

ANNEX F - Examples (informative) (1996 version)

F.1 Function WEIGH

Example function WEIGH provides the functions of BCD-to-binary conversion of a gross-weight input from a scale, the binary integer subtraction of a tare weight which has been previously converted and stored in the memory of the programmable controller, and the conversion of the resulting net weight back to BCD form, e.g., for an output display. The "EN" input is used to indicate that the scale is ready to perform the weighing operation.

The "ENO" output indicates that an appropriate command exists (e.g., from an operator pushbutton), the scale is in proper condition for the weight to be read, and each function has a correct result.

A textual form of the declaration of this function is:

```
FUNCTION WEIGH : WORD      (* BCD encoded *)
  VAR_INPUT  (* "EN" input is used to indicate "scale ready"
  *)
    weigh_command : BOOL;
    gross_weight  : WORD ; (* BCD encoded *)
    tare_weight   : INT  ;
  END_VAR
(* Function Body *)
END_FUNCTION                (* Implicit "ENO" *)
```

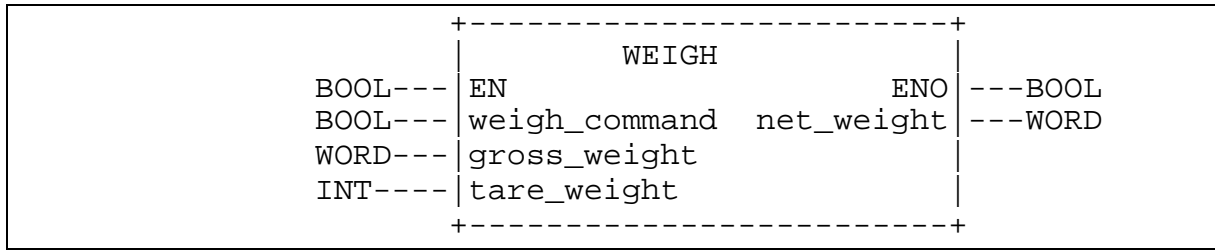
The body of function WEIGH in the IL language is:

```
LD      weigh_command
JMPC    WEIGH_NOW
ST      ENO      (* No weighing, 0 to "ENO" *)
RET
WEIGH_NOW: LD      gross_weight
          BCD_TO_INT
          SUB      tare_weight
          INT_TO_BCD      (* Return evaluated weight *)
```

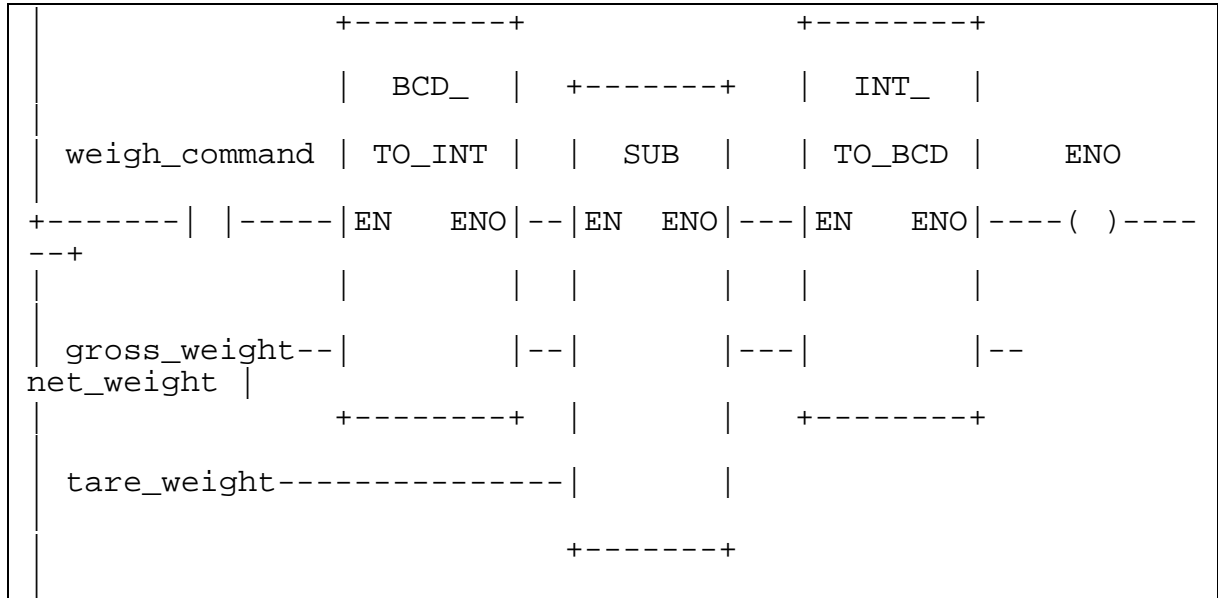
The body of function WEIGH in the ST language is:

```
IF weigh_command THEN
  WEIGH := INT_TO_BCD (BCD_TO_INT(gross_weight) -
tare_weight);
END_IF ;
```

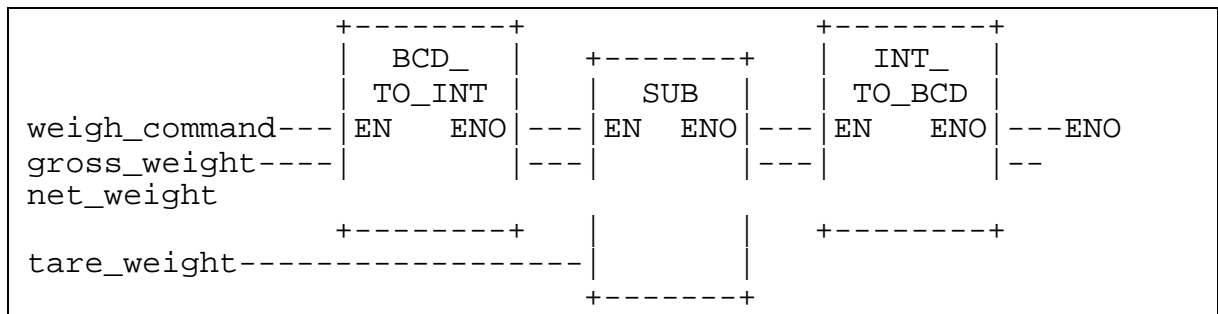
An equivalent graphical declaration of function WEIGH is:



The function body in the LD language is:



The function body in the FBD language is:



F.2 Function block CMD_MONITOR

Example function block CMD_MONITOR illustrates the control of an operative unit which is capable of responding to a Boolean command (the CMD output) and returning a Boolean feedback signal (the FDBK input) indicating successful completion of the commanded action. The function block provides for manual control via the MAN_CMD input, or automated control via the AUTO_CMD input, depending on the state of the AUTO_MODE input (0 or 1 respectively). Verification of the MAN_CMD input is provided via the MAN_CMD_CHK input, which must be 0 in order to enable the MAN_CMD input. If confirmation of command completion is not received on the FDBK input within a predetermined time specified by the T_CMD_MAX input, the command is cancelled and an alarm condition is signalled via the ALRM output. The alarm condition may be cancelled by the ACK (acknowledge) input, enabling further operation of the command cycle.

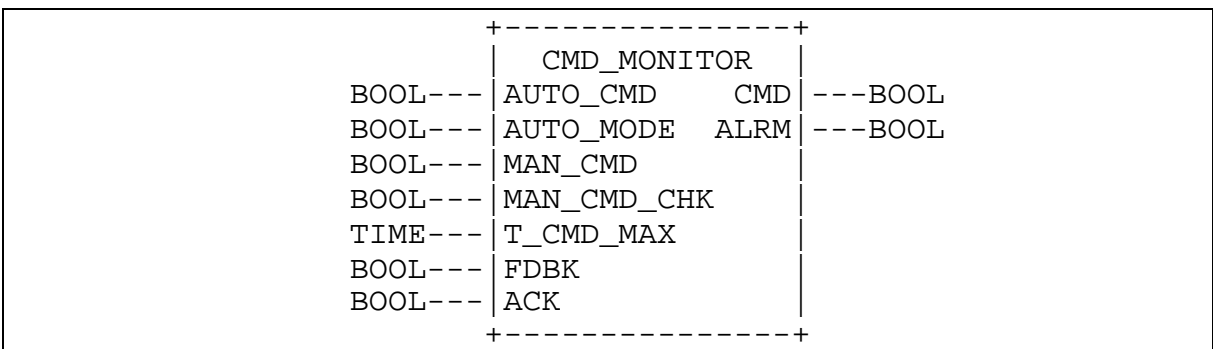
A textual form of the declaration of function block CMD_MONITOR is:

```

FUNCTION_BLOCK CMD_MONITOR
  VAR_INPUT AUTO_CMD : BOOL ; (* Automated command *)
            AUTO_MODE : BOOL ; (* AUTO_CMD enable *)
            MAN_CMD : BOOL ; (* Manual Command *)
            MAN_CMD_CHK : BOOL ; (* Negated MAN_CMD to debounce
*)
            T_CMD_MAX : TIME ; (* Max time from CMD to FDBK *)
            FDBK : BOOL ; (* Confirmation of CMD
completion
                                by operative unit *)
            ACK : BOOL ; (* Acknowledge/cancel ALRM *)
  END_VAR
  VAR_OUTPUT CMD : BOOL ; (* Command to operative unit *)
            ALRM : BOOL ; (* T_CMD_MAX expired without FDBK
*)
  END_VAR
  VAR CMD_TMR : TON ; (* CMD-to-FDBK timer *)
            ALRM_FF : SR ; (* Note over-riding "S" input: *)
  END_VAR (* Command must be cancelled before
            "ACK" can cancel alarm *)
  (* Function Block Body *)
END_FUNCTION_BLOCK

```

An equivalent graphical declaration is:



The body of function block CMD_MONITOR in the ST language is:

```

CMD := AUTO_CMD & AUTO_MODE
      OR MAN_CMD & NOT MAN_CMD_CHK & NOT AUTO_MODE ;
CMD_TMR (IN := CMD, PT := T_CMD_MAX);
ALRM_FF (S1 := CMD_TMR.Q & NOT FDBK, R := ACK);
ALRM := ALRM_FF.Q1;

```

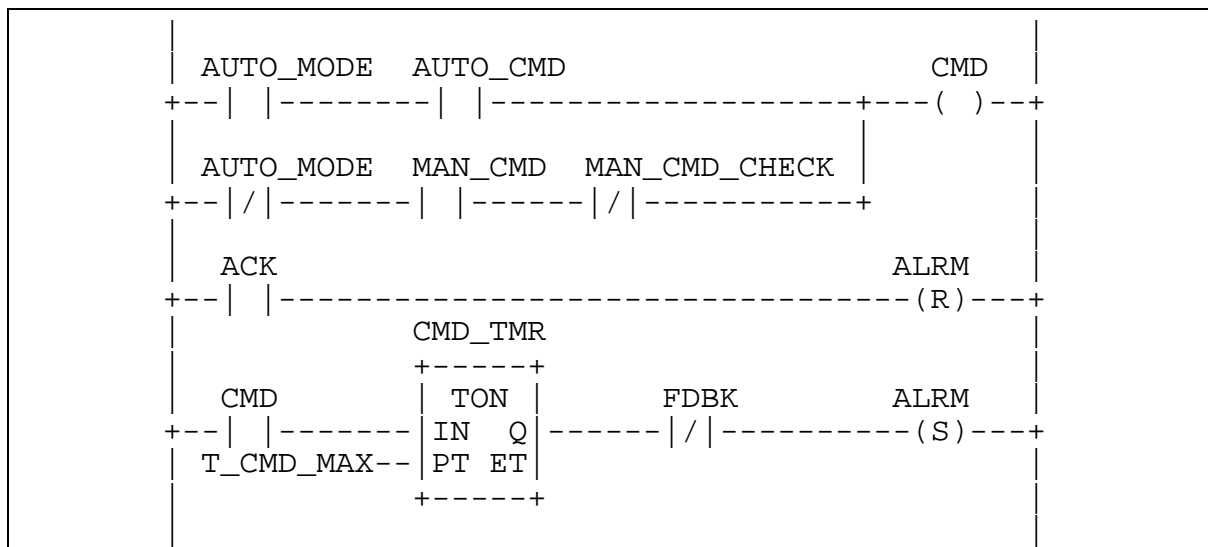
The body of function block CMD_MONITOR in the IL language is:

```

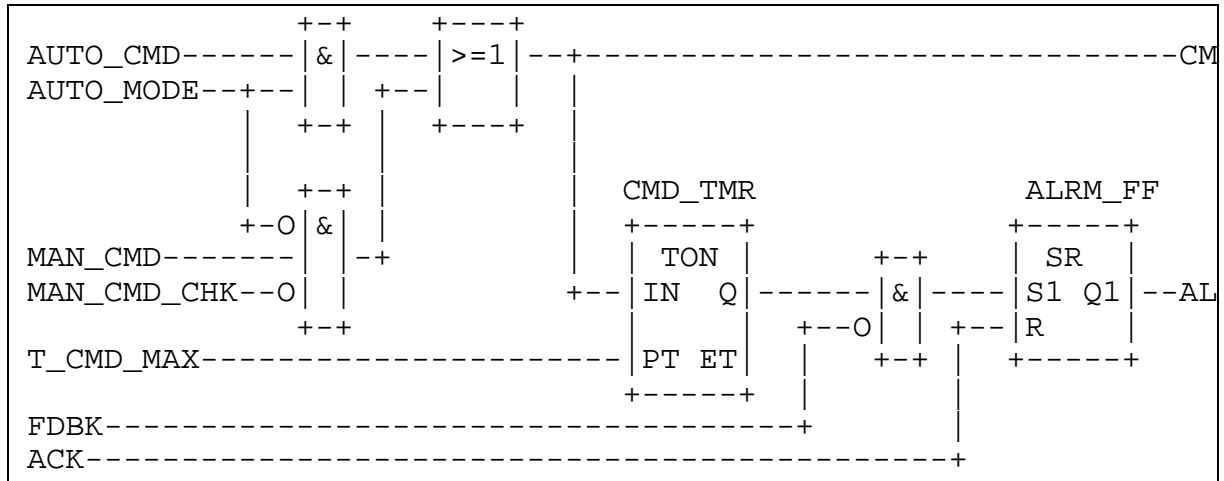
LD    T_CMD_MAX
ST    CMD_TMR.PT    (* Store an input to the TON FB *)
LD    AUTO_CMD
AND   AUTO_MODE
OR(   MAN_CMD
AND   AUTO_MODE
N
AND   MAN_CMD_CH
N    K
)
ST    CMD
IN    CMD_TMR    (* Invoke the TON FB *)
LD    CMD_TMR.Q
AND   FDBK
N
ST    ALRM_FF.S1    (* Store an input to the SR FB *)
LD    ACK
R     ALRM_FF    (* Invoke the SR FB *)
LD    ALRM_FF.Q1
ST    ALRM

```

The body of function block CMD_MONITOR in the LD language is:



