The operator must acknowledge the alarm by calling up on the screen the affected area. The cause of the alarm shall flash red until acknowledged by a dedicated key whereupon it will remain steady until returned to normal.

B.6 SHUTDOWN PROCEDURE

The shutdown procedure will take two forms - controlled and emergency trip.

B.6.1 Controlled Shutdown

The operator will be able to sequentially shutdown conveyors by taking the following steps :

1. From the overview the operator selects the appropriate route by using dedicated keys.

2. Having selected the route it will be displayed in detail on the screen. The operator will then indicate that he wishes to stop the route, wherupon 2 messages will be displayed asking for confirmation that the route is to be stopped.

3. Upon receiving an affirmative response the supervisory system will automatically transmit the sequential stopping information to the relevant sub-station PLC's.

   Belts will be stopped sequentially, beginning from "upstream" belts, with time delays to allow each belt to clear.

4. When the final conveyor in the route is stopped 2 messages confirming this will be displayed.

B.6.2 Emergency Trip/Stop

Upon operation of any field, MCC or keyboard mounted stop, trip or other simiar device the conveyor with which it is associated and all upstream devices will immediately stop. An alarm condition will then follow as per B.5 above.

Downstream equipment will stop sequentially as per B.6.1 above.

APPENDIX C.

SOFTWARE CONSTRUCTS FOR FAULT DETECTION AND RECOVERY

C.1 FAULT DETECTION CONSTRUCTS

C.1.1 Failure/Alarm Inputs

As examples of this condition consider motor contactor thermal overload trips, fire alarm trigger points, mechanical interlock contacts, etc. . Assume for this construct that healthy is indicated by an ON condition, (logic level 1), and unhealthy is indicated by an OFF condition, (logic level 0).

This means that a fault is signalled for all failure/alarm inputs by simply inverting the input. The construct for a set of trip inputs is thus :-

```
MODULE FAILURE

DO WHILE 1 = 1                    *  ALWAYS EXECUTE

  FAULT_1 = INV(TRIP_1)           *  INVERT TRIP
  FAULT_2 = INV(TRIP_2)           *  INPUTS

  .       .       .
  .       .       .
  .       .       .

  FAULT_x = INV(TRIP_x)

END DO

END FAILURE
```

In the above the FAULT_x and TRIP_x can be considered as single bit logic conditions, or if grouped in words, (16 bits per word say), this can be done for groups of failure/alarm inputs giving efficient coding.

C.1.2 Pattern Recognition and Template Matching

By extending the word logic construct of C.1.1 above, it is sometimes required that only certain faults are to be detected at different steps of a sequence or states of a process.

Assume a construct consisting of conditions COND_y for the steps of a sequence or states of a process, and a corresponding number of masks MASK_x/y for patterns to be recognised or templates to be matched :-

```
MODULE CONTROL

DO WHILE 1 = 1

  DO WHILE COND_1

    BEGIN

      .
      .

    MASK_1 = MASK_1/1          * SETUP MASK FOR BITS
    MASK_2 = MASK_2/1          * TO BE DETECTED WHILE
                               * COND_1 VALID
      .        .               * USED IN FAULT_x DETECT

    MASK_x = MASK_x/1

  END DO


.   .
.   .
```

```
  DO WHILE COND_y
    BEGIN

      .
      .

    MASK_1 = MASK_1/y
    MASK_2 = MASK_2/y

      .       .

    MASK_x = MASK_x/y
  END DO
END CONTROL


MODULE FAILURE
DO WHILE 1 = 1
  FAULT_1 = [INV(TRIP_1)] . MASK_1   * ONLY ALLOW BITS
  FAULT_2 = [INV(TRIP_2)] . MASK_2   * WHICH ARE INVERTED
                                     * AND MATCH BITS IN
     .          .          .         * MASK_x
     .          .          .

  FAULT_x = [INV(TRIP_x)] . MASK_x
END DO
END FAILURE
```

