i0)
```

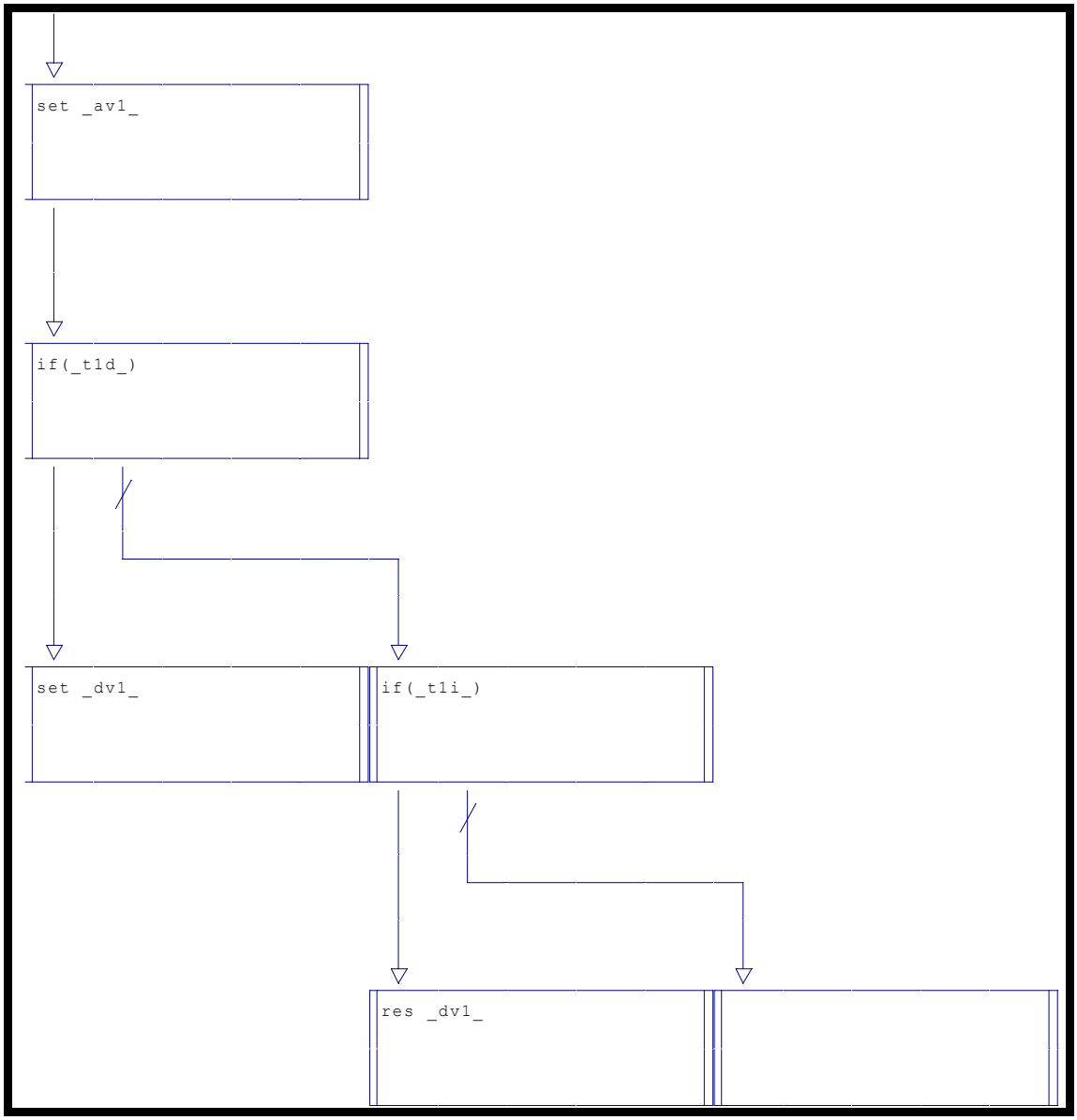
Il est possible d'écrire avant ce test des actions dans le rectangle de test.
Cela permet par exemple d'effectuer certains calculs avant le test.
Si par exemple nous voulons tester si le mot 200 est égal au mot 201 plus 4 :

```
m202=[m201+4]
IF(m200=m202)
```

1.14. Illustration

Notre premier exemple désormais classique, consistera à faire effectuer des allers et retours à une locomotive sur la voie 1 de la maquette.

Solution :



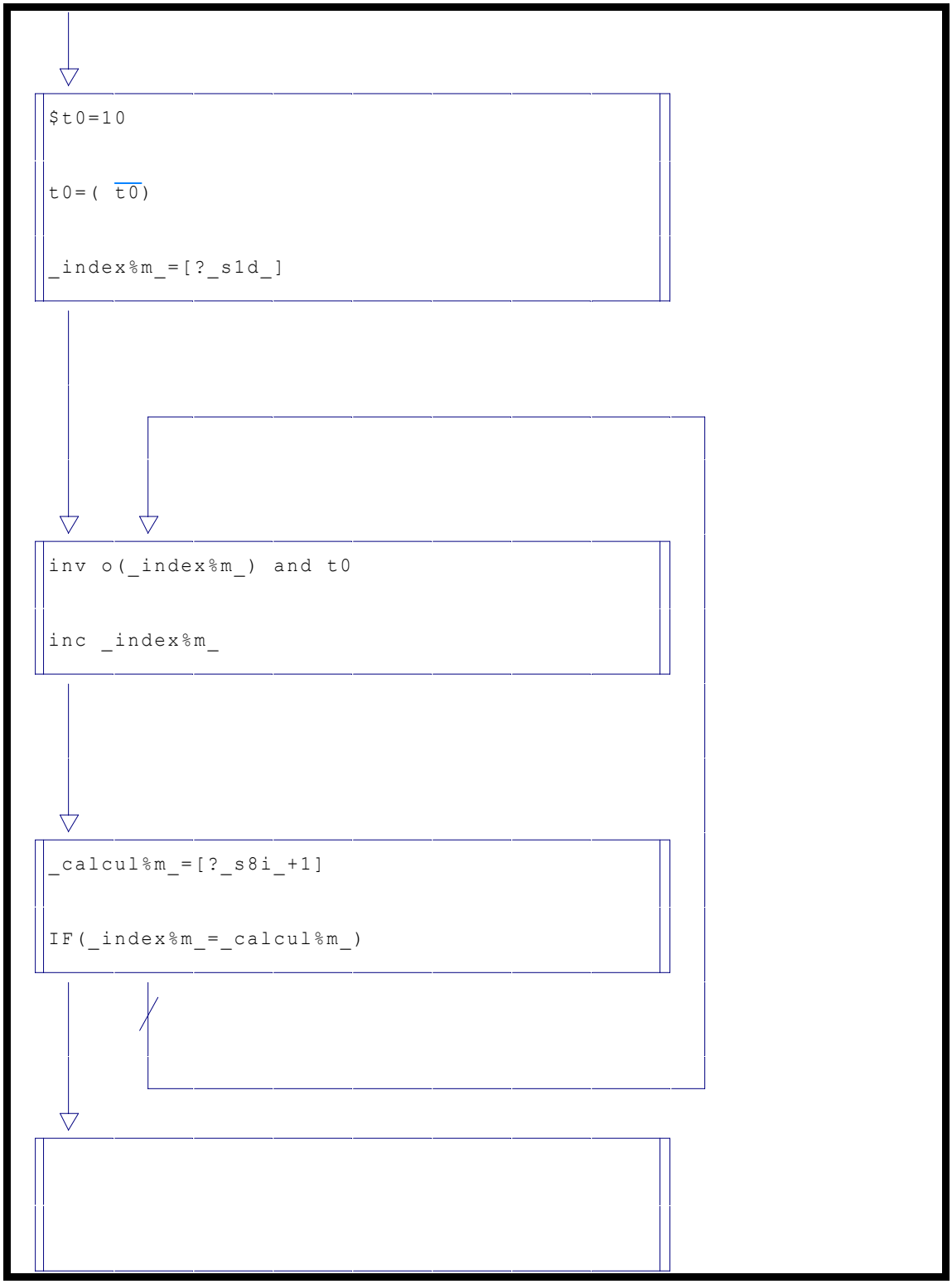
exemple\organigramme\organigramme 1.agn

Deuxième exemple

Cahier des charges :

Faire clignoter tous les feux de la maquette. Les feux changeront d'état toutes les secondes.

Solution :



exemple\organigramme\organigramme 2.agn

Notez l'utilisation de symboles automatiques dans cet exemple.

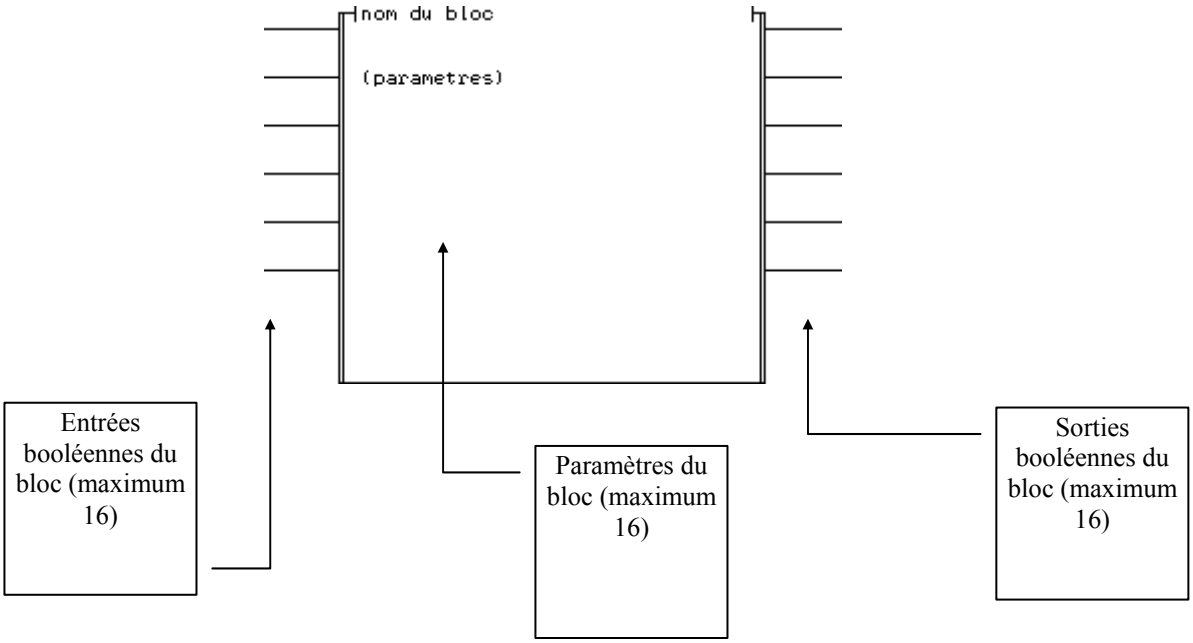
1.15. Blocs fonctionnels

AUTOMGEN implémente la notion de blocs fonctionnels.

Cette méthode de programmation modulaire permet d'associer à un élément graphique un ensemble d'instructions écrites en langage littéral.

Les blocs fonctionnels sont définissables par le programmeur. Leur nombre n'est pas limité. Il est ainsi possible de constituer des ensembles de blocs fonctionnels permettant une conception modulaire et standardisée des applications.

Les blocs fonctionnels s'utilisent à l'intérieur de schémas de types logigramme ou ladder, ils possèdent de une à n entrées booléennes et de une à n sorties booléennes. Si le bloc doit traiter des variables autres que booléennes, alors celles-ci seront mentionnées dans le dessin du bloc fonctionnel. L'intérieur du bloc peut recevoir des paramètres : constantes ou variables.



1.15.1. Création d'un bloc fonctionnel

Un bloc fonctionnel est composé de deux fichiers distincts. Un fichier portant l'extension « .ZON » qui contient le dessin du bloc fonctionnel et un fichier portant l'extension « .LIB » qui contient une suite d'instructions écrites en langage littéral définissant le fonctionnement du bloc fonctionnel.

Les fichiers « .ZON » et « .LIB » doivent porter le nom du bloc fonctionnel. Par exemple si nous décidons de créer un bloc fonctionnel « MEMOIRE », nous

devrons créer les fichiers « MEMOIRE.ZON » (pour le dessin du bloc) et « MEMOIRE.LIB » (pour le fonctionnement du bloc).

1.15.2. Dessin du bloc et création du fichier « .ZON »

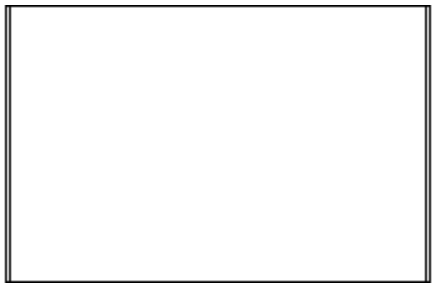
L'enveloppe d'un bloc fonctionnel est constituée d'une boîte de code à laquelle il faut ajouter des blocs dédiés aux blocs fonctionnels.

Pour dessiner un bloc fonctionnel il faut effectuer les opérations suivantes :

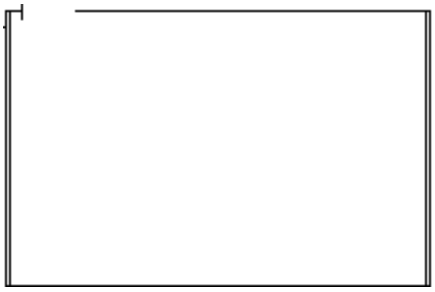
⇨ utilisez l'assistant (conseillé)

Ou :

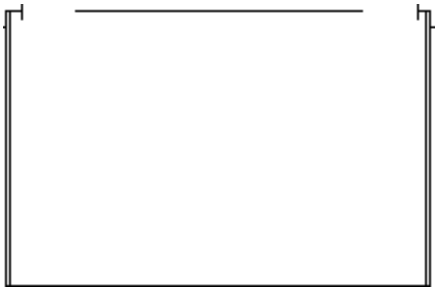
⇨ dessiner une boîte de code (utilisez la commande « Plus …/Boîte de code » du menu contectuel) :



⇨ poser un bloc  (touche [8]) sur le coin supérieur gauche de la boîte de code :

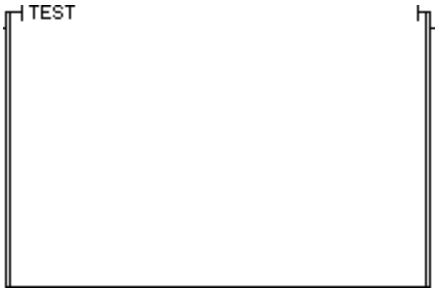


⇨ poser un bloc  (touche [9]) sur le coin supérieur droit de la boîte de code :



⇨ effacer la ligne qui reste en haut du bloc (la touche [A] permet de poser des blocs blancs) :

⇨ cliquez avec le bouton gauche de la souris sur le coin supérieur gauche du bloc fonctionnel, entrer alors le nom du bloc fonctionnel qui ne doit pas dépasser 8 caractères (les fichiers « .ZON » et « .LIB » devront porter ce nom), presser ensuite [ENTER].



⇨ si des entrées booléennes supplémentaires sont nécessaires, il faut utiliser un bloc (touche [;]) ou (touche [:]), les entrées ainsi ajoutées doivent se trouver immédiatement en dessous de la première entrée, aucun espace libre ne doit être laissé,

⇨ si des sorties booléennes supplémentaires sont nécessaires il faut ajouter un bloc (touche [>]) ou (touche [?]), les sorties ainsi ajoutées doivent se trouver immédiatement en dessous de la première sortie, aucun espace libre ne doit être laissé,

⇨ l'intérieur du bloc peut contenir des commentaires ou des paramètres, les paramètres sont écrits entre accolades « {...} ». Tout ce qui n'est pas écrit entre accolades est ignoré par le compilateur. Il est intéressant de repérer l'usage des entrées et des sorties booléennes à l'intérieur du bloc.

⇨ lorsque le bloc est terminé, il faut utiliser la commande « Sélectionner » du menu « Edition » pour sélectionner le dessin du bloc fonctionnel, puis le sauvegarder dans un fichier « .ZON » avec la commande « Copier vers » du menu « Edition ».

1.15.3. Création du fichier « .LIB »

Le fichier « .LIB » est un fichier texte contenant des instructions en langage littéral (bas niveau ou étendu). Ces instructions définissent le fonctionnement du bloc fonctionnel.

Une syntaxe spéciale permet de faire référence aux entrées booléennes du bloc, aux sorties booléennes du bloc et aux paramètres du bloc.

Pour faire référence à une entrée booléenne du bloc, il faut utiliser la syntaxe « {Ix} » ou x est le numéro de l'entrée booléenne exprimé en hexadécimal (0 à f).

Pour faire référence à une sortie booléenne du bloc, il faut utiliser la syntaxe « {Ox} » ou x est le numéro de la sortie booléenne exprimé en hexadécimal (0 à f).

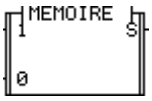
Pour faire référence à un paramètre du bloc, il faut utiliser la syntaxe « {?x} » ou x est le numéro du paramètre en hexadécimal (0 à f).

Le fichier .LIB peut être placé dans sous-répertoire « lib » du répertoire d'installation d'AUTOMGEN ou dans les ressources du projet.

1.15.4. Exemple simple de bloc fonctionnel

Créons le bloc fonctionnel « MEMOIRE » qui possède deux entrées booléennes (mise à un et mise à zéro) et une sortie booléenne (l'état de la mémoire).

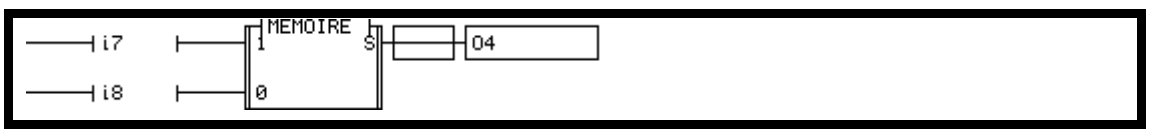
Le dessin du bloc contenu dans le fichier « MEMOIRE.ZON » est :



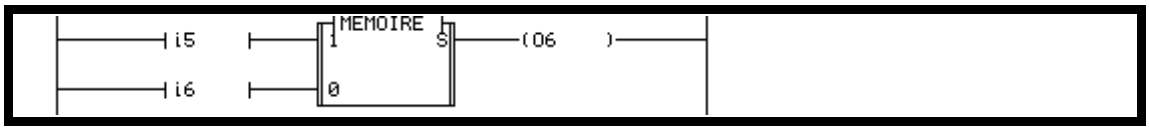
Le fonctionnement du bloc contenu dans le fichier « MEMOIRE.LIB » est :

```
{O0}=(1)({I0})
{O0}=(0)({I1})
```

Le bloc peut ensuite être utilisé de la façon suivante :



ou



Pour utiliser un bloc fonctionnel dans une application il faut choisir la commande « Coller à partir de » du menu « Edition » et choisir le fichier « .ZON » correspondant au bloc fonctionnel à utiliser.

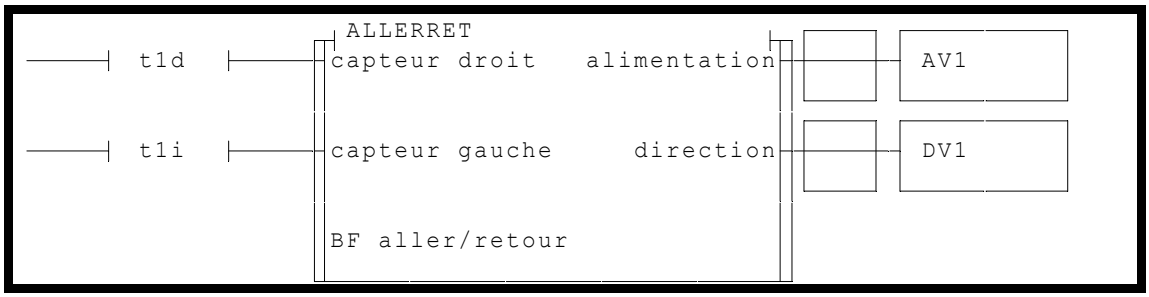
1.15.5. Illustration

Reprenons un exemple désormais classique.

Cahier des charges :

Aller et retour d'une locomotive sur la voie 1 de la maquette.

Solution :



exemple\bf\bloc-fonctionnel 1.agn

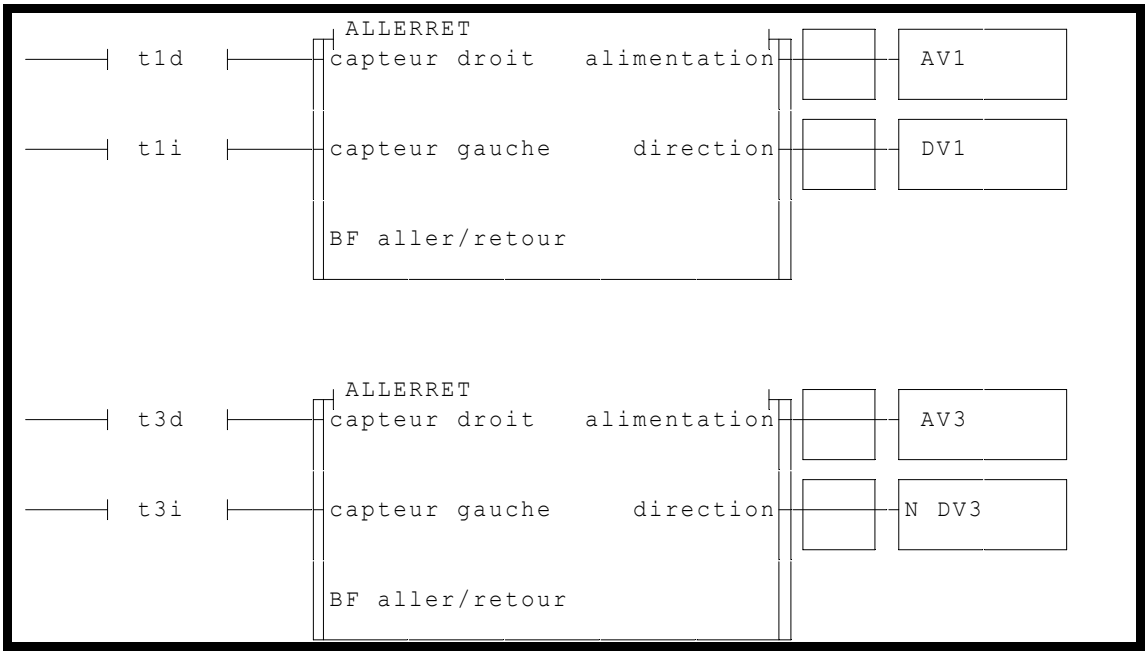
```
; bloc fonctionnel ALLERRET
; aller retour d'une locomotive sur une voie
; les entrées booléennes sont les fins de course
; les sorties booléennes sont l'alimentation de la voie (0) et la
direction (1)