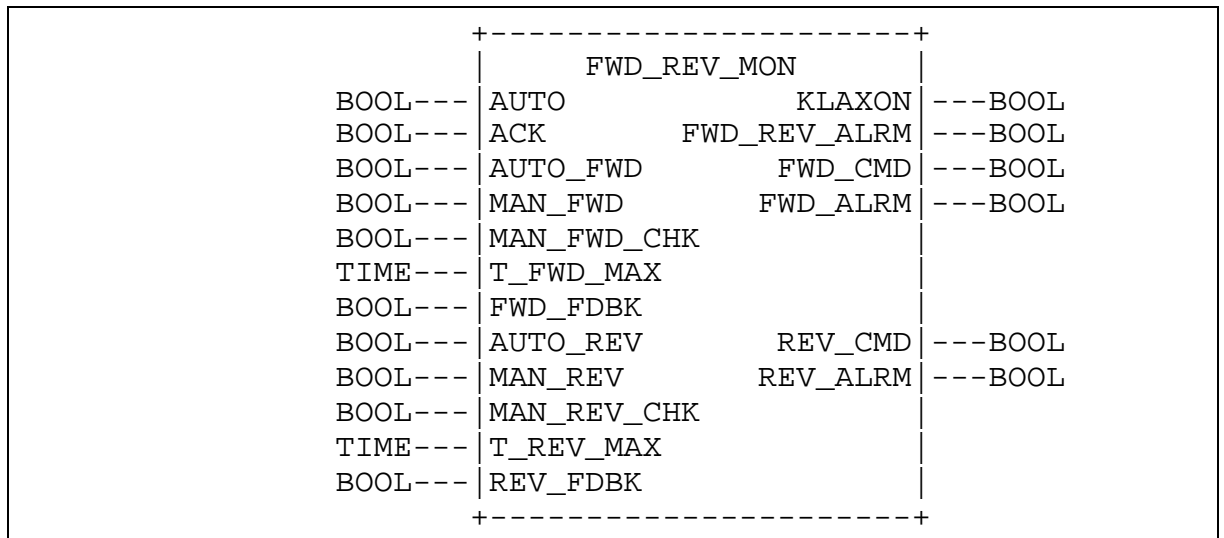
The body of function block CMD_MONITOR in the FBD language is:



F.3 Function block FWD_REV_MON

Example function block FWD_REV_MON illustrates the control of an operative unit capable of two-way positioning action, e.g., a motor-operated valve. Both automated and manual control modes are possible, with alarm capabilities provided for each direction of motion, as described for function block CMD_MONITOR above. In addition, contention between forward and reverse commands causes the cancellation of both commands and signalling of an alarm condition. The Boolean OR of all alarm conditions is made available as a KLAXON output for operator signaling.

A graphical declaration of this function block is:



A textual form of the declaration of function block FWD_REV_MON is:

```
FUNCTION_BLOCK FWD_REV_MON
VAR_INPUT AUTO : BOOL ;(* Enable automated commands *)
  ACK : BOOL ;          (* Acknowledge/cancel all alarms *)
  AUTO_FWD : BOOL ;     (* Automated forward command *)
  MAN_FWD : BOOL ;      (* Manual forward command *)
  MAN_FWD_CHK : BOOL ;  (* Negated MAN_FWD for debouncing *)
  T_FWD_MAX : TIME ;    (* Maximum time from FWD_CMD to FWD_FDBK
*)
  FWD_FDBK : BOOL ;     (* Confirmation of FWD_CMD completion *
(* by operative unit *)
  AUTO_REV : BOOL ;     (* Automated reverse command *)
  MAN_REV : BOOL ;      (* Manual reverse command *)
  MAN_REV_CHK : BOOL ;  (* Negated MAN_REV for debouncing *)
  T_REV_MAX : TIME ;    (* Maximum time from REV_CMD to REV_FDBK
*)
  REV_FDBK : BOOL ;     (* Confirmation of REV_CMD completion *
(* by operative unit *)
END_VAR
VAR_OUTPUT KLAXON : BOOL ;          (* Any alarm active *)
  FWD_REV_ALRM : BOOL; (* Forward/reverse command conflict *)
  FWD_CMD : BOOL ;      (* "Forward" command to operative unit
  FWD_ALRM : BOOL ;     (* T_FWD_MAX expired without FWD_FDBK *
  REV_CMD : BOOL ;      (* "Reverse" command to operative unit
  REV_ALRM : BOOL ;     (* T_REV_MAX expired without REV_FDBK *
END_VAR
VAR FWD_MON : CMD_MONITOR; (* "Forward" command monitor *)
  REV_MON : CMD_MONITOR;   (* "Reverse" command monitor *)
  FWD_REV_FF : SR ;        (* Forward/Reverse contention latch *)
END_VAR
(* Function Block body *)
END_FUNCTION_BLOCK
```

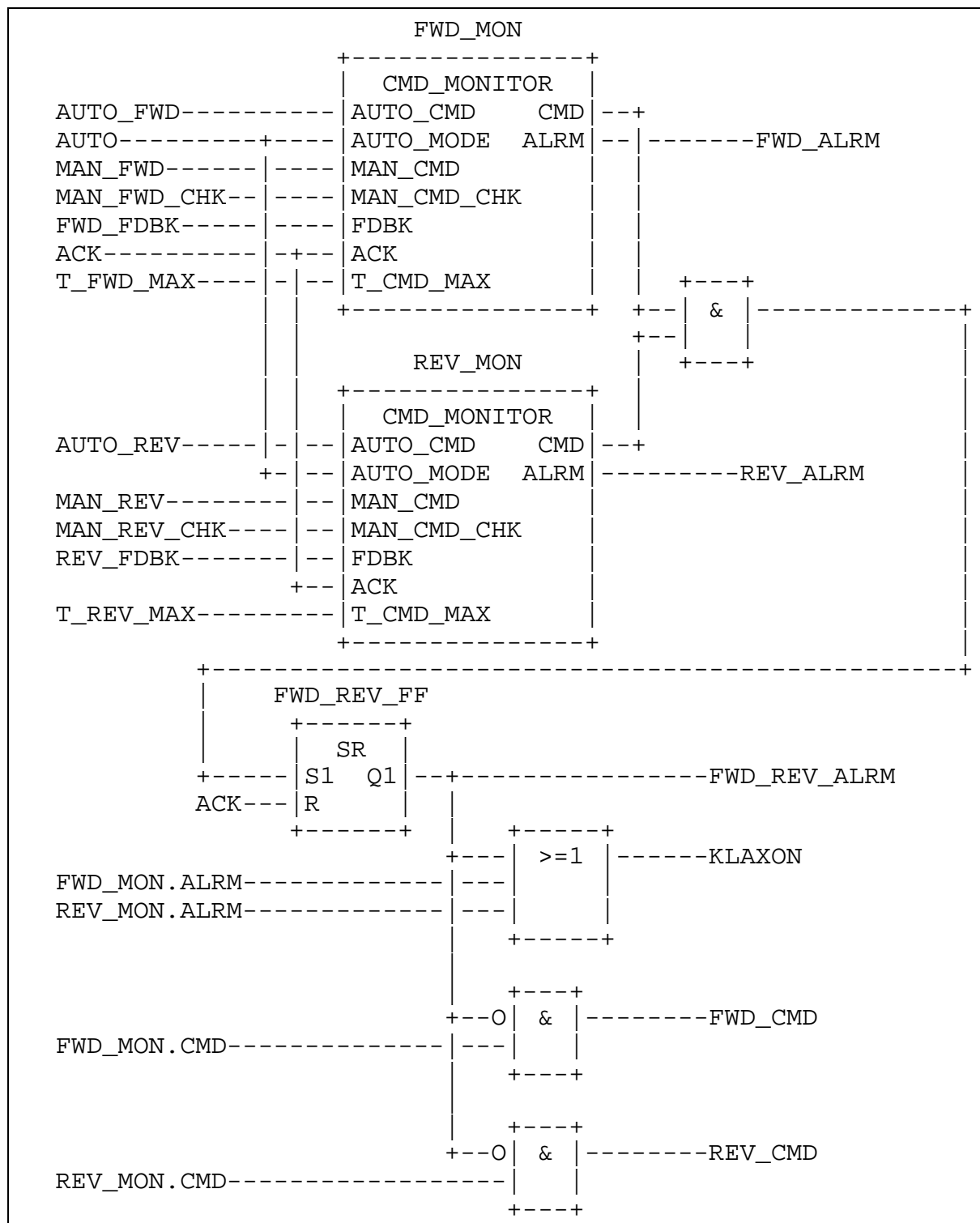
The body of function block FWD_REV_MON can be written in the ST language as:

```
(* Evaluate internal function blocks *)
FWD_MON    (AUTO_MODE    := AUTO,
            ACK           := ACK,
            AUTO_CMD     := AUTO_FWD,
            MAN_CMD      := MAN_FWD,
            MAN_CMD_CHK  := MAN_FWD_CHK,
            T_CMD_MAX    := T_FWD_MAX,
            FDBK         := FWD_FDBK);
REV_MON    (AUTO_MODE    := AUTO,
            ACK           := ACK,
            AUTO_CMD     := AUTO_REV,
            MAN_CMD      := MAN_REV,
            MAN_CMD_CHK  := MAN_REV_CHK,
            T_CMD_MAX    := T_REV_MAX,
            FDBK         := REV_FDBK);
FWD_REV_FF (S1 := FWD_MON.CMD & REV_MON.CMD, R := ACK);
(* Transfer data to outputs *)
FWD_REV_ALARM := FWD_REV_FF.Q1;
FWD_CMD := FWD_MON.CMD & NOT FWD_REV_ALARM;
FWD_ALARM := FWD_MON.ALARM;
REV_CMD := REV_MON.CMD & NOT FWD_REV_ALARM;
REV_ALARM := REV_MON.ALARM;
KLAXON := FWD_ALARM OR REV_ALARM OR FWD_REV_ALARM;
```

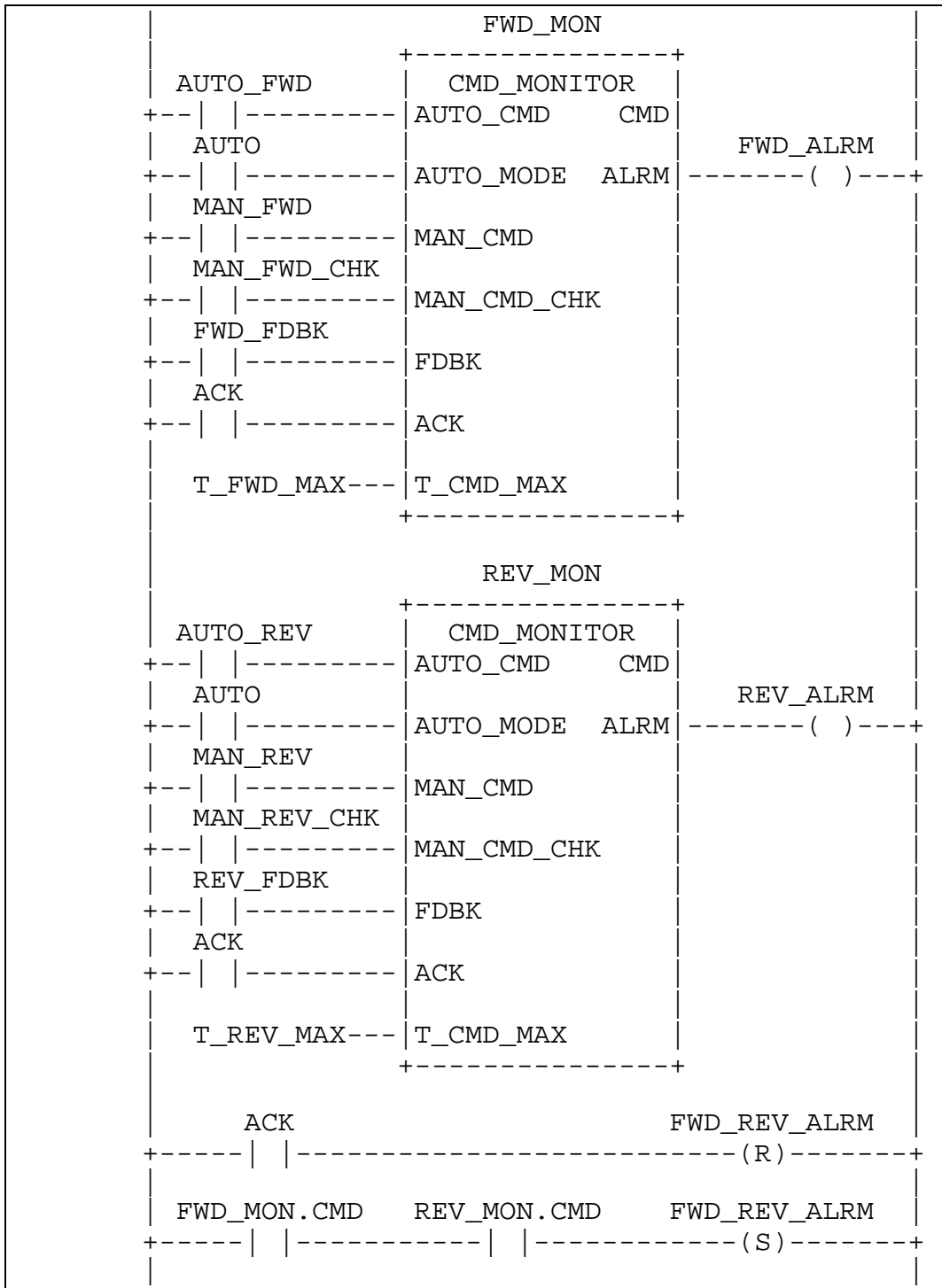
The body of function block FWD_REV_MON in the IL language is:

```
LD      AUTO                                (* Load common inputs *)
ST      FWD_MON.AUTO_MODE
ST      REV_MON.AUTO_MODE
LD      ACK
ST      FWD_MON.ACK
ST      REV_MON.ACK
ST      FWD_REV_FF.R
LD      AUTO_FWD                            (* Load inputs to FWD_MON *)
ST      FWD_MON.AUTO_CMD
LD      MAN_FWD
ST      FWD_MON.MAN_CMD
LD      MAN_FWD_CHK
ST      FWD_MON.MAN_CMD_CHK
LD      T_FWD_MAX
ST      FWD_MON.T_CMD_MAX
LD      FWD_FDBK
ST      FWD_MON.FDBK
CAL     FWD_MON                            (* Activate FWD_MON *)
LD      AUTO_REV                            (* Load inputs to REV_MON *)
ST      REV_MON.AUTO_CMD
LD      MAN_REV
ST      REV_MON.MAN_CMD
LD      MAN_REV_CHK
ST      REV_MON.MAN_CMD_CHK
LD      T_REV_MAX
ST      REV_MON.T_CMD_MAX
LD      REV_FDBK
ST      REV_MON.FDBK
CAL     REV_MON                            (* Activate REV_MON *)
LD      FWD_MON.CMD                        (* Check for contention *)
AND     REV_MON.CMD
S1      FWD_REV_FF                        (* Latch contention condition *)
LD      FWD_REV_FF.Q
ST      FWD_REV_ALARM                    (* Contention alarm *)
LD      FWD_MON.CMD                        (* "Forward" command and
alarm *)
AND     FWD_REV_ALARM
N
ST      FWD_CMD
LD      FWD_MON.ALARM
ST      FWD_ALARM
LD      REV_MON.CMD                        (* "Reverse" command and
alarm *)
AND     FWD_REV_ALARM
N
ST      REV_CMD
LD      REV_MON.ALARM
ST      REV_ALARM
OR      FWD_ALARM                        (* OR all alarms *)
OR      FWD_REV_ALARM
ST      KLAXON
```