The selection of the relevant bits is done with the mask words. To now get pattern recognition and template matching the fault word is inverted and compared to the mask by :-

```
IF MASK_x = INV(FAULT_x) THEN CONTINUE PROCESS

  ELSE ERROR
```

C.1.3 Detecting Operating Faults

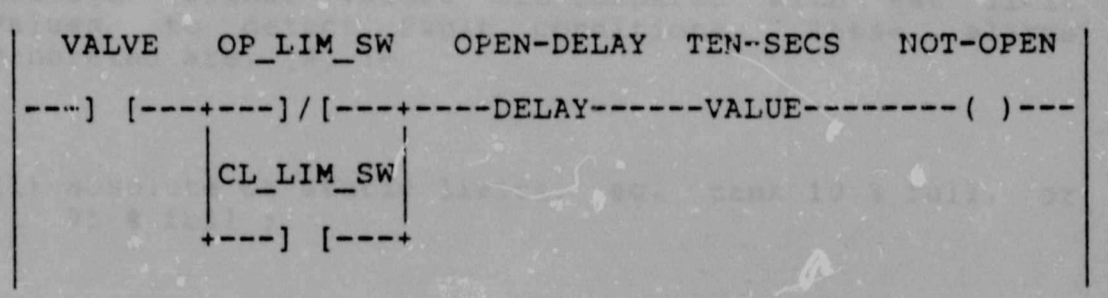
Typical of these faults are pumps not running/stopped, valves not open/closed, etc. .

In this a change of state of an output is checked against status feedback inputs. No device can be controlled instantaneously so this fault detection construct incorporates a time delay for the device to react, and for the feedback signals to steady.

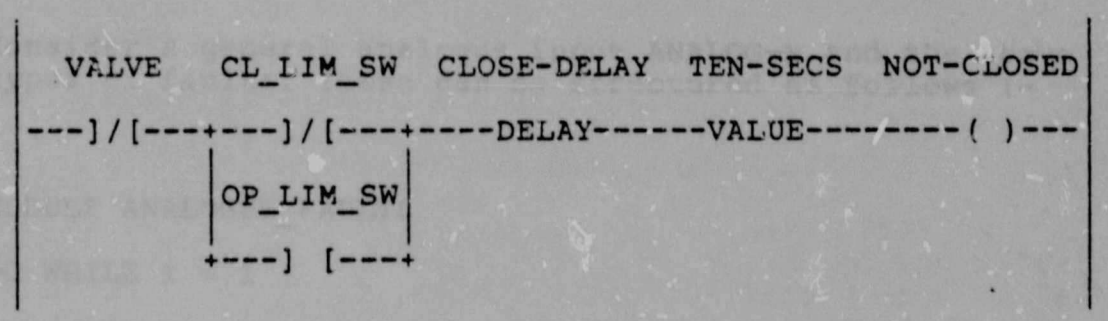
Consider an output for a device named ENERGISE, and two feedback signals OPERATED and NOT_OPERATED. Allowing 10 seconds for feedback signals, the construct is as follows :-

```
MODULE OPERATING_FAULT_DETECT

  DO WHILE 1 = 1                          * ALWAYS EXECUTE

    FAIL_NOT_OPERATED = DELAY(ENERGISE . [INV(OPERATED)]

                        OR NOT_OPERATED, 10 SECS)

                                          * DETECT NOT
                                          * OPERATED FAULT

    FAIL_OPERATED = DELAY( [INV(ENERGISE)] .

                    [INV(NOT_OPERATED)] OR OPERATED,
                                            10 SECS)

                                          * DETECT OPERATED
                                          * FAULT

  END DO

END OPERATING_FAULT_DETECT
```

Example : Consider an output for a valve with open and close limit switches. In ladder logic the detection routine is as follows :-



When the valve is energised, the close limit switch is checked if it indicates that the valve is closed, and the open limit switch is checked if it indicates a not open condition, ie. valve stuck halfway or slow acting.



