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" *)
```

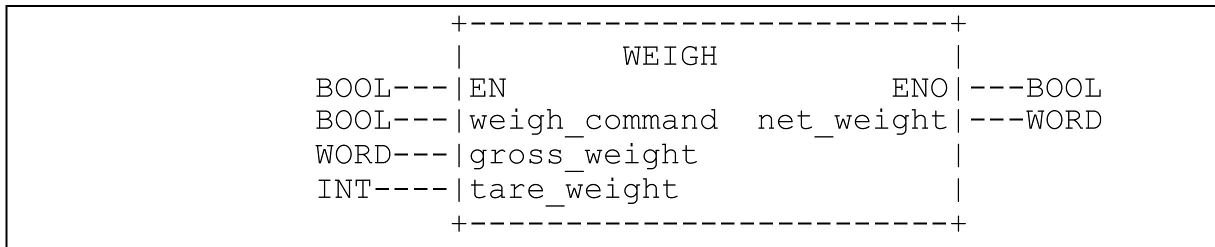
The body of function WEIGH in the IL language is:

```
            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 *)
```

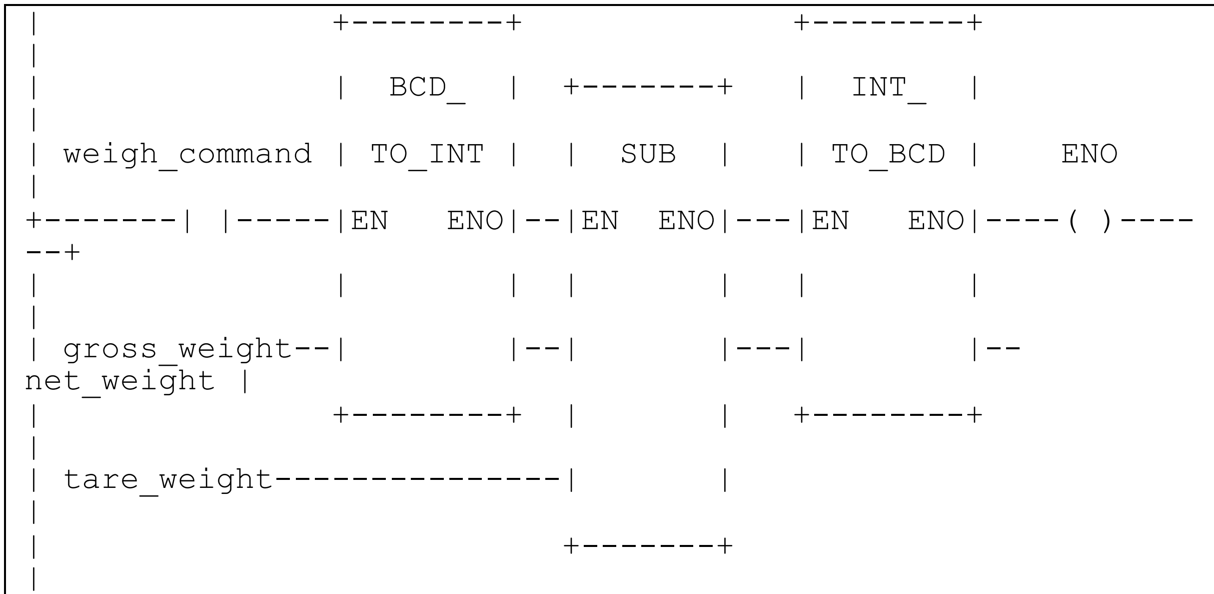
The body of function WEIGH in the ST language is:

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

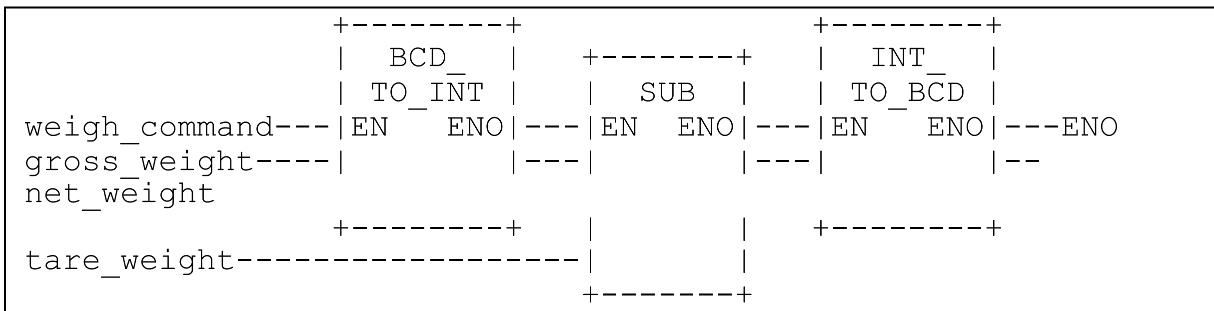
An equivalent graphical declaration of function WEIGH is:



The function body in the LD language is:



The function body in the FBD language is:



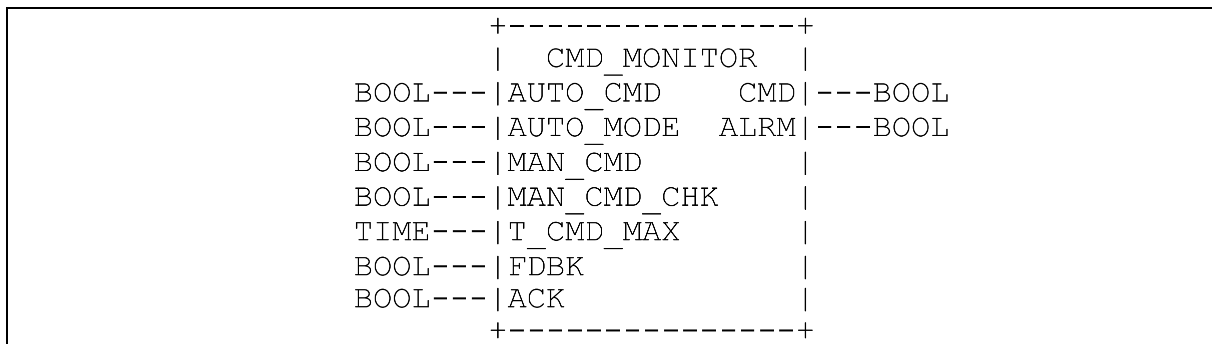
F.2 Function block CMD_MONITOR

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:



```
              +---------------+
              |  CMD_MONITOR  |
       BOOL---|AUTO_CMD    CMD|---BOOL
       BOOL---|AUTO_MODE  ALRM|---BOOL
       BOOL---|MAN_CMD        |
       BOOL---|MAN_CMD_CHK    |
       TIME---|T_CMD_MAX      |
       BOOL---|FDBK           |
       BOOL---|ACK            |
              +---------------+
```

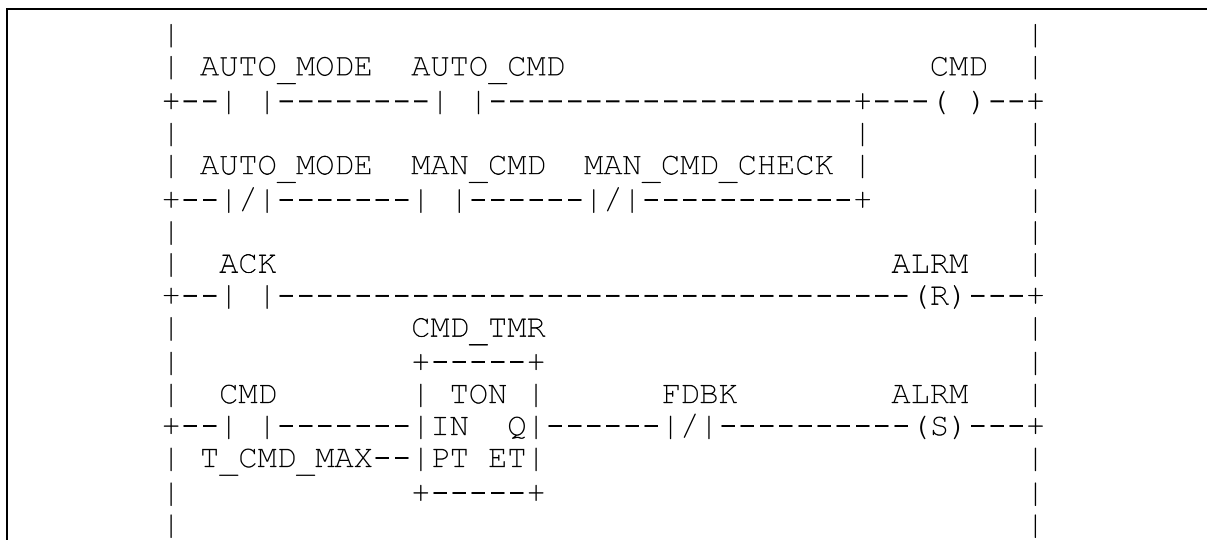
The body of function block CMD_MONITOR in the ST language is:

```
   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:

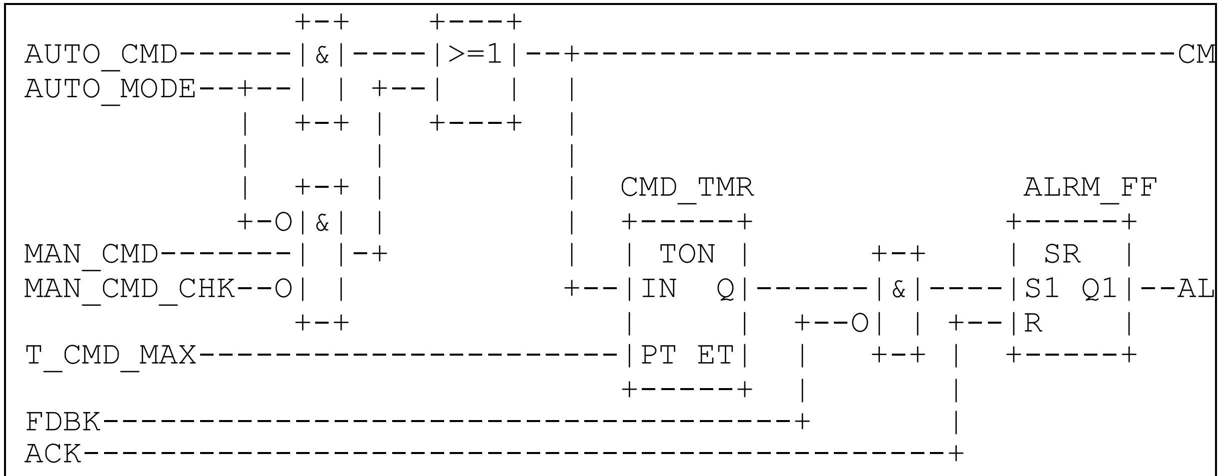
```
LD      T_CMD_MAX
ST      CMD_TMR.PT      (* Store an input to the TON FB *)
LD      AUTO_CMD
AND     AUTO_MODE
OR(     MAN_CMD
ANDN    AUTO_MODE
ANDN    MAN_CMD_CHECK
)
ST      CMD
IN      CMD_TMR         (* Invoke the TON FB *)
LD      CMD_TMR.Q
ANDN    FDBK
ST      ALRM_FF.S1      (* Store an 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:



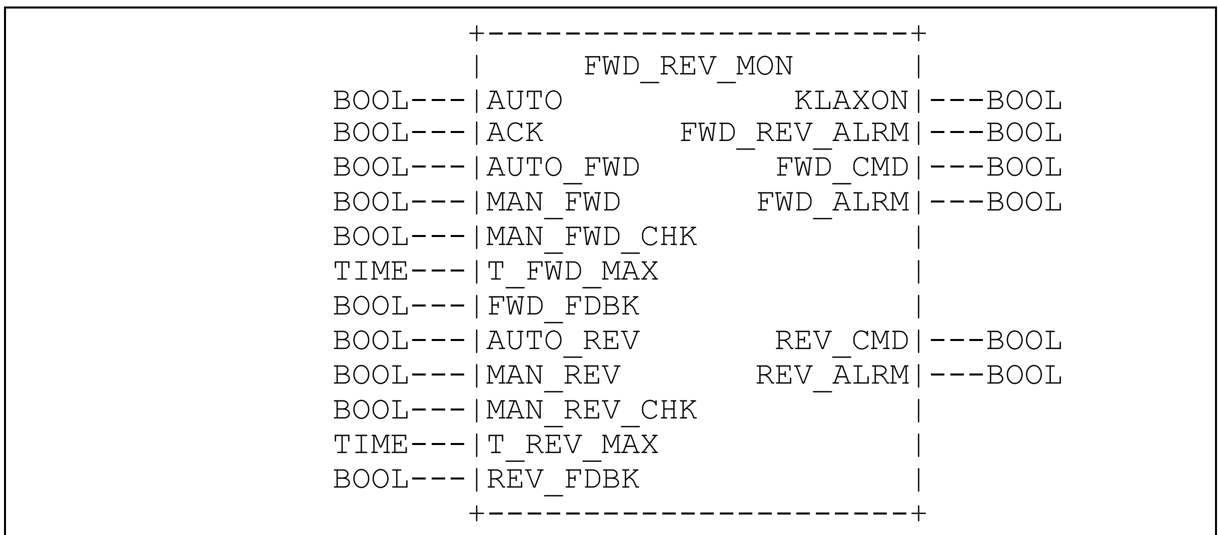
```
|                                                      |
| AUTO_MODE  AUTO_CMD                          CMD     |
+--| |--------| |-------------------+---( )--+
|                                   |                  |
| AUTO_MODE  MAN_CMD  MAN_CMD_CHECK |                  |
+--|/|-------| |------|/|-----------+                  |
|                                                      |
|  ACK                                        ALRM     |
+--| |----------------------------------(R)---+
|             CMD_TMR                                  |
|             +-----+                                  |
|  CMD        | TON |      FDBK           ALRM         |
+--| |--------|IN  Q|------|/|----------(S)---+
| T_CMD_MAX--|PT ET|                                   |
|             +-----+                                  |
|                                                      |
```

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 *
END_VAR                 (*    by operative unit *)
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
```