The body of function block FWD_REV_MON in the FBD language is:

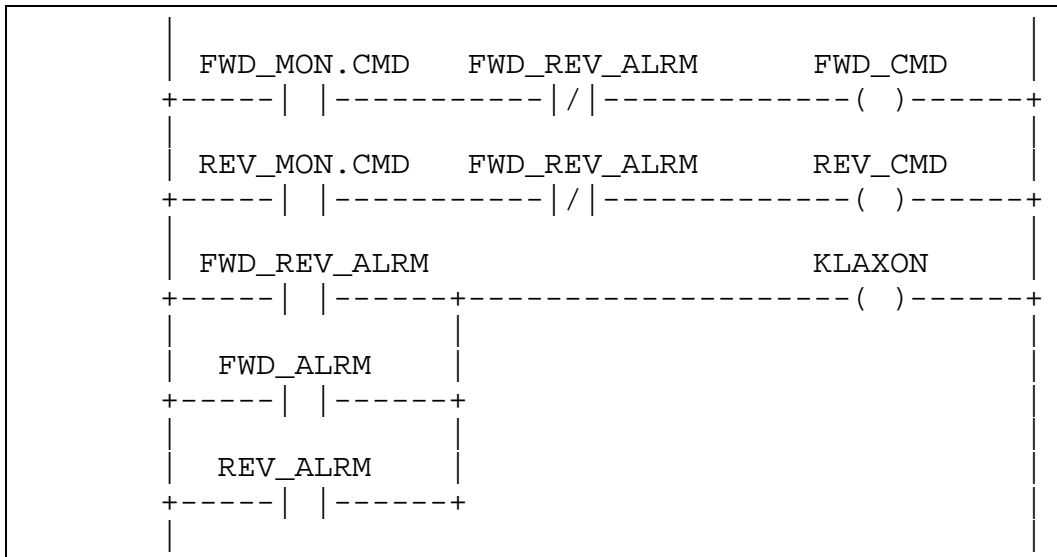


The body of function block FWD_REV_MON in the LD language is:



(continued on following page)

(FWD_REV_MON function block body - LD language - continued)



F.4 Function block STACK_INT

This function block provides a stack of up to 128 integers. The usual stack operations of PUSH and POP are provided by edge-triggered Boolean inputs. An overriding reset (R1) input is provided; the maximum stack depth (N) is determined at the time of resetting. In addition to the top-of-stack data (OUT), Boolean outputs are provided indicating stack empty and stack overflow states.

A textual form of the declaration of this function block is:

```
FUNCTION_BLOCK STACK_INT
  VAR_INPUT PUSH, POP: BOOL R_EDGE; (* Basic stack operations
    R1 : BOOL ; (* Over-riding reset *)
    IN : INT ; (* Input to be pushed *)
    N : INT ; (* Maximum depth after reset
  END_VAR
  VAR_OUTPUT EMPTY : BOOL := 1 ; (* Stack empty *)
    OFLO : BOOL := 0 ; (* Stack overflow *)
    OUT : INT := 0 ; (* Top of stack data *)
  END_VAR
  VAR STK : ARRAY[0..127] OF INT; (* Internal stack *)
    NI : INT :=128 ; (* Storage for N upon reset
    PTR : INT := -1 ; (* Stack pointer *)
  END_VAR
  (* Function Block body *)
END_FUNCTION_BLOCK
```


The body of function block STACK_INT in the IL language is:

	LD	R1	(* Dispatch on operations *)
	JMPC	RESET	
	LD	POP	
	ANDN	EMPTY	(* Don't pop empty stack *)
	JMPC	POP_STK	
	LD	PUSH	
	ANDN	OFLO	(* Don't push overflowed stack *)
	JMPC	PUSH_STK	
	RET		(* Return if no operations active *)
RESET:	LD	0	(* Stack reset operations *)
	ST	OFLO	
	LD	1	
	ST	EMPTY	
	LD	-1	
	ST	PTR	
	CAL	LIMIT(MN:=1,IN:=N,MX:=128)	
	ST	NI	
	JMP	ZRO_OUT	
POP_STK:	LD	0	
	ST	OFLO	(* Popped stack is not overflowing *)
	LD	PTR	
	SUB	1	
	ST	PTR	
	LT	0	(* Empty when PTR < 0 *)
	ST	EMPTY	
	JMPC	ZRO_OUT	
	LD	STK[PTR]	
	JMP	SET_OUT	
PUSH_STK:	LD	0	
	ST	EMPTY	(* Pushed stack is not empty *)
	LD	PTR	
	ADD	1	
	ST	PTR	
	EQ	NI	(* Overflow when PTR = NI *)
	ST	OFLO	
	JMPC	ZRO_OUT	
	LD	IN	
	ST	STK[PTR]	(* Push IN onto STK *)
	JMP	SET_OUT	
ZRO_OUT	LD	0	(* OUT=0 for EMPTY or OFLO *)
:			
SET_OUT:	ST	OUT	

F.5 Function block MIX_2_BRIX

Function block MIX_2_BRIX is to control the mixing of two bricks of solid material, brought one at a time on a belt, with weighed quantities of two liquid components, A and B, as shown in figure F.1. A "Start" (ST) command, which may be manual or automatic, initiates a measurement and mixing cycle beginning with simultaneous weighing and brick transport as follows:

- Liquid A is weighed up to mark "a" of the weighing unit, then liquid B is weighed up to mark "b", followed by filling of the mixer from weighing unit C;
- Two bricks are transported by belt into the mixer.

The cycle ends with the mixer rotating and finally tipping after a predetermined time "t1". Rotation of the mixer continues while it is emptying.

The scale reading "WC" is given as four BCD digits, and will be converted to type INT for internal operations. It is assumed that the tare (empty weight) "z" has been previously determined.

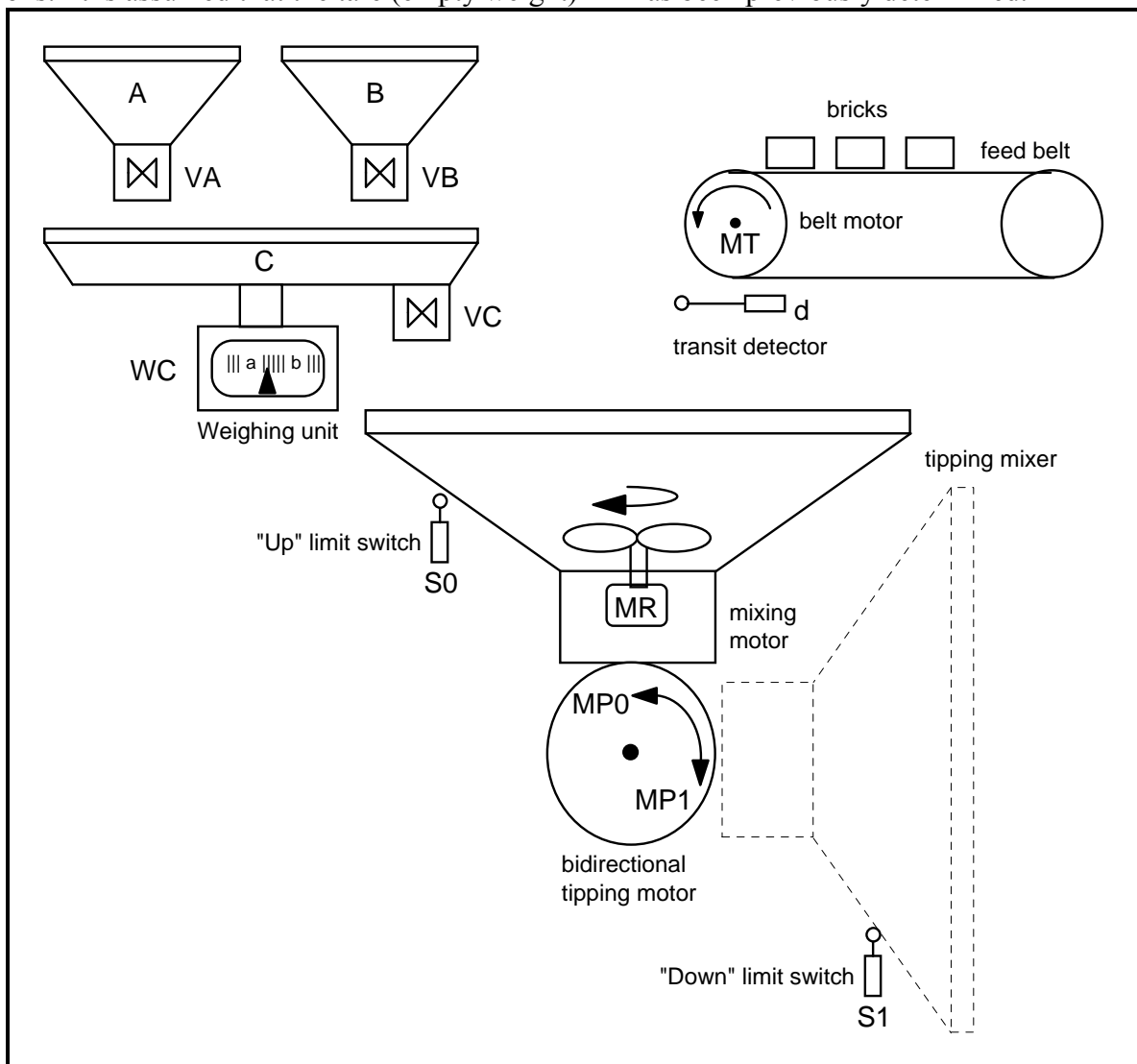


Figure F.1 - Function block MIX_2_BRIX - Physical model

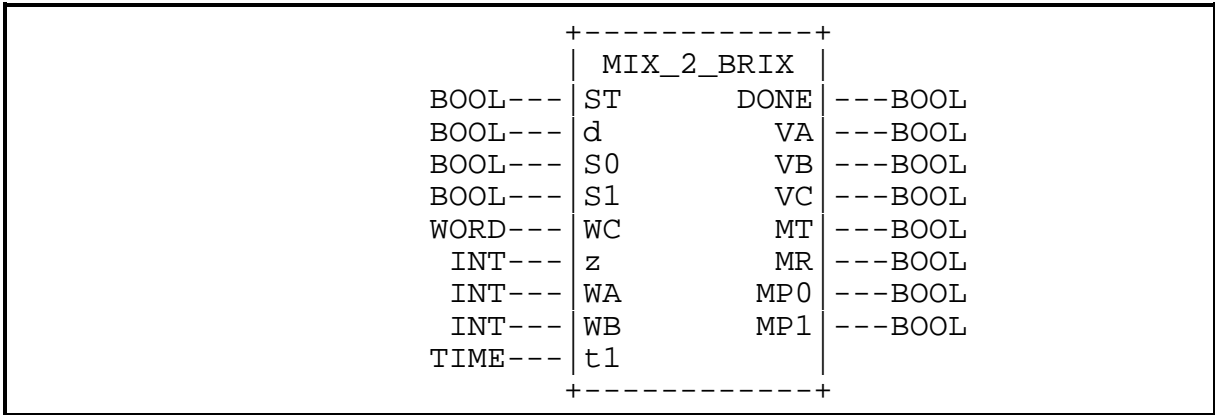
The textual form of the declaration of this function block is:

```

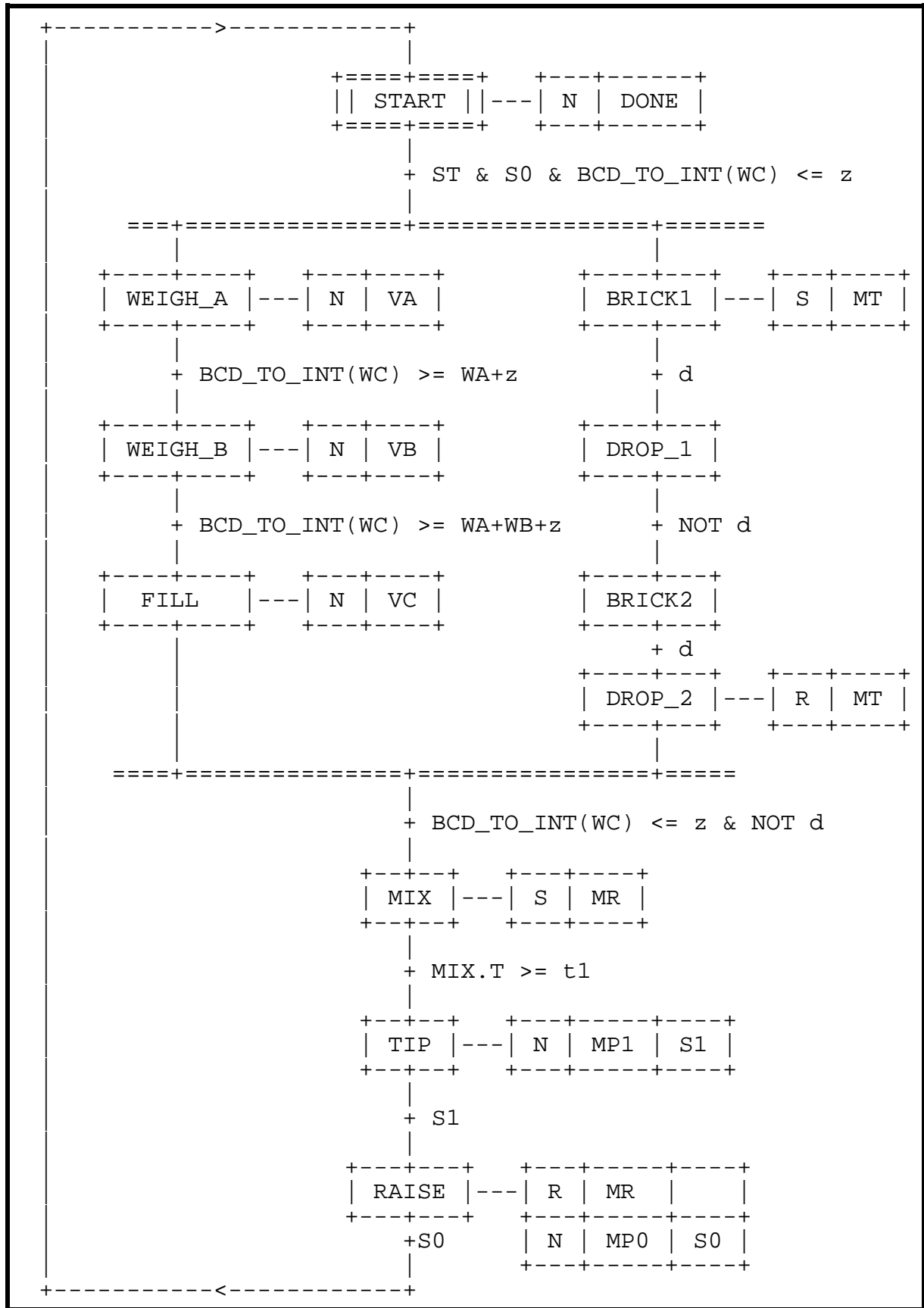
FUNCTION_BLOCK MIX_2_BRIX
VAR_INPUT
    ST : BOOL ;      (* "Start" command *)
    d  : BOOL ;      (* Transit detector *)
    S0 : BOOL ;      (* "Mixer up" limit switch *)
    S1 : BOOL ;      (* "Mixer down" limit switch *)
    WC : WORD;       (* Current scale reading in BCD *)
    z  : INT ;       (* Tare (empty) weight *)
    WA : INT ;       (* Desired weight of A *)
    WB : INT ;       (* Desired weight of B *)
    t1 : TIME ;      (* Mixing time *)
END_VAR
VAR_OUTPUT
    DONE ,          (* Valve "A" : 0 - close, 1 - open *)
    VA ,           (* Valve "B" : 0 - close, 1 - open *)
    VB ,           (* Valve "C" : 0 - close, 1 - open *)
    MT ,           (* Feed belt motor *)
    MR ,           (* Mixer rotation motor *)
    MP0 ,          (* Tipping motor "up" command *)
    MP1 : BOOL;    (* Tipping motor "down" command *)
END_VAR
(* Function block body *)
END_FUNCTION_BLOCK