; toujours alimenter la voie
set {O0}

; piloter la direction en fonction des fins de course

{O1}=(1)({I0})
{O1}=(0)({I1})
```

Pour illustrer l'intérêt de l'utilisation des blocs fonctionnels, complétons notre exemple.

Cahier des charges :

Aller et retour de deux locomotives sur les voies 1 et 3.

Solution :



exemple\bf\bloc-fonctionnel 2.agn

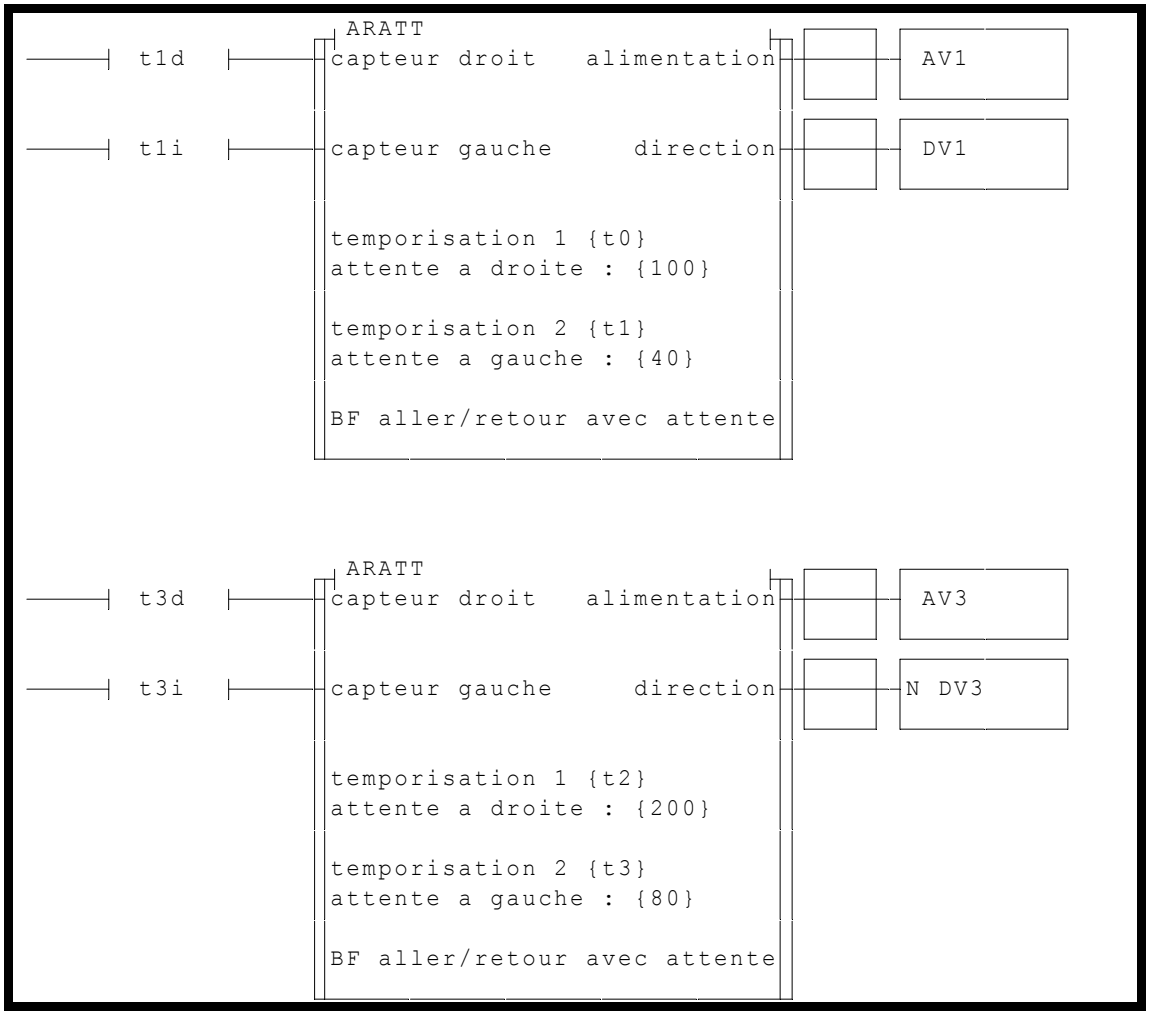
Cet exemple montre qu'avec le même bloc fonctionnel, il est aisé de faire fonctionner de façon identique différents modules d'une partie opérative.

Complétons notre exemple pour illustrer l'utilisation de paramètres.

Cahier des charges :

Les deux locomotives devront maintenant marquer une attente en bout de voie. Pour la locomotive 1: 10 secondes à droite et 4 secondes à gauche, pour la locomotive 2 : 20 secondes à droite et 8 secondes à gauche.

Solution :


ARATT
t1d
capteur droit
alimentation
AV1
t1i
capteur gauche
direction
DV1
temporisation 1 {t0}
attente a droite : {100}
temporisation 2 {t1}
attente a gauche : {40}
BF aller/retour avec attente
ARATT
t3d
capteur droit
alimentation
AV3
t3i
capteur gauche
direction
N DV3
temporisation 1 {t2}
attente a droite : {200}
temporisation 2 {t3}
attente a gauche : {80}
BF aller/retour avec attente

```
; bloc fonctionnel ARATT
; aller retour d'une locomotive sur une voie avec attente
; les entrées booléennes sont les fins de course
; les sorties booléennes sont l'alimentation de la voie (0) et la
direction (1)
; les paramètres sont :
;               0 : première temporisation
;               1 : durée de la première temporisation
;               2 : deuxième temporisation
;               3 : durée de la deuxième temporisation


; prédisposition des deux temporisations
${?0}={?1}
${?2}={?3}

; alimenter la voie si pas les fins de course ou si tempo. terminées
set {O0}
res {O0} orr {I0} eor {I1}
set {O0} orr {?0} eor {?2}

; gestion des temporisations
{?0}=({I0})
{?2}=({I1})

; piloter la direction en fonction des fins de course
{O1}=(1)({I0})
{O1}=(0)({I1})
```

exemple\bf\bloc-fonctionnel 3.agn

1.15.6. Complément de syntaxe

Une syntaxe complémentaire permet d'effectuer un calcul sur les numéros de variables référencées dans le fichier « .LIB ».

La syntaxe « ~+n » ajoutée à la suite d'une référence à une variable ou un paramètre, ajoute n.

La syntaxe « ~-n » ajoutée à la suite d'une référence à une variable ou un paramètre, soustrait n.

La syntaxe « ~*n » ajoutée à la suite d'une référence à une variable ou un paramètre, multiplie par n.

On peut écrire plusieurs de ces commandes à la suite, elles sont évaluées de la gauche vers la droite.

Ce mécanisme est utile lorsqu'un paramètre du bloc fonctionnel doit permettre de faire référence à une table de variables.

Exemples :

```
{?0}~+1
```

fait référence à l'élément suivant le premier paramètre, par exemple si le premier paramètre est m200 cette syntaxe fait référence à m201.

```
M{?2}~*100~+200
```

fait référence au troisième paramètre multiplié par 100 plus 200, par exemple si le troisième paramètre est 1 cette syntaxe fait référence à M 1*100 + 200 donc M300.

1.16. Blocs fonctionnels évolués

Cette fonctionnalité permet de créer des blocs fonctionnels très puissants avec plus de simplicité que les blocs fonctionnels gérés par des fichiers écrits en langage littéral. Cette méthode de programmation permet une approche de type analyse fonctionnelle.

N'importe quel folio ou ensemble de folios peut devenir un bloc fonctionnel (on parle parfois d'encapsuler un programme).

Le ou les folios décrivant le fonctionnement d'un bloc fonctionnel peuvent accéder aux variables externes du bloc fonctionnel : les entrées booléennes du bloc, les sorties booléennes et les paramètres.

Le principe d'utilisation et notamment l'utilisation des variables externes reste identique aux anciens blocs fonctionnels.

1.16.1. Syntaxe

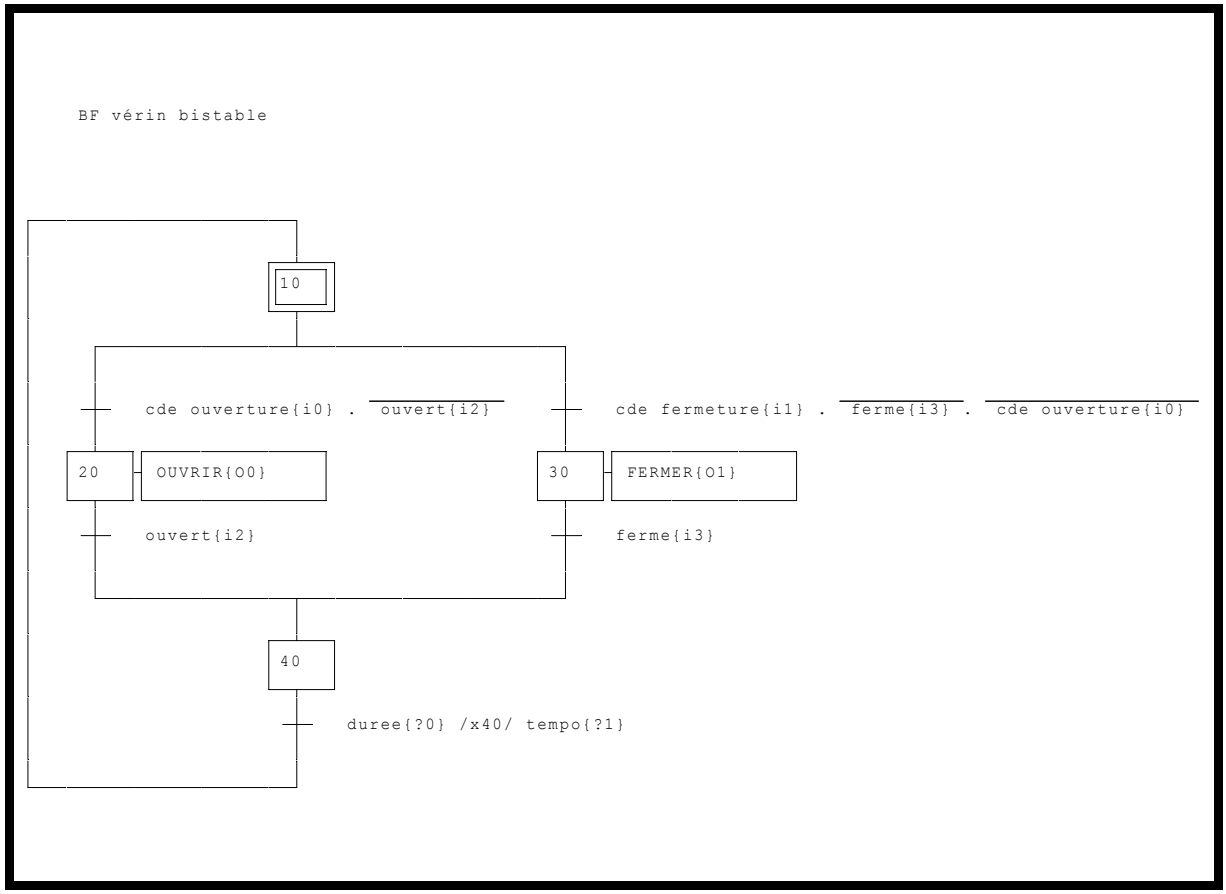
Pour référencer une variable externe d'un bloc fonctionnel il faut utiliser un mnémonique incluant le texte suivant : {In} pour référencer l'entrée booléenne n, {On} pour référencer la sortie booléenne n, {?n} pour référencer le paramètre n. Le mnémonique doit commencer par une lettre.

1.16.2. Différencier anciens et nouveaux blocs fonctionnels

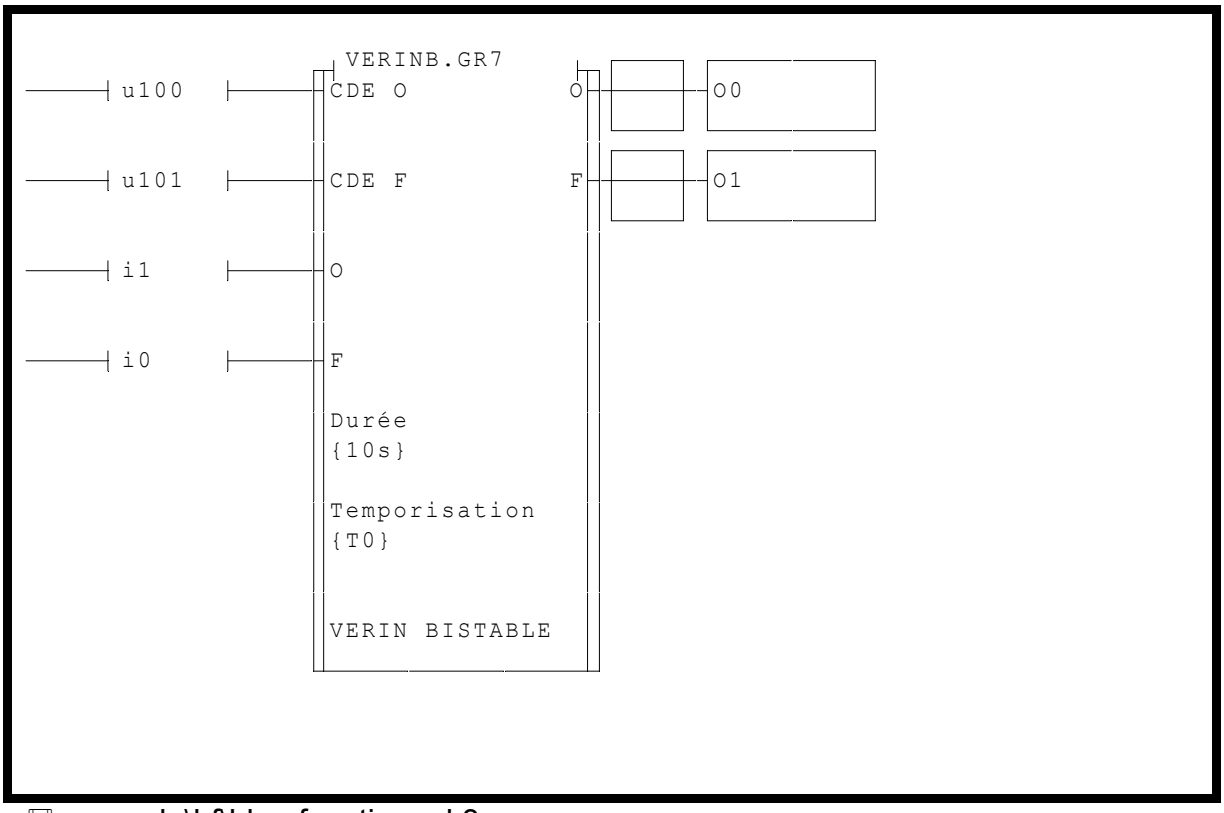
Le nom de fichier inscrit sur le dessin bloc fonctionnel indique s'il s'agit d'un ancien (géré par un fichier .LIB) ou d'un nouveau bloc fonctionnel (géré par un folio .GR7). Pour un ancien bloc fonctionnel le nom ne porte pas d'extension, pour un nouveau l'extension .GR7 doit être ajoutée. Le folio contenant le code qui gère le fonctionnement du bloc fonctionnel doit être intégré dans la liste des folios du projet. Dans les propriétés du folio, le type « Bloc-fonctionnel » doit être choisi.

1.16.3. Exemple

Contenu du folio VERINB :



Appel du bloc fonctionnel :



exemple\bf\bloc-fonctionnel 3.agn

1.17. Blocs fonctionnels prédéfinis

Des blocs fonctionnels de conversion se trouvent dans le sous-répertoire « \LIB » du répertoire où a été installé AUTOMGEN.

Les équivalents en macro-instructions sont également présents voir le chapitre 1.10.3. Le langage littéral bas niveau.

Pour insérer et paramétrer un bloc fonctionnel dans une application, sélectionnez la commande « Insérer un bloc fonctionnel » du menu « Outils ».

1.17.1. Blocs de conversion

ASCTOBIN : conversion ASCII vers binaire

BCDTOBIN : conversion BCD vers binaire

BINTOASC : conversion binaire vers ASCII

BINTOBCD : conversion binaire vers BCD

GRAYTOB : conversion code gra vers binaire

16BINTOM : transfert de 16 variables booléennes dans un mot

MTO16 BIN : transfert d'un mot vers 16 variables booléennes

1.17.2. Blocs de temporisation

TEMPO : temporisation à la montée
PULSOR : sortie à créneau
PULSE : impulsion temporisée

1.17.3. Blocs de manipulations de chaîne de caractères

STRCMP : comparaison
STRCAT : concaténation
STRCPY : copie
STRLEN : calcul de la longueur
STRUPR : mise en minuscules
STRLWR : mise en majuscules

1.17.4. Blocs de manipulation de table de mots

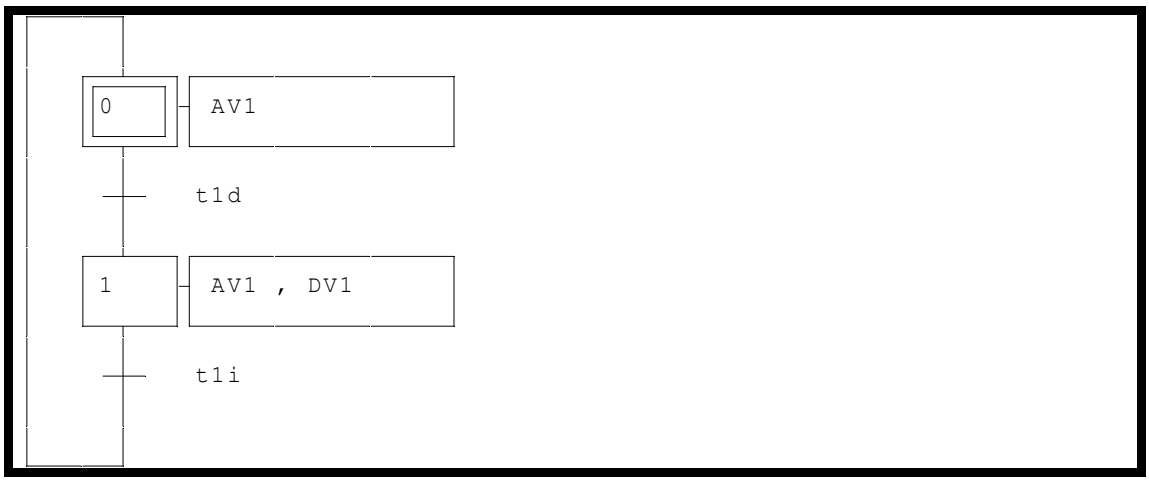
COMP : comparaison
COPY : copie
FILL : remplissage

1.18. Techniques avancées

1.18.1. Code généré par le compilateur

Nous allons aborder dans ce chapitre, la forme du code généré par la compilation de tel ou tel type de programme.
L'utilitaire « CODELIST.EXE »* permet de traduire « en clair » un fichier de code intermédiaire « .EQU » (aussi appelé langage pivot).
Faisons l'expérience suivante : chargeons et compilons le premier exemple de programme du chapitre « Grafcet » : « simple1.agn » du répertoire « exemple\grafcet » :

* Cet utilitaire doit être exécuté à partir de la ligne de commande DOS.



Double cliquez sur l'élément « Fichiers générés/Code pivot » dans le navigateur.
Vous obtenez la liste d'instructions suivantes :

```
; Le code qui suit a été généré par la compilation de : 'Folio : GRAF1'
:00000000: RES x0  AND i0
:00000002: SET x0  AND b0
:00000004: SET x0  AND x1   AND i1
:00000007: RES x1  AND i1
:00000009: SET x1  AND x0   AND i0
; Le code qui suit a été généré par la compilation de : 'affectations
(actions Grafcet, logigrammes et ladder)'
:0000000C: EQU o0  ORR @x0  EOR @x1
:0000000F: EQU o23 AND @x1
```

Elle représente la traduction de l'application « simple1.agn » en instructions du langage littéral bas niveau.
Les commentaires indiquent la provenance des portions de code, cela est utile si une application est composée de plusieurs folios.
Obtenir cette liste d'instructions peut être utile pour répondre aux questions concernant le code généré par telle ou telle forme de programme ou l'utilisation de tel ou tel langage.
Dans certains cas « critiques », pour lesquels il est important de connaître des informations comme « au bout de combien de cycles cette action devient-elle vraie ? » le mode pas à pas et l'examen approfondi du code généré s'avèrent indispensables.

1.18.2. Optimisation du code généré

Plusieurs niveaux d'optimisation sont possibles.

1.18.2.1. Optimisation du code généré par le compilateur

L'option d'optimisation du compilateur permet de réduire sensiblement la taille du code généré. Cette directive demande au compilateur de générer moins de lignes de langage littéral bas niveau, ce qui a pour conséquence d'augmenter le temps de compilation.
Suivant les post-processeurs utilisés, cette option entraîne un gain sur la taille du code et ou le temps d'exécution. Il convient d'effectuer des essais pour déterminer si cette directive est intéressante ou pas suivant la nature du programme et le type de cible utilisée.
Il est en général intéressant de l'utiliser avec les post-processeurs pour cibles Z.

1.18.2.2. Optimisation du code généré par les post-processeurs

Chaque post-processeur peut posséder des options pour optimiser le code généré. Pour les post-processeurs qui génèrent du code constructeur veuillez consulter la notice correspondante.

1.18.2.3. Optimisation du temps de cycle : réduire le nombre de temporisations sur cibles Z

Pour les cibles Z, le nombre de temporisations déclarées influe directement sur le temps de cycle. Veillez à déclarer le minimum de temporisations en fonction des besoins de l'application.

1.18.2.4. Optimisation du temps de cycle : annuler la scrutation de certaines parties du programme

Seules les cibles acceptant les instructions JSR et RET supportent cette technique.
Des directives de compilation spéciales permettent de valider ou de « dévalider » la scrutation de certaines parties du programme.
Ce sont les folios qui définissent ces portions d'application.
Si une application est décomposée en quatre folios alors chacun d'eux pourra être indépendamment « validé » ou « dévalidé ».
Une directive « #C(condition) » placée sur un folio conditionne la scrutation du folio jusqu'au folio contenant une directive « #R ».

Cette condition doit utiliser la syntaxe définie pour les tests voir le chapitre 1.3. Les tests

Exemple :

Si un folio contient les deux directives :

