Programming languages for PLC: International Standard IEC61131-3 (part two)

Cesare Fantuzzi (cesare.fantuzzi@unimore.it)

Ingegneria Meccatronica
AA 2012/2013
Instruction lists is the machine language of PLC programming. It has 21 instructions (see table).

Three modifiers are defined: "N" negates the result, "C" makes it conditional and "(" delays it.

All operations relate to one result register (RR) or accumulator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>N</td>
<td>Loads operand in RR</td>
</tr>
<tr>
<td>ST</td>
<td>N</td>
<td>Stores current result from RR</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>Sets the operand</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>Resets the operand</td>
</tr>
<tr>
<td>AND</td>
<td>N, (</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>OR</td>
<td>N, (</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>XOR</td>
<td>N, (</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>ADD</td>
<td>(</td>
<td>Arithmetic addition</td>
</tr>
<tr>
<td>SUB</td>
<td>(</td>
<td>Arithmetic subtraction</td>
</tr>
<tr>
<td>MUL</td>
<td>(</td>
<td>Arithmetic multiplication</td>
</tr>
<tr>
<td>DIV</td>
<td>(</td>
<td>Arithmetic division</td>
</tr>
<tr>
<td>GT</td>
<td>(</td>
<td>Comparison greater than</td>
</tr>
<tr>
<td>GE</td>
<td>(</td>
<td>Comparison greater than or equal to</td>
</tr>
<tr>
<td>EQ</td>
<td>(</td>
<td>Comparison equal</td>
</tr>
<tr>
<td>LE</td>
<td>(</td>
<td>Comparison less than</td>
</tr>
<tr>
<td>LT</td>
<td>(</td>
<td>Comparison less than or equal to</td>
</tr>
<tr>
<td>NE</td>
<td>(</td>
<td>Comparison not equal</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td>Executes delayed operation</td>
</tr>
<tr>
<td>CAL</td>
<td>C, N</td>
<td>Calls a function block</td>
</tr>
<tr>
<td>JMP</td>
<td>C, N</td>
<td>Jumps to label</td>
</tr>
<tr>
<td>RET</td>
<td>C, N</td>
<td>Returns from called function</td>
</tr>
</tbody>
</table>
Instruction Lists is the most efficient way to write code, but only for specialists.

Otherwise, IL should not be used, because this language:
- provides no code structuring
- has weak semantics
- is machine dependent

### Instruction Lists Example (2)

<table>
<thead>
<tr>
<th>Label</th>
<th>Operator</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>temp1</td>
<td></td>
<td>(<em>Load temp1 and</em>)</td>
</tr>
<tr>
<td>GT</td>
<td>temp2</td>
<td></td>
<td>(<em>Test if temp1 &gt; temp2</em>)</td>
</tr>
<tr>
<td>JMPCN</td>
<td>Greater</td>
<td></td>
<td>(<em>Jump if not true to Greater</em>)</td>
</tr>
<tr>
<td>LD</td>
<td>speed1</td>
<td></td>
<td>(<em>Load speed1</em>)</td>
</tr>
<tr>
<td>ADD</td>
<td>200</td>
<td></td>
<td>(<em>Add constant 200</em>)</td>
</tr>
<tr>
<td>JMP</td>
<td>End</td>
<td></td>
<td>(<em>Jump unconditional to End</em>)</td>
</tr>
<tr>
<td>Greater:</td>
<td>LD</td>
<td>speed2</td>
<td>(<em>Load speed2</em>)</td>
</tr>
</tbody>
</table>

End: ST temp3 (* result *)
All the programming languages share a common method for the variable definition.

Each variable used in IEC routine should be defined in the *declaration area*:

```
VAR
  AVERAGE_SPEED : REAL;
  Enable : BOOL;
END_VAR
```