```

A graphical declaration is:



The body of function block MIX_2_BRIX using graphical SFC elements with transition conditions in the ST language is:



The body of function block MIX_2_BRIX in a textual SFC representation using ST language elements is:

```

INITIAL_STEP START: DONE(N); END_STEP
  
```

```

TRANSITION FROM START TO (WEIGH_A, BRICK1)
    := ST & S0 & BCD_TO_INT(WC) <= z ;
END_TRANSITION
STEP WEIGH_A: VA(N); END_STEP
TRANSITION FROM WEIGH_A TO WEIGH_B := BCD_TO_INT(WC) >=
WA+z ;
END_TRANSITION
STEP WEIGH_B: VB(N); END_STEP
TRANSITION FROM WEIGH_B TO FILL := BCD_TO_INT(WC) >=
WA+WB+z ;
END_TRANSITION
STEP FILL: VC(N); END_STEP
STEP BRICK1: MT(S); END_STEP
TRANSITION FROM BRICK1 TO DROP_1 := d ; END_TRANSITION
STEP DROP_1: END_STEP
TRANSITION FROM DROP_1 TO BRICK2 := NOT d ; END_TRANSITION
STEP BRICK2: END_STEP
TRANSITION FROM BRICK2 TO DROP_2 := d ; END_TRANSITION
STEP DROP_1: MT(R); END_STEP
TRANSITION FROM (FILL,DROP_2) TO MIX
    := BCD_TO_INT(WC) <= z & NOT d ;
END_TRANSITION
STEP MIX: MR(S); END_STEP
TRANSITION FROM MIX TO TIP := MIX.T >= t1 ; END_TRANSITION
STEP TIP: MP1(N); END_STEP
TRANSITION FROM TIP TO RAISE := S1 ; END_TRANSITION
STEP RAISE: MR(R); MP0(N); END_STEP
TRANSITION FROM RAISE TO START := S0 ; END_TRANSITION

```

F.6 Analog signal processing

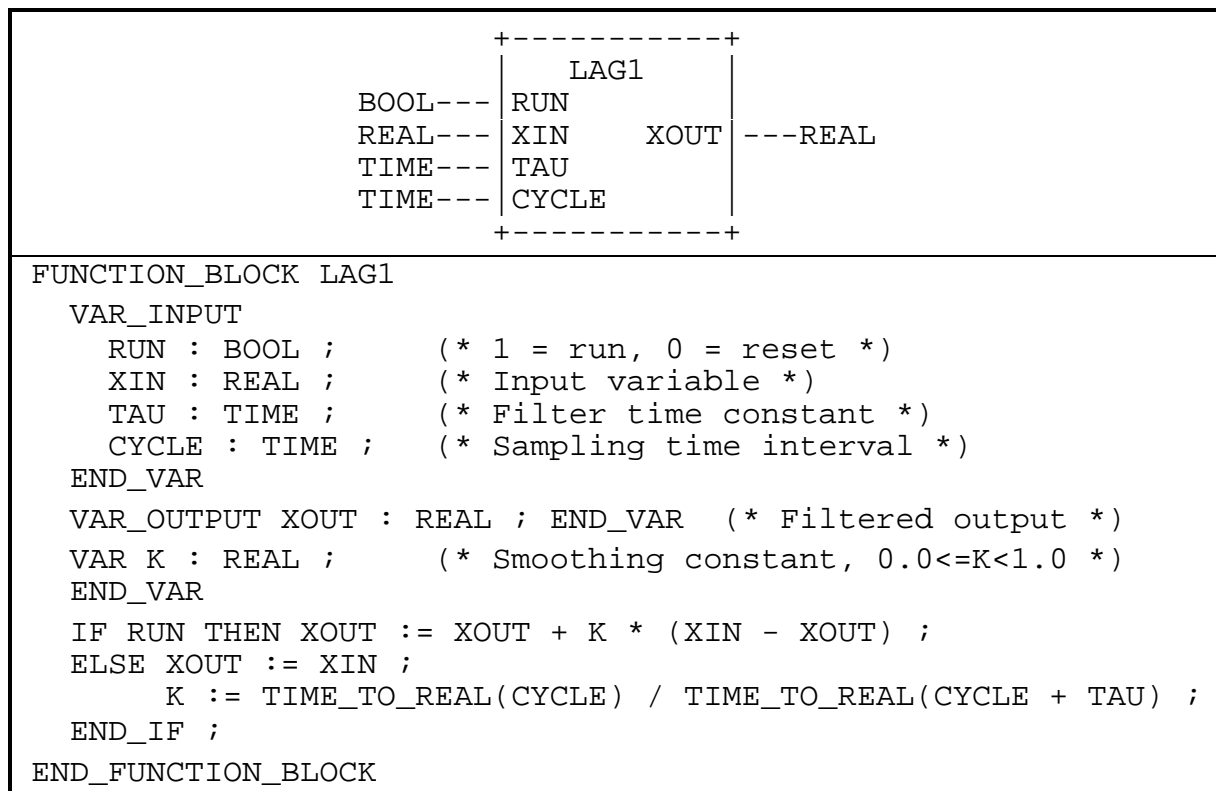
The purpose of this portion of of this annex is to illustrate the application of the programming languages defined in this standard to accomplish the basic measurement and control functions of process-computer aided automation. The blocks shown below are not restricted to analog signals; they may be used to process any variables of the appropriate types. Similarly, other functions and function blocks defined in this standard (e.g., mathematical functions) can be used for the processing of variables which may appear as analog signals at the programmable controller's I/O terminals.

These function blocks can be typed with respect to the input and output variables shown below as REAL (e.g., XIN, XOUT) by appending the appropriate data type name, e.g., LAG1_LREAL. The default data type for these variables is REAL.

These examples are given for illustrative purposes only. Manufacturers may have varying implementations of analog signal processing elements. The inclusion of these examples is not intended to preclude the standardization of such elements by the appropriate standards bodies.

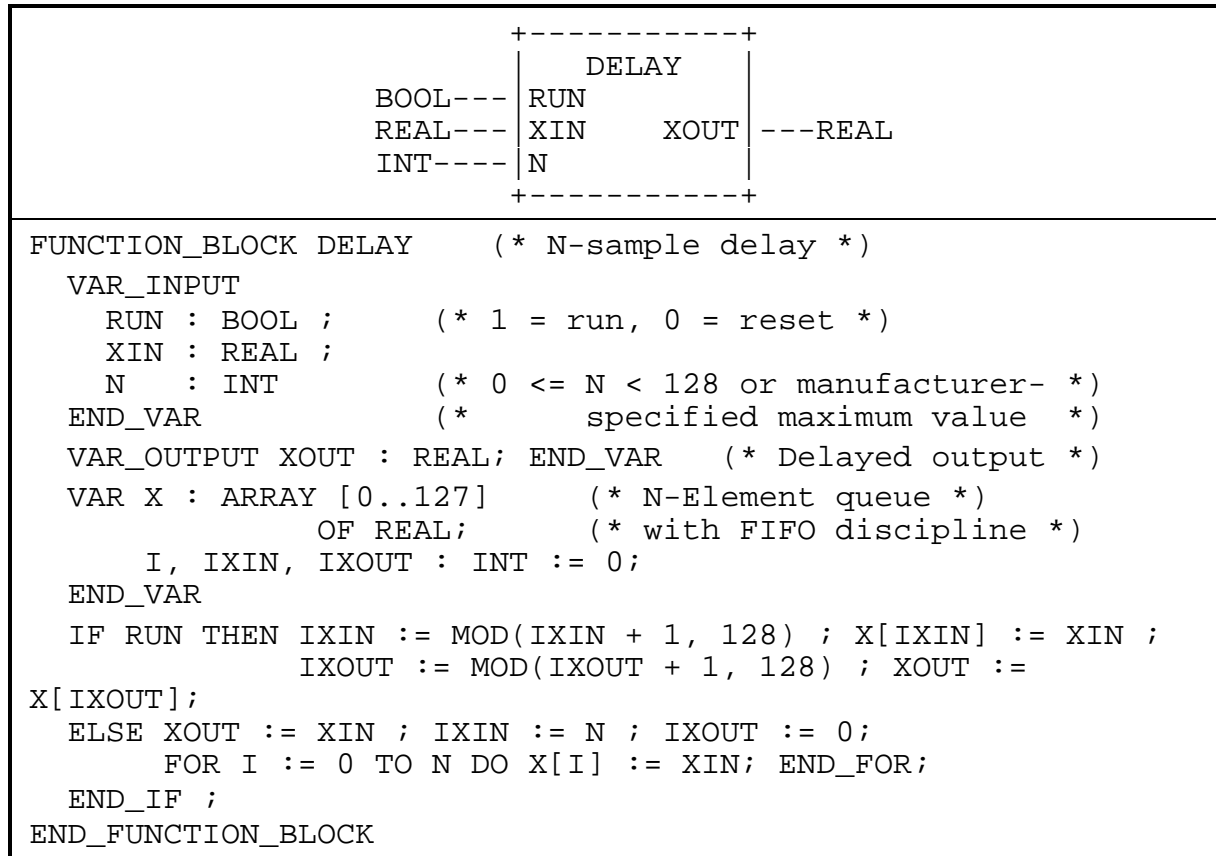
F.6.1 Function block LAG1

This function block implements a first-order lag filter.



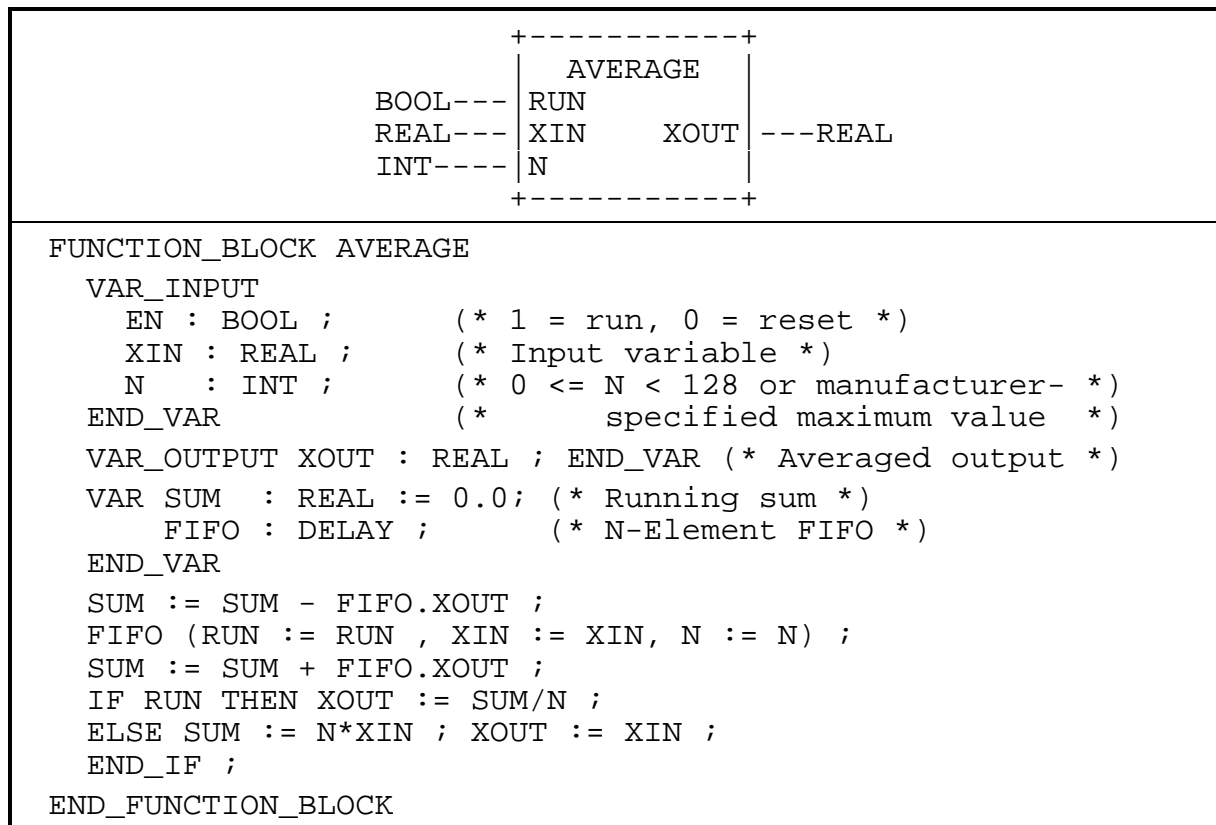
F.6.2 Function block DELAY

This function block implements an N-sample delay.