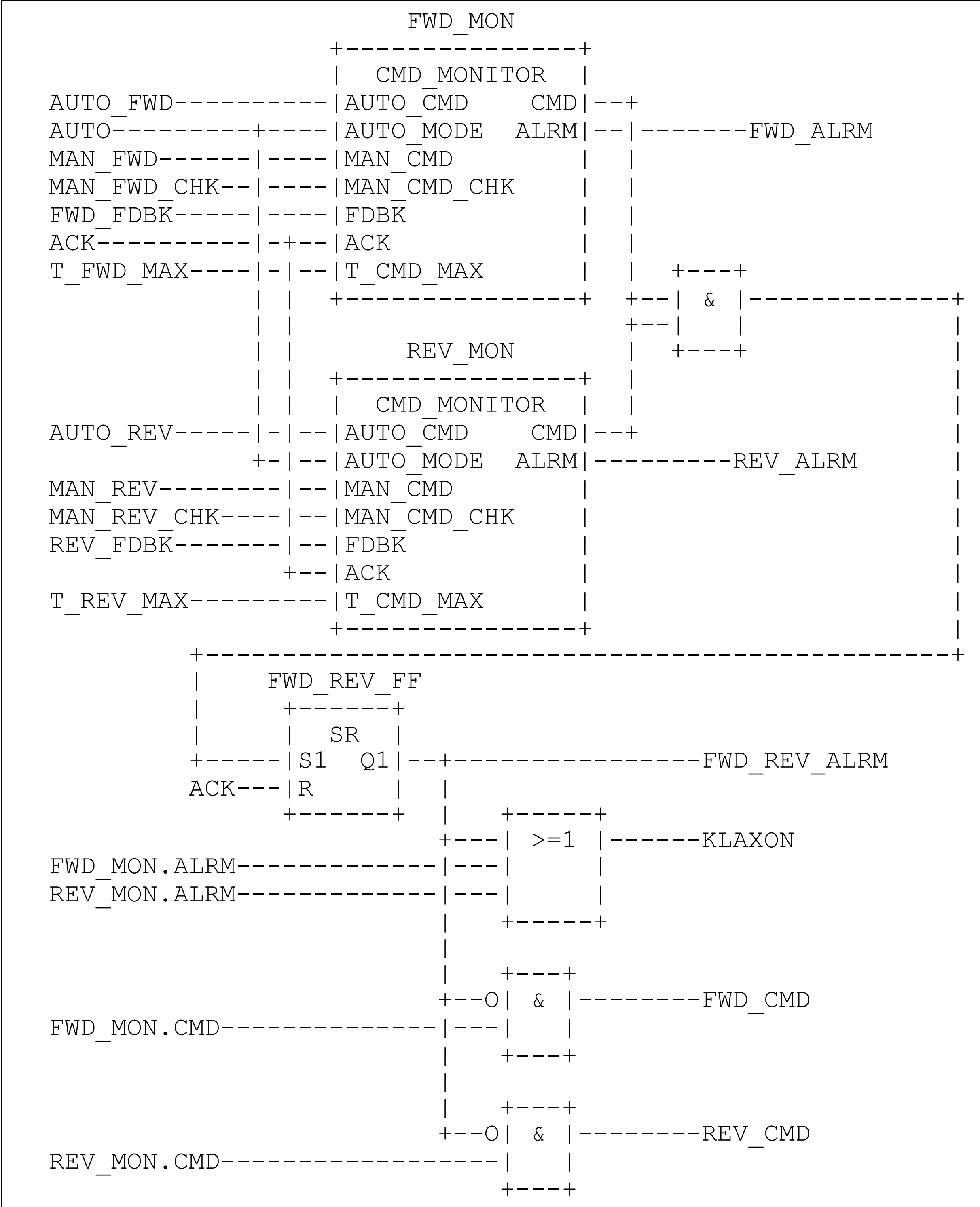
The body of function block FWD_REV_MON can be written in the ST language as:

```
(* 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_ALRM := FWD_REV_FF.Q1;
  FWD_CMD := FWD_MON.CMD & NOT FWD_REV_ALRM;
  FWD_ALRM := FWD_MON.ALRM;
  REV_CMD := REV_MON.CMD & NOT FWD_REV_ALRM;
  REV_ALRM := REV_MON.ALRM;
  KLAXON := FWD_ALRM OR REV_ALRM OR FWD_REV_ALRM;
```

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

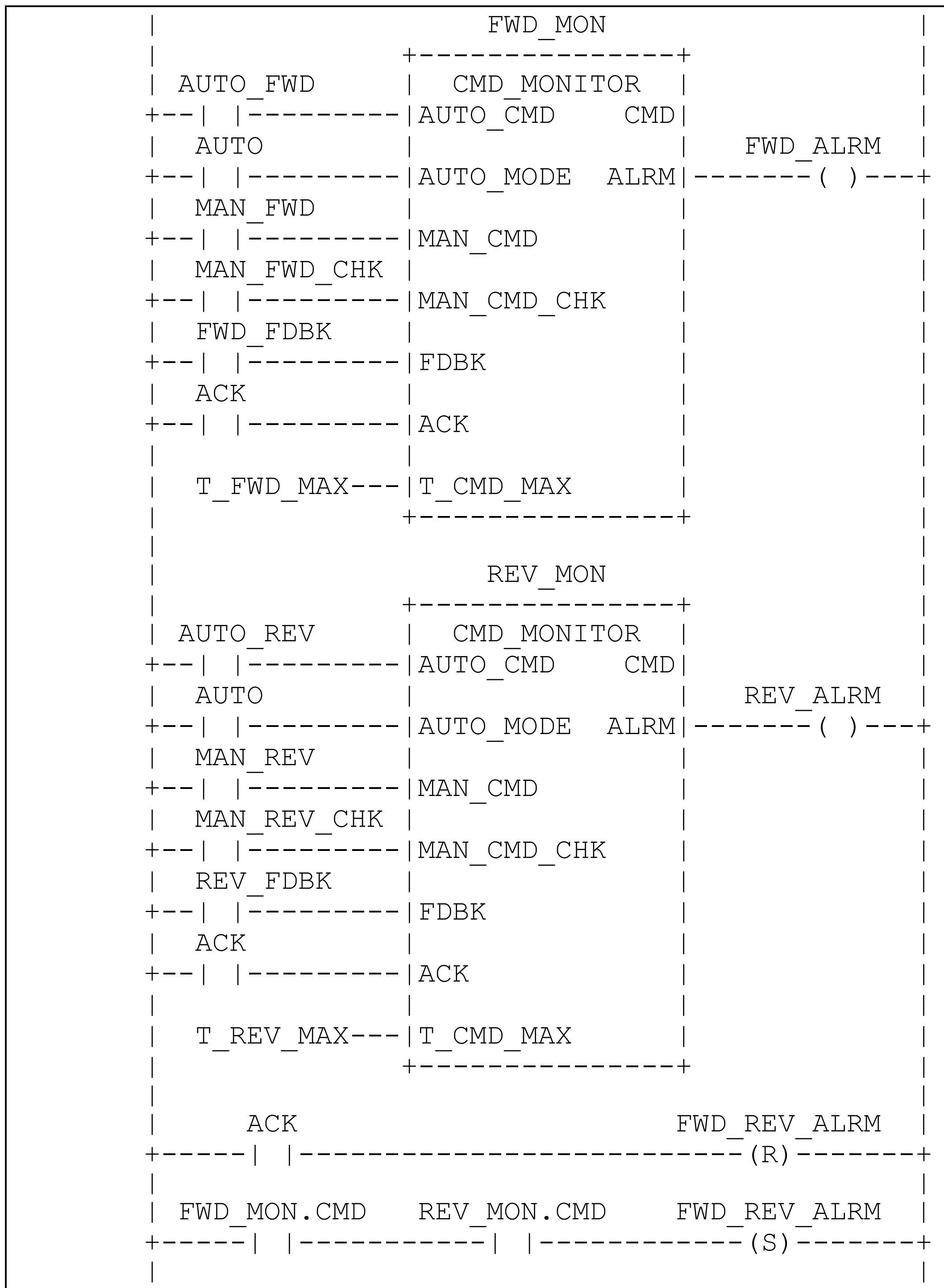
```
LD    AUTO                        (* Load common inputs *)
ST    FWD_MON.AUTO_MODE
ST    REV_MON.AUTO_MODE
LD    ACK
ST    FWD_MON.ACK
ST    REV_MON.ACK
ST    FWD_REV_FF.R
LD    AUTO_FWD                    (* Load inputs to FWD_MON *)
ST    FWD_MON.AUTO_CMD
LD    MAN_FWD
ST    FWD_MON.MAN_CMD
LD    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_ALRM                (* Contention alarm *)
LD    FWD_MON.CMD                 (* "Forward" command and
                                  alarm *)
AND   FWD_REV_ALRM
N
ST    FWD_CMD
LD    FWD_MON.ALRM
ST    FWD_ALRM
LD    REV_MON.CMD                 (* "Reverse" command and
                                  alarm *)
AND   FWD_REV_ALRM
N
ST    REV_CMD
LD    REV_MON.ALRM
ST    REV_ALRM
OR    FWD_ALRM                    (* OR all alarms *)
OR    FWD_REV_ALRM
ST    KLAXON
```

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



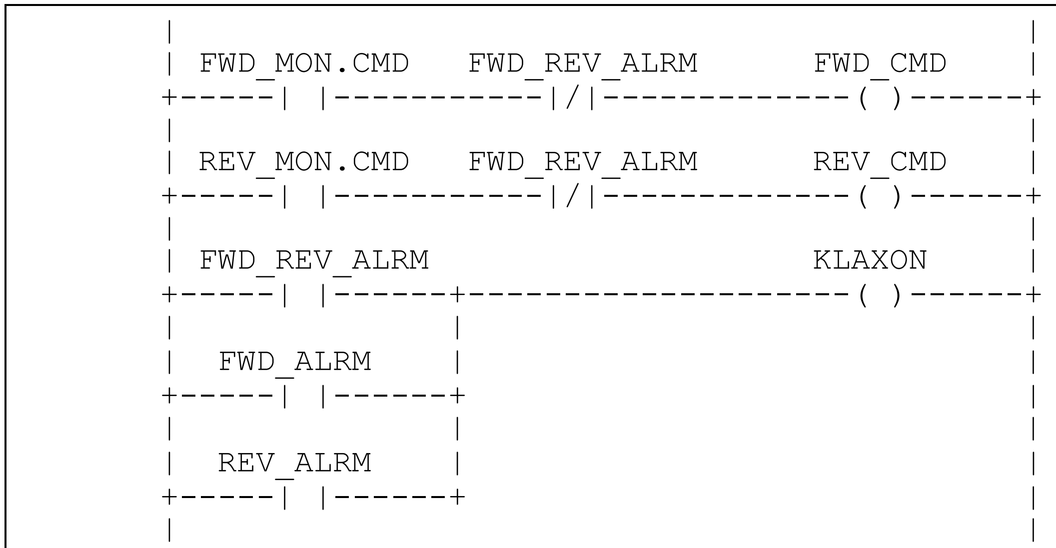
```
                       FWD_MON
                  +---------------+
                  |  CMD_MONITOR  |
AUTO_FWD----------|AUTO_CMD    CMD|--+
AUTO---------+----|AUTO_MODE  ALRM|--|-------FWD_ALRM
MAN_FWD------|----|MAN_CMD        |  |
MAN_FWD_CHK--|----|MAN_CMD_CHK    |  |
FWD_FDBK-----|----|FDBK           |  |
ACK----------|-+--|ACK            |  |
T_FWD_MAX----|-|--|T_CMD_MAX      |  |  +---+
             | |  +---------------+  +--| & |--------------+
             | |                     +--|   |              |
             | |       REV_MON       |  +---+              |
             | |  +---------------+  |                     |
             | |  |  CMD_MONITOR  |  |                     |
AUTO_REV-----|-|--|AUTO_CMD    CMD|--+                     |
             +-|--|AUTO_MODE  ALRM|---------REV_ALRM       |
MAN_REV--------|--|MAN_CMD        |                        |
MAN_REV_CHK----|--|MAN_CMD_CHK    |                        |
REV_FDBK-------|--|FDBK           |                        |
               +--|ACK            |                        |
T_REV_MAX---------|T_CMD_MAX      |                        |
                  +---------------+                        |
         +-------------------------------------------------+
         |    FWD_REV_FF
         |     +------+
         |     |  SR  |
         +-----|S1  Q1|--+----------------FWD_REV_ALRM
         ACK---|R     |  |
               +------+  |   +-----+
                         +---| >=1 |------KLAXON
FWD_MON.ALRM-------------|---|     |
REV_MON.ALRM-------------|---|     |
                         |   +-----+
                         |
                         |   +---+
                         +--O| & |--------FWD_CMD
FWD_MON.CMD--------------|---|   |
                         |   +---+
                         |
                         |   +---+
                         +--O| & |--------REV_CMD
REV_MON.CMD------------------|   |
                             +---+
```

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
```

A graphical declaration of function block STACK_INT is:

```
                             +-----------+
                             | STACK_INT |
                     BOOL--->PUSH  EMPTY|---BOOL
                     BOOL--->POP    OFLO|---BOOL
                     BOOL---|R1      OUT|---INT
                     INT----|IN         |
                     INT----|N          |
                             +-----------+

            (* Internal variable declarations *)
  VAR STK : ARRAY[0..127] OF INT ; (* Internal Stack *)
      NI : INT :=128  ;            (* Storage for N upon
Reset *)
      PTR : INT := -1 ;            (* Stack Pointer *)
  END_VAR
```

The function block body in the ST language is:

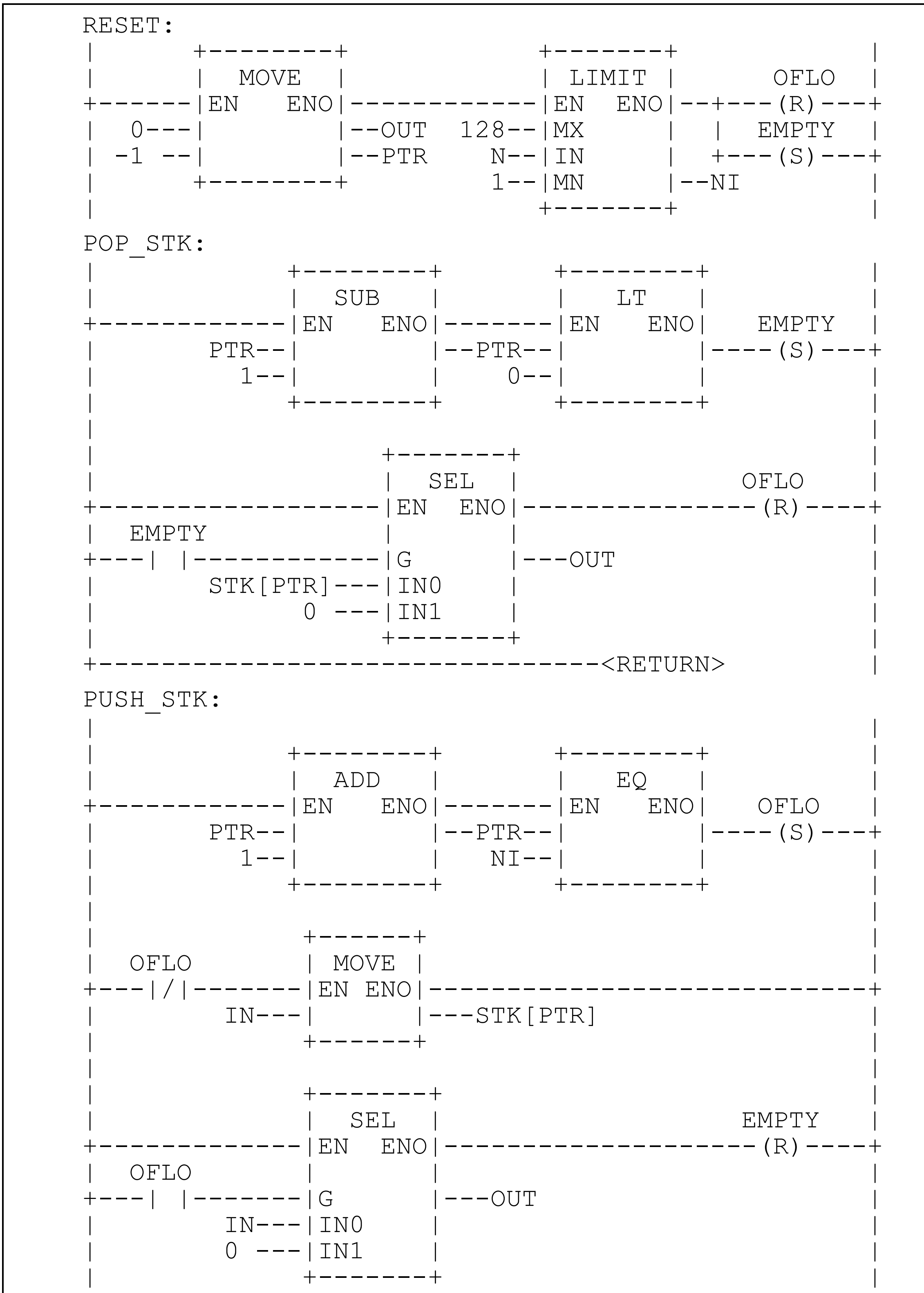
```
    IF R1 THEN
       OFLO := 0; EMPTY := 1; PTR := -1;
       NI := LIMIT (MN:=1,IN:=N,MX:=128); OUT := 0;
    ELSIF POP & NOT EMPTY THEN
       OFLO := 0; PTR := PTR-1; EMPTY := PTR < 0;
       IF EMPTY THEN OUT := 0;
       ELSE OUT := STK[PTR];
       END_IF ;
    ELSIF PUSH & NOT OFLO THEN
       EMPTY := 0; PTR := PTR+1; OFLO := (PTR = NI);
       IF NOT OFLO THEN OUT := IN ; STK[PTR] := IN;
       ELSE OUT := 0;
       END_IF ;
    END_IF ;
```

The function block body in the LD language is:

```
     |                                                  |
     |    R1                                            |
     +---| |--->>RESET                                  |
     |                                                  |
     |  POP   EMPTY                                     |
     +--| |---|/|--->>POP_STK                           |
     |                                                  |
     |  PUSH  OFLO                                      |
     +--| |---|/|--->>PUSH_STK                          |
     |                                                  |
     |                                                  |
     +--------------<RETURN>                            |
```

(continued on following page)

(STACK_INT function block body - LD language - continued)

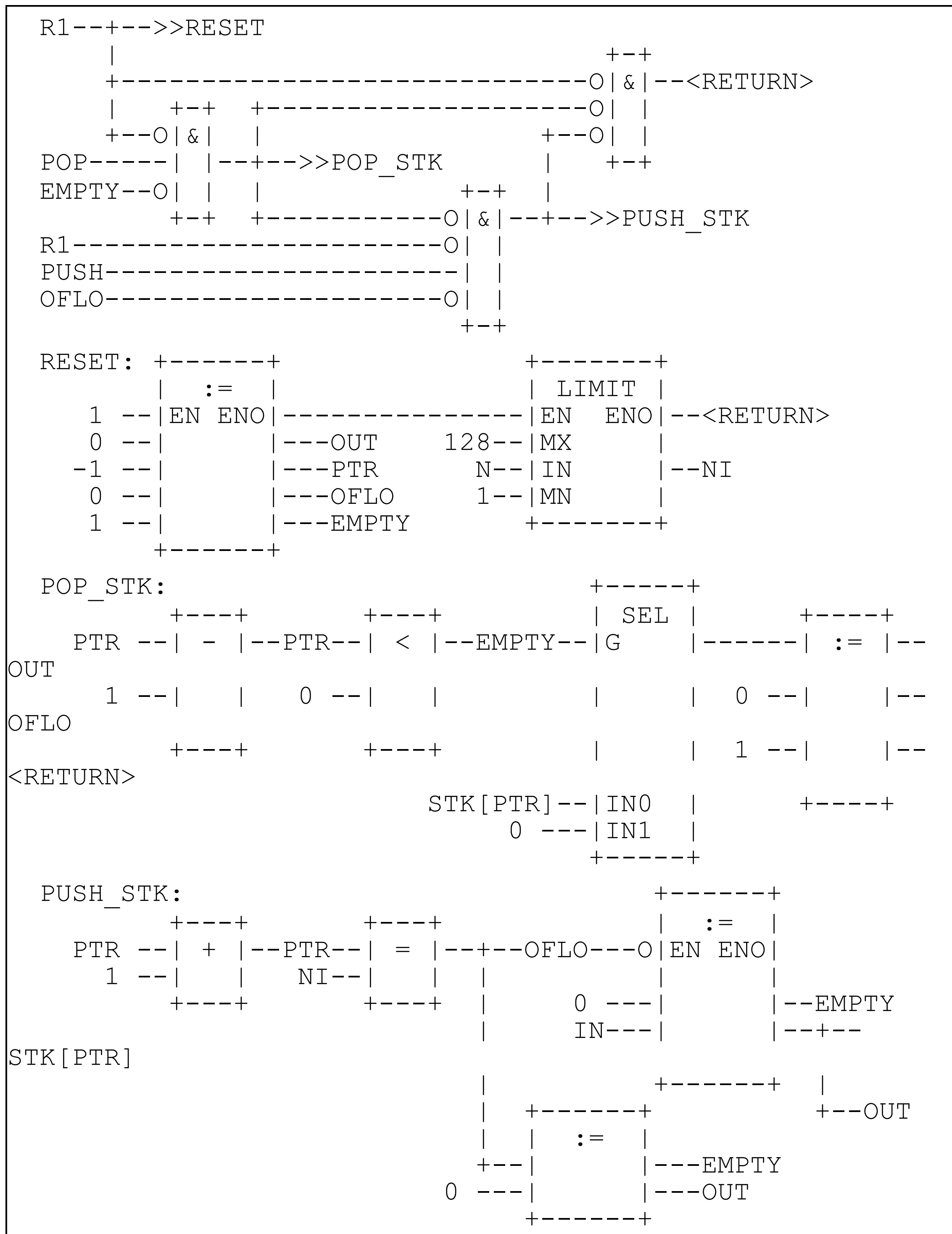


```
RESET:
|      +--------+            +-------+            |
|      |  MOVE  |            | LIMIT |      OFLO  |
+------|EN   ENO|------------|EN  ENO|--+---(R)---+
|  0---|        |--OUT  128--|MX     |  |  EMPTY  |
| -1 --|        |--PTR    N--|IN     |  +---(S)---+
|      +--------+         1--|MN     |--NI        |
|                            +-------+            |
POP_STK:
|            +--------+       +--------+          |
|            |  SUB   |       |   LT   |          |
+------------|EN   ENO|-------|EN   ENO|   EMPTY  |
|       PTR--|        |--PTR--|        |----(S)---+
|         1--|        |    0--|        |          |
|            +--------+       +--------+          |
|                                                 |
|                  +-------+                      |
|                  |  SEL  |              OFLO    |
+------------------|EN  ENO|---------------(R)----+
|  EMPTY           |       |                      |
+---| |------------|G      |---OUT                |
|       STK[PTR]---|IN0    |                      |
|             0 ---|IN1    |                      |
|                  +-------+                      |
+--------------------------------<RETURN>         |
PUSH_STK:
|                                                 |
|            +--------+       +--------+          |
|            |  ADD   |       |   EQ   |          |
+------------|EN   ENO|-------|EN   ENO|   OFLO   |
|       PTR--|        |--PTR--|        |----(S)---+
|         1--|        |   NI--|        |          |
|            +--------+       +--------+          |
|                                                 |
|             +------+                            |
|  OFLO       | MOVE |                            |
+---|/|-------|EN ENO|----------------------------+
|        IN---|      |---STK[PTR]                 |
|             +------+                            |
|                                                 |
|             +-------+                           |
|             |  SEL  |                 EMPTY    |
+-------------|EN  ENO|--------------------(R)----+
|  OFLO       |       |                           |
+---| |-------|G      |---OUT                     |
|        IN---|IN0    |                           |
|        0 ---|IN1    |                           |
|             +-------+                           |
```

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
```

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



```
R1--+-->>RESET
    |                              +-+
    +-----------------------------O|&|--<RETURN>
    |   +-+  +--------------------O| |
    +--O|&|  |                 +--O| |
POP-----| |--+-->>POP_STK      |   +-+
EMPTY--O| |  |             +-+ |
        +-+  +------------O|&|--+-->>PUSH_STK
R1-----------------------O| |
PUSH----------------------| |
OFLO---------------------O| |
                           +-+

RESET: +------+                +-------+
       |  :=  |                | LIMIT |
   1 --|EN ENO|----------------|EN  ENO|--<RETURN>
   0 --|      |---OUT    128--|MX      |
  -1 --|      |---PTR      N--|IN      |--NI
   0 --|      |---OFLO     1--|MN      |
   1 --|      |---EMPTY        +-------+
       +------+

POP_STK:                              +-----+
        +---+       +---+             | SEL |       +----+
   PTR --| - |--PTR--| < |--EMPTY--|G     |------| := |--
OUT
     1 --|   |   0 --|   |         |     |  0 --|    |--
OFLO
        +---+       +---+          |     |  1 --|    |--
<RETURN>
                         STK[PTR]--|IN0  |       +----+
                              0 ---|IN1  |
                                   +-----+

PUSH_STK:                              +------+
        +---+       +---+              |  :=  |
   PTR --| + |--PTR--| = |--+--OFLO---O|EN ENO|
     1 --|   |   NI--|   |  |          |      |
        +---+       +---+  |     0 ---|      |--EMPTY
                           |     IN---|      |--+--
STK[PTR]
                           |          +------+  |
                           |  +------+          +--OUT
                           |  |  :=  |
                           +--|      |---EMPTY
                         0 ---|      |---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.

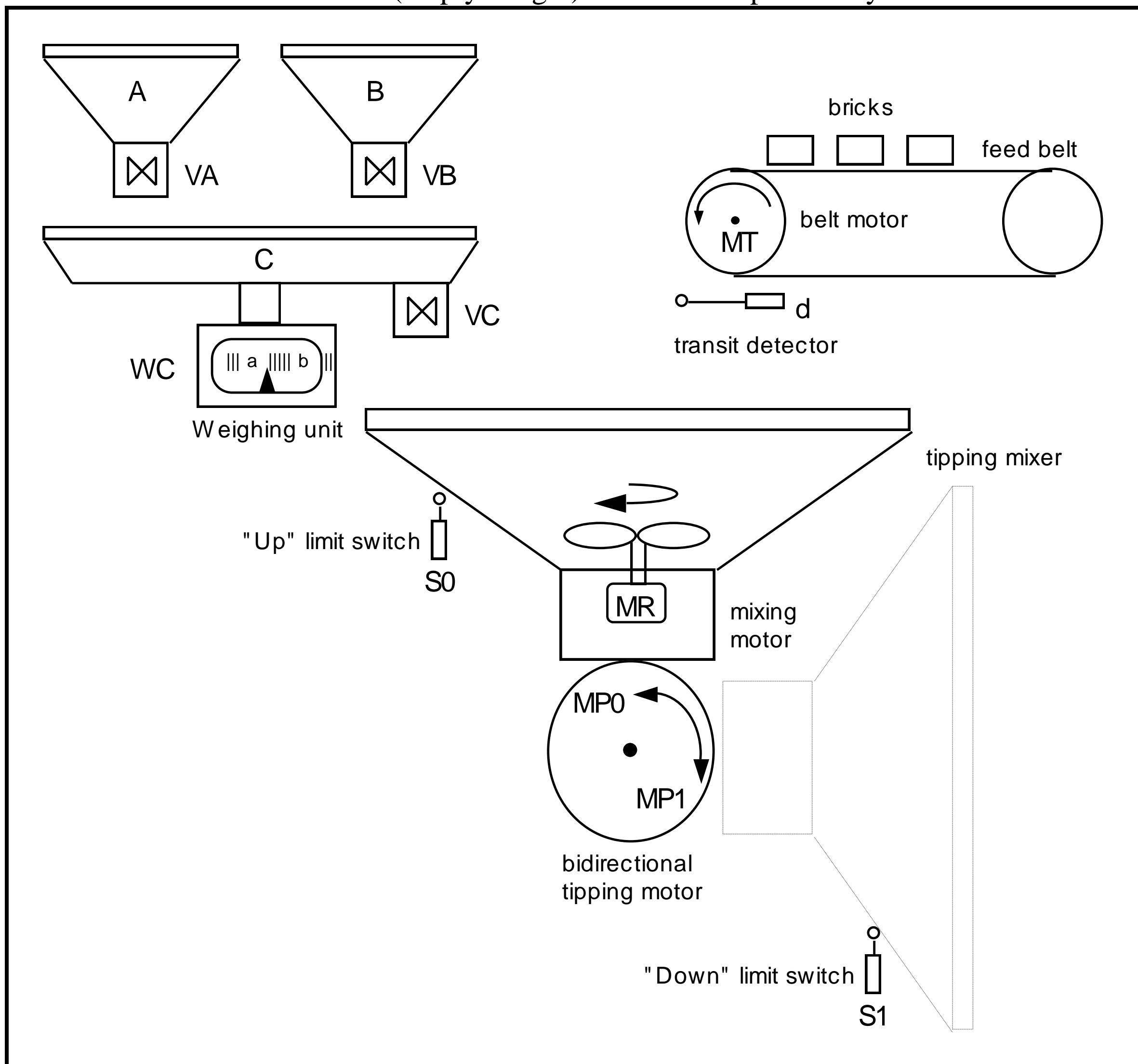
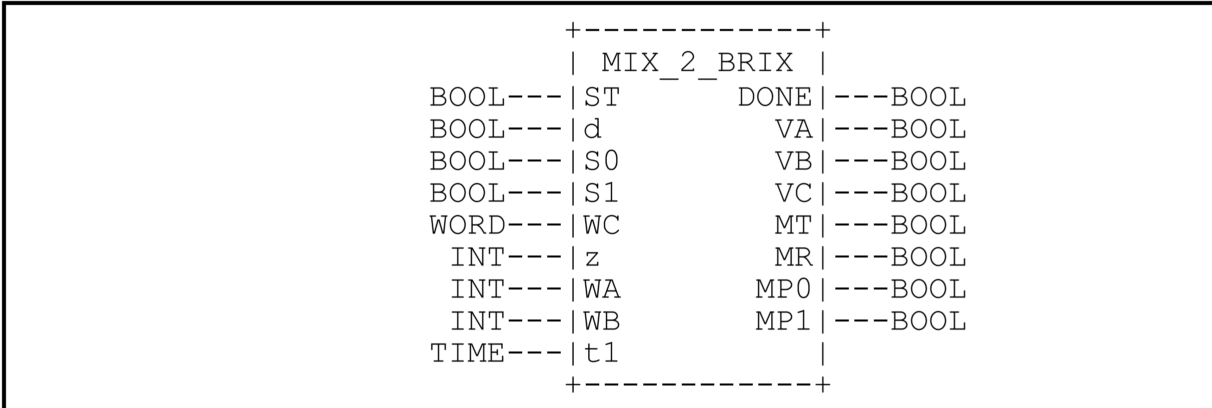


Figure F.1 - Function block MIX_2_BRIX - Physical model

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

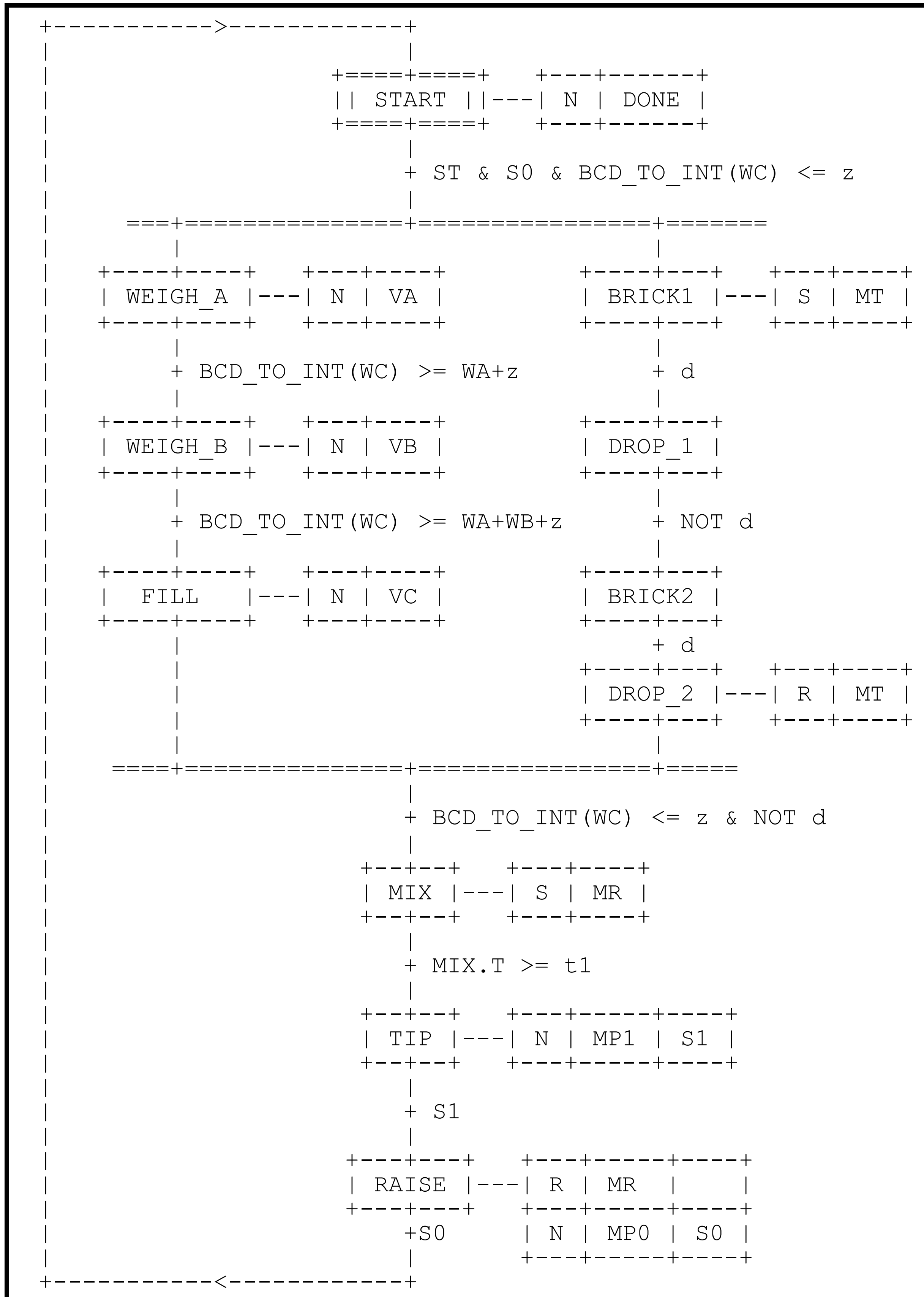
```
FUNCTION_BLOCK MIX_2_BRIX
  VAR_INPUT
    ST : BOOL ;       (* "Start" command *)
    d  : BOOL ;       (* Transit detector *)
    S0 : BOOL ;       (* "Mixer up" limit switch *)
    S1 : BOOL ;       (* "Mixer down" limit switch *)
    WC : WORD;        (* Current scale reading in BCD *)
    z  : 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 ,
    VA   ,        (* Valve "A" : 0 - close, 1 - open *)
    VB   ,        (* Valve "B" : 0 - close, 1 - open *)
    VC   ,        (* 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:



```
              +------------+
              | MIX_2_BRIX |
       BOOL---|ST      DONE|---BOOL
       BOOL---|d         VA|---BOOL
       BOOL---|S0        VB|---BOOL
       BOOL---|S1        VC|---BOOL
       WORD---|WC        MT|---BOOL
        INT---|z         MR|---BOOL
        INT---|WA       MP0|---BOOL
        INT---|WB       MP1|---BOOL
       TIME---|t1          |
              +------------+
```