When the valve is de-energised, the open limit switch is checked if it indicates that the valve is open, and the close limit switch is checked if it indicates a not closed condition, ie. valve stuck halfway or slow acting.

C.1.4 Analogue Signal Faults

Analogue signal values are compared with set limit values to detect fault conditions. These alarms generated are, [8] :-

(1) absolute or static limits, eg. tank 10 % full, or 95 % full ;

(2) the deviation of the variable from a set point ;

(3) zero or full scale range limits, eg. tank empty or tank full ;

(4) rate-of-change limits, eg. temperature rising too fast, calculated and checked against a limit.

Consider a general analogue input ANALOG-x and the above types of faults. These can be structured as follows :-

```
MODULE ANALOGUE_FAULTS

DO WHILE 1 = 1

  HI_LIM_FAULT_x = ANALOG_x > HI_LIMIT_x

      .          .          .       * LOGICAL DETECT OF
      .          .          .       * ANALOG VALUE ABOVE
                                    * HI_LIMIT

  LO_LIM_FAULT_x = ANALOG_x < LO-LIMIT_x

      .          .          .       * LOGICAL DETECT OF
      .          .          .       * ANALOG VALUE BELOW
                                    * LO_LIMIT

  DEVIATION_FAULT_x = ABS(ANALOG_x - SETPOINT_x)

                    > DEVIATION_LIMIT_x
```

```
                .           .           .       * ABSOLUTE DIFFER-
                .           .           .       * ENCE GREATER THAN
                                                * DEVIATION_LIMIT

FULL_SCALE_FAULT_x = ANALOG_x > FULL_SCALE_x

                .           .           .       * ANALOG VALUE ABOVE
                .           .           .       * FULL_SCALE


ZERO_FAULT_x = ANALOG_x < ZERO_x

                .           .           .       * ZERO_x MAY NOT BE
                .           .           .       * TRUE ZERO (IE. 0 )
                                                * EG. 4-20 mA
                                                * SIGNALS


DO WHEN TIMER_x = 10 SECS

                                                * RATE-OF-CHANGE
                                                * TIMER INTERVAL
                                                * ( 10 SECS AS
                                                * EXAMPLE)

  BEGIN

  RATE_CHANGE_FAULT_x = ABS(ANALOG_x - PREV_x)

                      > RATE_CHANGE_LIMIT_x

                                                * DIFFERENCE IN
                                                * PRESENT AND PREV-
                                                * IOUS ANALOG VALUES
                                                * GREATER THAN RATE_
                                                * CHANGE_LIMIT

  PREV_x = ANALOG_x                             * SET PREVIOUS TO
                                                * PRESENT VALUE

  TIMER_x = 0                                   * RESET TIMER

 END DO

                .           .           .       * REPEAT FOR ALL
                .           .           .       * RATE_OF_CHANGE_
                                                * FAULTS


END DO

END ANALOGUE_FAULTS
```

C.1.5 General Fault Analysis

The analysis of faults involves determining what sub-system of the plant the fault affects, and how critical it is to the whole plant. This means that fault analysis constructs become a way of tranposing a fault tree, as in section 2, into a logic system.

Sub-system faults can occur when a number of faults must occur simultaneously or singly, and this is explicitly encoded as AND or OR functions :-

```
MODULE SIMULTANEOUS_FAULTS

DO WHILE 1 = 1

  SIM_FAULT_1 = AND [ FAULT_11, FAULT_12, ... FAULT_1N ]
                  :             :             :
                  :             :             :
  SIM_FAULT_x = AND [ FAULT_x1, FAULT_x2, ... FAULT_xN ]

                                  * DETECT WHEN ALL
                                  * FAULT_xn ARE PRESENT

END DO

END SIMULTANEOUS_FAULTS
```

```
MODULE SINGLE_FAULTS

DO WHILE 1 = 1

  SINGLE_FAULT_1 = OR [ FAULT_11, FAULT_12, ...
                        FAULT_1N ]
```

```
SINGLE_FAULT_x = OR [ FAULT_x1, FAULT_x2, ...
                      FAULT_xN ]

                    * DETECT WHEN ANY OF
                    * FAULT_xn ARE PRESENT

END DO

END SINGLE_FAULTS
```

To create the next level of the fault tree, this could be an AND or OR of some of the above constructs, eg.