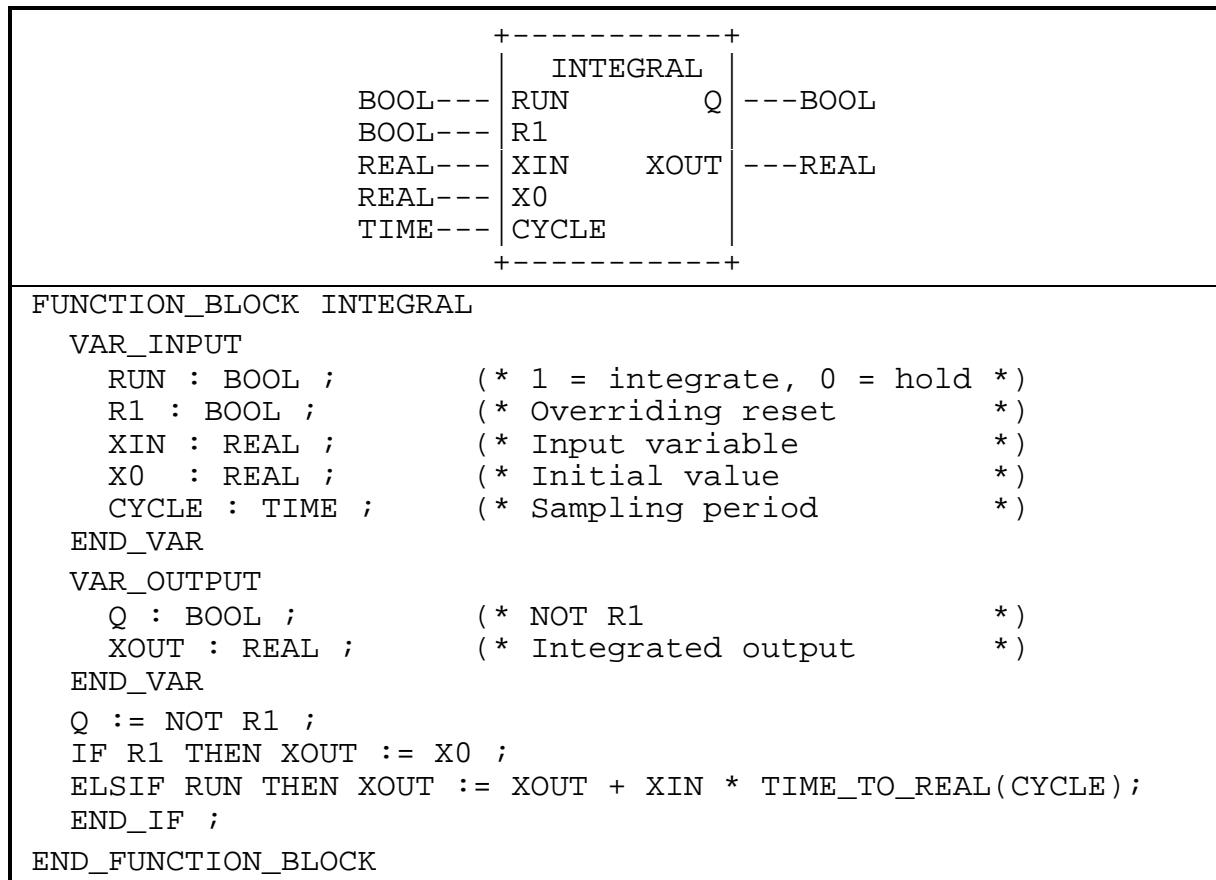
F.6.3 Function block AVERAGE

This function block implements a running average over N samples.



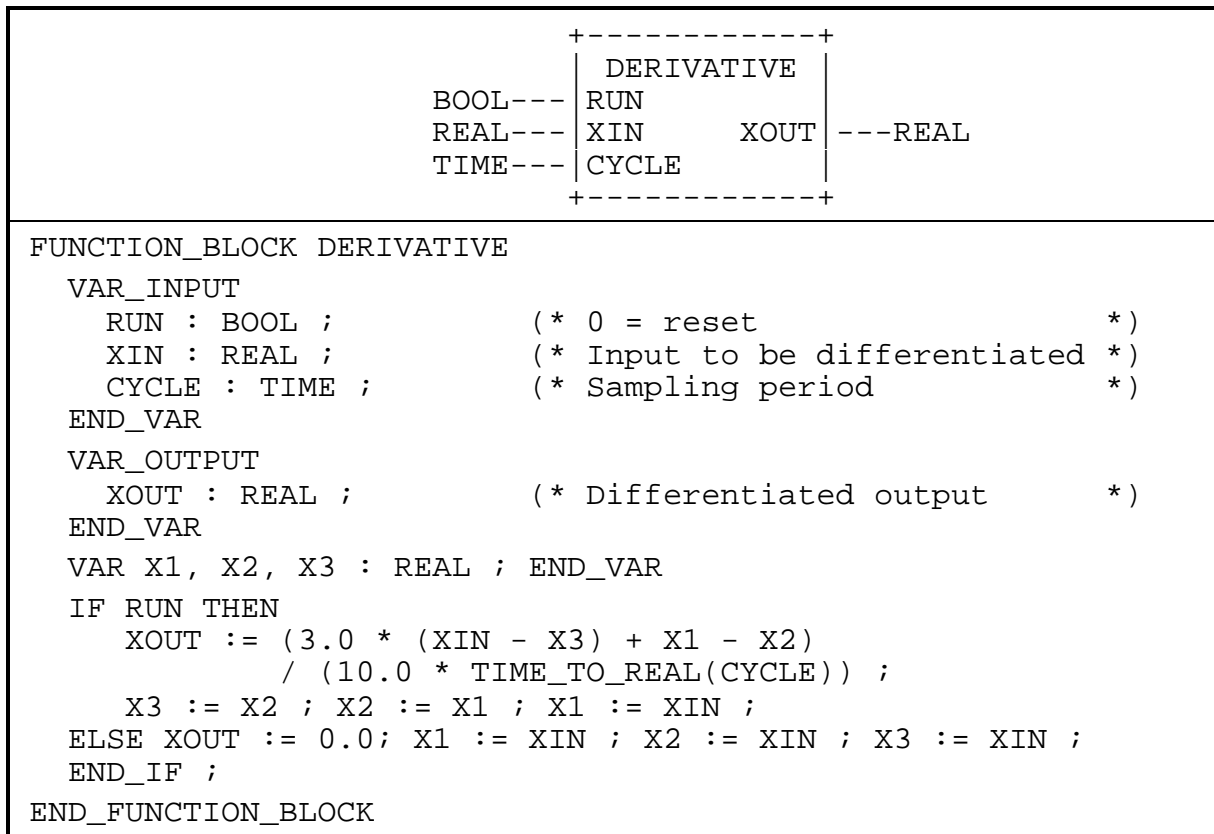
F.6.4 Function block INTEGRAL

This function block implements integration over time.



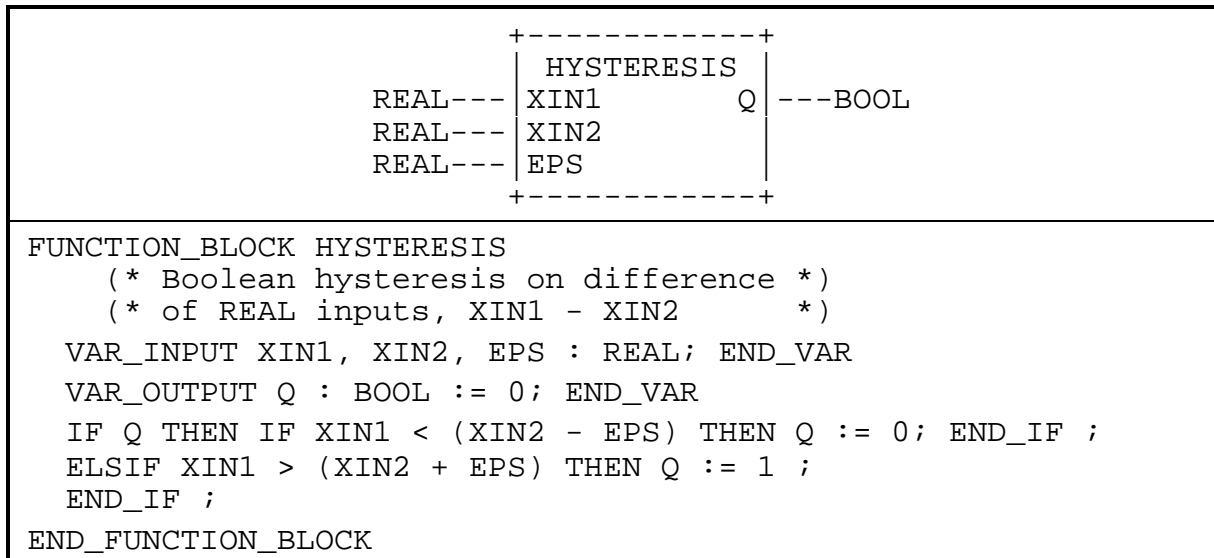
F.6.5 Function block DERIVATIVE

This function block implements differentiation with respect to time.



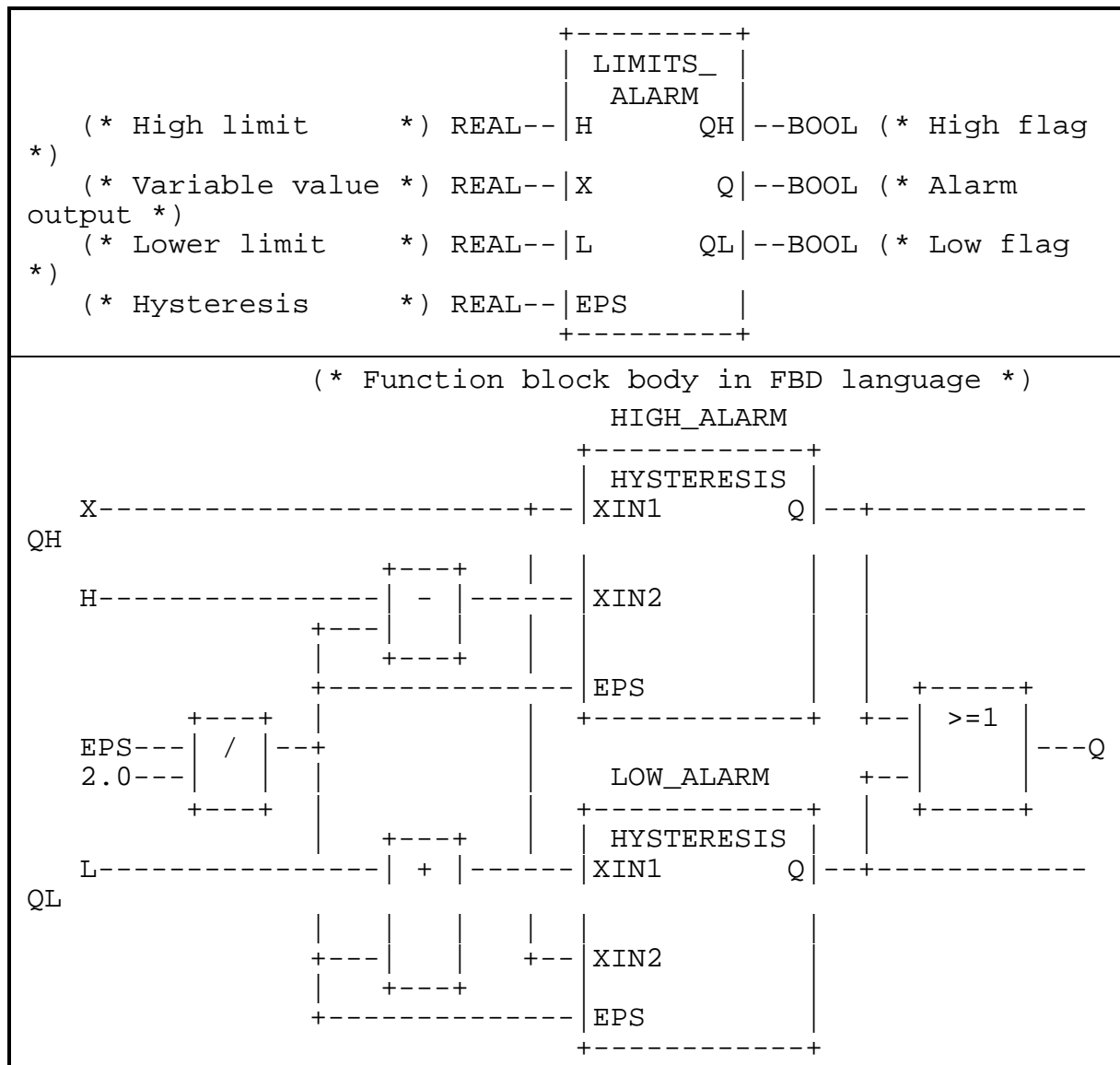
F.6.6 Function block HYSTERESIS

This function block implements Boolean hysteresis on the difference of REAL inputs.



F.6.7 Function block LIMITS_ALARM

This function block implements a high/low limit alarm with hysteresis on both outputs.



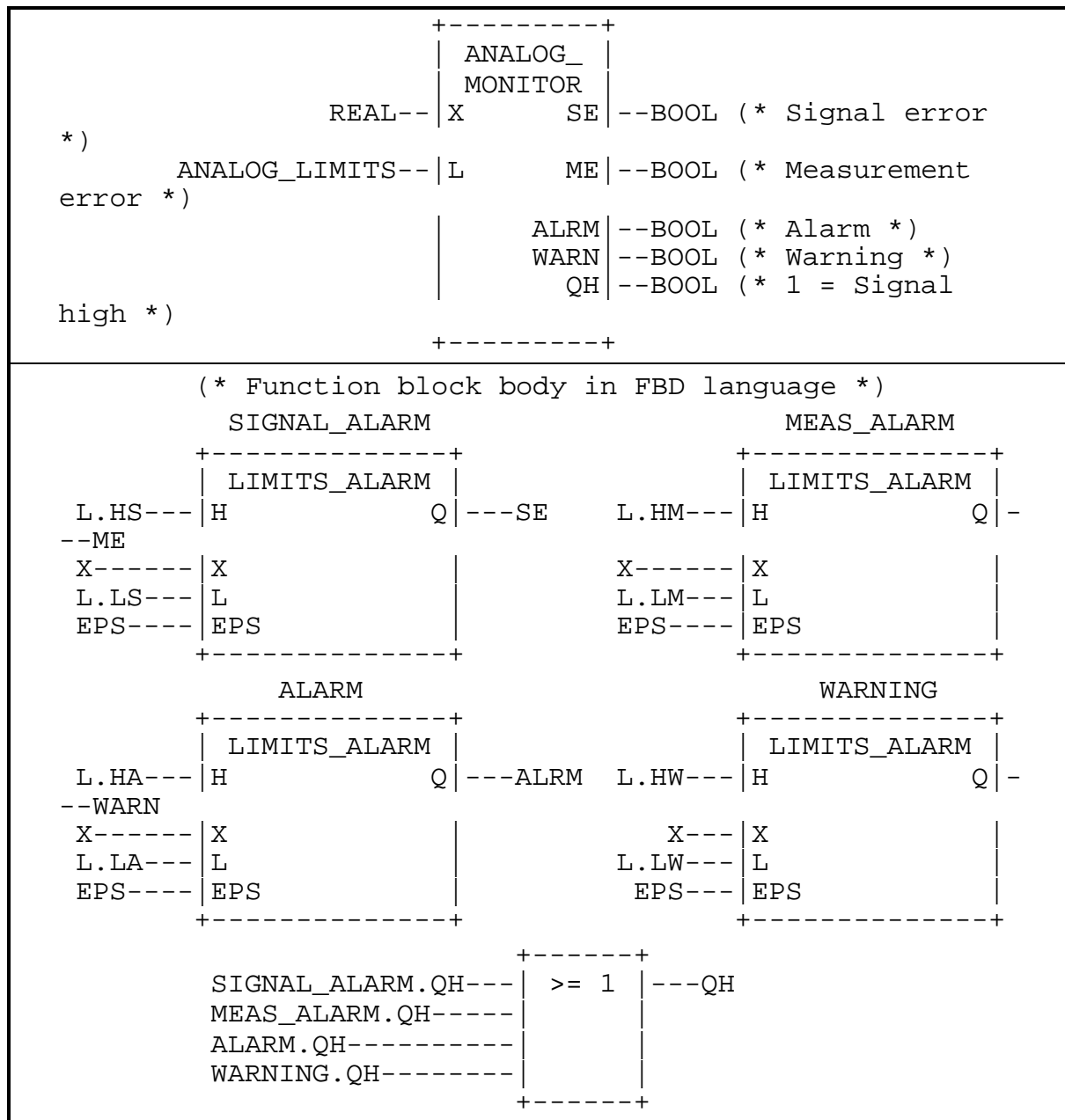
F.6.8 Structure ANALOG_LIMITS

This data type implements the declarations of parameters for analog signal monitoring.

```
TYPE ANALOG_LIMITS :  
  STRUCT  
    HS : REAL ;      (* High end of signal range *)  
    HM : REAL ;      (* High end of measurement range *)  
    HA : REAL ;      (* High alarm threshold *)  
    HW : REAL ;      (* High warning threshold *)  
    NV : REAL ;      (* Nominal value *)  
    EPS : REAL ;     (* Hysteresis *)  
    LW : REAL ;      (* Low warning threshold *)  
    LA : REAL ;      (* Low alarm threshold *)  
    LM : REAL ;      (* Low end of measurement range *)  
    LS : REAL ;      (* Low end of signal range *)  
  END_STRUCT  
END_TYPE
```