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



```
+----------->------------+
|                        |
|                   +====+====+   +---+------+
|                   || START ||---| N | DONE |
|                   +====+====+   +---+------+
|                        |
|                        + ST & S0 & BCD_TO_INT(WC) <= z
|                        |
|     ===+===============+================+=======
|        |                                |
|    +----+----+   +---+----+         +----+---+   +---+----+
|    | WEIGH_A |---| N | VA |         | BRICK1 |---| S | MT |
|    +----+----+   +---+----+         +----+---+   +---+----+
|        |                                |
|        + BCD_TO_INT(WC) >= WA+z         + d
|        |                                |
|    +----+----+   +---+----+         +----+---+
|    | WEIGH_B |---| N | VB |         | DROP_1 |
|    +----+----+   +---+----+         +----+---+
|        |                                |
|        + BCD_TO_INT(WC) >= WA+WB+z      + NOT d
|        |                                |
|    +----+----+   +---+----+         +----+---+
|    |  FILL   |---| N | VC |         | BRICK2 |
|    +----+----+   +---+----+         +----+---+
|        |                                + d
|        |                            +----+---+   +---+----+
|        |                            | DROP_2 |---| R | MT |
|        |                            +----+---+   +---+----+
|        |                                |
|    ====+===============+================+=====
|                        |
|                        + BCD_TO_INT(WC) <= z & NOT d
|                        |
|                     +--+--+   +---+----+
|                     | MIX |---| S | MR |
|                     +--+--+   +---+----+
|                        |
|                        + MIX.T >= t1
|                        |
|                     +--+--+   +---+----+----+
|                     | TIP |---| N | MP1 | S1 |
|                     +--+--+   +---+----+----+
|                        |
|                        + S1
|                        |
|                    +---+---+   +---+-----+----+
|                    | RAISE |---| R | MR  |    |
|                    +---+---+   +---+-----+----+
|                        +S0     | N | MP0 | S0 |
|                        |       +---+-----+----+
+-----------<------------+
```

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
```

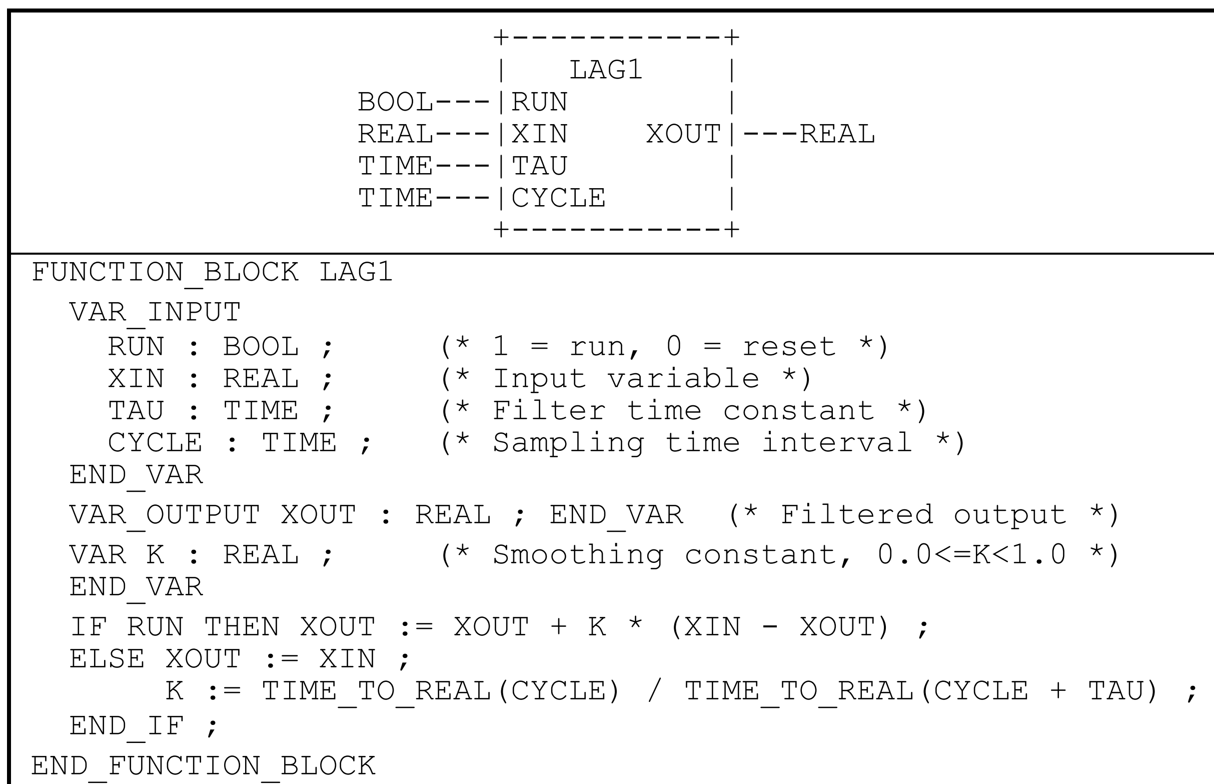
```
   TRANSITION FROM START TO (WEIGH_A, BRICK1)
      := ST & S0 & BCD_TO_INT(WC) <= z;
   END_TRANSITION
   STEP WEIGH_A: VA(N); END_STEP
   TRANSITION FROM WEIGH_A TO WEIGH_B := BCD_TO_INT(WC) >=
WA+z ;
   END_TRANSITION
   STEP WEIGH_B: VB(N); END_STEP
   TRANSITION FROM WEIGH_B TO FILL := BCD_TO_INT(WC) >=
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.

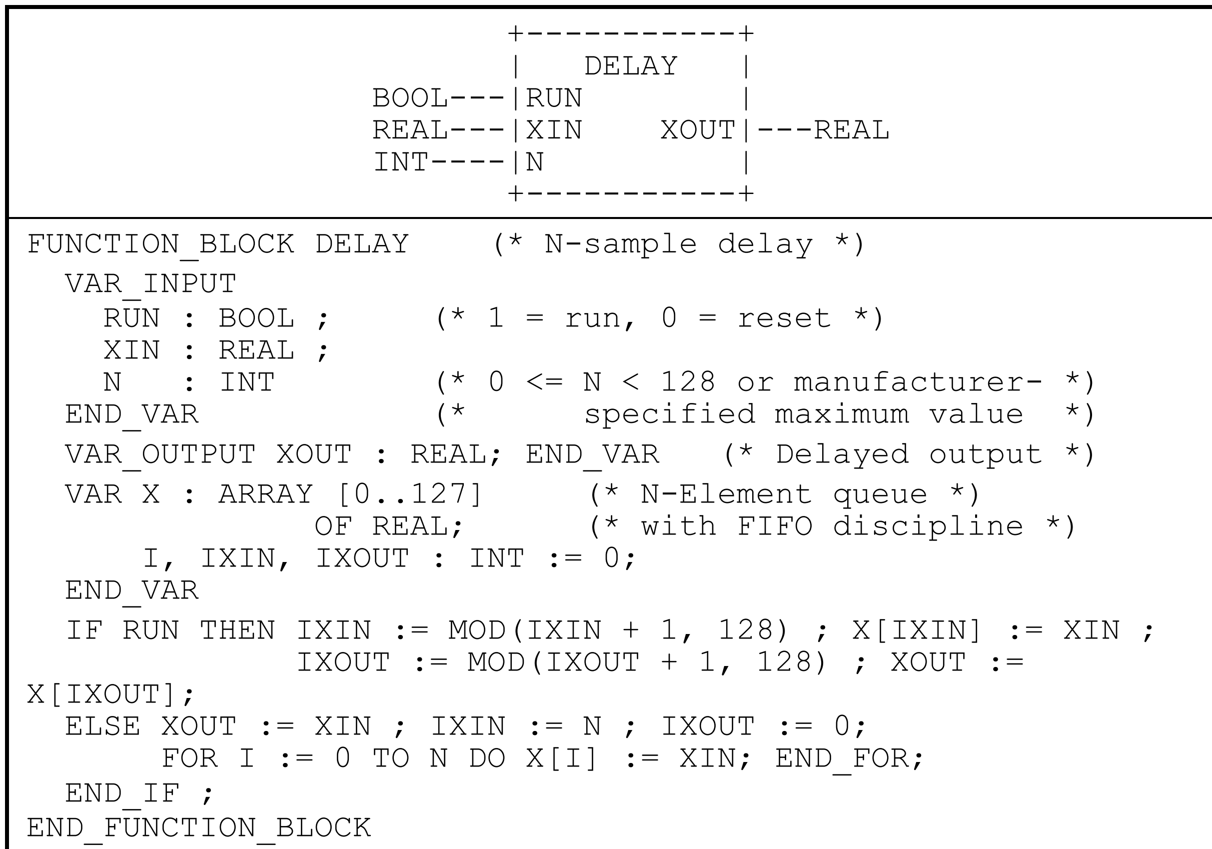


```
                   +-----------+
                   |   LAG1    |
            BOOL---|RUN        |
            REAL---|XIN    XOUT|---REAL
            TIME---|TAU        |
            TIME---|CYCLE      |
                   +-----------+
```

```
FUNCTION_BLOCK LAG1
  VAR_INPUT
    RUN : BOOL ;      (* 1 = run, 0 = reset *)
    XIN : REAL ;      (* Input variable *)
    TAU : TIME ;      (* Filter time constant *)
    CYCLE : TIME ;    (* Sampling time interval *)
  END_VAR
  VAR_OUTPUT XOUT : REAL ; END_VAR  (* Filtered output *)
  VAR K : REAL ;      (* Smoothing constant, 0.0<=K<1.0 *)
  END_VAR
  IF RUN THEN XOUT := XOUT + K * (XIN - XOUT) ;
  ELSE XOUT := XIN ;
       K := TIME_TO_REAL(CYCLE) / TIME_TO_REAL(CYCLE + TAU) ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.2 Function block DELAY

This function block implements an N-sample delay.

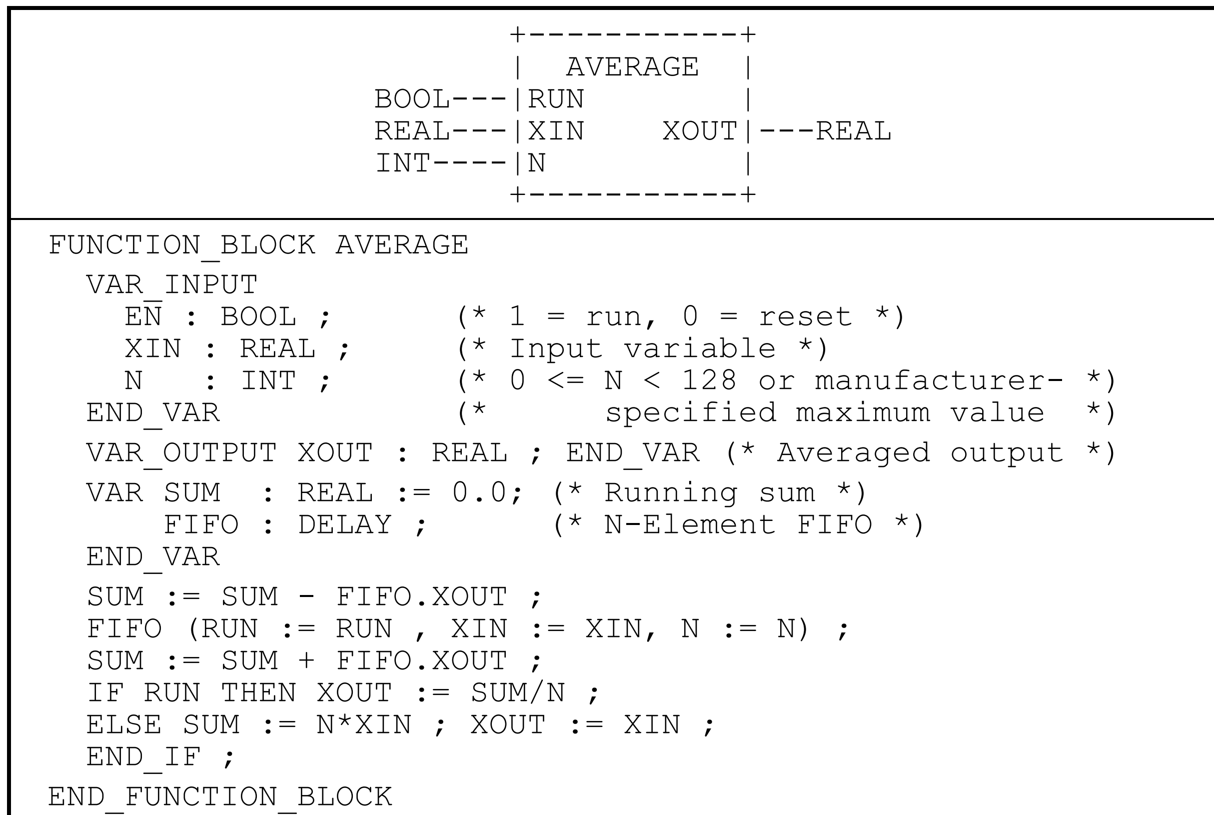


