Language reference manual

Examples

The adventures of Doctor R.

www.irai.com

1. Common elements

This chapter describes the common elements for all the languages used in AUTOMGEN.

The logo  refers to innovations used in version 7 of AUTOMGEN.

1.1. Variables

The following types of variables are present :

⇨ boolean type : the variable may have a true (1) or false (0) value.

⇨ numeric type : the variable may have a numeric value, different from the existing types: 16 bits variables, 32 bits and floating point.

⇨ time delay type : structured type, it is a combination of a boolean and numeric type.

Starting from version 6 the variable name syntax may be AUTOMGEN's or the syntax of IEC standard 1131-3.

1.1.1. Booleen variables

The following table provides a complete list of the Booleen variables used

Type	Syntax AUTOMGEN	Syntax IEC 1131-3	Comments
Input	I0 to I9999	%I0 to %I9999	May or may not correspond to physical input (depending on the I/O configuration of the target).
Output	O0 to O9999	%Q0 to %Q9999	May or may not correspond to physical output (depending on the I/O configuration of the target).
System Bits	U0 to U99	%M0 to %M99	For information on the system bits see the manual on the environment.
User bits	U100 to U9999	%M100 to %M9999	Internal bits for general use.

Grafcet Steps	X0 to X9999	%X0 to %X9999	Grafcet step bits
Word bits	M0#0 to M9999#15	%MW0 :X0 à %MW9999 :X15	Word bits: the number of bits is expressed in decimals and is included between 0 (lower weight bits) and 15 (higher weight bits).

1.1.2. Numeric variables

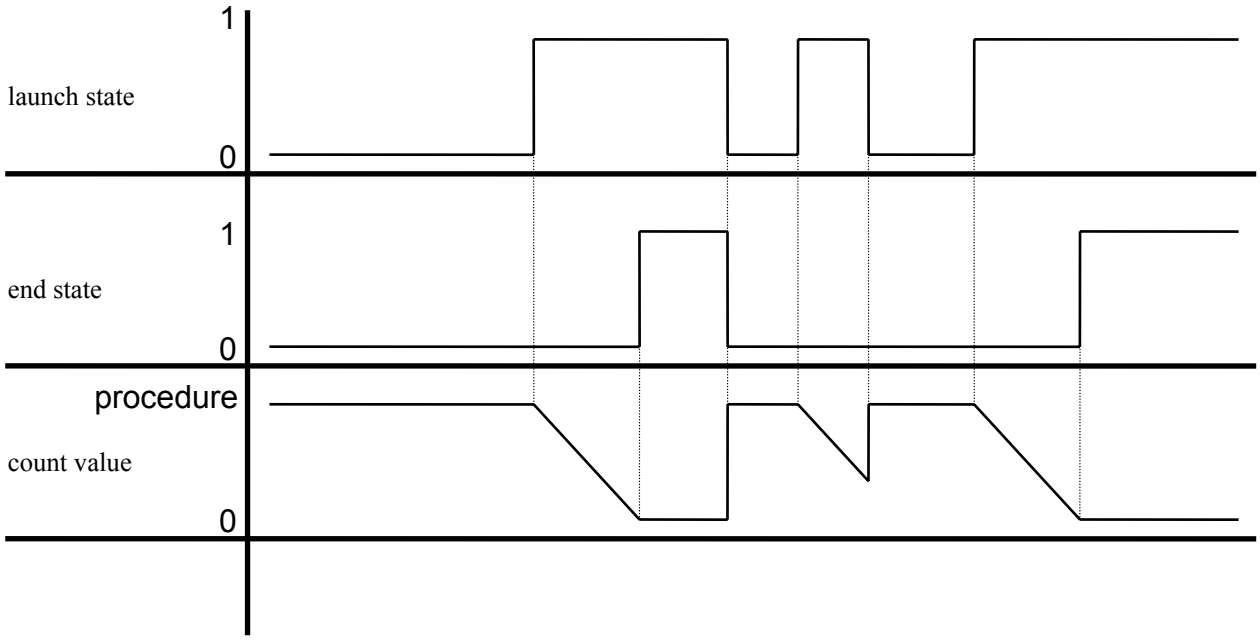
The following table provides a complete list of the numeric variables.

Type	Syntax AUTOMGEN	*IEC Syntax 1131-3*	Comments
Counter	C0 to C9999	%C0 to %C9999	16 bit counter, can be initialized, increased, decreased and tested with boolean languages without using literal language.
System Words	M0 to M199	%MW0 to %MW199	For information on the system words see the manual on the environment.
User words	M200 to M9999	%MW200 to %MW9999	16 bit words for general use.
Long integer	L100 to L4998	%MD100 to %MD4998	Integer value of 32 bits
Float	F100 to F4998	%MF100 to %MF4998	Real value of 32 bits (format IEEE).

1.1.3. Time delay

Time delay is a combined type which groups two boolean variables (launch state, end state) and two numeric variables on 32 bits (procedure and counter).

The following model shows a time chart of time delay functionality:



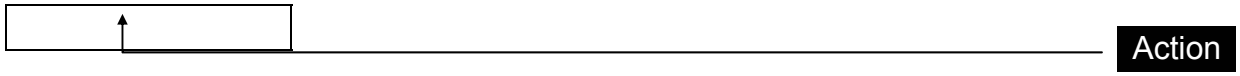
A time delay procedure value is between 0 ms and 4294967295 ms (a little over 49 days)

The time delay procedure can be modified by the program, see chapter 1.10.3. (instruction STA).
The time delay counter can be read by the program, see chapter 1.10.3. (instruction LDA).

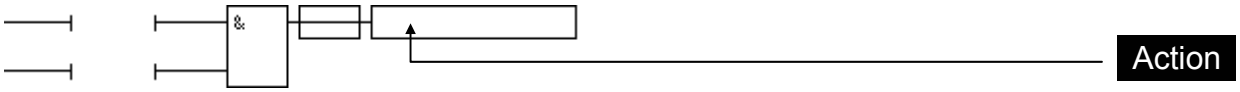
1.2. Actions

Actions are used in :

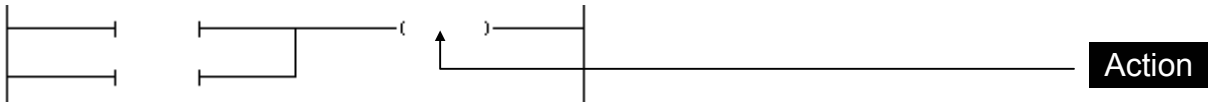
⇨ Grafcet language action rectangles,



⇨ flow chart language action rectangles,



⇨ ladder language coils.



1.2.1. Assignment of a boolean variable

The « Assignment » action syntax is :

«boolean variable»

Operation :

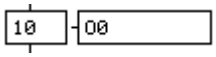
⇨ if the action rectangle or coil command is in a true state then the variable is put at 1 (true state),

⇨ if the action rectangle or coil command is in a false state then the variable is put at 1 (false state).

Truth table :

Command	Variable state (result)
0	0
1	1

Example :



If step 10 is active then O0 takes the value of 1, if not O0 takes the value 0.

Various « Assignment » actions can be used for the same variable in one program. In this case, the different commands are combined in « Or » logic.

Example :

State of X10	State of X50	State of O5
0	0	0
1	0	1
0	1	1
1	1	1

Complement assignment of a boolean variable

The « Complement assignment » action syntax is :

«N boolean variable»

Operation :

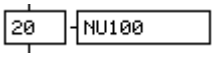
⇨ if the action rectangle or coil command is in a true state then the variable is reset (false state),

⇨ if the action rectangle or coil command is in a false state then the variable is set at 1 (true state).

Truth table :

Command	Variable state (result)
0	1
1	0

Example :



If step 20 is active, then U100 takes the value 0, if not U100 takes the value 1.

Various « Complement assignment » actions can be used for the same variable in one program. In this case, the different commands are combined in « Or » logic.

Example :

State of X100	State of X110	State of O20
0	0	1
1	0	0
0	1	0
1	1	0

1.2.2. Setting a boolean variable to one

The « Set to one » syntax is :

«S boolean variable»

Operation :

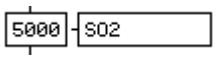
⇨ if the action rectangle or coil command is in a true state then the variable is set to 1 (true state),

⇨ if the action rectangle or coil command is in a false state then the state of the variable is not modified.

Truth table :

Command	Variable state (result)
0	unchanged
1	1

Example :



If step 5000 is active then O2 takes the value of 1, if not O2 keeps the same state.

1.2.3. Resetting a boolean variable

The « Reset» action syntax is :

«R boolean variable»

Operation :

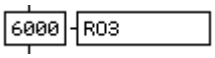
⇨ if the action rectangle or coil command is in a true state then the variable is reset (false state),

⇨ if the action rectangle or coil command is in a false state then the variable state is not modified.

Truth table :

Command	Variable state (result)
0	unchanged
1	0

Example :



If step 6000 is active then O3 takes the value of 0, if not O3 keeps the same state.

1.2.4. Inverting a boolean variable

The « Inversion » action syntax is :

«I boolean variable»

Operation :

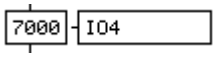
⇨ if the action rectangle or coil command is in a true state then the variable state is inverted for each execution cycle,

⇨ if the action rectangle or coil command is in a false state then variable state is not modified.

Truth table :

Command	Variable state (result)
0	unchanged
1	inverted

Example :



If step 7000 is active then the state of O4 is inverted, if not O4 keeps the same state.

1.2.5. Resetting a counter, a word or a long

The « Reset a counter, word or long» syntax is :

«R counter or word»

Operation :

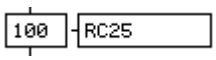
⇨ if the action rectangle or coil command is in a true state then the counter, word or long is reset,

⇨ if the action rectangle or coil command is in a false state then the counter, word or long is not modified.

Truth table :

Command	Value of counter, word or long (result)
0	unchanged
1	0

Example :



If step 100 is active then counter 25 is reset, if not C25 keeps the same value.

1.2.6. Incrementing a counter, a word or a long

The «Increment a counter » action syntax is :

«+ counter, word or long»

Operation :

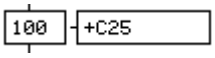
⇨ if the action rectangle or coil command is in a true state then the counter, word or long is incremented for each execution cycle,

⇨ if the action rectangle or coil command is in a false state then the counter, word or long is not modified.

Truth table :

Command	Counter, word or long value (result)
0	Unchanged
1	current value +1

Example :



If step 100 is active then counter 25 is incremented, if not then C25 keeps the same value.

1.2.7. Decrementing a counter, word or long

The « Decrement a counter » action syntax is :

«- counter, word or long»

Operation :

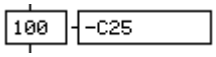
⇨ if the action rectangle or coil command is in a true state then the counter, word or long is decremented for each execution cycle,

⇨ if the action rectangle or coil command is in a false state then the counter, word or long is not modified..

Truth table :

Command	Value or counter, word or long (result)
0	unchanged
1	current value -1

Example :



If step 100 is active then counter 25 is decreased, if not C25 keeps the same value.

1.2.8. Time delays

Time delays are considered as boolean variables and can be used with « Assignment », « Complement assignment », « Set to one », « Reset », and « Invert ». The time delay order can be written after the action. The syntax is::
« time delay(duration) »
By default the duration is expressed in tenths of seconds. The letter « S » at the end of the duration indicates that it is expressed in seconds.

Examples :

Step 10 launches a time delay of 2 seconds which remains active as long as the step is. Step 20 sets a time delay of 6 seconds which remains active while step 20 is deactivated.
The same time delay can be used by different places with the same procedure and at different instants. In this case the time delay procedure must only be indicated once.

Note : other syntaxes exist for time delays. See chapter 1.3.3.

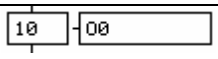
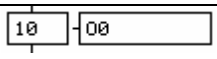
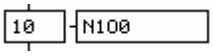
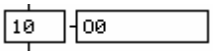
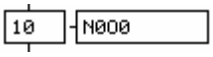
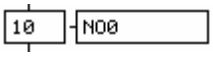
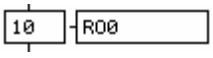
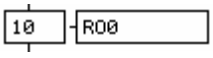
1.2.9. Interferences among the actions

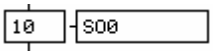
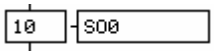
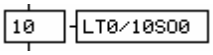
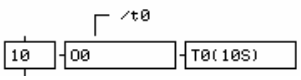
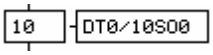
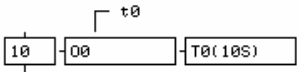
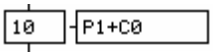
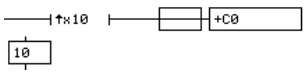
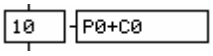
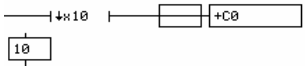
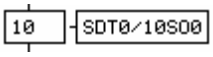
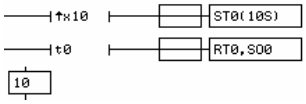
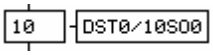
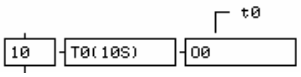
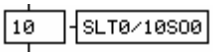
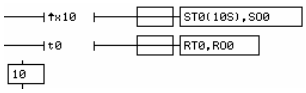
Certain types of actions cannot be used at the same time on a variable. The table below shows the combinations which cannot be used..

	Assignment	**Complement assignment**	**Set to one**	**Reset**	**Inversion**
Assignment	YES	NO	NO	NO	NO
Complement assignment	NO	YES	NO	NO	NO
Set to one	NO	NO	YES	YES	YES
Reset	NO	NO	YES	YES	YES
Inversion	NO	NO	YES	YES	YES

1.2.10. IEC1131-3 standard actions

The table below provides the IEC 1131-3 standard actions which can be used with AUTOMGEN V>=6 based on the AUTOMGEN. V5 standard syntax.

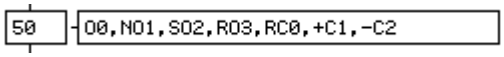
Name	*AUTOMGEN V>=6 Syntax*	*AUTOMGEN V5 Syntax*	*AUTOMGEN V>=6 Example*	*Equivalent example AUTOMGEN V5*
Not memorized	No value	No value		
Not memorized	N1	No value		
Complement not memorized	N0	N		
Reset	R	R		

Set to 1	S	S		
Limited in time	LTn/duration	Non-existent		
Time delay	DTn/duration	Non-existent		
Pulse on rising edge	P1	Non-existent		
Pulse on falling edge	P0	Non-existent		
Memorized and time delay	SDTn/duration	Non-existent		
Time delay and memorized	DSTn/duration	Non-existent		
Memorized limited in time	SLTn/duration	Non-existent		

1.2.11. Multiple actions

Within the same action rectangle or coil, multiple actions can be written by separating them with « , » (comma).

Example :



Multiple action rectangles (Grafcet and flow chart) or coils (ladder) can be combined. See the chapters on the relative languages for more information.

1.2.12. Literal code

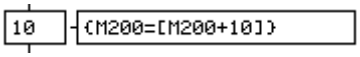
Literal code can be entered in an action rectangle or coil.

The syntax is :

« { literal code } »

Multiple lines of literal language can be written in braces. A « , » (comma) is also used here to separate them.

Example :

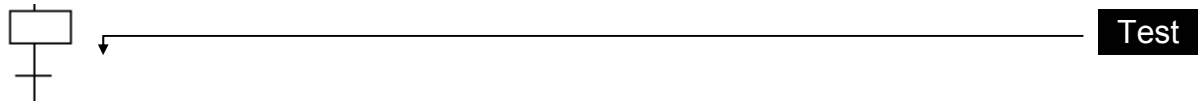


For more information see the chapters « Low level literal language », «Extended literal language » and «ST literal language».

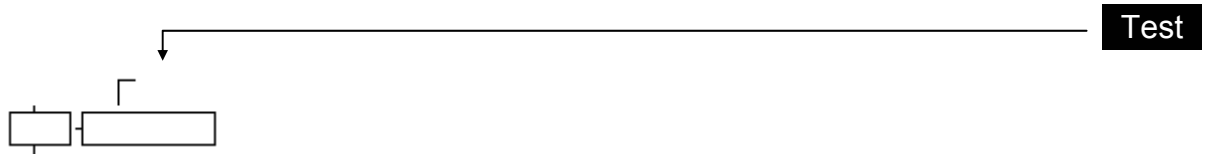
1.3. Tests

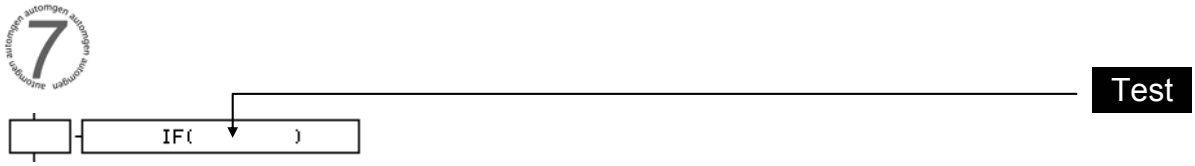
Tests are used in :

⇨ Grafcet language transitions,

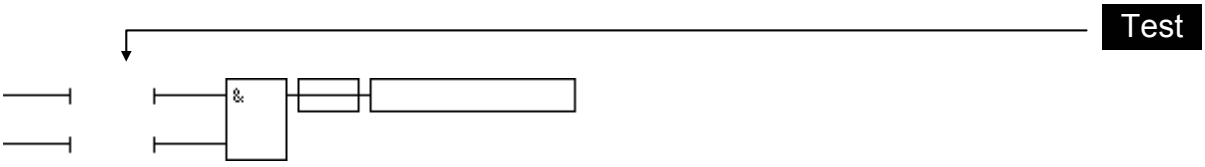


⇨ conditions based on Grafcet language action,

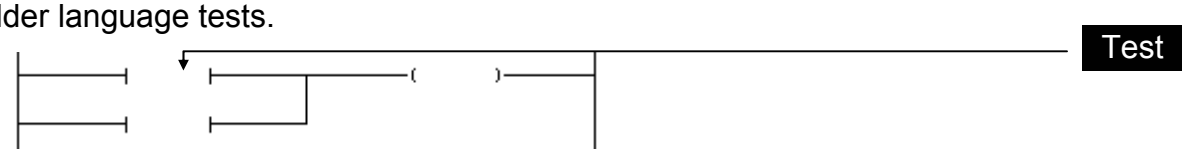




⇨ flow chart language tests,



⇨ ladder language tests.



1.3.1. General form

A test is a boolean equation composed of one or n variables separated by the operators « + » (or) or « . » (and).

Example of a test :

```
i0 (test input 0)
i0+i2 (test input 0 « or » input 2)
i10.i11 (test input 10 « and » input 11)
```

1.3.2. Test modifier

By default if only the name of one variable is specified, the test is « equal to one» (true). Modifiers make it possible to test the complement state, the rising edge and the falling edge.

⇨ the character « / » placed before a variable tests the complement state,

⇨ the character « u » or the character « ↑* » placed before a variable tests the rising edge

⇨ the character « d » or the character « ↓** » placed before a variable tests the falling edge

Text modifiers can be applied to one variable or to an expression between parentheses.

Examples :

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

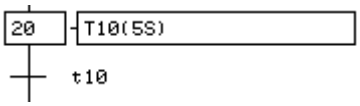
* To obtain this character when editing a test press the [↑] key.

** To obtain this character when editing a test press the [↓] key.

1.3.3. Time delays

Four syntaxes are available for time delays.

In the first the time delay is activated in the action and the time delay is simply mentioned in a test to check the end state :



For the others, everything is written in the test. The general form is :

« time delay /launch variable / duration »

or

« duration / launch variable / time delay »

or

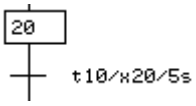
 « duration / launch variable »

In this case, a time delay is automatically attributed. The attribution range is that of the automatic symbols, see chapter 1.4.2. .

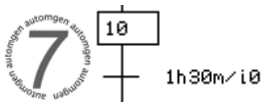
By default the duration is expressed in tenths of seconds.

 The duration can be expressed in days, hours, minutes, seconds, milliseconds with the operators « d », « h », « m », « s » and « ms ». For example : 1d30s = 1 day and 30 seconds.

Example using the second syntax :



Example using the normalized syntax :



1.3.4. Priority of boolean operators

By default the boolean operator «. » (AND) has a greater priority than the operator «+» (OR). Parentheses can be used to set a different priority.

Examples :

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

1.3.5. Always true test

The syntax of an always true test is :

« » (no value) or « =1 »

1.3.6. Numeric variable test

Numeric variable tests must use the following syntax :

« numeric variable » « test type » « constant or numeric variable »

The test type can be :

⇨ « = » equal,

⇨ « ! » or « <> » different,

⇨ « < » less than (not signed),

⇨ « > » greater than (not signed),

⇨ « << » less than (signed),

⇨ « >> » greater than (signed),

⇨ « <= » less than or equal to (not signed),

⇨ « >= » greater than or equal to (not signed),

⇨ « <<= » less than or equal to (signed),

⇨ « >>= » greater than or equal to (signed).

A float can only appear with another float or a real constant.

A long can only appear with another long or a long constant.

A word or a counter can only appear with a word, a counter or a 16 bit constant.

Real constants must be followed by the letter « R ».

Long constants (32 bits) must be followed by the letter « L ».

16 or 32 bit integer constants are written in decimal by default. They can be written in hexadecimal (suffix « $ » or « 16# ») or in binary (suffix « % » or « 2# »).

Numeric variable tests are used in equations like boolean variable tests. They can be used with test modifiers as long as they are in parentheses.

Examples :

```
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 on multiple lines

Transition text can be extended to multiple lines. The end of a transition line must be an operator « . » or « + ». A combination of key [CTRL] + [↓] and [CTRL] + [↑] makes it possible to move the cursor from one line to another.

1.4. Use of symbols

Symbols make it possible to associate a text to a variable.
Symbols can be used with all the languages.
A symbol must be associated to one and only one variable.

1.4.1. Symbol syntax

The symbols are composed of :

⇨ an optional character « _ » (low dash, generally associated with key [8] on the keyboard) which indicates the beginning of the symbol,

⇨ the name of the symbol,

⇨ an optional character « _ » (low dash) which indicates the end of the symbol.

The characters « _ » enclosing the symbol names are optional. They must be used if the symbol starts with a digit or an operator (+,-, etc…).

1.4.2. Automatic symbols

It can be a nuisance to have to set the attribution in each symbol and a variable, particularly if the precise attribution of a variable number is not very important. Automatic symbols are a solution to this problem, they are used to let the compiler automatically generate the attribution of a symbol to a variable number. The type of variable to use is provided in the name of the symbol.

1.4.3. Automatic symbol syntax

The syntax of automatic symbols is as follows :

```
_« symbol name » %« variable type »_
```

« variable type » can be :

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

It is possible to reserve multiple variables for a symbol. This is useful for setting tables. In this case the syntax is :