```
#C(m200=4)
#R
```

Alors tout ce qu'il contient ne sera exécuté que si le mot 200 contient 4.

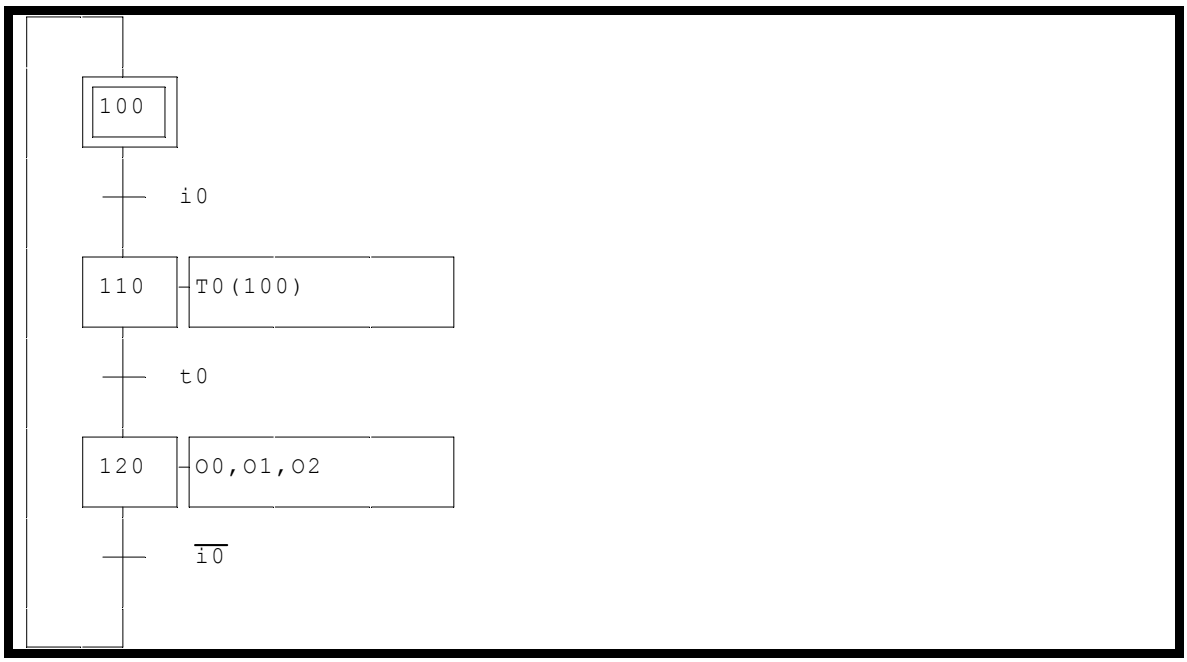
2. Exemples

2.1. A propos des exemples

Cette partie regroupe une série d'exemples donnant une illustration des différentes possibilités de programmation offertes par AUTOMGEN.
Tous ces exemples se trouvent dans le sous répertoire « exemple » du répertoire où a été installé AUTOMGEN.
Cette partie contient également des exemples plus complets et plus complexes développés pour une maquette de train. La description de cette maquette se trouve au début du manuel de référence langage.

2.1.1. Grafcet simple

Ce premier exemple est un Grafcet simple en ligne :



exemple\grafcet\sample1.agn

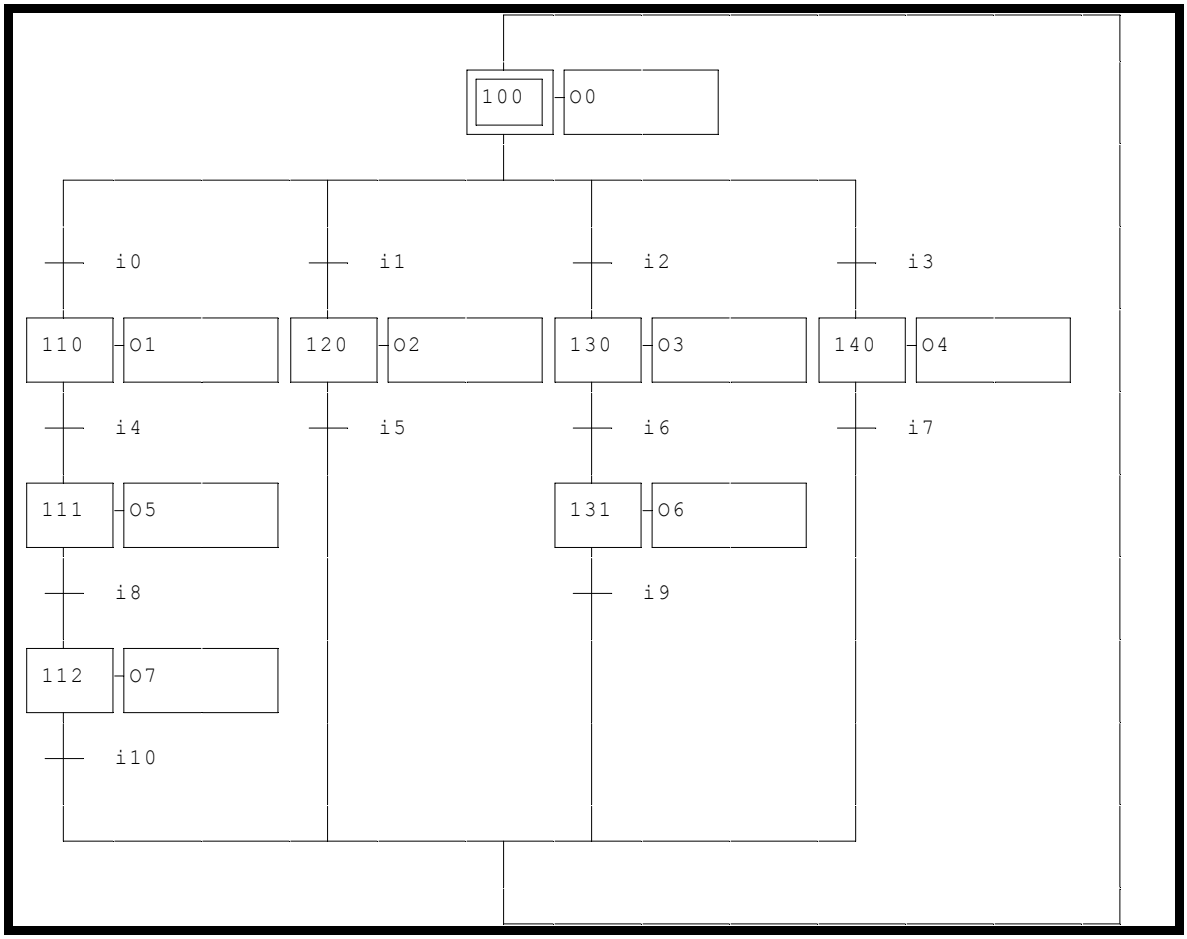
⇨ la transition entre l'étape 100 et l'étape 110 est constituée du test sur l'entrée 0,

⇨ l'étape 110 active la temporisation 0 d'une durée de 10 secondes, cette temporisation est utilisée comme transition entre l'étape 110 et l'étape 120,

⇨ l'étape 120 active les sorties 0, 1 et 2,

⇨ le complément de l'entrée 0 sert de transition entre l'étape 120 et 100.

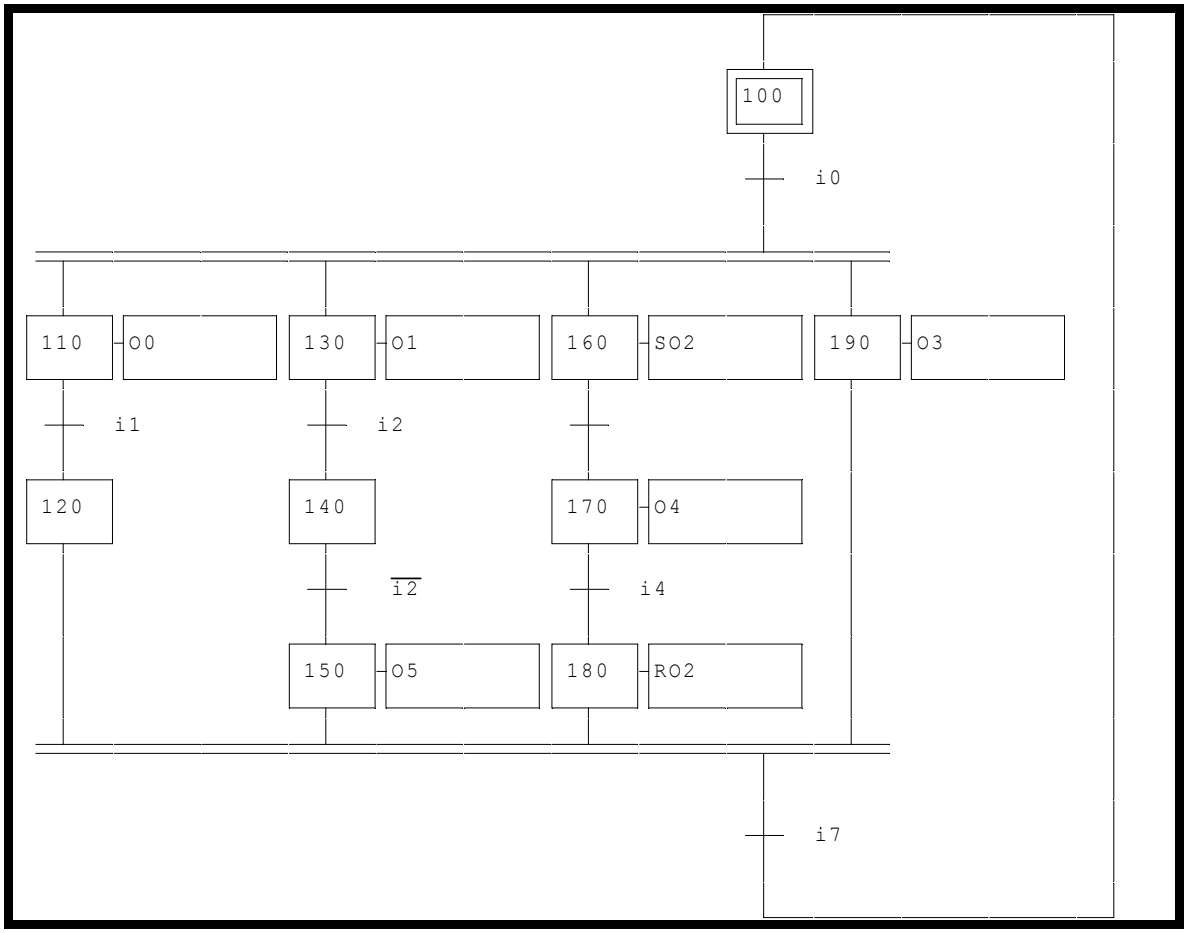
2.1.2. Grafcet avec divergence en OU



exemple\grafcet\sample2.agn

Cet exemple illustre l'utilisation des divergences et convergences en « Ou ». Le nombre de branches n'est limité que par la taille du folio. Il s'agit comme le prévoit la norme, d'un « Ou » non exclusif. Si par exemple, les entrées 1 et 2 sont actives, alors les étapes 120 et 130 seront mises à un.

2.1.3. Grafcet avec divergence en ET

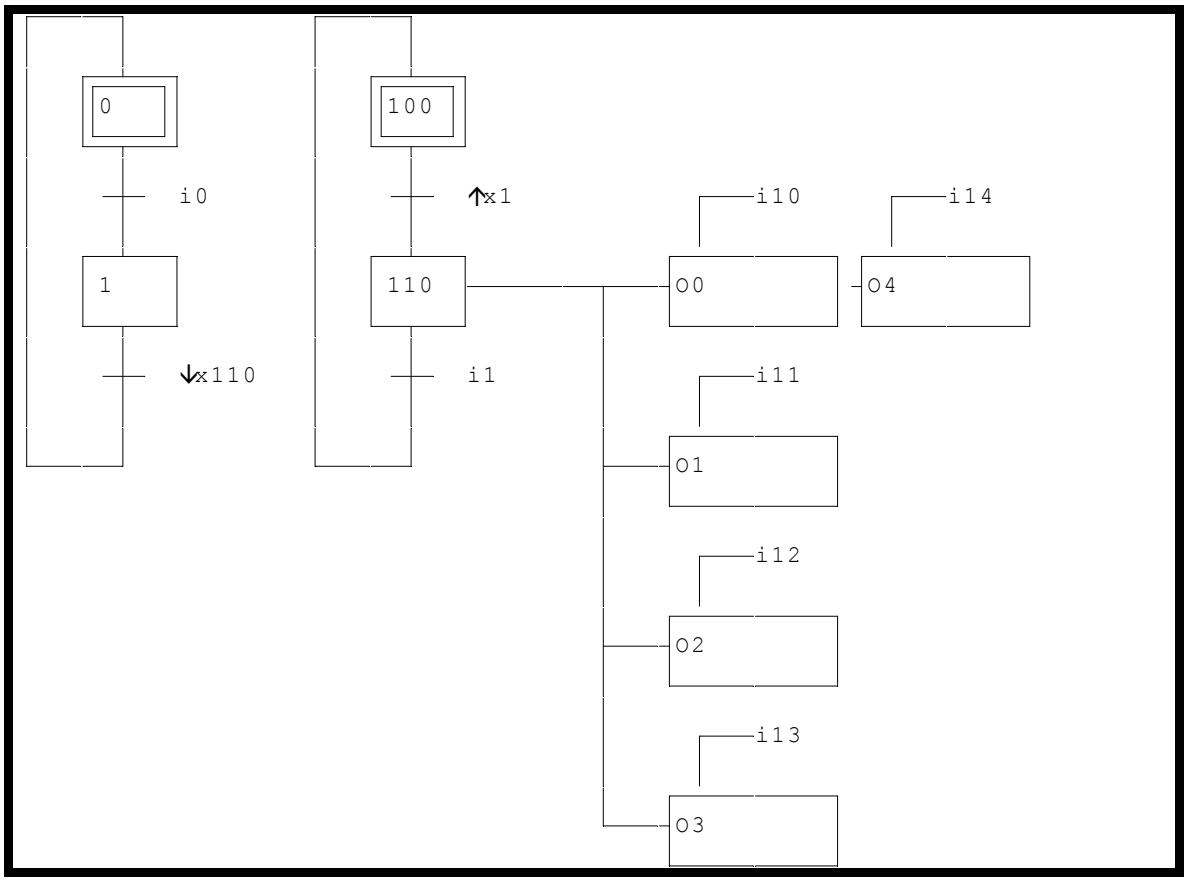


exemple\grafcet\sample3.agn

Cet exemple illustre l'utilisation des divergences et convergences en « Et ». Le nombre de branches n'est limité que par la taille du folio. Notons également au passage les points suivants :

- ⇨ une étape peut ne pas comporter d'action (cas des étapes 100, 120, et 140),
- ⇨ les ordres « S » et « R » ont été utilisés avec la sortie o2 (étapes 160 et 180),
- ⇨ la transition entre l'étape 160 et 170 est laissée blanche, elle est donc toujours vraie, la syntaxe « =1 » aurait pu aussi être utilisée.

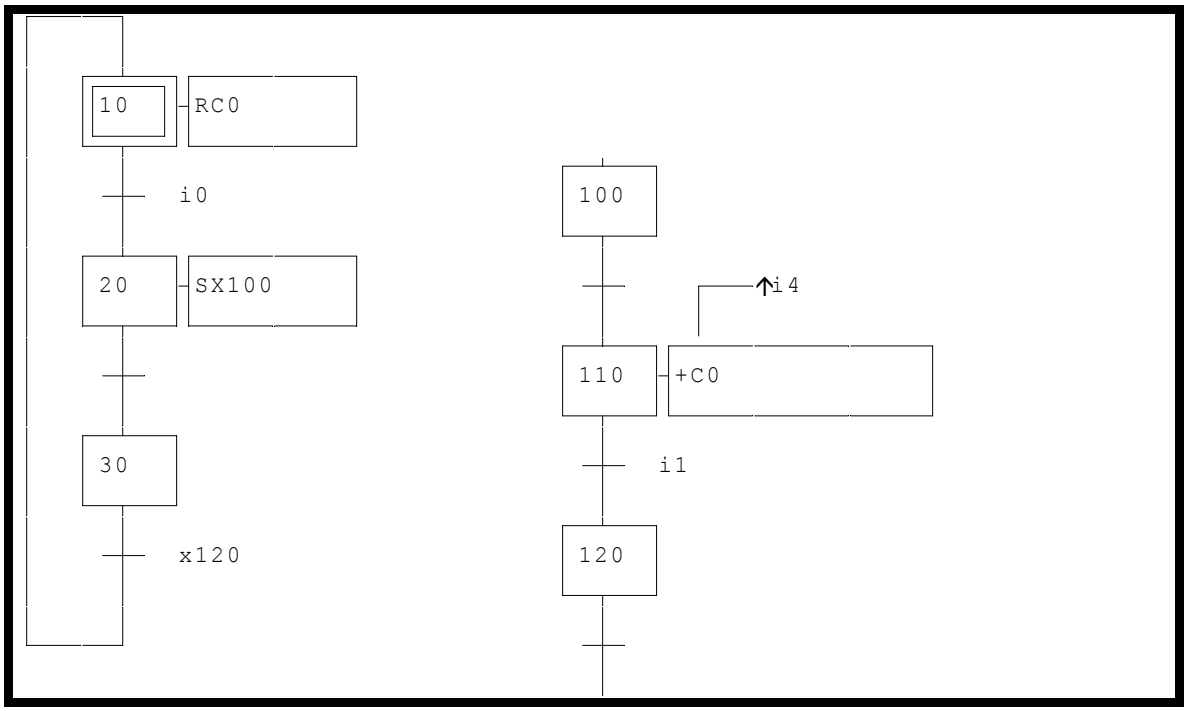
2.1.4. Grafcet et synchronisation



exemple\grafcet\sample4.agn

Cet exemple illustre une des possibilités offertes par AUTOMGEN pour synchroniser plusieurs Grafcets. La transition entre l'étape 100 et 110 « ↑x1 » signifie « attendre un front montant sur l'étape 1 ». La transition « ↓x110 » signifie « attendre un front descendant sur l'étape 110 ». L'exécution pas à pas de ce programme montre l'évolution exacte des variables et de leur front à chaque cycle. Ceci permet de comprendre exactement ce qu'il se passe lors de l'exécution. Notons également l'utilisation d'actions multiples associées à l'étape 110, qui sont ici conditionnées individuellement.

2.1.5. Forçage d'étapes

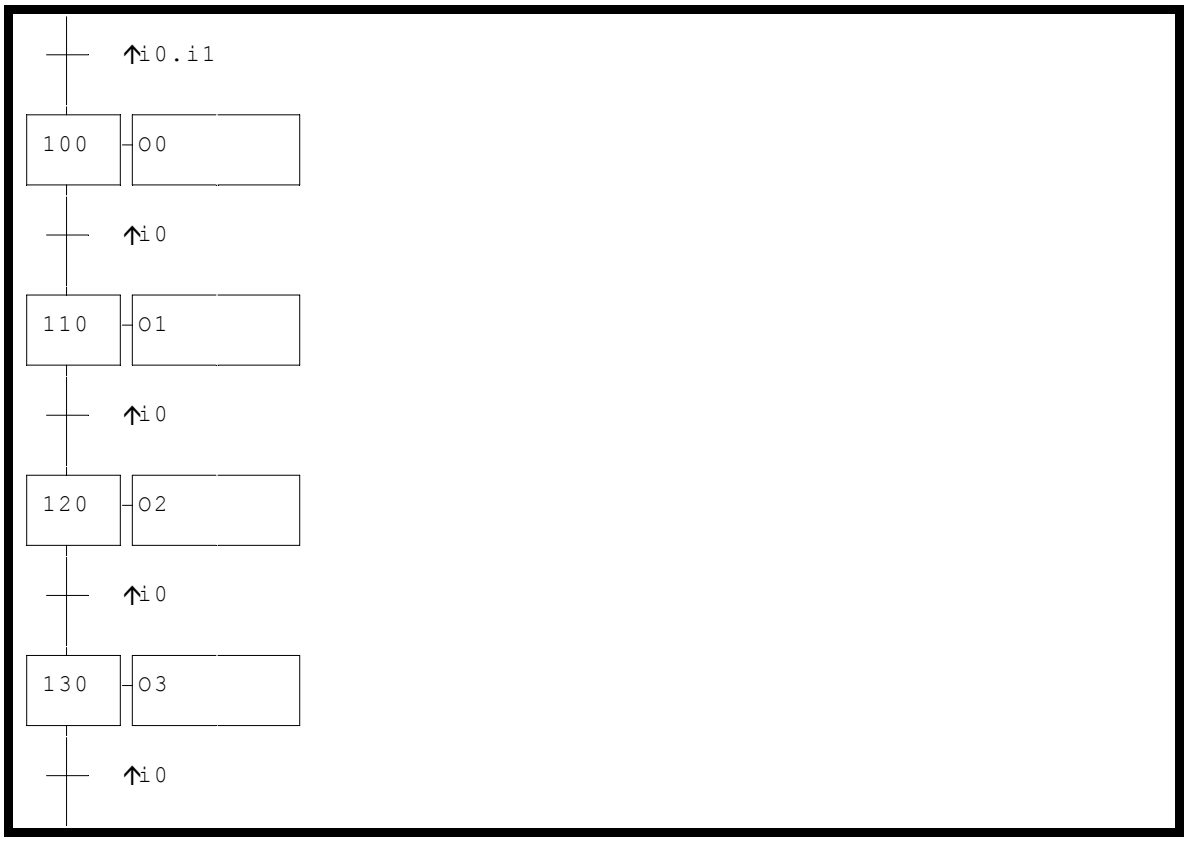


exemple\grafcet\sample5.agn

Dans cet exemple, un ordre « S » (mise à un) a été utilisé pour forcer une étape. AUTOMGEN autorise également le forçage d'un Grafcet entier (voir exemples 8 et 9) Le mode d'exécution pas à pas permet, pour cet exemple aussi, de comprendre de façon précise l'évolution du programme dans le temps. Notons également :