```
                         +-----------+
                         |   DELAY   |
                  BOOL---|RUN        |
                  REAL---|XIN    XOUT|---REAL
                  INT----|N          |
                         +-----------+
```

```
FUNCTION_BLOCK DELAY    (* N-sample delay *)
  VAR_INPUT
    RUN : BOOL ;      (* 1 = run, 0 = reset *)
    XIN : REAL ;
    N   : INT         (* 0 <= N < 128 or manufacturer- *)
  END_VAR             (*      specified maximum value  *)
  VAR_OUTPUT XOUT : REAL; END_VAR   (* Delayed output *)
  VAR X : ARRAY [0..127]     (* N-Element queue *)
               OF REAL;      (* with FIFO discipline *)
      I, IXIN, IXOUT : INT := 0;
  END_VAR
  IF RUN THEN IXIN := MOD(IXIN + 1, 128) ; X[IXIN] := XIN ;
              IXOUT := MOD(IXOUT + 1, 128) ; XOUT :=
X[IXOUT];
  ELSE XOUT := XIN ; IXIN := N ; IXOUT := 0;
       FOR I := 0 TO N DO X[I] := XIN; END_FOR;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.3 Function block AVERAGE

This function block implements a running average over N samples.



```
                  +-----------+
                  |  AVERAGE  |
           BOOL---|RUN        |
           REAL---|XIN    XOUT|---REAL
           INT----|N          |
                  +-----------+
```

```
FUNCTION_BLOCK AVERAGE
  VAR_INPUT
    EN : BOOL ;       (* 1 = run, 0 = reset *)
    XIN : REAL ;      (* Input variable *)
    N   : INT ;       (* 0 <= N < 128 or manufacturer- *)
  END_VAR             (*      specified maximum value  *)
  VAR_OUTPUT XOUT : REAL ; END_VAR (* Averaged output *)
  VAR SUM  : REAL := 0.0; (* Running sum *)
      FIFO : DELAY ;      (* N-Element FIFO *)
  END_VAR
  SUM := SUM - FIFO.XOUT ;
  FIFO (RUN := RUN , XIN := XIN, N := N) ;
  SUM := SUM + FIFO.XOUT ;
  IF RUN THEN XOUT := SUM/N ;
  ELSE SUM := N*XIN ; XOUT := XIN ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.4 Function block INTEGRAL

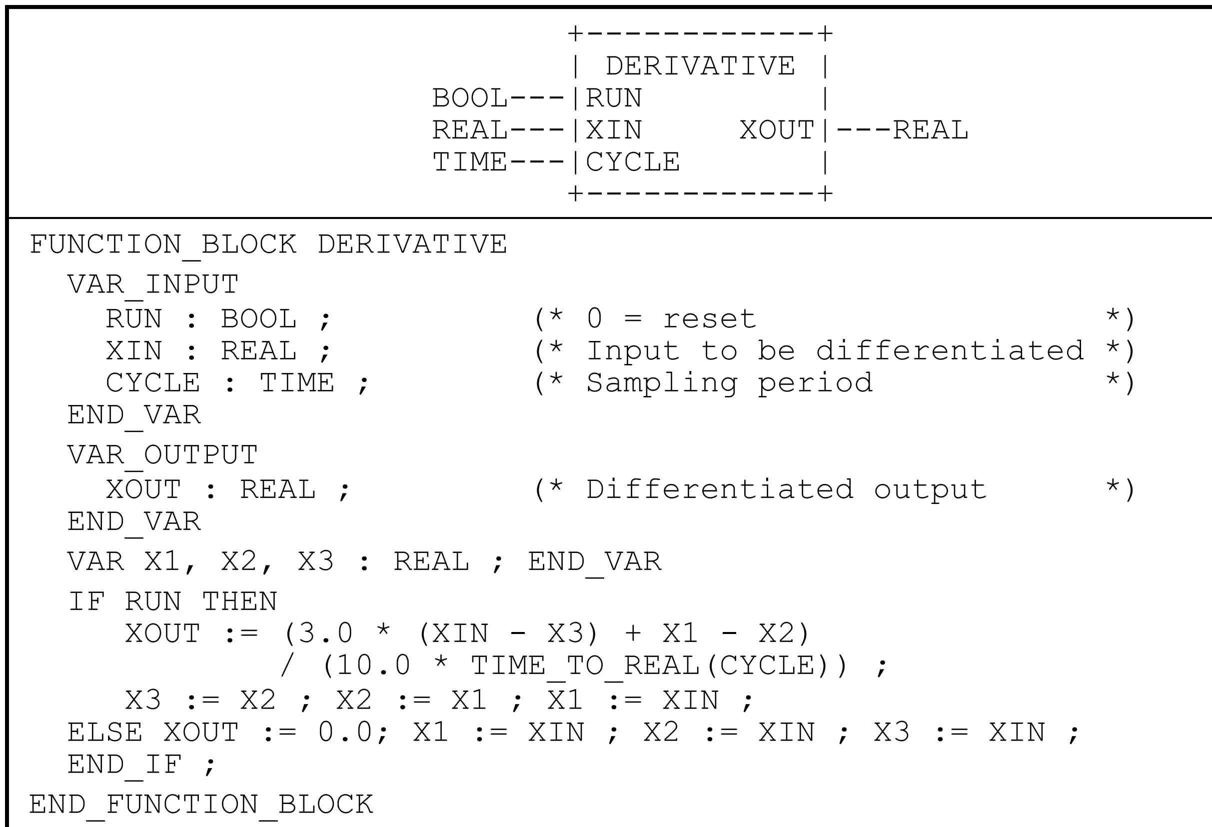
This function block implements integration over time.

```
                 +-----------+
                 |  INTEGRAL |
         BOOL---|RUN        Q|---BOOL
         BOOL---|R1          |
         REAL---|XIN     XOUT|---REAL
         REAL---|X0          |
         TIME---|CYCLE       |
                 +-----------+
```

```
FUNCTION_BLOCK INTEGRAL
  VAR_INPUT
    RUN : BOOL ;        (* 1 = integrate, 0 = hold *)
    R1 : BOOL ;         (* Overriding reset        *)
    XIN : REAL ;        (* Input variable          *)
    X0  : REAL ;        (* Initial value           *)
    CYCLE : TIME ;      (* Sampling period         *)
  END_VAR
  VAR_OUTPUT
    Q : BOOL ;          (* NOT R1                  *)
    XOUT : REAL ;       (* Integrated output       *)
  END_VAR
  Q := NOT R1 ;
  IF R1 THEN XOUT := X0 ;
  ELSIF RUN THEN XOUT := XOUT + XIN * TIME_TO_REAL(CYCLE);
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.5 Function block DERIVATIVE

This function block implements differentiation with respect to time.

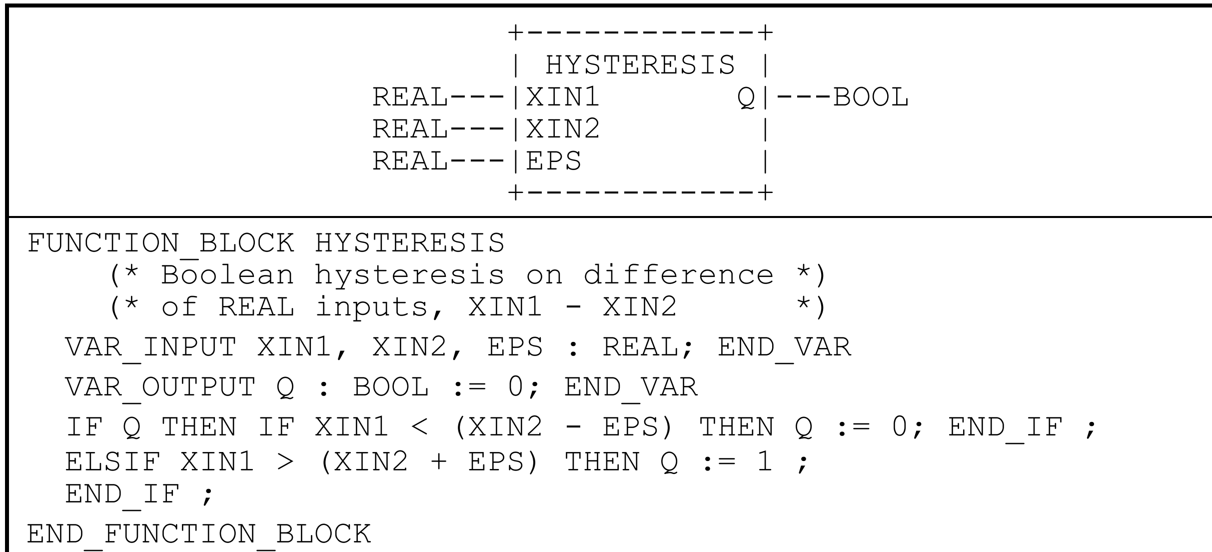


```
              +------------+
              | DERIVATIVE |
      BOOL---|RUN          |
      REAL---|XIN      XOUT|---REAL
      TIME---|CYCLE        |
              +------------+
```

```
FUNCTION_BLOCK DERIVATIVE
  VAR_INPUT
    RUN : BOOL ;          (* 0 = reset                    *)
    XIN : REAL ;          (* Input to be differentiated *)
    CYCLE : TIME ;        (* Sampling period              *)
  END_VAR
  VAR_OUTPUT
    XOUT : REAL ;         (* Differentiated output       *)
  END_VAR
  VAR X1, X2, X3 : REAL ; END_VAR
  IF RUN THEN
     XOUT := (3.0 * (XIN - X3) + X1 - X2)
             / (10.0 * TIME_TO_REAL(CYCLE)) ;
     X3 := X2 ; X2 := X1 ; X1 := XIN ;
  ELSE XOUT := 0.0; X1 := XIN ; X2 := XIN ; X3 := XIN ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.6 Function block HYSTERESIS

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



```
              +------------+
              | HYSTERESIS |
      REAL---|XIN1        Q|---BOOL
      REAL---|XIN2         |
      REAL---|EPS          |
              +------------+
```

```
FUNCTION_BLOCK HYSTERESIS
    (* Boolean hysteresis on difference *)
    (* of REAL inputs, XIN1 - XIN2      *)
  VAR_INPUT XIN1, XIN2, EPS : REAL; END_VAR
  VAR_OUTPUT Q : BOOL := 0; END_VAR
  IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0; END_IF ;
  ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1 ;
  END_IF ;
END_FUNCTION_BLOCK
```

F.6.7 Function block LIMITS_ALARM

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

```
                                +---------+
                                | LIMITS_ |
                                |  ALARM  |
   (* High limit     *) REAL--|H      QH|--BOOL (* High flag
*)
   (* Variable value *) REAL--|X       Q|--BOOL (* Alarm
output *)
   (* Lower limit    *) REAL--|L      QL|--BOOL (* Low flag
*)
   (* Hysteresis     *) REAL--|EPS      |
                                +---------+
```