```
_« symbol name » %« variable »« length »_
```

«length » represents the number of variables to be reserved.

1.4.4. How does the compiler manage the automatic symbols ?

When starting to compile an application, the compiler cancels all the automatic symbols located in the « .SYM » file of the application. Each time the compiler finds an automatic symbol it creates a unique attribution for the symbol based of the variable type specified in the symbol name. The symbol that is then generated is written in the « .SYM » file. If the same automatic symbol is present more than once in an application, it refers to the same variable.

1.4.5. Range of variable attribution

By default, an attribution range is set for each type of variable :

Type	Start	End
I or %I	0	9999
O or %Q	0	9999
U or %M	100	9999
T or %T	0	9999
C or %C	0	9999
M or %MW	200	9999
L or %MD	100	4998
F or %MF	100	4998

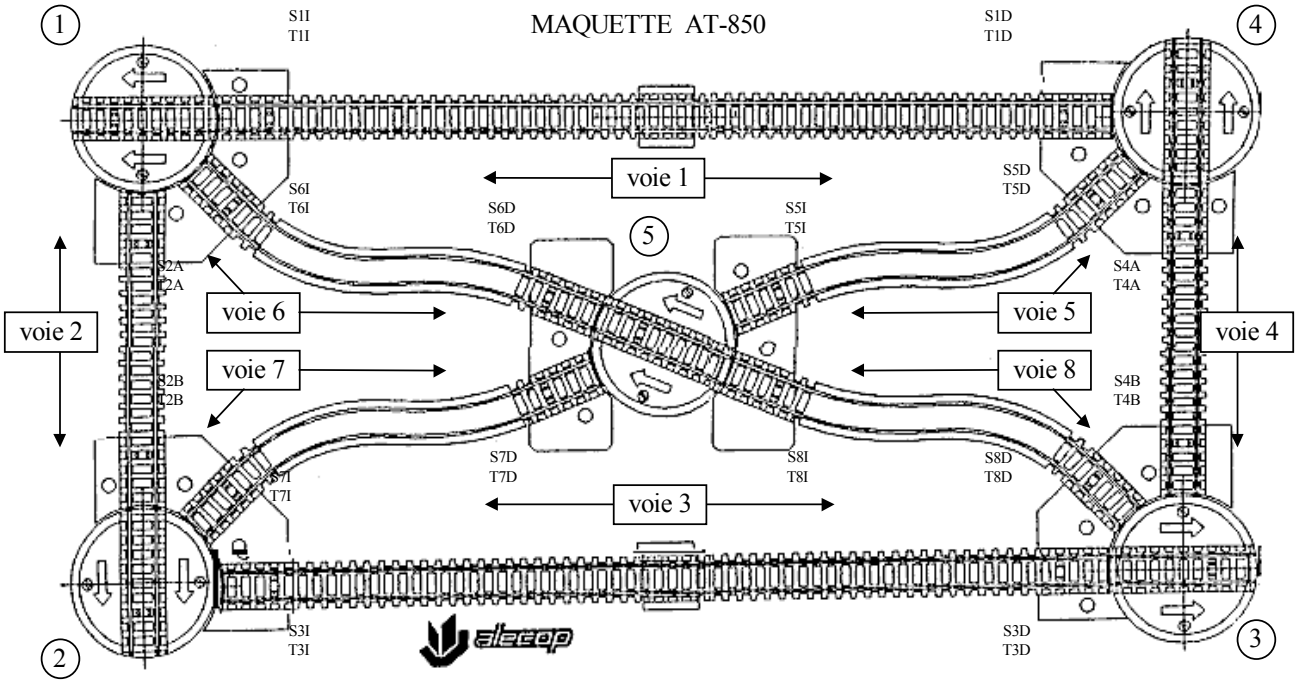
The attribution range can be changed for each type of variable by using the compilation command #SR« type »=« start », « end »

« type » designates the type of variable, start and end and the new limits to be used.

This command modifies the attribution of automatic variables for the entire sheet where it is written and up to the next « #SR » command.

1.5. Examples

To better illustrate this manual we have developed some functional examples with a model of a train as in the diagram below



We have used I/O cards on a PC to pilot this model. The symbols set by the constructor of the model have been saved.

The following symbol file was created :

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 supports the following elements :

- ⇨ divergences and convergences in « And» and in « Or »,
- ⇨ destination and source steps,
- ⇨ destination and source transitions,
- ⇨ synchronization,
  - ⇨ setting Grafcets,
  - ⇨ memorization of Grafcets,
  - ⇨ fixing,
  - ⇨ macro-steps.

1.6.1. Simple Grafcet

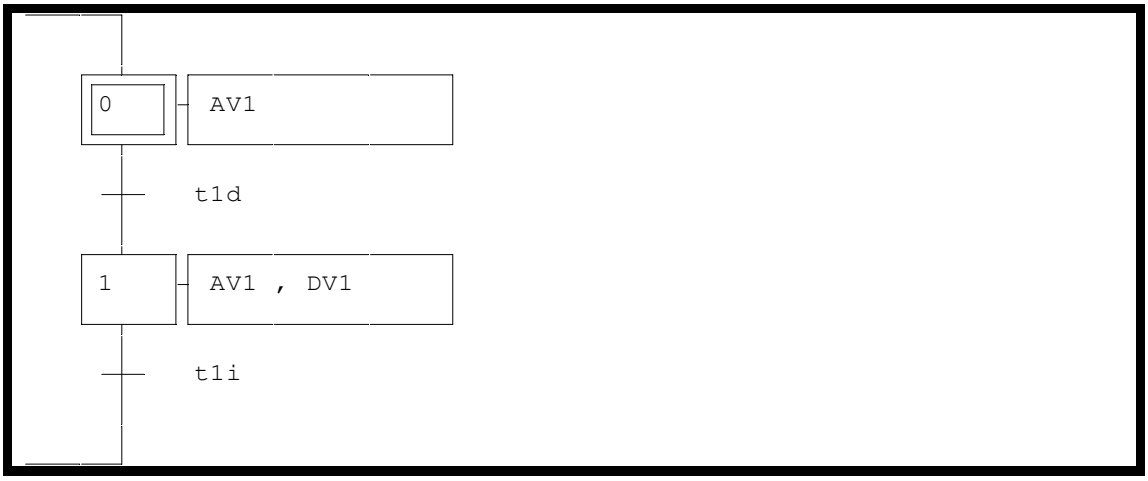
Grafcet line writing involves combined steps and transitions.

The example below illustrates a Grafcet line :

Conditions :

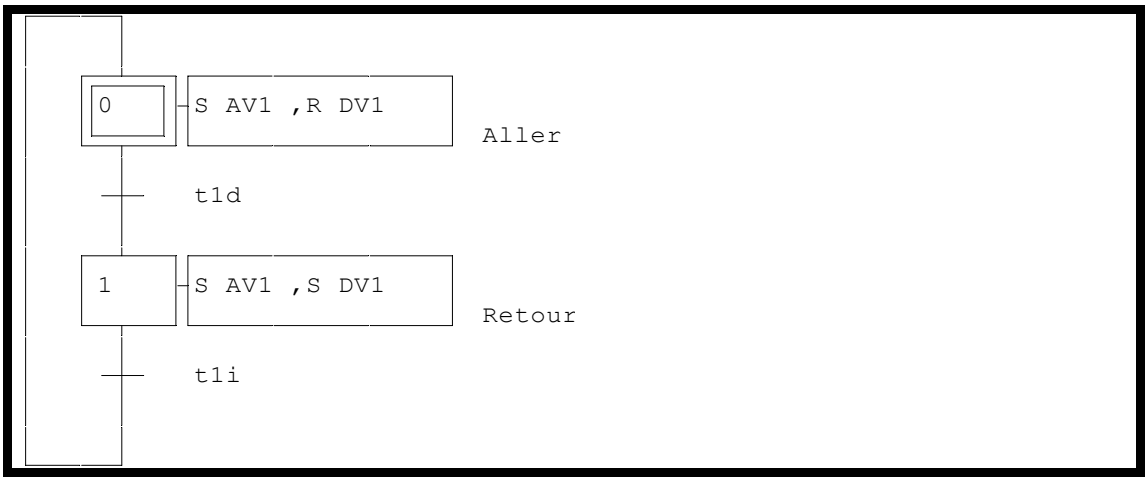
The locomotive needs to leave on track 3 towards the right, up to the end of the track. It returns in the opposite direction to the other end and starts again.

Solution 1 :



examples\grafcet\simple1.agn

Solution 2 :

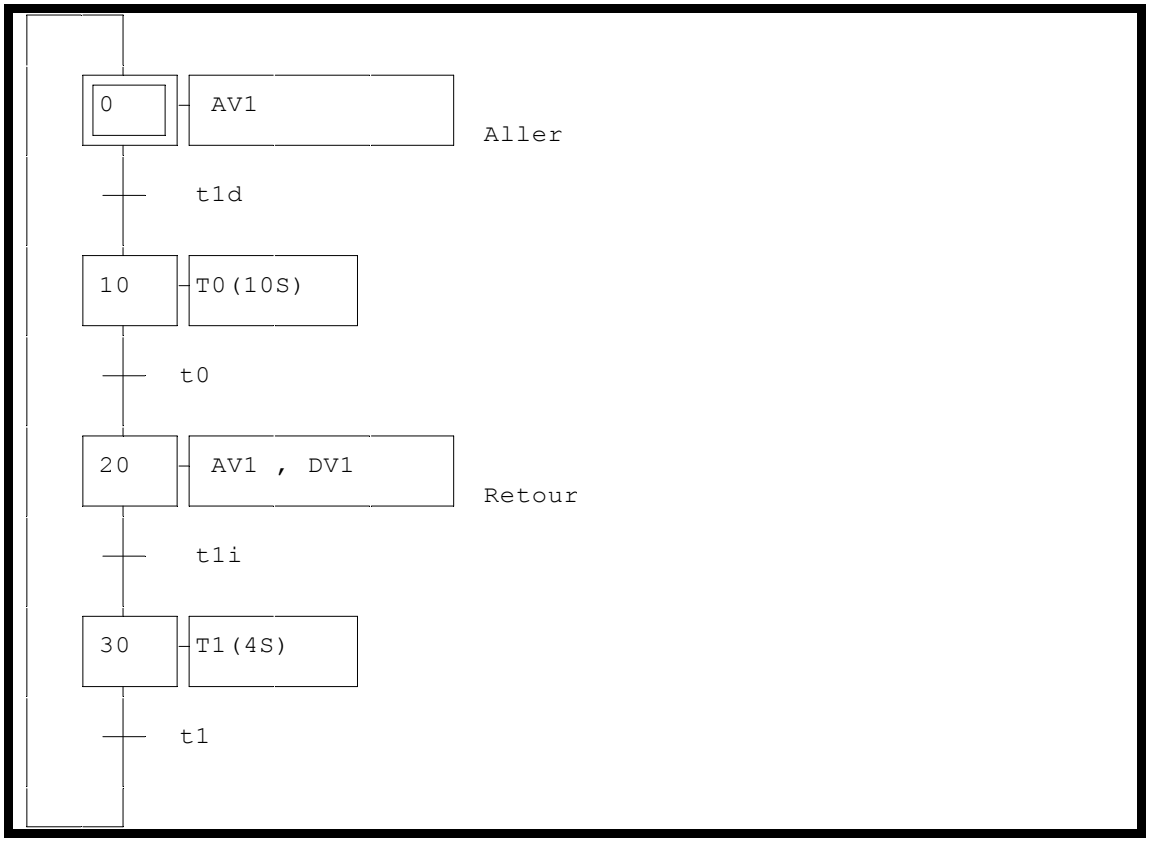


example\grafcet\simple2.agn

The difference between the two solutions is that the first example uses « Assignment » actions and the second uses « Set to one » and «Reset ».

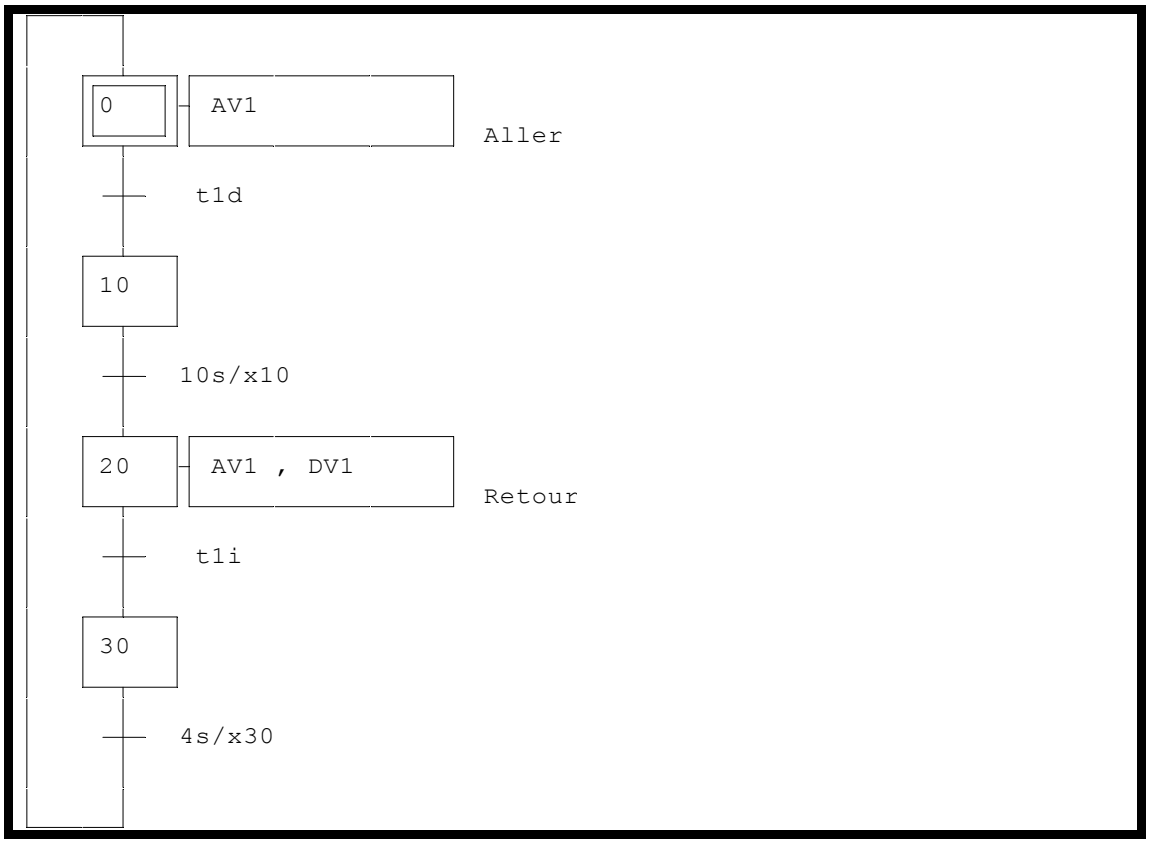
We change the conditions by setting a delay of 10 seconds when the locomotive arrives to the right of track 1 and a delay of 4 seconds when the locomotive arrives to the left of track 1.

Solution 1 :



example\grafcet\simple3.agn

Solution 2 :

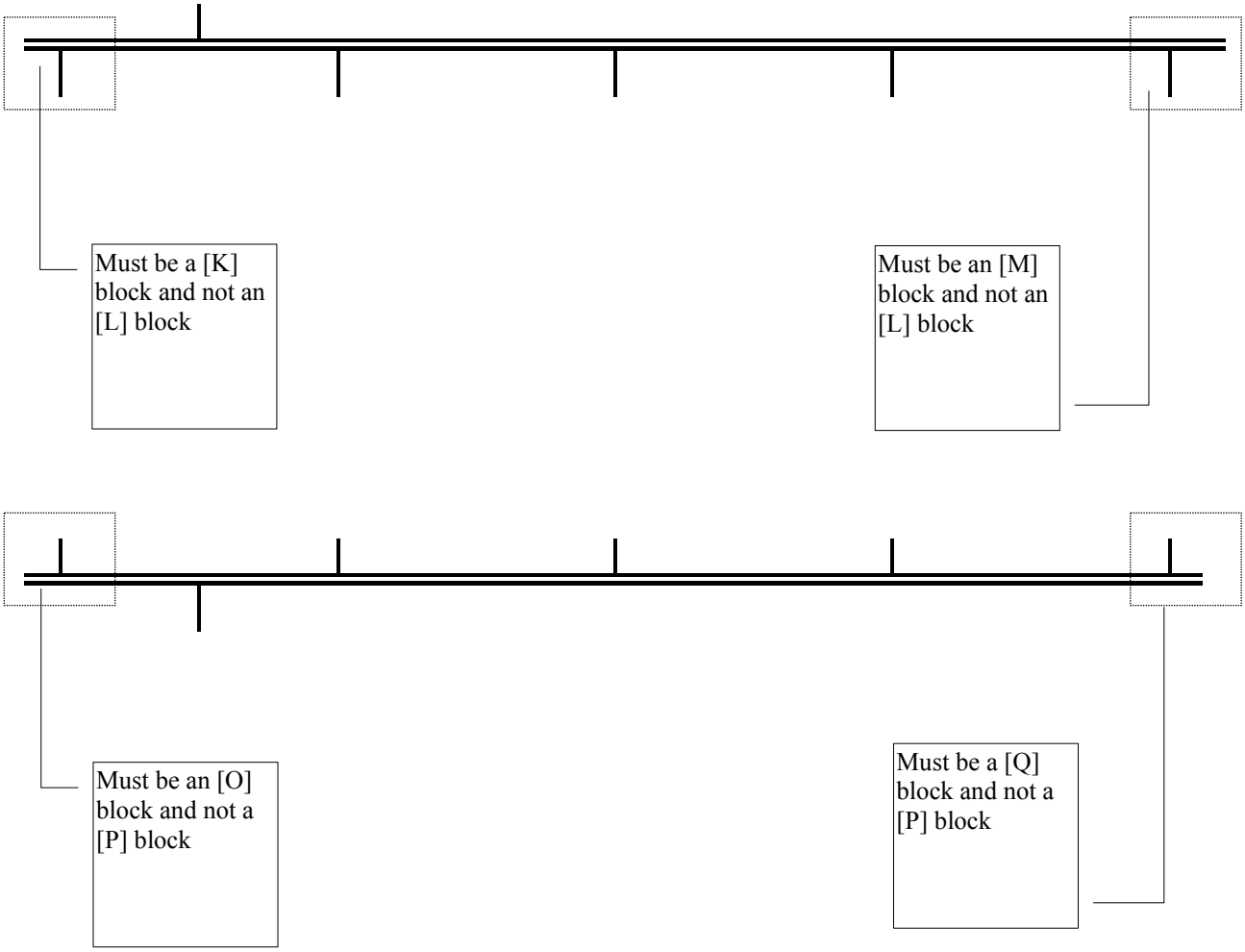


example\grafcet\simple4.agn

The difference between example 3 and 4 is in the choice of syntax used to set the time delays. In terms of operation the result is identical.

1.6.2. Divergence and convergence in « And »

Divergences in « And » can have n points. It is important to observe the use of the function blocks:

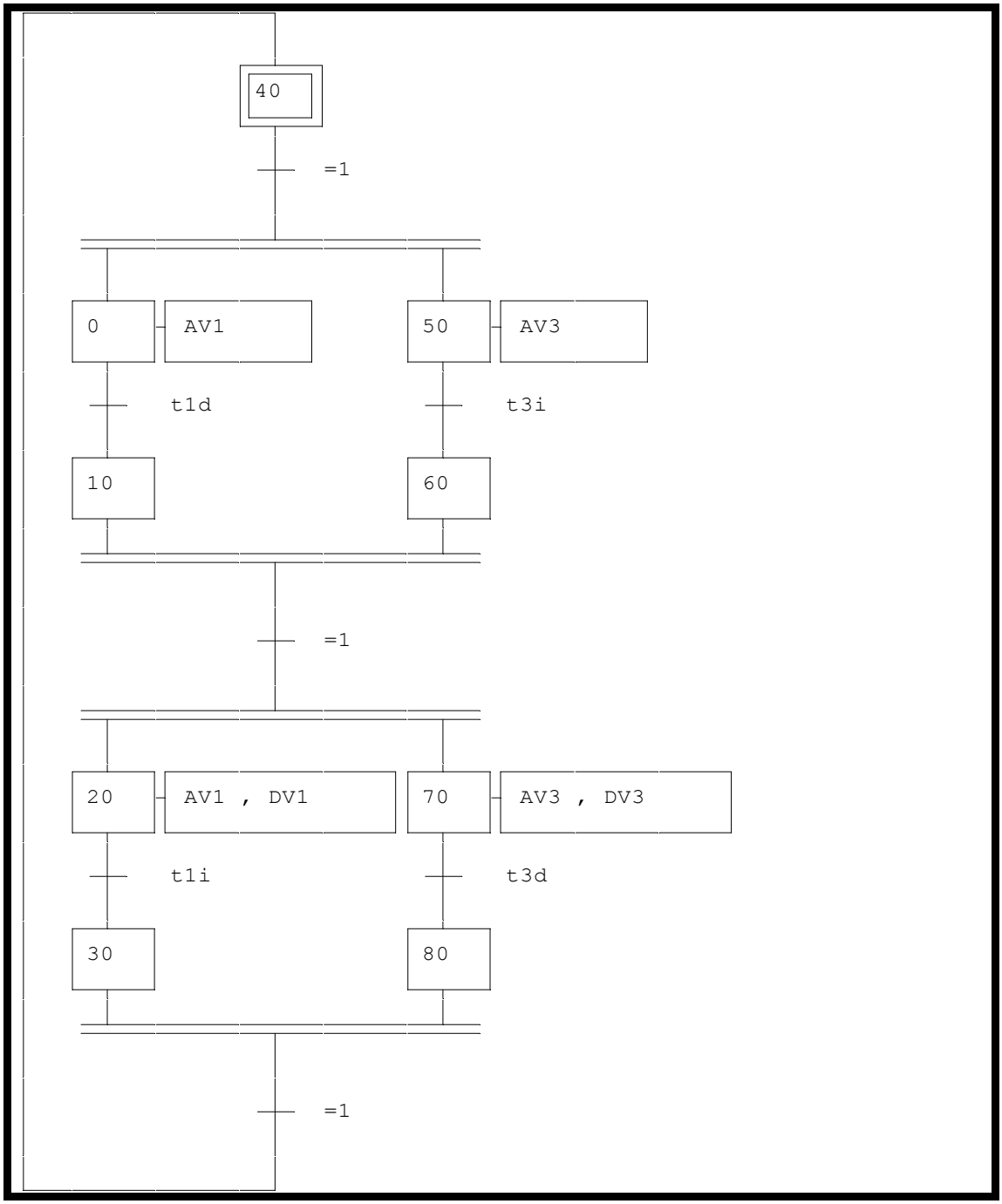


A description of the use of divergences and convergences in « And » follows.

Conditions :

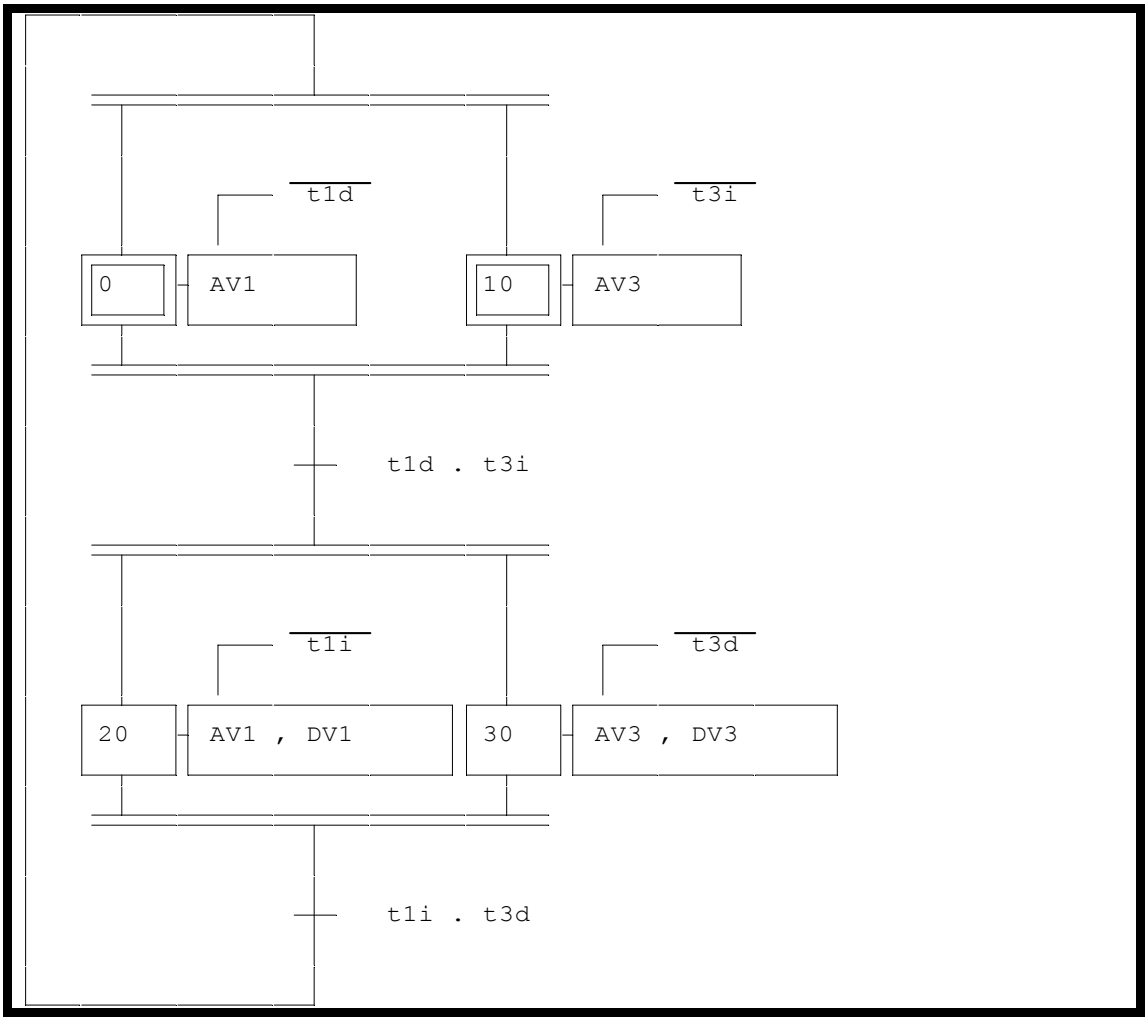
We are going to use two locomotives: the first makes round trip journeys on track 1, the second on track 3. The two locomotives are synchronized (they wait at the end of the track).

Solution 1 :



example\grafcet\divergence et 1.agn

Solution 2 :

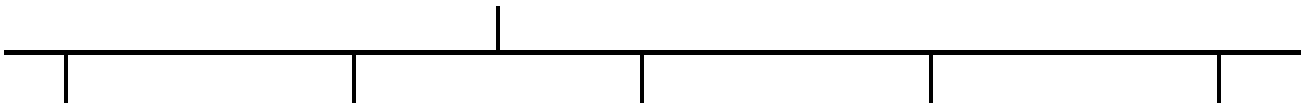


example\grafcet\divergence and 2.agn

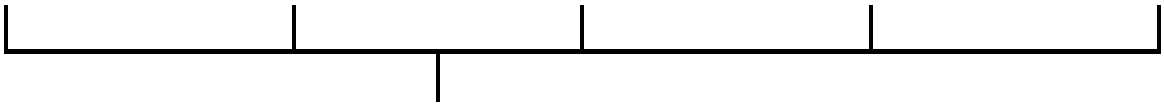
The two solutions are equivalent in terms of operation. The second is a more compact version which uses conditioned actions.

1.6.3. Divergence and convergence in « Or »

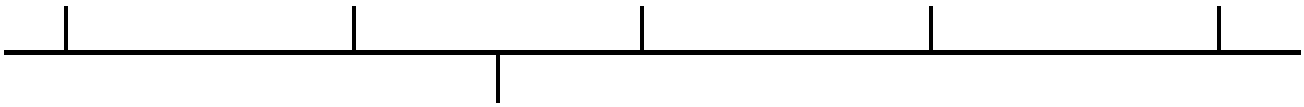
Divergences in « Or » can have n points. It is important to observe the use of the function blocks:



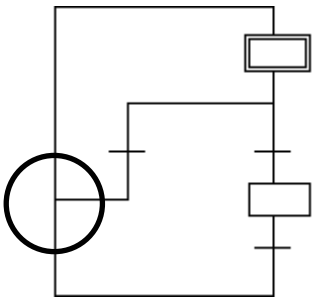
or



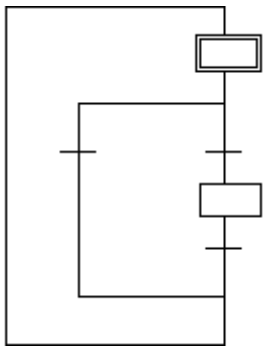
or



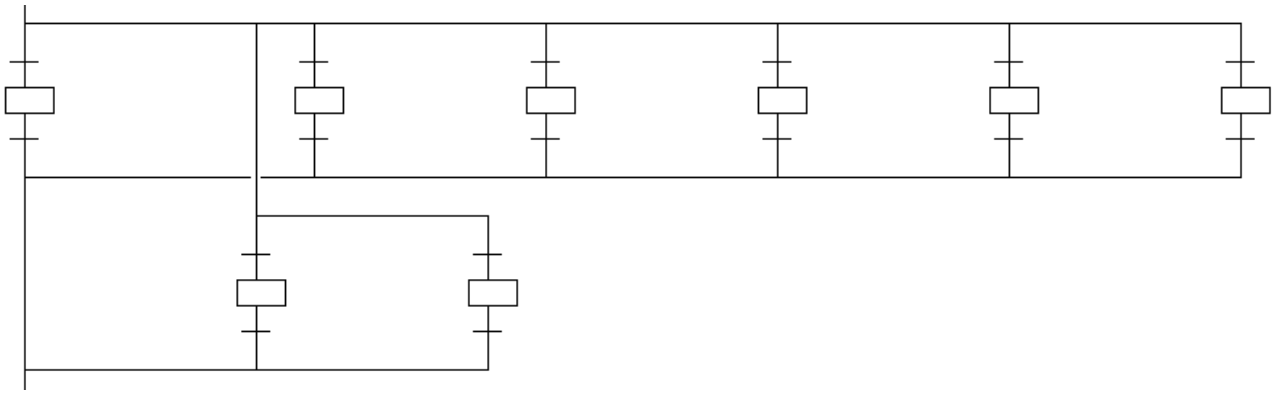
Divergences in « Or » must connect on descending links. For example :



incorrect, the right drawing is :



If the width of the page prevents you from writing a large number of divergences you can use the following type of structure:

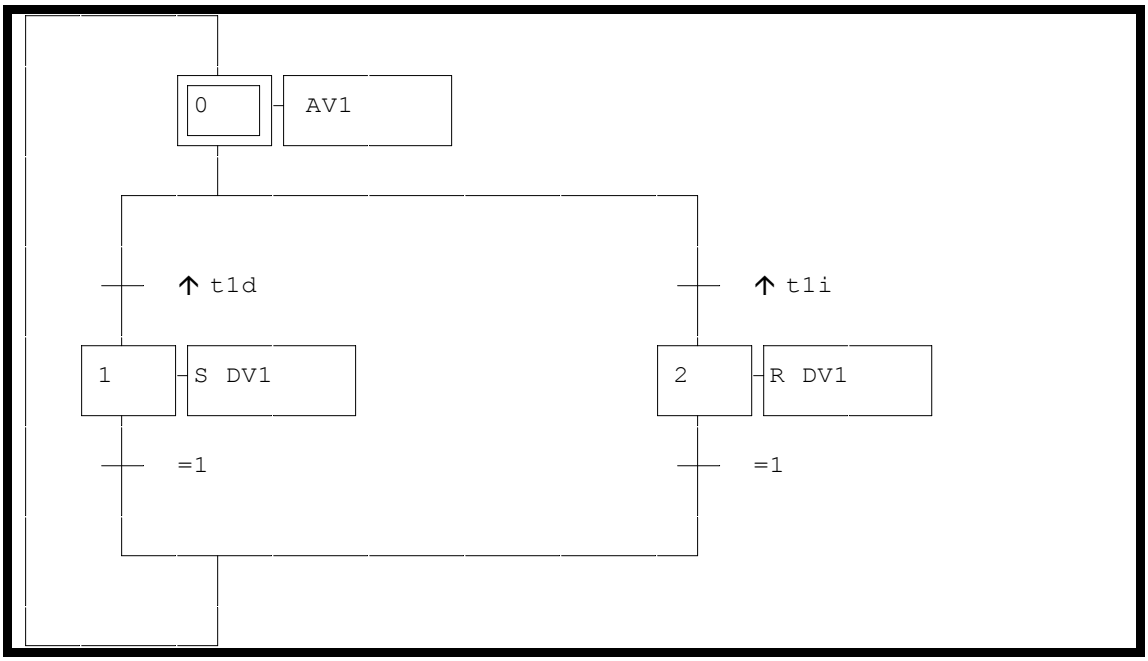


The following is an example to illustrate the use of divergences and convergences in « Or » :

Conditions :

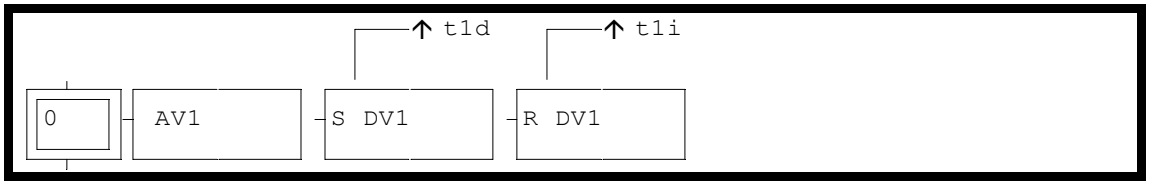
Let's look at the conditions for the first example in the chapter: roundtrip of a locomotive on track 1.

Solution :



example\grafcet\divergence or.agn

This Grafcet could be restated in a step using conditioned actions as in this example :



example\grafcet\action conditionnée.agn

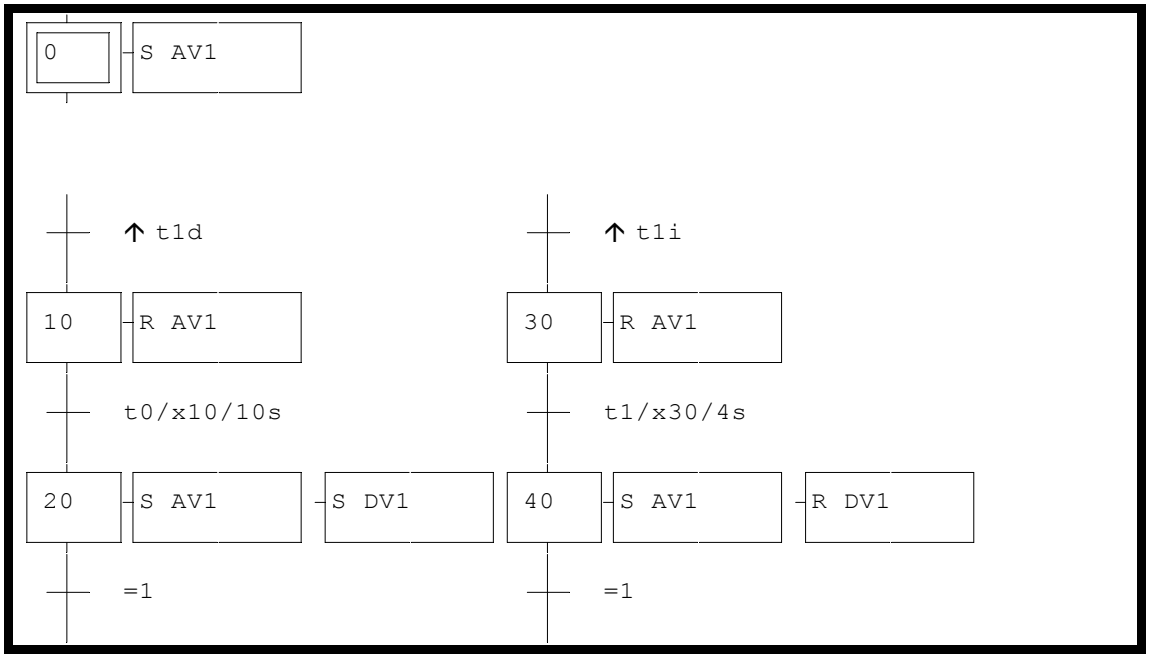
1.6.4. Destination and source steps, destination and source transitions

The principles are illustrated in the examples below:

Conditions :

Let's look at the second example in this chapter: round trip of a locomotive on track 1 with a delay at the end of the track.

Solution :



example\grafcet\étapes puits et sources.agn

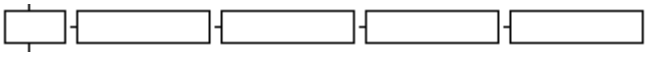
1.6.5. Multiple actions, conditioned actions

We have already used multiple and conditioned actions in this chapter. The two principles are described in detail below.

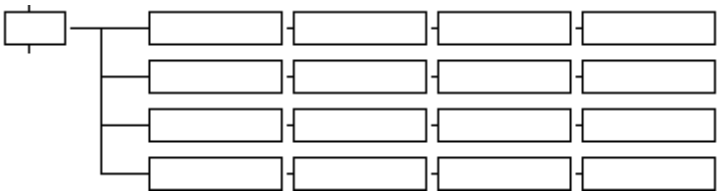
As described in the chapter dedicated to the compiler, multiple actions can be written in the same rectangle, by entering the character « , » (comma) as a separator.

When a condition is added to an action rectangle, all of the actions which continue in the rectangle are conditioned.

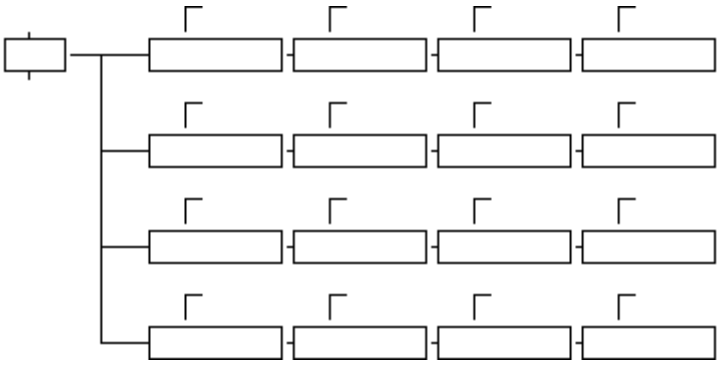
Multiple actions rectangles can be associated to a step.



another possibility :



Each of the rectangles can receive a different condition:



To draw a conditioned action, place the cursor on the action rectangle, click on the right side of the mouse and select « Conditional action » from the menu. To document the condition on the action, click on the element. .

IF(condition) syntax makes it possible to write a condition on the action in the action rectangle.

1.6.6. Synchronization

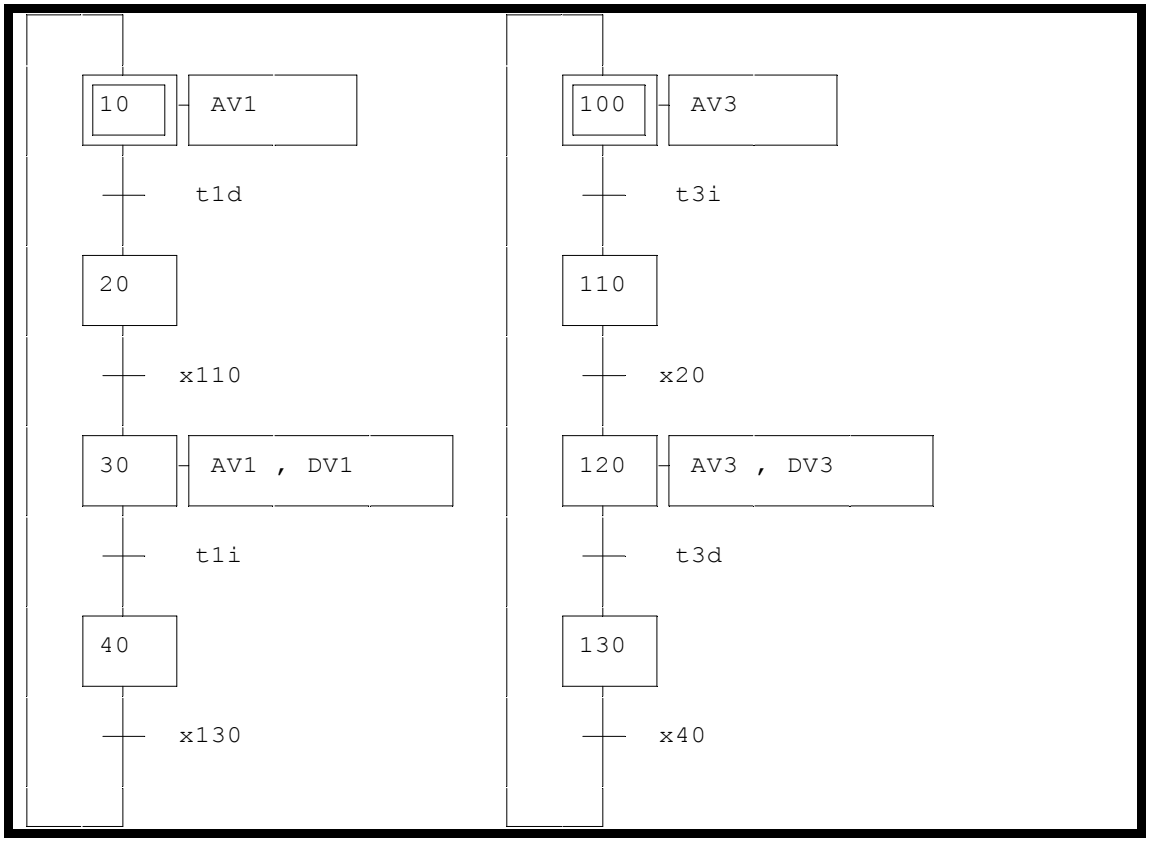
Let's return to a previous example to illustrate Grafcets synchronization.

Conditions:

Round trip of two locomotives on tracks 1 and 3 with a delay at the end of the track.

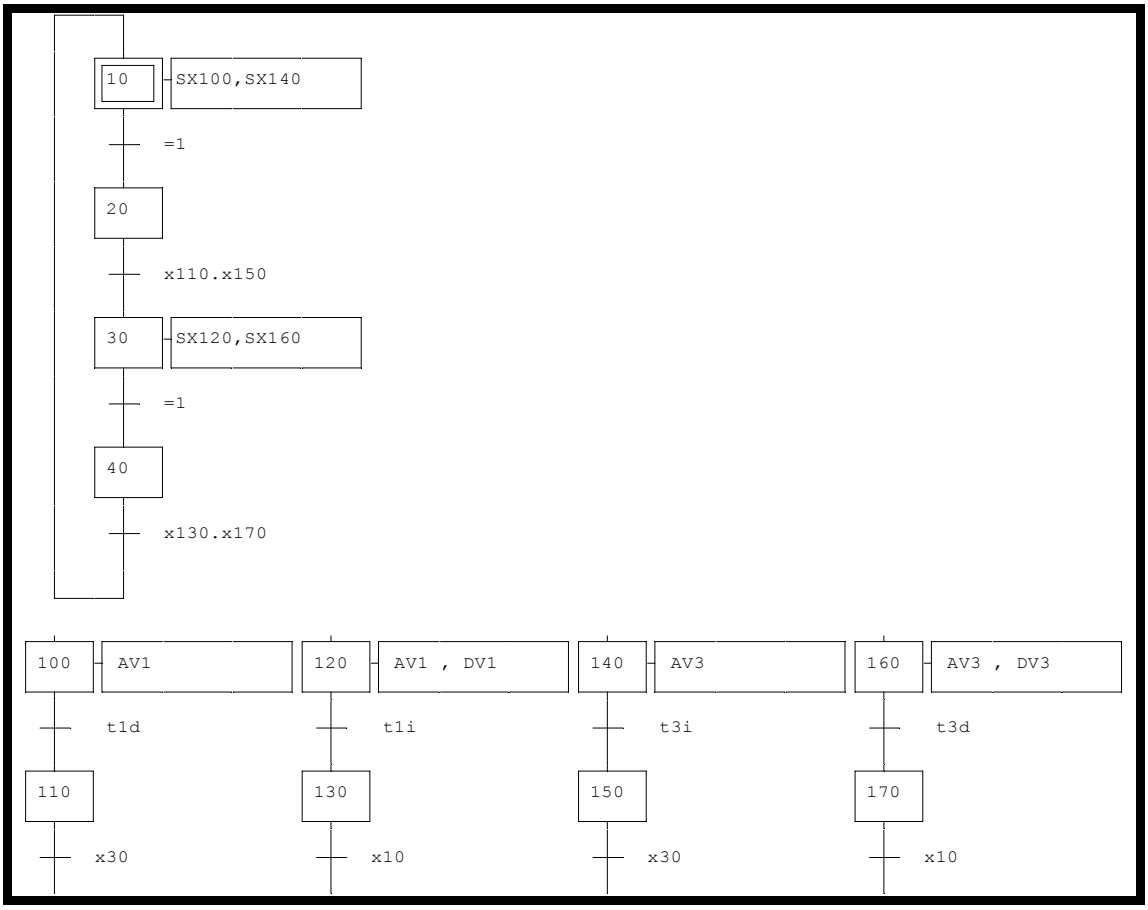
This example was used with a divergence in « And ».

Solution 1 :



example\grafcet\synchro1.agn

Solution 2 :



example\grafcet\synchro2.agn

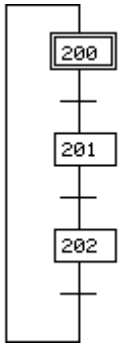
The second is an excellent example of how to complicate the simplest things for teaching purposes.

1.6.7. Grafcet setting

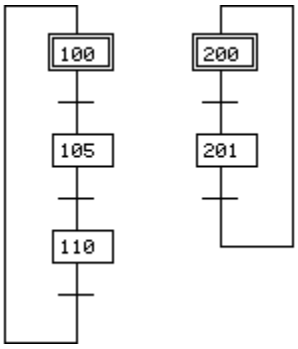
The compiler regroups the steps based on the links established within them. To draw a Grafcet, just refer to one of the steps making up that Grafcet.

It is also possible to draw all of the Grafcets present on a sheet by mentioning the name of the sheet..

For example:



To draw a Grafcet we use Grafcet 200, Grafcet 201 or Grafcet 202.
Thus the Grafcet of all the steps becomes a structured type variable. made up of n steps, each of these steps, being either active or idle.
As we have seen, AUTOMGEN divides the steps into independent groups. These groups can be regrouped, making it possible to consider them as a single Grafcet. To regroup multiple Grafcets, the compilation command « `#G:g1,g2` » (command to be included in a comment) must be used. This command regroups the Grafcets g1 and g2. Remember that the designation of a Grafcet is affected by mentioning the number of one of its steps.
Here is an example :
#G:105,200
this compilation command regroups the two Grafcets:



Note :multiple « #G » commands can be used to regroup more than two Grafcets.
We are now going to describe the useable setting orders. They are simply written inside the action rectangles as normal assignments. They also support the operator S(set to one), R(reset), N(complement assignment) and I(Inversion) as well as conditional actions.

1.6.7.1. Grafcet setting according to a list of active steps

Syntax:

« F<Grafcet>:{<list of active steps>} »

or

 « F/<sheet name>:{<list of active steps>} »

The Grafcet/s thus designated will be set to the state established for the list of active steps if they are within braces. If multiple steps need to be active they need to be separated with a « , » (comma). If the Grafcet/s need to be set to an idle state (not active step) then no step should be present within the braces.

 The number of steps may be preceded by an « X ». It is also possible to associate a symbol to the name of a step.

Examples :

« F10:{0} »

set all the steps of Grafcet 10 to 0 except step 0 which will be set to 1.

« F0:{4,8,9,15} »

sets all the steps of Grafcet 0 to 0 except steps 4,8,9 and 15 which will be set to 1.

« F/normal run :{} »

sets all the Grafcets on the « normal run » sheet to an idle state.

1.6.7.2. Memorization of a Grafcet state

Current state of a Grafcet:

Syntax:

« G<Grafcet>:<bit N°> »

or

 « G/<sheet name>:<bit N°> »

This command memorizes the state of one or more Grafcets in a series of bits. It is necessary to reserve a storage space for the state of the Grafcet/s (one bit per step). These storage bits must be consecutive. You must use the #B command to reserve a linear bit space.

 The step number designating the Grafcet can be preceded by an « X » . It is also possible to associate a symbol to a step name. The bit number can be preceded by « U » or « B ». A symbol can be associated to the first bit of the state storage area.

Particular Grafcet state :
Syntax:
« G<Grafcet>:<Bit N°> {list of active steps} »
or
 « G/<sheet name> :<Bit N°> {list of active steps} »
This command memorizes the state set for the list of active steps applied to the specified Grafcets starting with the indicated bit. Also here it is necessary to reserve a sufficient number of bits. If an idle situation needs to be memorized then no steps should appear between the braces.

 The step number can be preceded by an « X » . It is also possible to associate a symbol to a step name. The bit number can be preceded by « U » or « B ». A symbol can be associated to the first bit of the state storage area.
Examples:
« G0:100 »
memorizes the current state of Grafcet 0 starting from U100.
« G0:U200 »
memorizes the idle state of Grafcet 0 starting from U200.
« G10:150{1,2} »
memorizes the state of Grafcet 10, in which only steps 1 and 2 are active, starting from U150.
« G/PRODUCTION :_SAVE PRODUCTION STATE_ »
memorizes the state of the Grafcets on the « PRODUCTION » spreadsheet in the_SAVE PRODUCTION STATE_ variable.

1.6.7.3. Setting a Grafcet from a memorized state

Syntax:

« F<Grafcet>:<Bit N°> »

or

 « F/<sheet name>:<Bit N°> »

Sets the Grafcet/s with a memorized state to start from the specified bit.

The step number designated by the Grafcet can be preceded by an « X » . It is also possible to associate a symbol to a step name. The bit number can be preceded by « U » or « B ». A symbol can be associated to the first bit of the state storage area.

Example:

« G0:100 »

memorizes the current state of Grafcet 0

« F0:100 »

and resets that state

1.6.7.4. Fixing a Grafcet

Syntax:

« F<Grafcet> »

or

 « F/<sheet name> »

Fixes a Grafcet/s : no evolution of these is permitted.

Example :

« F100 »

fixes Grafcet 100

« F/production »

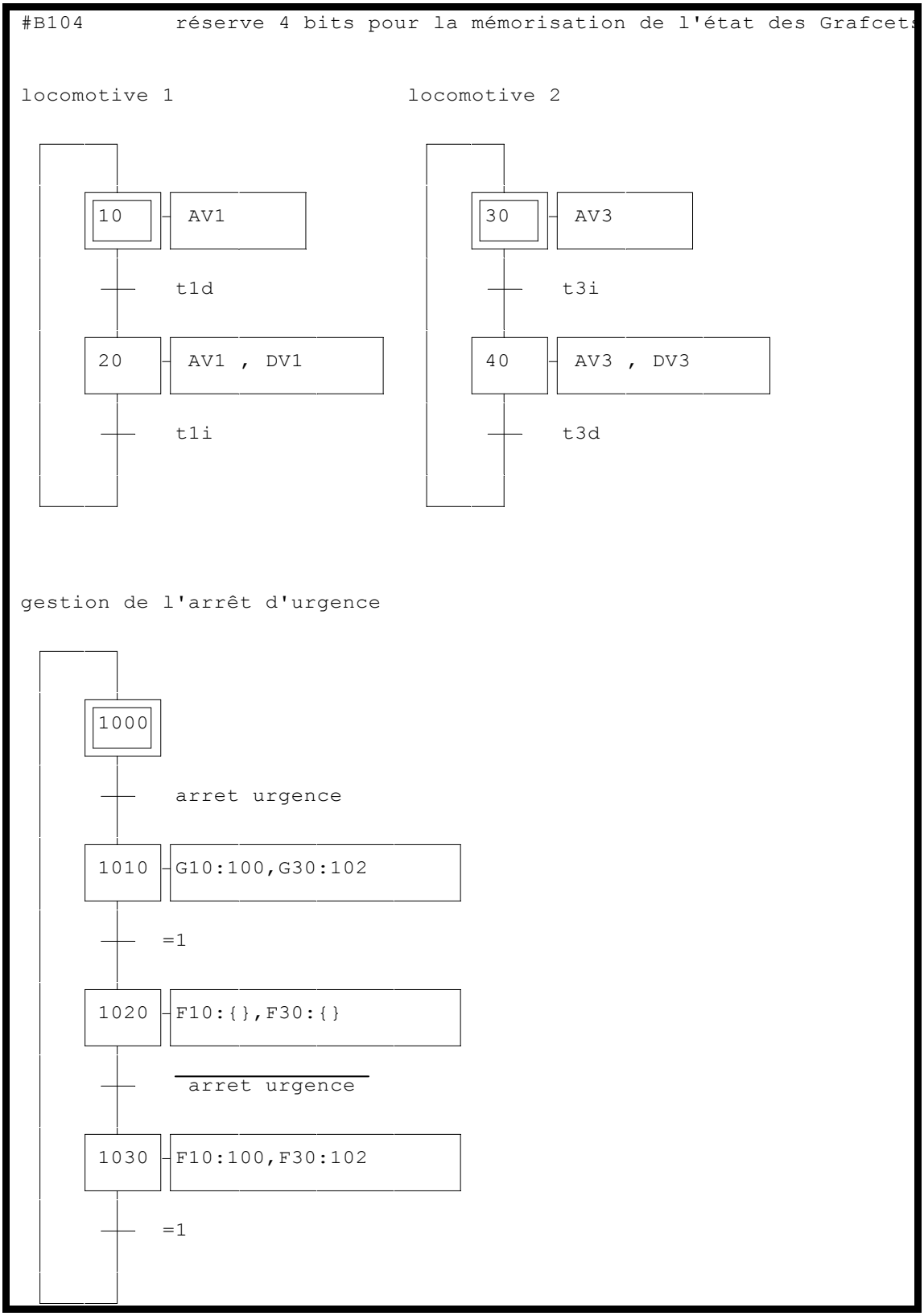
fixes the Grafcets on the « production » sheet

An illustration of setting is shown in the example below.

Conditions :

Let's look at a previous example: the round trip of two locomotives on tracks 1 and 3 (this time with no delay between the locomotives) and let's add an emergency stop. When the emergency stop is detected all the outputs are cleared. When the emergency stop disappears the program should start where it stopped.

Solution 1 :

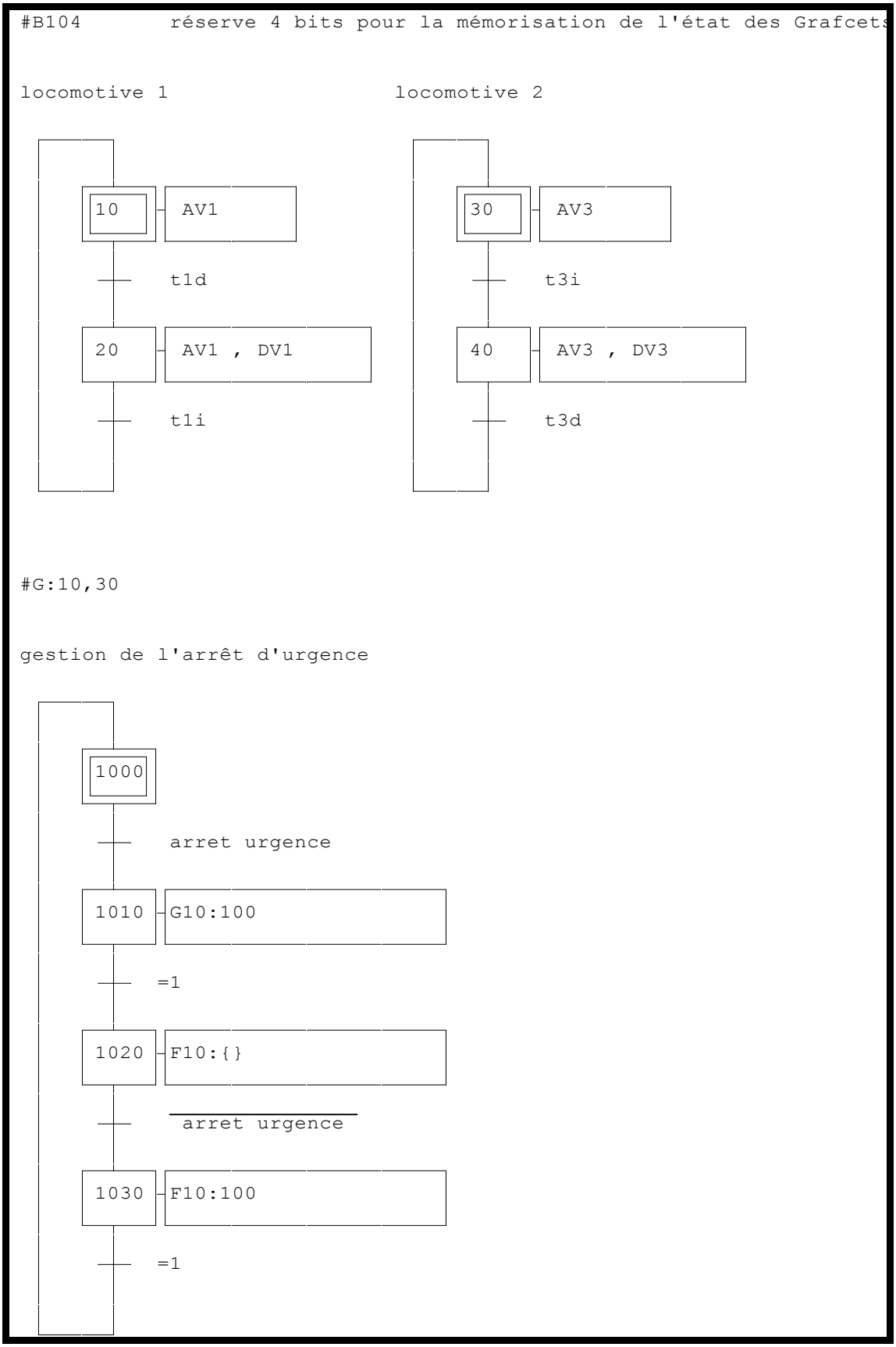


example\grafcet\forçage1.agn

Note the use of command #B104 which makes it possible to reserve four consecutive bits (U100 to U103) to memorize the state of the two Grafcets.

« _emergency stop_ » was associated to a bit (U1000). Its state can thus be modified starting from the environment by clicking below when the dynamic display is active.

Solution 2 :



example\grafcet\forçage2.agn

This second solution shows the use of the compilation command « #G » which makes it possible to regroup the Grafcets with setting command.

1.6.8. Macro-steps

AUTOMGEN implements macro-steps.

Additional information is given below :

A macro-step MS is the single representation of single group of steps and transitions called « MS expansion».

A macro-step obeys the following rules :

- ⇨ an MS expansion involves a special step called **input step** and a special step called **output step**.
- ⇨ the input step has the following property : complete clearing of a transition upstream from the macro-step, it activates the input step of its expansion.
- ⇨ the output step has the following property : it is involved in the validation of transitions downstream from the macro-step.
- ⇨ if outside the transitions upstream and downstream from the MS, there is no input structural connection, on one side with a step or transition of the MS expansion and on the other side, a step or a transition is not part of MS.

The use of a macro-step with AUTOMGEN is set as follows :

- ⇨ the expansion of a macro-step is a Grafcet if it is on a distinct sheet,
- ⇨ the input step of the macro-step expansion must bear the number 0 or the reference Exxx, (with xxx = any number),
- ⇨ the output step of a macro-step expansion must bear the number 9999 or the reference Sxxx, with xxx = any number,
- ⇨ aside from these two requirements, a macro-step expansion can be any Grafcet and as such can contain macro-steps.

1.6.8.1. How can a macro-step be set ?

The symbol  must be used. To obtain this symbol, click on an empty space on the sheet and select « Add …/Macro-step » from the menu. To open the menu click on the bottom of the sheet with the right side of the mouse.

To set a macro-step expansion, create a sheet, designate the expansion and assign the sheet properties (by clicking with the right side of the mouse on the name of the sheet in the browser). Record the type of sheet on «Macro-step expansion » and the number of the macro-step.

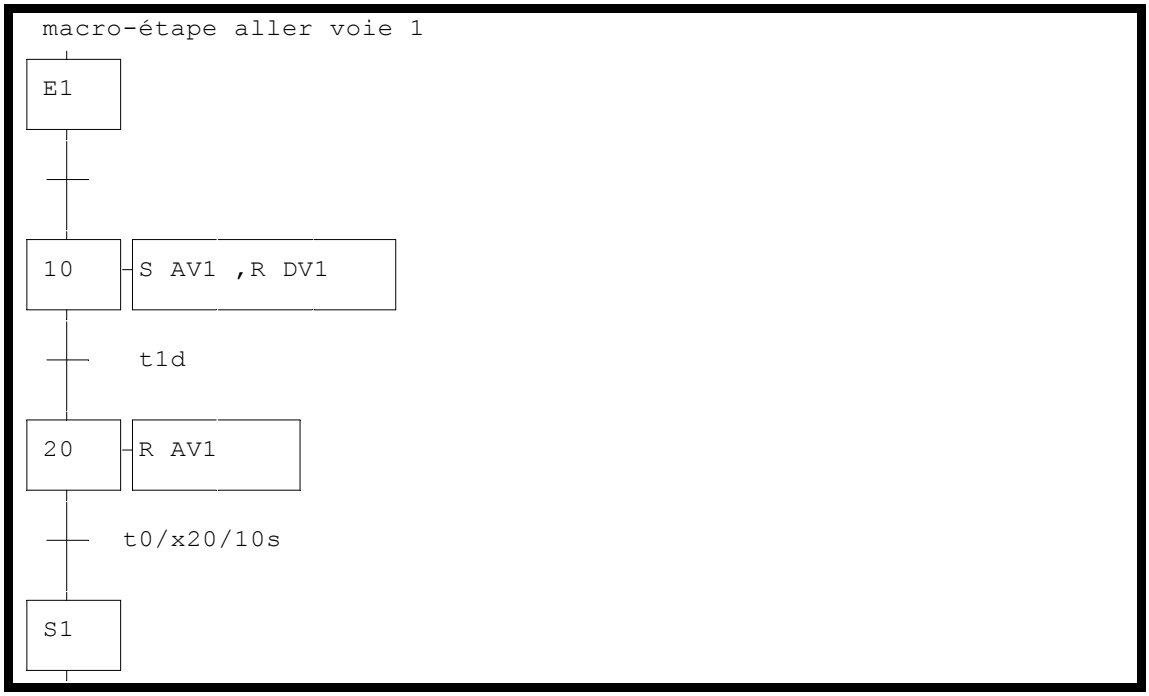
In run mode it is possible to display a macro-step expansion. To do so place the cursor on the macro-step and click on the left side of the mouse.

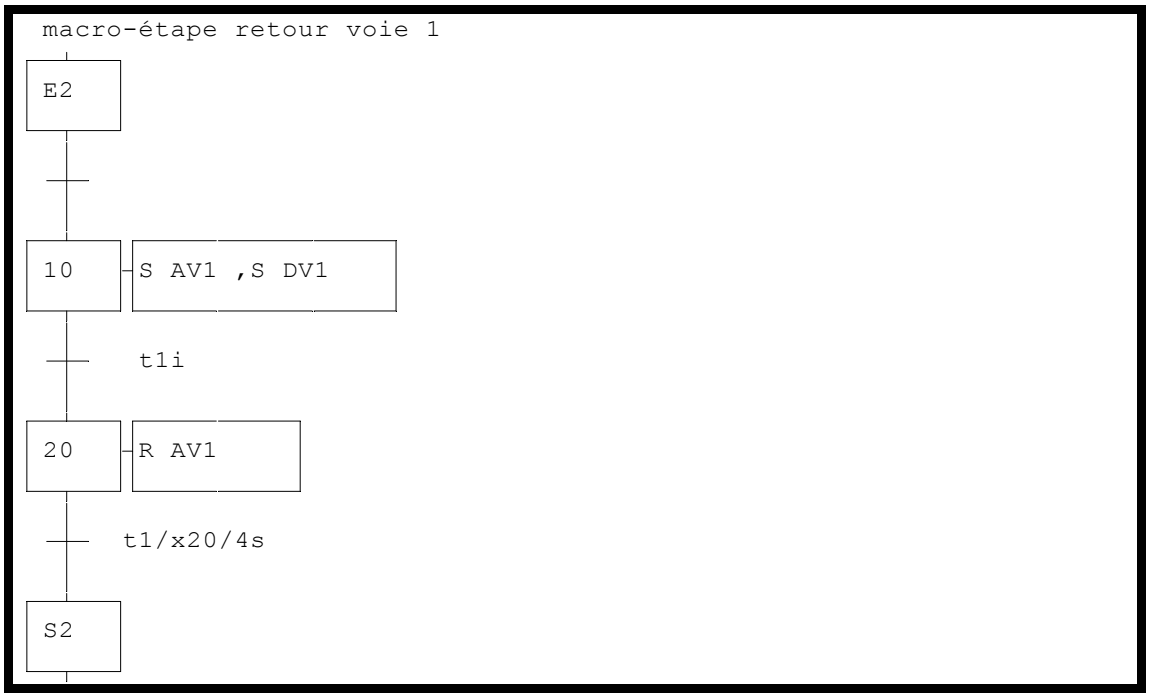
Notes :

- ⇨ user steps and bits used in a macro-step expansion are local, this means that they have no connection with the steps and bits of other Grafcets. All the other types of variables do not have this characteristic: they are common for all levels.
- ⇨ if an area of bits needs to be used in an overall method it is necessary to state this using the command « #B ».
- ⇨ assignment of non-local variables for different levels or different expansions is not managed by the system. In other words, it is necessary to use the assignments « S » « R » or « I » to ensure that the system operates correctly..

Let's use one of our previous examples to illustrate the use of macro-steps: a round trip voyage of a train on track 1 with a delay at the end of the track. We have broken down the legs of the trip into two separate macro-steps.

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

example\grafcet\macro-étape.agn

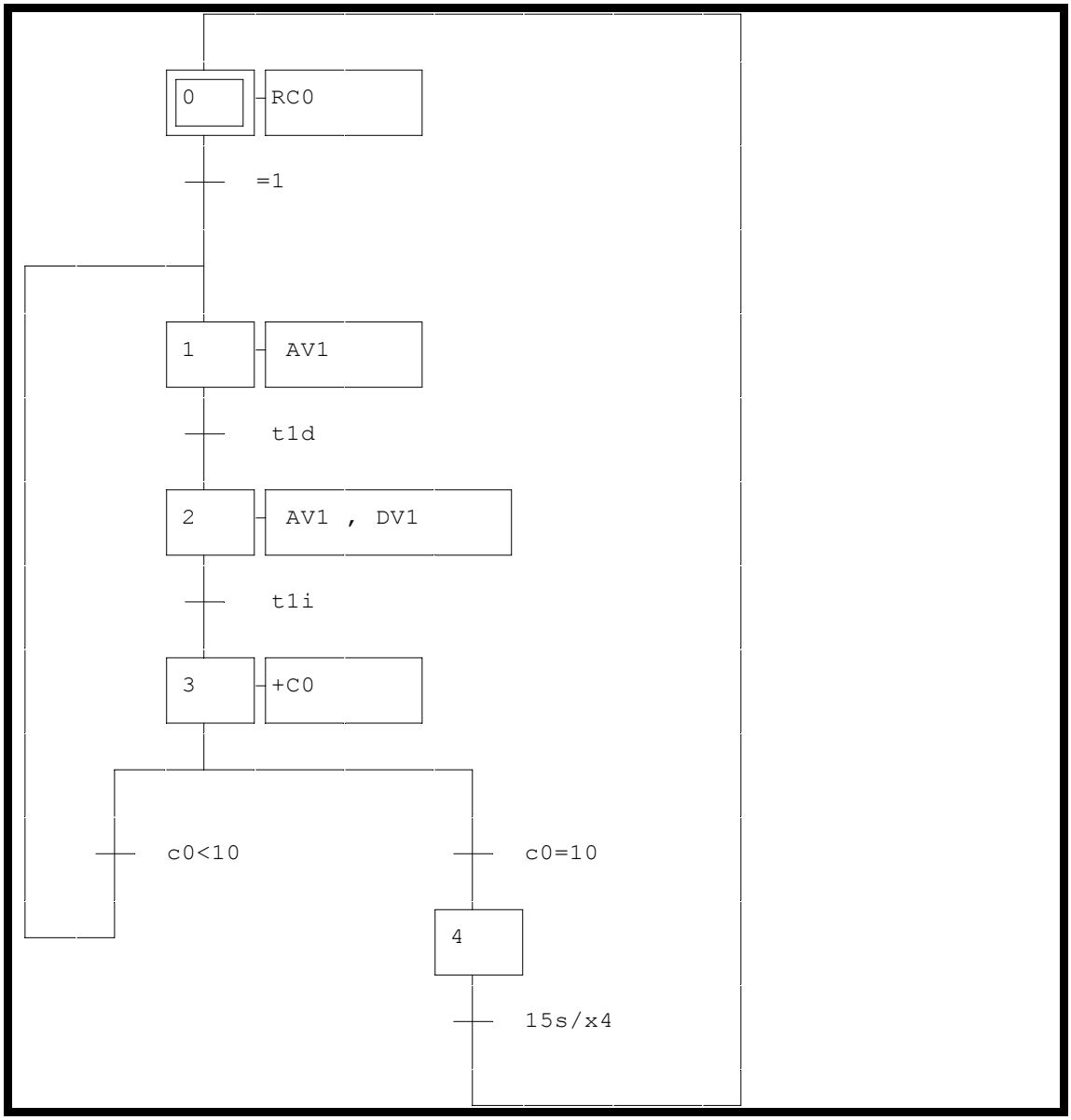
1.6.9. Counters

We are going to use an example to describe the use of counters.

Conditions :

A locomotive must make 10 round trip journeys on track 1, stop for fifteen seconds and start again.

Solution :

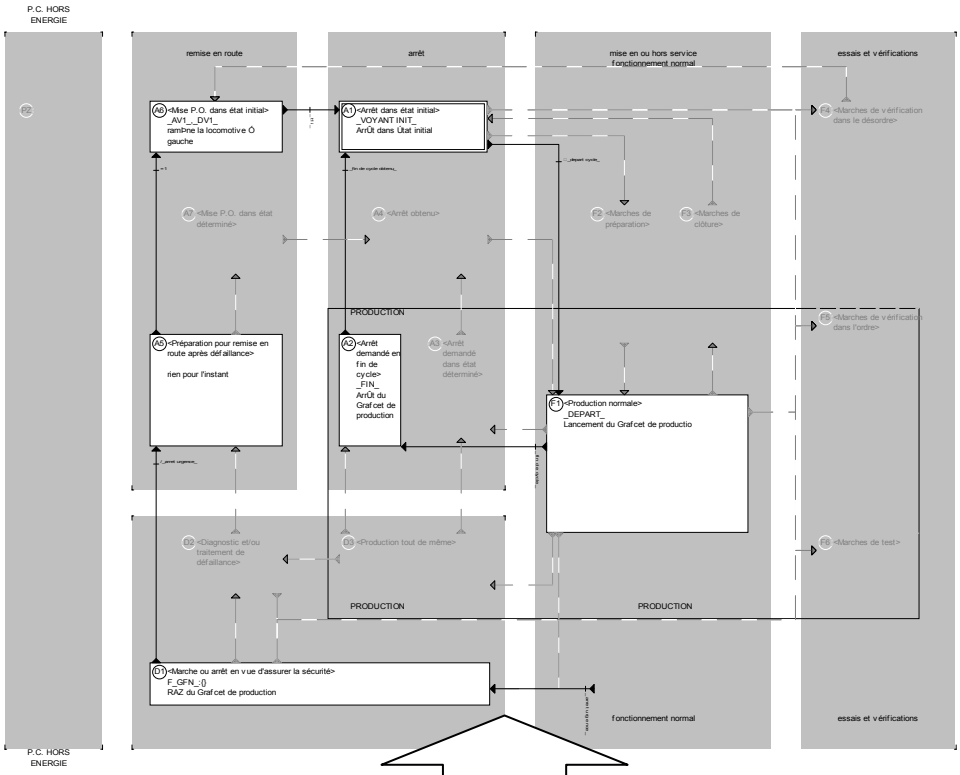


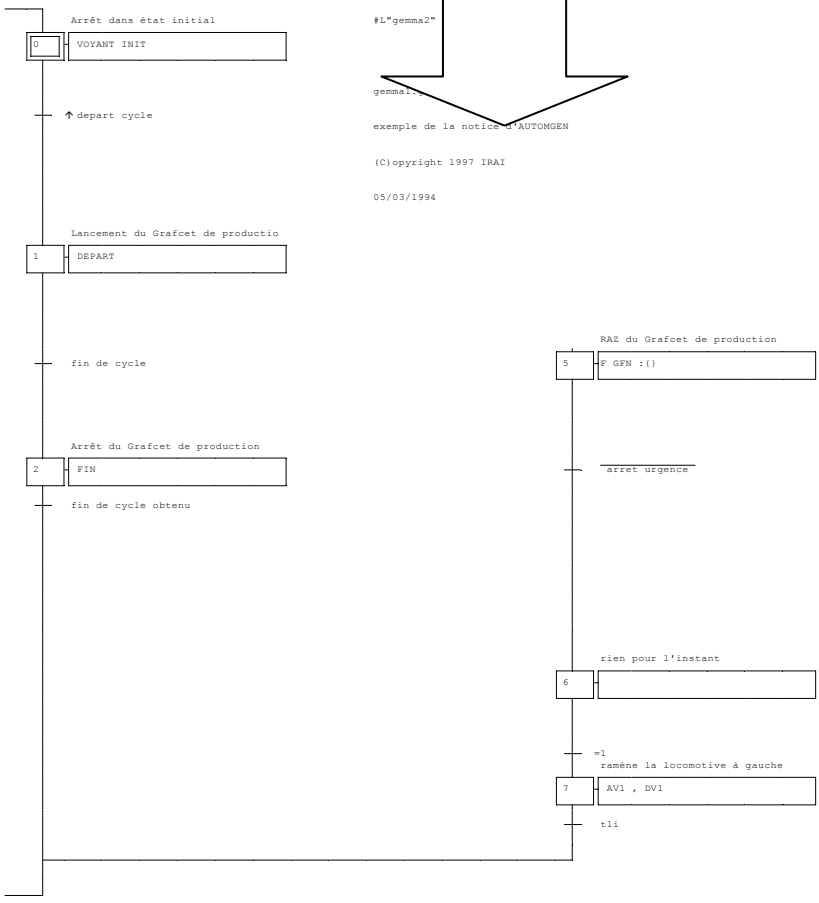
Example\grafcet\compteur.agn

1.7. Gemma

AUTOMGEN implements the Grafcet description of run mode management in a Gemma form. The main feature is an editing method open to the Grafcet mode. It is possible to go from the Grafcet editing mode to the Gemma editing mode. The translation of a Gemma into a Grafcet run mode management is therefore automatic and immediate..

The command « Editing Gemma » in the « Toolbar» makes it possible to pass from one mode to the other.


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

1.7.1. Creating a Gemma

To create gemma proceed as follows :

⇨ click on « Sheet » on the browser with the right side of the mouse and select the command « Add a new sheet»,

⇨ from the list of sizes select « Gemma »,

⇨ click on « OK »,

⇨ use the right side of the mouse to click on the sheet name created on the browser,

⇨ select properties « Proprieties » from the menu,

⇨ check « Display Gemma form ».

The window will contain a Gemma where all the links are gray. To validate a rectangle or a connection click on it with the right side of the mouse.

To edit the contents of a rectangle or the type of connection click on it with the left side of the mouse.

The contents of Gemma rectangles will be placed in the Grafcet action rectangles. The type of connection will be placed in the Grafcet transitions.

1.7.2. Content of Gemma rectangles

Gemma rectangles can receive any action used by Grafcet. Because this involves setting a structure for managing run and stop modes, it is a good idea to use the lowest level setting orders for Grafcet, see chapter 1.6.7. .

1.7.3. Obtaining a corresponding Grafcet

Check "Display Gemma form" again in sheet properties to call up a Grafcet representation. It is always possible to call up a Gemma representation because the Grafcet structure has not been changed. The transitions, the action rectangle contents and comments can be edited with automatic updating of Gemma.

1.7.3.1. Deleting blank spaces in Grafcet

It is possible that the obtained Grafcet occupies more space than necessary on the page. The command « Change page layout » from the « Tools » menu makes it possible to eliminate all the unused spaces.

1.7.4. Printing Gemma

When editing is in Gemma mode use the « Print » command to print the Gemma.

1.7.5. Exporting Gemma

Use the « Copy to EMF format » in the « Editing » menu to export a Gemma to a vectorial form.

1.7.6. Example of Gemma

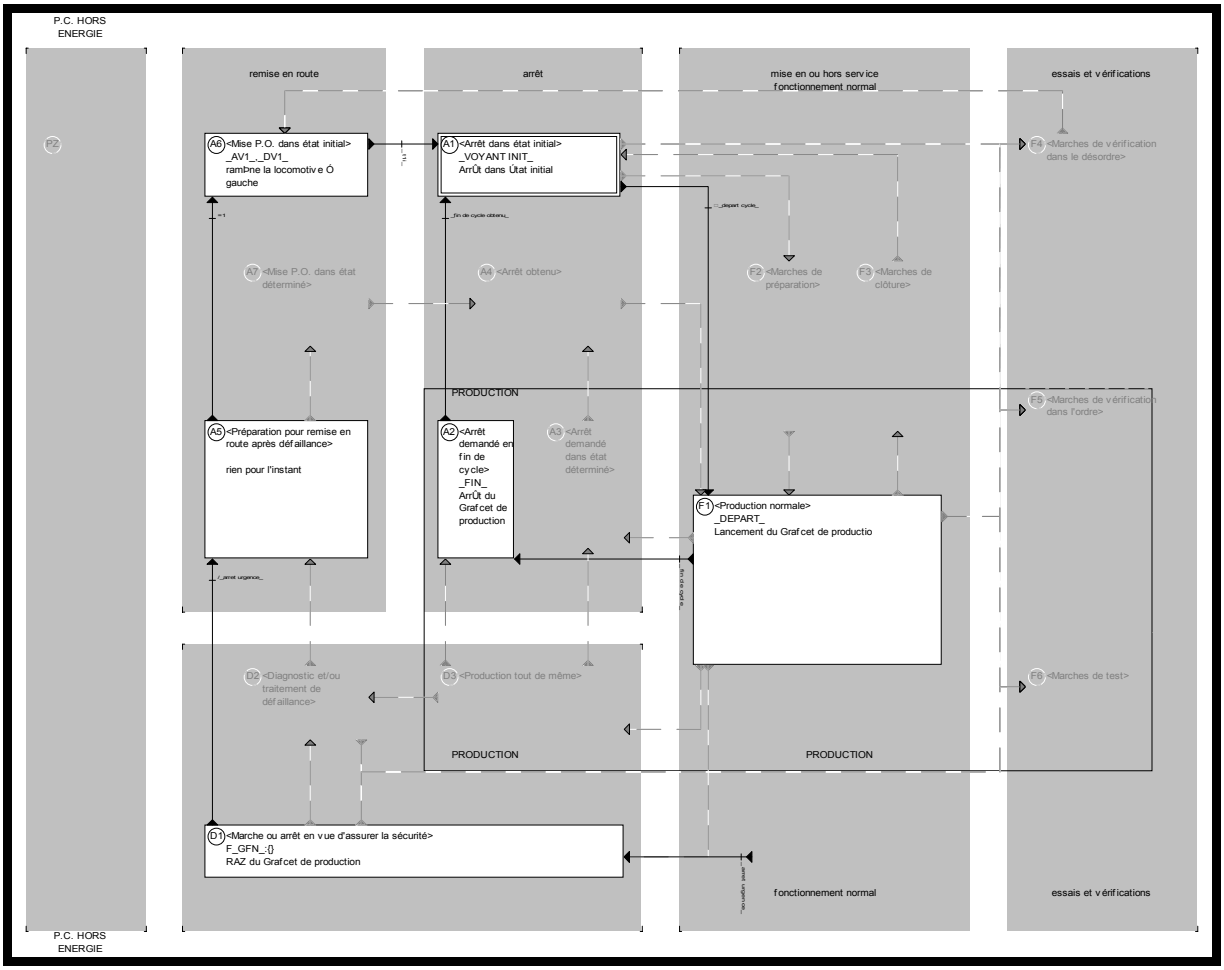
A description of how to use Gemma is below..

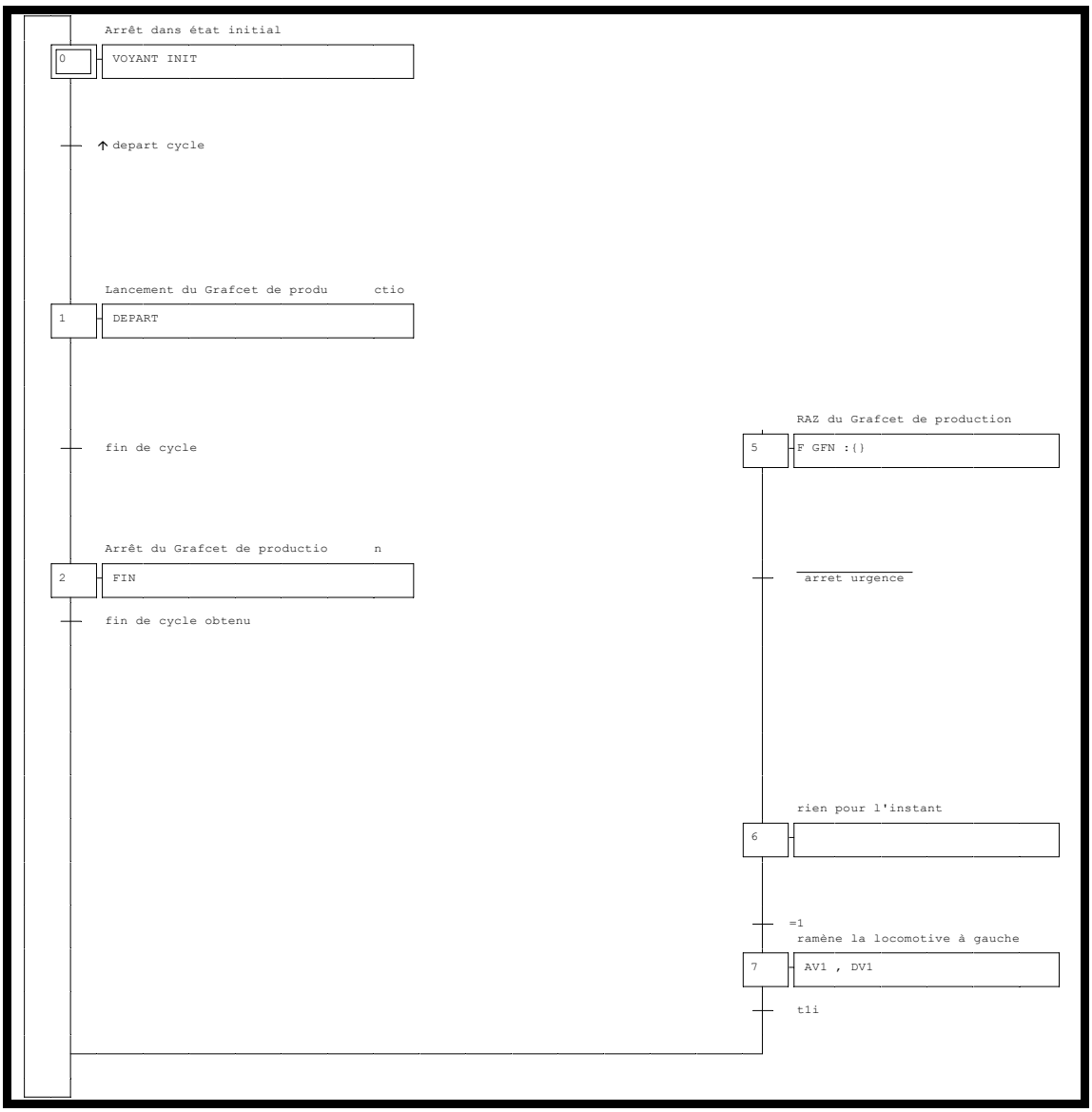
Conditions :

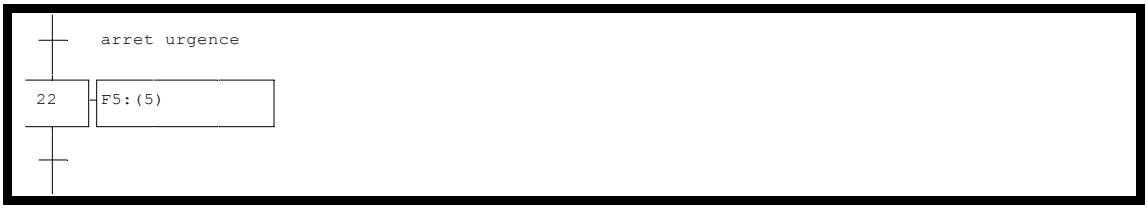
Imagine a panel with the following pushbuttons : « start cycle », « end cycle » and « emergency stop » a light « INIT ».

The main program will consist of a locomotive making round trip journeys on track 1.

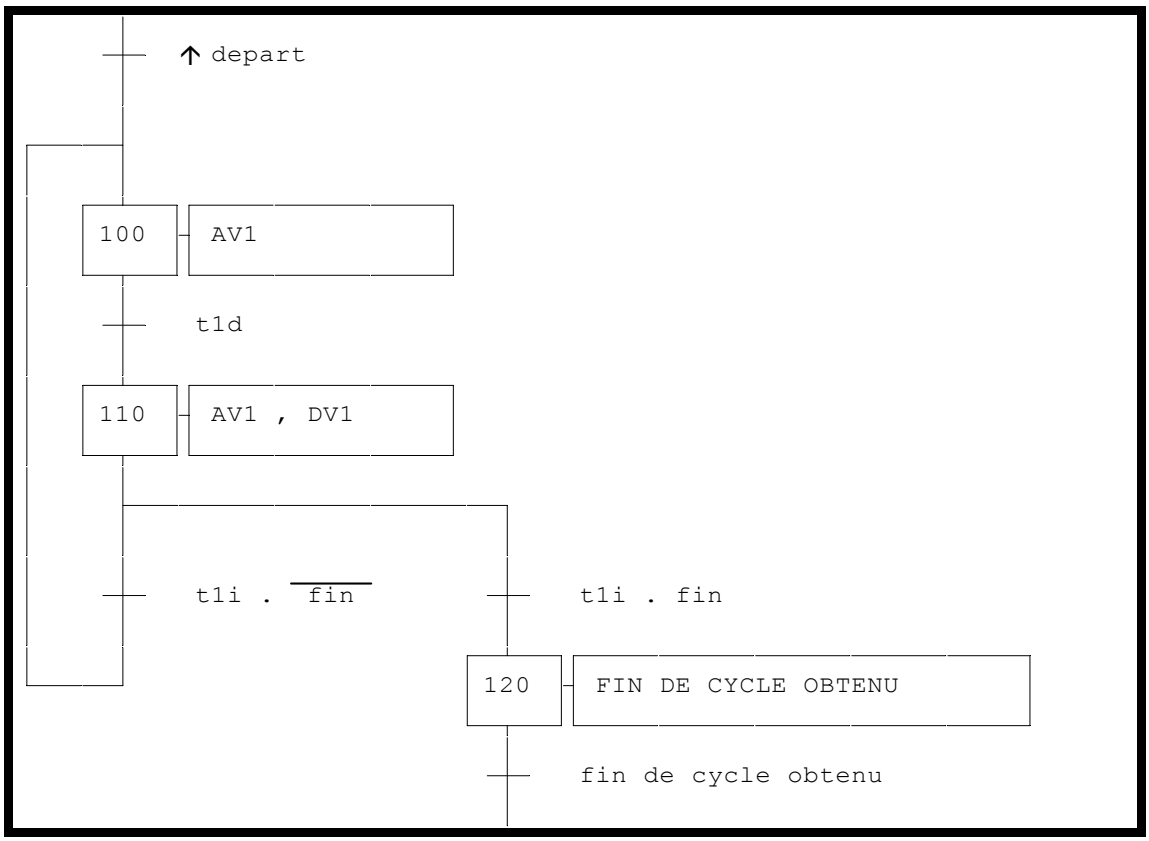
Solution :


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_ 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





(editing with Grafcet form)



example\gemma\gemma.agn

1.8. Ladder

Ladder language, also called contact model, is for graphically describing boolean equations. To create a logical function « And » it is necessary to write contacts in series. To write an « Or » function it is necessary to write contacts in parallel.

« And » function

« Or » function

The content of contacts must comply with the syntax established for the tests which is explained in the «Common elements» chapter of this manual.

The content of the coils must comply with the syntax established for the actions which is also explained in the «Common elements » chapter of this manual.

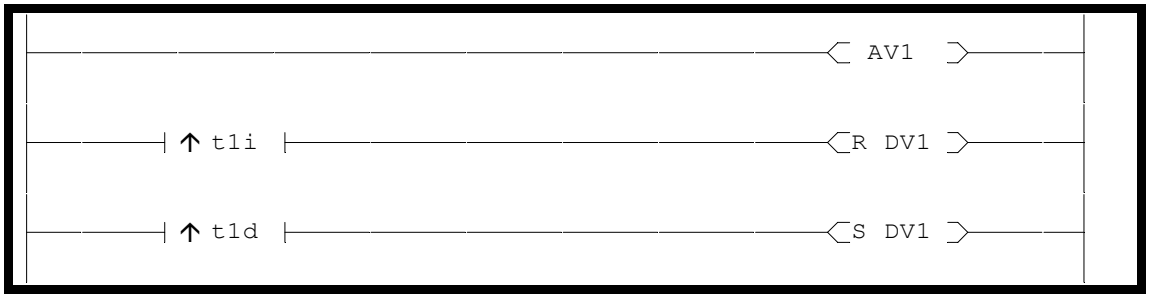
1.8.1. Example of Ladder

Let's start with the simplest example.

Conditions :

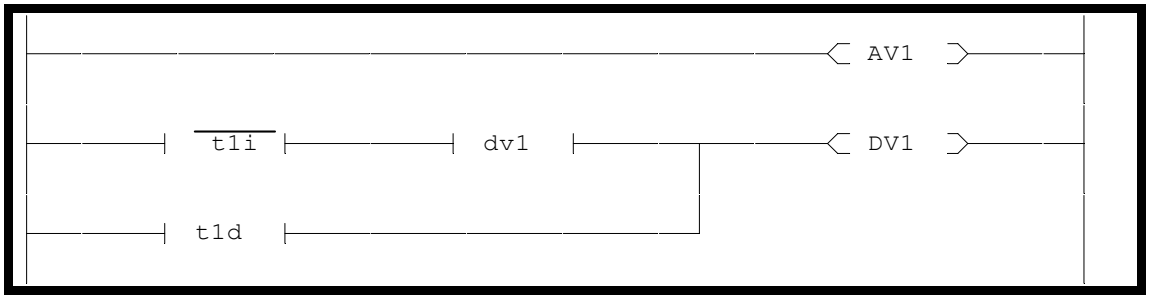
Round trip of a locomotive on track 1.

Solution 1 :



Example\ladder\ladder1.agn

Solution 2 :



Example\ladder\ladder2.agn

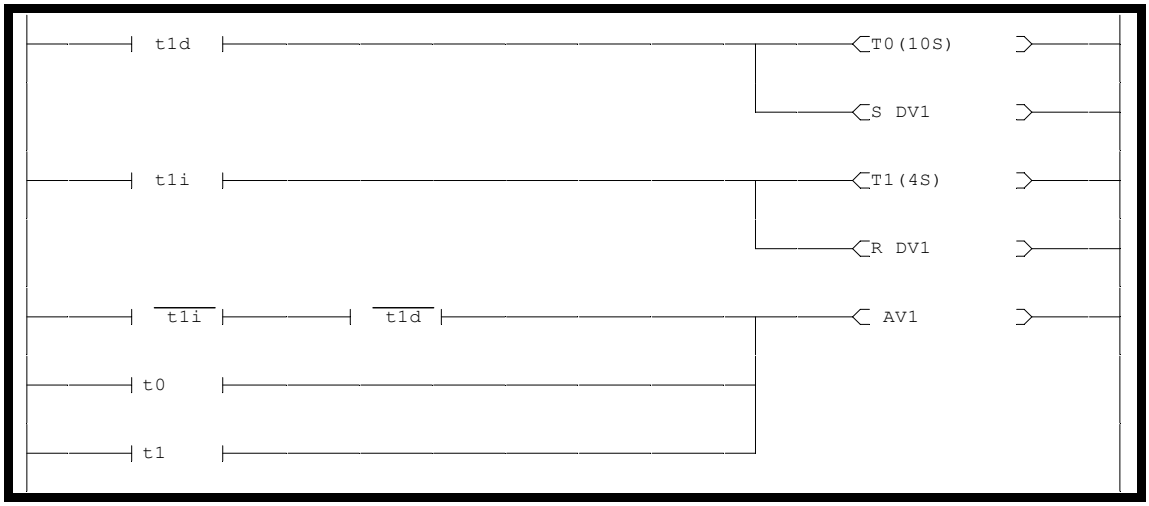
The second solution is identical from an operational point of view. It is used to display the use of a self-controlled variable.

Let's make our example more complex.

Conditions :

The locomotive must stop for 10 seconds to the right of track 1 and 4 seconds to the left.

Solution :



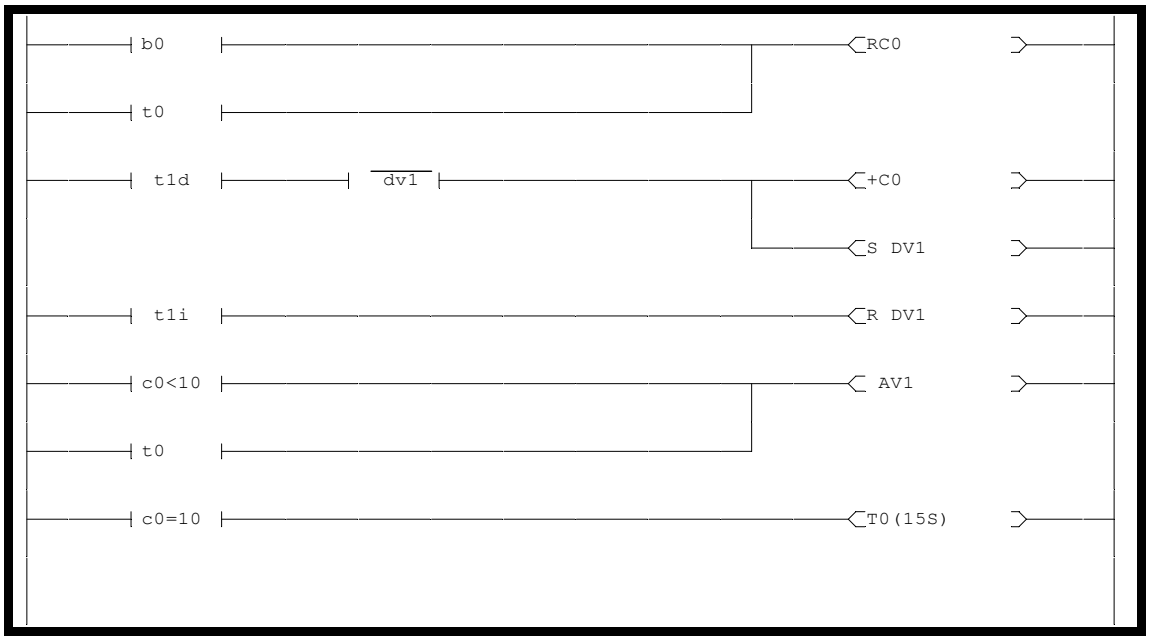
Example\ladder\ladder3.agn

A final example, even a little more complicated

Conditions:

Again a locomotive which makes round trips on track 1. For each 10 round trips it must stop for 15 seconds.

Solution :



Example\ladder\ladder4.agn

1.9. Flow chart

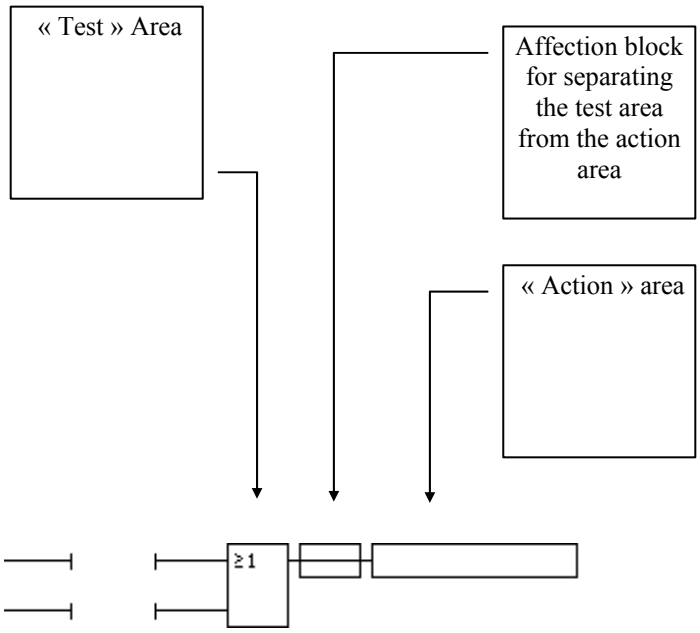
AUTOMGEN implements flow chart language in the following way :

⇨ use of a special block called « assignment block », this block separates the action area and test area, it has the following form  and is associated with key [0] (zero),

⇨ it uses the functions« No », « And » and « Or »,

⇨ it uses action rectangles to the right of the action block.

Flow chart language is used for graphically writing boolean equations.
The test content must comply with the syntax established in the « Common elements » chapter in this manual.
The content of action rectangles must comply with the syntax for actions, also described in the « Common elements » chapter of this manual.



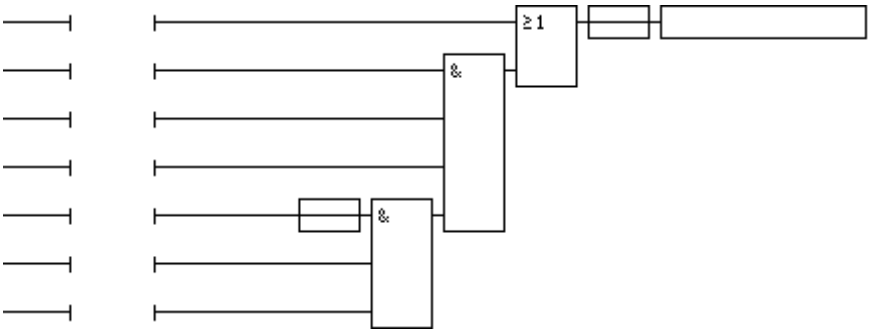
1.9.1. Drawing flow charts

1.9.1.1. Number of input of functions « And » and « Or »

The « And » and « Or » functions are respectively composed of a block  (key [2]) or a block  (key [3]), and possible blocks  (key [4]) for adding inputs to blocks and finally block  (key [5]).
The functions « And » and « Or » thus involve a minimum of two inputs..

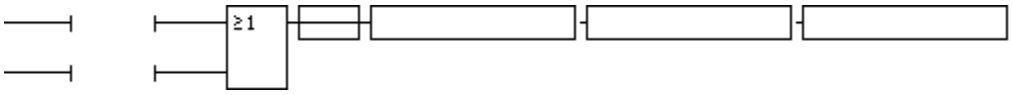
1.9.1.2. Chaining the functions

The functions can be chained.

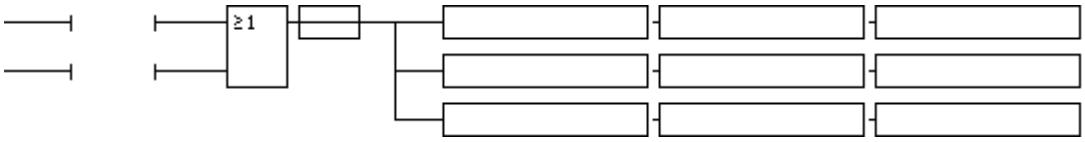


1.9.1.3. Multiple actions

Multiple action rectangles can be associated to a flow chart after the assignment block..



or



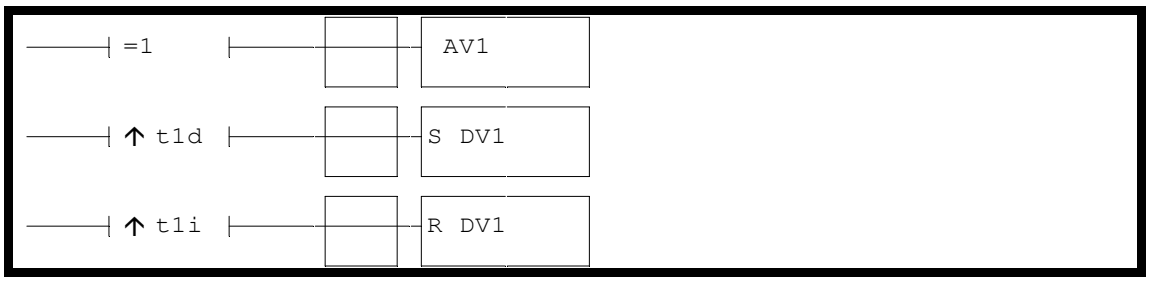
1.9.2. Example of a flow chart

Let's start with the simplest example:

Conditions :

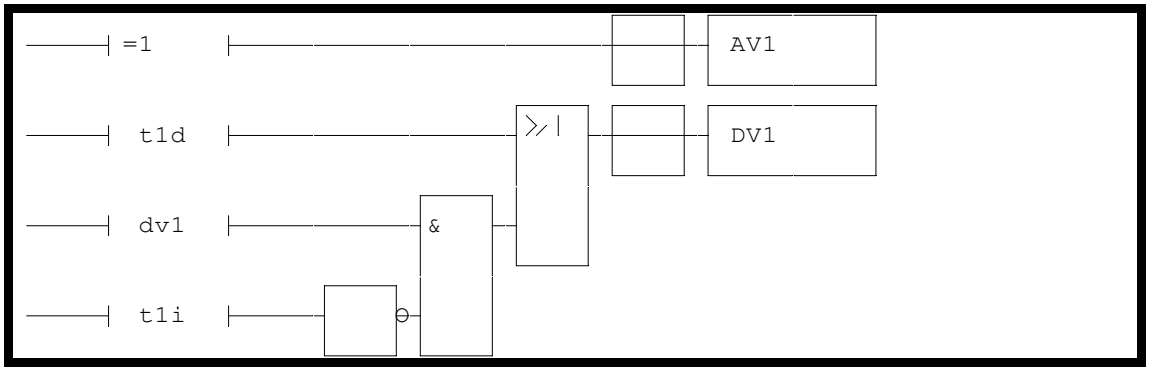
Roundtrip of a locomotive on track 1.

Solution 1 :



Example\logigramme\logigramme1.agn

Solution 2 :



Example\logigramme\logigramme2.agn

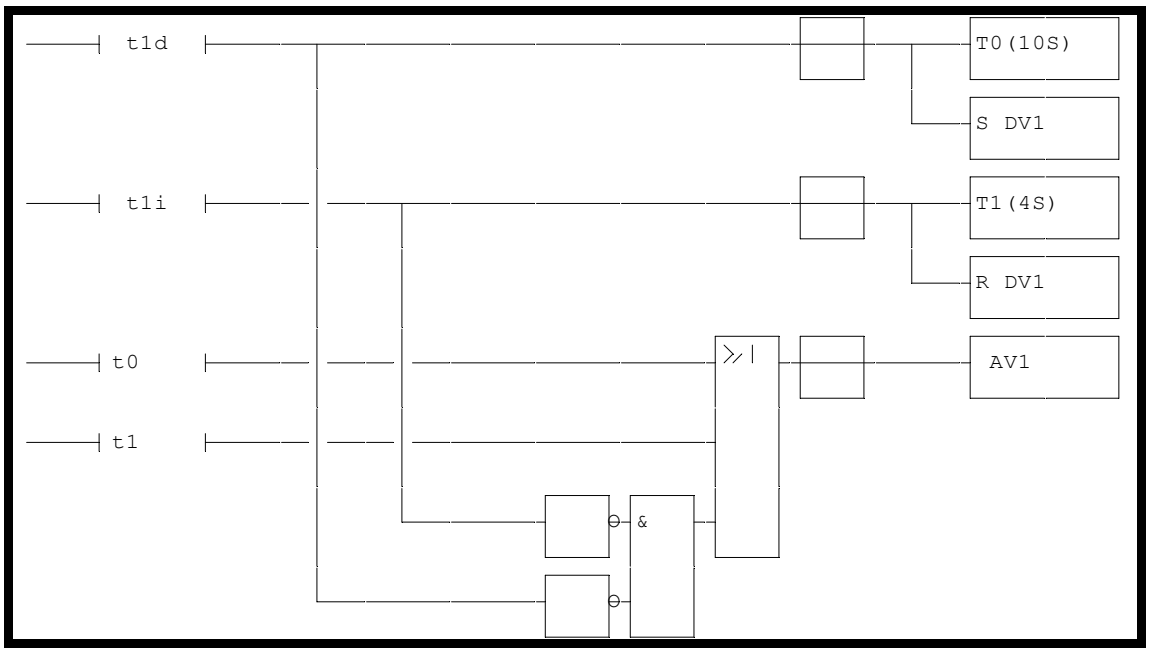
The second solution is identical from an operational point of view. It is used to display the use of a self-controlled variable.

Let's make the example more complex.

Conditions :

The locomotive must stop for 10 seconds to the right of track 1 and 4 seconds to the left.

Solution :



Example\logigramme\logigramme3.agn

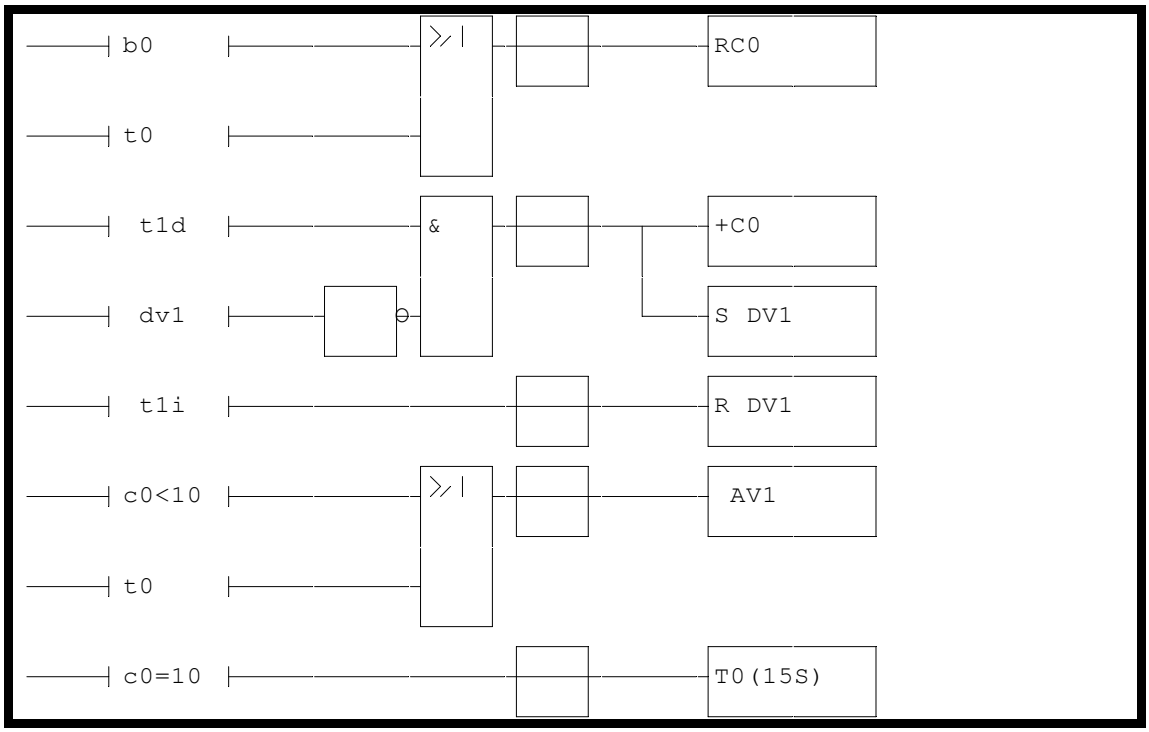
Note the reuse of the «And» block in the lower part of the example towards the inputs « _t1d_ » and « _t1i_ ». This prevents having to write the two tests a second time.

A final example a bit more complicated.

Conditions :

Again a locomotive which makes round trips on track 1. Each 10 round trips it must stop for 15 seconds.

Solution :



Example\logigramme\logigramme4.agn

1.10. Literal languages

This chapter describes the use of the three forms of literal language which are available in AUTOMGEN :

⇨ low level literal language,

⇨ extended literal language,

⇨ IEC 1131-3 standard ST literal language

1.10.1. How is a literal language used?

Literal language can be used in the following forms :

⇨ code files associated to an action (Grafcet, Ladder, flow chart),

⇨ code boxes associated to an action (Grafcet, flow chart),

⇨ literal code in action rectangle or coil (Grafcet, Ladder, flow chart),

⇨ code boxes used in the form of an organizational chart (see the «Organizational chart » chapter),

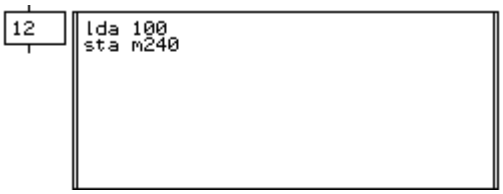
⇨ code files which support the function block functionality (see the « Function blocks » chapter),

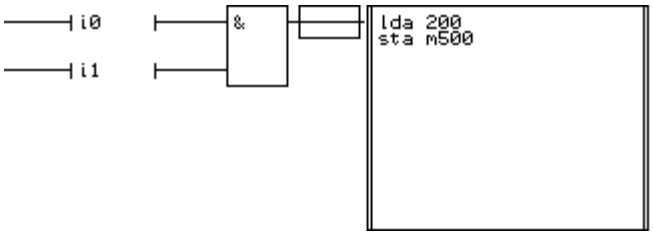
⇨ code files which support a macro-instruction functionality see chapter 1.10.4. Macro-instruction.

1.10.1.1. Code box associated with a step or flow chart

A code box associated with an action is for being able to write lines of literal language on an application page.

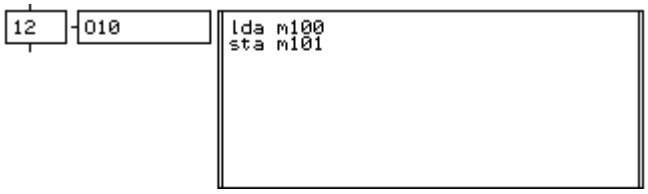
Examples :





The code used above is scanned as long as the action is true.
It is possible to use the action rectangles and code boxes together.

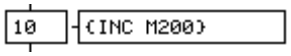
Example :

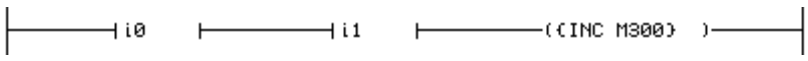


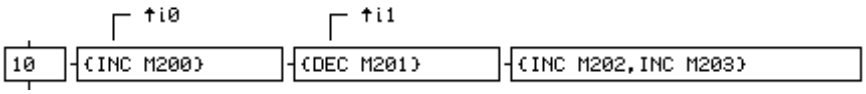
1.10.1.2. Literal code in an action rectangle or coil

The characters « { » and « } » are used to directly enter instructions in literal language into an action rectangle (Grafcet and flow chart languages). The character« , » (comma) is used as a separator if multiple instructions are present in « { » and « } ».
This type of entry can be used with conditional orders.

Examples :







1.10.2. Setting a code box

To create a code box, follow the steps below:

⇨ click on an empty space on the sheet with the right side of the mouse,

⇨ select « Add … / Code box » from the menu,

⇨ click on the edge of the code box to edit its contents.

To exit the code box after editing click on [Enter] or click outside it.

1.10.3. Low level literal language

This chapter describes the use of low level literal language. This language is an intermediate code between the evolved languages of Grafcet, flow chart, ladder, organizational chart, function block, extended literal, ST literal and executable languages. It is also know as pivot code. Post-processors translate low level literal language into executable code for the PC, automate or microprocessor card.
Literal language can also be used for an application in order to effect various boolean, numeric or algebraic operations.
Low level literal language is an assembler type language. It uses the idea of an accumulator for numeric treatment.
Extended literal language and ST literal language described in the following chapters, offer a simplified and higher level alternative for writing programs in literal language.
The general syntax for a line of low level literal language is:
«action » [[[« Test »] « Test »]...]
The actions and tests of low level literal language are represented by mnemonics formed of three letters. An instruction is always followed by an expression: variable, constant etc.
A line is composed of a single action and possibly a test. If a line only includes an action, then the instruction is always executed.

1.10.3.1. Variables

The variables used are the same as those described in the « Common elements » chapter.

1.10.3.2. Accumulators

Some instructions use an accumulator. The accumulators are internal registers which execute the final program and make it possible to temporarily store values.
There are three accumulators : a 16 bit accumulator known as AAA, a 32 bit accumulator known as AAL and a float accumulator known as AAF.

1.10.3.3. Flags

Flags are boolean variables which are positioned based on the result of numeric operations.
There are four permanent flags to test the result of a calculation. These four indicators are:

- ⇨ carry indicator C :it indicates if an operations has generated a carry figure (1) or not (0),
- ⇨ zero indicator Z :it indicates if an operations has generated a nil result (1) or not nil (0),
- ⇨ sign indicator S : it indicates if an operation has generated a negative result (1) or positive one (0),
- ⇨ overflow indicator O : it indicates if an operation has generated an overflow (1).

1.10.3.4. Addressing modes

Low level literal language has 5 addressing modes. An addressing mode is a characteristic associated to each literal language instruction.
Addressing modes used appear below: :

TYPE	SYNTAX	EXAMPLE
Immediate 16 bits	{constant}	100
Immediate 32 bits	{constant}L	100000L
Immediate float	{constant}R	3.14R
Absolute	{variable} {variable reference}	O540
16 bit accumulator	AAA	AAA
32 bit accumulator	AAL	AAL
Float accumulator	AAF	AAF

Indirect	{variable}{(word reference)}	O(220)
Label	:{label name} :	:loop

Thus an instruction has two characteristics: the type of variable and the addressing mode. Certain instructions support or do not support certain addressing modes and certain variable types. For example, an instruction may only apply to two words and not to other types of variables.

Note : Variables X and U can not be associated to an indirect address due to the non-linear nature of their assignments. If it is necessary to access a U variable table then a command #B must be used to make a table of linear bits.

1.10.3.5. Tests

Tests that can be associated to instructions are composed of a mnemonic, a type of test and a variable.

Test mnemonics are used to set combination tests on multiple variables (and, or). If a test is composed of a single variable, an AND operator needs to be associated to it.

There are only three test mnemonics:

AND and

ORR or

EOR end or

Here are some examples of equivalencies in boolean equations and low level literal language :