- ⇨ l'utilisation d'un Grafcet non rebouclé (100, 110, 120),
- ⇨ l'utilisation de l'ordre « RC0 » (mise à zéro du compteur 0),
- ⇨ l'utilisation de l'ordre « +C0 » (incrémenter le compteur 0), conditionné par le front montant de l'entrée 4, pour exécuter l'incrémentation du compteur, il faut donc que l'étape 110 soit active et qu'un front montant soit détecté sur l'entrée 4.

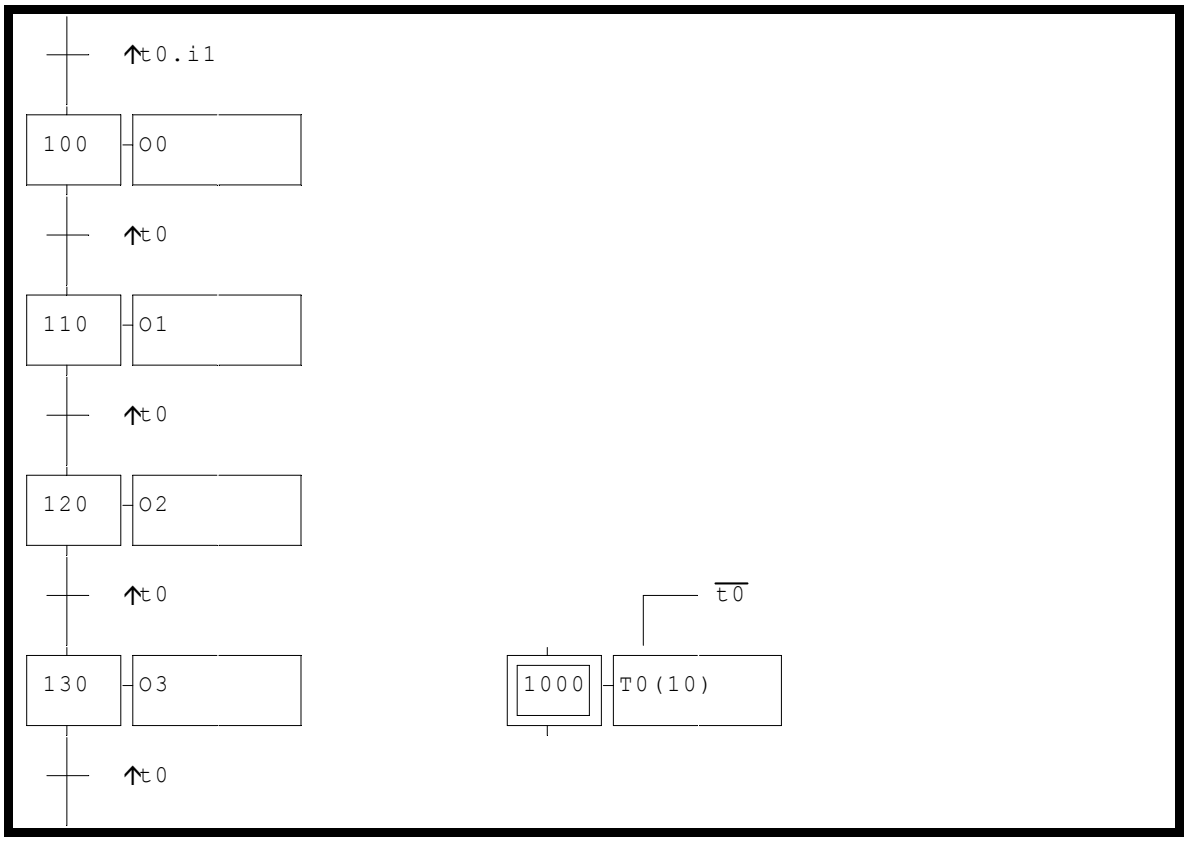
2.1.6. Etapes puits et sources



exemple\grafcet\sample6.agn

Nous avons déjà rencontré des formes similaires, dont la première étape était activée par un autre Grafcet. Ici l'activation de l'étape 100 est réalisée par la transition « ↑i0 . i1 » (front montant de l'entrée 0 et l'entrée 1). Cet exemple représente un registre à décalage. « i1 » est l'information à mémoriser dans le registre et « i0 » est l'horloge qui fait progresser le décalage. L'exemple 7 est une variante qui utilise une temporisation comme horloge.

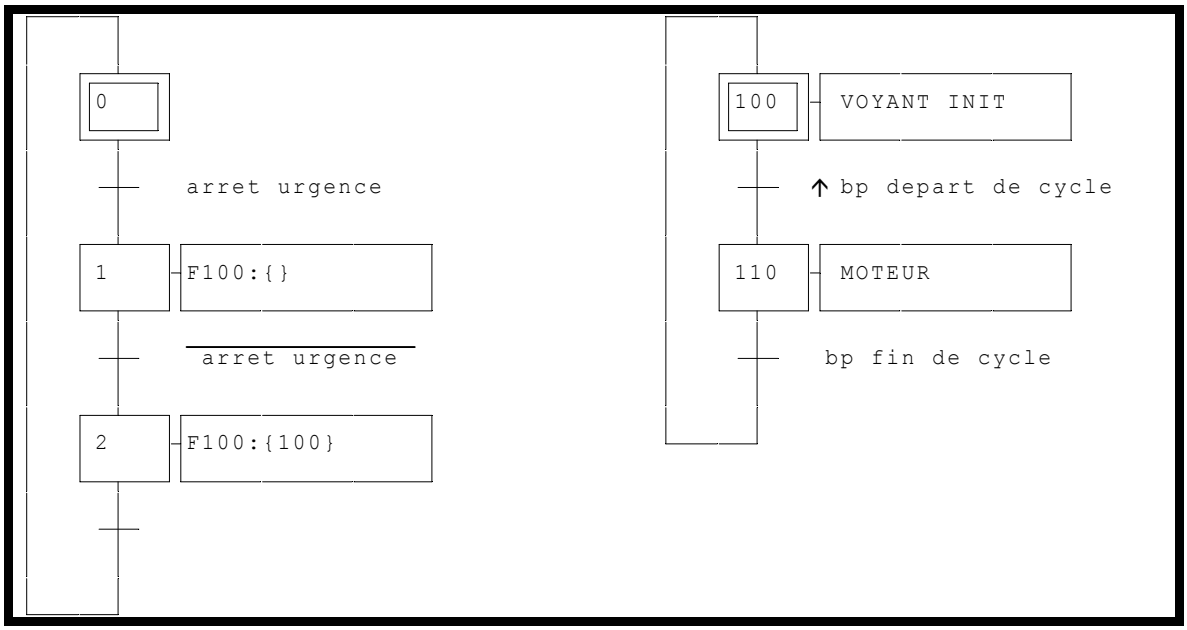
2.1.7. Etapes puits et sources



exemple\grafcet\sample7.agn

Nous retrouvons la structure de registre à décalage utilisé dans l'exemple 6. L'information de décalage est cette fois générée par une temporisation (t0). « ↑t0 » représente le front montant de la temporisation, cette information est vraie pendant un cycle lorsque la temporisation a fini de décompter. L'étape 1000 gère le lancement de la temporisation. On peut résumer l'action de cette étape par la phrase suivante : « activer le décomptage si celui-ci n'est pas terminé, dans le cas contraire, réarmer la temporisation ». Le diagramme de fonctionnement des temporisations de ce manuel vous aidera à comprendre le fonctionnement de ce programme.

2.1.8. Forçage de Grafcets

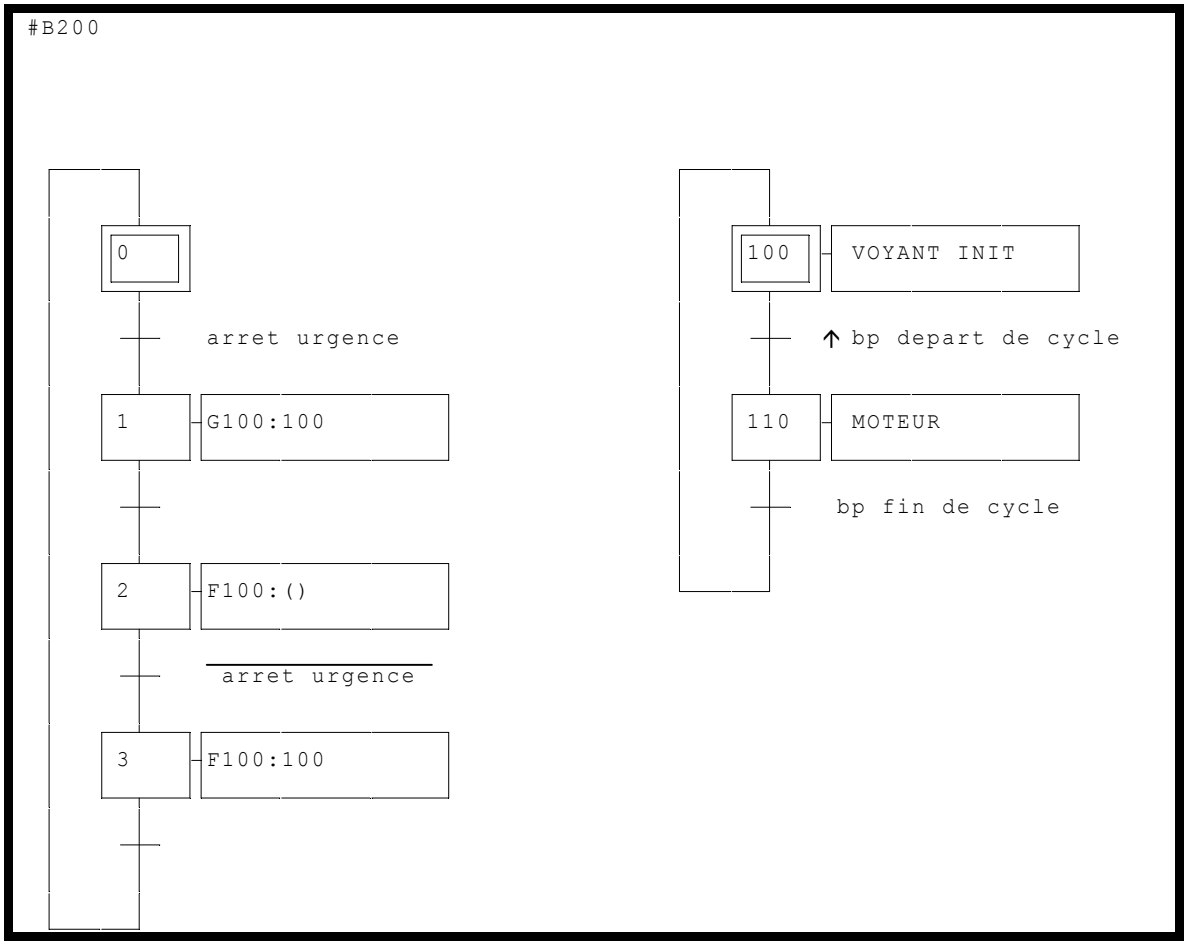


exemple\grafcet\sample8.agn

Cet exemple illustre l'utilisation d'une commande de forçage de Grafcet. L'ordre « F100:{} » signifie « forcer toutes les étapes du Grafcet dont une des étapes porte le numéro 100 à zéro ». L'ordre « F100:{100} » est identique mais force l'étape 100 à 1. Pour cet exemple, nous avons utilisé des symboles :

arret urgence	i0	
bp depart de cycle	i1	
bp fin de cycle	i2	
VOYANT INIT	o0	
MOTEUR	o1	

2.1.9. Mémorisation de Grafcets

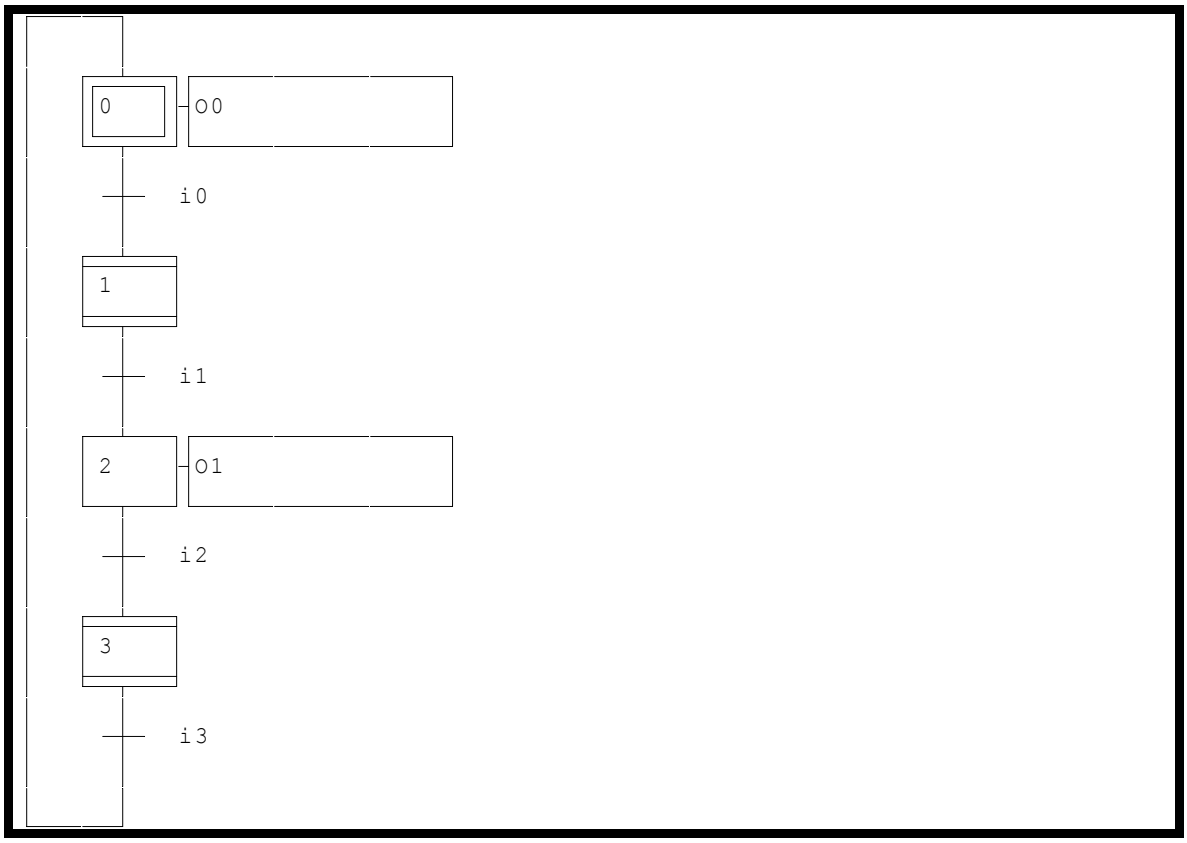


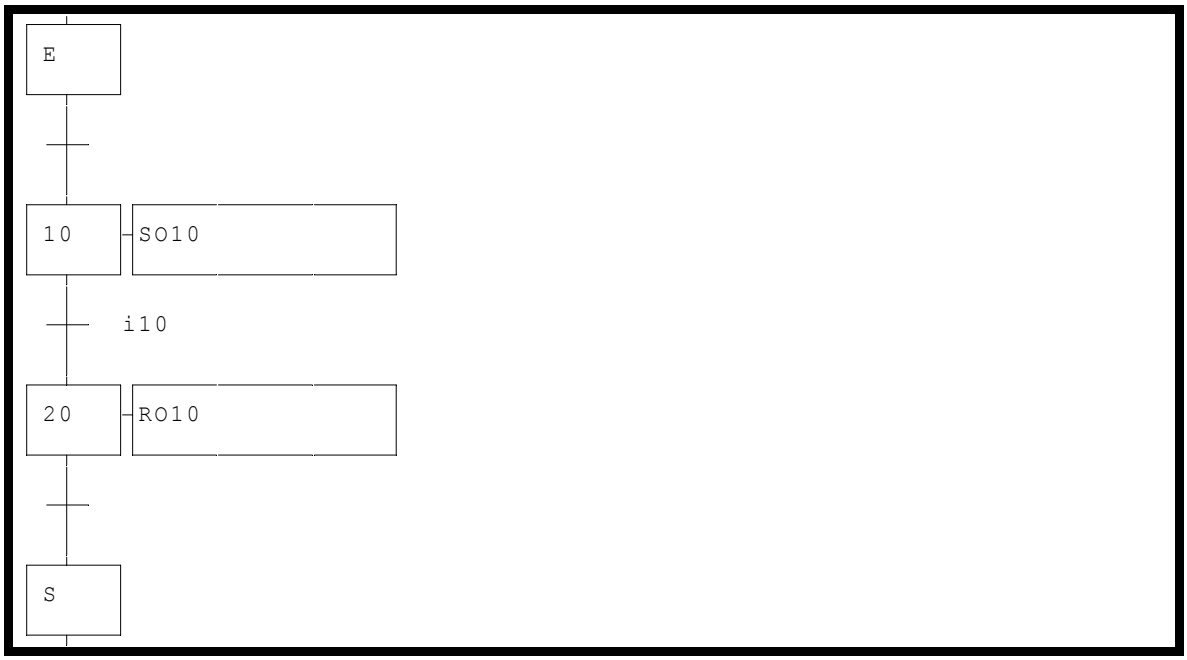
exemple\grafcet\sample9.agn

arret urgence	i0	
bp depart de cycle	i1	
bp fin de cycle	i2	
VOYANT INIT	o0	
MOTEUR	o1	

Cet exemple est une variante du programme précédent. L'ordre « G100:100 » de l'étape 1 mémorise l'état du Grafcet de production avant qu'il ne soit forcé à zéro. A la reprise, le Grafcet de production sera replacé dans l'état où il était avant la coupure, avec l'ordre « F100:100 ». L'état du Grafcet de production est mémorisé à partir du bit 100 (c'est le deuxième paramètre des ordres « F » et « G » qui précise cet emplacement), la directive de compilation « #B200 » réserve les bits u100 à u199 pour ce type d'utilisation. Notons qu'une directive « #B102 » aurait suffit ici puisque le Grafcet de production ne nécessite que deux bits pour être mémorisé (un bit par étape).