```
                 (* Function block body in FBD language *)
                                   HIGH_ALARM
                                 +------------+
                                 | HYSTERESIS |
   X------------------------+--|XIN1       Q|--+------------
QH
                  +---+    |   |            |   |
   H---------------| - |------|XIN2         |   |
              +---|   |    |   |            |   |
              |   +---+    |   |            |   |
              +--------------|EPS          |   |  +-----+
        +---+  |           |   +------------+  +--| >=1 |
   EPS---| / |--+           |                      |     |---Q
   2.0---|   |  |           |    LOW_ALARM     +--|     |
        +---+  |           |   +------------+  |  +-----+
              |   +---+    |   | HYSTERESIS |  |
   L---------------| + |------|XIN1       Q|--+------------
QL
              |   |   |    |   |            |
              +---|   |    +--|XIN2         |
              |   +---+        |            |
              +--------------|EPS          |
                                 +------------+
```

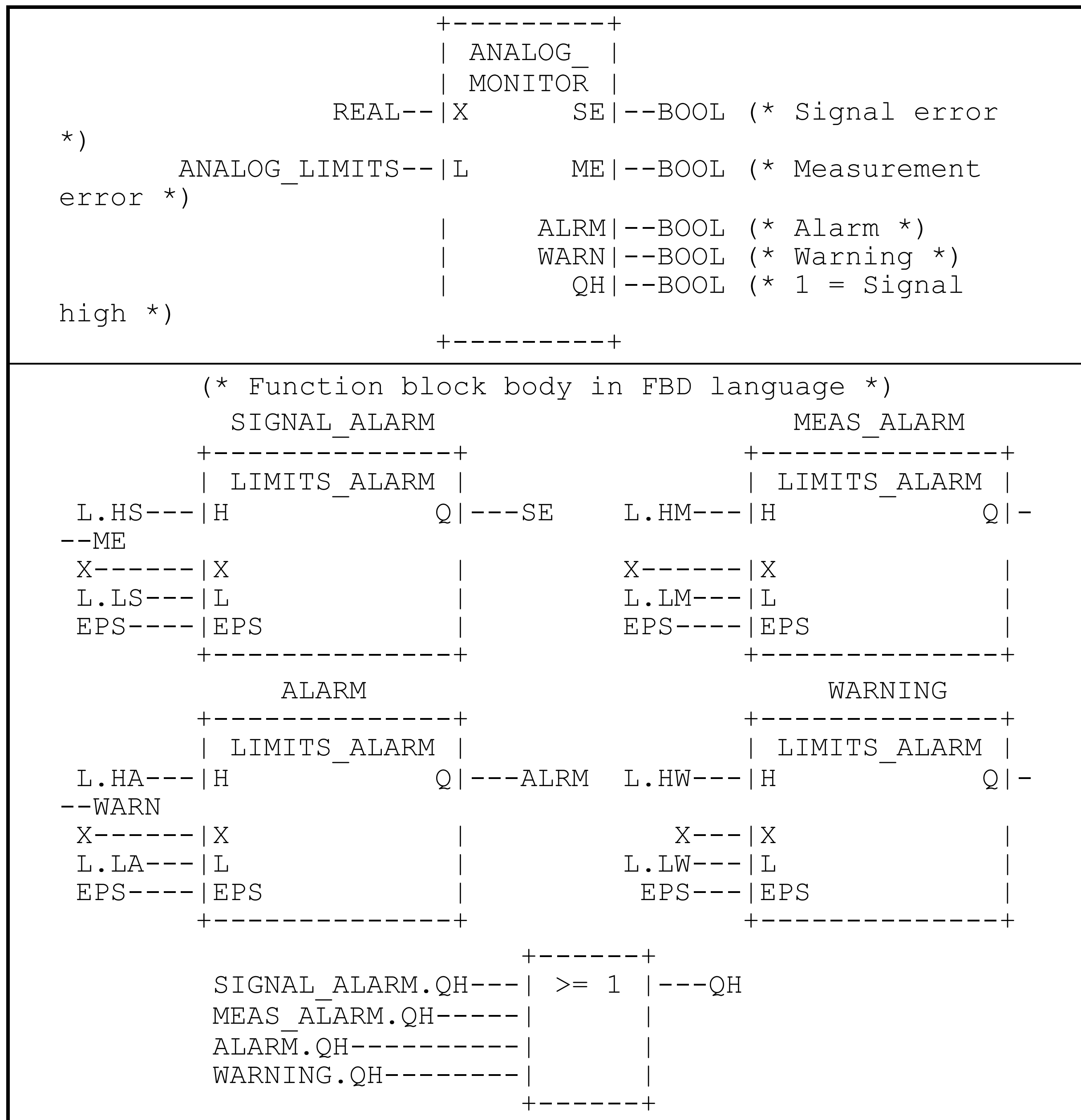
F.6.8 Structure ANALOG_LIMITS

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

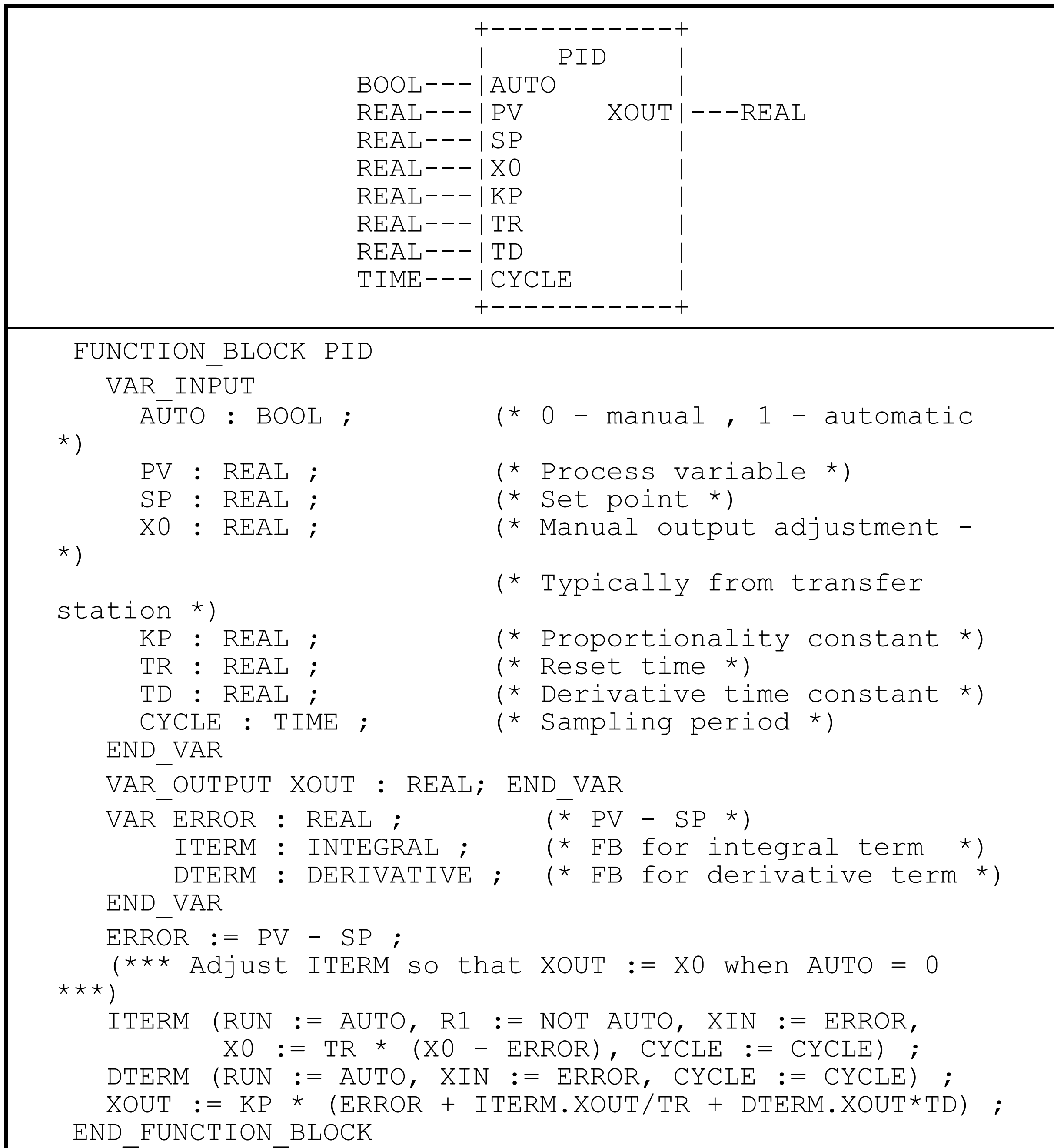
This function block implements analog signal monitoring.



```
                         +---------+
                         | ANALOG_ |
                         | MONITOR |
                   REAL--|X      SE|--BOOL (* Signal error
*)
       ANALOG_LIMITS--|L      ME|--BOOL (* Measurement
error *)
                         |     ALRM|--BOOL (* Alarm *)
                         |     WARN|--BOOL (* Warning *)
                         |       QH|--BOOL (* 1 = Signal
high *)
                         +---------+
        (* Function block body in FBD language *)
          SIGNAL_ALARM                          MEAS_ALARM
        +--------------+                +--------------+
        | LIMITS_ALARM |                | LIMITS_ALARM |
 L.HS---|H            Q|---SE    L.HM---|H            Q|-
--ME
 X------|X             |         X------|X             |
 L.LS---|L             |         L.LM---|L             |
 EPS----|EPS           |         EPS----|EPS           |
        +--------------+                +--------------+
             ALARM                          WARNING
        +--------------+                +--------------+
        | LIMITS_ALARM |                | LIMITS_ALARM |
 L.HA---|H            Q|---ALRM  L.HW---|H            Q|-
--WARN
 X------|X             |            X---|X             |
 L.LA---|L             |         L.LW---|L             |
 EPS----|EPS           |          EPS---|EPS           |
        +--------------+                +--------------+
                              +------+
         SIGNAL_ALARM.QH---| >= 1 |---QH
         MEAS_ALARM.QH-----|      |
         ALARM.QH----------|      |
         WARNING.QH--------|      |
                              +------+
```

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.

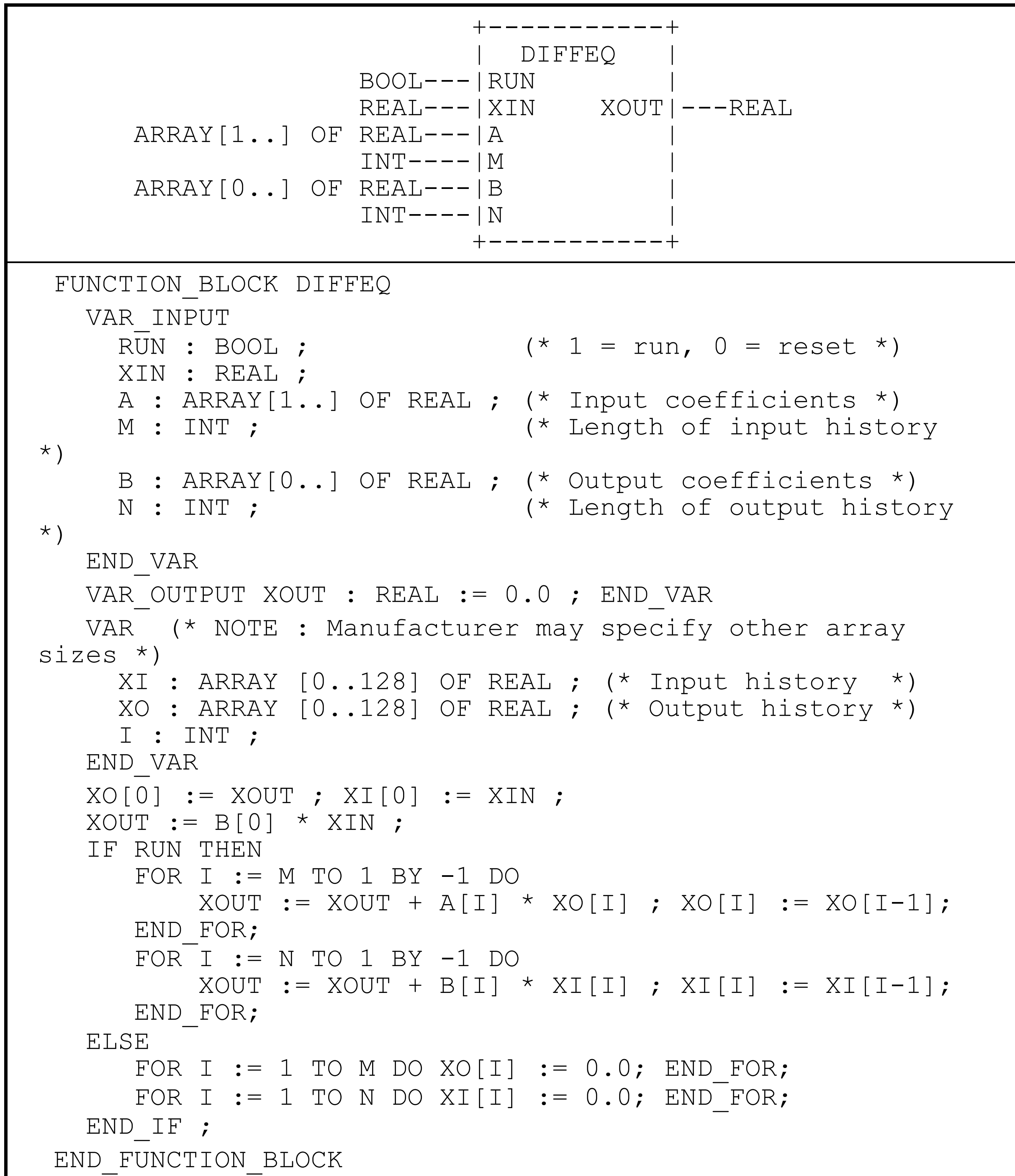


```
                  +-----------+
                  |    PID    |
          BOOL---|AUTO        |
          REAL---|PV      XOUT|---REAL
          REAL---|SP          |
          REAL---|X0          |
          REAL---|KP          |
          REAL---|TR          |
          REAL---|TD          |
          TIME---|CYCLE       |
                  +-----------+
```