The general declaration syntax:

```
VAR
  <VAR_NAME> : <TYPE> ;
  <VAR_NAME> : <TYPE> := <INITIAL_VALUE> ;
END_VAR
```
<table>
<thead>
<tr>
<th>No.</th>
<th>Keyword</th>
<th>Data Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOL</td>
<td>Boolean</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>SINT</td>
<td>Short integer</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>INT</td>
<td>Integer</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>DINT</td>
<td>Double integer</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>LINT</td>
<td>Long integer</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>USINT</td>
<td>Unsigned short integer</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>UINT</td>
<td>Unsigned integer</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>UDINT</td>
<td>Unsigned double integer</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>UINTEGER</td>
<td>Unsigned long integer</td>
<td>64</td>
</tr>
<tr>
<td>10</td>
<td>REAL</td>
<td>Real numbers</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>LREAL</td>
<td>Long reals</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>TIME</td>
<td>Duration</td>
<td>depends</td>
</tr>
<tr>
<td>13</td>
<td>DATE</td>
<td>Date (only)</td>
<td>depends</td>
</tr>
<tr>
<td>14</td>
<td>TIME_OF_DAY</td>
<td>Time of day (only)</td>
<td>depends</td>
</tr>
<tr>
<td>15</td>
<td>DATE_AND_TIME</td>
<td>Date and time of day</td>
<td>depends</td>
</tr>
<tr>
<td>16</td>
<td>STRING</td>
<td>Character string</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>BYTE</td>
<td>Bit string of length 8</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td>WORD</td>
<td>Bit string of length 16</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>DWORD</td>
<td>Bit string of length 32</td>
<td>32</td>
</tr>
<tr>
<td>20</td>
<td>LWORD</td>
<td>Bit string of length 64</td>
<td>64</td>
</tr>
</tbody>
</table>
To define an array:

```
VAR
  <ARRAY_NAME> : ARRAY [<INDEX_MIN>..<INDEX_MAX>]
  OF <TYPE> ;
END_VAR
```

…or a matrix

```
VAR
  <MATRIX_NAME> : ARRAY [<ROW_MIN>..<ROW_MAX>]
  OF ARRAY[<COL_MIN>..<COL_MAX>]
  OF <TYPE> ;
END_VAR
```

Example

```
POSITIONS : ARRAY [0..10] OF REAL ;
```
A structure is a collection of heterogeneous variables:

```
VAR
  STRUCT
    <VAR_1> : <VAR_1_TYPE>;
    ......  
    <VAR_N> : <VAR_N_TYPE>;
  END_STRUCT;
END_VAR;
```
Derived Data Type

- The IEC allows the definition of custom data type.
  - This improves the readability of the software.
  - Improve the checking possibility of the compiler on the user code.
- Syntax:
  ```
  TYPE
  <TYPE_NAME> : <TYPE> ;
  END_TYPE
  ```
- Example
  ```
  TYPE
  PUMP_PRESSURE : REAL ;
  END_TYPE
  
  VAR
  Pump_cooler : PUMP_PRESSURE;
  END_VAR;
  ```
A User Defined data types can be...

- A subrange of a elementary data type:
  ```
  TYPE
    ROBOT_POSITION : REAL(0..100) ;
  END_TYPE
  ```

- A list of constant (the *enum* data type)
  ```
  TYPE
    MACHINE_STATUS : (STOP, RUN, FAULT, IDLE) ;
  END_TYPE
  ```

- An Array or Matrix.

- A structure.
Example of Derived Types

TYPE
   ANALOG_CHANNEL_CONFIGURATION
   STRUCT
      RANGE : ANALOG_SIGNAL_RANGE;
      MIN_SCALE : ANALOG_DATA;
      MAX_SCALE : ANALOG_DATA;
   END_STRUCT;
   ANALOG_16_INPUT_CONFIGURATION :
      STRUCT
         SIGNAL_TYPE : ANALOG_SIGNAL_TYPE;
         FILTER_CHARACTERISTIC : SINT (0.99)
         CHANNEL : ARRAY [1..16] OF
            ANALOG_CHANNEL_CONFIGURATION;
      END_STRUCT;
   END_TYPE
Connecting to Input / Output,  
Symbol table: Variables configuration

All program variables must be declared with name and type, initial value and volatility. A variable may be connected to an input or an output, giving it an I/O address. Several properties can be set: default value, fall-back value, store at power fail,… These variables may not be connected as input, resp. output to a function block.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Attributes</th>
<th>Initial Value</th>
<th>I/O Address</th>
<th>Access Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>HK_Float</td>
<td>real</td>
<td>retain</td>
<td>0.0</td>
<td>Controller_1.0.11.3.1</td>
<td>Controller</td>
</tr>
<tr>
<td>HK_Min</td>
<td>real</td>
<td>retain</td>
<td>-100.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HK_Max</td>
<td>real</td>
<td>retain</td>
<td>+100.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HK_OnOff</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.1.1</td>
<td></td>
</tr>
<tr>
<td>DZ_Input</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.1.2</td>
<td></td>
</tr>
<tr>
<td>HK_DInput1</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.1</td>
<td></td>
</tr>
<tr>
<td>HK_DInput2</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.2</td>
<td></td>
</tr>
<tr>
<td>HK_DInput3</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.3</td>
<td></td>
</tr>
<tr>
<td>HK_DInput4</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.4</td>
<td></td>
</tr>
<tr>
<td>HK_DInput5</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.5</td>
<td></td>
</tr>
<tr>
<td>HK_DInput6</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.6</td>
<td></td>
</tr>
<tr>
<td>HK_DInput7</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.7</td>
<td></td>
</tr>
<tr>
<td>HK_DInput8</td>
<td>bool</td>
<td>retain</td>
<td></td>
<td>Controller_1.0.11.2.8</td>
<td></td>
</tr>
<tr>
<td>Right2LeftCnt</td>
<td>int</td>
<td>retain</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HK_MemTime</td>
<td>time</td>
<td>retain</td>
<td>5s23msec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