2.1.10. Grafcet et macro-étapes

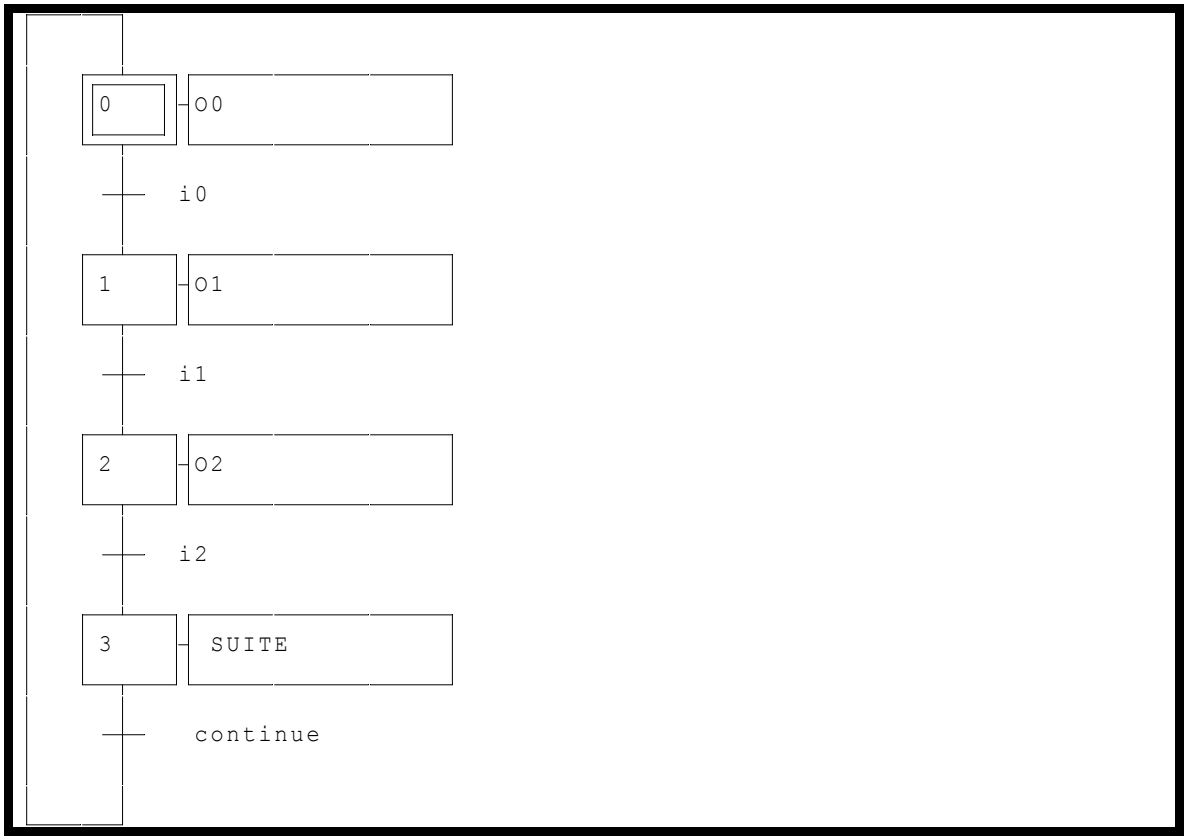


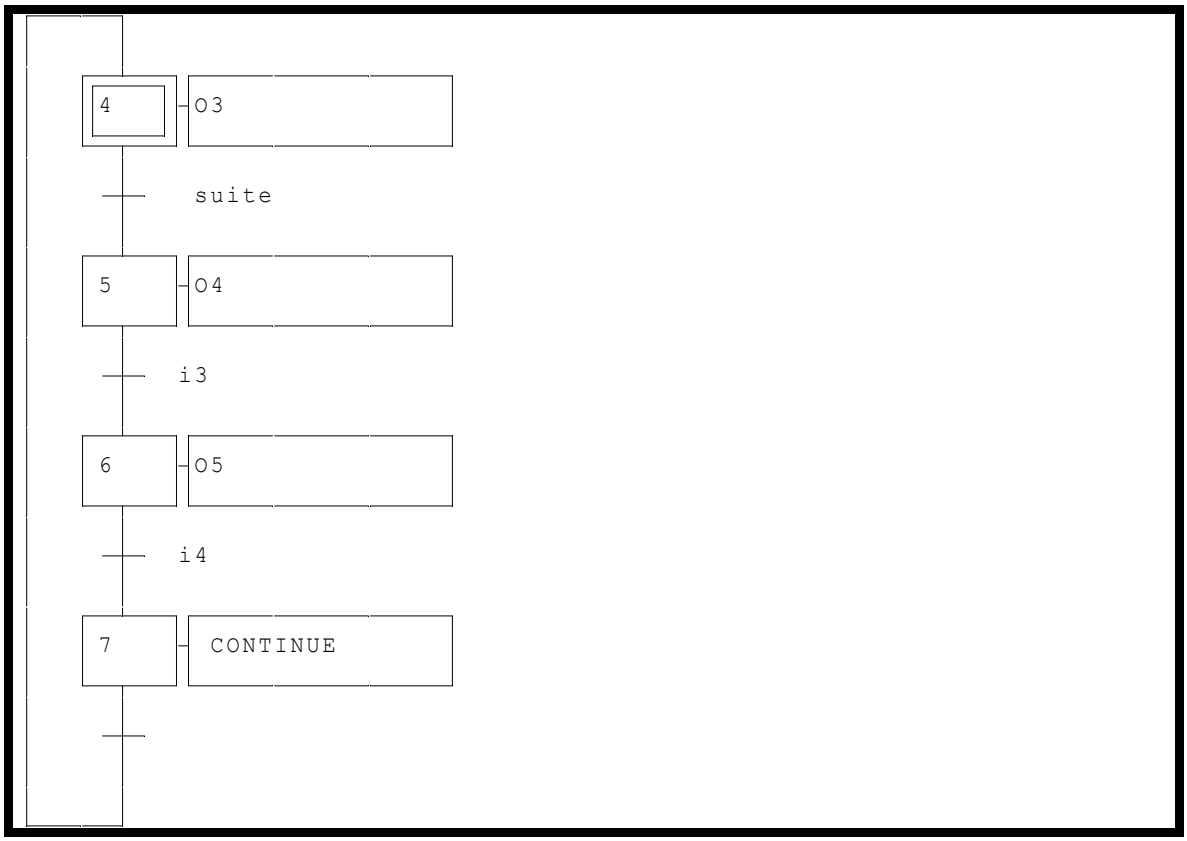


exemple\grafcet\sample11.agn

Cet exemple illustre l'utilisation des macro-étapes. Les folio « Macro étape 1 » et « Macro étape 3 » représentent l'expansion des macro-étapes avec les étapes d'entrées et de sorties. Les étapes 1 et 3 du folio « Programme principal » sont définies comme macro-étapes. L'accès aux expansions de macro-étapes en visualisation est réalisé en cliquant avec le bouton gauche de la souris sur les macro-étapes.

2.1.11. Folios chaînés

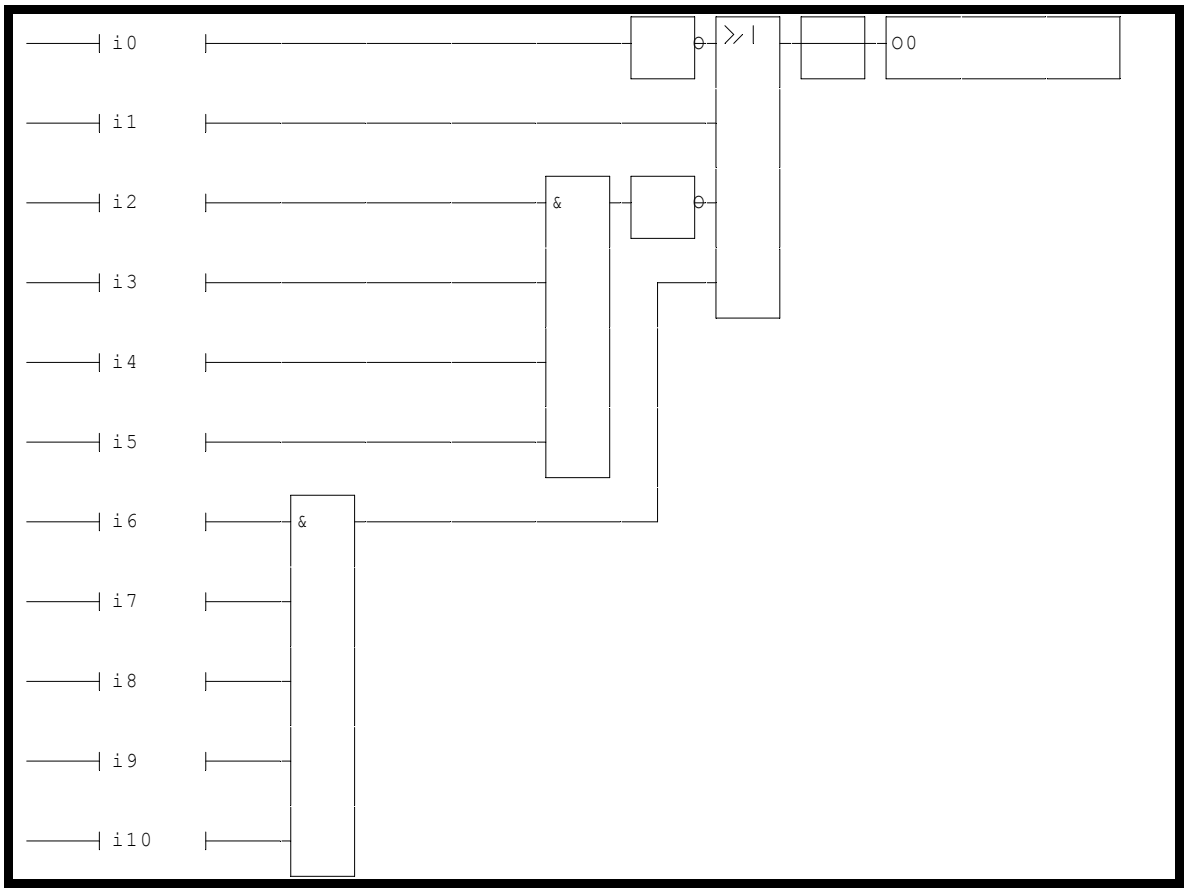




exemple\grafcet\sample12.agn

Dans cet exemple, deux folios ont été utilisés pour écrire le programme. Les symboles « _SUITE_ » et « _CONTINUE_ » ont été déclarés comme bits (voir le fichier des symboles) et permettent de faire le lien entre les deux Grafcets (c'est une autre technique de synchronisation utilisable avec AUTOMGEN).

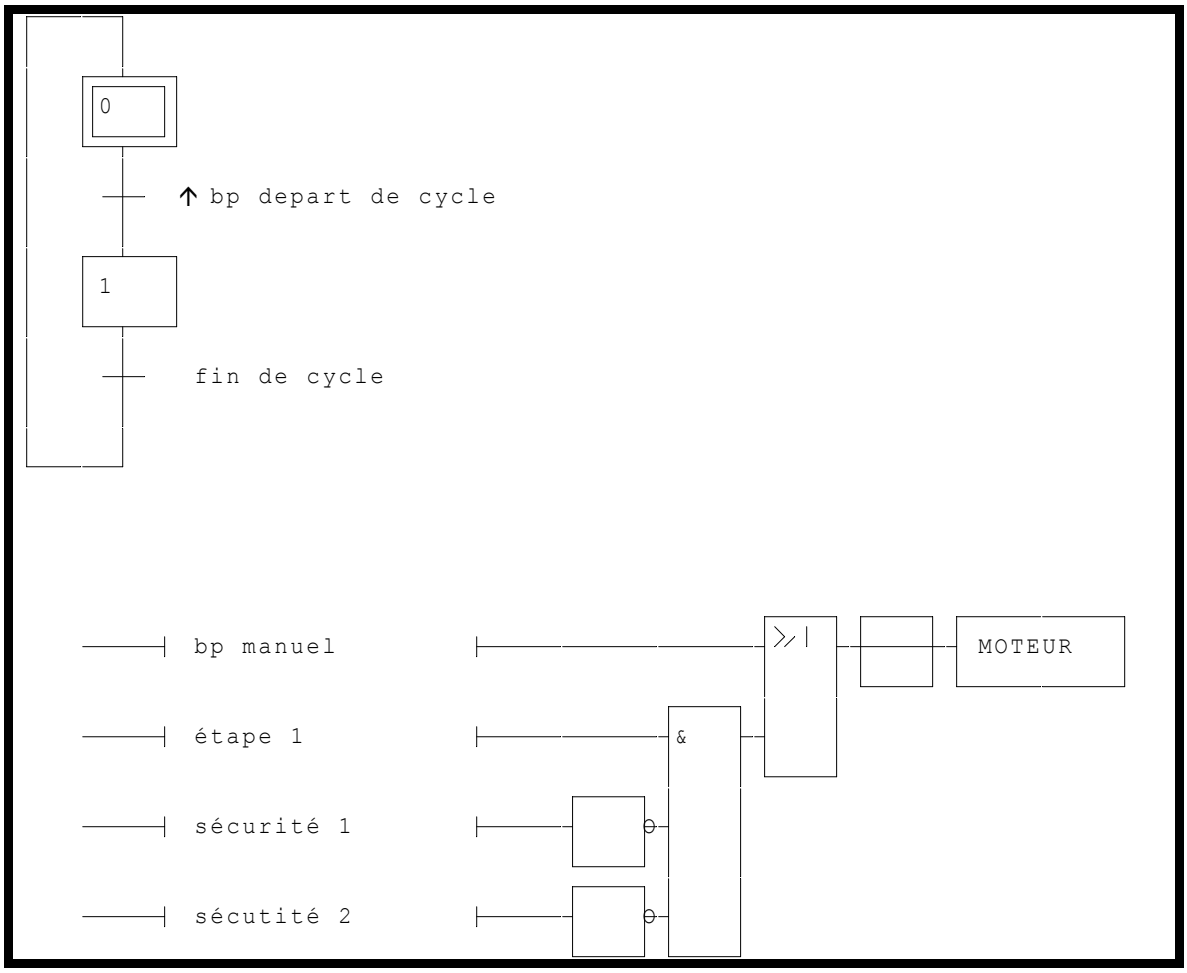
2.1.12. Logigramme



exemple\logigramme\sample14.agn

Cet exemple développé en logigrammes montre l'utilisation des différents blocs : le bloc d'affectation associé à la touche [0] à gauche du rectangle d'action, le bloc « pas » associé à la touche [1] qui complémente un signal, ainsi que les blocs d'ancrage de tests et les fonctions « Et » et « Ou ».

2.1.13. Grafcet et Logigramme

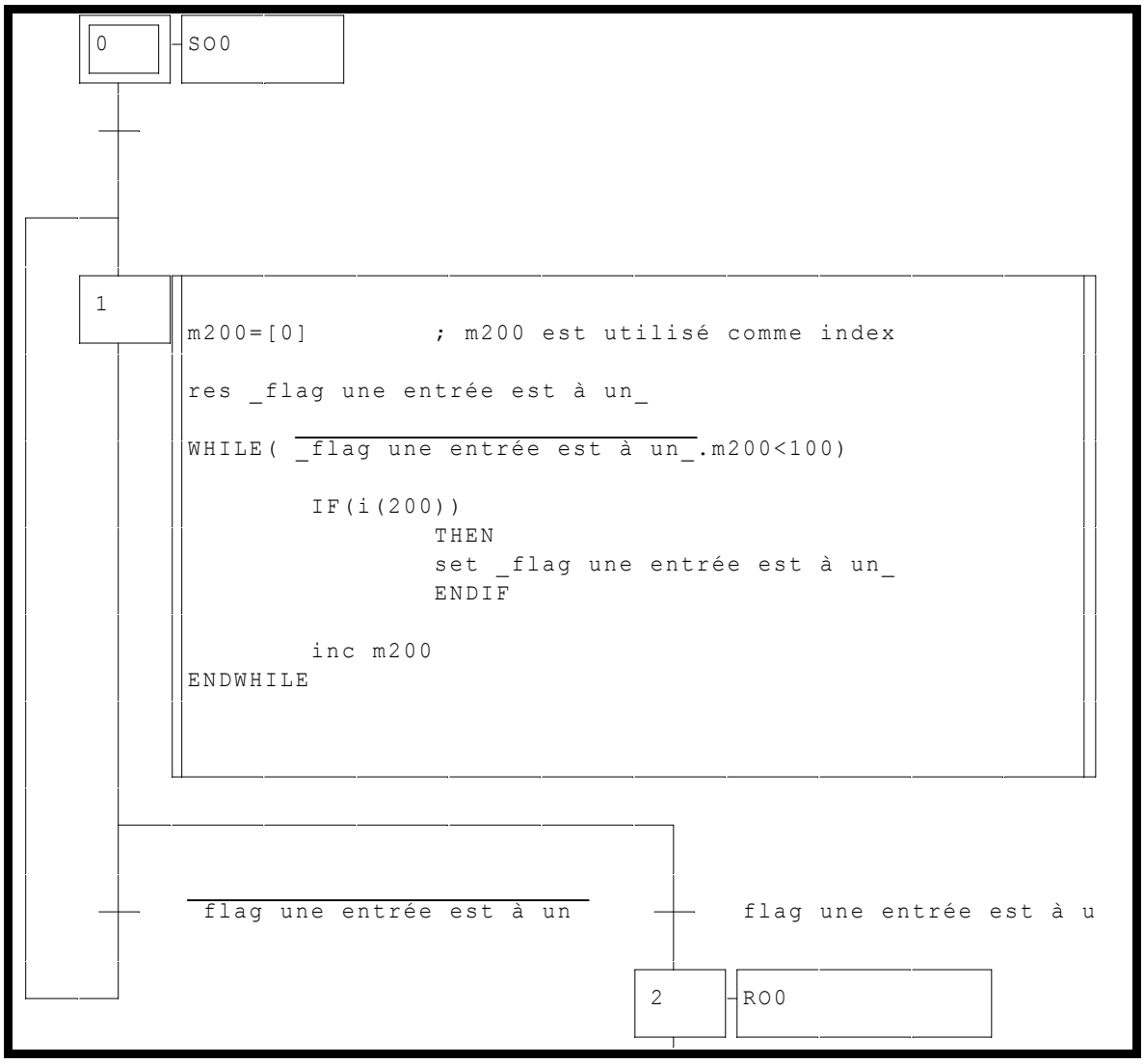


exemple\logigramme\exempl15.agn

Dans cet exemple un Grafcet et un Logigramme sont utilisés conjointement. Le symbole « _étape 1_ » utilisé dans le logigramme est associé à la variable « x1 ».

Ce type de programmation laisse apparaître clairement les conditions d'activation d'une sortie.

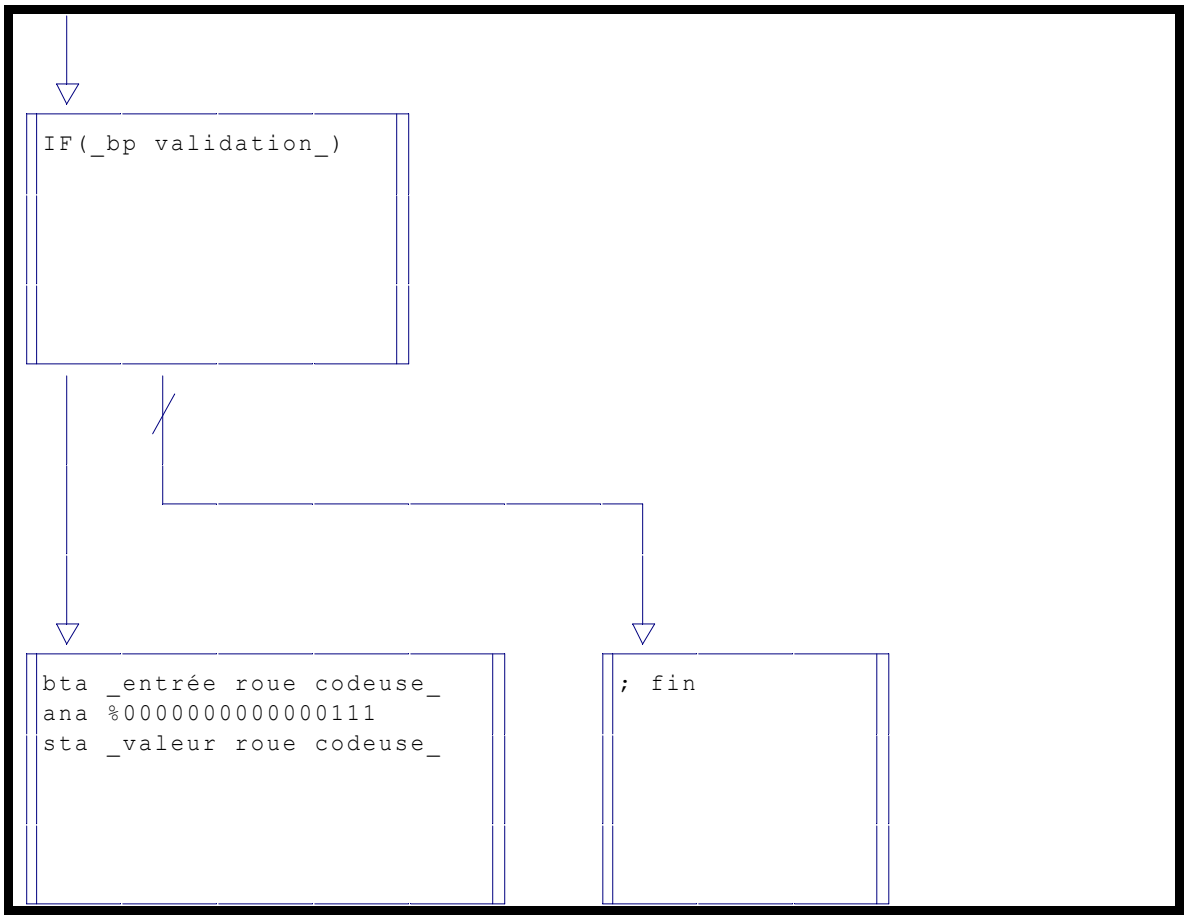
2.1.14. Boîte de langage littéral



exemple\lit\sample16.agn

Ce programme qui associe Grafcet et boîte de langage littéral a pour objet de tester les entrées i0 à i99. Si une de ces entrées est à un, alors l'étape 2 est activée et le Grafcet se retrouve dans un état ou toute évolution est interdite. Le symbole « _flag une entrée est à un_ » est associé au bit u500. Un adressage indexé a été utilisé pour balayer les 100 entrées. Notons également l'emploi simultané du langage littéral bas niveau et étendu.

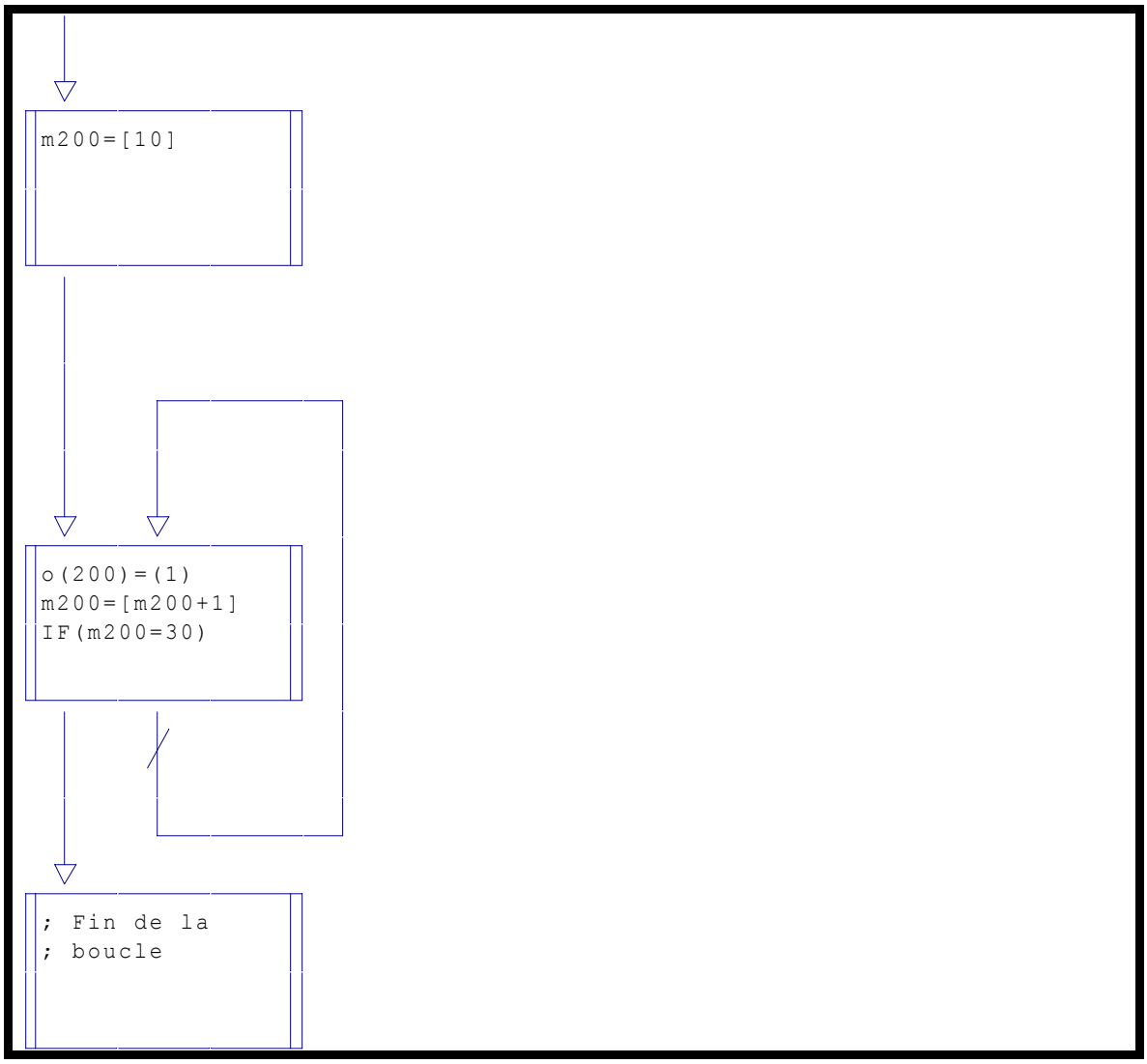
2.1.15. Organigramme



exemple\organigramme\sample18.agn

Cet exemple illustre l'utilisation d'un organigramme pour effectuer un traitement algorithmique et numérique. Ici trois entrées provenant d'une roue codeuse sont lues et stockées dans un mot si une entrée de validation est active.

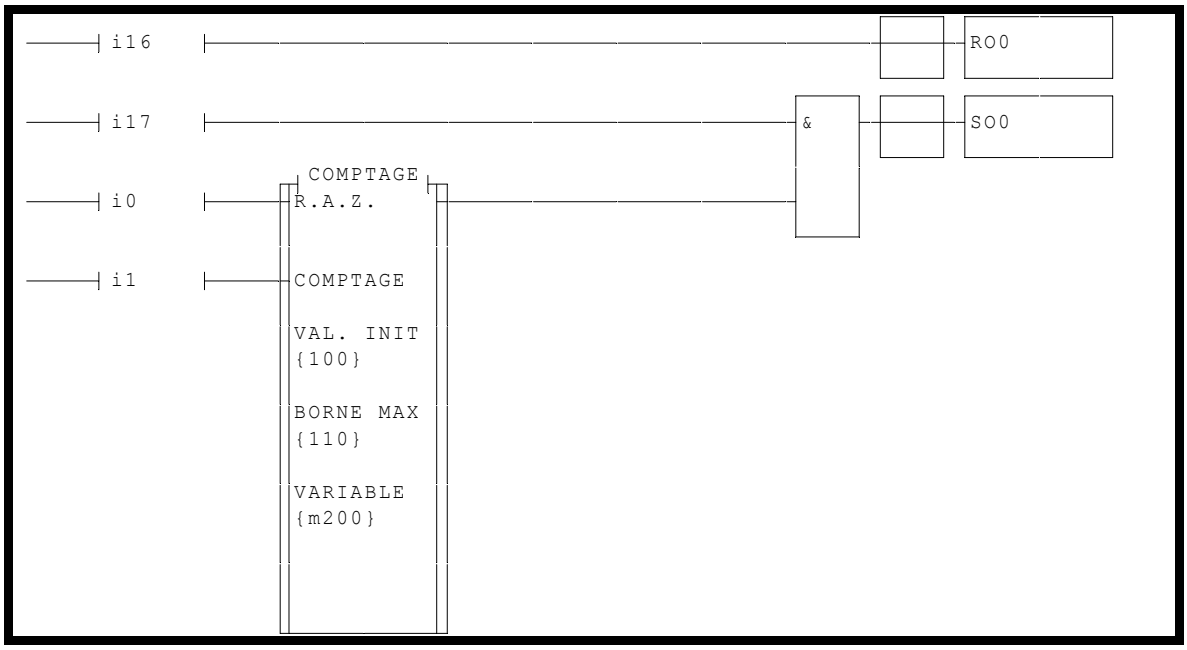
2.1.16. Organigramme



exemple\organigramme\sample19.agn

Ce deuxième exemple d'organigramme réalise une structure de boucle qui permet de forcer à un une série de sorties (o10 à o29) avec un adressage indirect (« o(200) »).

2.1.17. Bloc fonctionnel



exemple\bf\sample20.agn

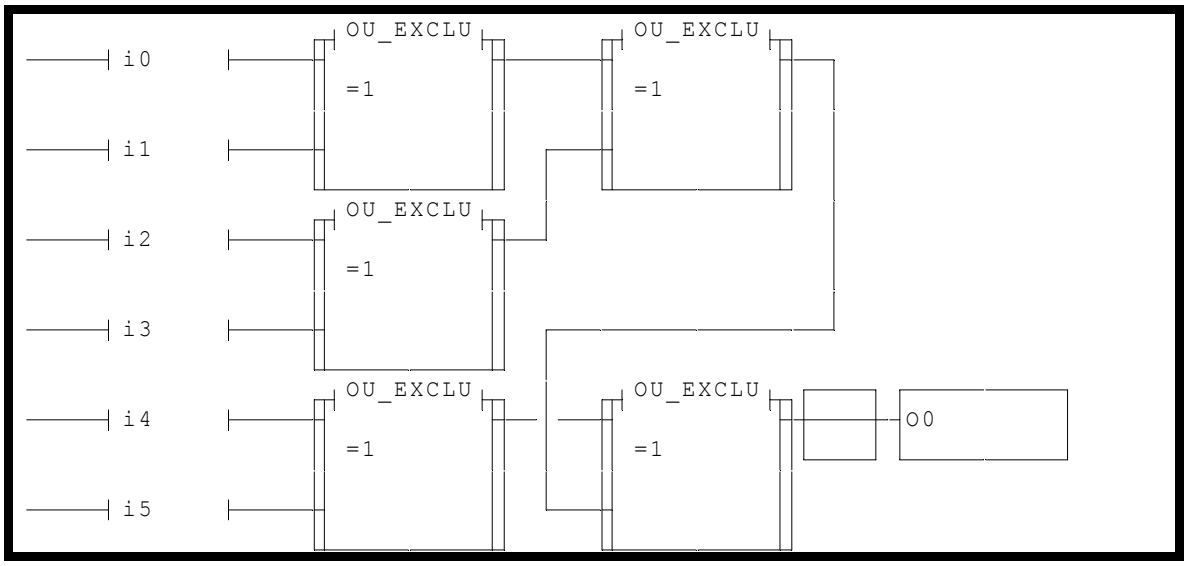
```
; Gestion de l'entrée de RAZ
IF({I0})
        THEN
        {?2}=[{?0}]
        ENDIF