```
 FUNCTION_BLOCK PID
   VAR_INPUT
     AUTO : BOOL ;         (* 0 - manual , 1 - automatic
*)
     PV : REAL ;           (* Process variable *)
     SP : REAL ;           (* Set point *)
     X0 : REAL ;           (* Manual output adjustment -
*)
                           (* Typically from transfer
station *)
     KP : REAL ;           (* Proportionality constant *)
     TR : REAL ;           (* Reset time *)
     TD : REAL ;           (* Derivative time constant *)
     CYCLE : TIME ;        (* Sampling period *)
   END_VAR
   VAR_OUTPUT XOUT : REAL; END_VAR
   VAR ERROR : REAL ;         (* PV - SP *)
       ITERM : INTEGRAL ;     (* FB for integral term  *)
       DTERM : DERIVATIVE ;   (* FB for derivative term *)
   END_VAR
   ERROR := PV - SP ;
   (*** Adjust ITERM so that XOUT := X0 when AUTO = 0
***)
   ITERM (RUN := AUTO, R1 := NOT AUTO, XIN := ERROR,
          X0 := TR * (X0 - ERROR), CYCLE := CYCLE) ;
   DTERM (RUN := AUTO, XIN := ERROR, CYCLE := CYCLE) ;
   XOUT := KP * (ERROR + ITERM.XOUT/TR + DTERM.XOUT*TD) ;
 END_FUNCTION_BLOCK
```

F.6.11 Function block DIFFEQ

This function block implements a general difference equation.

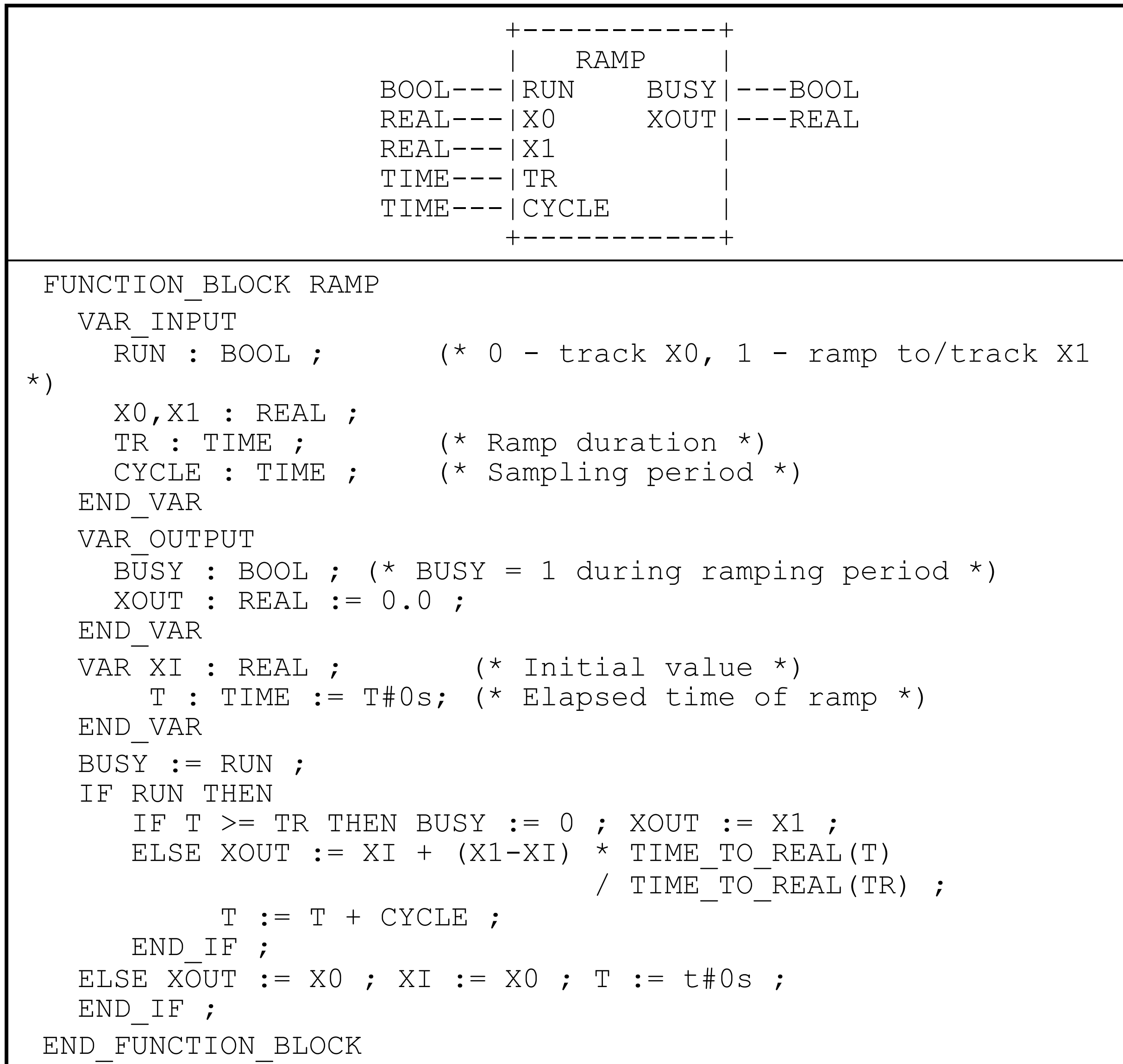


```
                              +-----------+
                              |  DIFFEQ   |
                       BOOL---|RUN        |
                       REAL---|XIN    XOUT|---REAL
      ARRAY[1..] OF REAL---|A          |
                       INT----|M          |
      ARRAY[0..] OF REAL---|B          |
                       INT----|N          |
                              +-----------+
```

```
 FUNCTION_BLOCK DIFFEQ
   VAR_INPUT
     RUN : BOOL ;               (* 1 = run, 0 = reset *)
     XIN : REAL ;
     A : ARRAY[1..] OF REAL ; (* Input coefficients *)
     M : INT ;                  (* Length of input history
*)
     B : ARRAY[0..] OF REAL ; (* Output coefficients *)
     N : INT ;                  (* Length of output history
*)
   END_VAR
   VAR_OUTPUT XOUT : REAL := 0.0 ; END_VAR
   VAR  (* NOTE : Manufacturer may specify other array
sizes *)
     XI : ARRAY [0..128] OF REAL ; (* Input history  *)
     XO : ARRAY [0..128] OF REAL ; (* Output history *)
     I : INT ;
   END_VAR
   XO[0] := XOUT ; XI[0] := XIN ;
   XOUT := B[0] * XIN ;
   IF RUN THEN
      FOR I := M TO 1 BY -1 DO
          XOUT := XOUT + A[I] * XO[I] ; XO[I] := XO[I-1];
      END_FOR;
      FOR I := N TO 1 BY -1 DO
          XOUT := XOUT + B[I] * XI[I] ; XI[I] := XI[I-1];
      END_FOR;
   ELSE
      FOR I := 1 TO M DO XO[I] := 0.0; END_FOR;
      FOR I := 1 TO N DO XI[I] := 0.0; END_FOR;
   END_IF ;
 END_FUNCTION_BLOCK
```

F.6.12 Function block RAMP

This function block implements a time-based ramp.

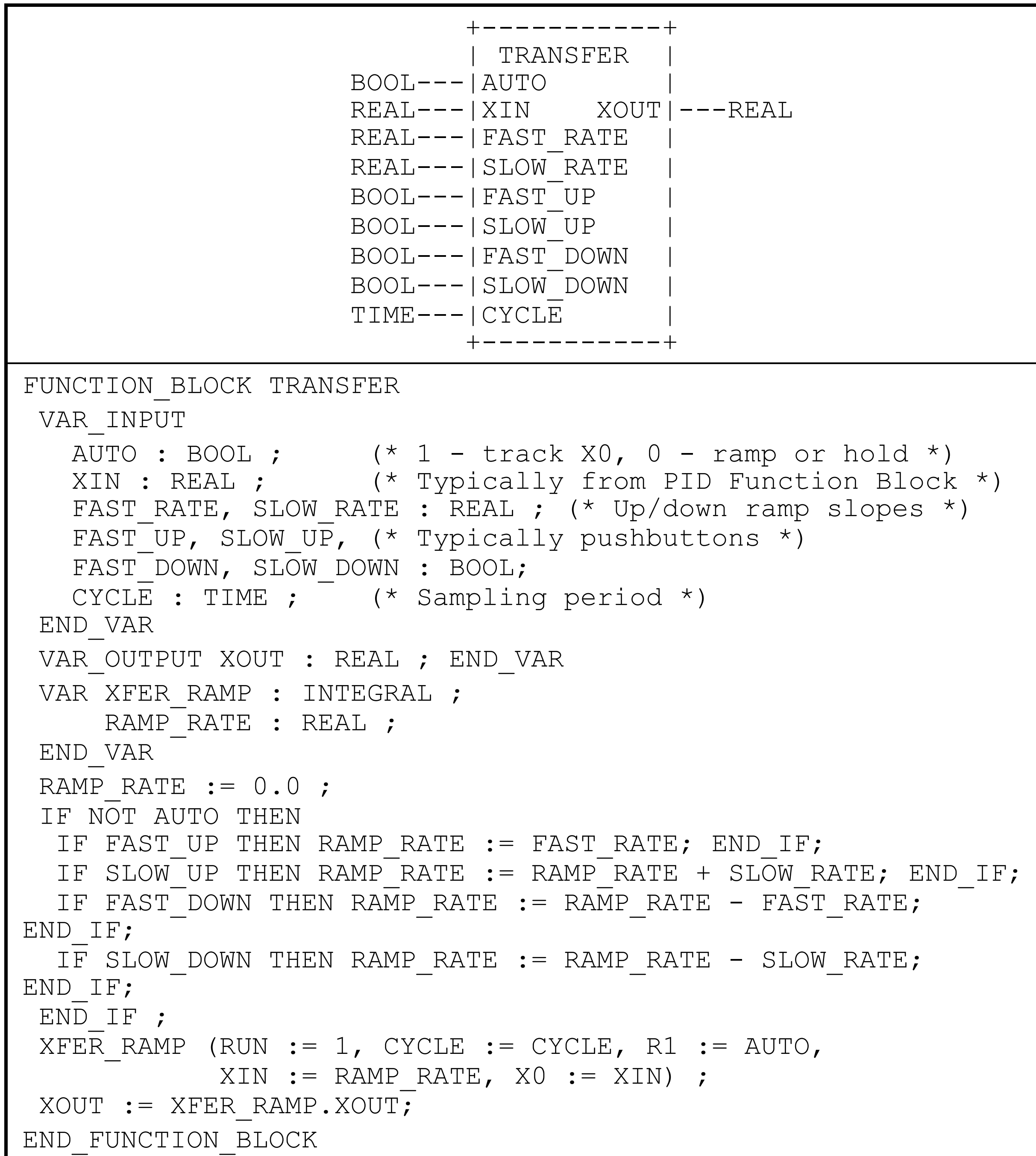


```
                           +-----------+
                           |   RAMP    |
                    BOOL---|RUN    BUSY|---BOOL
                    REAL---|X0     XOUT|---REAL
                    REAL---|X1         |
                    TIME---|TR         |
                    TIME---|CYCLE      |
                           +-----------+
```

```
 FUNCTION_BLOCK RAMP
   VAR_INPUT
     RUN : BOOL ;       (* 0 - track X0, 1 - ramp to/track X1
*)
     X0,X1 : REAL ;
     TR : TIME ;        (* Ramp duration *)
     CYCLE : TIME ;     (* Sampling period *)
   END_VAR
   VAR_OUTPUT
     BUSY : BOOL ; (* BUSY = 1 during ramping period *)
     XOUT : REAL := 0.0 ;
   END_VAR
   VAR XI : REAL ;        (* Initial value *)
       T : TIME := T#0s; (* Elapsed time of ramp *)
   END_VAR
   BUSY := RUN ;
   IF RUN THEN
      IF T >= TR THEN BUSY := 0 ; XOUT := X1 ;
      ELSE XOUT := XI + (X1-XI) * TIME_TO_REAL(T)
                                / TIME_TO_REAL(TR) ;
           T := T + CYCLE ;
      END_IF ;
   ELSE XOUT := X0 ; XI := X0 ; T := t#0s ;
   END_IF ;
 END_FUNCTION_BLOCK
```

F.6.13 Function block TRANSFER

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



```
                  +-----------+
                  | TRANSFER  |
           BOOL---|AUTO       |
           REAL---|XIN    XOUT|---REAL
           REAL---|FAST_RATE  |
           REAL---|SLOW_RATE  |
           BOOL---|FAST_UP    |
           BOOL---|SLOW_UP    |
           BOOL---|FAST_DOWN  |
           BOOL---|SLOW_DOWN  |
           TIME---|CYCLE      |
                  +-----------+
```

```
FUNCTION_BLOCK TRANSFER
 VAR_INPUT
   AUTO : BOOL ;      (* 1 - track X0, 0 - ramp or hold *)
   XIN : REAL ;       (* Typically from PID Function Block *)
   FAST_RATE, SLOW_RATE : REAL ; (* Up/down ramp slopes *)
   FAST_UP, SLOW_UP, (* Typically pushbuttons *)
   FAST_DOWN, SLOW_DOWN : BOOL;
   CYCLE : TIME ;     (* Sampling period *)
 END_VAR
 VAR_OUTPUT XOUT : REAL ; END_VAR
 VAR XFER_RAMP : INTEGRAL ;
     RAMP_RATE : REAL ;
 END_VAR
 RAMP_RATE := 0.0 ;
 IF NOT AUTO THEN
  IF FAST_UP THEN RAMP_RATE := FAST_RATE; END_IF;
  IF SLOW_UP THEN RAMP_RATE := RAMP_RATE + SLOW_RATE; END_IF;
  IF FAST_DOWN THEN RAMP_RATE := RAMP_RATE - FAST_RATE;
END_IF;
  IF SLOW_DOWN THEN RAMP_RATE := RAMP_RATE - SLOW_RATE;
END_IF;
 END_IF ;
 XFER_RAMP (RUN := 1, CYCLE := CYCLE, R1 := AUTO,
            XIN := RAMP_RATE, X0 := XIN) ;
 XOUT := XFER_RAMP.XOUT;
END_FUNCTION_BLOCK
```

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.

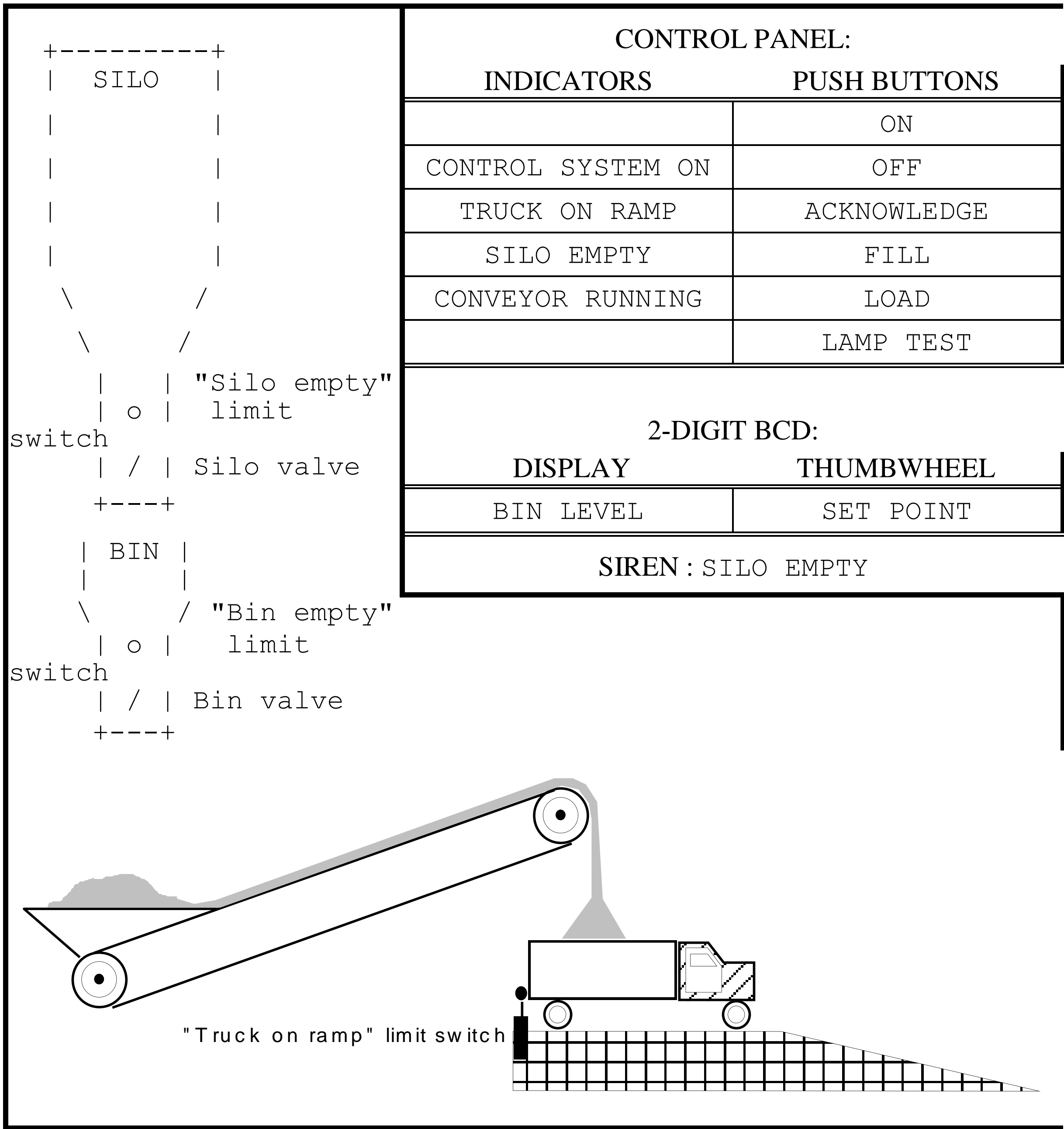


Figure F.2 - Gravel measurement and loading system