predefined addresses
SFC (Sequential Flow Chart)

SFC describes sequences of operations and interactions between parallel processes. It is derived from the languages Grafcet and SDL (used for communication protocols), its mathematical foundation lies in Petri Nets.
event condition
("1" = always true)

transitions

example transition condition

states

token

rule: there is always a transition between two states, there is always a state between two transitions

example: Sc is true, S0, Sa, Sb are false
SFC Rules of execution

- The sequential program consists of states connected by transitions.
- A state is activated by the presence of a token (the corresponding variable becomes TRUE).
- The token leaves the state when the transition condition (event) on the state output is true.
- Only one transition takes place at a time.
- The execution period is a configuration parameter (task to which this program is attached).
SFC: Initial state

State which come into existence with a token are called *initial states*.

All initial states receive exactly one token, the other states receive none.

Initialization takes place explicitly at start-up.
**token switch**: the token crosses the first active transition (at random if both Ea and Eb are true)
Note: transitions are after the alternance

**token forking**: when the transition Ee is true, the token is replicated to all connected states
Note: transition is **before** the fork

**token join**: when all connected states have tokens and transition Eg is true, one single token is forwarded.
Note: transition is **after** the join
### SFC: P1, N and P0 actions

<table>
<thead>
<tr>
<th></th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>State1_P1: do at enter</td>
</tr>
<tr>
<td>N</td>
<td>State1_N: do while</td>
</tr>
<tr>
<td>P0</td>
<td>State1_P0: do at leaving</td>
</tr>
</tbody>
</table>

P1 (pulse raise) action is executed once when the state is entered
P0 (pulse fall) action is executed once when the state is left
N (non-stored) action is executed continuously while the token is in the state

P1 and P0 actions could be replaced by additional states.

The actions are described by a code block written e.g. in Structured Text.
Special action: the timer

Rather than define a P0 action "reset timer....", there is an implicit variable defined as State.t that express the time spent in that state.

\[ S.t > t#5s \]

C. Fantuzzi  3. I Linguaggi di Programmazione
SFC: graphic rules

The input and output flow of a state are always in the same vertical line (simplifies structure)

Alternative paths are drawn such that no path is placed in the vertical flow (otherwise would mean this is a preferential path)

Priority:
- The alternative path most to the left has the highest priority, priority decreases towards the right.
- Loop: exit has a higher priority than loopback.
Many PLC applications mix continuous and discrete control. A PLC may execute alternatively function blocks and flow charts. A communication between these program parts must be possible.

Principle:

The SFC taken as a whole can be considered a function block with binary inputs (transitions) and binary outputs (states).
A function block may be implemented in three different ways:

procedure xy(...);
begin ...
end xy;

extern (ST) function block
flow chart

Function blocks and flow chart chart communicate over binary signals.
SFC or Function Blocs?

A task can sometimes be written indifferently as function blocs or as SFC. The application may decide which representation is more appropriate:
In this example, SFC seems to be more appropriate:

C. Fantuzzi 3. I Linguaggi di Programmazione 23
A POU is a portion of code which is encapsulated in a routine, which has an internal memory:

- Function Block.
- Program

or hasn’t

- Function
Types of POUs

1) "Functions"
   - are part of the base library.
   - have **no memory**.
     Example are: adder, multiplier, selector,....

2) "Function Blocks" (FB)
   - are part of the base library
   - have a **memory** ("static" data).
   - may access global variables (side-effects !)
     Examples: counter, filter, integrator,.....

3) "Programs" (Compound blocks)
   - user-defined or application-specific blocks
   - may **have a memory**
   - may be configurable
     Examples: PID controller, Overcurrent protection, Motor sequence
The programmer chooses the blocks in a block library, similarly to the hardware engineer who chooses integrated circuits out of the catalogue.