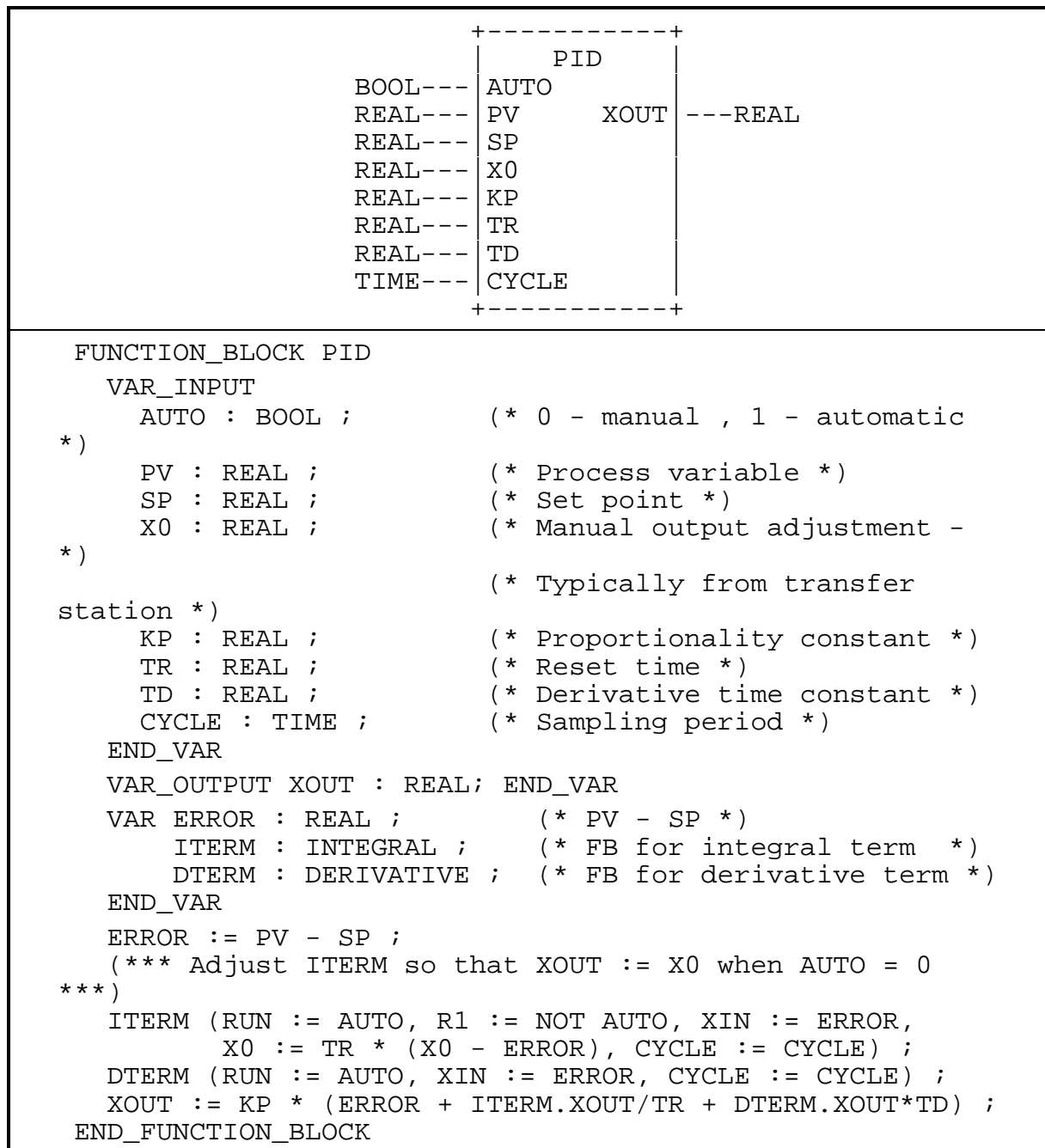
F.6.9 Function block ANALOG_MONITOR

This function block implements analog signal monitoring.



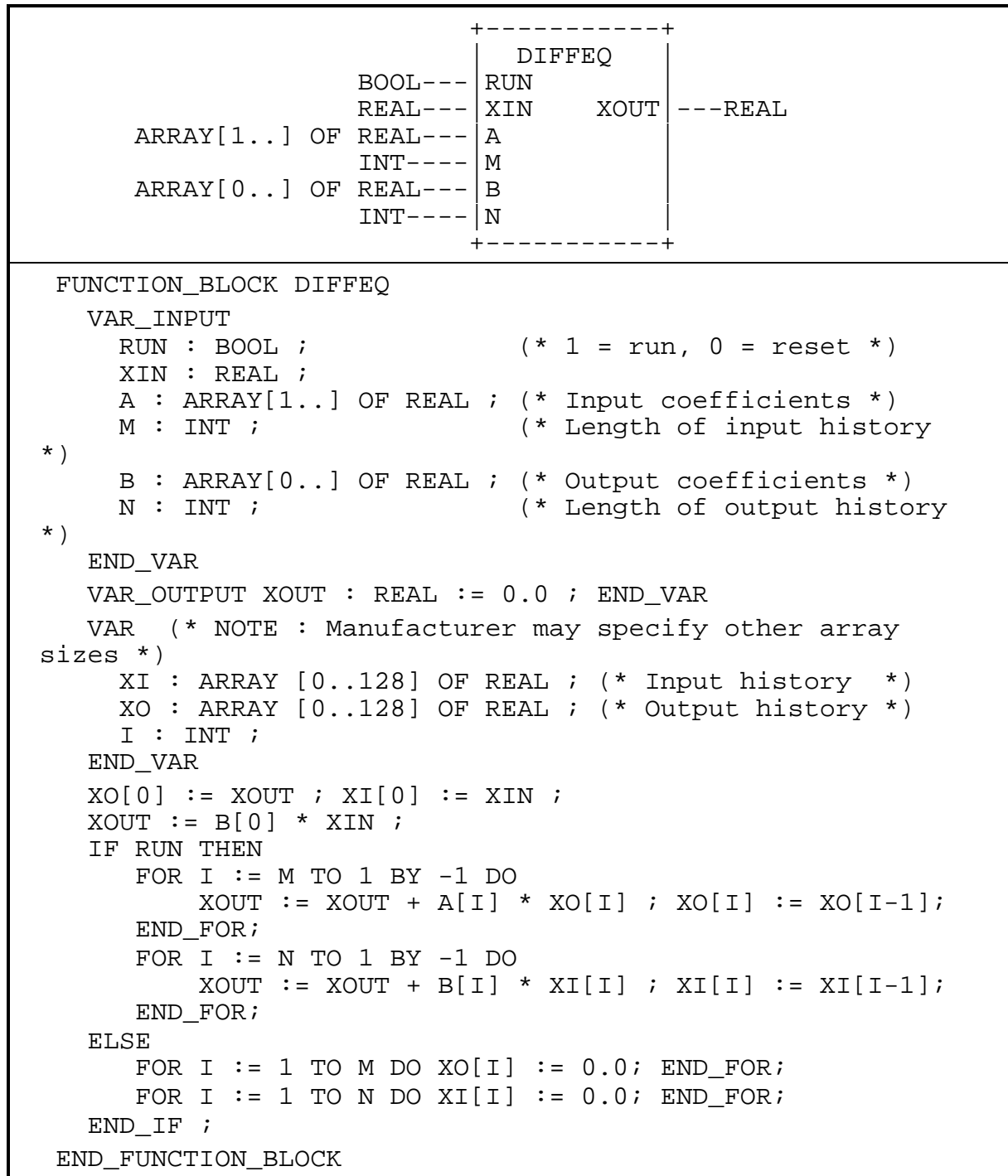
F.6.10 Function block PID

This function block implements Proportional + Integral + Derivative control action. The functionality is derived by functional composition of previously declared function blocks.



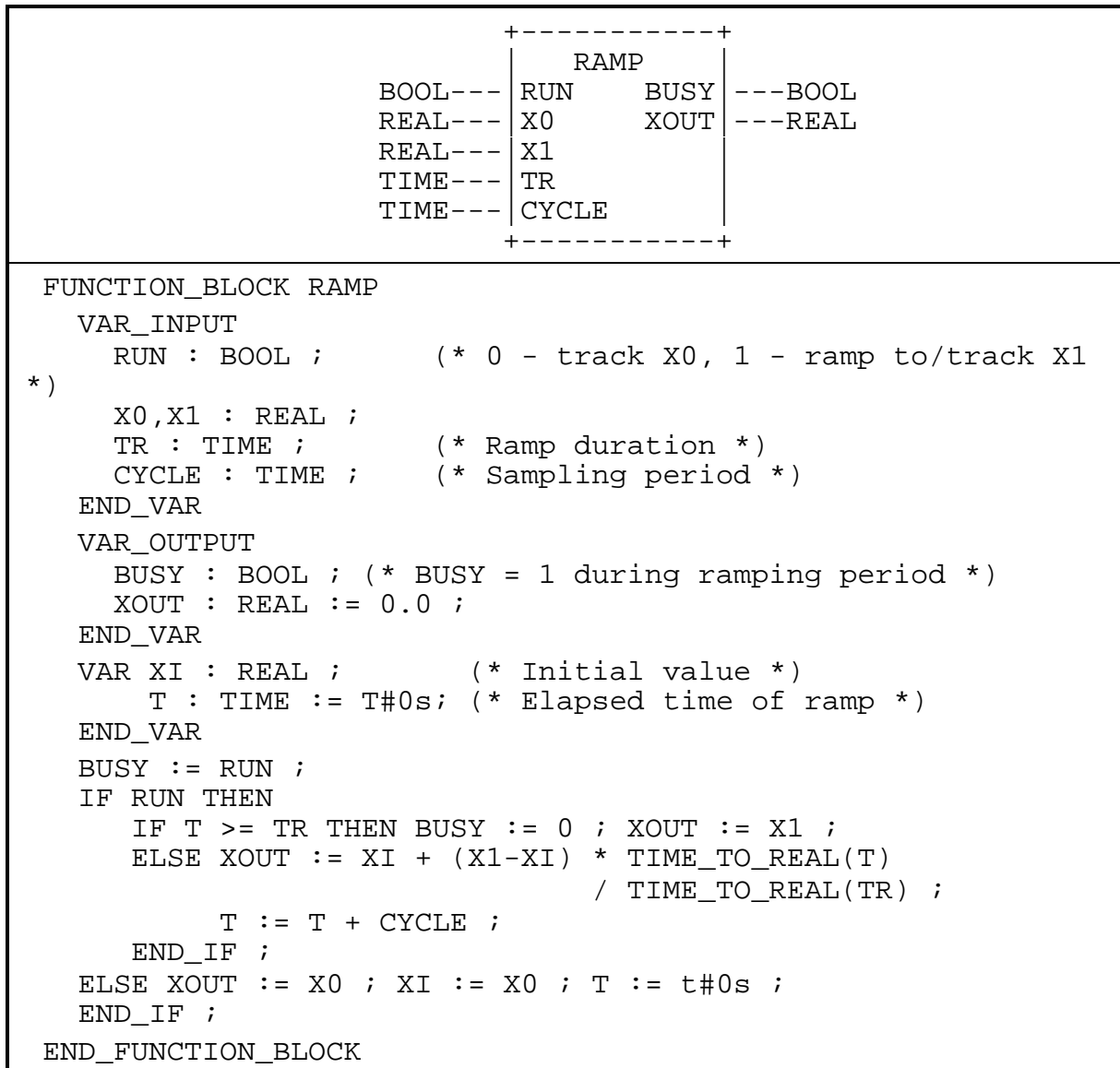
F.6.11 Function block DIFFEQ

This function block implements a general difference equation.



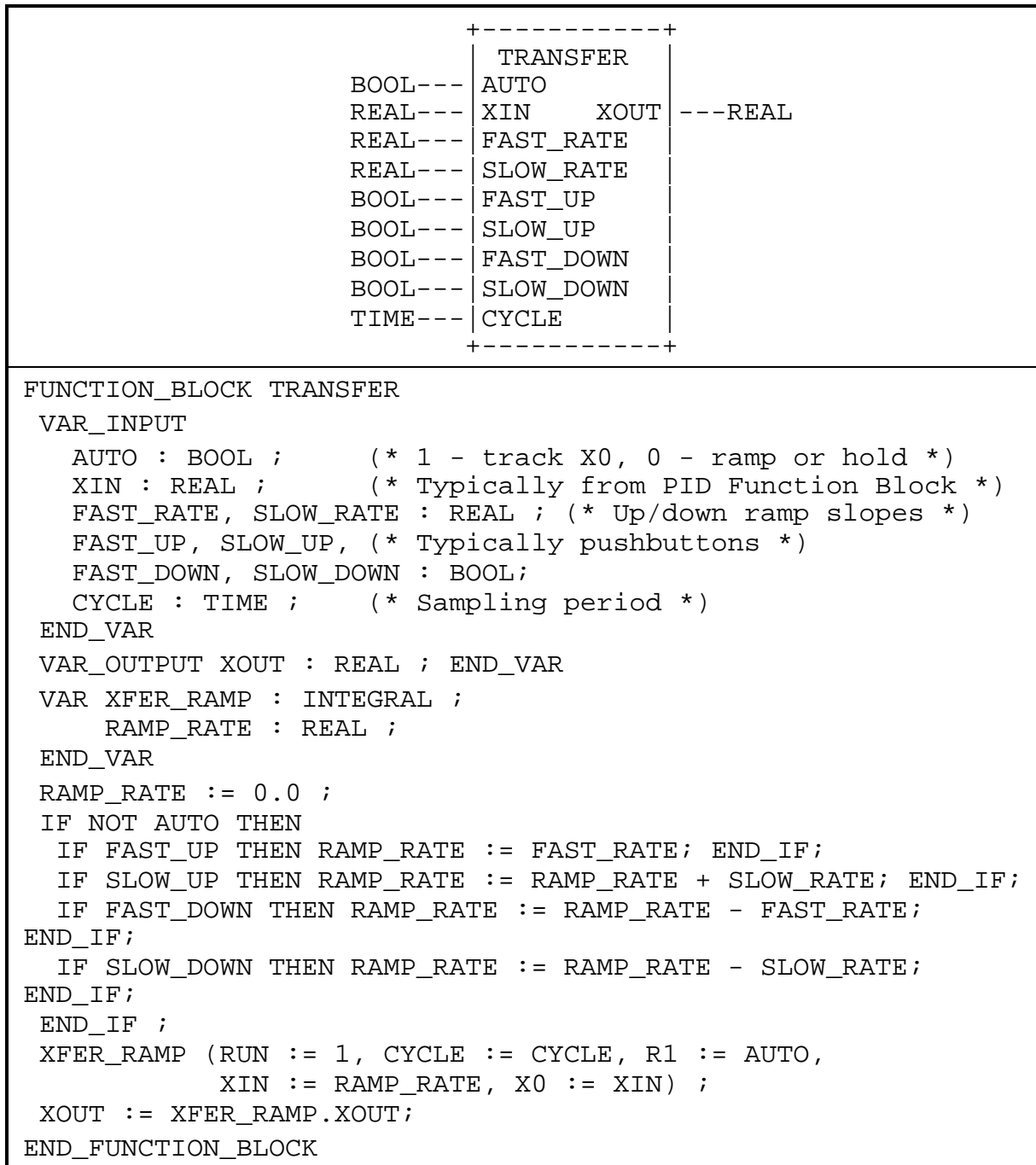
F.6.12 Function block RAMP

This function block implements a time-based ramp.