```
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

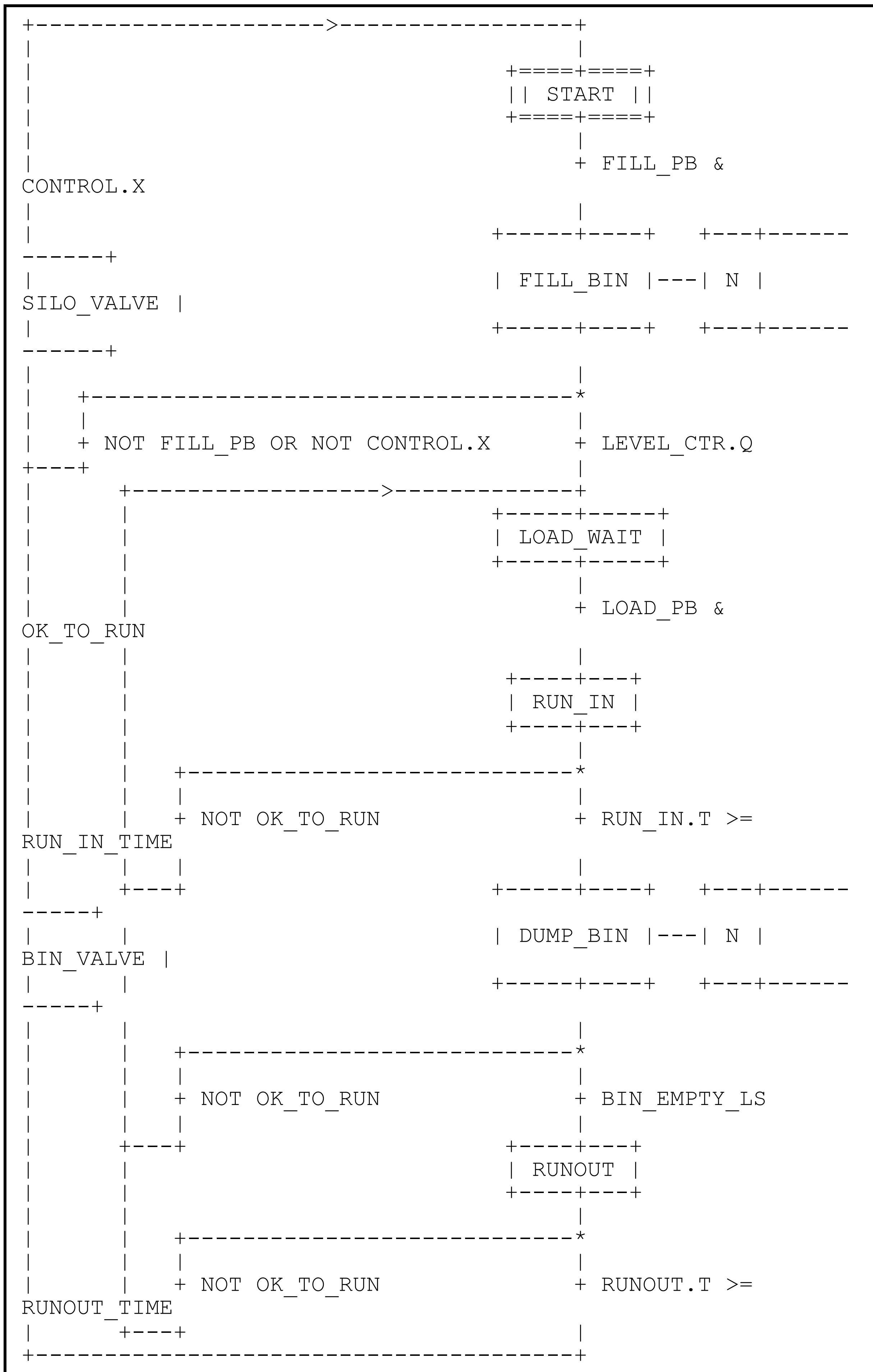

START
FILL_PB & CONTROL.X
FILL_BIN
N
SILO_VALVE
NOT FILL_PB OR NOT CONTROL.X
LEVEL_CTR.Q
LOAD_WAIT
LOAD_PB & OK_TO_RUN
RUN_IN
NOT OK_TO_RUN
RUN_IN.T >= RUN_IN_TIME
DUMP_BIN
N
BIN_VALVE
NOT OK_TO_RUN
BIN_EMPTY_LS
RUNOUT
NOT OK_TO_RUN
RUNOUT.T >= RUNOUT_TIME

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

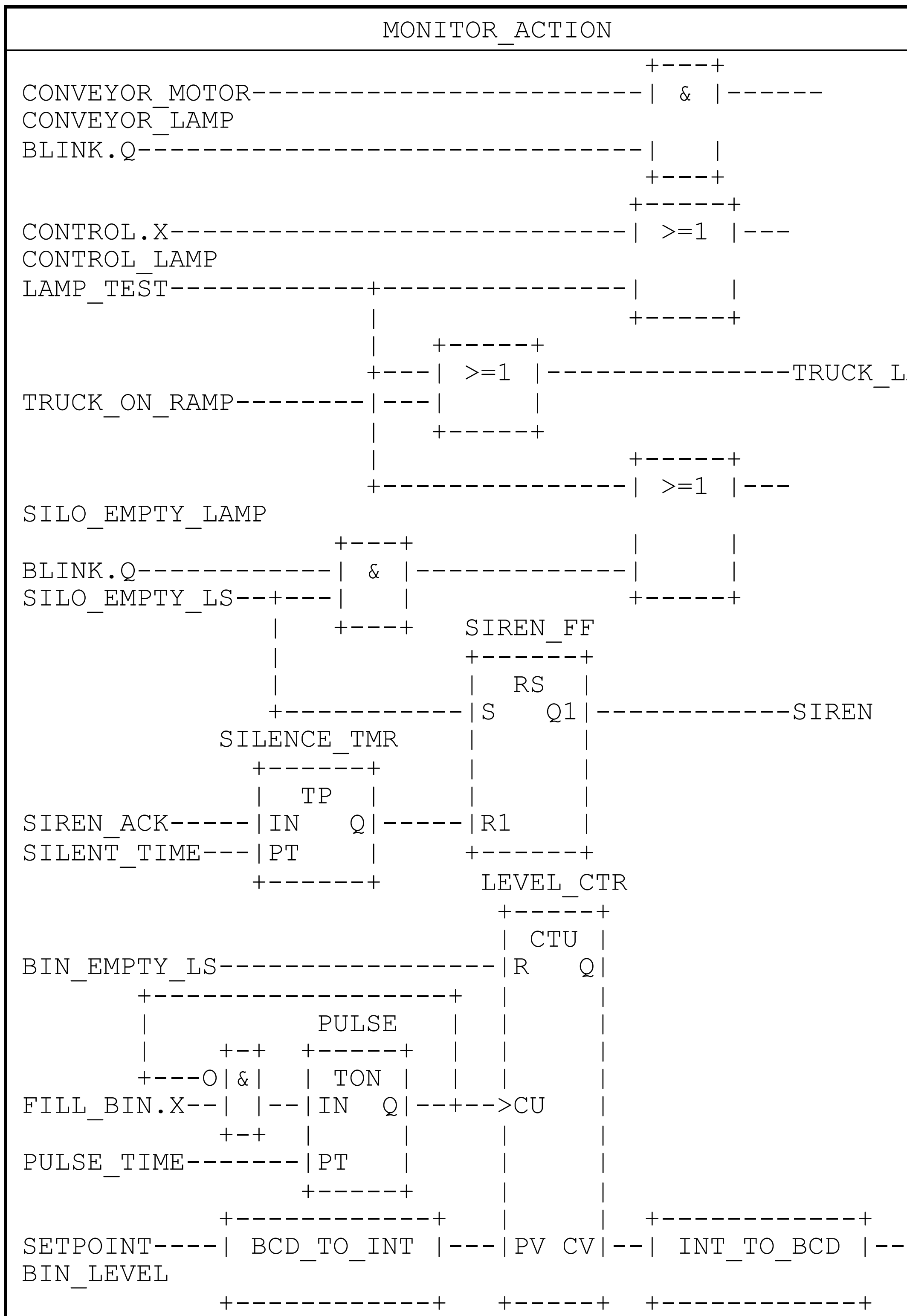


Figure F.6 - Body of action MONITOR_ACTION in FBD language

```
   (* 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.

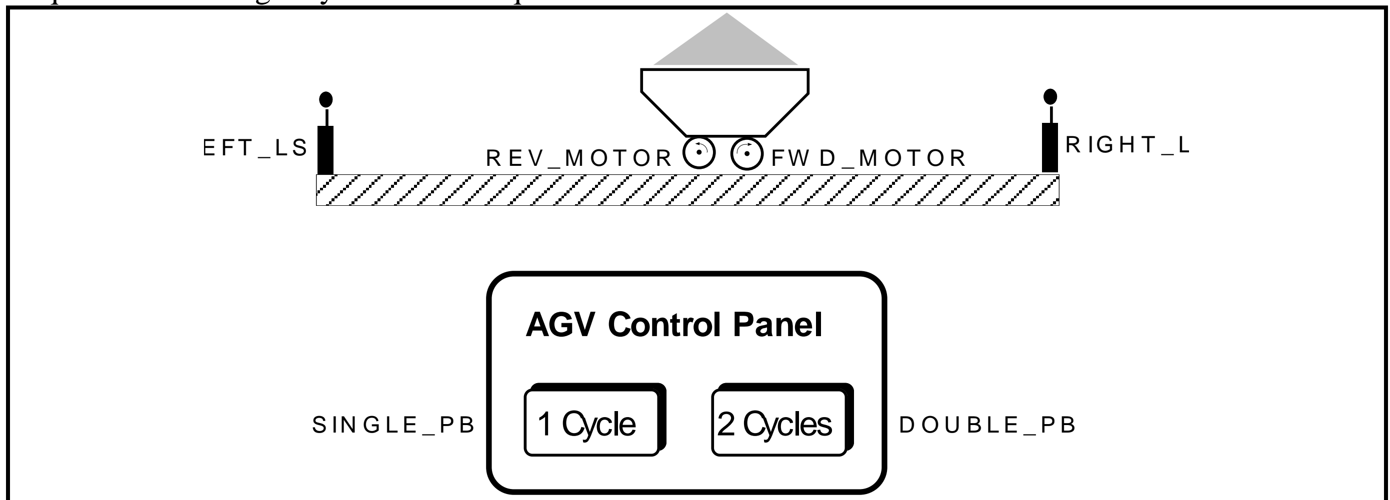


Figure F.9 - Physical model for program AGV

```
        +----------------------+
        |         AGV          |
BOOL---|SINGLE_PB    FWD_MOTOR|---BOOL
BOOL---|DOUBLE_PB    REV_MOTOR|---BOOL
BOOL---|LEFT_LS               |
BOOL---|RIGHT_LS              |
        +----------------------+
```

Figure F.10 - Graphical declaration of program AGV

```
CONFIGURATION AGV_CONTROL

RESOURCE AGV_PROC: SMALL_PC
                 AGV_1
        +--------------------+
        |        AGV         |
%IX1---|SINGLE_PB  FWD_MOTOR|---%QX1
%IX2---|DOUBLE_PB  REV_MOTOR|---%QX2
%IX3---|LEFT_LS             |
%IX4---|RIGHT_LS            |
        +--------------------+
```

Figure F.11 - A graphical configuration of program AGV

```
     +-----------------------------+
     ¦                             ¦
     ¦                         +===+===+    (* Main sequence *)
     ¦                         |¦START¦|
     ¦                         +===+===+
     ¦                             ¦
     ¦       +---------------------*-----------+
     |       |                                 |
     ¦       + READY.X & SINGLE_PB             + READY.X &
DOUBLE_PB
     |       |                                 |
     ¦    +--+---+ +-+-----+               +---+----+  +-+-----+
     ¦    ¦SINGLE+-¦N¦CYCLE¦               ¦DOUBLE_1+--¦N¦CYCLE¦
     ¦    +--+---+ +-+-----+               +---+----+  +-+-----+
     |       |                                 |
     ¦       *----------+                      + DONE.X
     ¦       ¦          + DONE.X & DOUBLE_PB   ¦
     ¦       ¦          +----------------------+
     |       |                                 |
     ¦       ¦                           +-----+-----+
     ¦       + DONE.X & NOT DOUBLE_PB    ¦DOUBLE_WAIT¦
     ¦       ¦                           +-----+-----+
     |       |                                 |
     ¦       ¦                                 + READY.X
     |       |                                 |
     ¦       ¦                             +---+----+  +-+-----+
     ¦       ¦                             ¦DOUBLE_2+--¦N¦CYCLE¦
     ¦       ¦                             +---+----+  +-+-----+
     |       |                                 |
     ¦       ¦                                 + DONE.X
     |       |                                 |
     ¦       +---------------------+-----------+
     ¦                             ¦
     ¦                         +----+-----+
     ¦                         ¦NON_REPEAT¦
     ¦                         +----+-----+
     ¦                              ¦
     ¦                              +NOT(SINGLE_PB OR
DOUBLE_PB)
     ¦                              ¦
     +------------------------------+
```

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)