```
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 to translate directly,
                               ; intermediate variables
                               ; must be used :
```

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

Test modifiers make it possible to test things other than the truth of a variable:

⇨ / no

⇨ # rising edge

⇨ * falling edge

⇨ @ immediate state

Notes :

⇨ boolean variables are updated after each execution cycle. In other words, if a binary variable is positioned at a state during a cycle, then its new state will be detected during the following cycle. The text modifier @ makes it possible to obtain the real state of a boolean variable without waiting for the following cycle.

⇨ test modifiers cannot be used with numeric tests.

Examples:

```
set o100
equ o0 and @o100          ; true test of the first cycle
equ o1 and o100           ; true test at the second cycle
```

Only two addressing modes are available for tests: absolute and indirect
A test for counters, words, longs and floating points is available:

Syntax :

```
« {variable} {=, !, <, >, << , >>} {constant or variable} »
```

= equal,
! different,
< less than not signed,
> greater than not signed,
<< less than signed,
>> greater than signed,

By default, constants are written in decimals. The suffixes « $ » and « % » are used for writing in hexadecimal or binary. The quotation marks are for writing in ASCII.

32 bit constants must be followed by the letter « L ».

Real constants must be followed by the letter « R ».

A word or a counter can be compared to a word, a counter of a 16 bit constant..

A long can be compared to a long or a 32 bit constant.

A float can be compared to a float or a real constant.

Examples :

```
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. Comments

Comments need to start with the character « ; » (semi-colon), all the characters after it are ignored.