; Gestion de l'entrée de comptage

IF(#{I1})
        THEN
        {?2}=[{?2}+1]
        ENDIF

; Teste la borne maxi


IF({?2}={?1})
        THEN
        {O0}=(1)
        {?2}=[{?0}]
        ENDIF
        ELSE
        {O0}=(0)
        ENDIF
```

comptage.lib (inclus dans les ressources du projet)

Cet exemple illustre l'utilisation d'un bloc fonctionnel. Les fonctions du bloc « COMPTAGE » que nous avons définies ici sont les suivantes :

- ⇨ le comptage se fera en partant d'une valeur d'init et se terminera à une valeur de borne maximale,
- ⇨ lorsque la variable de comptage atteindra la borne maximale elle sera forcée à la valeur d'init et la sortie du bloc passera à un pendant un cycle programme,
- ⇨ le bloc possédera une entrée booléenne de RAZ et une entrée de comptage sur front montant.

2.1.18. Bloc fonctionnel



exemple\bf\sample21.agn

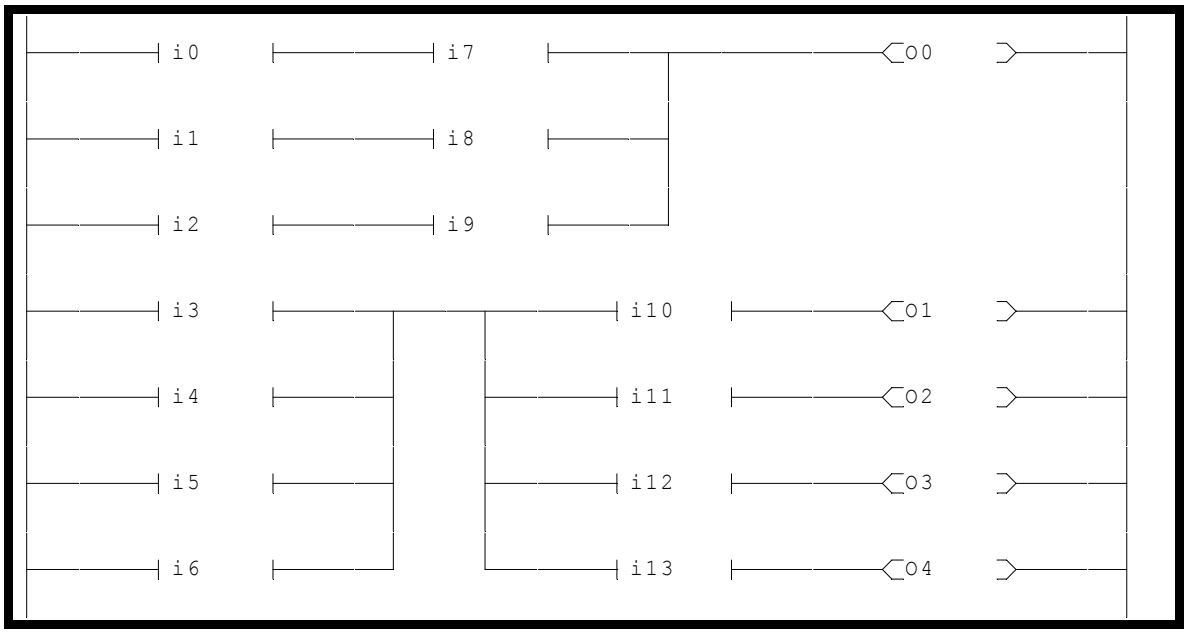
```
; Ou exclusif
neq {o0} orr /{i0} eor {i1} orr {i0} eor /{i1}
```

ou_exclu.lib (inclus dans les ressources du projet)

Ce deuxième exemple de bloc fonctionnel illustre l'utilisation multiple d'un même bloc. Le bloc « OU_EXCLU » réalise un ou exclusif entre les deux entrées booléennes. Cet exemple utilise 5 blocs pour réaliser un ou exclusif entre 6 entrées (i0 à i5). Le fichier « OU_EXCLU.LIB » listé dessous régit le fonctionnement du bloc.

L'équation booléenne du ou exclusif est la suivante : « (i0./i1)+(/i0.i1) ». La forme équivalente utilisée ici permet de coder l'équation sur une seule ligne de langage littéral bas niveau sans utiliser de variables intermédiaires.

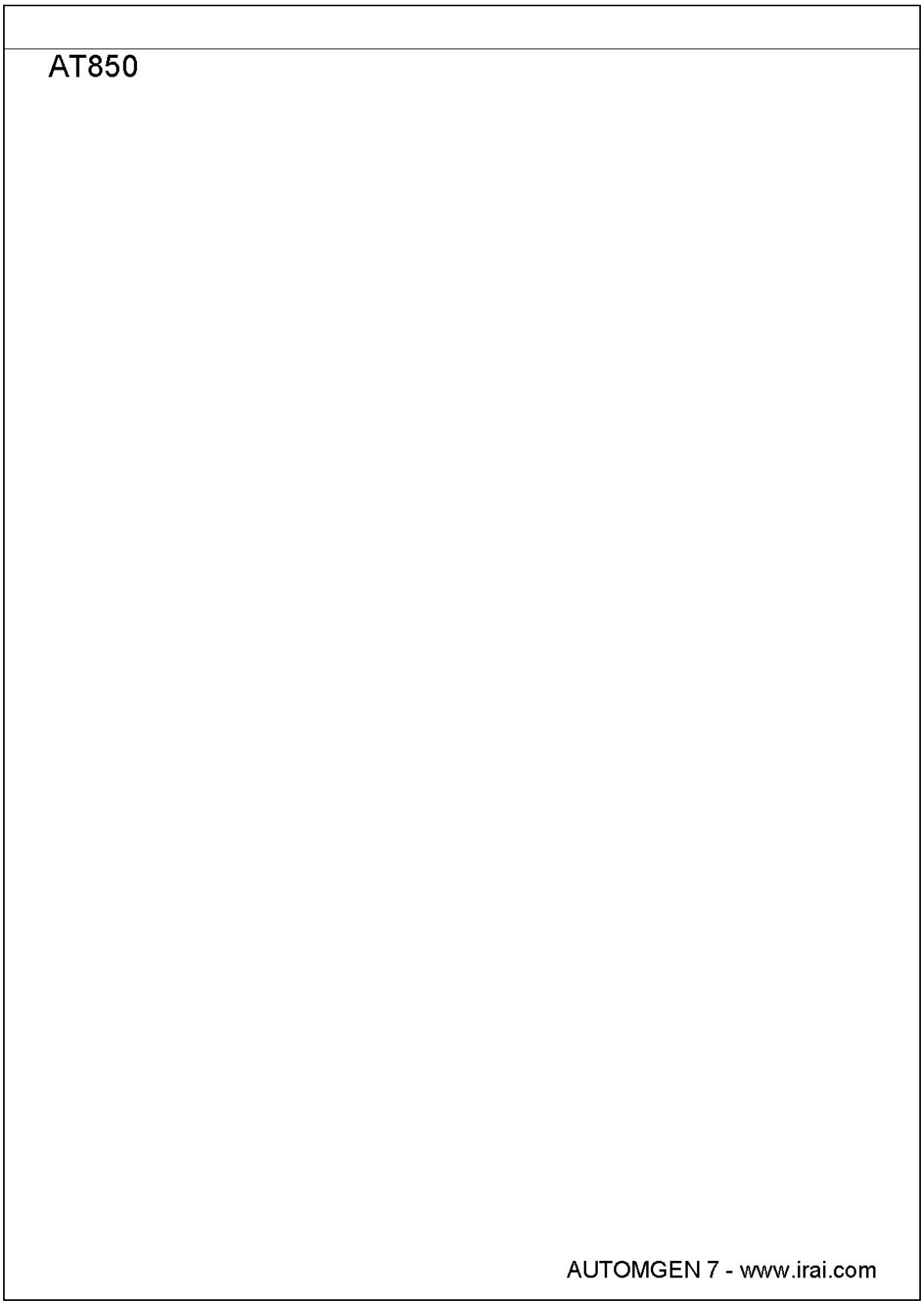
2.1.19. Ladder



exemple\laddersample22.agn

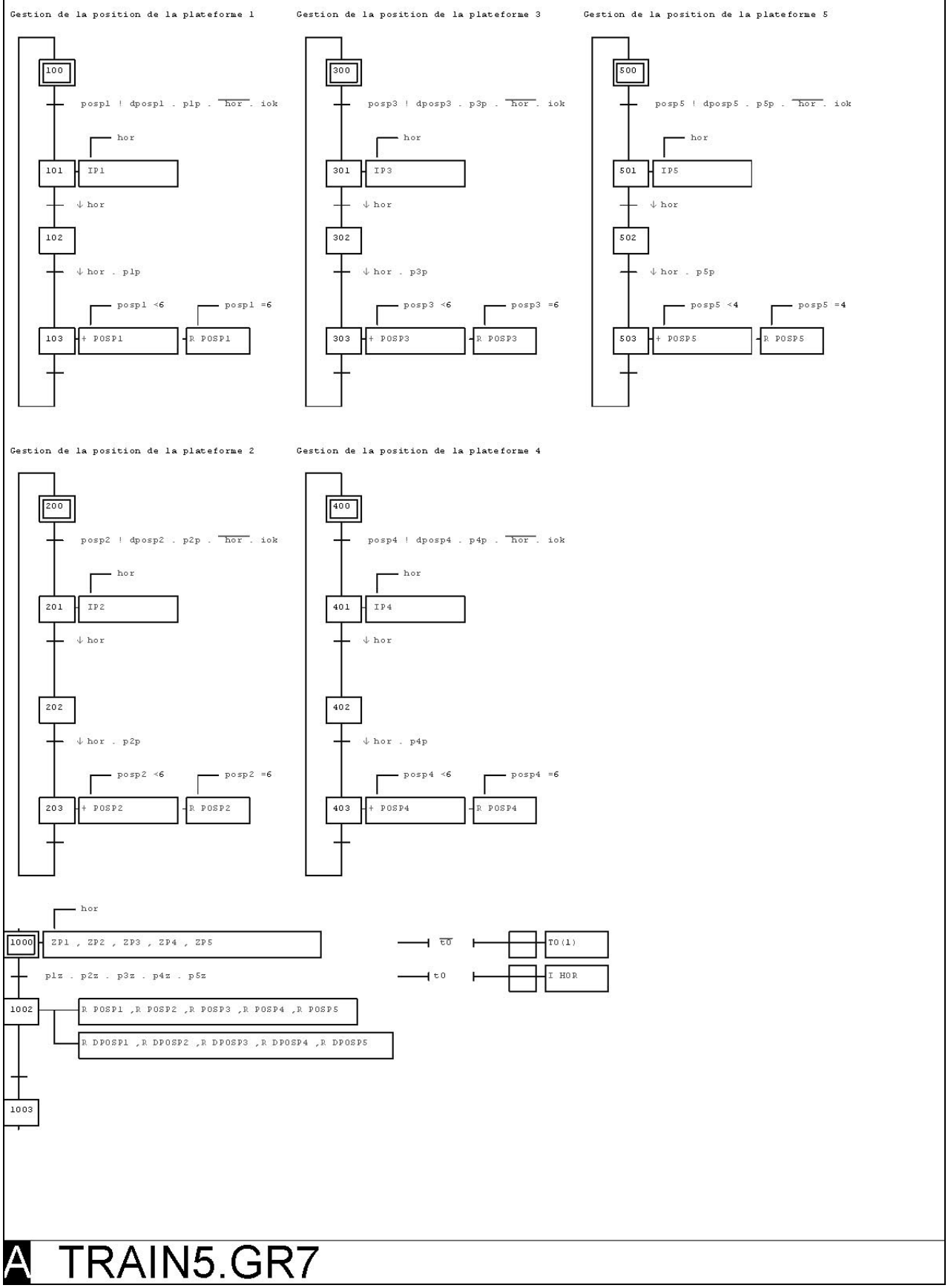
Cet exemple illustre l'utilisation de la programmation en ladder.

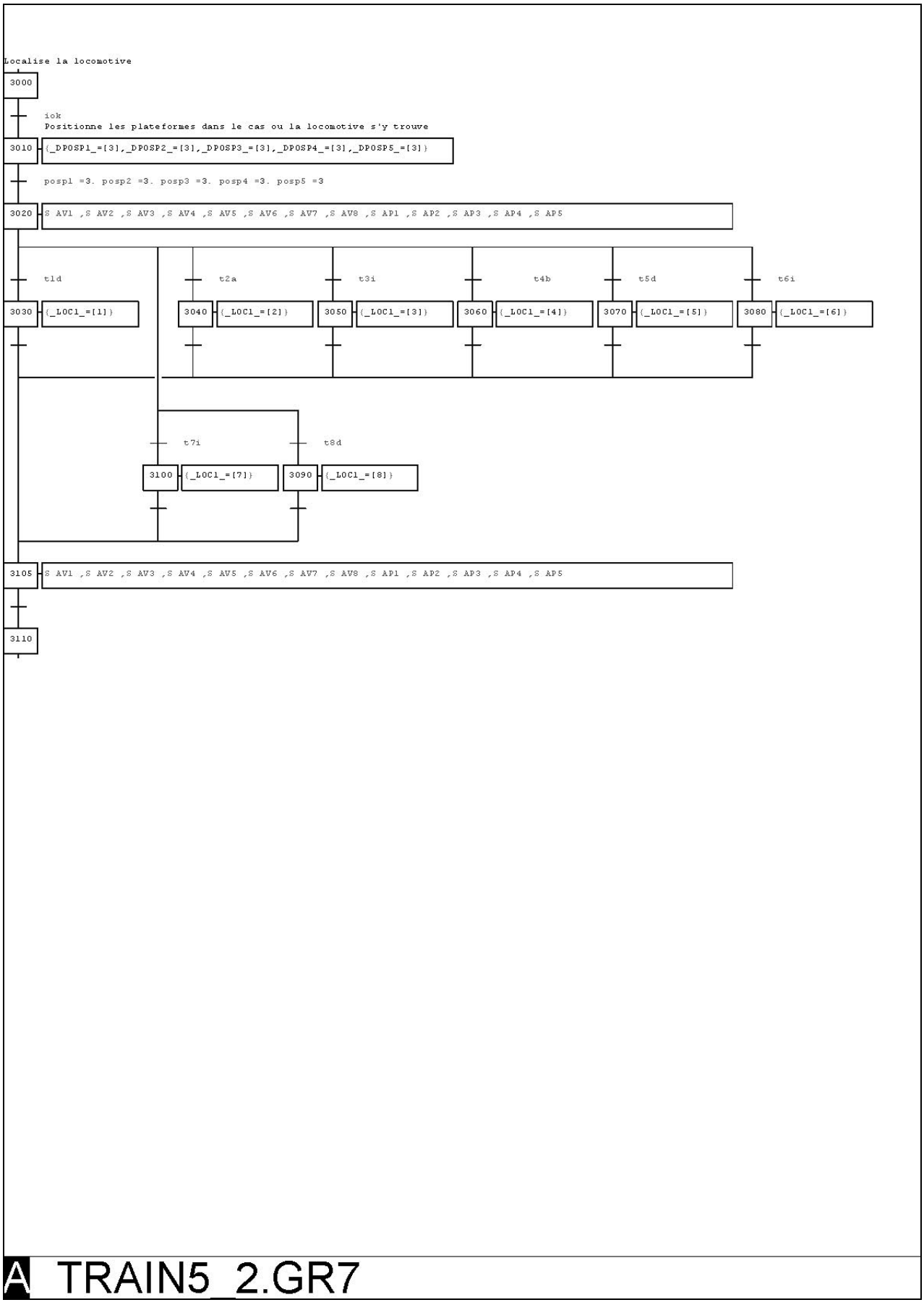
2.1.20. Exemple développé sur une maquette de train



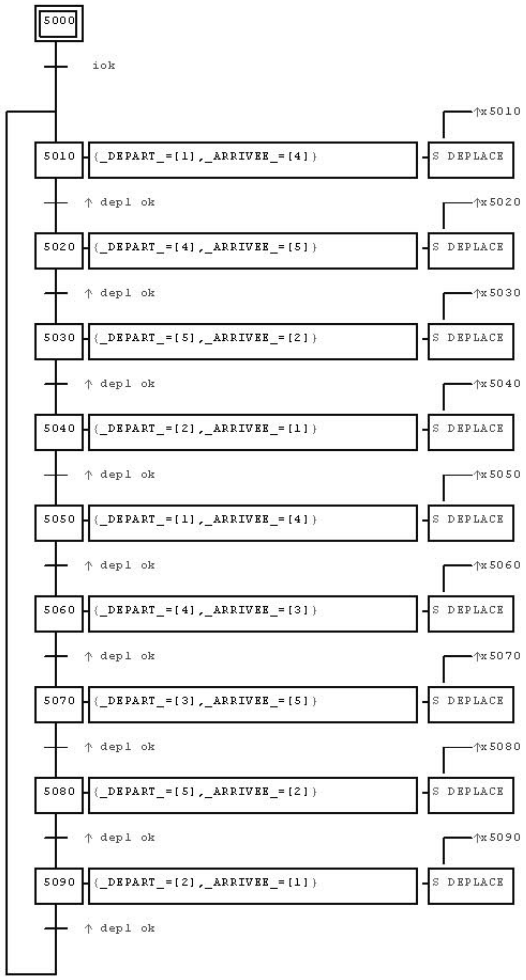
AV1	O0	alimentation voie 1
AV2	O1	alimentation voie 2
AV3	O2	alimentation voie 3
AV4	O3	alimentation voie 4
AV5	O4	alimentation voie 5
AV6	O5	alimentation voie 6
AV7	O6	alimentation voie 7
AV8	O7	alimentation voie 8
AP1	O8	alimentation plateforme 1
AP2	O9	alimentation plateforme 2
AP3	O10	alimentation plateforme 3
AP4	O11	alimentation plateforme 4
AP5	O12	alimentation plateforme 5
IP1	O13	rotation plateforme 1
IP2	O14	rotation plateforme 2
IP3	O15	rotation plateforme 3
IP4	O16	rotation plateforme 4
IP5	O17	rotation plateforme 5
ZP1	O18	initialisation plateforme 1
ZP2	O19	initialisation plateforme 2
ZP3	O20	initialisation plateforme 3
ZP4	O21	initialisation plateforme 4
ZP5	O22	initialisation plateforme 5
DV1	O23	direction voie 1
DV2	O24	direction voie 2
DV3	O25	direction voie 3
DV4	O26	direction voie 4
DV5	O27	direction voie 5
DV6	O28	direction voie 6
DV7	O29	direction voie 7
DV8	O30	direction voie 8
S1D	O31	feu droit voie 1
S1I	O32	feu gauche voie 1
S2A	O33	feu haut voie 2
S2B	O34	feu bas voie 2
S3D	O35	feu droit voie 3
S3I	O36	feu gauche voie 3
S4A	O37	feu haut voie 4
S4B	O38	feu bas voie 4
S5D	O39	feu droit voie 5
S5I	O40	feu gauche voie 5
S6D	O41	feu droit voie 6
S6I	O42	feu gauche voie 6
S7D	O43	feu droit voie 7
S7I	O44	feu gauche voie 7
S8D	O45	feu droit voie 8
S8I	O46	feu gauche voie 8
T1D	i0	train droit voie 1
T1I	i1	train gauche voie 1
T2A	i2	train haut voie 2
T2B	i3	train bas voie 2
T3D	i4	train droit voie 3
T3I	i5	train gauche voie 3
T4A	i6	train haut voie 4
T4B	i7	train bas voie 4
T5D	i8	train droit voie 5

1/1 Symboles


Gestion de la position de la plateforme 1
100
posp1 ! dposp1 . p1p . hor . iok
hor
101
IP1
↓ hor
102
↓ hor . p1p
posp1 <6
posp1 =6
103
+ POSP1
R POSP1
Gestion de la position de la plateforme 3
300
posp3 ! dposp3 . p3p . hor . iok
hor
301
IP3
↓ hor
302
↓ hor . p3p
posp3 <6
posp3 =6
303
+ POSP3
R POSP3
Gestion de la position de la plateforme 5
500
posp5 ! dposp5 . p5p . hor . iok
hor
501
IP5
↓ hor
502
↓ hor . p5p
posp5 <4
posp5 =4
503
+ POSP5
R POSP5
Gestion de la position de la plateforme 2
200
posp2 ! dposp2 . p2p . hor . iok
hor
201
IP2
↓ hor
202
↓ hor . p2p
posp2 <6
posp2 =6
203
+ POSP2
R POSP2
Gestion de la position de la plateforme 4
400
posp4 ! dposp4 . p4p . hor . iok
hor
401
IP4
↓ hor
402
↓ hor . p4p
posp4 <6
posp4 =6
403
+ POSP4
R POSP4
hor
1000
ZP1 , ZP2 , ZP3 , ZP4 , ZP5
t0
T0(1)
p1z . p2z . p3z . p4z . p5z
t0
I HOR
1002
R POSP1 ,R POSP2 ,R POSP3 ,R POSP4 ,R POSP5
R DPOSP1 ,R DPOSP2 ,R DPOSP3 ,R DPOSP4 ,R DPOSP5
1003
A TRAIN5.GR7


Localise la locomotive
3000
iok
Positionne les plateformes dans le cas ou la locomotive s'y trouve
3010
{_DPOSP1_=[3],_DPOSP2_=[3],_DPOSP3_=[3],_DPOSP4_=[3],_DPOSP5_=[3]}
posp1 =3. posp2 =3. posp3 =3. posp4 =3. posp5 =3
3020
S AV1 ,S AV2 ,S AV3 ,S AV4 ,S AV5 ,S AV6 ,S AV7 ,S AV8 ,S AP1 ,S AP2 ,S AP3 ,S AP4 ,S AP5
t1d
t2a
t3i
t4b
t5d
t6i
3030
{_LOC1_=[1]}
3040
{_LOC1_=[2]}
3050
{_LOC1_=[3]}
3060
{_LOC1_=[4]}
3070
{_LOC1_=[5]}
3080
{_LOC1_=[6]}
t7i
t8d
3100
{_LOC1_=[7]}
3090
{_LOC1_=[8]}
3105
S AV1 ,S AV2 ,S AV3 ,S AV4 ,S AV5 ,S AV6 ,S AV7 ,S AV8 ,S AP1 ,S AP2 ,S AP3 ,S AP4 ,S AP5
3110
A TRAIN5_2.GR7

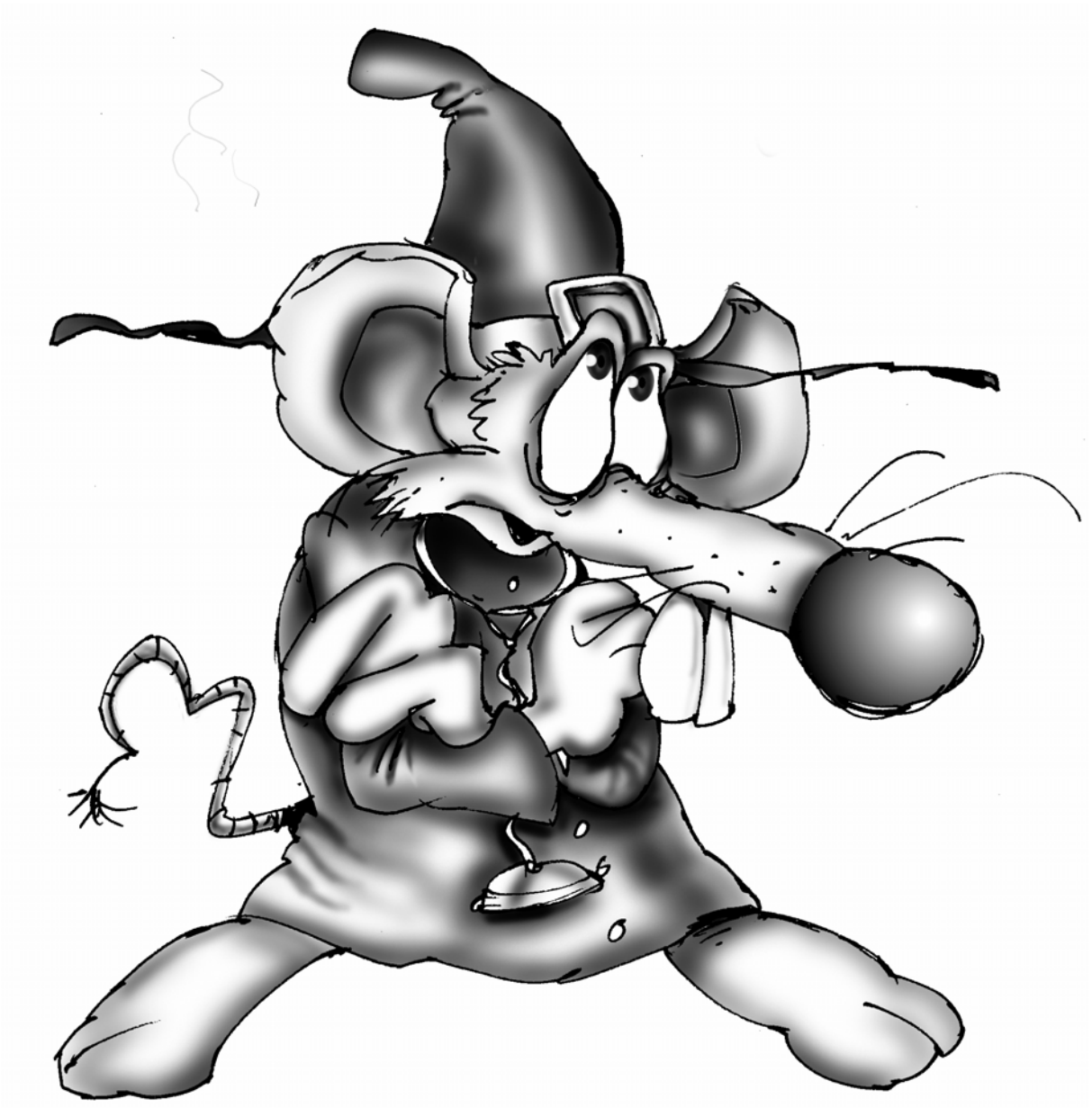

sous-programme : déplace la locomotive de la plateforme (_depart_) à la plateforme (_arrivee_) déplacement élémentaire
#$_rotation plateforme_=0,?_dposp1_,?_dposp2_,?_dposp3_,?_dposp4_,?_dposp5_
#$_alimentation plateforme_=0,?_ap1_,?_ap2_,?_ap3_,?_ap4_,?_ap5_
#$_direction depart_=1,4,3,2,1, 4,3,3,2,4, 3,2,3,2,3, 2,1,3,2,2, 5,1,0,1,6, 5,2,3,1,7, 4,5,4,3,5, 3,5,4,0,8, 0,0,0,0,0
#$_alimentation voie_=0,?_av1_,?_av2_,?_av3_,?_av4_,?_av5_,?_av6_,?_av7_,?_av8_
#$_presence plateforme_=0,?_tp1_,?_tp2_,?_tp3_,?_tp4_,?_tp5_
#$_presence voie_=0,?_t1d_,?_t2a_,?_t3i_,?_t4b_,?_t5i_,?_t6i_,?_t7i_,?_t8I_
↑ deplace
600
(_INDEX_=[?_DIRECTION DEPART_]),R DEPLACE ,R DEPL OK ,R DEPL ERR
depart ! arrivee
601
(_TEST DEPART_=[M(_INDEX_)],INC _INDEX_)
(_TEST ARRIVEE_=[M(_INDEX_)],INC _INDEX_)
(_DIRECTION_=[M(_INDEX_)],INC _INDEX_)
(_DIRECTION2_=[M(_INDEX_)],INC_INDEX_)
(_VOIE_=[M(_INDEX_)],INC _INDEX_)
voie !0. depart = test depart . arrivee = test arrivee
voie !0.(depart ! test depart + arrivee ! test arrivee)