Consider where SIM_FAULT_1, SIM_FAULT_3 AND SINGLE_FAULT_6 must occur simultaneously for the next level of faults, say critical faults. The construct then takes the following form :-

```
MODULE CRITICAL_FAULTS

DO WHILE 1 = 1

      .   .
      .   .

  CRIT_FAULT_z  =  AND  [  SIM_FAULT_1,  SIM_FAULT_2,
                           SINGLE_FAULT_6 ]

                       * DETECT WHEN ALL THREE
                       * FAULTS CAN CAUSE A CRITICAL
                       * FAULT

      .   .
      .   .

END DO

END CRITICAL_FAULTS
```

C.2 FAULT RECOVERY CONSTRUCTS

C.2.1 Redundancy Control

Redundancy control specifies that if a sub-system fails, then a duplicate will then take over it's function. This is done in a construct called a recovery block. A recovery block is typically the following :-

```
ENSURE T
        BY P
                ELSE BY Q
ELSE ERROR
```

In the above, T can be a process condition or a plant state required, and P and Q are two operational systems which do controls to satisfy T. The ERROR is to be considered as a complete shutdown signal.

Using the fault tree system and the constructs for fault analysis, system P will eventually get a fault construct of the following form :-

```
CRIT_FAULT_P = OR [ SIM_FAULT_P1, SIM_FAULT_P2, ....
                    SINGLE_FAULT_P1, ... ]
```

And for sub-system Q :-

```
CRIT_FAULT_Q = OR [ SIM_FAULT_Q1, SIM_FAULT_Q2, ....
                    SINGLE_FAULT_Q1, .... ]
```

Then the recovery block construct is as follows :-

```
MODULE REDUNDANCY

DO WHILE CRIT_FAULT_P = 0
  BEGIN

      .   .   .              * CONTROLS SOFTWARE
      .   .   .              * FOR SUB-SYSTEM P

  END
END DO

DO WHILE AND [ CRIT_FAULT_P = 1, CRIT_FAULT_Q = 0 ]
  BEGIN

      .   .   .              * CONTROLS SOFTWARE
      .   .   .              * FOR SUB-SYSTEM Q

  END
END DO

DO WHILE AND [ CRIT_FAULT_P = 1, CRIT_FAULT_Q = 1 ]
  BEGIN

      .   .   .              * SHUTDOWN CONTROL
      .   .   .              * SOFTWARE

  END
END DO

END REDUNDANCY
```

C.2.2 Backward State Recovery

To do backward state recovery to a known safe state in the past the software engineer must be sure that the real-time characteristics of the process being contolled are reversible in the control software. In complex processes this is generally never true, and so forward recovery and conditional forward recovery should only be implemented. For batch sequences, it is possible to affect backward recovery.

To affect backward recovery in a single process element the states of outputs, inputs and internal states, (eg. timers, sequence step flags), must be stored at regular intervals. This data is termed Class I, II and III data respectively, [4] :-

```
MODULE STORE_DATA

DO WHEN TIMER_x = 10 SECS              * 10 SECS STORAGE
                                       * TIME FOR EXAMPLE

  BEGIN

  STORE (TIME)                         * STORE TIME OF
                                       * DATA STORAGE
  STORE (OUTPUTS)                      *
                                       * STORE CLASS I, II
  STORE (INPUTS)                       * AND III DATA
                                       *
  STORE (INTERNAL DATA)                *
                                       *
  INCREMENT  STORE POINTERS            * POINTERS USED FOR
                                       * DATA  STORAGE
                                       * OFFSET ADDRESSES

  TIMER_x = 0                          * RESET TIMER

  END

END DO

END STORE_DATA
```