1.10.3.7. Numbering base

The values (variable references or constants) can be written in decimal, hexadecimal, binary or ASCII.

The following syntax must be applied for 16 bit constants :

⇨ decimal :possibly the character « - » plus 1 to 5 digits « 0123456789 »,

⇨ hexadecimal : the prefix « $ » or « 16# » followed by 1 to 4 digits « 0123456789ABCDEF »,

⇨ binary : the prefix « % » or « 2# » followed by 1 to 16 digits « 01 »,

⇨ ASCII : the character « " » followed by 1 or 2 characters followed by « " ».

The following syntax must be applied for 32 bit constants:

⇨ Decimal : possibly the character « - » plus 1 to 10 digits « 0123456789 »,

⇨ Hexadecimal : the prefix « $ » or « 16# » followed by 1 to 8 digits « 0123456789ABCDEF »,

⇨ Binary : the prefix « % » or « 2# » followed by 1 to 32 digits « 01 »,

⇨ ASCII : the character « " » followed by 1 to 4 characters followed by « " ».

The following syntax must be applied for real constants :
[-] i [[.d] Esx]
i is the whole part
of a decimal part
s possible sign of an exponent
x possible exponent

1.10.3.8. Presettings

A presetting is used to fix the value of a variable before starting the application.
The variables T or %T, M or %MW, L or %MD and F or %F can be preset.

The syntax is as follows :

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

For time delays the variable must be written in decimal and be included between 0 and 65535.
For words the following syntax must be used :

⇨ Decimal : possibly the character « - » plus 1 to 5 digits « 0123456789 »,

⇨ Hexadecimal : the prefix « $ » or « 16# » followed by 1 to 4 digits « 0123456789ABCDEF »,

⇨ Binary : the prefix « % » or « 2# » followed by 1 to 16 digits « 01 »,

⇨ ASCII : (two characters per word) the character « " » followed by n characters followed by « " »,

⇨ ASCII : (one character per word) the character « ’ » followed by n characters followed by « ’ ».

For longs the following syntax must be used :

⇨ Decimal : possible the character « - » plus 1 to 10 digits « 0123456789 »,

⇨ Hexadecimal : the prefix « $ » or « 16# » followed by 1 to 8 digits « 0123456789ABCDEF »,

⇨ Binary : the character « % » or « 2# » followed by 1 to 32 digits « 01 »,

⇨ ASCII : (four characters per long) the character « " » followed by n characters followed by « " »,

⇨ ASCII : (one character per long) the character « ’ » followed by n characters followed by « ’ »

For floats the value must be written in the following form :

[-] i [[.d] Esx]
i is the whole part
d a possible decimal part
s a possible exponent sign
x a possible exponent

Examples :

```
$t25=100
```

fixes the time delay order 25 at 10 s

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

places the values 100,200,300,400 in the words 200, 201, 202, 203

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

places the string « ABCDEF » starting from m200 (« AB » in m200, « CD » in m201, « EF » in m202)

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

places the string « ABCDEF » starting from m200, each word receives a character

```
$f1000=3.14
```

places the value 3,14 in f1000

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

places the value 5,1 * 10 exponent -15 in %mf100

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

places the value 12345678 (hexa) in the long l200

It is easier to write text in the presettings.

Example :

```
$m200=" Stop the gate N°10 "
```

Places the message starting from word 200 by placing two characters in each word.

```
$m400=‘ Motor fault ‘
```

Places the message starting from word 400 by placing a character in the byte of lower weights of each word, the byte of higher weights contains 0.

The syntax « $...= » is used to continue a table of presettings after the previous one.

For example :

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

Place the variables 1 to 9 in the words m200 à m208.

Presettings can be written in the same manner as low level literal language or in a command on a sheet. In this case, the presetting starts with the character « # ».
Example of a presetting written in a code box :

Example of a presetting written in a command :

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

1.10.3.9. Indirect addressing

Indirect addressing is used to effect an operation on a variable with an index..

These are M variables (words) which are used as an index

Syntax :

« variable (index) »

Example :

```
lda 10       ; load 10 in the accumulator
sta m200     ; enter in the word 200
set o(200)   ; set to one the output indicated by the word 200 (o10)
```

1.10.3.10. Address of a variable

The character « ? » is used to specify the address of a variable.

Example :

```
lda ?o10           ; enters the value 10 in the accumulator
```

This syntax is primarily of interest if symbols are used.

Example :

```
lda ?_gate_        ; enters the variable number in the accumulator
                   ; associated to symbol « _gate_ »
```

This syntax can also be used in presettings to create variable address tables..

Example :

```
$m200=?_gate1_,?_gate2_,?_gate3_
```

1.10.3.11. Jumps and labels

Jumps must be referred to a label. Label syntax is :

« :label name: »

Example :

```
jmp :next:
...
:next:
```

1.10.3.12. Function list by type

1.10.3.12.1. Boolean functions

SET	set to one
RES	reset
INV	inversion
EQU	equivalence
NEQ	non-equivalence

1.10.3.12.2. Loading and storage functions on integers and floats

LDA	load
STA	storage

1.10.3.12.3. Arithmetic functions on integers and floats

ADA	addition
SBA	subtraction
MLA	multiplication
DVA	division
CPA	comparison

1.10.3.12.4. Arithmetic functions on floats

ABS	absolute value
SQR	square root

1.10.3.12.5. Access functions for PC input/output ports

AIN	access input
AOU	access output

1.10.3.12.6. Access functions for PC memory

ATM	input address memory
MTA	output address memory

1.10.3.12.7. Binary functions on integers

ANA	and bit to bit
ORA	or bit to bit
XRA	exclusive or bit to bit
TSA	test bit to bit

SET	set all bits to one
RES	reset all bits
RRA	shift to the right
RLA	shift to the left

1.10.3.12.8. Other functions on integers

INC	incrementation
DEC	decrementation

1.10.3.12.9. Conversion functions

ATB	integers to booleans
BTA	booleans to integers
FTI	float to integer
ITF	integer to float
LTI	32 bit integer to 16 bit integer
ITL	16 bit integer to 32 bit integer

1.10.3.12.10. Connection functions

JMP	jump
JSR	jump to sub routine
RET	return from sub routine

1.10.3.12.11. Test functions

RFZ	zero result flag
RFS	sign flag
RFO	overflow flag
RFC	carry flag

1.10.3.12.12. Asynchronous access functions to inputs outputs

RIN	read inputs
WOU	write outputs

1.10.3.12.13. Information contained in the function list

The following are provided for each instruction :

⇨ Name : mnemonic.

⇨ Function : a description of the function created by the instruction.

⇨ Variables : the types of variables used with the instruction

⇨ Addressing : the types of addressing used

⇨ Also see : the other instructions related to the mnemonic.

⇨ Example : a example of the use.

The post-processors which generate construction language are subject to certain limitations. See the information on these post-processors for details on these limitations.

ABS

Name	:	ABS - abs accumulator
Function	:	calculate the absolute value of the floating accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	SQR
Example	:	

```
lda f200
abs aaf
sta f201
; leaves f201 in the absolute value of f200
```


ADA

Name	:	ADA - adds accumulator
Function	:	adds a value to the accumulator
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	SBA
Example	:	

```
ada 200
; adds 200 to the 16 bit accumulator

ada f124
; adds the content of f124 to the float accumulator

ada l200
; adds the content of l200 to the 32 bit accumulator

ada 200L
; adds 200 to the 32 bit accumulator

ada 3.14R
; adds 3.14 to the float accumulator
```


AIN

Name	:	AIN - accumulator input
Function	:	reads an input port (8 bits) and stores in the lower part of the 16 bit accumulator ;  reads a 16 bit input port and stores in the 16 bit accumulator (in this case the port address must be written in the form of a 32 bit constant) only useable with PC compiler
Variables	:	M or %MW
Addressing	:	indirect, immediate
Also see	:	AOU
Example	:	

```
ain $3f8
; reads port $3f8 (8 bits)

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

ANA

Name	:	ANA - and accumulator
Function	:	effects an AND logic in the 16 bit accumulator and a word or a constant or the 32 bit accumulator and a long or a constant
Variables	:	M or %MW, L or %MD
Addressing	:	absolute, indirect, immediate
Also see	:	ORA, XRA
Example	:	

```
ana %1111111100000000
; masks the 8 bits of lower weight of
; the 16 bit accumulator

ana $ffff0000L
; masks the 16 bits of lower weight of the 32 bit accumulator
```

AOU

Name	:	AOU - accumulator output
Function	:	transfers the lower part (8 bits) of the 16 bit accumulator on an output port ;  transfers the 16 bits of the 16 bit accumulator on an output port (in this case the port address must be written in the form of a 32 bit constant) only useable with PC compiler
Variables	:	M or %MW
Addressing	:	indirect, immediate
Also see	:	AIN
Example	:	

```
lda "A"
aou $3f8
; places the character« A » on output port $3f8

lda $3f8
sta m200
lda "z"
aou m(200)
; places character « z » on output port $3f8

lda $1234
aou $300l
; places the 16 bit value 1234 on output port $300
```

ATB

Name	:	ATB - accumulator to bit
Function	:	transfers the 16 bits of the 16 bit accumulator towards the subsequent 16 boolean variables ; the the lower weight bit correspond to the first boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, U*
Addressing	:	absolute
Also see	:	BTA
Example	:	

```
lda m200
atb o0
; recopies the 16 bits of m200 in variables
; o0 to o15
```

* Note : to be able to use the U bits with this function it is necessary to create a linear table of bits using command #B.

ATM

Name	:	ATM - accumulator to memory
Function	:	transfers the 16 bit accumulator to a memory address; the word or specified constant defines the memory address offset to reach, the word m0 must be loaded with the segment value of the memory address to reach only useable with PC compiler
Variables	:	M or %MW
Addressing	:	indirect, immediate
Also see	:	MTA
Example	:	

```
lda $b800
sta m0
lda 64258
atm $10
; places the value 64258 at address $b800:$0010
```


BTA

Name	:	BTA - bit to accumulator
Function	:	transfers the subsequent 16 boolean variables towards the 16 bits of the 16 bit accumulator ; the lower weight bit corresponds to the first boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, U*
Addressing	:	absolute
Also see	:	ATB
Example	:	

```
bta i0
sta m200
; recopies the 16 inputs i0 to i15 in the word m200
```

* Note : to be able to use the U bits with this function it is necessary to create a linear table of bits using command #B.

CPA

Name	:	CPA - compares accumulator
Function	:	compares a value at the 16 bit or 32 bit or floating accumulator, effects the same operation as SBA but without changing the content of the accumulator
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	SBA
Example	:	

```
lda m200
cpa 4
rfz o0
; sets o0 to 1 if m200 is equal to 4, otherwise o0
; is reset to 0

lda f200
cpa f201
rfz o1
; sets o1 to 1 if f200 is equal to f201, otherwise o1
; is reset to 0
```

DEC

Name	:	DEC – decrement
Function accumulator	:	decrements a word, a counter, a long, the 16 bit or 32 bit
Variables	:	M or %MW, C or %C, L or %MD
Addressing	:	absolute, indirect, accumulator
Also see	:	INC
Example	:	

```
dec m200
; decrements m200

dec aal
; decrements the 32 bit accumulator

dec m200
dec m201 and m200=-1
; decrements a 32 bit value composed of
; m200 (lower weights)
; et m201 (higher weights)
```


DVA

Name	:	DVA - divides accumulator
Function	:	division of the 16 bit accumulator by a word or a constant; division of the float accumulator by a float or a constant; division of the 32 bit by a long or a constant, for the 16 bit accumulator the remainder is placed in word m0, if the division is by 0 system bit 56 passes to 1
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	MLA
Example	:	

```
lda m200
dva 10
sta m201
; m201 is equal to m200 divided by 10, m0 contains the
; remainder

lda l200
dva $10000L
sta l201
```


EQU

Name	:	EQU - equal
Function	:	sets a variable to 1 if the test is true, if not the variable is set to 0
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute, indirect (except for X variables)
Also see	:	NEQ, SET, RES, INV
Example	:	

```
equ o0 and i10
; sets the output of o0 to the same state as input i10

lda 10
sta m200
equ o(200) and i0
; sets o10 to the same state as input i0

$t0=100
equ t0 and i0
equ o0 and t0
; sets o0 to the state of i0 with an activation delay
; of 10 seconds
```


FTI

Name	:	FTI - float to integer
Function	:	transfers the float accumulator to the 16 bit accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	ITF
Example	:	

```
lda f200
fti aaa
sta m1000
; leaves the integer part of f200 in m1000
```

INC

Name	:	INC - increment
Function	:	increments a word, a counter, a long the 16 or 32 bit accumulator
Variables	:	M or %MW, C or %C, L or %MD
Addressing	:	absolute, indirect, accumulator
Also see	:	DEC
Example	:	

```
inc m200
; adds 1 to m200

inc m200
inc m201 and m201=0
; increments a value on 32 bits, m200
; represents the
; lower weights, and m201 the higher weights

inc l200
; increments long l200
```

INV

Name	:	INV - inverse
Function	:	inverts the state of a boolean variable or inverts all the bits of a word, a long or the 16 bit or 32 bit accumulator
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U, M or %MW, L or %MD
Addressing	:	absolute, indirect, accumulator
Also see	:	EQU, NEQ, SET, RES
Example	:	

```
inv o0
; inverts the state of output 0

inv aaa
; inverts all the bits of the 16 bit accumulator

inv m200 and i0
; inverts all m200 bits if i0 is at state 1
```

ITF

Name	:	ITF - integer to float
Function	:	transfers the 16 bit accumulator to the float accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	FTI
Example	:	

```
lda 1000
itf aaa
sta f200
; leaves the constant  1000 in f200
```


ITL

Name	:	ITL - integer to long
Function	:	transfers the 16 bit accumulator to the 32 bit accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	LTI
Example	:	

```
lda 1000
itl aaa
sta f200
; leaves the constant 1000 in l200
```

JMP

Name	:	JMP - jump
Function	:	jump to a label
Variables	:	label
Addressing	:	label
Also see	:	JSR
Example	:	

```
jmp :end of program:
; unconditional connection to end of
; program label:


jmp :string: and i0
set o0
set o1
:string:
; conditional connection to a label :string:
; following the state of i0
```


JSR

Name	:	JSR - jump sub routine
Function	:	effects a connection to a sub routine
Variables	:	label
Addressing	:	label
Also see	:	RET
Example	:	

```
        lda m200
        jsr :square:
        sta m201
        jmp end:

:square:
        sta m53
        mla m53
        sta m53
        ret m53

:end:

; the sub routine « square » raises the content
; of the accumulator to the square
```


LDA

Name	:	LDA - load accumulator
Function	:	loads a constant, word or counter in the 16 bit accumulator; loads a long or constant in the 32 bit accumulator; loads a float or a constant in the float accumulator; loads a counter or a time delay in the 16 bit accumulator
Variables	:	M or %MW, C or %C, L or %MD, F or %MF, T or %T
Addressing	:	absolute, indirect, immediate
Also see	:	STA
Example	:	

```
lda 200
; loads the constant 200 in the 16 bit accumulator

lda 0.01R
; loads the real constant 0.01 in the float accumulator

lda t10
; loads the counter of time delay 10 in the
; accumulator
```


LTI

Name	:	LTI - long to integer
Function	:	transfers the 32 bit accumulator to the 16 bit accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	ITL
Example	:	

```
lda l200
lti aaa
sta m1000
; leaves the 16 bits of lower weight of l200 in m1000
```


MLA

Name	:	MLA - multiples accumulator
Function	:	multiplies the 16 bit accumulator by a word or a constant; multiplies the 32 bit accumulator by a long or a constant; multiplies the float accumulator by a float or a constant; for the 16 bit accumulator the 16 bits of higher weight result of the multiplication will be transferred in m0
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	DVA
Example	:	

```
lda m200
mla 10
sta m201
; multiplies m200 by 10, m201 is loaded with the
; 16 bits of lower weight, and m0 with the 16 bits of
; higher weight
```


MTA

Name	:	MTA - memory to accumulator
Function	:	transfers the contents of a memory address to the 16 bit accumulator, the specified word or constant defines the offset of the memory address to reach; the word m0 must be loaded with the segment value of the memory address to be reached; only useable with a PC compiler
Variables	:	M or %MW
Addressing	:	indirect, immediate
Also see	:	ATM
Example	:	

```
lda $b800
sta m0
mta $10
; places the value contained at address $b800:$0010
; in the 16 bit accumulator
```


NEQ

Name	:	NEQ - not equal
Function	:	sets a variable to 0 if the test is true, if not the variable is set to 1
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute, indirect (except for X variables)
Also see	:	EQU, SET, RES, INV
Example	:	

```
neq o0 and i00
; sets the output of o0 to a complement state of input
; i10

lda 10
sta m200
neq o(200) and i0
; sets o10 to a complement state of input i0

$t0=100
neq t0 and i0
neq o0 and t0
; sets o0 to the state of i0 with a deactivation
; delay of 10 seconds
```

ORA

Name	:	ORA - or accumulator
Function	:	effects an OR logic on the 16 bit accumulator and a word or a constant, or on the 32 bit accumulator and a long or a constant
Variables	:	M or %M, L or %MD
Addressing	:	absolute, indirect, immediate
Also see	:	ANA, XRA
Example	:	

```
ora %1111111100000000
; sets the 8 bits of lower weight of
; the 16 bit accumulator to 1

ora $ffff0000L
; sets the 16 bits of higher weight of the 32 bit accumulator
; to 1
```


RES

Name	:	RES - reset
Function	:	sets a boolean variable, a word a counter, a long, the 16 bit accumulator or the 32 bit accumulator to 0
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U, M or %MW, C or %C, L or %MD
Addressing	:	absolute, indirect (except for X variables), accumulator
Also see	:	NEQ, SET, EQU, INV
Example	:	

```
res o0
; sets the output of o0 to 0

lda 10
sta m200
res o(200) and i0
; sets o10 to 0 if input i0 is at 1

res c0
; sets counter 0 to 0
```

RET

Name	:	RET - return
Function	:	indicates the return of a sub routine and places a word or a constant in the 16 bit accumulator; or places a long or a constant in the 32 bit accumulator; or places a float or a constant in the float accumulator
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	JSR
Example	:	

```
ret 0
; returns to a sub routine by placing 0 in
; the 16 bit accumulator

ret f200
; returns to a sub routine by placing the content of
; f200 in the float accumulator
```

RFC

Name	:	RFC - read flag : carry
Function	:	transfers the carry indicator in a boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute
Also see	:	RFZ, RFS, RFO
Example	:	

```
rfc o0
; transfers the carry indicator to o0

lda m200
ada m300
sta m400
rfc b99
lda m201
ada m301
sta m401
inc m401 and b99
; effects an addition on 32 bits
; (m400,401)=(m200,201)+(m300,301)
; m200, m300 and m400 are lower weights
; m201, m301 and m401 are higher weights
```

RFO

Name	:	RFO - read flag : overflow
Function	:	transfers the contents of the overflow indicator in a boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute
Also see	:	RFZ, RFS, RFC
Example	:	

```
rfo o0
; transfers the overflow indicator to o0
```


RFS

Name	:	RFS - read flag : sign
Function	:	transfers the sign indicator in a boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute
Also see	:	RFZ, RFC, RFO
Example	:	

```
rfs o0
; transfers the sign indicator to o0
```


RFZ

Name	:	RFZ - read flag : zero
Function	:	transfers the content of a zero result indicator in a boolean variable
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U
Addressing	:	absolute
Also see	:	RFC, RFS, RFO
Example	:	

```
rfz o0
; transfers the zero result indicator to o0

lda m200
cpa m201
rfz o0
; position o0 at 1 if m200 is equal to m201
; or 0 if not
```

RIN

Name	:	RIN - read input
Function	:	effects a reading of physical input. This function is only implemented on Z targets and varies following the target. See the documentation related to each executor for more information..
Variables	:	none
Addressing	:	immediate
Also see	:	WOU

RLA

Name : RLA - rotate left accumulator
Function : effects a left rotation of the bits of the
16 bit or 32 bit accumulator; the bits evacuated to the left enter on the right, the subject of this function is a constant which sets the number of shifts to be made, the size of the subject (16 or 32 bits) determines which of the accumulators will undergo rotation

Variables : none

Addressing : immediate

Also see : RRA

Example :

```
ana $f000
; separates the digit of higher weight of the 16 bit
accumulator

rla 4
; and brings it to the right


rla 8L
; effects 8 rotations to the left of the bits of the 32 bit
; accumulator
```


RRA

Name : RRA - rotate right accumulator
Function : effects a right rotation of the bits of the

16 bit or 32 bit accumulator; the bits evacuated to the right enter on the left, the subject of this function is a constant which sets the number of shifts to be made, the size of the subject (16 or 32 bits) determines which of the accumulators will undergo rotation

Variables : none

Addressing : immediate

Also see : RLA

Example :

```
ana $f000
; separates the digit of higher weight of the 16 bit
rra 12
; and brings it to the right

rra 1L
; effects a rotation of the bits of the 32 bit accumulator
; to a position towards the right
```


SBA

Name	:	SBA - subtracts accumulator
Function	:	removes the content of a word or constant from the 16 bit accumulator; removes the content of a long or a constant from the 32 bit accumulator; removes the content of a float or constant from the float accumulator
Variables	:	M or %MW, L or %MD, F or %MF
Addressing	:	absolute, indirect, immediate
Also see	:	ADA
Example	:	

```
sba 200
; removes 200 from the 16 bit accumulator

sba f(421)
; removes the float content if the number is contained
; in word 421 from the float accumulator
```

SET

Name	:	SET - set
Function	:	sets a boolean variable to 1; sets all the bits of a word, a counter, a long, the 16 bit or the 32 bit accumulator to 1
Variables	:	I or %I, O or %Q, B or %M, T or %T, X or %X, U, M or %MW, C or %C, L or %MD
Addressing	:	absolute, indirect (except for X variables), accumulator
Also see	:	NEQ, RES, EQU, INV
Example	:	

```
set o0
; sets the output of o0 to 1

lda 10
sta m200
set o(200) and i0
; sets o10 to 1 if input i0 is at 1

set m200
; sets m200 to the value -1

set aal
; sets all the bits of the 32 bit accumulator to 1
```

SQR

Name	:	SQR - square root
Function	:	calculates the square root of the float accumulator
Variables	:	none
Addressing	:	accumulator
Also see	:	ABS
Example	:	

```
lda 9
itf aaa
sqr aaf
fti aaa
sta m200
; leaves value 3 in m200
```

STA

Name	:	STA - store accumulator
Function	:	stores the 16 bit accumulator in a counter or a word; stores the 32 bit accumulator in a long; stores the float accumulator in a float, stores the 16 bit accumulator in a time delay order
Variables	:	M or %MW, C or %C, L or %MD, F or %MF, T or %T
Addressing	:	absolute, indirect
Also see	:	LDA
Example	:	

```
sta m200
; transfers the content of the 16 bit accumulator
; to word 200

sta f200
; transfers the content of the float accumulator
; to float 200

sta l200
; transfers the 32 bit accumulator to long l200
```


TSA

Name	:	TSA - test accumulator
Function	:	effects AND logic on the 16 bit accumulator and a word or a constant, effects AND logic on the 32 bit accumulator and a long or a constant, operates in a similar manner to ANA instruction but without changing the accumulator content
Variables	:	M or %MW, L or %MD
Addressing	:	absolute, indirect, immediate
Also see	:	ANA
Example	:	

```
tsa %10
rfz b99
jmp :follow: and b99
; connection to label :follow: if bit 1
; of the 16 bit accumulator is at 0
```


WOU

Name	:	WOU - write output
Function	:	effects a writing of the physical outputs. This function is only implemented on Z targets (and varies following the target) See the documentation related to each executor for more information
Variables	:	none
Addressing	:	immediate
Also see	:	RIN

XRA

Name	:	XRA - xor accumulator
Function	:	effects an EXCLUSIVE OR on the 16 bit accumulator and a word or a constant, effects an EXCLUSIVE OR on the 32 bit accumulator and a long or a constant
Variables	:	M or %MW, L or %MD
Addressing	:	absolute, indirect, immediate
Also see	:	ORA, ANA,
Example	:	

```
xra %1111111100000000
; inverts the 8 bits of higher weight of the 16 bit accumulator

xra 1L
; inverts the lower weight bit of the 32 bit accumulator
```

1.10.4. Macro-instruction

Macro-instructions are new literal language instructions which hold a set of basic instructions.
Call up syntax for a macro-instruction :
« %<Macro-instruction name*> {parameters ...} »
Statement syntax for a macro-instruction :

```
#MACRO
<program>
#ENDM
```

This statement is found in a file with the name of the macro-instruction and the extension « .M ».

The file M can be placed in a sub-directory « lib » of the AUTOMGEN installation directory or in project resources.

Ten parameters can be passed to the macro-instruction. When called up these parameters are placed on the same line as the macro-instruction and are separated by a space
The syntax « {?n} » in the macro-instruction program refers to the n parameter.

Example :
We are going to create a « square » macro-instruction which raises the first parameter of the macro-instruction to its square and puts the results in the second parameter.
Call up of the macro-instruction :
lda 3
sta m200
%square m200 m201
; m201 will contain 9 here

* The name of the macro-instruction can be a complete access path to the file « .M », it can contain a read and directory designation.

« SQUARE.M » file:

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

1.10.5. Libraries

A library is used to define the resources which will be compiled one time in an application, no matter how many times those resources are called up..

Syntax for defining a library :

```
#LIBRARY <Library name>
<program>
#ENDL
```

<library name > is the function name which will be called up for a
jsr :<library name> instruction:
The first time the library code is called up by the compiler its code is compiled. The following times, the call up is simply directed to the existing routine..
This mechanism is especially suited to the use of function blocks and macro-instructions to limit the generation of codes in the event that there is multiple use of the same program resources.
Words m120 to m129 are reserved for libraries and can be used for passing parameters.

1.10.6. Pre-defined macro-instructions

Inversion macro-instructions are in the sub-directory « LIB » of the AUTOMGEN installation directory.
Functional block equivalents are also present.

1.10.7. Description of pre-defined macro-instructions

1.10.7.1. Conversions

```
%ASCTOBIN <first two digits> <last two digits> <binary result>
```

Effecting a hexadecimal ASCII conversion (first two parameters) to binary (third parameter), by exiting the accumulator containing $FFFF if the first two parameters are not valid ASCII numbers, otherwise 0. All the parameters are 16 bit words.

```
%BCDTOBIN <value in BCD> <binary value>
```

Effecting a BCD conversion to binary. In the output of the accumulator containing $FFFF if the first parameter is not a valid bcd number, otherwise 0. The two parameters are 16 bit words.

```
%BINTOASC <binary value> <upper part result> <lower part result>
```

Effecting a binary conversion (first parameter) to hexadecimal ASCII (second and third parameters). All parameters are 16 bit words.

```
%BINTOBCD <binary value> <BCI value>
```

Effecting a BCD (first parameter) conversion to binary (second parameter). In the accumulator containing $FFFF if the binary number can be converted in BCD, otherwise 0.

```
%GRAYTOB <GRAY code value> <binary value>
```

Effecting a Gray code conversion (first parameter) to binary (second parameter).

1.10.7.2. Treatment on word tables

```
%COPY <first word table source> <first word table destination> <number of words>
```

Copy a table of source words to a table of destination words. The length is given by the number of words..

```
%COMP <first word table 1> <first word table 2> <number of words> <result>
```

Compares two tables of words. The result is a binary variable which takes the value 1 if all the elements in table 1 are identical to those in table 2.

```
%FILL <first word table> <value> <number of words>
```

Fills a word table with a value.

1.10.7.3. Treatment on strings

The coding of strings is as follows: one character per word, one word containing the value 0 indicates the end of the chain. In macro-instructions the strings are passed in parameters by designating by the first word they are composed of.

`%STRCPY <source string> <destination string>`

Copies a string to another.

`%STRCAT <source string> <destination string>`

Adds the source string to the end of the destination string.

`%STRCMP <string 1> <string 2> <result>`

Compares to strings. The result is a boolean variable which passes to 1 if the two strings are identical.

`%STRLEN <string> <result>`

Places the length of the string in the result word.

`%STRUPR <string>`

Transforms all the characters of the string into capital letters.

`%STRLWR <string>`

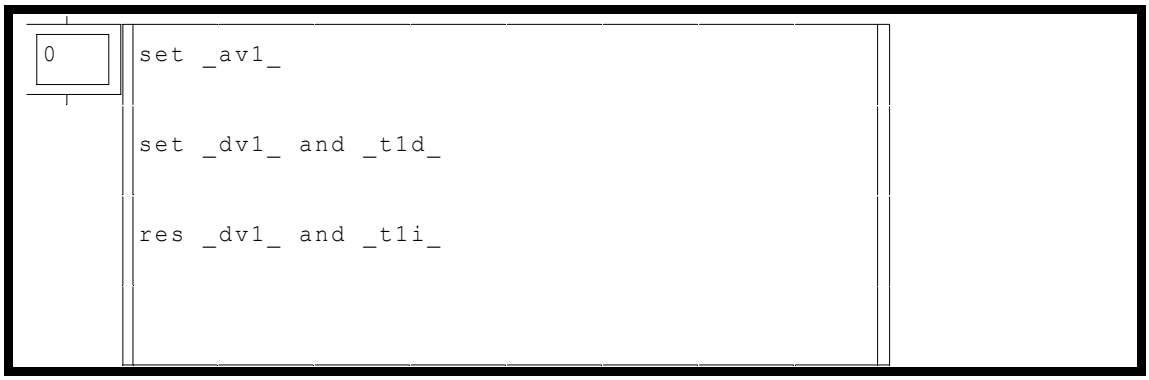
Transforms all the characters of the string into lower case letters.

Example:
Conversion of m200 (binary) to m202, m203 in 4 digits (ASCII bcd)
%bintobcd m200 m201
%bintoasc m201 m202 m203

1.10.8. Example of low level literal language

Conditions : let's start with the simplest example: round trip of a locomotive on track 1.

Solution :



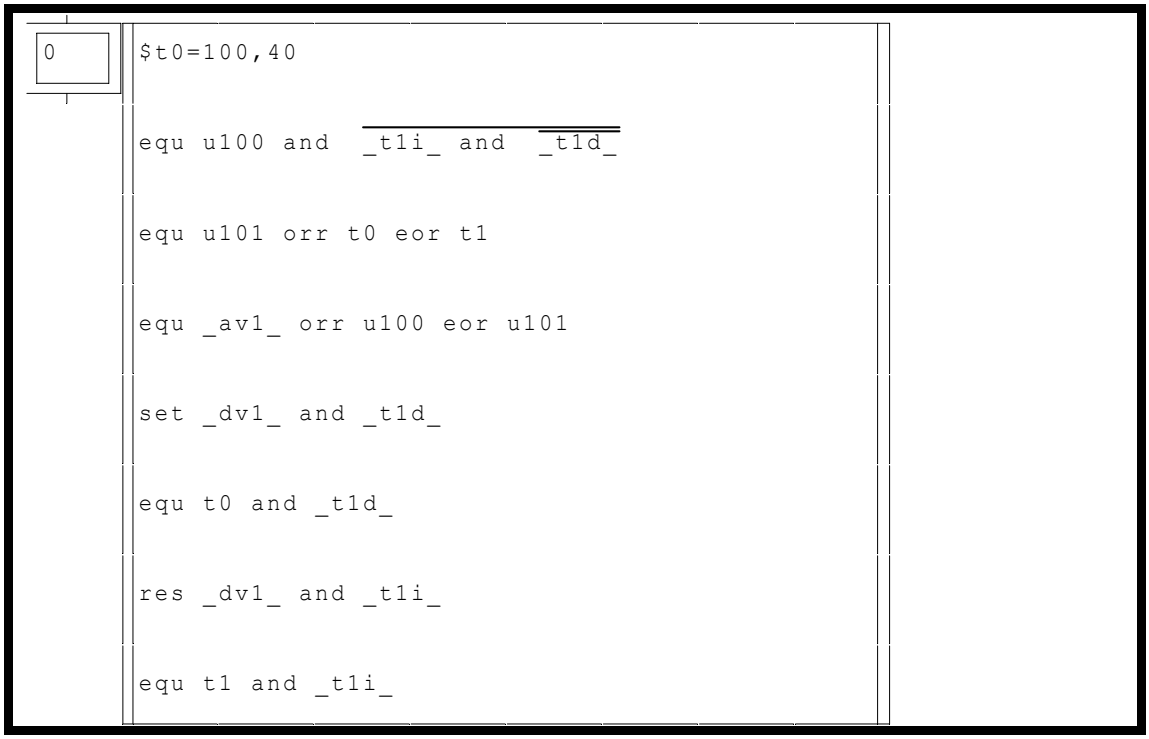
Example\lit\littéral bas niveau1.agn

A more evolved example.

Conditions :

The locomotive must make a 10 second delay at the right end of the track and a 4 second delay at the left end.

Solution :



Example\lit\littéral bas niveau 2.agn

Another example:

Conditions :

Make all of the model lights flash:

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:
```

Example\lit\littéral bas niveau 3.agn

This example shows the use of presettings. They are used here to create a variable address table. The table contains the addresses of all the outputs which pilot the model lights.

For each execution cycle, the state of all the lights is inverted.

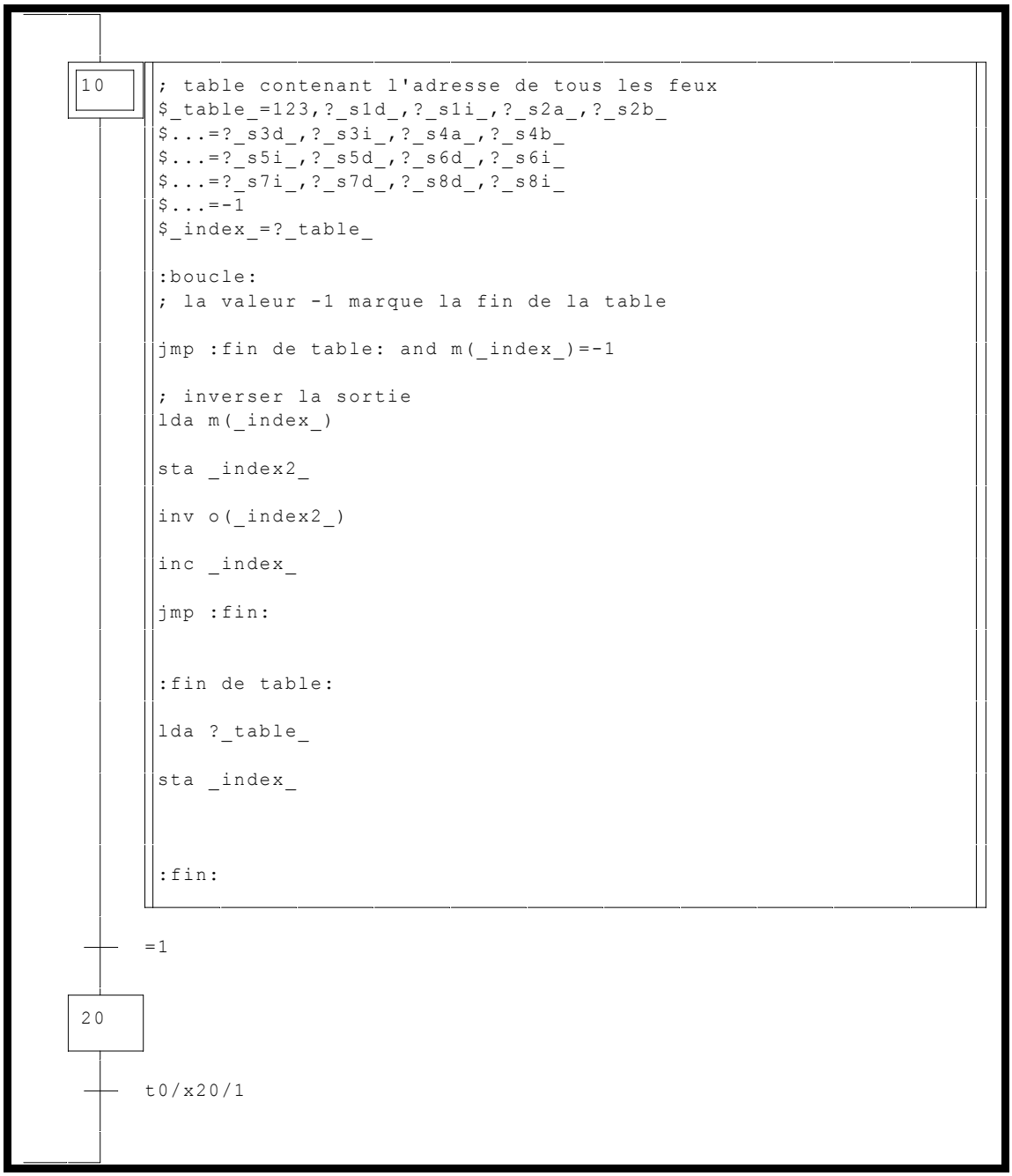
A problem occurs, all the lights flash very quickly and it is hard to see much.

Let's modify our example.

Conditions:

The state of all the lights must remain inverted every ten seconds one by one.

Solution :



Example\lit\littéral bas niveau 4.agn

1.11. Extended literal language

Extended literal language is a subset of low level literal language. It is used for writing boolean and numeric equations more simply and concisely.

It is still possible to write structures like IF ... THEN ... ELSE and WHILE ... ENDWHILE (loop).

Use of extended literal language is subject to the same rules as low level literal language, it uses the same syntax for variables, mnemonics, the test types (fronts, complement state, immediate state) and addressing modes.
It is possible to mix low level literal language with extended literal language.
When the compiler of literal language detects a line written in extended literal language, it decomposes it into low level literal language instructions, then compiles it.

1.11.1. Writing boolean equations

General syntax :

```
« bool. variable=(assignment type) (bool. variable 2 operator 1 bool. variable
3... operator n -1 bool. variable n ) »
```

The type of assignment must be indicated if it is other than « Assignment »
It can be :

⇨ « (/) » : complement assignment,

⇨ « (0) » : reset,

⇨ « (1) » :set to one.

The operators can be :

⇨ « . » : and,

⇨ « + » : or.

The equations can contain various levels of parentheses to indicate the evaluation order. By default, the equations are evaluated from the left towards the right.

Examples and equivalencies with low level literal language

```
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)
o0=(/)(i0.i1)
o0=(/i0)
o0=(/i0./i1)
o0=(c0=10)
o0=(m200<100+m200>200)
```

```
neq o0 and i0
neq o0 and i0 and i1
equ o0 and /i0
equ o0 and /i0 and /i1
equ o0 and c0=10
equ o0 orr m200<100 eor m200>200
```

1.11.2. Writing numeric equations

General equations for integers :

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

The equations can contain various levels of braces for indicating the evaluation order. By default, the equations are evaluated from left to right. Operators for 16 and 32 bit integers can be:

« + » : addition (equivalent to instruction ADA),
« - » : subtraction (equivalent to instruction SBA),
« * » : multiplication (equivalent to instruction MLA),
« / » : division (equivalent to instruction DVA),
« < » : shift to left (equivalent to instruction RLA),
« > » : shift to right (equivalent to instruction RRA),
« & » : « And » binary (equivalent to instruction ANA),
« | »* : « Or » binary (equivalent to instruction ORA),
« ^ » : « Exclusive or » binary (equivalent to instruction XRA).

Operators for floats can be :

⇨ « + » : addition (equivalent to instruction ADA),

⇨ « - » : subtraction (equivalent to instruction SBA),

⇨ « * » : multiplication (equivalent to instruction MLA),

⇨ « / » : division (equivalent to instruction DVA).

It is possible to indicate the constant in float equations. If this is necessary use the presettings on floats.
Equations on floats can call up the « SQR » and « ABS » functions

* This character is normally associated to the [ALT] + [6] keys on keyboards

Note : depending on the complexity the compiler may use intermediate variables. These variables are the words m53 to m59 for 16 bit integers, the longs l53 to l59 for 32 bit integers and the floats f53 à f59.