602
index2=[_depart_+?_rotation plateforme_]
index=[m(_index2_)]
m(_index_)=[_direction_]
dec _index_
pos depart=[m(_index_)]
index2=[_arrivee_+?_rotation plateforme_]
index=[m(_index2_)]
m(_index_)=[_direction2_]
dec _index_
pos arrivee=[m(_index_)]
pos depart = direction
603
index2=[_arrivee_+?_presence plateforme_]
index3=[M(_index2_)]
index2=[_voie_+?_presence voie_]
fin voie=[M(_index2_)]
index2=[_depart_+?_alimentation plateforme_]
alimp1=[M(_index2_)]
index2=[_arrivee_+?_alimentation plateforme_]
alimp2=[M(_index2_)]
index2=[_voie_+?_alimentation voie_]
alimv=[M(_index2_)]
?(I(_index3_)) ; Train sur la plateforme d'arrivee ?
->(
set _depl ok_
res 0(_alimp1_)
res 0(_alimp2_)
res 0(_alimv_)
)<-
->/(
set 0(_alimp1_)
set 0(_alimp2_)
pos arrivee=[m(_index_)]
equ 0(_alimv_) orr I(_fin voie_) eor _pos arrivee_=_direction2_
)<-
depl ok
voie =0
604
S DEPL ERR
A TRAIN5_3.GR7



A TRAIN5_4.GR7

Les aventures de Docteur R.

Manuel pédagogique à l'usage des utilisateurs d'AUTOMGEN



Dessins par TABAN

Distribution

Docteur R. …… Monsieur R.

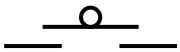
Docteur R. au royaume de la domotique

Nous allons aborder différents exemples pouvant directement s'appliquer dans un projet de domotique. D'un premier abord simple, ces exemples nous permettront d'appréhender différents aspects de la base des automatismes et de l'apprentissage d'AUTOMGEN et d'IRIS.

Ce symbole évoquera dans ce qui suit une partie commande (un automate programmable par exemple.

Est-il besoin de préciser que ceci

évoquera une ampoule,

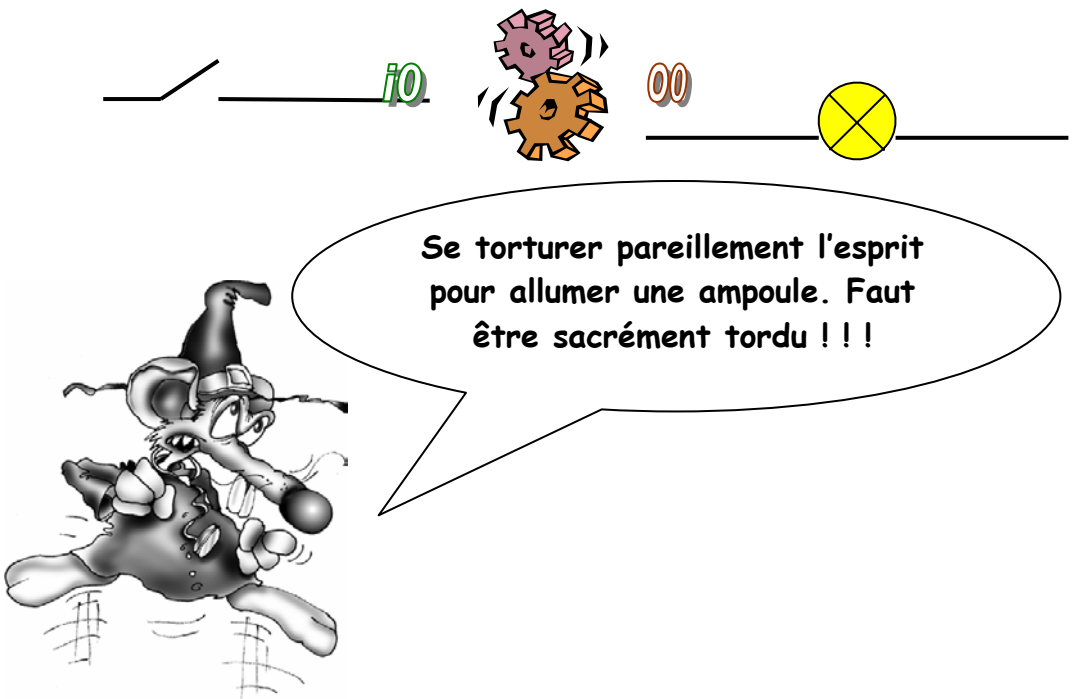


un bouton poussoir

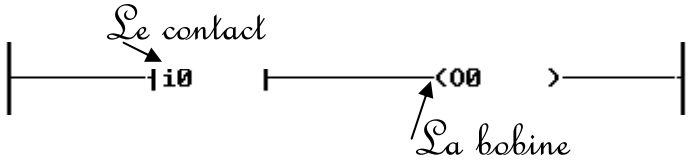
et cela un interrupteur ?

Premier exemple : « qui de l'interrupteur ou de l'ampoule était le premier … »

Un simple interrupteur et une simple ampoule : l'interrupteur est câblé sur l'entrée i0, l'ampoule sur la sortie o0. Si l'interrupteur est fermé alors l'ampoule s'allume, si l'interrupteur est ouvert l'ampoule s'éteint. Difficile de faire plus simple.

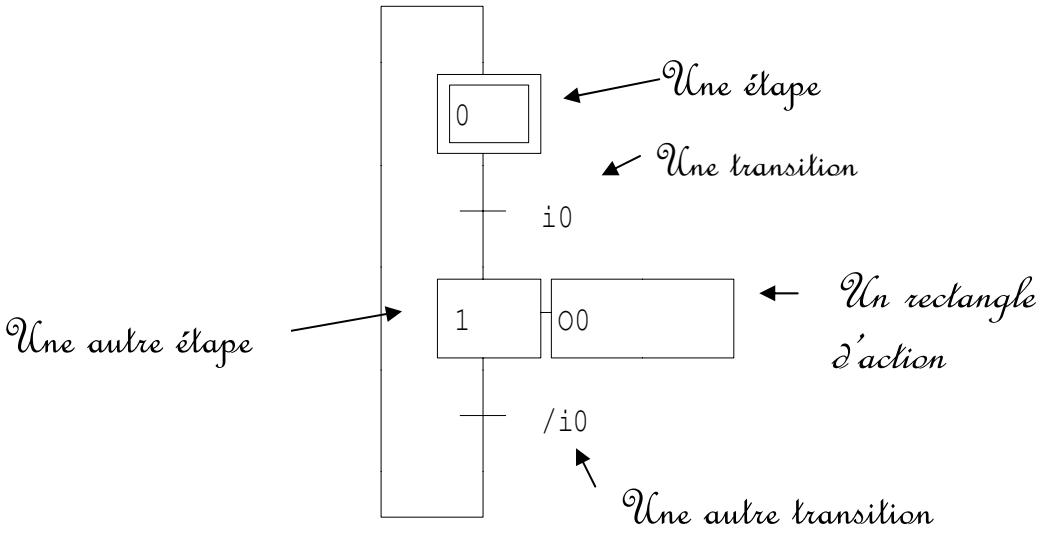


Solution 1 : le langage naturel de l'électricien : le ladder



Le ladder est la transcription la plus directe du schéma électrique. Le contact reçoit le nom de l'entrée où est câblé l'interrupteur, la bobine le nom de la sortie où est câblée l'ampoule.

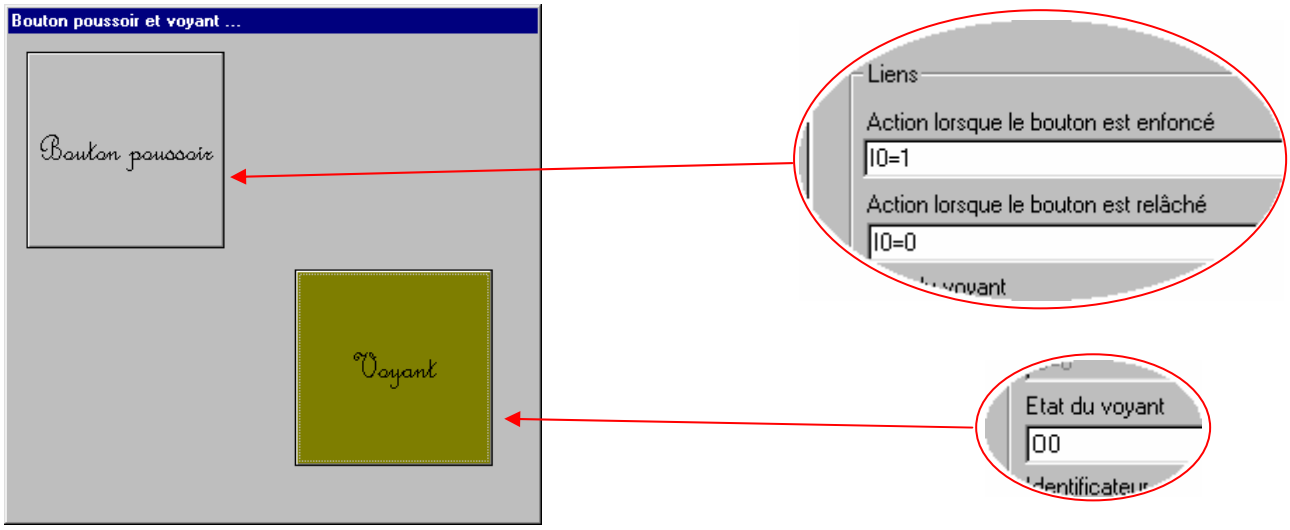
Solution 2 : le langage séquentiel de l'automaticien : le Grafcet

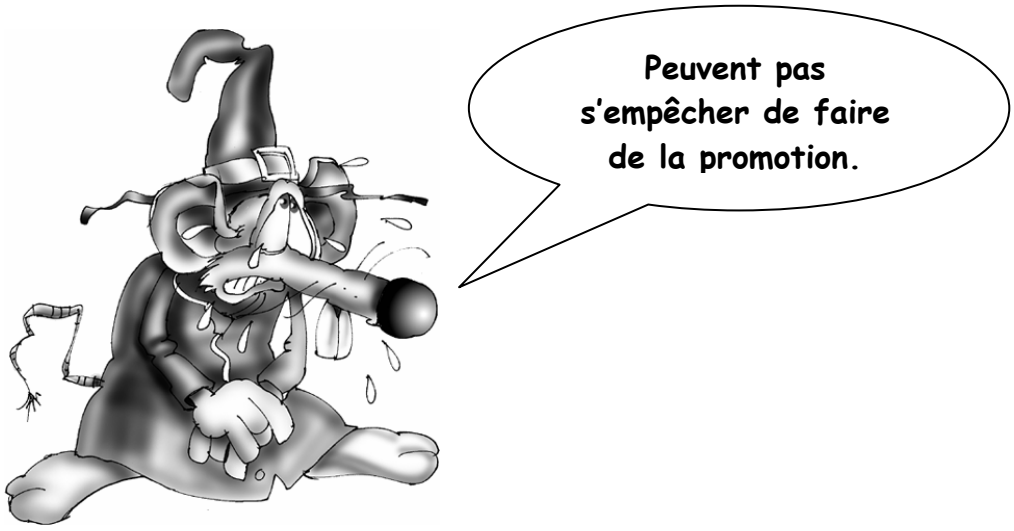


Le Grafcet est basé sur la notion d'état. Pour notre problème nous pouvons dire qu'il y a deux états : l'état allumé et l'état éteint. Chaque étape représente un état : ici l'étape 0 représente l'état éteint et l'étape 1 l'état allumé. Reste à déterminer la

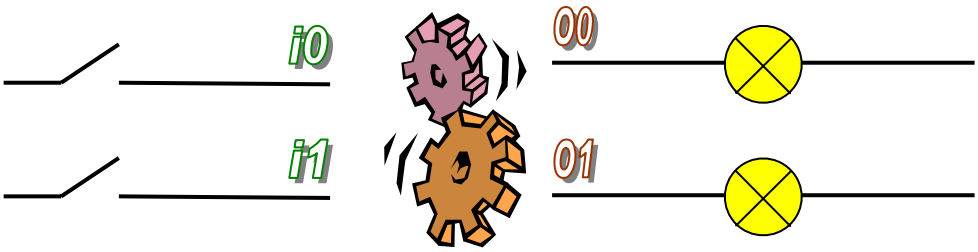
condition qui fait évoluer de l'état éteint à l'état allumé : ici l'interrupteur fermé (noté i0) puis la condition qui fait passer de l'état allumé à l'état éteint : ici l'interrupteur ouvert (noté / i0). Les conditions sont écrites à droite de l'élément noté transition. Le rectangle associé à l'étape 1 nommé rectangle d'action contient le nom de la sortie O0 (sortie où est câblée notre ampoule). Ainsi, à tout instant l'état de l'ampoule est le même que celui de l'étape 1.

Les heureux possesseurs d'IRIS peuvent utiliser deux objets BPVOYANT pour simuler ce premier exemple.



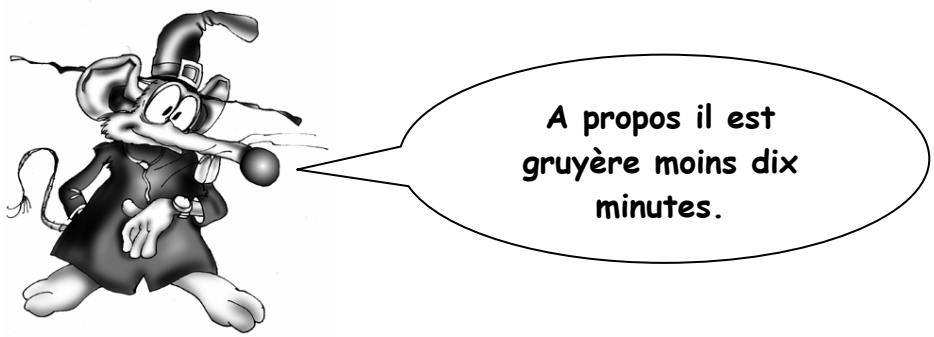


A vous de jouer …



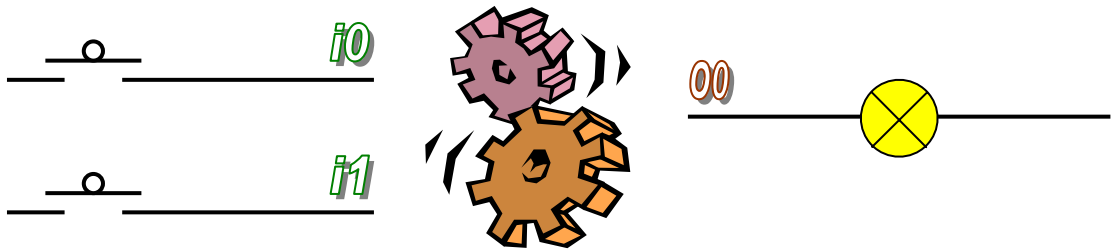
L'interrupteur 1 allume l'ampoule 1, l'interrupteur 2 l'ampoule numéro 2. Une solution Grafcet vous est proposée à la fin de ce document.

Deuxième exemple : « temporisations, minuteries et autres amusements temporels… »

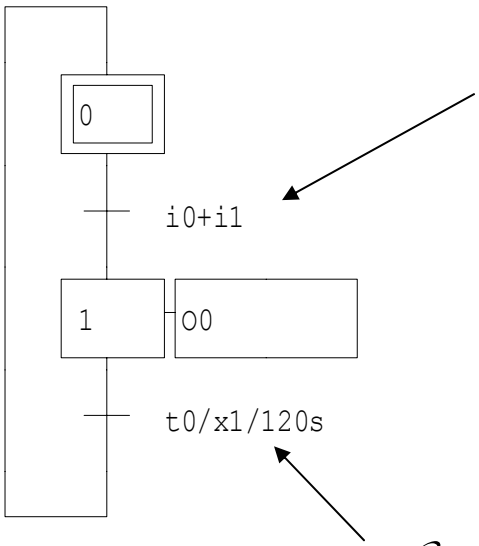


Comme vous vous en doutez certainement, la notion de temporisation est utilisée lorsqu'un programme doit, d'une façon ou d'une autre, effectuer des actions en tenant compte d'une donnée relative au temps. Attendre un certain temps avant de faire une action ou faire une action pendant un certain temps par exemple.

Notre deuxième exemple est le suivant : un couloir est équipé d'une ampoule et de deux boutons poussoirs. L'appui sur un des deux boutons poussoirs provoque l'allumage de l'ampoule pendant 2 minutes (d'après Dr R. c'est largement suffisant pour traverser le couloir).



Solution 1 : la simplicité

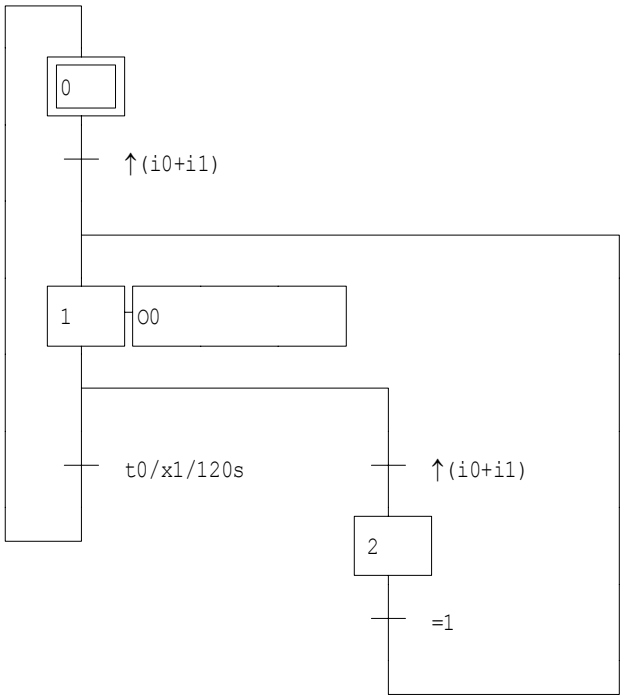


*Allumer l'ampoule si le bouton poussoir 1 est pressé **ou** si le bouton poussoir 2 est pressé. « Ou » s'écrit « + » dans une transition.*

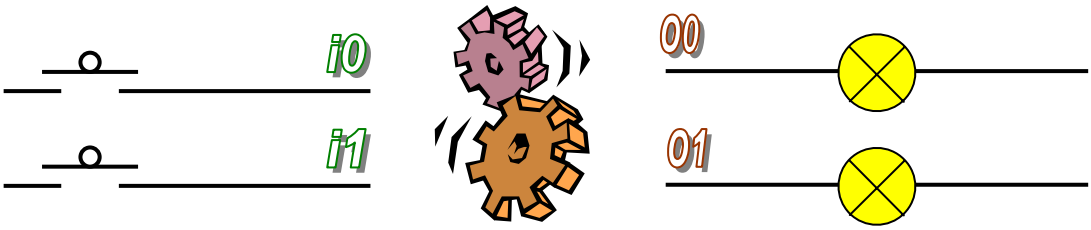
Attendre 2 minutes (120 secondes) en utilisant la temporisation 0, c'est l'étape 1 qui lance la temporisation.

Solution 2 : amélioration

Un problème posé par cette solution est que si l'on appuie sur un bouton poussoir pendant que l'ampoule est allumée, alors on ne réarme pas la temporisation. Ainsi Dr R. croyant avoir réarmé la minuterie s'est retrouvé dans le noir la semaine dernière.



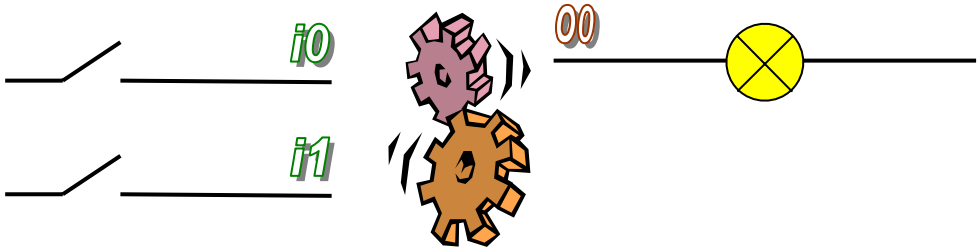
Et si vous vous lanciez dans l'écriture d'un vrai programme …



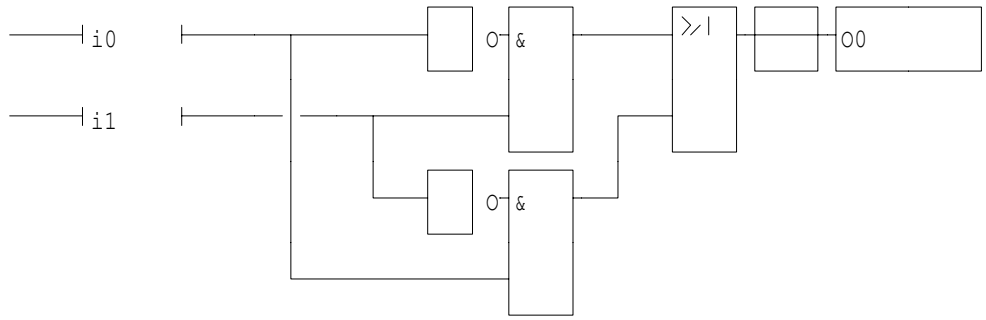
Une gestion intelligente de l'éclairage du couloir : une ampoule a été placée à chaque extrémité. Lorsqu'on appuie sur un interrupteur : les deux ampoules s'allument, puis l'ampoule se trouvant du côté de l'interrupteur pressé s'éteint au bout de 30 secondes et enfin l'autre ampoule au bout d'une minute.

Troisième exemple : « variation sur le thème du va et vient… »

Rappelons le principe au combien génial du va et vient : deux interrupteurs permettent d'allumer ou d'éteindre la même ampoule.

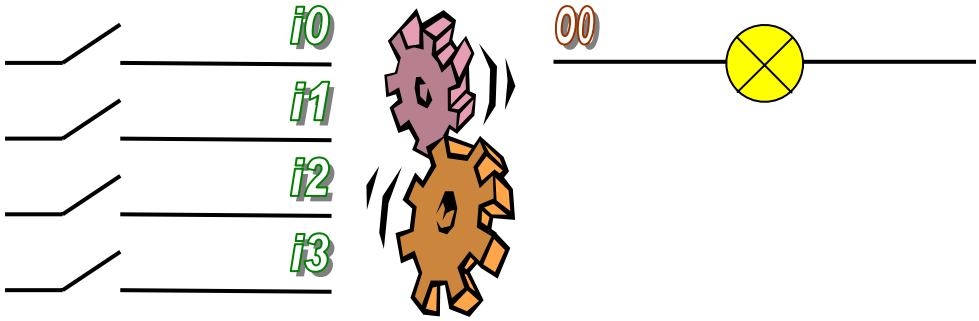


Voici une solution en logigramme :

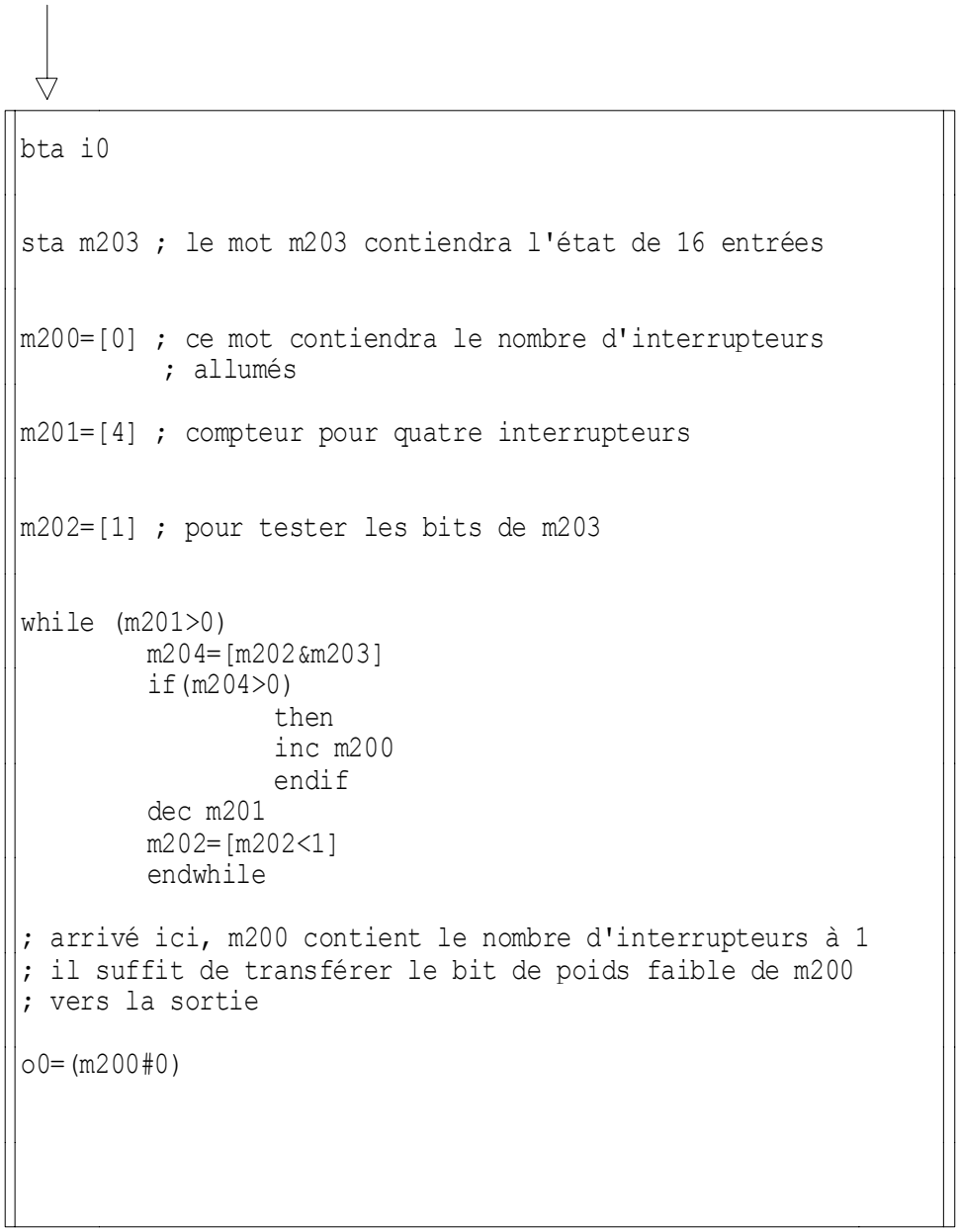


Les puristes auront reconnu l'équation booléenne du ou exclusif.

Les choses deviennent réellement intéressantes si l'on souhaite conserver les propriétés du va et vient avec un nombre d'interrupteurs supérieur à 2.

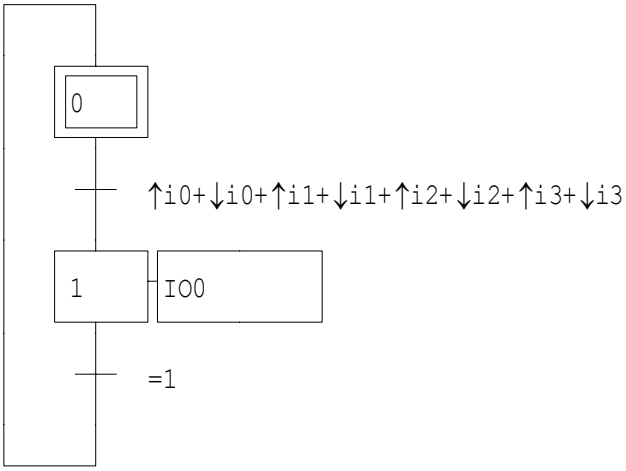


Une solution utilisant le langage littéral d'AUTOMGEN.