To restore the states on a fault condition, use the following construct :-

```
MODULE RESTORE_STATES

DO WHILE FAULT = 1

  BEGIN

  DECREMENT POINTERS

  RESTORE (TIME)                    * RESTORE THE LAST
                                    * TIME STORED

  RESTORE (OUTPUTS)                 * RESTORE ALL
                                    * OUTPUTS

  RESTORE (INPUTS)                  * RESTORE INPUTS

  RESTORE (INTERNAL DATA)           * RESTORE INTERNAL
                                    * DATA

  TIMER_x = 0                       * RESET TIMER

  END

END DO

END RESTORE_STATES
```

The possible faults can still occur and the module above would be re-executed repeatedly for a critical fault. This would carry on until the pointer was back to zero and the process would have affected a complete shutdown. The recurrence of faults, especially in muli-tasking or multi-processor systems creates the Domino Effect, and this would require each task or process to store at the same time, thus given a consistent check point in the processing sequence/s.

eg. Take case study I. If an unplanned shutdown occurs in step 03 of the process, backward recovery goes back to step 02.

C.2.3 Forward State Recovery

Forward state recovery is simpler to affect in that known states are enforced to cause a known condition.

The construct is as follows :-

```
MODULE FORWARD_RECOVERY

DO WHILE FAULT = 1
  BEGIN

  SET (OUTPUTS)                    * SET UP STATES
                                   * FOR A KNOWN
  SET (INPUTS)                     * GOOD STATE

  SET (INTERNAL DATA)

  END

END DO

END FORWARD_RECOVERY
```

eg. Take case study I. If the system check in step 01 fails then the system aborts the process back to Rest mode. In this all outputs are reset to OFF, and all control timers reset.

REFERENCES.

1. Himmelblau , D.M. - " Fault Detection and Diagnosis in Chemical and Petrochemical Processes" , Elsevier Scientific Publishing Co. , 1975.

2. Laduzinsky , A.J. - " Software Takes Modular Forms for Effective Process and Factory Control " , Control Engineering , Vol 29 , No. 12 , November 1982.

3. Rosenhof , H.P. - " Successful Batch Control Planning : A Path to Plant-Wide Automation " , Control Engineering , Vol 29 , No. 10 , September 1982.

4. Kopetz , H. - " Distributed Computer Control Systems " , WITS , 1981.

5. Weber , G.W. - " A Study on the Aspects of Fault-Tolerance of Computing Systems " , MSc(Eng) dissertation , WITS , 1984.

6. Editor - " Dual Dissimilar Microprocessors give Fail-Safe Aircraft Control " , Control Engineering, Vol 29 , No. 10 , 1982.

7. Merry , M. - " APEX3 : An Expert System Shell for Fault Diagnosis " , the GEC Journal of Research , Vol 1 , No. 1 , 1983.

8. Plamping , K. - " The design of process alarm systems " , Transactions of the Institute of Measurement and Control , Vol 5 , No. 3 , July-September 1983.

9. Ruth , S. - " CRT terminals - an overview " , Instruments & Control Systems , May 1981.

10. Laduzinsky , A.J. - " Displays and Annunciators :- Interfaces to the Effective Operator Control " , Control Engineering , Vol 29 , No. 11 , October 1982.

11. Sengupta , A. , et al - " Realisation of Fault-Tolerant Machines - Linear Code Application " , IEEE transactions on Computers , Vol C-30 , No. 3 , March 1981.

12. Bellon , C. , Saucier , G. - " Protection Against External Errors in a Dedicated System " , IEEE transactions on Computers , Vol C-31 , No. 4 , 1982.