Examples and equivalencies with low level literal language

```
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. IF...THEN...ELSE...structure

General syntax :

```
IF(test)
      THEN
      action if true test
      ENDIF
      ELSE
      action if false test
      ENDIF
```

The test must comply with the syntax described in the chapter dedicated to boolean equations.
Only if an action tests true or tests false can it appear.
It is possible to connect multiple structures of this type.
System bits u90 to u99 are used as temporary variables for managing this type of structure.

Examples :

```
IF(i0)
      THEN
      inc m200            ; increments word 200 if i0
      ENDIF

IF(i1+i2)
      THEN
      m200=[m200+10]      ; adds 10 to word 200 if the or i2
      ENDIF
      ELSE
      res m200            ; else effect m200
      ENDIF
```

1.11.4. WHILE ... ENDWHILE structure

General syntax :

```
WHILE(test)
      action is repeated as long as the test is true
ENDWHILE
```

The test must comply with the syntax described in the chapter dedicated to boolean equations.

It is possible to connect multiple structures of this type.

System bits u90 to u99 are used as temporary variables for managing this type of structure.

Examples :

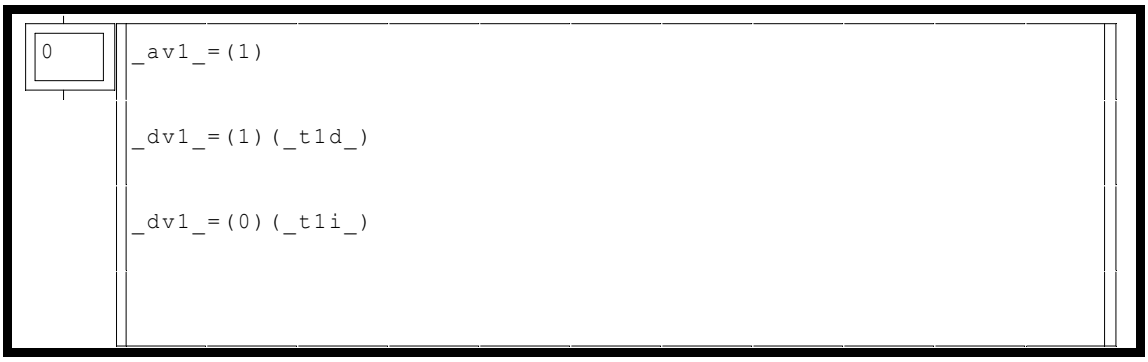
```
m200=[0]
WHILE(m200<10)
       set o(200)
       inc m200            ; increments word 200
ENDWHILE
```

This example sets outputs o0 to o9 to one.

1.11.5. Example of a program in extended literal language

Let's go back to the example from the previous chapter

Solution :



Example\lit\littéral étendu 1.agn

Let's complicate our example with some calculations

Conditions :

Calculate the speed in millimeters per second and meters per hour of the locomotive on the left to right trajectory.

Solution :

Example\lit\littéral étendu 2.agn

Word 32 is used to read the system time. The value is then transferred to the float to effect the calculations without compromising exactness.

1.12. ST literal language

ST literal language is a structured literal language defined by IEC1131-3 standard. This language is used to write boolean and numeric equations as well as program structures.

1.12.1. General Information

ST literal language is used in the same way as low level literal language and extended literal language.

Commands are used to establish the sections in ST literal language
« #BEGIN_ST » indicates the beginning of an ST language section.
« #END_ST » indicates the end of an ST language section.
Example :

```
m200=[50]           ; extended literal language
#BEGIN_ST
m201:=4;            (* ST language *)
#END_ST
```

It is also possible to choose to use ST language for an entire sheet.
This selection is made in the properties dialogue box on each sheet.

On a sheet where ST language is the default language it is possible to enter low level and extended literal language by using the commands « #END_ST » and « #BEGIN_ST ».

Comments for ST language must start with « (* » and end with « *) ».

ST language instructions end with the character « ; ». Multiple instructions can be written on the same line.

Example :

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

1.12.2. Boolean equations

The general syntax is :

```
variable := boolean equation;
```

Boolean equations can be composed of a constant, a variable or multiple variables separated by operators.

Constants can be : 0, 1, FALSE or TRUE.

Examples :

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

The operators used to separate multiple variables are : + (or), . (and), OR or AND.

« And » has priority over« Or ».

Example :

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

Will be treated as :

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

Parentheses can be used in the equations to indicate priorities.

Example :

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

Numeric tests can be used.

Example :

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

1.12.3. Numeric equations

The general syntax is :

```
variable := numeric equation;
```

Numeric equations can be composed of a constant, a variable or multiple variables separated by operators.

The constants can be expressed as decimal, hexadecimal (prefix #16) or binary (prefix #2) values.

Examples :

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

Operators are used to separate multiple variables or constants in their order of priority.

* (multiplication),/ (division), + (addition), - (subtraction), & or AND (binary and), XOR (binary exclusive or), OR (binary or).

Examples :

```
m200:=1000*m201;
m200:=m202-m204*m203;            (* equivalent to m200:=m202-(m204*m203) *)
```

Parentheses can be used in the equations to indicate priority.

Example :

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

1.12.4. Programming structures

1.12.4.1. IF THEN ELSE test

Syntax :

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

and

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

Example :

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

1.12.4.2. WHILE loop

Syntax :

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

Example :

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

1.12.4.3. REPEAT UNTIL loop

Syntax :

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

Example :

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

1.12.4.4. FOR TO loop

Syntax :

```
FOR variable:=start value TO end value DO action ENDFOR;
```

or

```
FOR variable:=start value TO end value BY no DO action ENDFOR;
```

Example :

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

1.12.4.5. Exiting a loop

The key word « EXIT » is used to exit a loop.

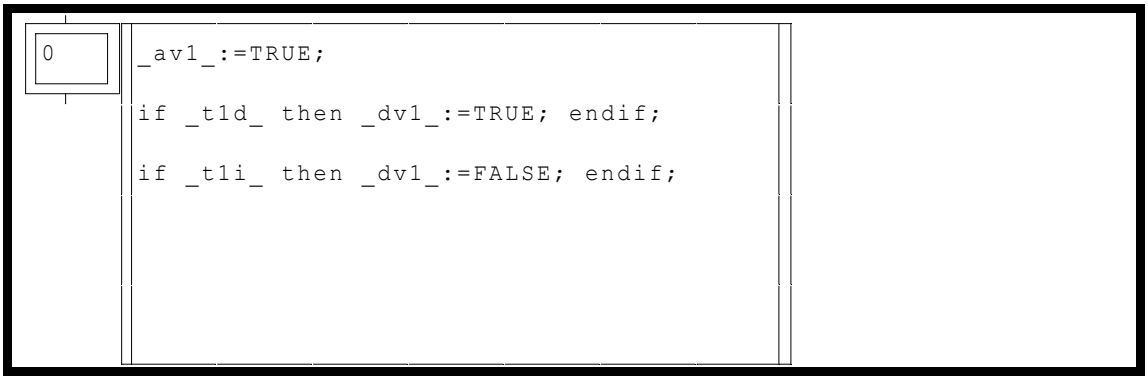
Example :

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

1.12.5. Example of a program in extended literal language

Let's go back to our example in the previous chapter

Solution :



Example\lit\littéral ST 1.agn

1.13. Organization chart

AUTOMGEN implements a « organization chart » type program.
Literal languages must be used with this type of program. See the previous chapters to learn how to use these languages.
The basis of programming with an organizational chart form is the graphic representation of an algorithmic treatment.
Unlike Grafcet language, programming in the organizational chart form generates a code which will be executed one time per search cycle. This means that it is not possible to remain in an organizational chart rectangle it is mandatory for the execution to exit the organizational chart to continue to execute the rest of the program..

This is a very important point and must not be forgotten when this language is selected.
Only rectangles can be drawn. The contents of a rectangle and its connections determine if the rectangle is an action or a test.

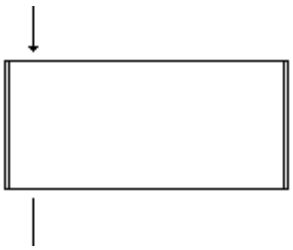
1.13.1. Creating an organizational chart

The rectangles are drawn by selecting the command « Add … / Code box» from the menu (click on the right side of the mouse on the bottom of the sheet to open the menu).

It is necessary to place a block (key [<]) at the entry of each rectangle, this must be placed on the upper part of the rectangle.

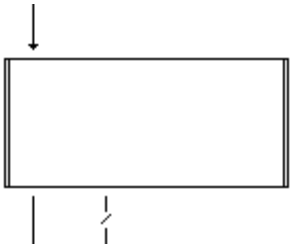
If the rectangle is an action it will have only one exit represented by a block (key [E]) on the lower left side of the rectangle.

An action rectangle :



If the rectangle is a test it must have two outputs. The first is represented by a block. (key [E]) on the lower left side and is for a true test, the second represented by a block (key [=]) is immediately to the right of the other output and is for a false test.

A test rectangle :



The branches of the organizational chart must always end with a rectangle without an output that could remain empty.

1.13.2. Rectangle content

1.13.2.1. Action rectangle content

Action rectangles can contain any kind of literal language instructions.

1.13.2.2. Test rectangle content

Test rectangles must contain a test that complies with the test syntax of the IF...THEN...ELSE... type structure of extended literal language.

For example :

```
IF (i0)
```

It is possible to write actions before this test in the test rectangle.

This can be used to make certain calculations before the test

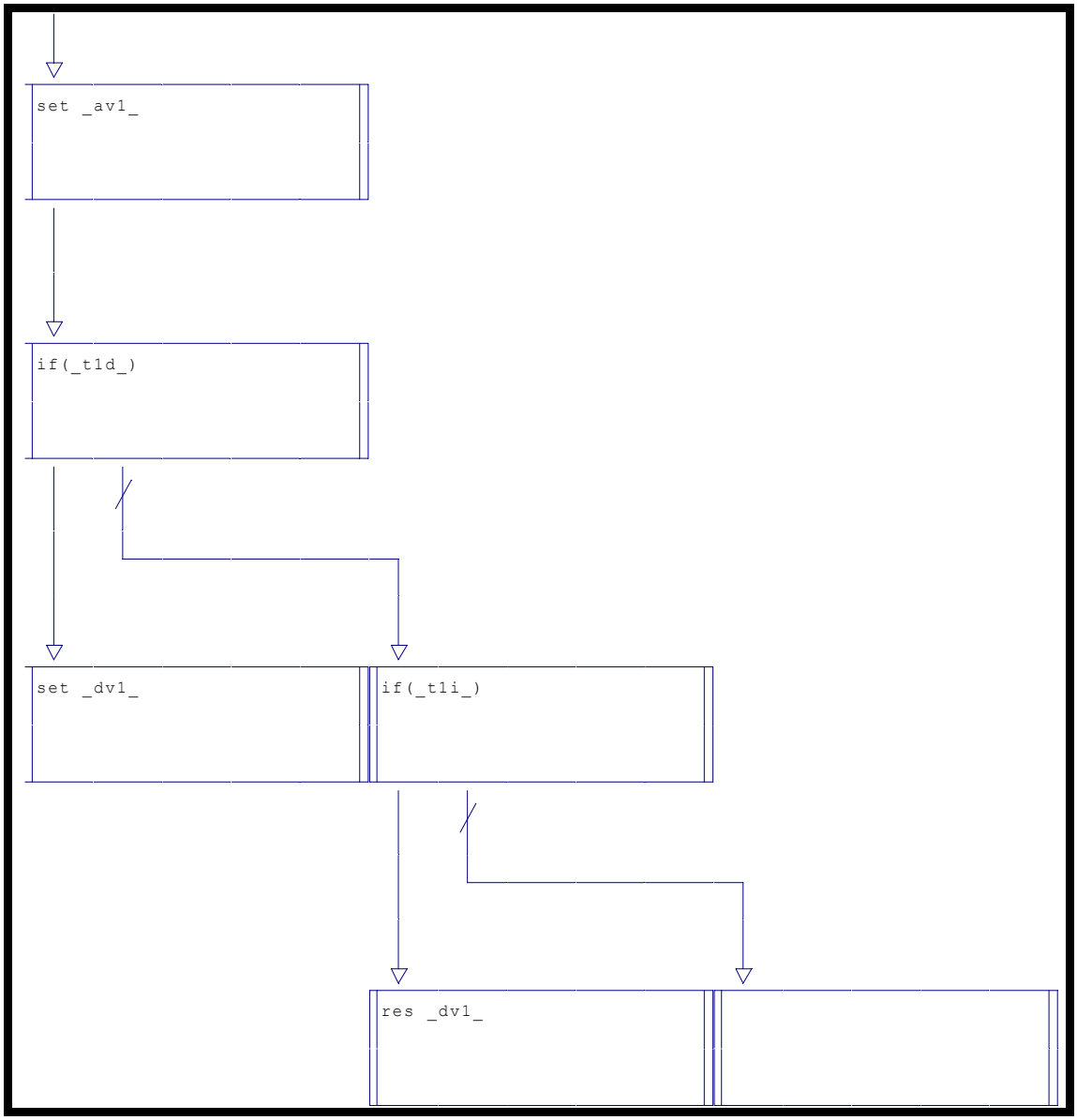
For example, if we want to test if the word 200 is equal to the word 201 plus 4 :

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

1.14. Illustration

Our first, now typical, example, is to make a locomotive make round trips on track 1 of the model.

Solution :



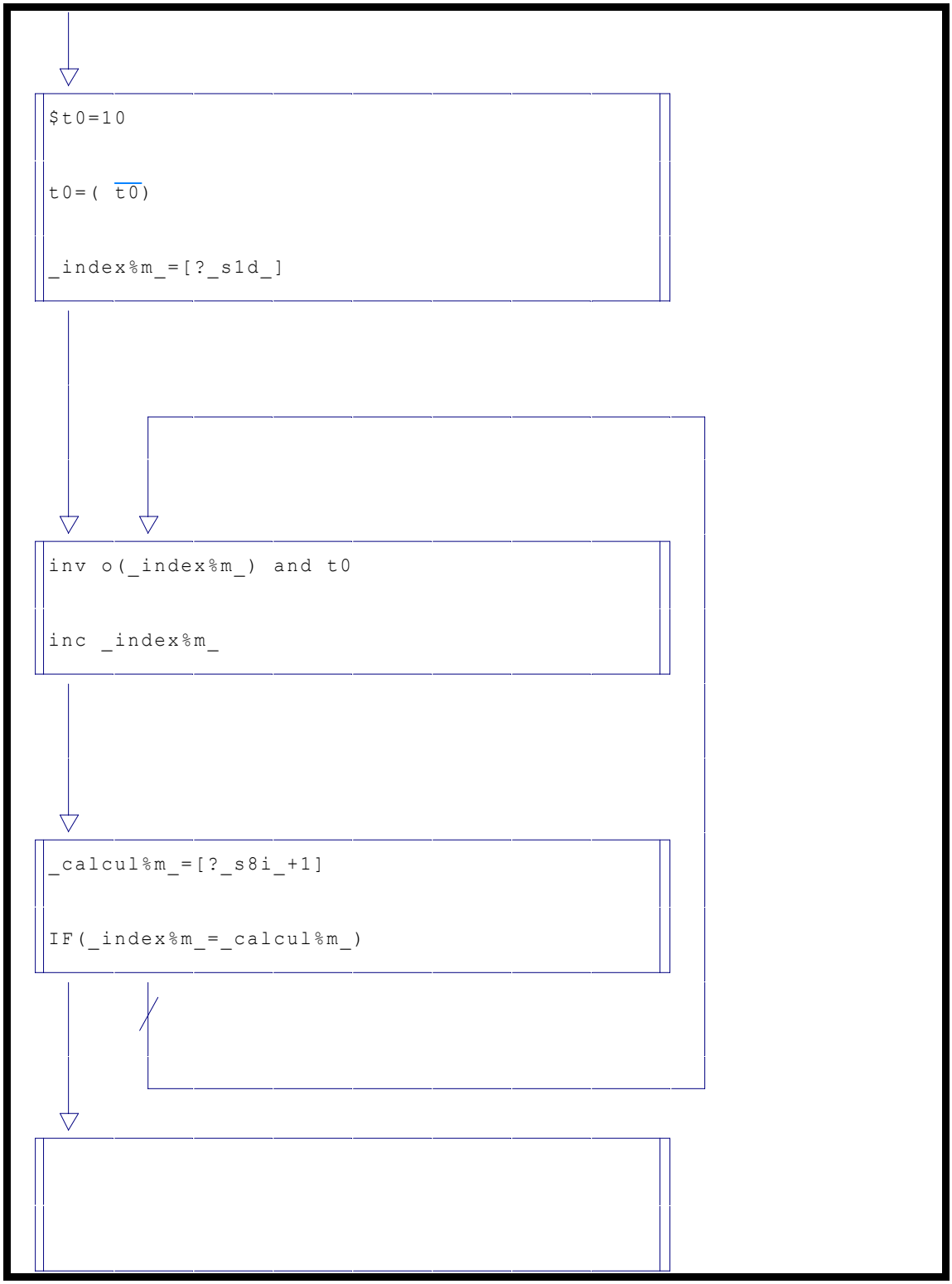
Example\organigramme\organigramme 1.agn

Second example

Conditions :

Make all the model light flash. The light change states every second.

Solution :



Example\organigramme\organigramme 2.agn

Note the use of automatic symbols in this example.

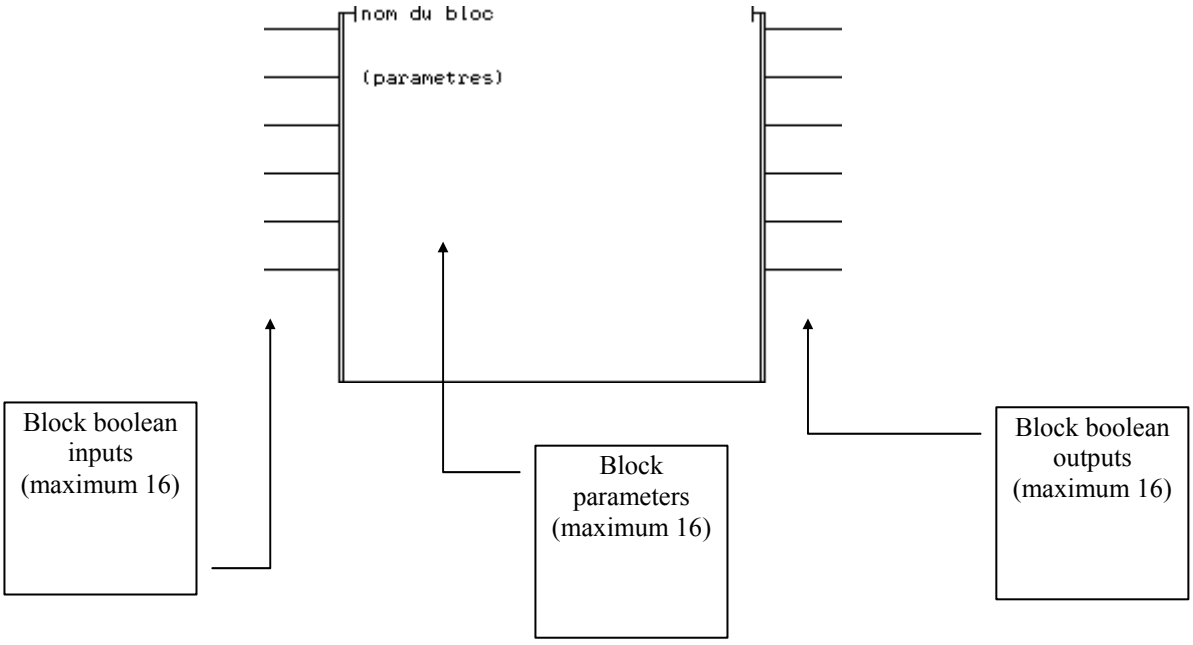
1.15. Function blocks

AUTOMGEN implements the use of function blocks.

This modular programming method is used to associate a set of instructions written in literal language to a graphic element .

Function blocks are defined by the programmer. Their number is not limited. It is possible to create sets of function blocks to allow a modular and standardize concept of applications.

Function blocks are used within flow chart or ladder type models, they have n boolean inputs and n boolean outputs. If the block is going to treat variables which are not boolean, then they will be mentioned in the drawing of the function block. The inside of the block can receive parameters: constant or variable.



1.15.1. Creating a function block

A function block is composed of two separate files. One file has « .ZON » extension which contains the drawing of the function block and a file with « .LIB » extension which contains a series of instructions written in literal language which establish the functionality of the function block.

The « .ZON » and « .LIB » files must bear the name of the function block. For example, if we decide to create a function block « MEMORY », we need to create the files « MEMORY.ZON » (to draw the block) and « MEMORY.LIB » (for the functionality of the block).

1.15.2. Drawing a block and creating a « .ZON » file

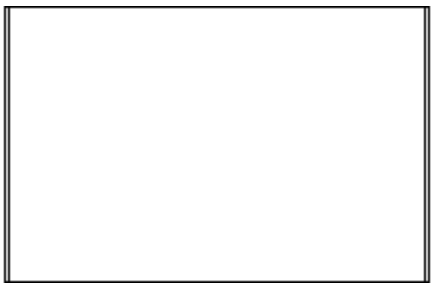
The envelop of a function block is composed of a code box to which blocks dedicated for the function block are added.

To draw a function block follow the steps below:

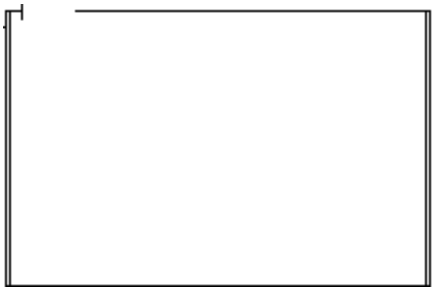
⇨ use the assistant (recommended)

Or :

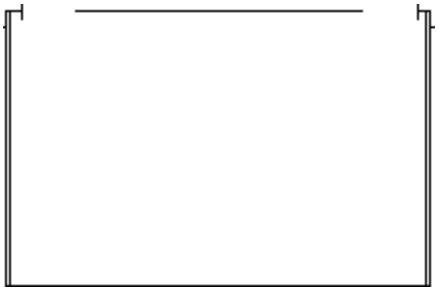
⇨ draw a code box (use the command « Add …/Code box » from the menu) :



⇨ place a block (key [8]) on the upper right corner of the code box :

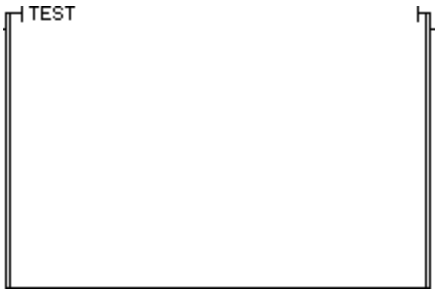


⇨ place a block (key [9]) on the upper right corner of the code box :



⇨ delete the line at the top of the block (key [A] is used to place blank blocks) :

⇨ click with the left side of the mouse on the upper left corner of the functional block, then enter the name of the functional block which must not be more than 8 characters (the « .ZON » and « .LIB » files must bear this name), then press [ENTER].



⇨ if additional boolean inputs are necessary, a block must be used (key [;]) or (key [:]), the added inputs must be located right below the first input, no free space should be left,

⇨ if additional boolean outputs are needed a block must be added (key [>]) or (key [?]), the added outputs must be located right below the first output, no free space should be left,

⇨ the interior of the block can contain comments or parameters, the parameters are written between braces « {...} ». Everything not written between braces is ignored by the compiler. It is interesting to indicate the use of boolean inputs and outputs inside the block.

⇨ when the block is finished, the command « Select » must be used from the « Edit » menu to select the drawing of the functional block, then save it in the « .ZON » file with the « Copy to» command from the « Edit » menu.

1.15.3. Creating an « .LIB » file

The « .LIB » file is a text file containing instructions in literal language (low level or extended). These instructions establish the functionality of the function block.

A special syntax is used to refer to block boolean inputs, block boolean outputs and block parameters.

To refer to a block boolean input, use the syntax« {Ix} » where x is the number of the boolean input expressed in hexadecimal (0 to f).

To refer to a block boolean output, use the syntax« {Ox} » where x is the number of the boolean output expressed in hexadecimal (0 to f).

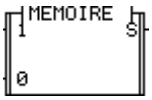
To refer to a block parameter use the syntax « {?x} » where x is the number of the parameter in hexadecimal (0 to f).

The .LIB can be placed in the « lib » sub-directory of the AUTOMGEN installation directory or in the project resources.

1.15.4. Simple example of a function block

We are going to create a « MEMORY » function block which contains two boolean inputs (set to one and reset) and a boolean output (memory state).

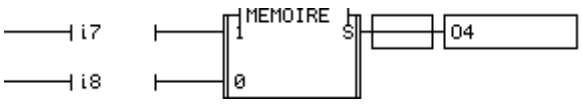
The block drawing contained in the « MEMORY.ZON » file is :



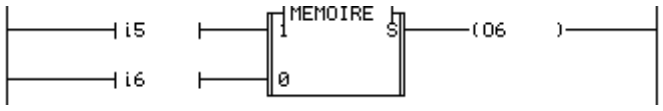
Block functionality contained in the « MEMORY.LIB » file is :

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

The block can then be used in the following way :



or



To use a function block in an application, select the command «Paste from » from the « Edit » menu and select the « .ZON » file corresponding to the function block used.

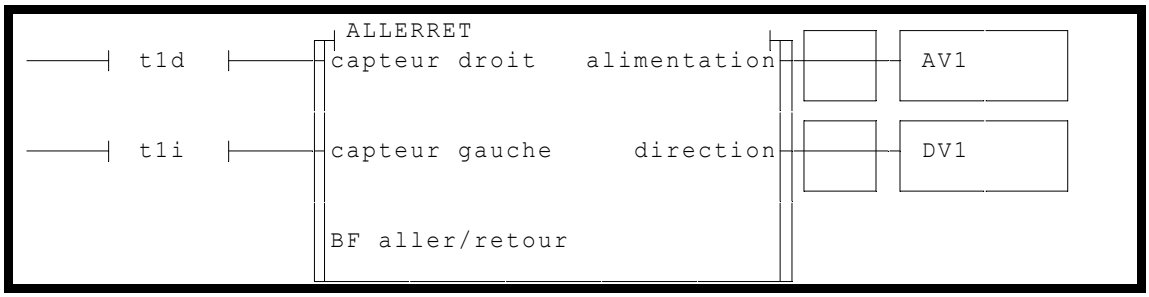
1.15.5. Illustration

Let's go back to our typical example.

Conditions :

Round trip of a locomotive on track 1 of the model.

Solution :



Example\bf\bloc-Functionnel 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

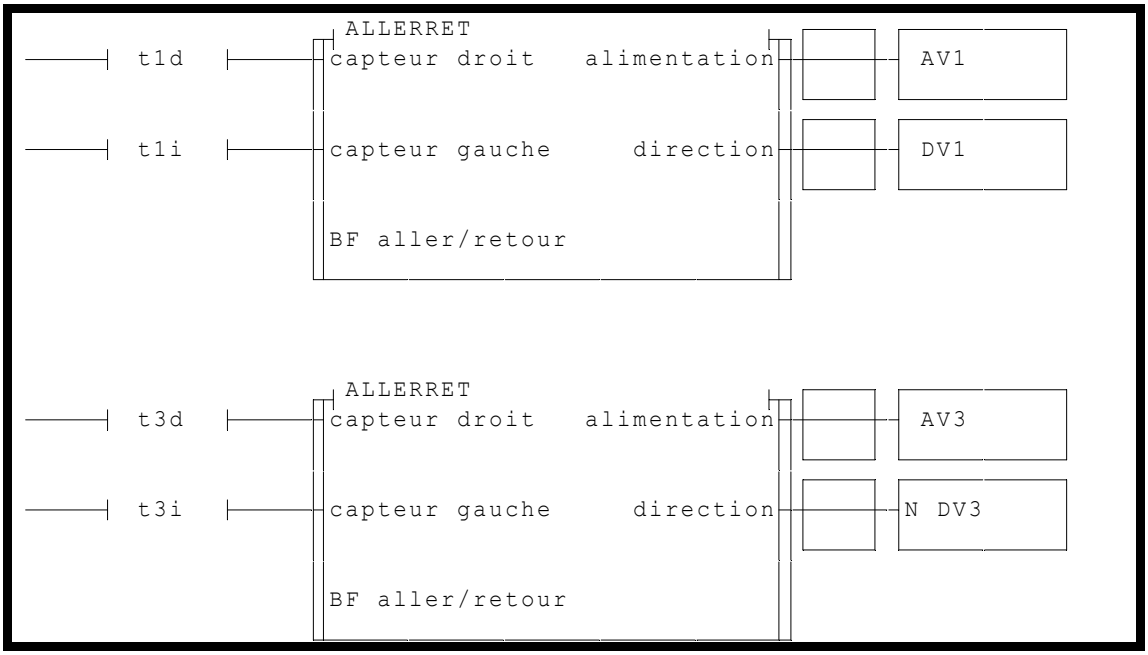
{O1}=(1)({I0})
{O1}=(0)({I1})

To illustrate the use of function blocks, let's complete our example.

Conditions :

Round trip of two locomotives on tracks 1 and 3.

Solution :



Example\bf\bloc-Functionnel 2.agn

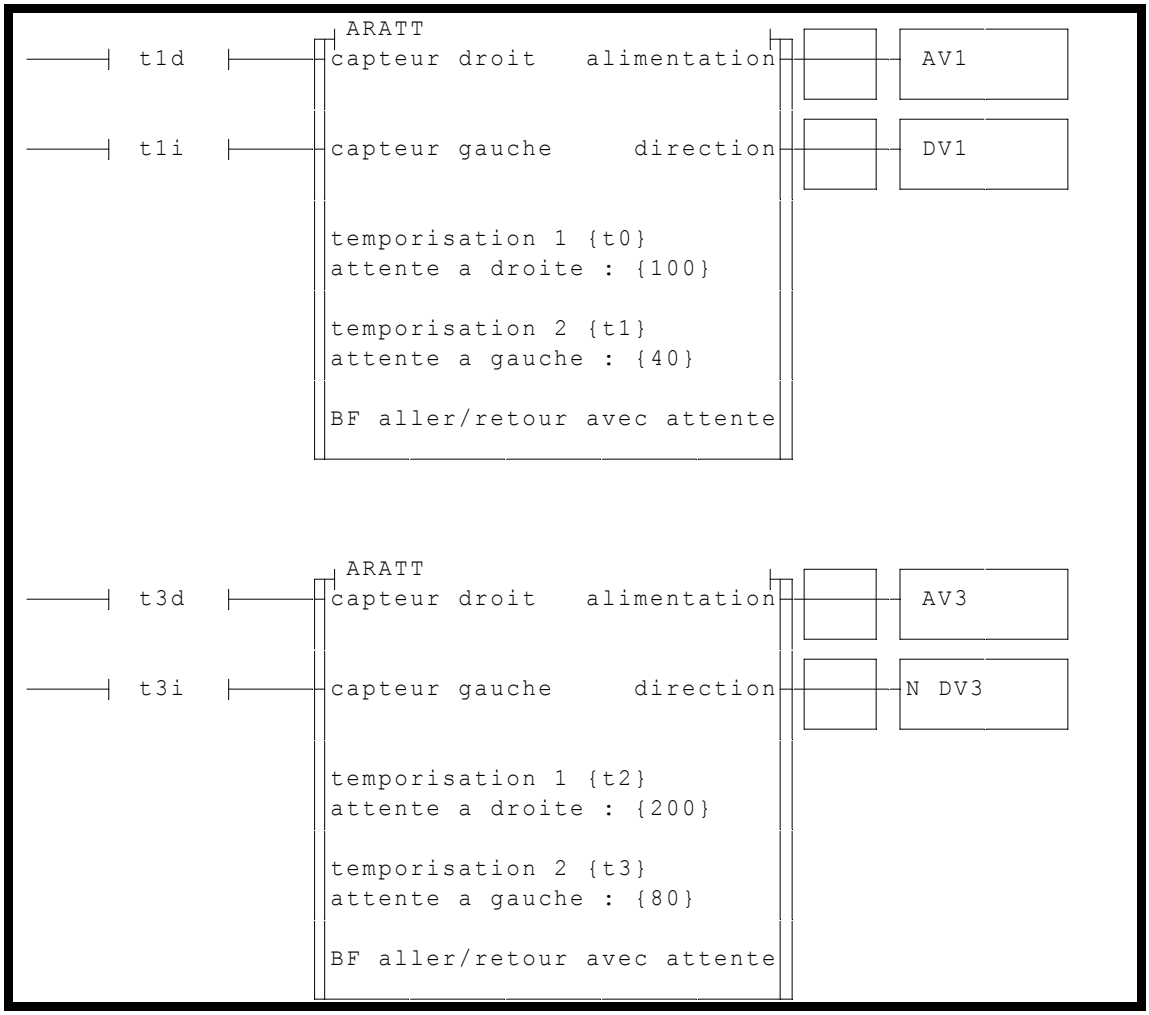
This example shows that with the same function block it is easy to make different modules of an operative party function in the identical manner.

Let's complete our example to illustrate the use of parameters

Conditions :

The two locomotives must make a delay at the end of the track. For locomotive 1: 10 seconds on the right and 4 seconds on the left, for locomotive 2: 20 seconds on the right and 8 seconds on the left.

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})
```

Example\bf\bloc-Functionnel 3.agn

1.15.6. Supplementary syntax

Supplementary syntax is used to make a calculation on the reference variable numbers in the « .LIB » file.

The syntax « ~+n » added after a reference to a variable or a parameter, adds n.

The syntax « ~-n » added after a reference to a variable or a parameter subtracts n.

The syntax « ~*n » added after a reference to a variable or parameter, multiplies by n.

It is possible to write many of these commands, one after the other, they are evaluated from left to right.

This mechanism is useful when a function block parameter needs to be used to refer to a table of variables.

Examples :

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

referring to the following element the first parameter, for example if the first parameter is m200 this syntax refers to m201.

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

referring to the third parameter multiplied by 100 plus 200, for example if the third parameter is 1 that syntax refers to M 1*100 + 200 thus M300.

1.16. Evolved function blocks

This functionality is used to create very powerful function blocks with greater ease than the function blocks managed by files written in literal language. This programming method uses a functional analysis approach.
It does not matter which sheet or set of sheets become a function block (sometimes called encapsulating a program).
The sheet or sheet which describe the functionality of a function block can access variables which are outside the function block: block boolean inputs, boolean outputs and parameters.
Principles for use and more importantly the use of external variables is identical to the old function blocks.

1.16.1. Syntax

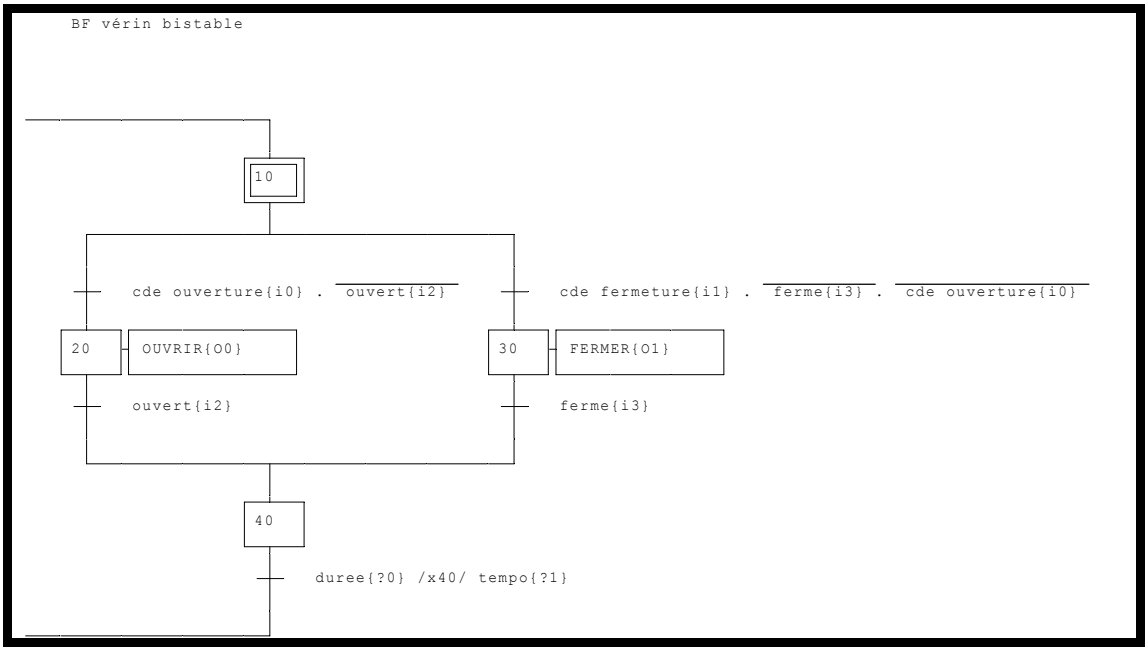
To refer a variable outside a function block it is necessary to use a mnemonic included in the following text : {In} to refer the boolean input n, {On} to refer the boolean output n, {?n} to refer parameter n. The mnemonic must start with a letter.

1.16.2. Differentiating between new and old function blocks

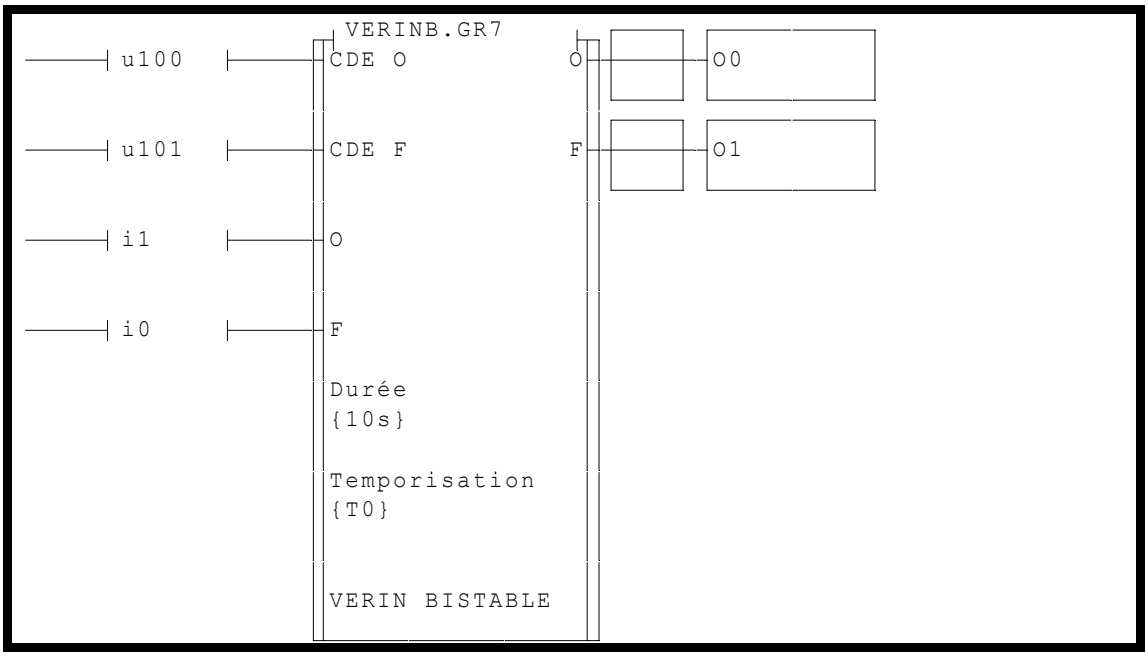
The file name written on the function block drawing indicates if it is an old (managed by an LIB file) or new function block (managed by a GR7 sheet). The name of an old function block does not have an extension, for a new one the extension GR 7 must be added. The sheet containing the code which manages the functionality of the function block must be entered in the list of project sheets. In the sheet properties « Function Block » must be selected.

1.16.3. Example

Contents of VERINB sheet :



Call up a function block



Example\bf\bloc-Functionnel 3.agn

1.17. Predefined function blocks

Conversion function blocks are located in the sub-directory « \LIB » of the directory where AUTOMGEN is installed.

The equivalents in macro-instructions are also present, see chapter 1.10.3. .

To insert a function blocks and its parameters in an application select the command « Insert function block » from the« Tools » menu.

1.17.1. Conversion blocks

ASCTOBIN : converts ASCII to binary

BCDTOBIN : converts BCD to binary

BINTOASC : converts binary to ASCII

BINTOBCD : converts binary to BCD

GRAYTOB : converts gray code to binary

16BINTOM : transfers 16 boolean variables to a word

MTO16 BIN : transfers a word to 16 boolean variables

1.17.2. Time delay blocks

TEMPO : upstream time delay

PULSOR : parallel output

PULSE : time delay pulse

1.17.3. String blocks

STRCMP : comparison

STRCAT : concatenation

STRCPY : copy

STRLEN : calculate the length

STRUPR : set in lower case

STRLWR : set in upper case

1.17.4. Word table blocks

COMP : comparison

COPY : copy

FILL : fill

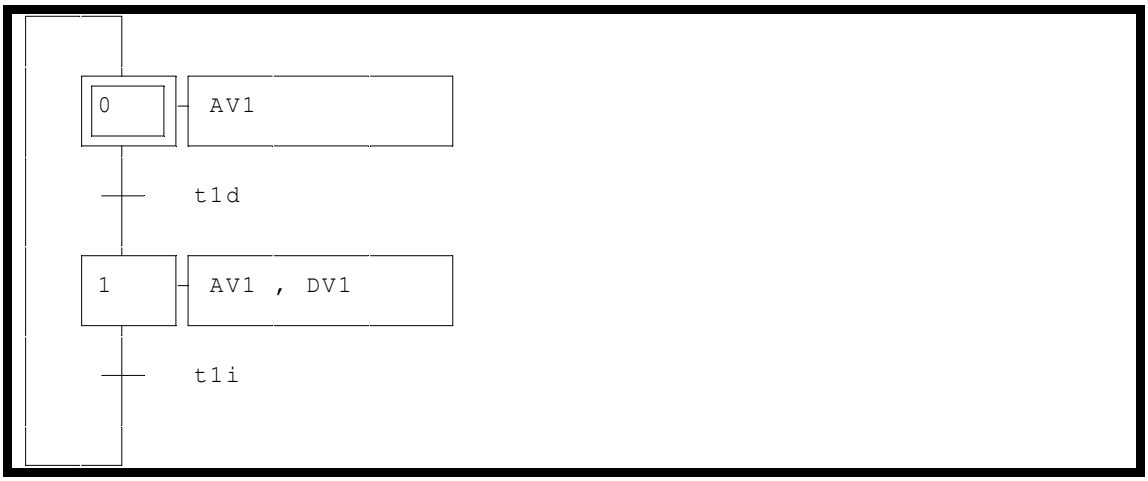
1.18. Advanced techniques

1.18.1. Compiler generated code

This chapter deal with the form of code generated by compilation of such or that type of program.

The utility « CODELIST.EXE »* is used to translate « in clear » a file of intermediate code « .EQU » (also called pivot language).

We are going to do the following: load and compile the first programming example in the « Grafcet » chapter: « simple1.agn » from the directory « Example\grafcet » :



Double click on « Generated files/Pivot code » in the browser.

You will obtain the following list of instructions :

```
: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
```

* This utility must be executed statrting from the DOS command line.

This represents the translation of a « simple1.agn » application into low level literal language

The comments indicate where the portions of code came from, this is useful if an application is composed of multiple sheets.

Obtaining this list of instructions may be useful for answering questions regarding code generated for some program form or the use of some language.

In certain cases « critiques », for which it is important to know information such as « how many cycles does it take before this action becomes true ? » a step by step way and an in-depth examination of generated code will prove to be indispensable.

1.18.2. Optimizing generated code

Various levels of optimization are possible.

1.18.2.1. Optimizing compiler generated code

The compiler optimization option is used to greatly reduce the size of generated code. This command requires that the compiler manage fewer lines of low level literal language, consequently increasing compiling time.

Depending on the post-processors used, this option involves an improvement in the size of the code and/or the execution time. It is advisable to carry out some tests to determine if this command is of interest or not depending on the nature of the program and the type of target used.

Normally, it is useful with post-processors for Z targets.

1.18.2.2. Optimizing post-processor generated code

Each post-processor may possess options for optimizing generated code. For post-processors which generate construction code, see the corresponding information.

1.18.2.3. Optimizing cycle time : reducing the number of time delays on Z targets

For Z targets, the number of stated time delays directly affects the cycle time. Try to state the minimum time delays based on the application requirements.

1.18.2.4. Optimizing cycle time: canceling scanning of certain parts of the program

Only targets which accept JSR and RET instruction support this technique.
Special compilation commands are used to validate or « invalidate » scanning of certain parts of the program.
They are the sheets which define the portions of applications.
If an application is broken down into four sheets than each one can be separately « validated » or « invalidate ».
A command « #C(condition) » placed on the sheet conditions the searching of the sheet up to a sheet containing a « #R » command.

This condition must use the syntax established for the tests, see chapter 1.3.

Example :
If a sheet contains the two commands :

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

Then everything that it contains will not be executed except word 200 containing 4.

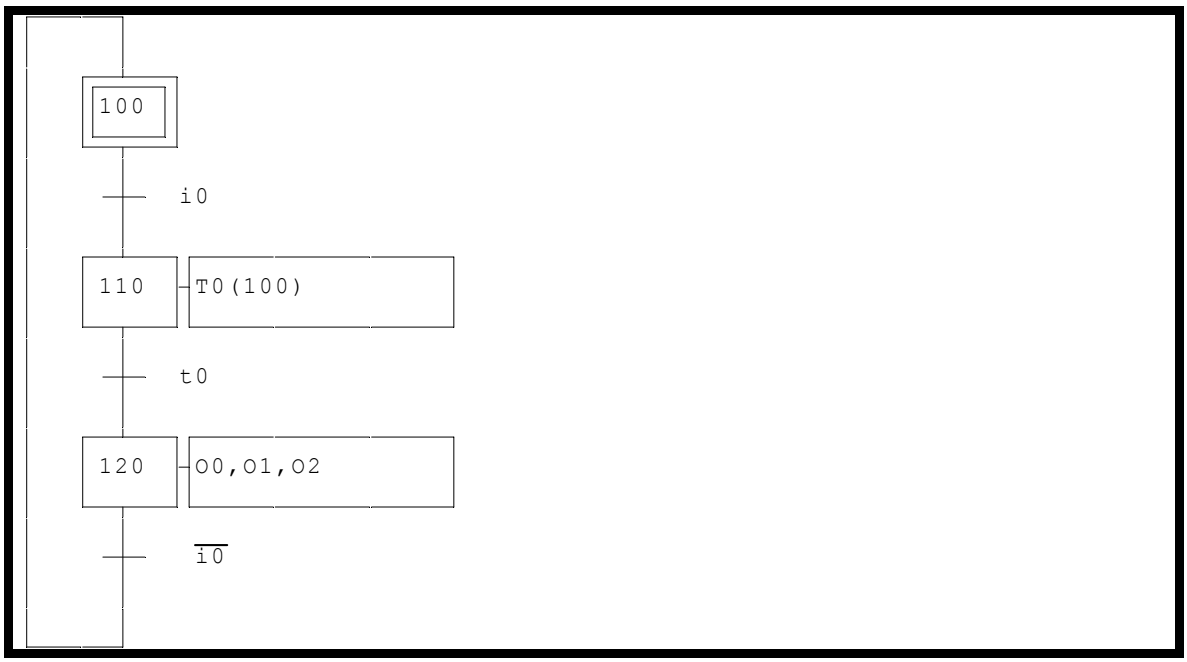
2. Examples

2.1. Regarding examples

This part contains a series of examples providing an illustration of the different programming possibilities offered by AUTOMGEN.
All of these examples are located in the « example » sub-directory in the directory where AUTOMGEN is installed.
This section contains the most complete and complex examples developed for a train model. The description of this model is located at the beginning of the language reference manual.