```
bta i0


sta m203 ; le mot m203 contiendra l'état de 16 entrées


m200=[0] ; ce mot contiendra le nombre d'interrupteurs
         ; allumés

m201=[4] ; compteur pour quatre interrupteurs


m202=[1] ; pour tester les bits de m203


while (m201>0)
        m204=[m202&m203]
        if(m204>0)
                then
                inc m200
                endif
        dec m201
        m202=[m202<1]
        endwhile

; arrivé ici, m200 contient le nombre d'interrupteurs à 1
; il suffit de transférer le bit de poids faible de m200
; vers la sortie

o0=(m200#0)
```

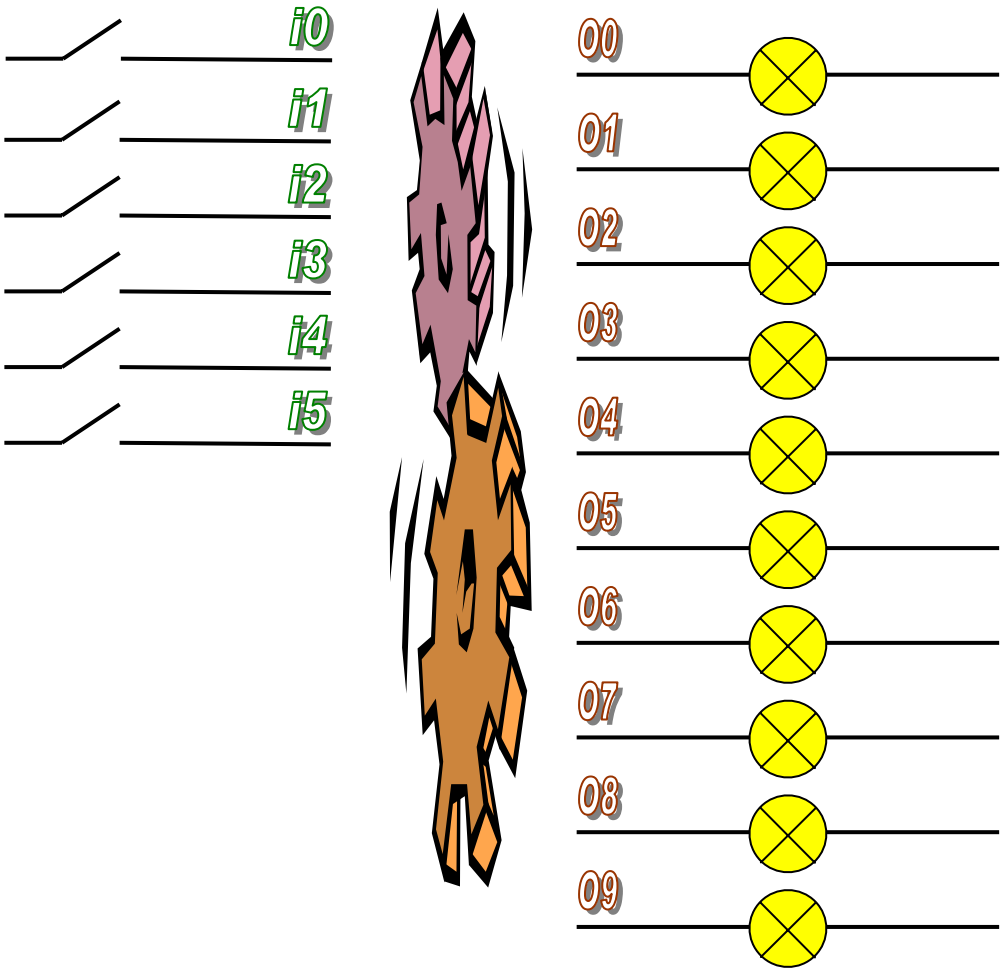
Une autre plus astucieuse :



« IO0 » signifie « inverser l'état de la sortie 0 ».

Essayez ceci :

Une grande pièce avec 6 interrupteurs et 10 ampoules. Chaque interrupteur permet d'éclairer plus ou moins la pièce (en clair de passer d'un état où tout est éteint à un état où une ampoule est allumée, puis deux, etc …).

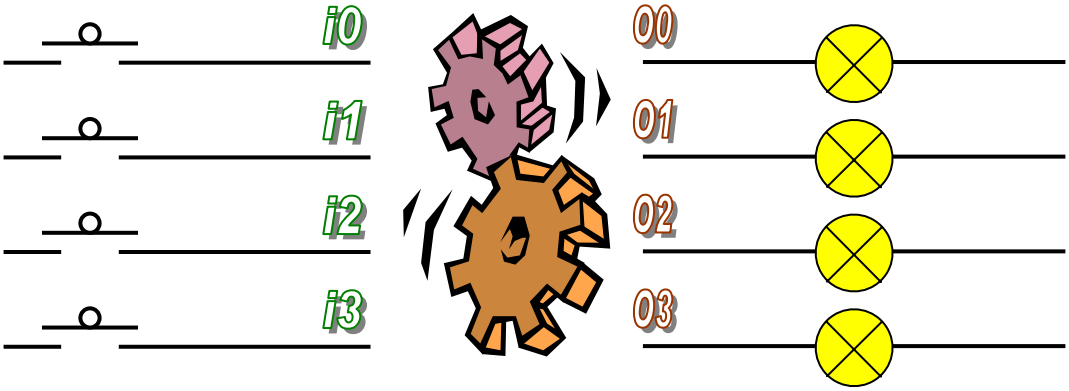


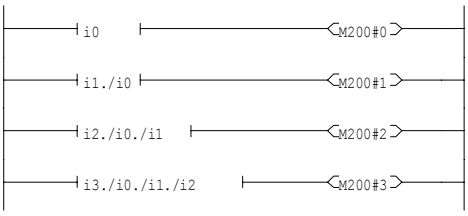
Quatrième exemple : « Et le bouton poussoir devint intelligent … »

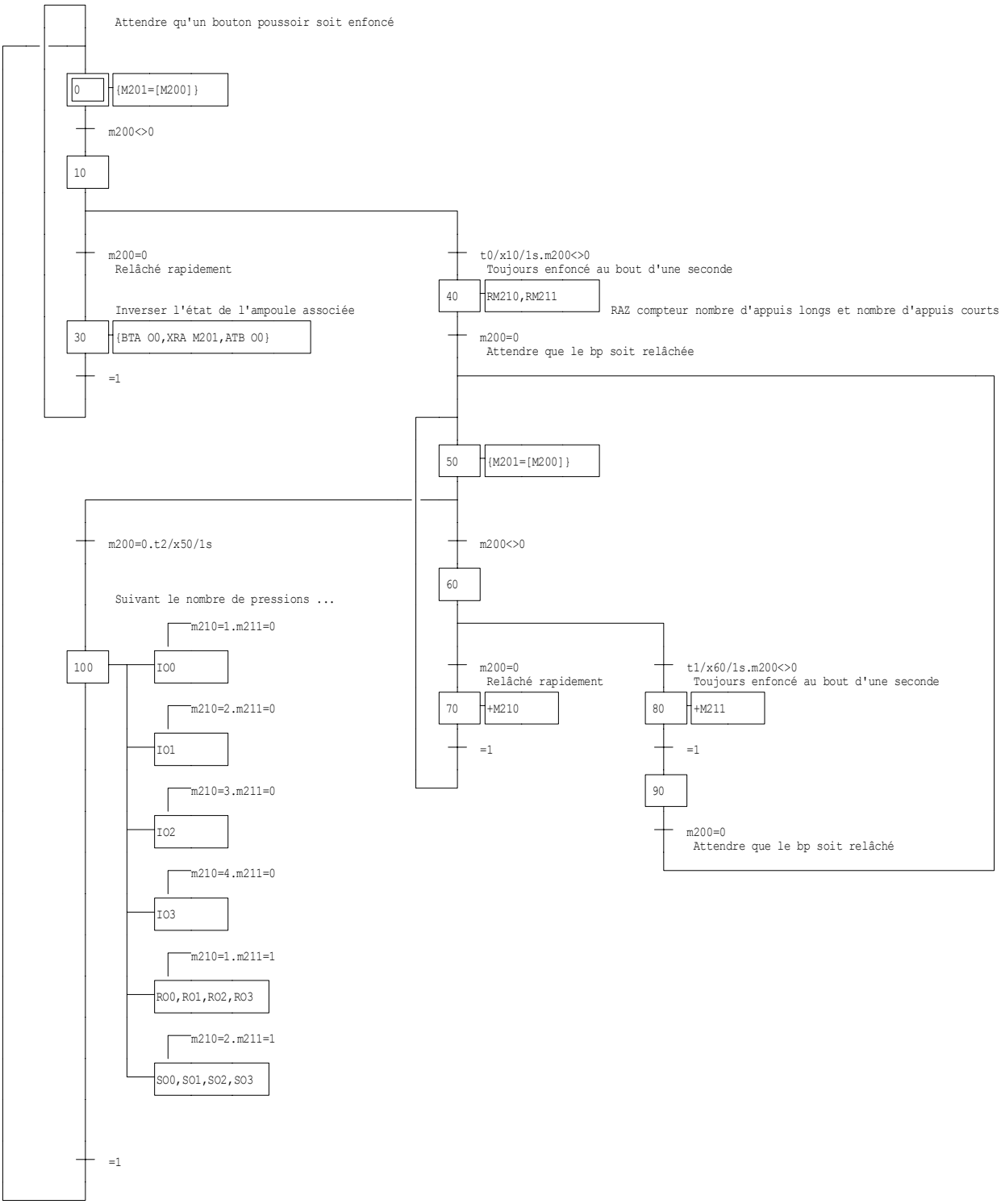
Dans tous les exemples précédents, les boutons poussoirs n'ont réalisés qu'une seule fonction. Entendez par là que la personne qui les manipule n'a que deux choix : ne pas appuyer dessus ou appuyer dessus pour obtenir une fonction (allumer ou éteindre). Imaginons un bouton poussoir « plus performant » capable de recevoir deux types de pression : une pression courte (arbitrairement moins de 1 seconde) ou une pression longue (arbitrairement au moins une seconde).

Pour cet exemple quatre boutons poussoirs et quatre ampoules. Par défaut, en utilisation normale chacun des boutons poussoirs est associé à une ampoule. Une pression courte sur un bouton poussoir allume ou éteint l'ampoule associée. Chaque bouton poussoir doit permettre de piloter chaque ampoule ou la totalité des ampoules. Le tableau ci-dessous résume le fonctionnement.

Type d'action sur les boutons poussoirs	**Résultat**
Une pression courte	L'ampoule associée change d'état
Une pression longue et une pression courte	L'ampoule numéro 1 change d'état
Une pression longue et deux pressions courtes	L'ampoule numéro 2 change d'état
Une pression longue et trois pressions courtes	L'ampoule numéro 3 change d'état
Une pression longue et quatre pressions courtes	L'ampoule numéro 4 change d'état
Deux pressions longues et une pression courte	Toutes les ampoules s'éteignent
Deux pressions longues et deux pressions courtes	Toutes les ampoules s'allument




i0
M200#0
i1./i0
M200#1
i2./i0./i1
M200#2
i3./i0./i1./i2
M200#3


Attendre qu'un bouton poussoir soit enfoncé
0
{M201=[M200]}
m200<>0
10
m200=0
Relâché rapidement
Inverser l'état de l'ampoule associée
30
{BTA O0,XRA M201,ATB O0}
=1
t0/x10/1s.m200<>0
Toujours enfoncé au bout d'une seconde
40
RM210,RM211
RAZ compteur nombre d'appuis longs et nombre d'appuis courts
m200=0
Attendre que le bp soit relâchée
50
{M201=[M200]}
m200=0.t2/x50/1s
Suivant le nombre de pressions ...
m210=1.m211=0
100
IO0
m210=2.m211=0
IO1
m210=3.m211=0
IO2
m210=4.m211=0
IO3
m210=1.m211=1
RO0,RO1,RO2,RO3
m210=2.m211=1
SO0,SO1,SO2,SO3
=1
m200<>0
60
m200=0
Relâché rapidement
70
+M210
=1
t1/x60/1s.m200<>0
Toujours enfoncé au bout d'une seconde
80
+M211
=1
90
m200=0
Attendre que le bp soit relâché

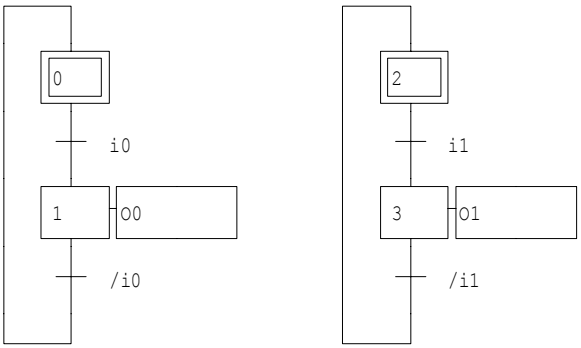
Ceci termine ce manuel pédagogique. Nous espérons qu'il vous aura permis de découvrir les possibilités d'AUTOMGEN.

Nous vous proposons un ultime exercice.

Automatiser l'appartement de votre tante Hortense en respectant son goût immodéré pour les interrupteurs nickelés.

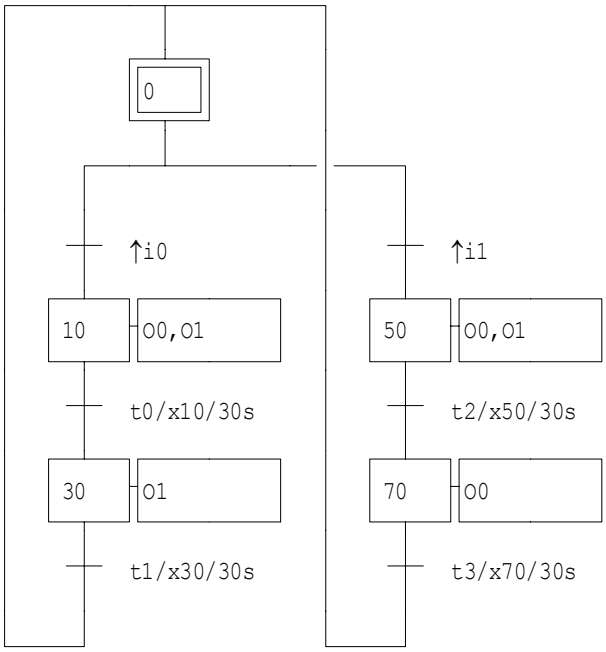
Les solutions …

« qui de l'interrupteur ou de l'ampoule était le premier … »

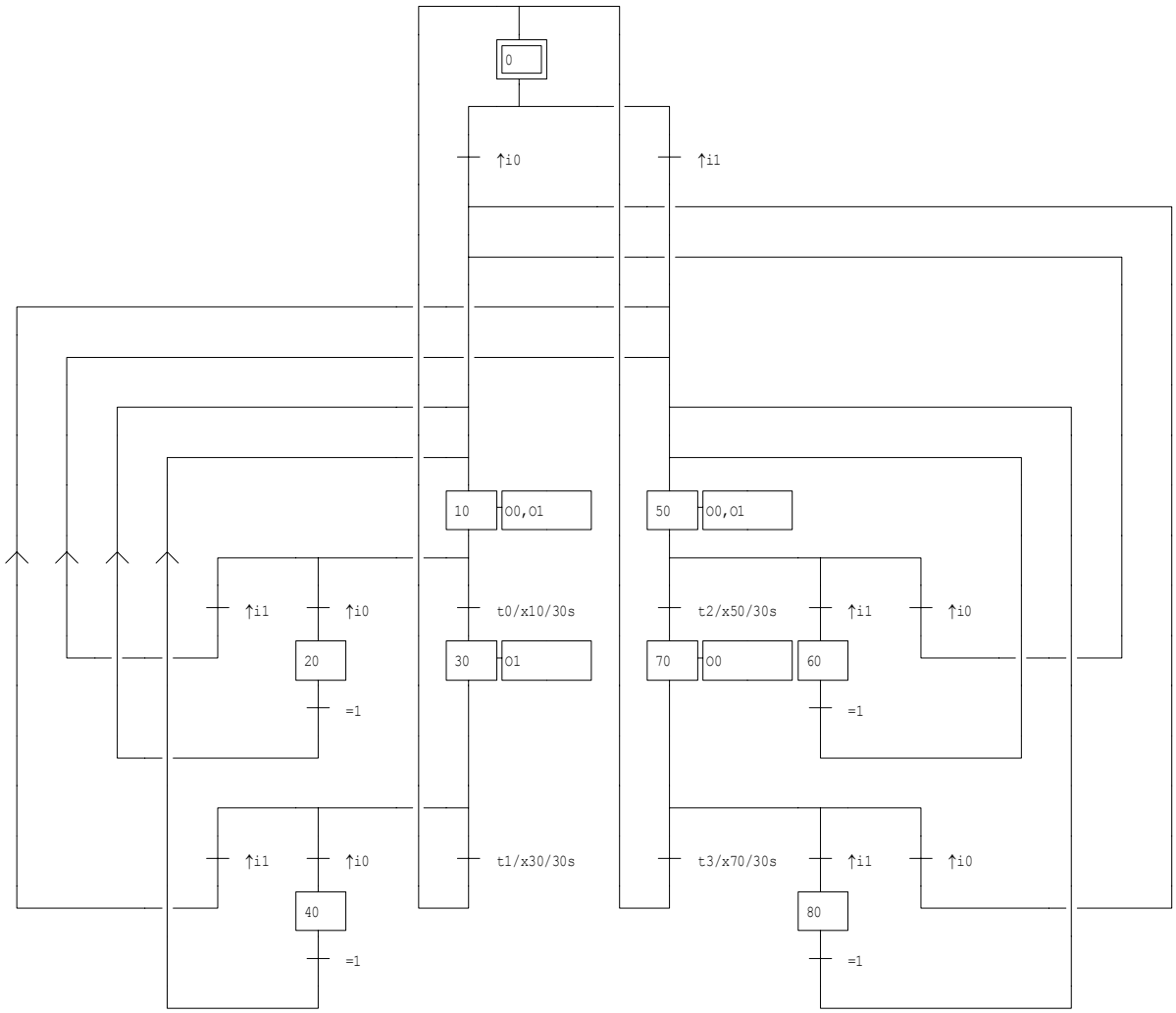


Il suffit d'écrire deux Grafcets identiques. Chacun s'occupe indépendamment d'un interrupteur et d'une ampoule.

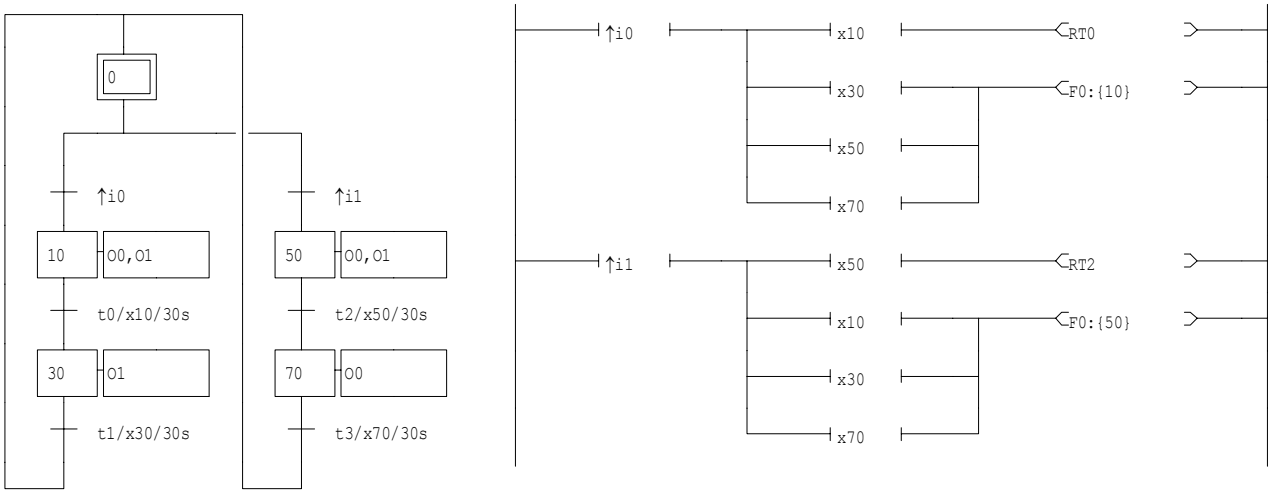
« temporisations, minuteries et autres amusements temporels… »



Une version simple sans la gestion du réarmement de la minuterie.



Le réarmement de la minuterie rend le programme très complexe.



Une troisième solution utilisant Grafcet, langage ladder et forçages de Grafcet. Le programme reste lisible.

« variation sur le thème du va et vient …»