F.6.13 Function block TRANSFER

This function block implements a manual transfer station with bumpless transfer.



F.7 Program GRAVEL

A control system is to be used to measure an operator-specified amount of gravel from a silo into an intermediate bin, and to convey the gravel after measurement from the bin into a truck.

The quantity of gravel to be transferred is specified via a thumbwheel with a range of 0 to 99 units. The amount of gravel in the bin is indicated on a digital display.

For safety reasons, visual and audible alarms must be raised immediately when the silo is empty. The signalling functions are to be implemented in the control program.

A graphic representation of the control problem is shown in figure F.2, while the variable declarations for the control program are given in figure F.3.

As shown in figure F.4, the operation of the system consists of a number of major states, beginning with filling of the bin upon command from the FILL push button. After the bin is filled, the truck loading sequence begins upon command by the LOAD pushbutton when a truck is present on the ramp. Loading consists of a "run-in" period for starting the conveyor, followed by dumping of the bin contents onto the conveyor. After the bin has emptied, the conveyor "runs out" for a predetermined time to assure that all gravel has been loaded to the truck. The loading sequence is stopped and re-initialized if the truck leaves the ramp or if the automatic control is stopped by the OFF push button.

Figure F.5 shows the OFF/ON sequence of automatic control states, as well as the generation of display blinking pulses and conveyor motor gating when the control is ON.

Bin level monitoring, operator interface and display functions are defined in figure F.6.

A textual version of the body of program GRAVEL is given in figure F.7, using the ST language with SFC elements.

An example configuration for program GRAVEL is given in figure F.8.

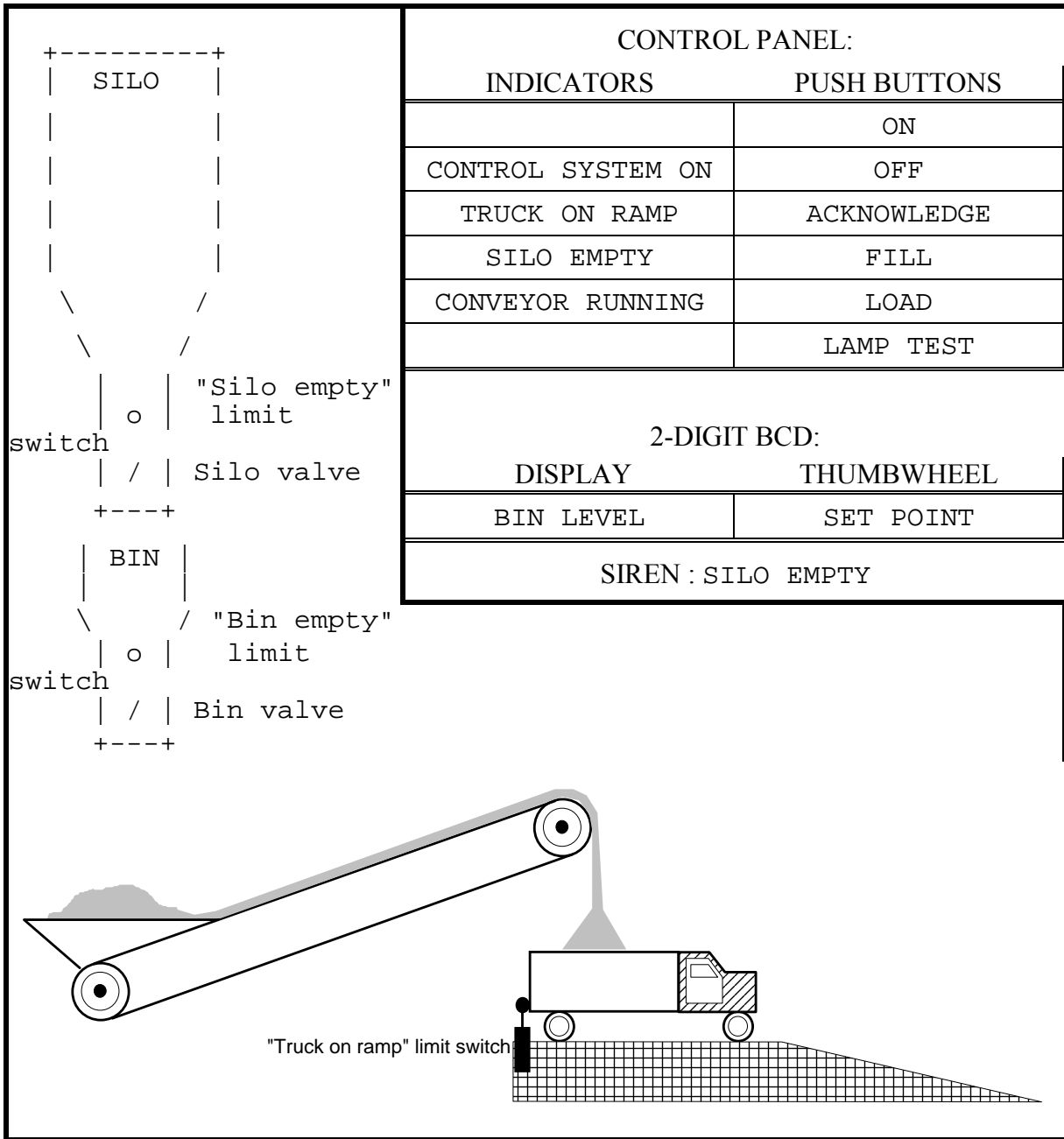


Figure F.2 - Gravel measurement and loading system

```

PROGRAM GRAVEL  (* Gravel measurement and loading system *)
VAR_INPUT
  OFF_PB          : BOOL ;
  ON_PB           : BOOL ;
  FILL_PB         : BOOL ;
  SIREN_ACK       : BOOL ;
  LOAD_PB         : BOOL ; (* Load truck from bin *)
  JOG_PB          : BOOL ;
  LAMP_TEST       : BOOL ;
  TRUCK_ON_RAMP   : BOOL ; (* Optical sensor *)
  SILO_EMPTY_LS   : BOOL ;
  BIN_EMPTY_LS    : BOOL ;
  SETPOINT        : BYTE ; (* 2-digit BCD *)
END_VAR
VAR_OUTPUT
  CONTROL_LAMP    : BOOL ;
  TRUCK_LAMP      : BOOL ;
  SILO_EMPTY_LAMP : BOOL ;
  CONVEYOR_LAMP   : BOOL ;
  CONVEYOR_MOTOR  : BOOL ;
  SILO_VALVE      : BOOL ;
  BIN_VALVE       : BOOL ;
  SIREN           : BOOL ;
  BIN_LEVEL       : BYTE ;
END_VAR
VAR
  BLINK_TIME : TIME; (* BLINK ON/OFF time *)
  PULSE_TIME : TIME; (* LEVEL_CTR increment interval *)
  RUNOUT_TIME: TIME; (* Conveyor running time after loading
*)
  RUN_IN_TIME: TIME; (* Conveyor running time before loading
*)
  SILENT_TIME: TIME; (* Siren silent time after SIREN_ACK *)
  OK_TO_RUN  : BOOL; (* 1 = Conveyor is allowed to run *)
  (* Function Blocks *)
  BLINK: TON; (* Blinker OFF period timer / ON output *)
  BLANK: TON; (* Blinker ON period timer / blanking pulse *)
  PULSE: TON; (* LEVEL_CTR pulse interval timer *)
  SIREN_FF: RS;
  SILENCE_TMR: TP; (* Siren silent period timer *)
END_VAR
VAR RETAIN LEVEL_CTR : CTU ; END_VAR
  (* Program body *)
END_PROGRAM

```

Figure F.3 - Declarations for Program GRAVEL

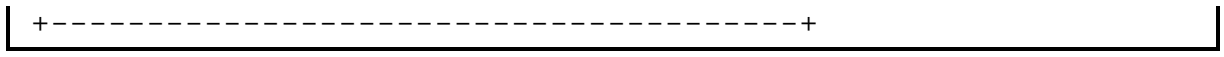


Figure F.4 - SFC of program GRAVEL body

```

+-----+
|         |
|         | + OFF_PB
|         |
| +-----+ +-----+ +-----+
+-----+
| | CONTROL_OFF | | MONITOR | |---| N |
MONITOR_ACTION |
| +-----+ +-----+ +-----+
+-----+
|         |
|         | + ON_PB & NOT OFF_PB
|         |
| +-----+ +-----+
+-----+
| CONTROL |--| N | CONTROL_ACTION |
| +-----+ +-----+
+-----+
|         |
|         | +-----+
+-----+
|         | | BLINK | BLANK | | | | | |
|         | | +++ | +-----+ |
|         | | +---O |&| | TON | | TON | |
|         | | CONTROL.X-- | |-----| IN Q |-----| IN Q |---+
|         | | +++ |---| PT | |---| PT |
|         | | | +-----+ | +-----+
|         | | BLINK_TIME-- +-----+
|         | | +++
|         | | CONTROL.X----- |&|
|         | | TRUCK_ON_RAMP-- | |---+-----OK_TO_RUN
|         | | +++ |
|         | | | +++
|         | | +-----+ |---|&|---
CONVEYOR_MOTOR |
|         | | JOG_PB----- | >=1 |-----| |
|         | | RUN_IN.X---- | | +++
|         | | DUMP_BIN.X-- | |
|         | | RUNOUT.X---- | |
|         | | +-----+

```

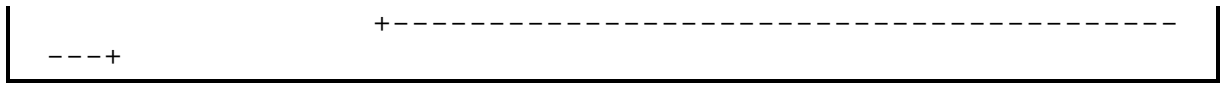


Figure F.5 - Body of program GRAVEL (continued)
Control state sequencing and monitoring

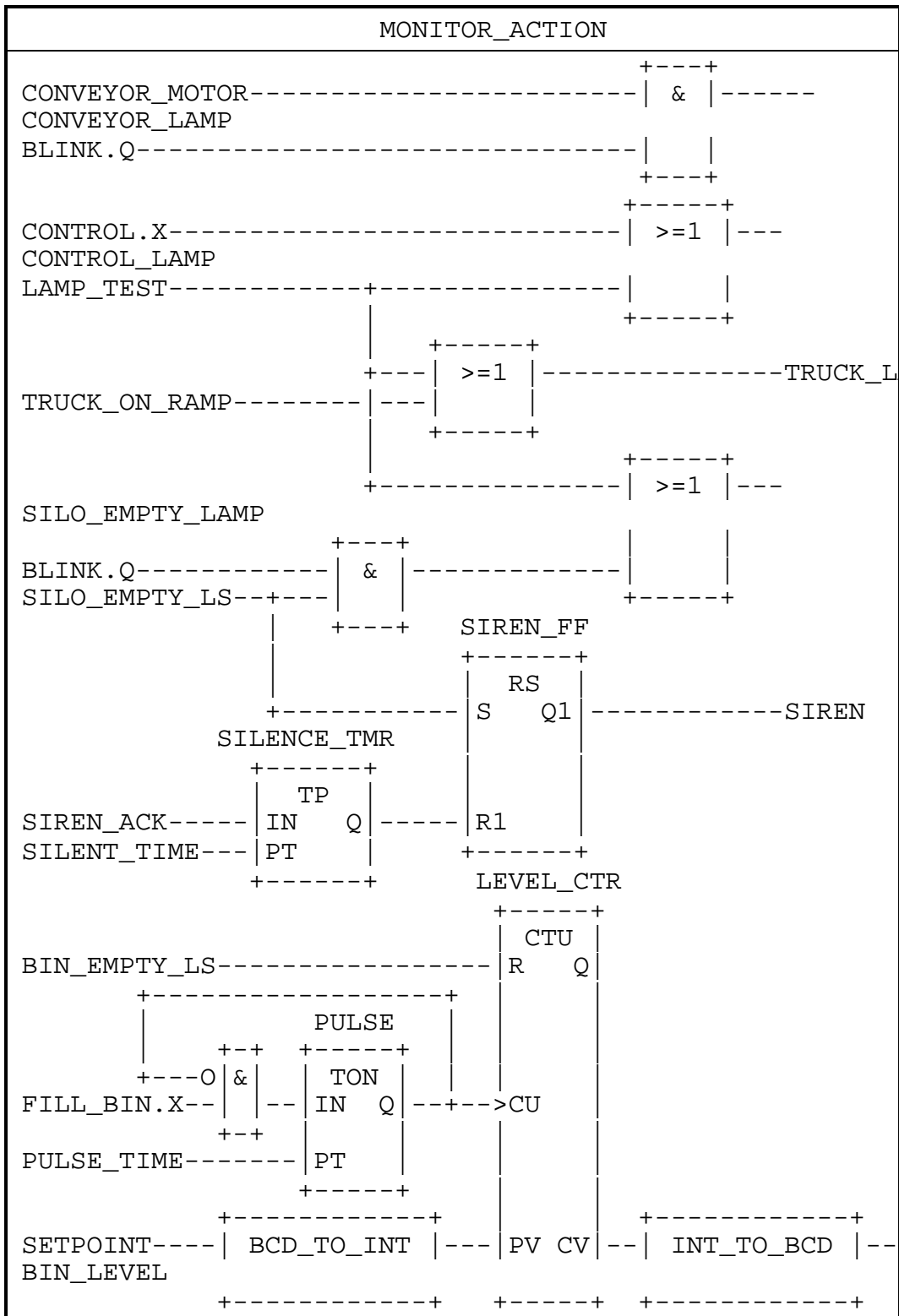


Figure F.6 - Body of action MONITOR_ACTION in FBD language

```

(* Major operating states *)
INITIAL_STEP START : END_STEP
TRANSITION FROM START TO FILL_BIN
    := FILL_PB & CONTROL.X ; END_TRANSITION
STEP FILL_BIN: SILO_VALVE(N); END_STEP
TRANSITION FROM FILL_BIN TO START
    := NOT FILL_PB OR NOT CONTROL.X ; END_TRANSITION
TRANSITION FROM FILL_BIN TO LOAD_WAIT :=
LEVEL_CTR.Q ;
END_TRANSITION
STEP LOAD_WAIT : END_STEP
TRANSITION FROM LOAD_WAIT TO RUN_IN
    := LOAD_PB & OK_TO_RUN ; END_TRANSITION
STEP RUN_IN : END_STEP
TRANSITION FROM RUN_IN TO LOAD_WAIT := NOT
OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM RUN_IN TO DUMP_BIN
    := RUN_IN.T > RUN_IN_TIME;
END_TRANSITION
STEP DUMP_BIN: BIN_VALVE(N); END_STEP
TRANSITION FROM DUMP_BIN TO LOAD_WAIT := NOT
OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM DUMP_BIN TO RUNOUT := BIN_EMPTY_LS
;
END_TRANSITION
STEP RUNOUT : END_STEP
TRANSITION FROM RUNOUT TO LOAD_WAIT := NOT
OK_TO_RUN ;
END_TRANSITION
TRANSITION FROM RUNOUT TO START
    := RUNOUT.T >= RUNOUT_TIME ; END_TRANSITION