2.1.1. Simple grafcet

The first example is a simple line Grafcet



Example\grafcet\sample1.agn

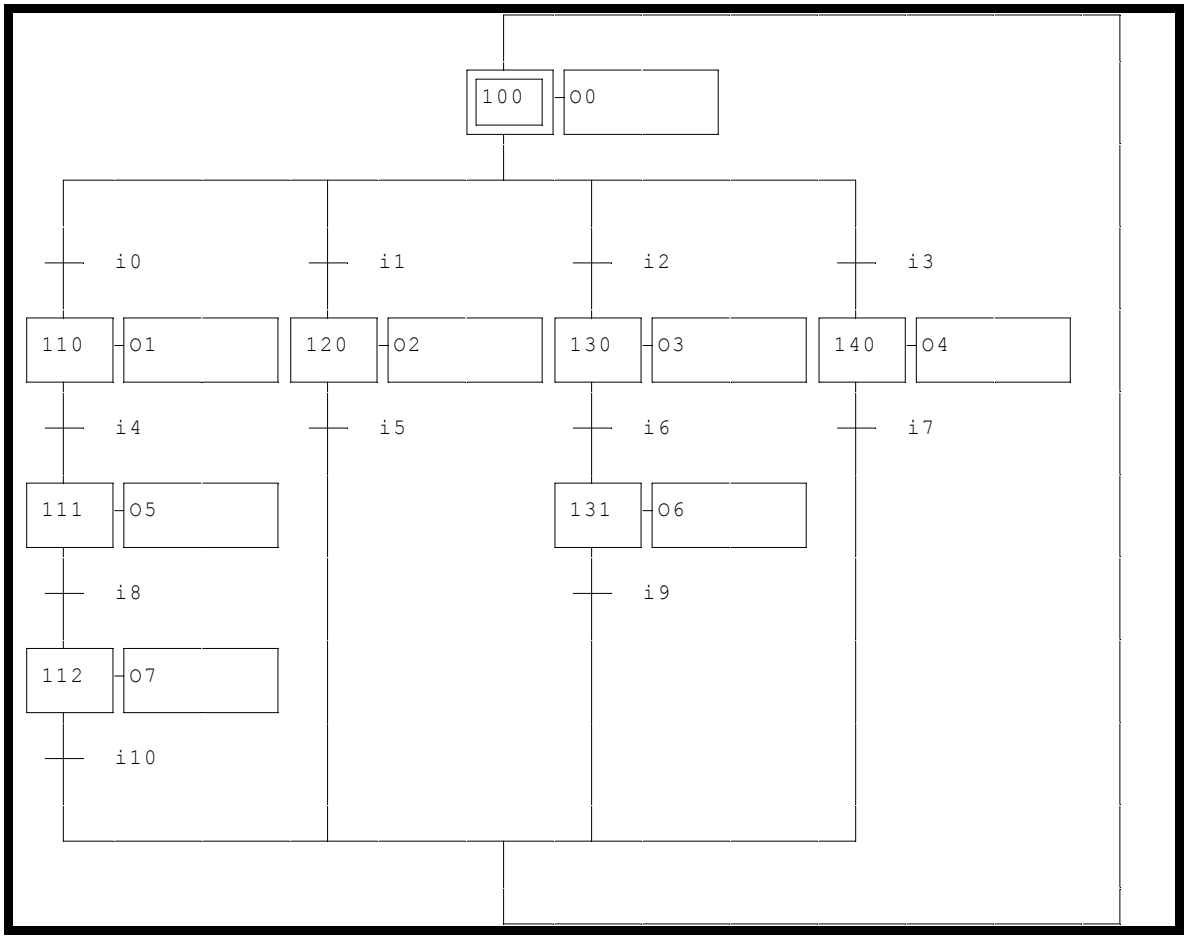
⇨ the transition in step 100 and step 110 is made up of a test on input 0,

⇨ step 110 activates the time delay 0 for a duration of 10 seconds, this time delay is used as a transition between step 110 and step 120,

⇨ step 120 activates outputs 0, 1 and 2,

⇨ the complement of input 0 will be the transition between step 120 and 100.

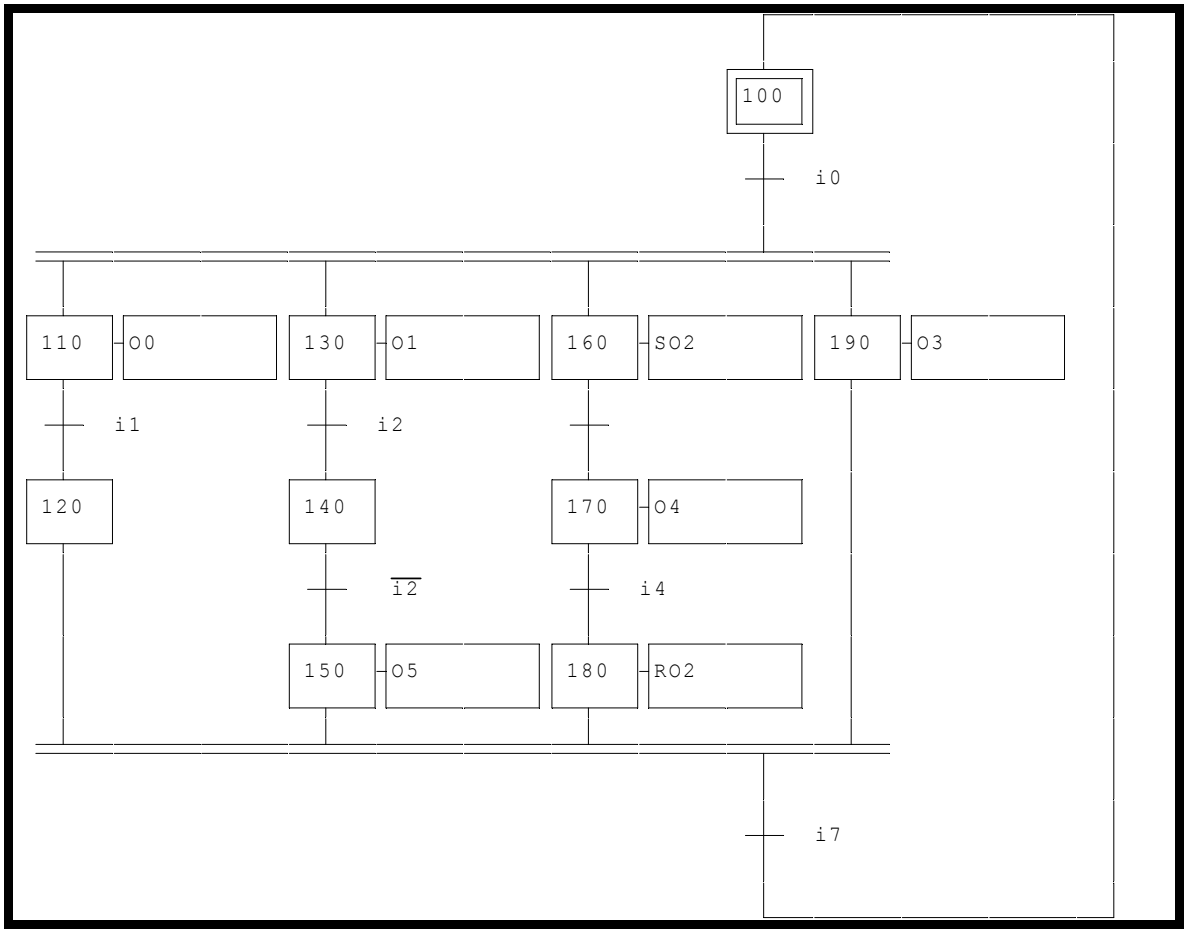
2.1.2. Grafcet with an OR divergence



Example\grafcet\sample2.agn

This example shows the use of « Or » divergences and convergences. The number of branches is not limited by the size of the sheet. It is a non-exclusive « Or » by standard. For example, if inputs 1 and 2 are active, then steps 120 and 130 will be set to one.

2.1.3. Grafcet with an AND divergence

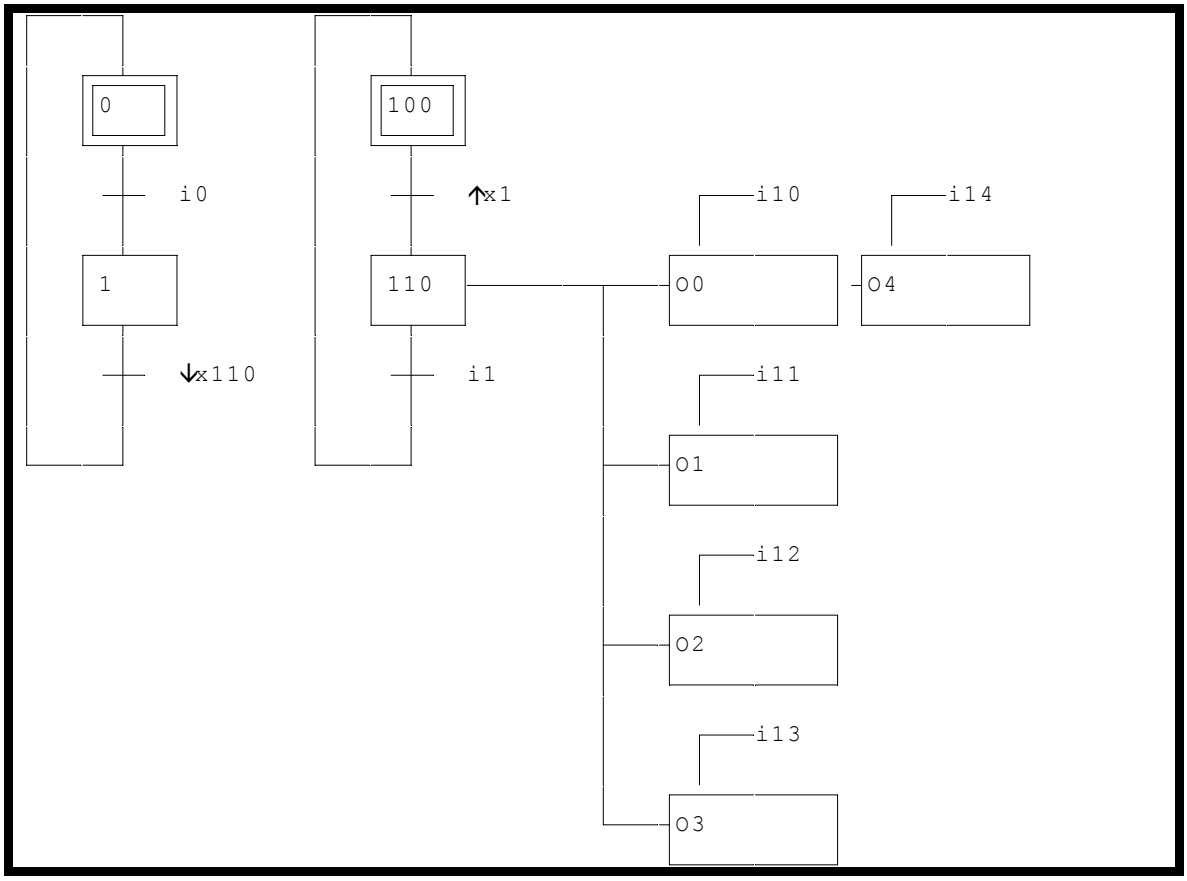


Example\grafcet\sample3.agn

This example shows the use of « And» divergences and convergences. The number of branches is not limited by the size of the sheet.. Also note the following points

- ⇨ a step may not lead to an action (case of steps 100, 120, and 140),
- ⇨ orders « S » and « R » were used with output o2 (steps 160 and 180),
- ⇨ the transition between step 160 and 170 is left blank, so it is always true, the syntax « =1 » could also have been used.

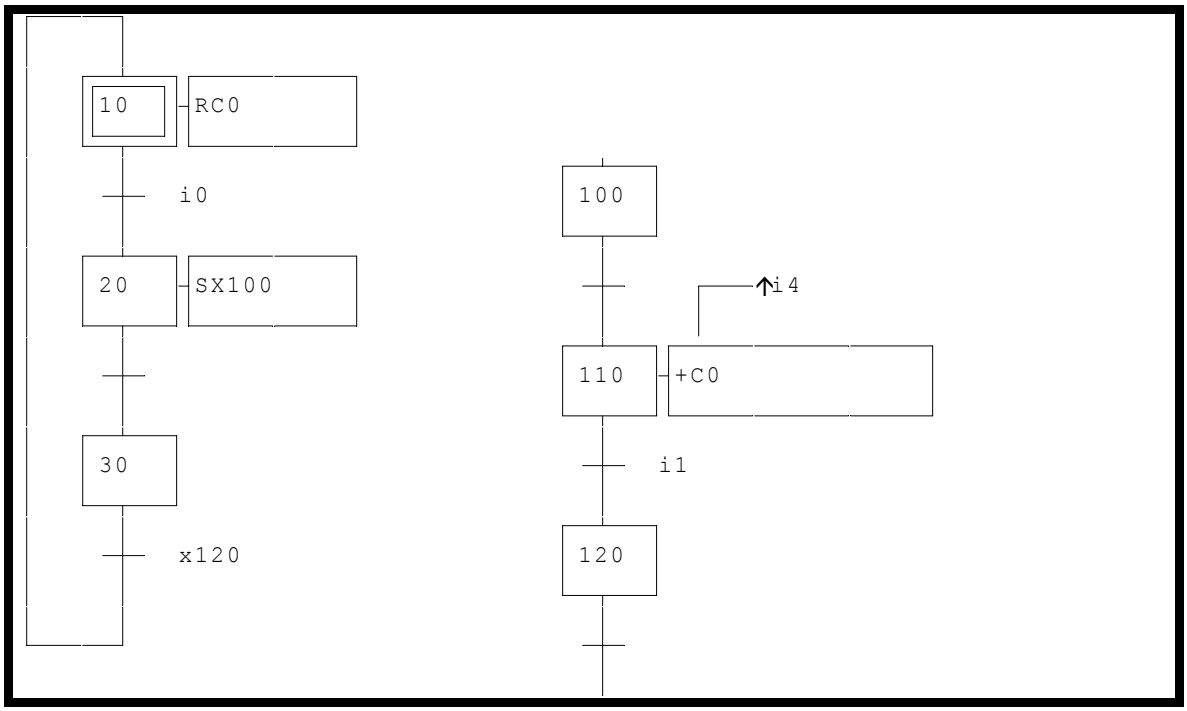
2.1.4. Grafcet and synchronization



Example\grafcet\sample4.agn

This example shows the possibilities AUTOMGEN offers for synchronizing multiple Grafcets. The transition between step 100 and 110 « ↑x1 » means « wait for a rising edge on 1 ». The transition« ↓x110 » means « wait for a falling edge on step 110 ». The step by step execution of this program shows the exact evolution of the variables and their front at each cycle. This makes it possible to understand exactly what happens during the execution. We can also see the use of multiple actions associated to step 110, which are individually conditioned here.

2.1.5. Step setting

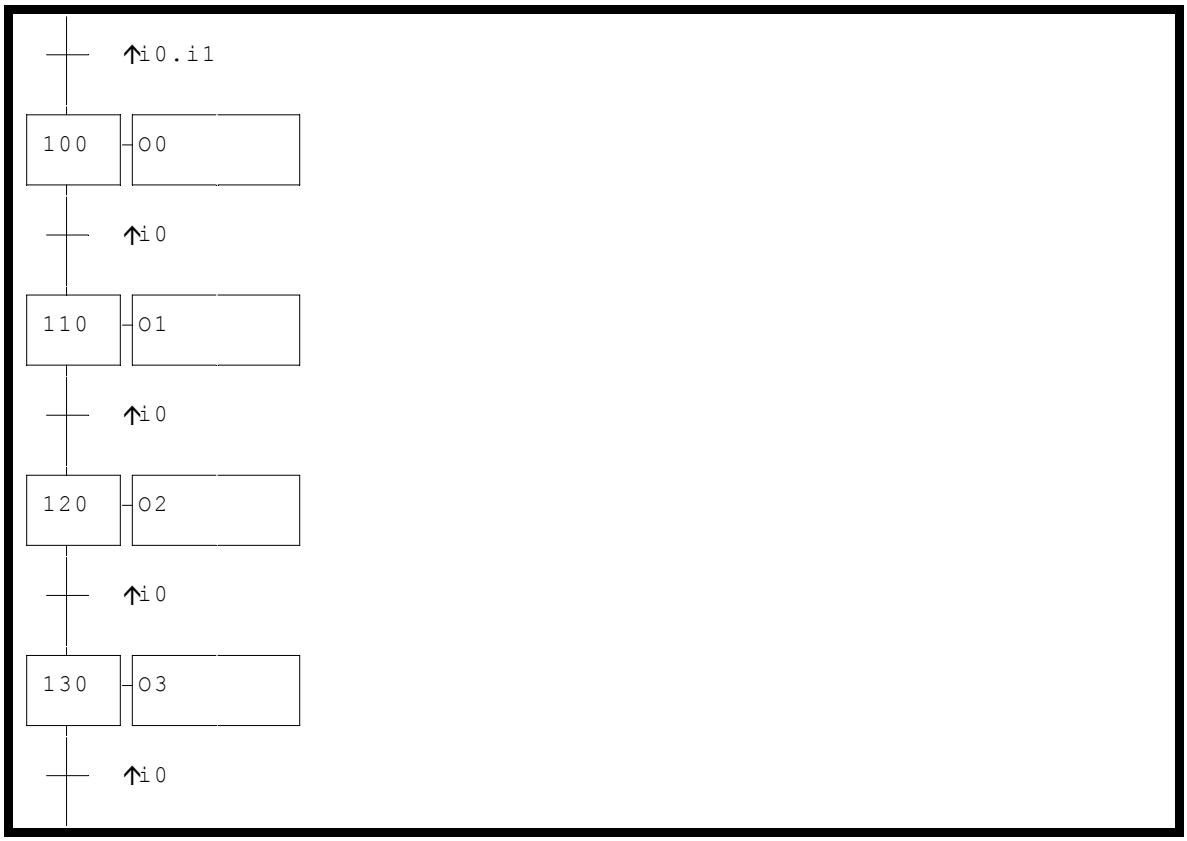


Example\grafcet\sample5.agn

In this example an order « S » (set to one) has been used to set a step. AUTOMGEN also authorizes setting of a Grafcet integer (see examples 8 and 9). Again in this example, the step by step execution lets us understand the exact evolution of the program over time. We can also see:

- ⇨ use of an non-looped Grafcet (100, 110, 120),
- ⇨ use of the order « RC0 » (reset by counter 0),
- ⇨ use of the order « +C0 » (incremented by counter 0),conditioned by the rising edge of input 4, due to incrementation by the counter, so it is necessary that step 100 be active and that a rising edge is detected on input 4.

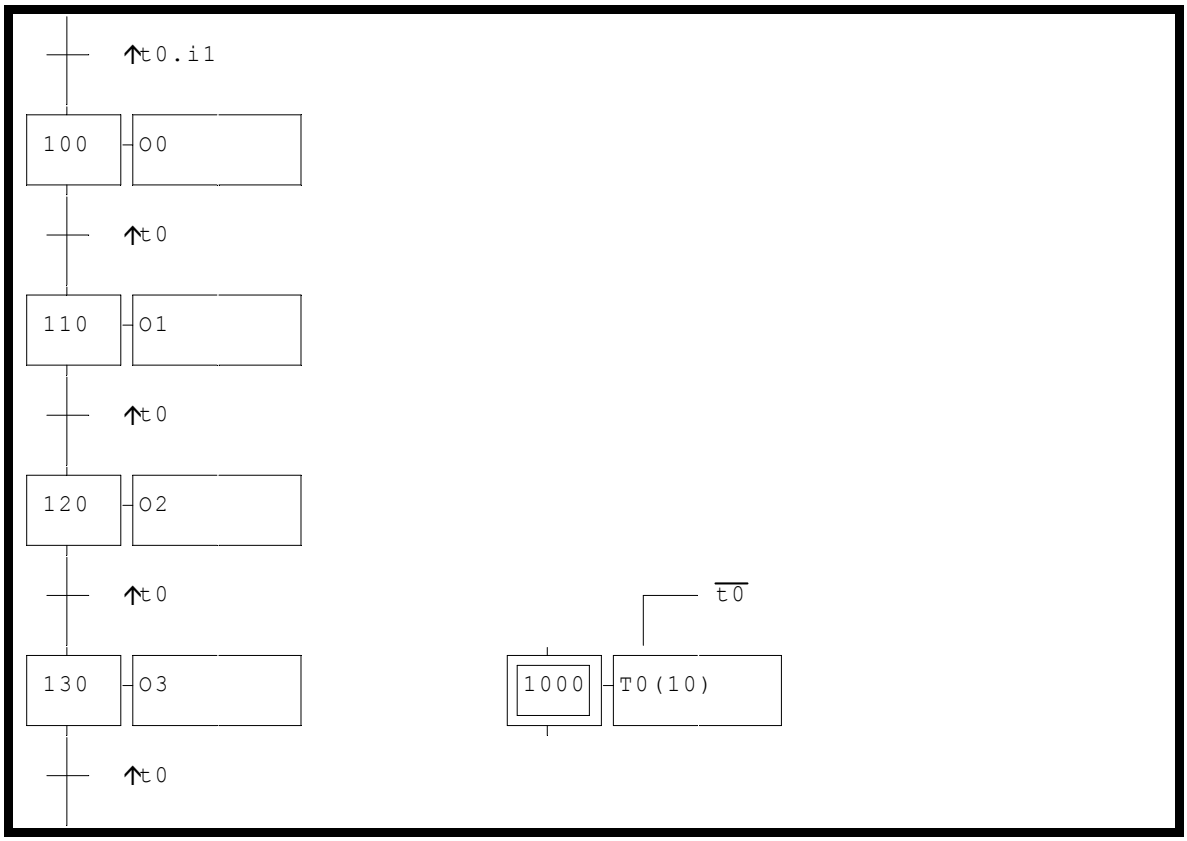
2.1.6. Destination and source steps



Example\grafcet\sample6.agn

We have already seen similar forms, where the first step is activated by another Grafcet. Here activation of step 100 is realized by the transition « ↑i0 . i1 » (rising edge of input 0 and input 1). This example represents a shift register. « i1 » is information to memorize in the register and « i0 » is the clock which makes the shift progress. Example 7 is a variation which uses a time delay as a clock.

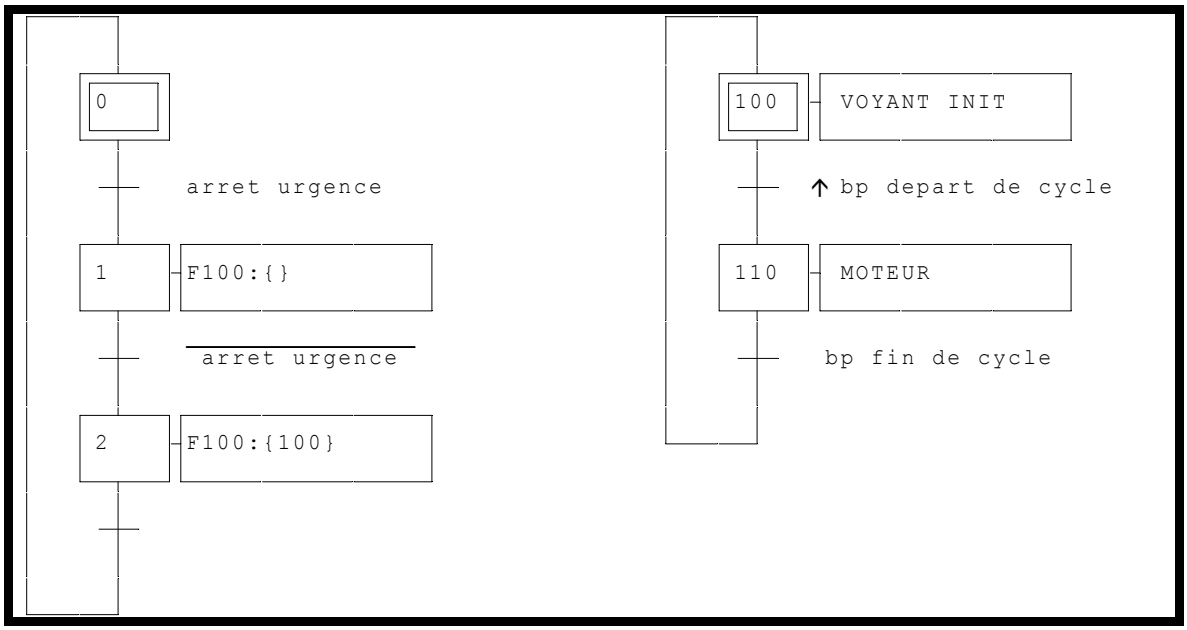
2.1.7. Destination and source steps



Example\grafcet\sample7.agn

Here again is the structure of the shift register used in example 6. This time the shift information is generated by a time delay (t0). « ↑t0 » represent the rising edge of the time delay, this information is true during a cycle when the time delay has finished. Step 1000 manages the launch of the time delay. The action of this step can be summed up as : « activate the count if it is not finished, otherwise reset the time delay». The functionality diagram of the time delays of this manual will help you to understand the functionality of this program.

2.1.8. Setting Grafcets

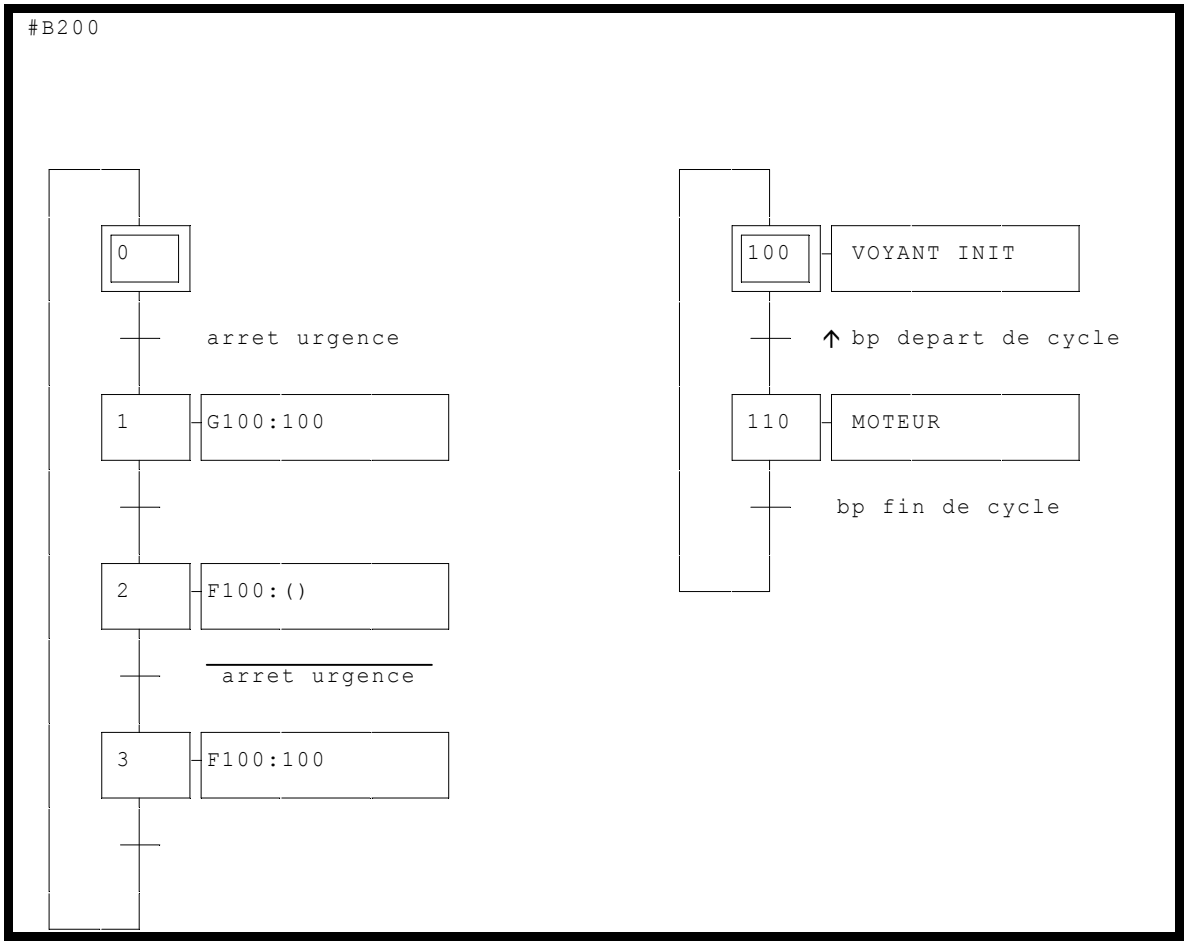


Example\grafcet\sample8.agn

This example illustrates the use of a Grafcet set command. The order « F100:{} » means « reset all the Grafcet steps where one of the steps bears the number 100 ». Order « F100:{100} » is identical but sets step 100 to 1. We have used symbols for this example :

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

2.1.9. Memorizing Grafcets

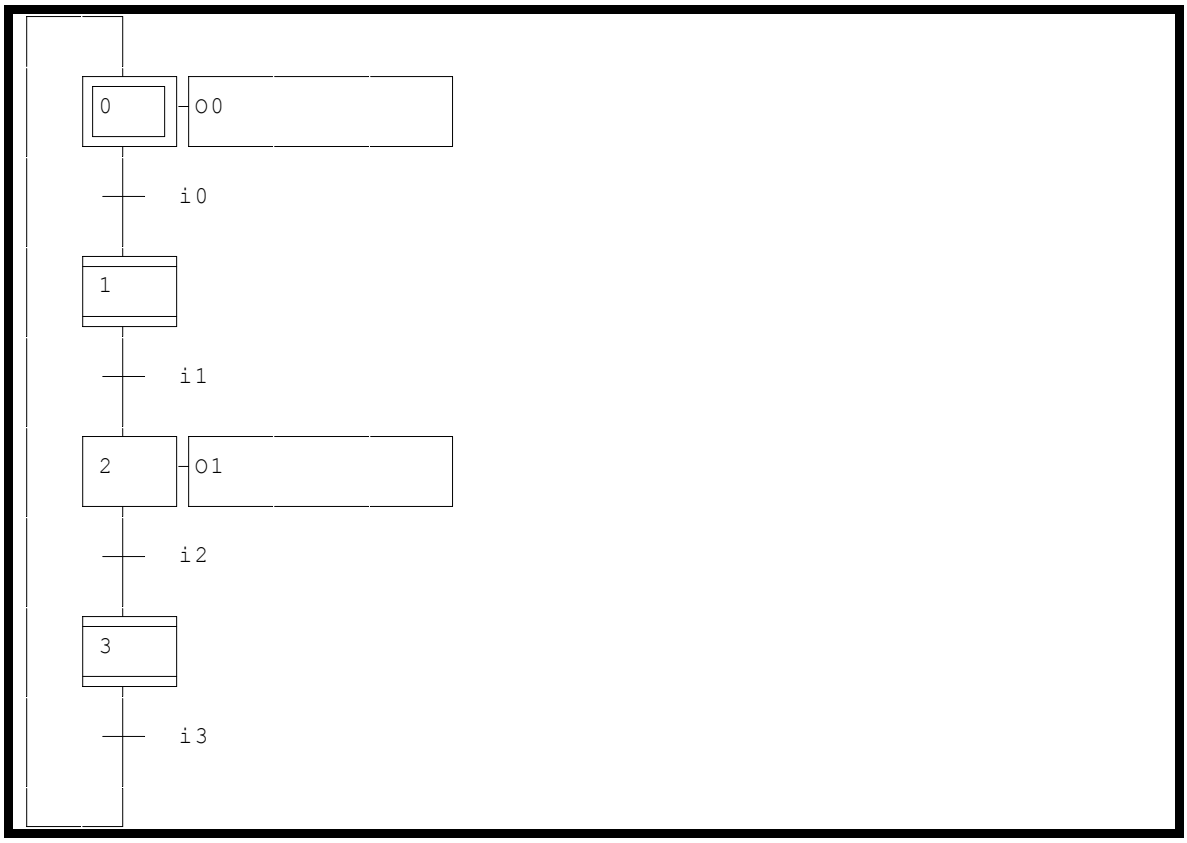


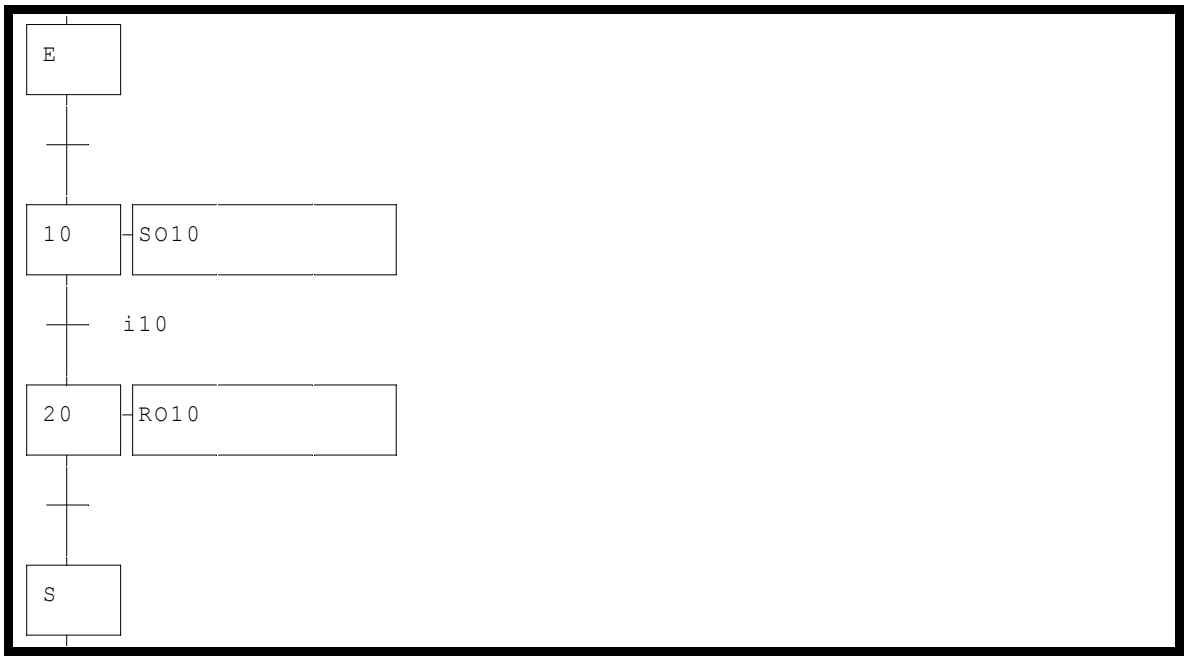
Example\grafcet\sample9.agn

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

This example is a variation of the previous program. The order « G100:100 » of step 1 memorizes the Grafcet production state before it is reset. When it starts again the production Grafcet will be put back in the state or the state it was in before the break, with order « F100:100 ». The Grafcet production state is memorized starting from bit 100 (this is the second parameter of orders « F » and « G » which indicates this site), command « #B200 » reserves bits u100 to u199 for this type of use. We can see that a « #B102 » command would have been sufficient here because the production Grafcet only needed two bits to be memorized (one bit per step).

2.1.10. Grafcet and macro-steps

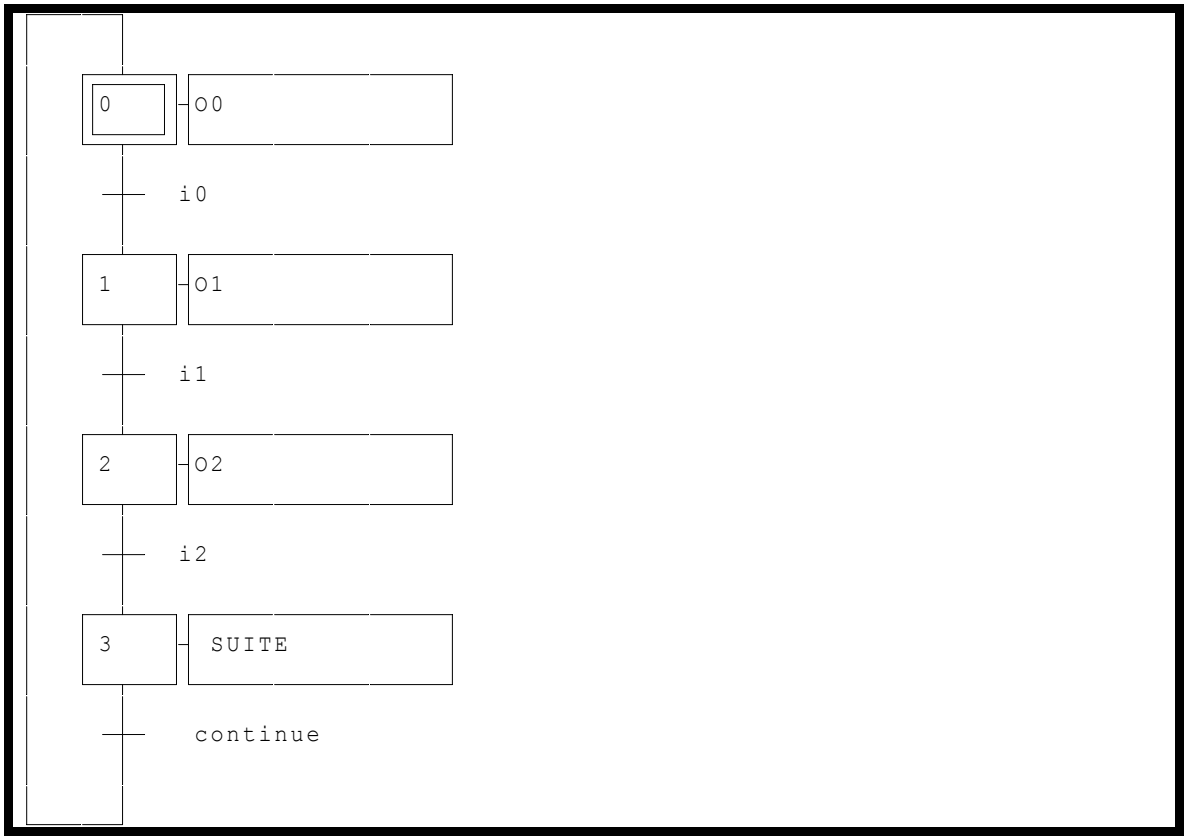


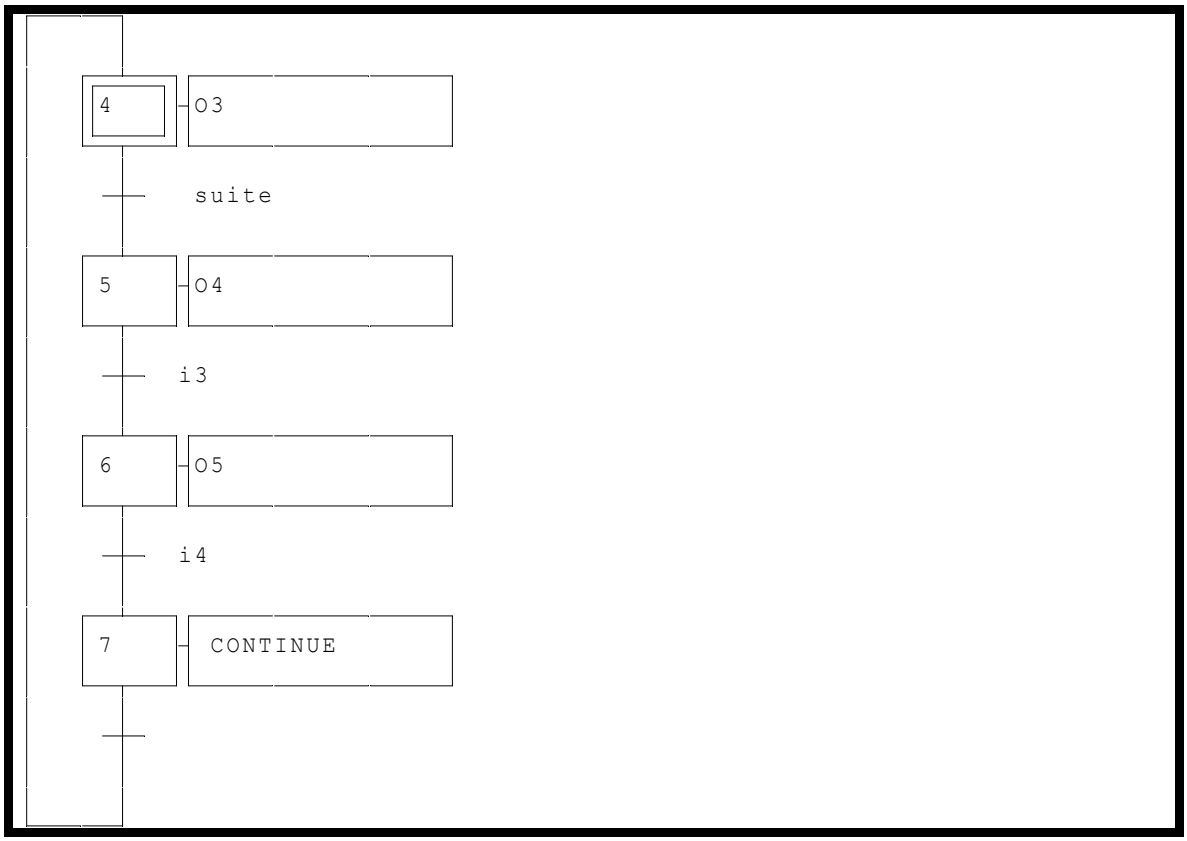


Example\grafcet\sample11.agn

This example illustrate the use of macro-steps. The «Macro-step 1 » and « Macro-step 3 » sheets represent the expansion of macro-steps with the input and output steps. Steps 1 and 3 of the «Main program » sheet are defined as macro-steps. Access to macro-step expansion display can be done by clicking the left side of the mouse on the macro-steps.

2.1.11. Linked sheets

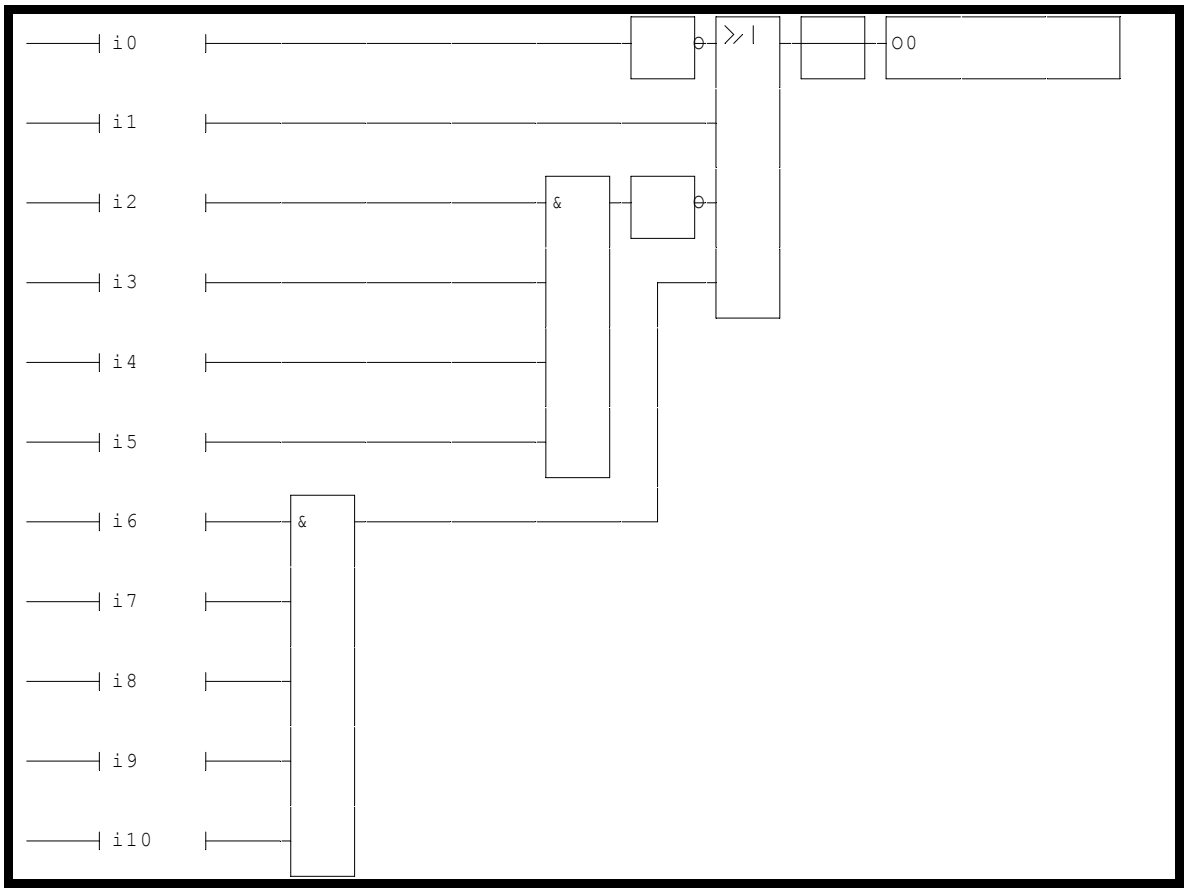




Example\grafcet\sample12.agn

In this example two sheets have been used to write a program. The symbols « _NEXT_ » and « _CONTINUE_ » have been stated as bits (see the symbol file) and are used to make a link between the two Grafcets (this is another synchronization technique that can be used with AUTOMGEN).