This library indicates the pinning of each block, its semantics and the execution time.

The programmer may extend the library by defining function block macros out of library elements.

If some blocks are often used, they will be programmed in an external language (e.g. “C”, micro-code) following strict rules.
IEC 61131-3 library (extract)

binary elements

| AND | or |
| OR  | exclusive-or |
| XOR | flip-flop |
| SR  | positive edge |
| R_TRIG |  |

<table>
<thead>
<tr>
<th>adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT</td>
</tr>
<tr>
<td>LT</td>
</tr>
<tr>
<td>LE</td>
</tr>
<tr>
<td>selector (1:2)</td>
</tr>
<tr>
<td>(1:N)</td>
</tr>
</tbody>
</table>

analog elements

| MUX | multiplexer |
|------|
| bool |
| int |

C. Fantuzzi 3. I Linguaggi di Programmazione 27
Exercise: Tooth saw generator

exercise: build a tooth-saw (asymmetric) generator with the IEC 61131 elements of the preceding page

C. Fantuzzi 3. I Linguaggi di Programmazione 28
Function Block library for specialized applications

MoveAbsolute

<table>
<thead>
<tr>
<th>Axis Ref</th>
<th>Execute</th>
<th>Position</th>
<th>CommandAborted</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Done</td>
<td>REAL</td>
<td>BOOL</td>
<td>BOOL</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>WORD</td>
</tr>
<tr>
<td>REAL</td>
<td>Acceleration</td>
<td>Deceleration</td>
<td>ErrorID</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>Jerk</td>
<td>Direction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: FB for motion control

C. Fantuzzi 3. I Linguaggi di Programmazione
Specifying the behaviour of Function Block

Time Diagram:

Truth Table:

Mathematical Formula:

\[ y = K_p x + K_d \frac{dx}{dt} + K_i \int_0^t x(t) dt \]
FUNCTION_BLOCK HYSTERESIS
VAR_INPUT
  XIN1, XIN2 : REAL;
  EPS : REAL; (* Hysteresis:
END_VAR
VAR_OUTPUT
  Q : BOOL := 0
END_VAR
IF Q THEN
  IF XIN1 < (XIN2–EPS) THEN
    Q := 0 (* XIN1 decrease:
  END_IF;
ELSIF XIN1 > (XIN2 + EPS)
  Q := 1; (* XIN1 increase)
END_IF;
END_FUNCTION_BLOCK
Function Block decomposition

A function block describes a *data flow interface*. Its *body* can be implemented differently:

**Elementary block**

The body is implemented in an *external language* (micro-code, assembler, java, IEC 61131 ST):

```plaintext
procedure xy(a,b:BOOLEAN; VAR b,c: BOOLEAN);
begin
   ..... 
end xy;
```

**Compound block**

The body is realized as a *function block program*

Each input (output) pin of the interface is implemented as exactly one input (output) of the function block.

All signals must appear at the interface to guarantee freedom from *side effects.*
An application program is decomposed into segments ("Programs") for modularity. Within a segment, the connections are represented *graphically*. Between the segments, the connections are expressed by *signal names*.
Execution of Function Blocks

Segment or POU (program organization unit)

Machine Code:

The function blocks are translated to machine language (often an assembly language),
Blocks are executed in sequence, normally from upper left to lower right
The sequence is scheduled repeated every x ms.
The function blocks are executed cyclically.

- all inputs are read from memory (corresponding to a sensor on the plant).
- the segment is executed
- the results are written into memory (corresponding to a actuator on the plant)

The different segments may be assigned a different individual period.
Parallel execution

Function blocks are particularly well suited for true multiprocessing (parallel processors).

The performance limit is given by the needed exchange of signals by means of a shared memories.

Semaphores are not used since they could block an execution and make the concerned processes non-deterministic.
The software application (the complete control program) is divided into tasks:

- A task is a run-time execution of a software routine.
- An execution period might be assigned to each task to respect real time deadline, if not, the task run continuously (free-run).
- A priority is assigned to respect task priorities.
- A task might be triggered by an external event.
A PLC programming environment (ABB, Siemens, CoDeSys,...) allows:

- programming of the PLC in one of the IEC 61131 languages
- defining the variables (name and type)
- binding of the variables to the input/output (binary, analog)
- simulating
- downloading to the PLC of programs and firmware
- uploading of the PLC (seldom provided)
- monitoring of the PLC
- documenting and printing.
61131 Programming environment

configuration, editor, compiler, library

variable monitoring and forcing for debugging

network

PLC

symbols
code
firmware
download