```

Figure F.7 - Body of program GRAVEL in textual SFC representation using ST language elements
(continued on following page)


```

(* Control state sequencing *)
INITIAL_STEP CONTROL_OFF: END_STEP
TRANSITION FROM CONTROL_OFF TO CONTROL
    := ON_PB & NOT OFF_PB ; END_TRANSITION
STEP CONTROL: CONTROL_ACTION(N); END_STEP
ACTION CONTROL_ACTION:
    BLINK(EN:=CONTROL.X & NOT BLANK.Q, PT :=
BLINK_TIME) ;
    BLANK(EN:=BLINK.Q, PT := BLINK_TIME) ;
    OK_TO_RUN := CONTROL.X & TRUCK_ON_RAMP ;
    CONVEYOR_MOTOR :=
        OK_TO_RUN & OR(JOG_PB, RUN_IN.X, DUMP_BIN.X,
RUNOUT.X);
END_ACTION
TRANSITION FROM CONTROL TO CONTROL_OFF := OFF_PB ;
END_TRANSITION
(* Monitor Logic *)
INITIAL_STEP MONITOR: MONITOR_ACTION(N); END_STEP
ACTION MONITOR_ACTION:
    CONVEYOR_LAMP := CONVEYOR_MOTOR & BLINK.Q ;
    CONTROL_LAMP := CONTROL.X OR LAMP_TEST ;
    TRUCK_LAMP := TRUCK_ON_RAMP OR LAMP_TEST ;
    SILO_EMPTY_LAMP := BLINK.Q & SILO_EMPTY_LS OR
LAMP_TEST ;
    SILENCE_TMR(EN:=SIREN_ACK, PT:=SILENT_TIME) ;
    SIREN_FF(S:=SILO_EMPTY_LS, R1:=SILENCE_TMR.Q) ;
    SIREN := SIREN_FF.Q1 ;
    PULSE(EN:=FILL_BIN.X & NOT PULSE.Q,
PT:=PULSE_TIME) ;
    LEVEL_CTR(EN := BIN_EMPTY_LS, CU := PULSE.Q,
        PV := BCD_TO_INT(SETPOINT)) ;
    BIN_LEVEL := INT_TO_BCD(LEVEL_CTR.CV) ;
END_ACTION

```

Figure F.7 - Body of program GRAVEL in textual SFC representation using ST language elements (continued)

```

CONFIGURATION GRAVEL_CONTROL
RESOURCE PROC1 ON PROC_TYPE_Y
PROGRAM G : GRAVEL
(* Inputs *)
(OFF_PB      := %I0.0 ,
ON_PB       := %I0.1 ,
FILL_PB     := %I0.2 ,
SIREN_ACK   := %I0.3 ,
LOAD_PB     := %I0.4 ,
JOG_PB      := %I0.5 ,
LAMP_TEST   := %I0.7 ,
TRUCK_ON_RAMP := %I1.4 ,
SILO_EMPTY_LS := %I1.5 ,
BIN_EMPTY_LS := %I1.6 ,
SETPOINT    := %IB2 ,
(* Outputs *)
CONTROL_LAMP => %Q4.0 ,
TRUCK_LAMP   => %Q4.2 ,
SILO_EMPTY_LAMP => %Q4.3 ,
CONVEYOR_LAMP => %Q5.3 ,
CONVEYOR_MOTOR => %Q5.4 ,
SILO_VALVE   => %Q5.5 ,
BIN_VALVE    => %Q5.6 ,
SIREN        => %Q5.7 ,
BIN_LEVEL    => %B6) ;
END_RESOURCE
END_CONFIGURATION

```

Figure F.8 - Example configuration for program GRAVEL

F.8 Program AGV

As illustrated in figure F.9, a program is to be devised to control an automatic guided vehicle (AGV). The AGV is to travel between two extreme positions, left (indicated by limit switch S3) and right (indicated by limit switch S4). The normal position of the AGV is on the left.

The AGV is to execute one cycle of left-to-right and return motion when the operator actuates pushbutton S1, and two cycles when the operator actuates pushbutton S2. It is also possible to pass from a single to a double cycle by actuating pushbutton S2 during a single cycle. Finally, non-repeat locking is to be provided if either S1 or S2 remains actuated.

Figure F.10 illustrates the graphical declaration of program AGV, while figure F.11 shows a typical configuration for this program. Figure F.12 shows the AGV program body, consisting of a main control sequence and a single-cycle control sequence.

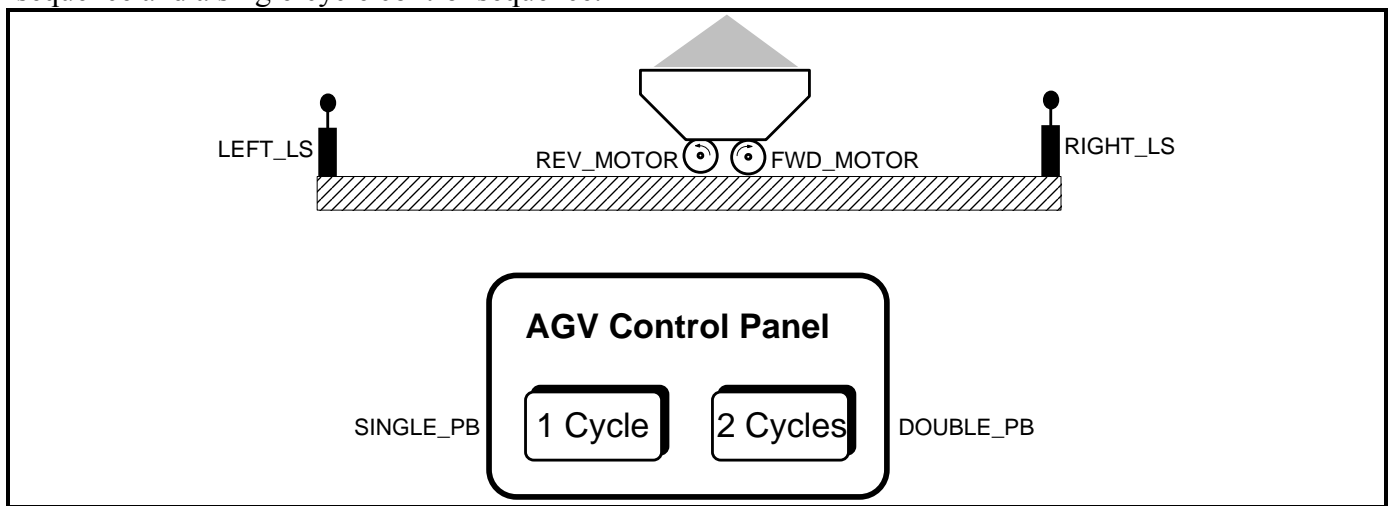


Figure F.9 - Physical model for program AGV

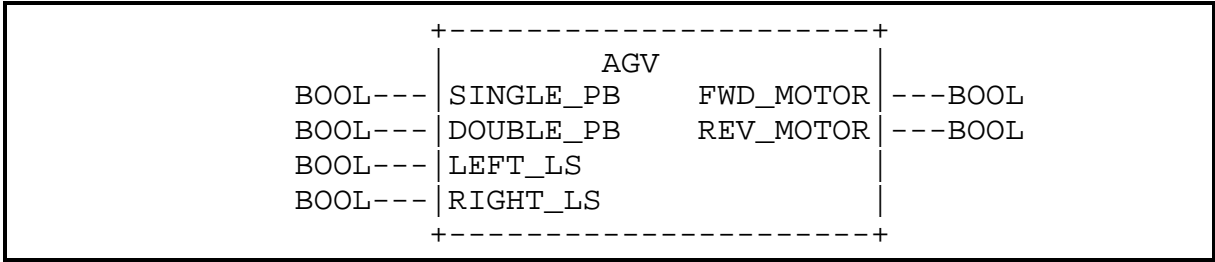


Figure F.10 - Graphical declaration of program AGV

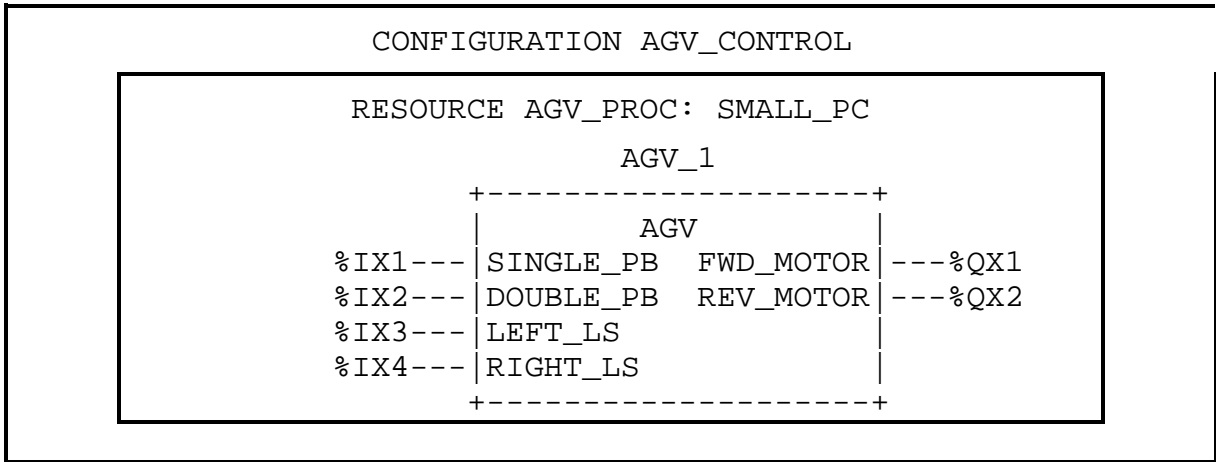


Figure F.11 - A graphical configuration of program AGV

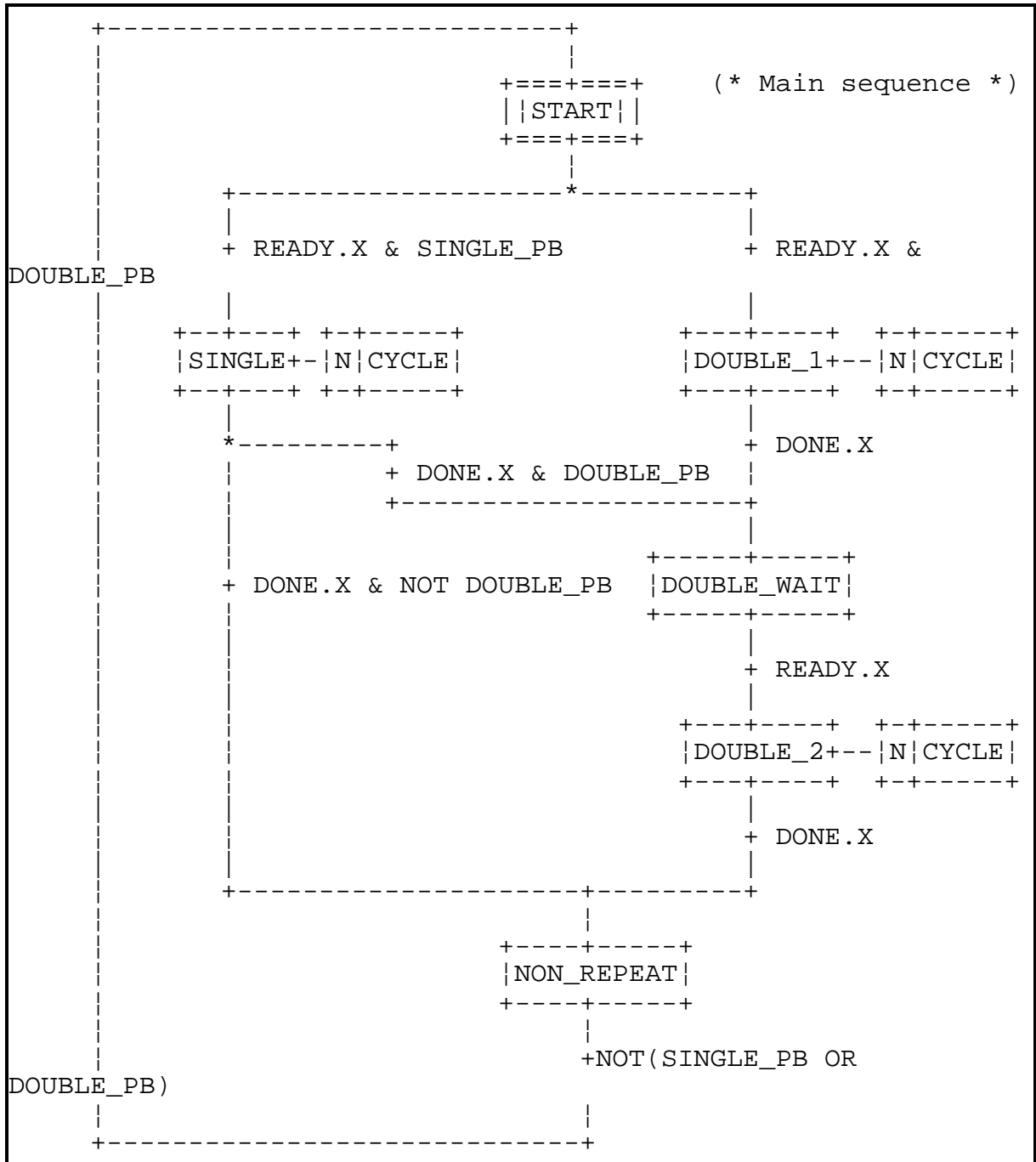


Figure F.12 - Body of program AGV
(continued on following page)

```

*)
+-----+
|
+====+====+      (* Perform a single cycle
|
| |READY| |
+====+====+
|
+ CYCLE
|
+-----+-----+ +-----+
| FORWARD+- |N| FWD_MOTOR |
+-----+-----+ +-----+
|
+ RIGHT_LS
|
+-----+-----+ +-----+
| REVERSE+- |N| REV_MOTOR |
+-----+-----+ +-----+
|
+ LEFT_LS
|
+---+---+
| DONE |
+---+---+
|
+ NOT CYCLE
|
+-----+

```

Figure F.12 - Body of program AGV (continued)