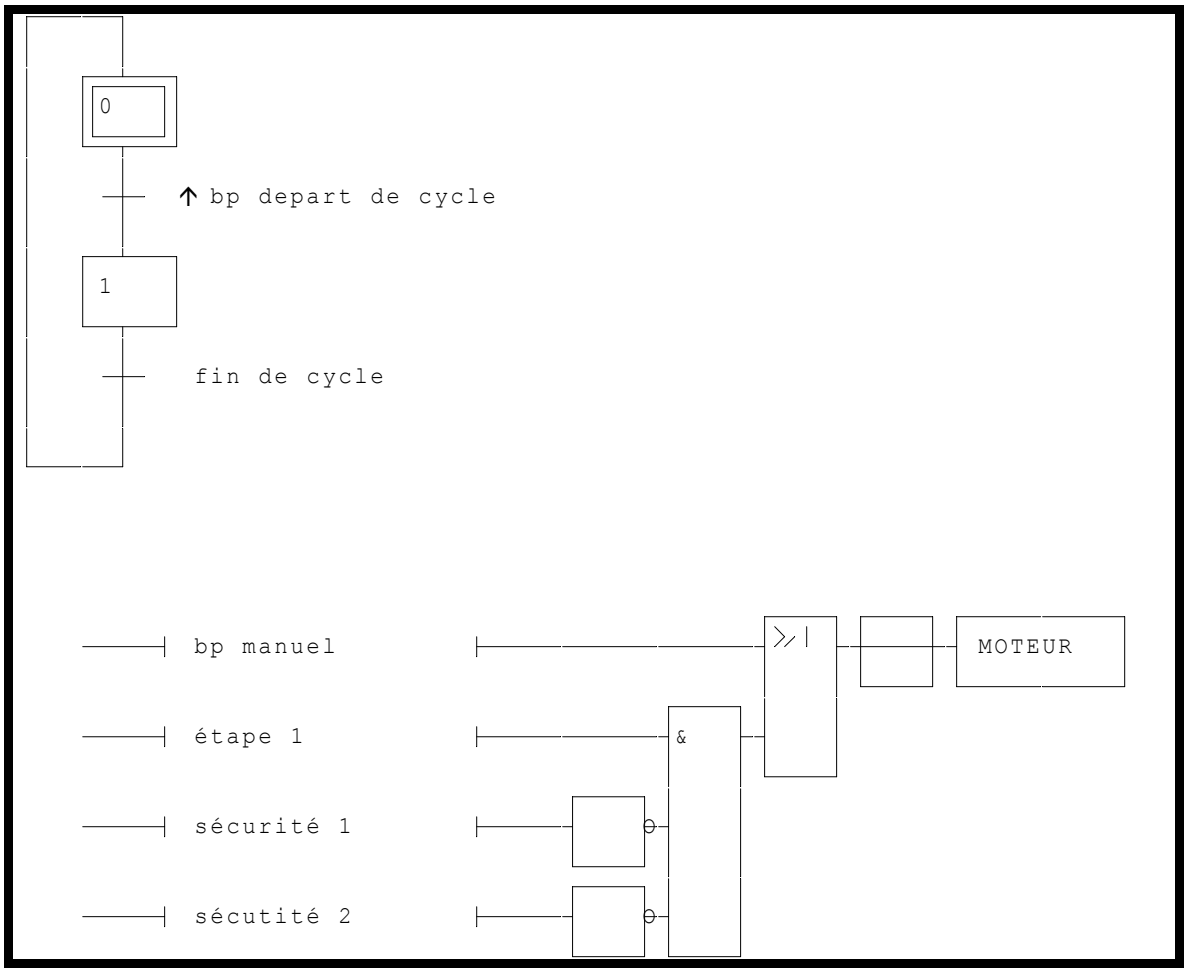
2.1.12. Flow chart



Example\logigramme\sample14.agn

The flow chart example shows the use of different blocks: the assignment block associated to key [0] to the left of the action rectangle the « no » block associated with key [1] which complements a signal and the test fixing blocks and « And» and « Or ».functions

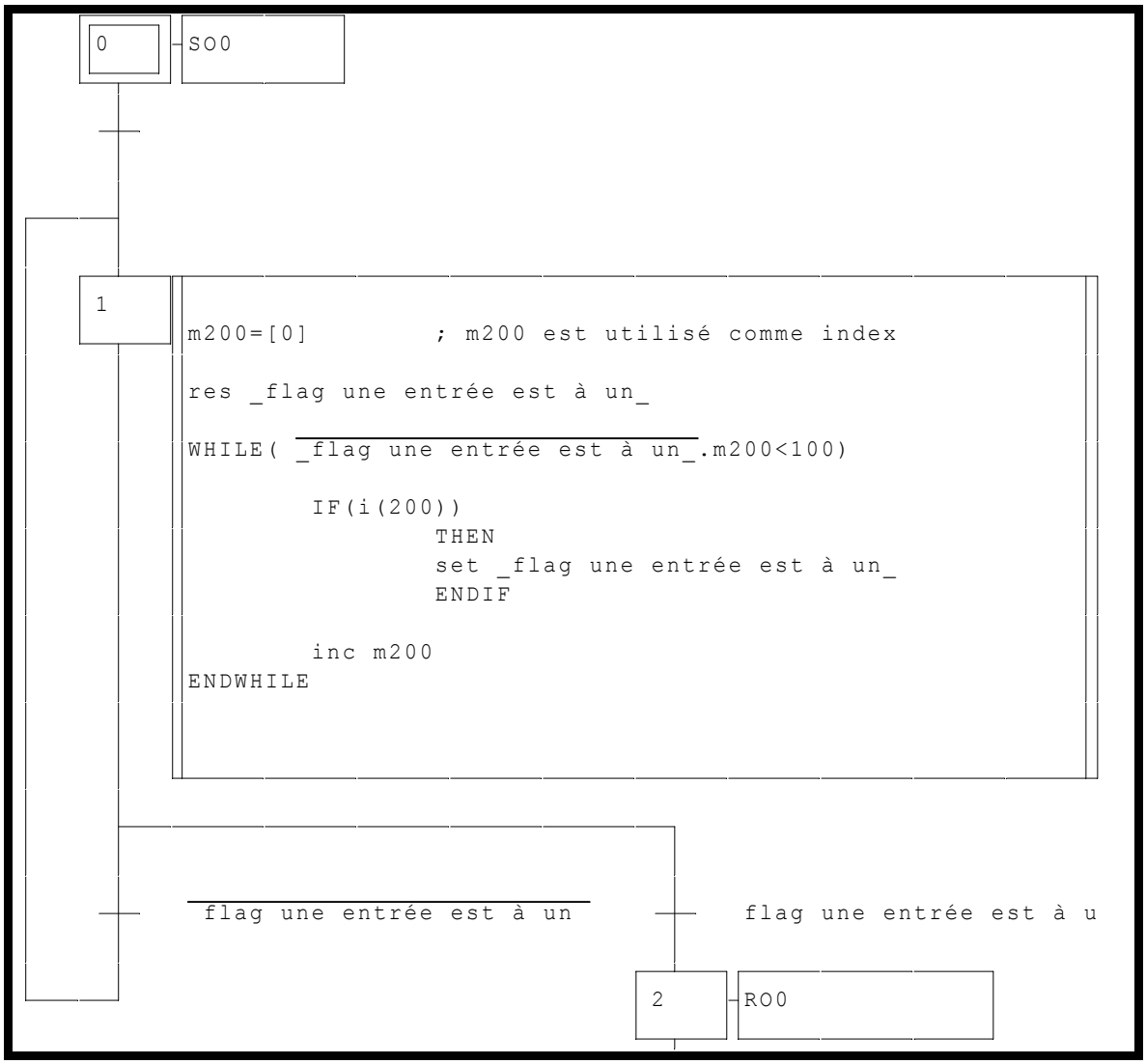
2.1.13. Grafcet and Flow Chart



Example\logigramme\exempl15.agn

In this example a Grafcet and a Flow Chart are used together. The symbol « _step1_ » used in the flow chart is associated to variable « x1 ».
This type of programming clearly displays activation conditions of an output.

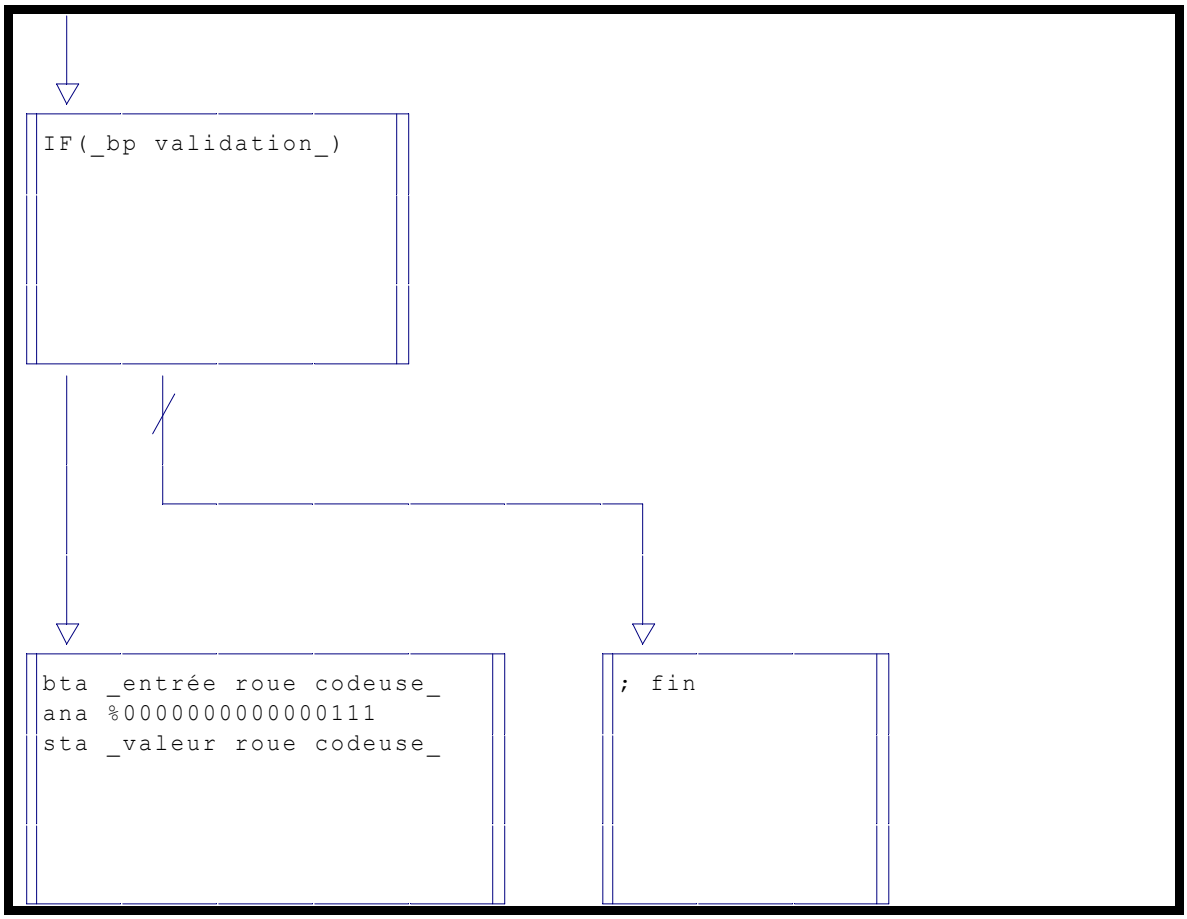
2.1.14. Literal language box



Example\lit\sample16.agn

This program which associates Grafcet and literal language box is for testing inputs i0 to i99. If one of the inputs is at one, then step 2 is active and the Grafcet is in a state where all evolution is prohibited. The symbol. « _flag an input is at one_ » is associated to bit u500. An indexed addressing is used to scan the 100 inputs. We can also see the simultaneous use of low level and extended literal language.

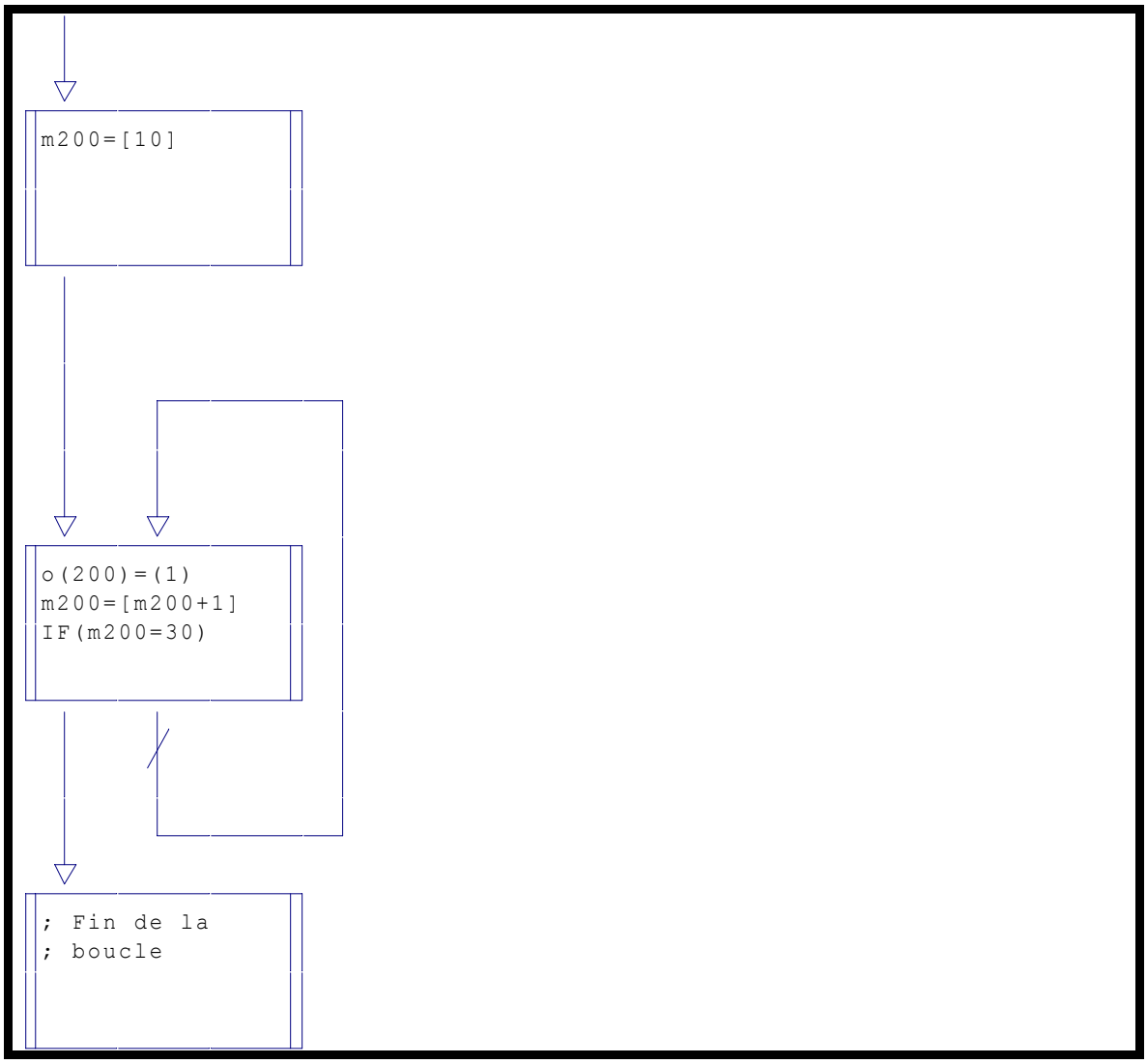
2.1.15. Organizational chart



Example\organigramme\sample18.agn

This example shows the use of an organizational chart for effecting an algorithmic and numeric treatment. Here three inputs from a code wheel is read and stored in a word if a validation input is active.

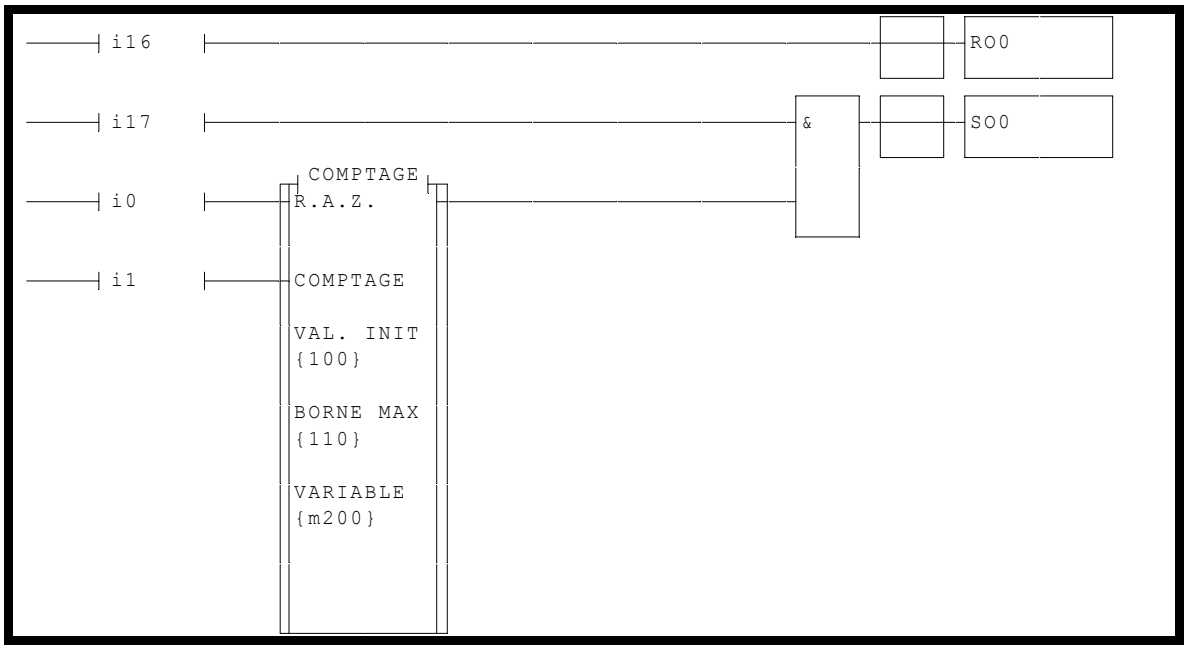
2.1.16. Organizational chart



Example\organigramme\sample19.agn

This second example of an organizational chart creates a loop structure which is used to set a series of outputs (o10 to o29) with an indirect addressing(« o(200) »).

2.1.17. Function block



Example\bf\sample20.agn

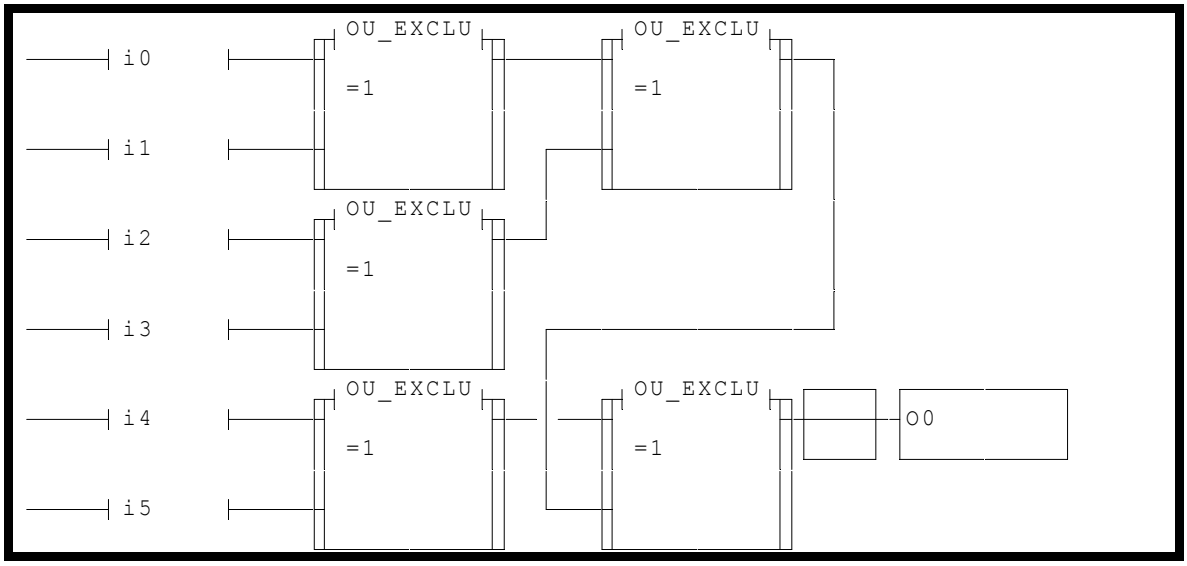
```
; Gestion de l'entrée de RAZ
IF({I0})
        THEN
        {?2}=[{?0}]
        ENDIF

; 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
```

count lib (included in project resources)

This example illustrates the use of a function block. The functions of the« COUNT » block that we have defined here are as follows :

- ⇨ the count will start from an init value and will finish at a maximum limit value
- ⇨ while the count value waits for the maximum limit it will be set the initial value and the block output will pass to one during a program cycle.,
- ⇨ the block will have a RAZ boolean input and a count input on the rising edge.

2.1.18. Function block



Example\bf\sample21.agn

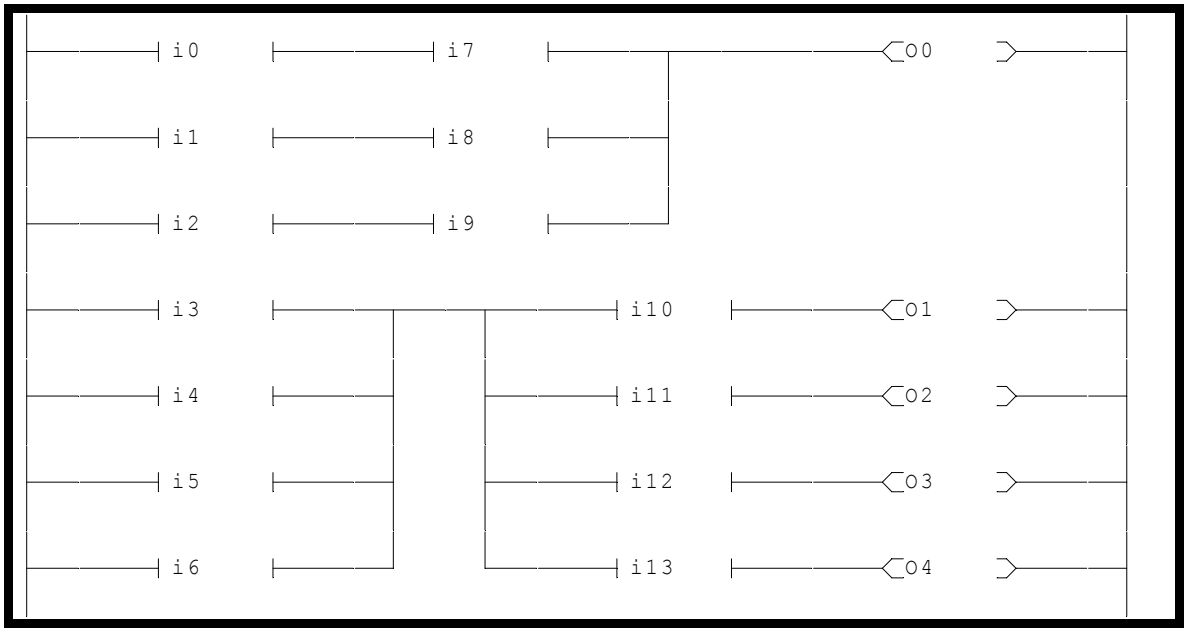
; Ou exclusif
neq {o0} orr /{i0} eor {i1} orr {i0} eor /{i1}

ou_exclu.lib (included in the project resources)

This second example of a function block shows the multiple use of the same block. The « EXCLUSIVE_OR » block creates an exclusive or between the two boolean inputs This example uses 5 blocks to create an exclusive or among 6 inputs (i0 à i5). The « EXCLUSIVE_OR.LIB » listed below supports the functionality of the block. The exclusive or boolean equation is as follows : « (i0./i1)+(/i0.i1) ».

The equivalent form used here makes it possible to code the equation on a single line of low level literal language without using intermediate variables.

2.1.19. Ladder



Example\laddersample22.agn

This example illustrates the use of ladder programming.

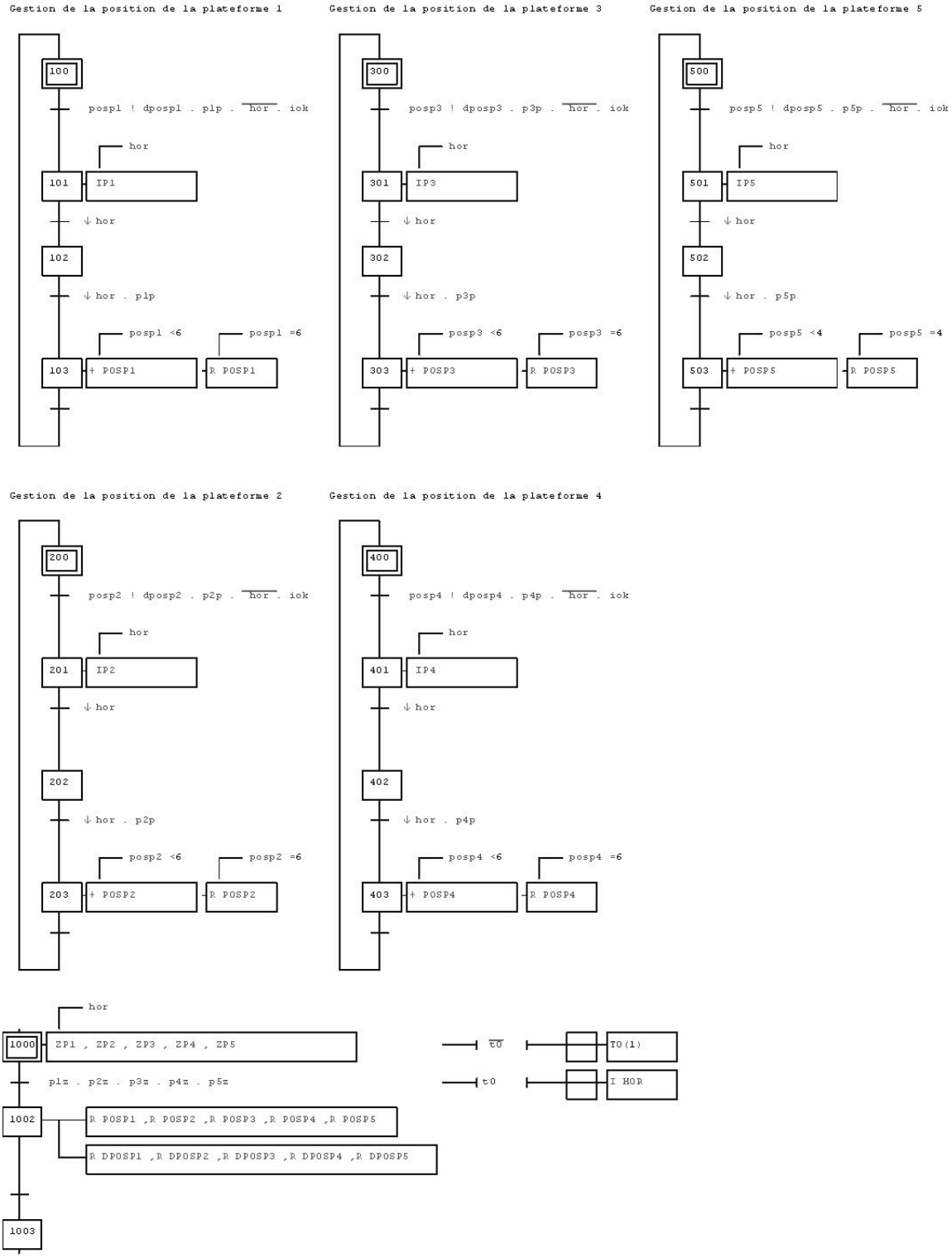
2.1.20. Example developed on a train model

AT850

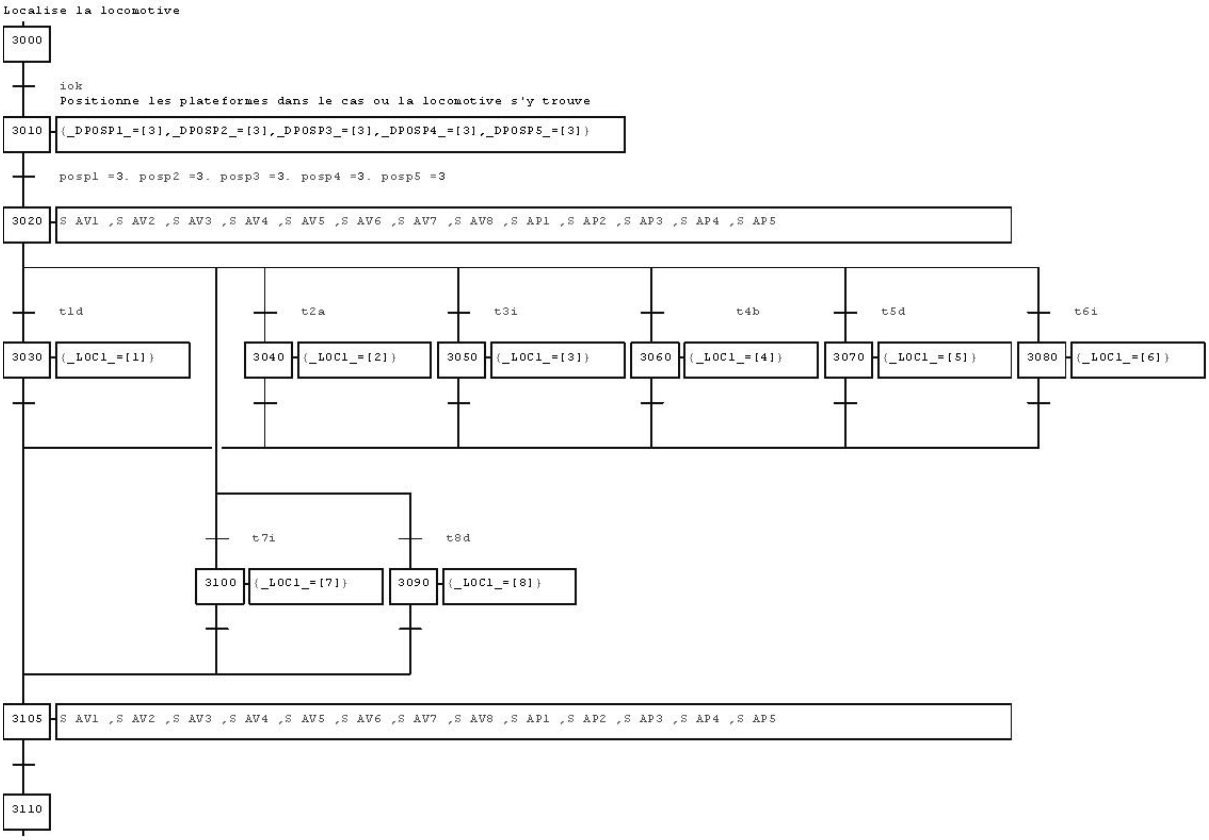
AUTOMGEN 7 - www.irai.com

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

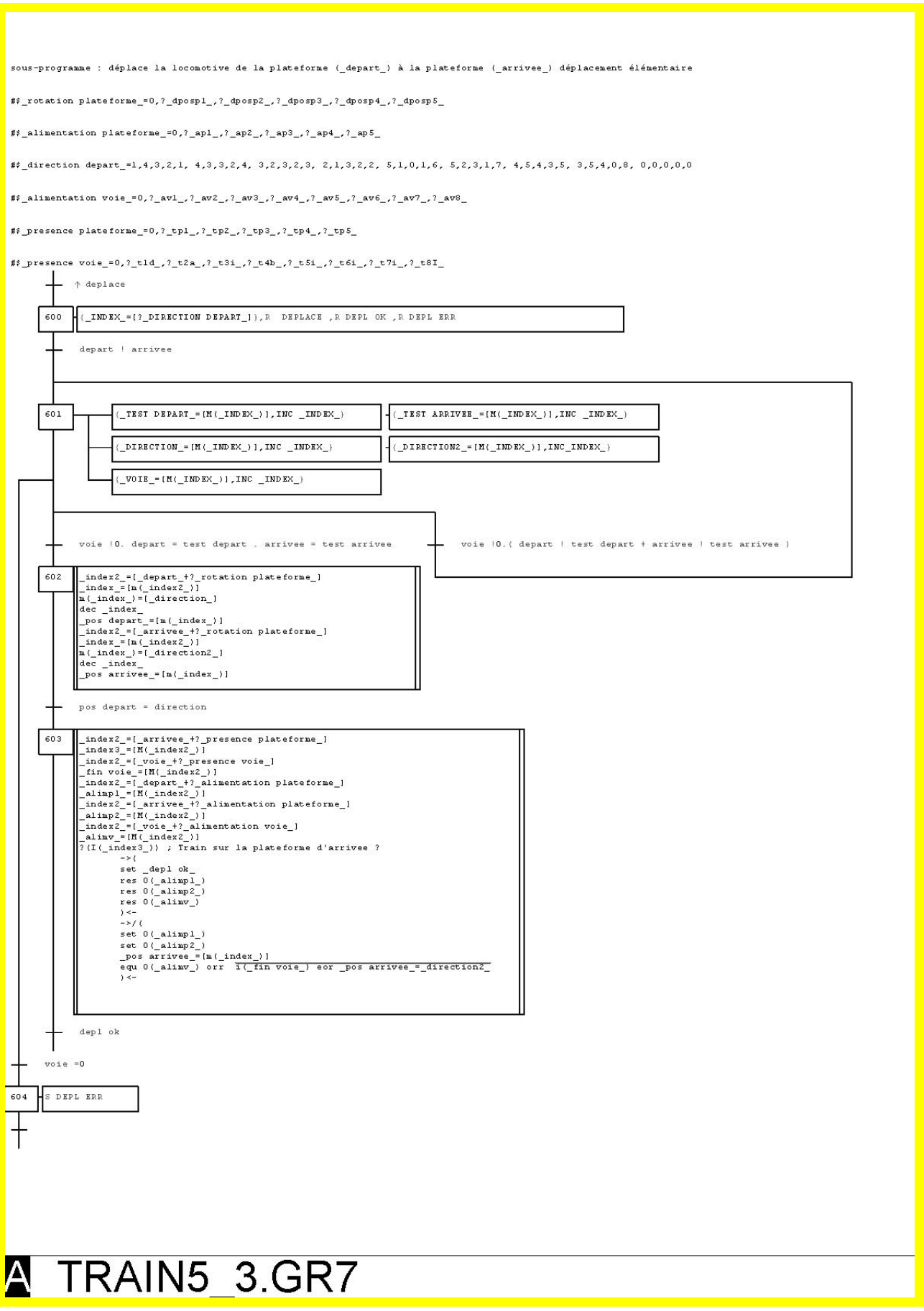
1/1 Symboles

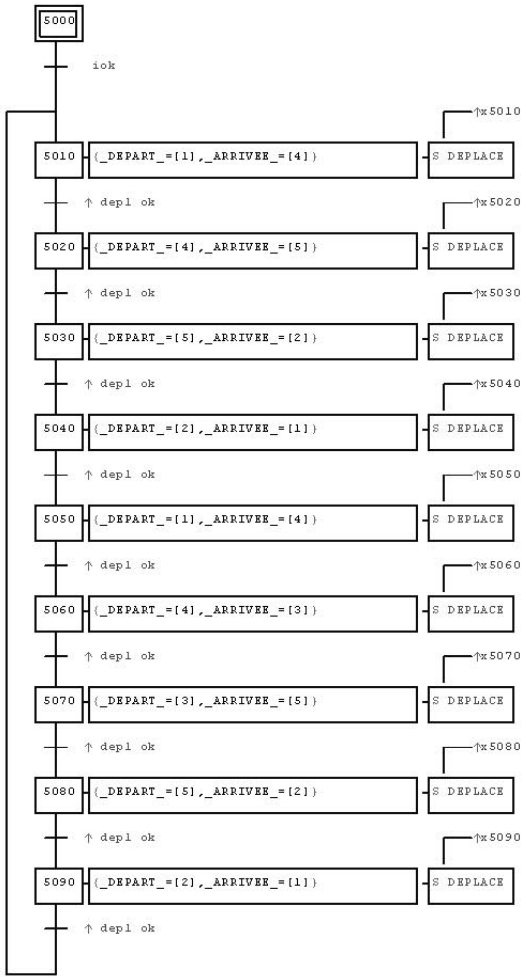

Gestion de la position de la plateforme 1
100
posp1 ! dposp1 . p1p . hor . iok
hor
101
IP1
↓ hor
102
↓ hor . p1p
posp1 <6
posp1 =6
103
+ POSP1
R POSP1
Gestion de la position de la plateforme 3
300
posp3 ! dposp3 . p3p . hor . iok
hor
301
IP3
↓ hor
302
↓ hor . p3p
posp3 <6
posp3 =6
303
+ POSP3
R POSP3
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

TRAIN5.GR7



TRAIN5_2.GR7

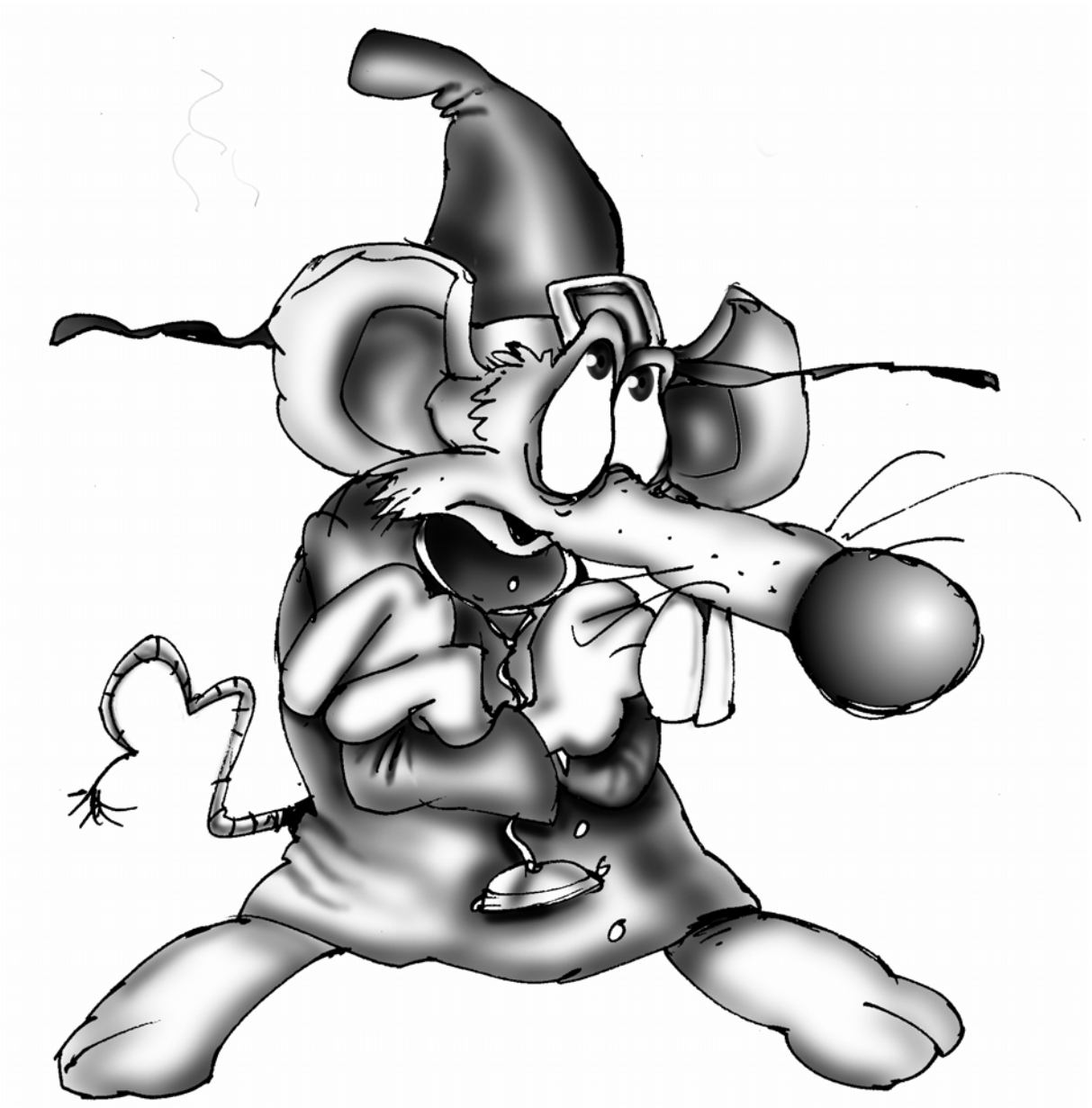

sous-programme : déplace la locomotive de la plateforme (_depart_) à la plateforme (_arrivee_) déplacement élémentaire
#$_rotation plateforme_=0,?_dposp1_,?_dposp2_,?_dposp3_,?_dposp4_,?_dposp5_
#$_alimentation plateforme_=0,?_ap1_,?_ap2_,?_ap3_,?_ap4_,?_ap5_
#$_direction depart_=1,4,3,2,1, 4,3,3,2,4, 3,2,3,2,3, 2,1,3,2,2, 5,1,0,1,6, 5,2,3,1,7, 4,5,4,3,5, 3,5,4,0,8, 0,0,0,0,0
#$_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

The adventures of Doctor R.

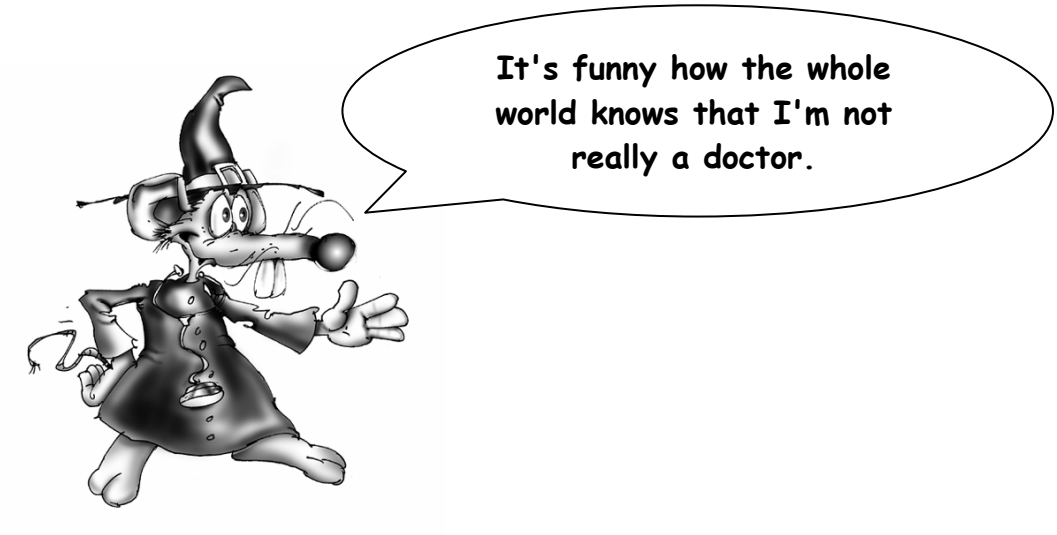
Educational training manual for AUTOMGEN users



Designed by TABAN

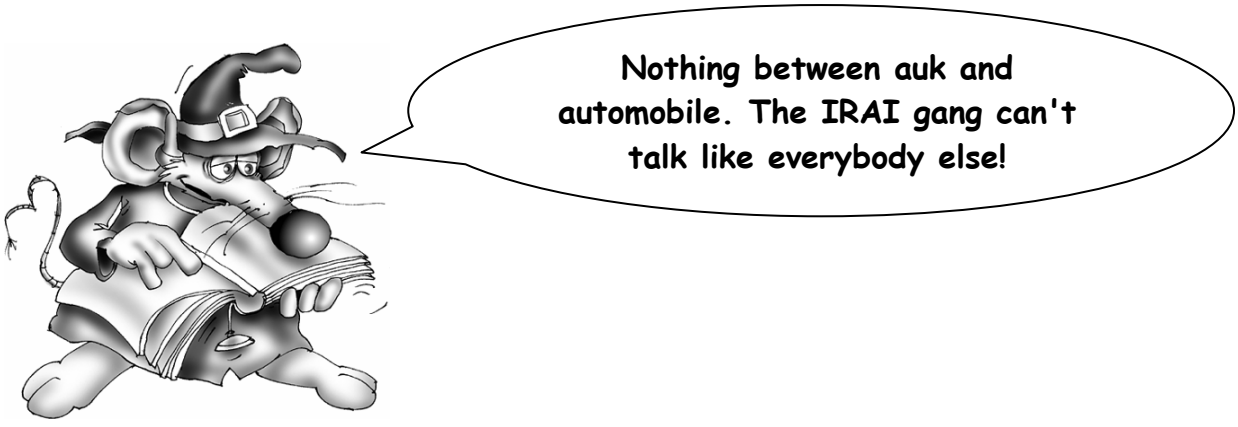
Distribution

Doctor R. …… Mister R.



Doctor R. in the home automation kingdom

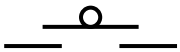
We are going to look at different example which can be directly applied in a home automation project. From an initial easy manner, these examples can be used to learn the different aspects that are the basis of automatisms and AUTOMGEN and IRIS training



This symbol will be used in the following for partial command (for example a programmable robot.

Is it necessary to explain that this

represents an bulb,

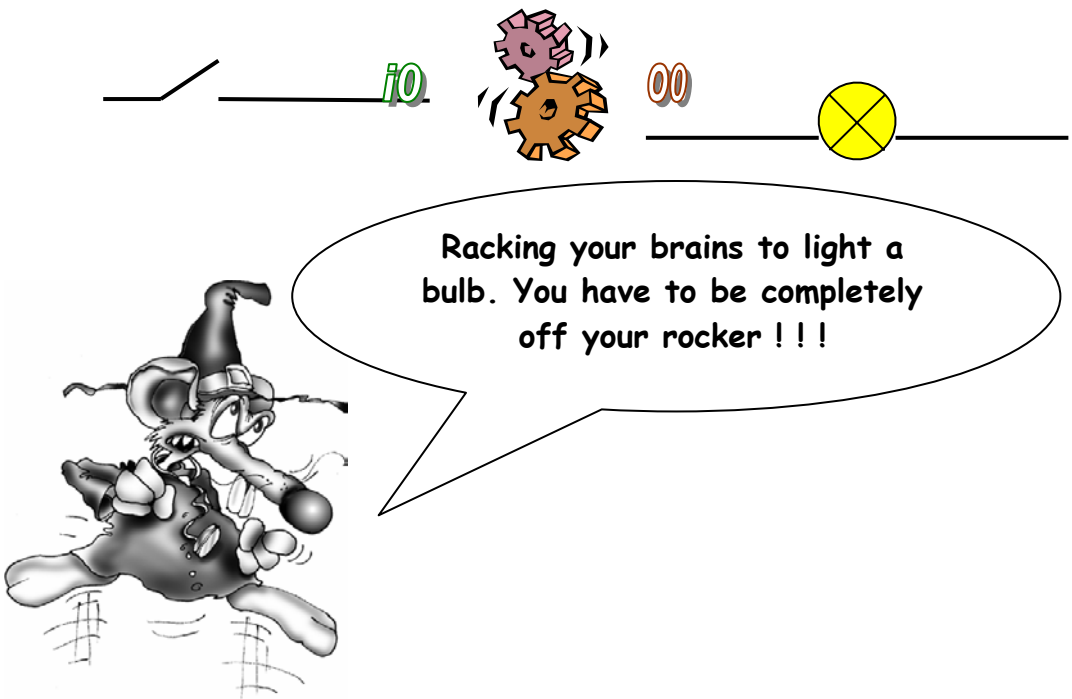


a push button

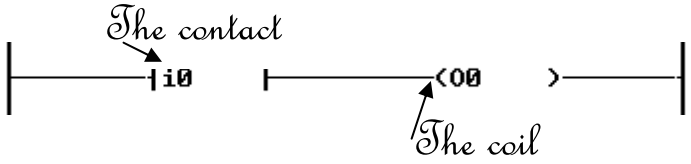
and a switch ?

First example : « which came first the bulb or the switch … »

A simple switch and a simple bulb: the switch is wired on input i0, the bulb on output o0. If the switch is closed then the bulb lights up, if the switch is open the bulb goes off. Hard to make it any simpler..

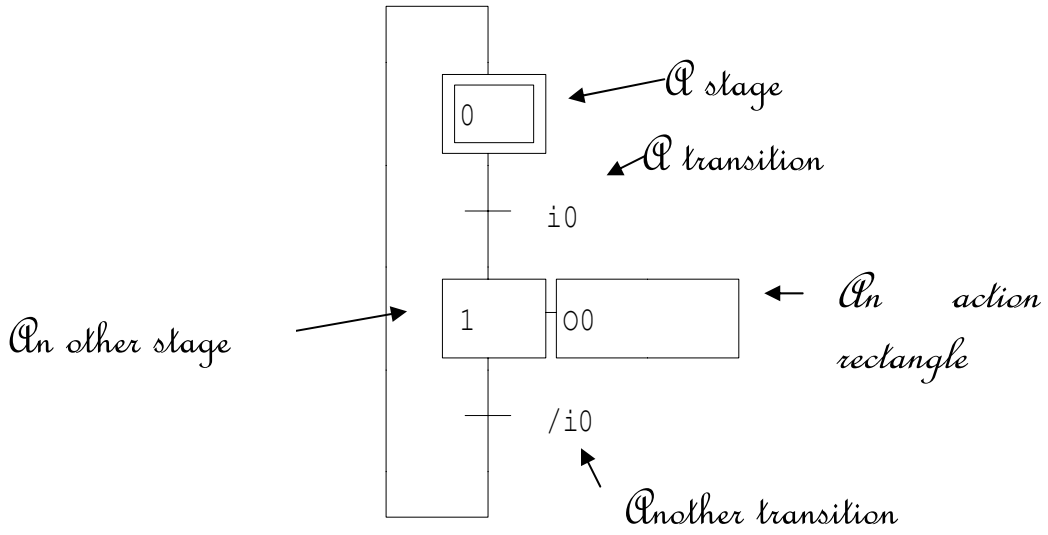


Solution 1 : natural language of an electrician: ladder

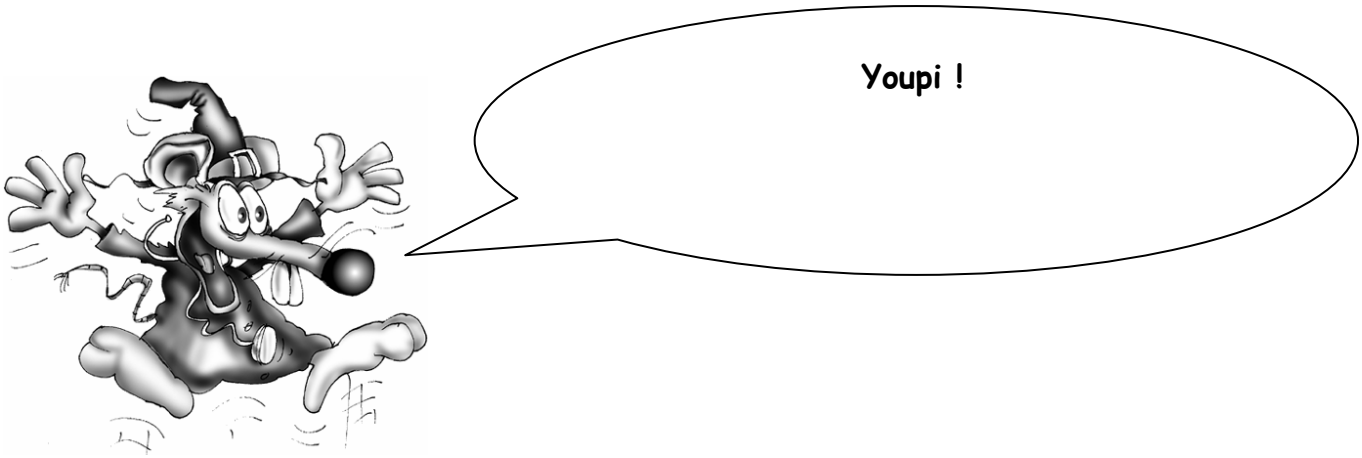


The ladder is the most direct transcription of a wiring diagram. The contact receives the name of the input where the switch is wired, the coil the name of the output where the bulb is wired.

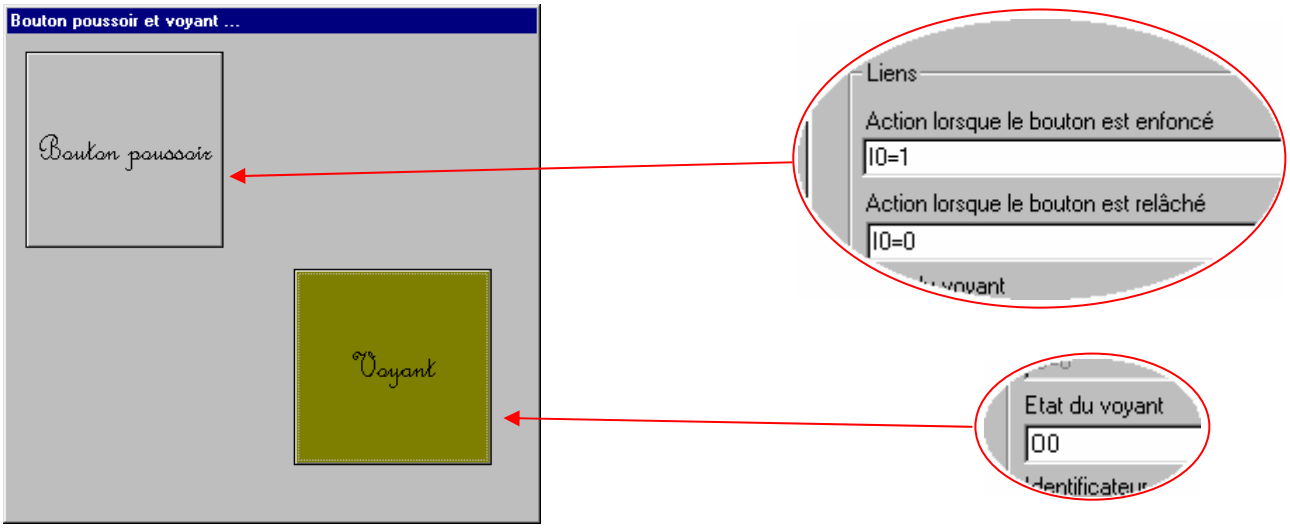
Solution 2 : the sequential language of the automation specialist: Grafcet

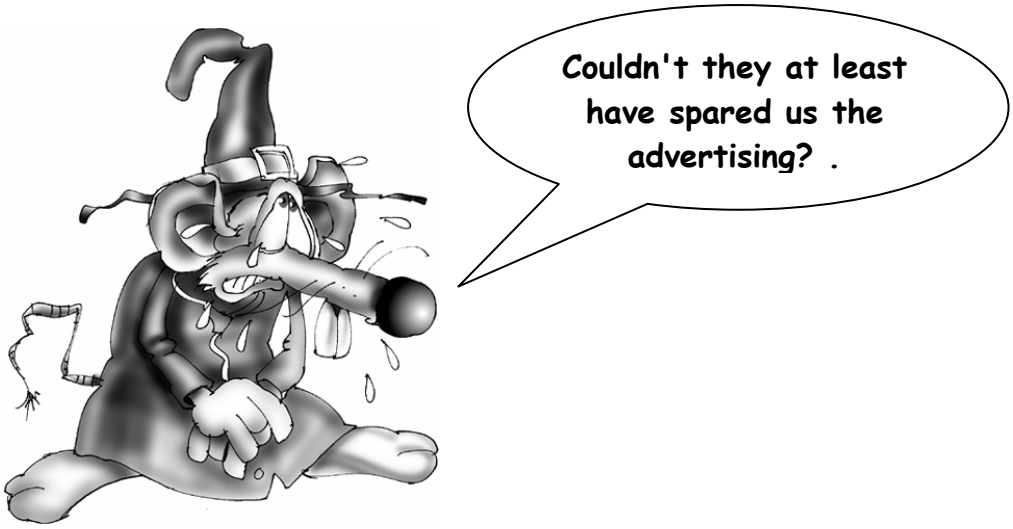


Grafcet is based on the idea of state. For our problem we could say that there are two states: the on state and the off state. Each step represents a state: here step 0 represents the off state and step 1 the on state. The condition which makes the off state go to the on state still needs to be determined: here the off switch (see i0) then the condition the causes the change from the on state to the off state : here the open switch (see / i0). The conditions are written to the right of the element marked transition. The rectangle associated to step 1 called action rectangle contains the name of output O0 (the output where our bulb is wired). So, the state of the bulb is always the same as that for step 1.

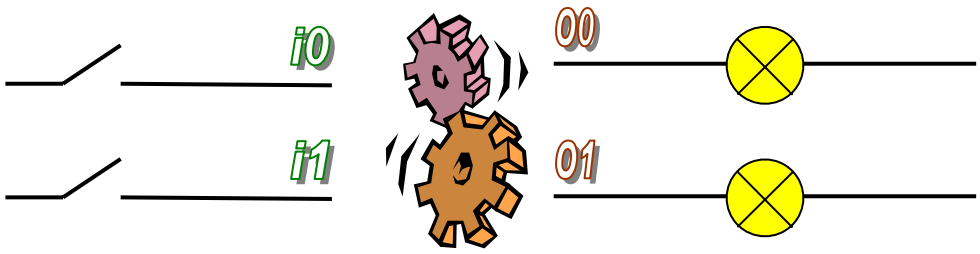


The happy owners of IRIS can use two BPVOYANT objects to simulate this first example.

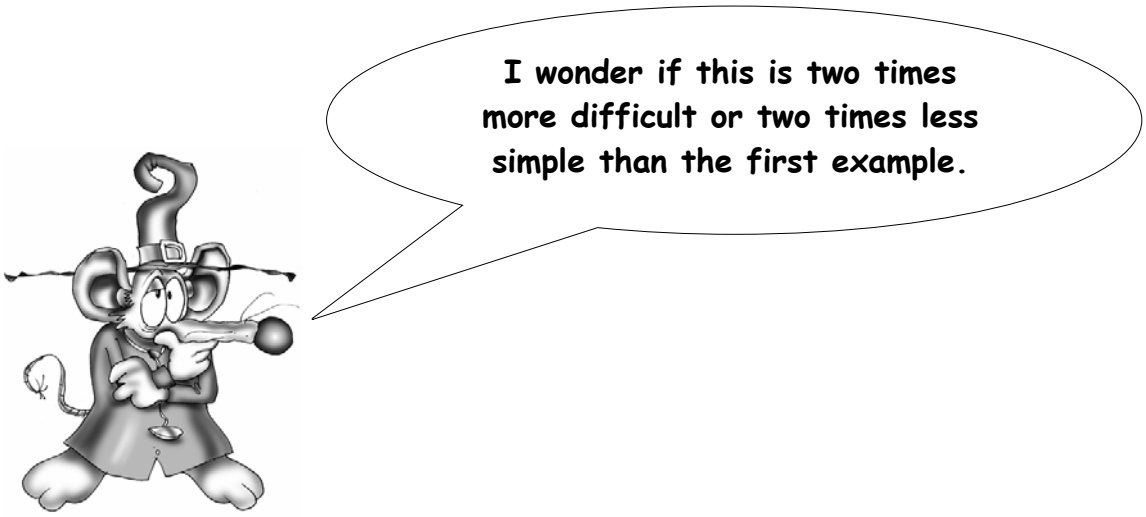




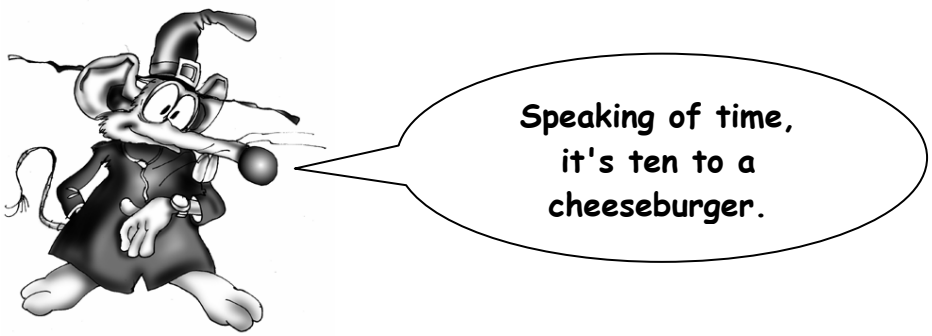
It's your turn to play …



Switch 1 lights bulb 1, switch 2 bulb 2. A Grafcet solution is proposed at the end of this document.

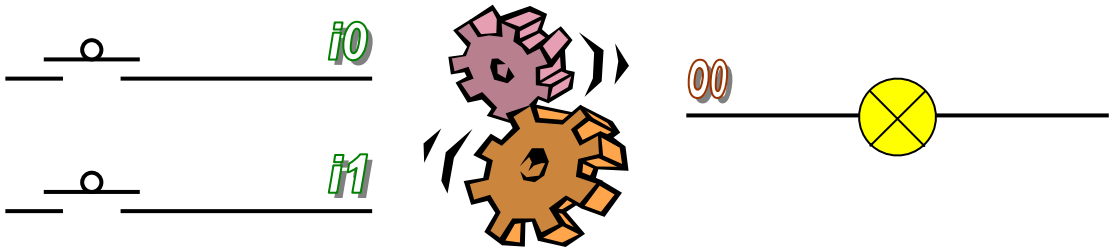


Second example : « time frames, time-switches and other time fun… »

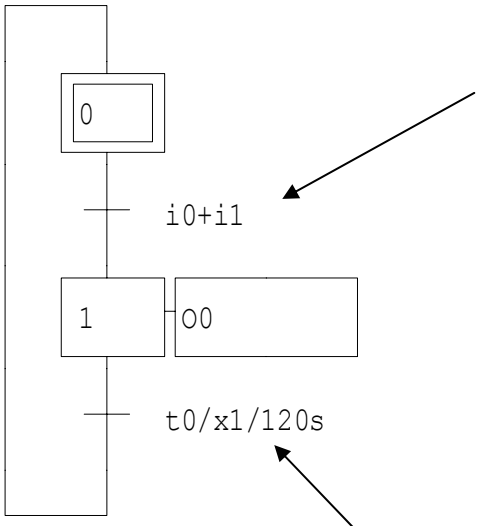


As you certainly deduce, the idea of a time frame is used when a program, in one way or another, must carry out actions taking into account data relative to time. For example, waiting for a certain amount of time before carrying out an action or carrying out an action within a certain amount of time.

Our second example is as follows: a hallway is equipped with a bulb and two push button. Pressing one of the push buttons lights the bulb for two minutes (after this gives Dr. R. plenty of time to cross the hallway).



Solution 1 : simplicity

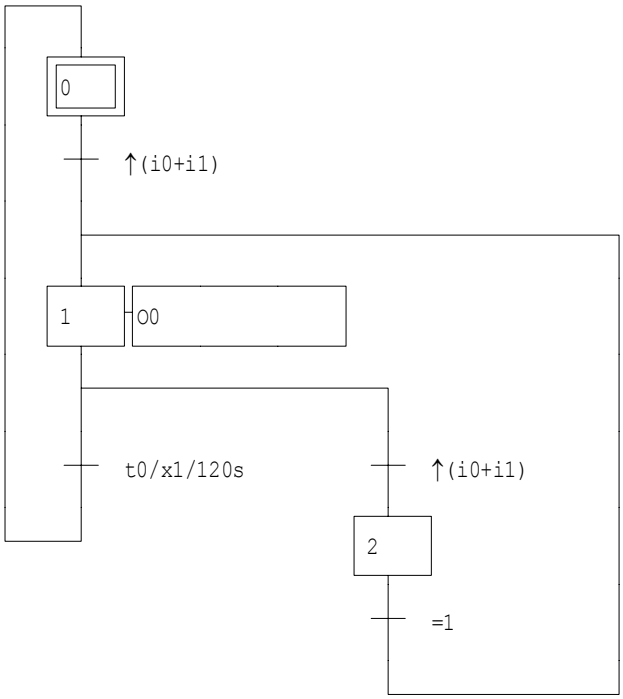


*Light the bulb if push button 1 is pressed **or** if push button 2 is pressed "or" is written "+ " in a transition pressé. « Ou » s'écrit « + » dans une transition.*

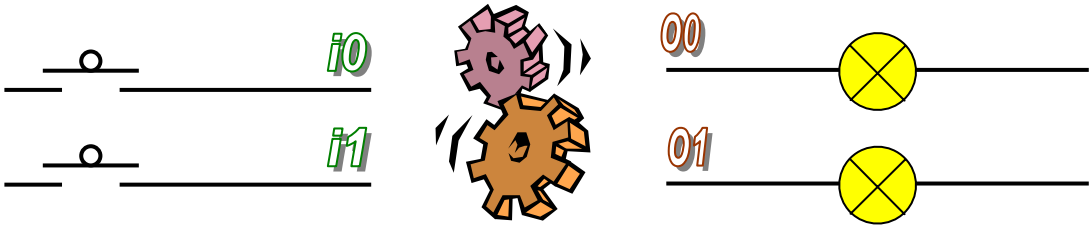
Wait 2 minutes (120 seconds) using time delay 0, step 1 launches the transition lance la temporisation.

Solution 2 : improvement

A problem is posed by this solution, if the push button is pressed while the bulb is lit, the time delay is not reset. Earlier Dr. R thought he had reset the time switch and found himself in the dark last week.



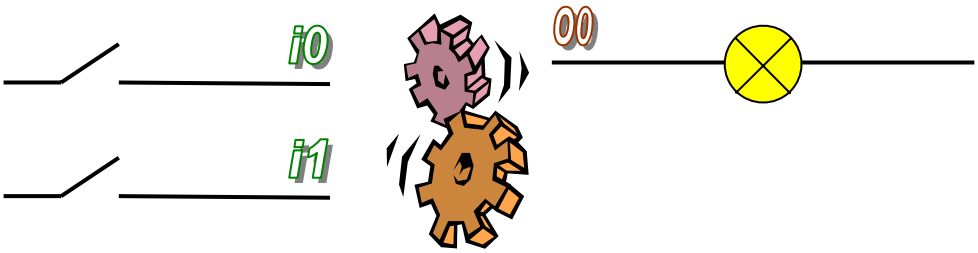
And if you want to try your hand at writing a real program ...



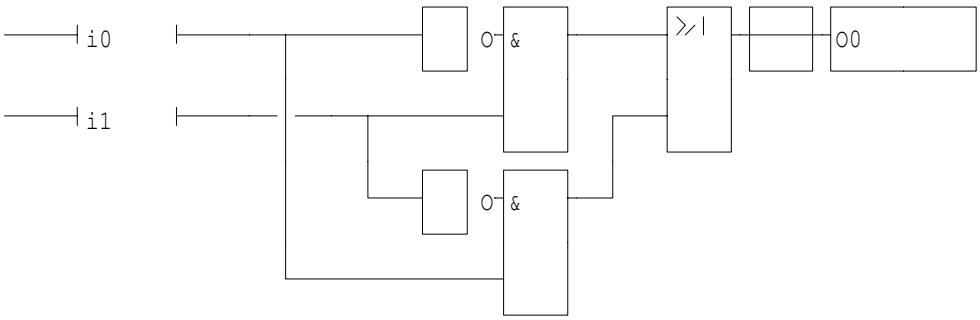
Intelligent management of lighting a hallway: a bulb is placed at each end. When a switch is pressed: the two bulbs light, then the bulb that is at the end near the pressed switch goes off at the end of 30 seconds and then the other at the end of one minute.

Third Example : « variation on the theme of coming and going… »

Remember the ingenious principle of coming and going : two switches make it possible to turn on or turn off the same bulb.

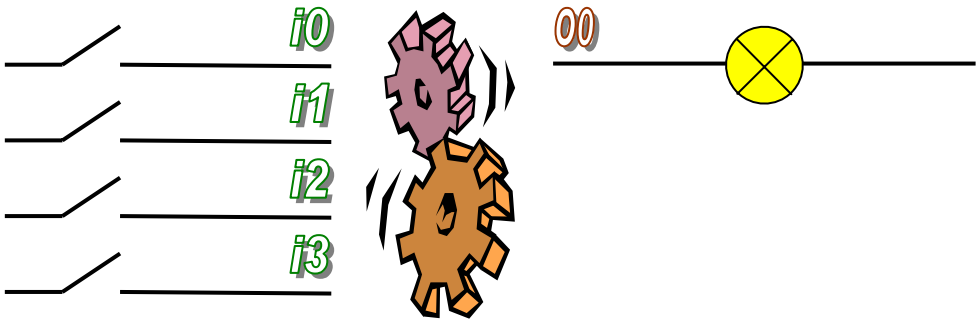


Here is a flow chart solution :



Purists will have recognized the exclusive or boolean equation.

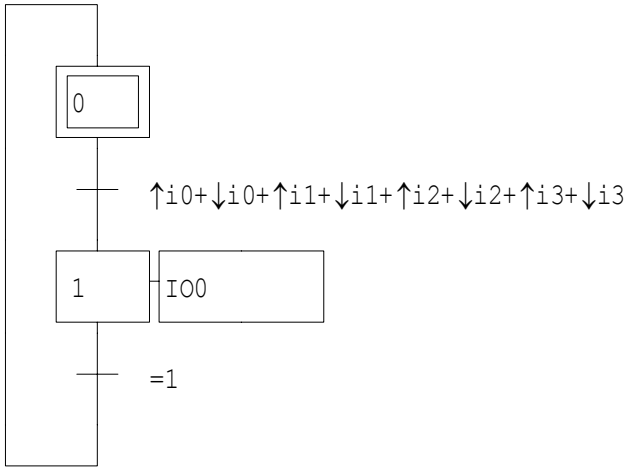
Things become really interesting if you want to keep the coming and going properties with a number of switches greater than 2.



A solution using AUTOMGEN literal language.

```
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)
```

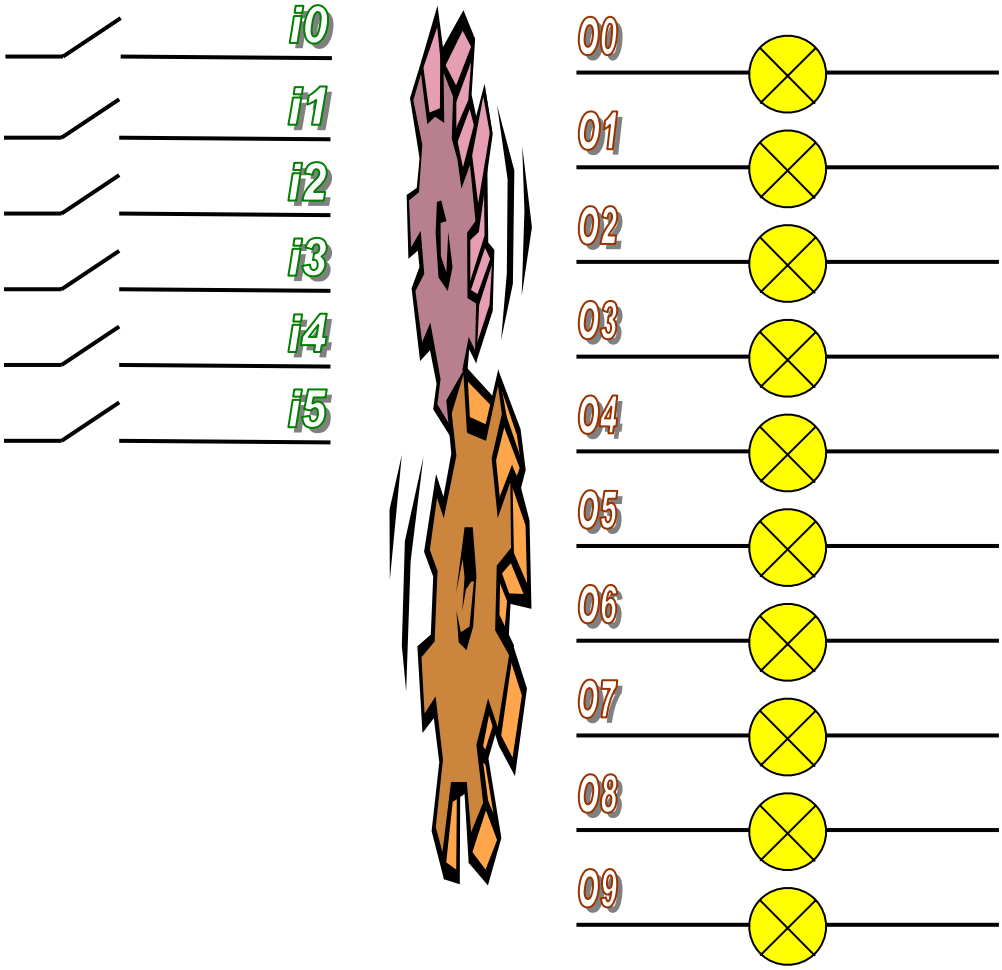
A cleverer one :



« IO0 » means « invert the state of output 0 ».

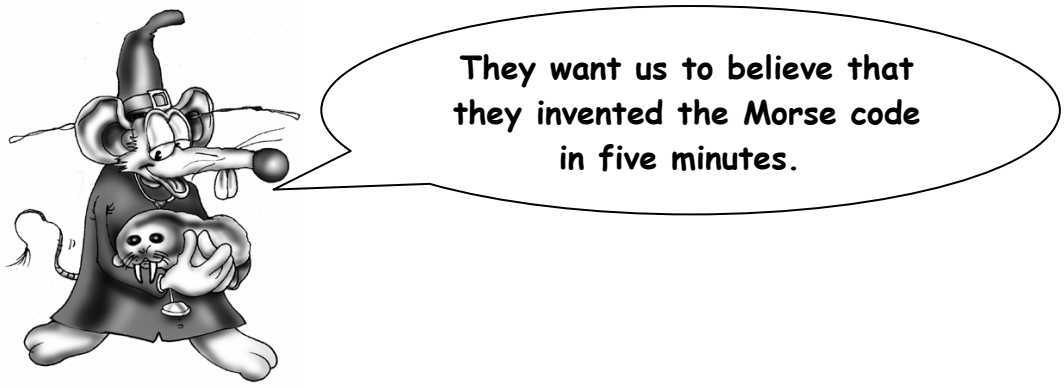
Try this :

A big room with 6 switches and 10 bulbs. Each switch can be used to more or less light the room (clearly passing from a state where everything is off to a state where one bulb is on, then two etc …).



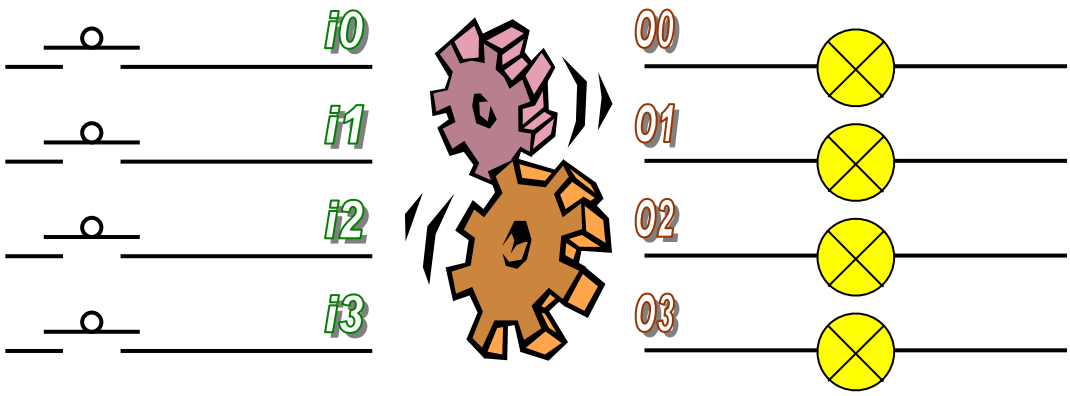
Fourth example : « And the push button became intelligent … »

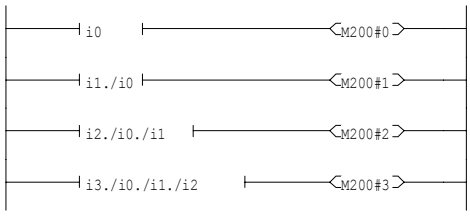
In all the previous examples, the push buttons carried out only one function. And so the person who used them only had two choices: to press them or not to press them in order to obtain a function (turning on or off). Image a « better performing » push button able to receive two types of pressure: a short pressure (arbitrarily less than one second) or a long pressure (arbitrarily at least a second).

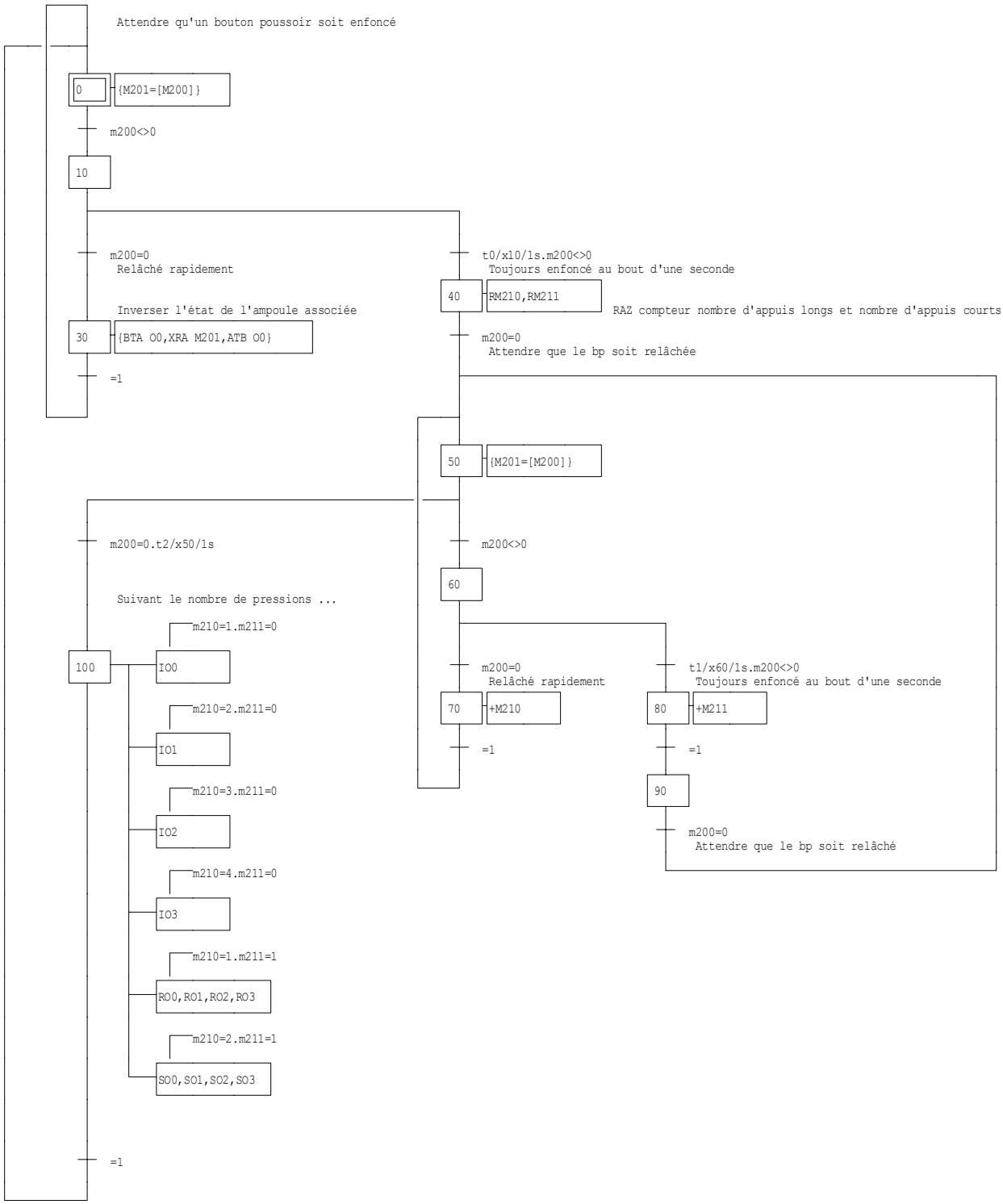


For this example four push buttons and four bulbs. By default, in normal use each push button is associated to a bulb. A short press on a push button turns on or off the associated bulb. Each push button must allow piloting one bulb or all of the bulbs. The table below describes this functionality.

Type of action on push button	**Result**
One short press	The associated bulb changes state
One long press and one short one	Bulb 1 changes state
One long press and two short ones	Bulb 2 changes state
One long press and three short ones	Bulb 3 changes state
One long press and four short ones	Bulb 4 changes state
Two long presses and a short one	All the bulbs go out
Tow long presses and two short ones	All the bulbs go on




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
m200<>0
60
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
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

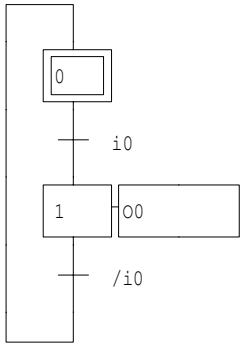
This ends our educational training manual. We hope that it has allowed you to discover the possibilities of AUTOMGEN.

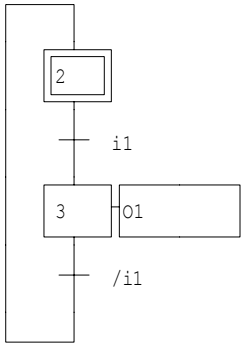
We have one last exercise for you.

Automate your aunt Hortense's apartment bearing in mind her excessive fondness for nickel-plated switches.

The solutions …

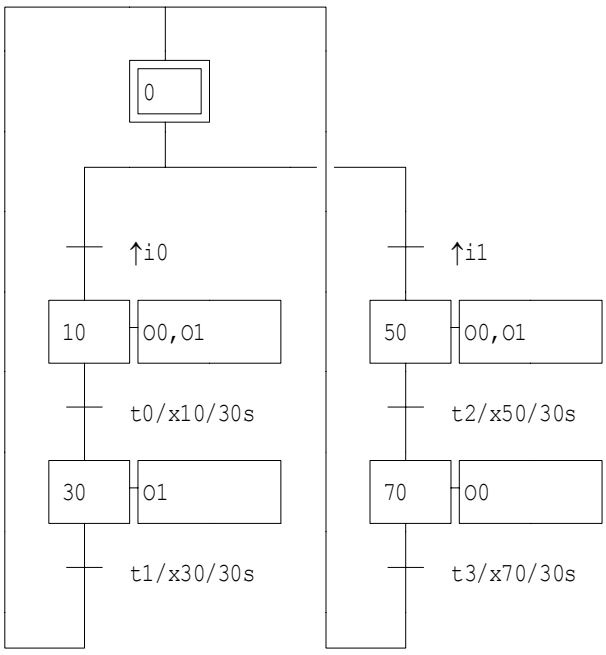
« which came first the switch or the bulb … »



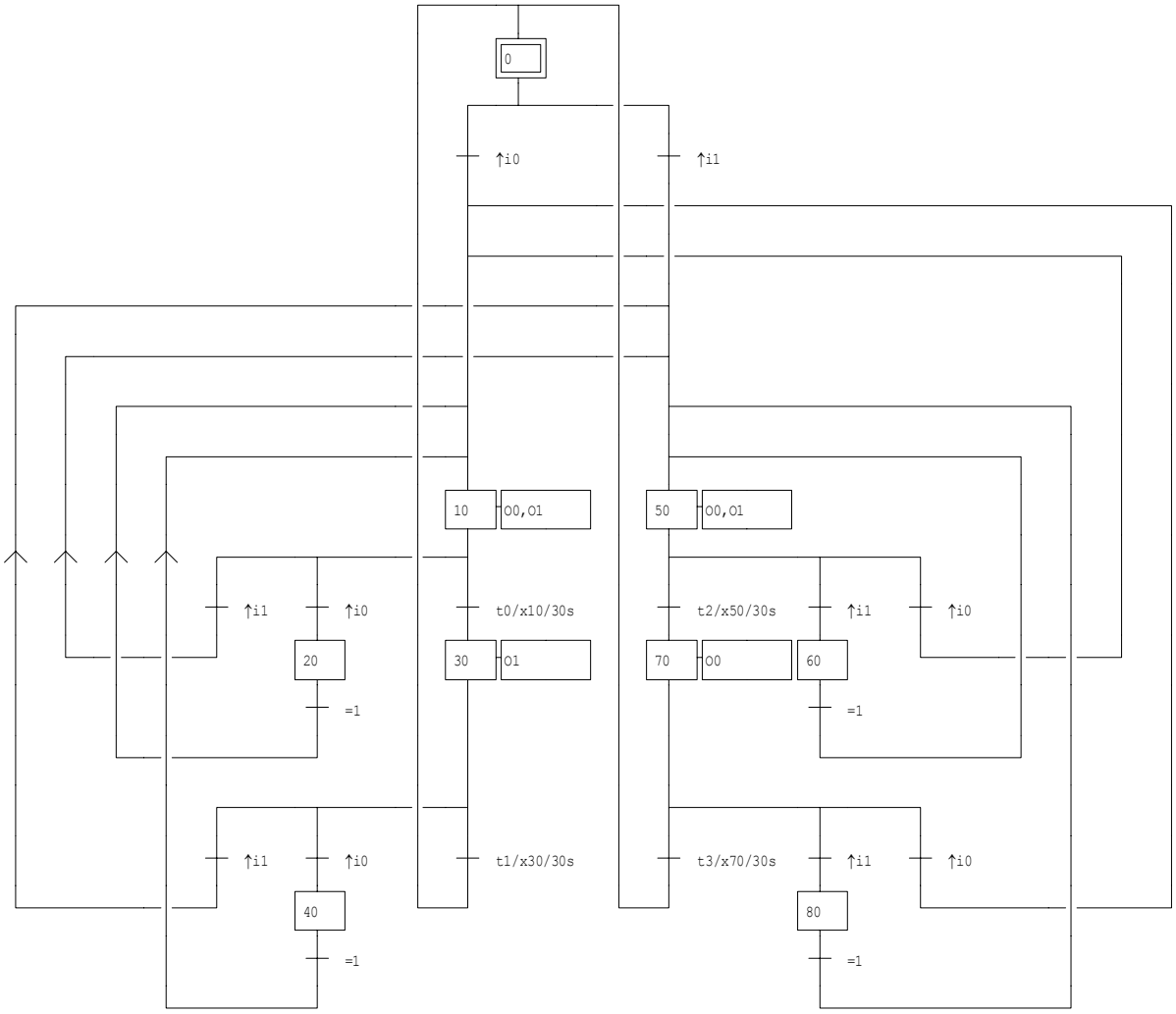


Just write two identical Grafcets. Each one is separately occupied by a switch and a bulb.

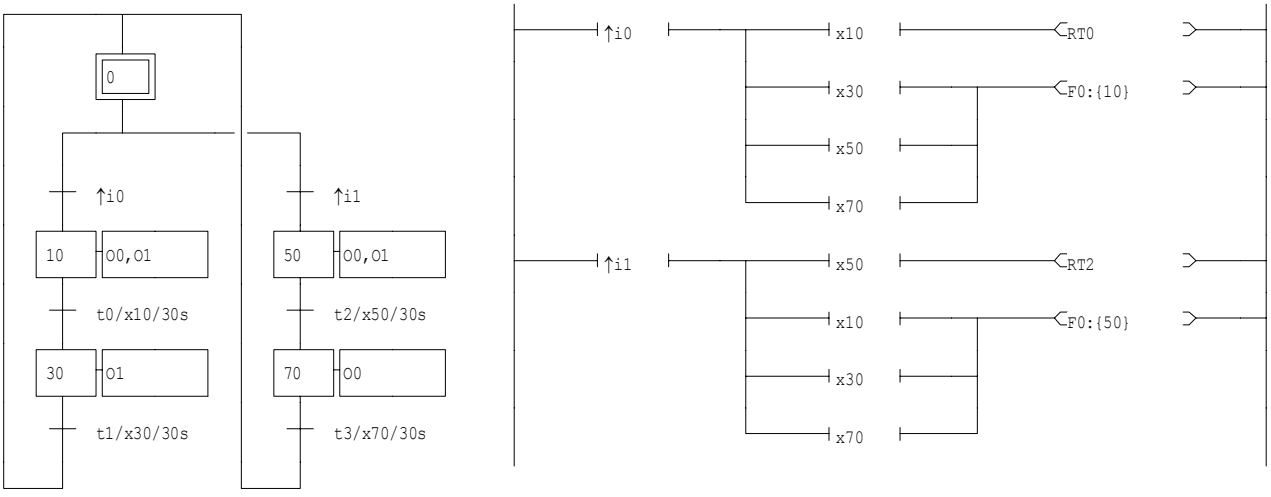
« time delays, time switches and other time fun… »



A simple version with no time switch reset management.



Resetting the time switch makes the program very complex.



The third solution uses Grafcet, ladder language and Grafcet settings. The program remains readable.

« variation on the theme of coming and going …»