Diagnostics Using PLC Communications

Abstract

Many applications use ORMEC PLC communications support such as DATA HIGHWAY and MODBUS for transferring information between a GEN-III controller and a host controller or PLC. This Tech Note shows a way to set up the error handling and diagnostics in such applications.

Description

The example program use MODBUS to connect the GEN-III to another computer. The GEN-III is the MODBUS slave and the computer is the MODBUS master. The example program focuses on the following areas of the application.

- Configuring MODBUS
- Fault Handling
- Making diagnostic information available to the computer.
- Clearing faults and restarting.

The techniques shown can also be used on applications that use DATA HIGHWAY.

Modbus Configuration

The INIT.MODBUS routine configures MODBUS to use the MotionNET port as slave station number 13. This configuration can be adjusted to suit your application. The routine also goes on to MAP 22 elements of an integer array called DIAG to registers 1 through 22. These registers will allow the MODBUS master to access the GEN-III diagnostic information.

E-STOP State

Once the initialization is complete, the program enters a routine called ESTOP.STATE. In this routine the program looks a the FAULT@ variable. If FAULT@ is false it checks the MotionBASIC® version number and uses one of two procedures to set FAULT@ true in order to make sure the NO-FAULT relay is open.

If you know you will be using MotionBASIC® version 2.1a or higher, you can simply set FAULT@=TRUE. Otherwise you must follow the more complex procedure contained in the FAKE.A.FAULT routine.

Once in the ESTOP.STATE routine, the program loop until it sees the ESTOP.OK@ input make a transition from false to true. When this happens, the program exits the loop, clears any faults and goes to the RESTART routine which is where your application program restarts after a fault.

Fault Response

When a fault or program error occurs, the program goes to ERROR.HDLR. Here the program calls GET.FAULTS to put the diagnostic data into the MODBUS diagnostic registers. It then executes a routine called ESTOP which stops the axes and performs any other actions required when the system faults. It then goes into the ESTOP.STATE to await the ESTOP.OK@ transition that clears faults and restarts your application.
GET.FAULTS

This routine transfers MotionBASIC’s diagnostic data to the MODBUS diagnostic registers. Because MODBUS only supports integer registers, some of the data must be converted into integer form so the master can access the data.

Two types of conversion are used:

Numbers that can be larger than an integer, such as the line number where the fault occurred, are converted to low word and high word registers using modulo 10,000. Your master program can reconstitute the number by multiplying the high word by 10000 and adding the low word.

Set variables are converted to a binary integer. For example the set value \{1,6,8\} is converted to \(2^{1-1} + 2^{6-1} + 2^{8-1} = 162\). Each bit in the resulting integer corresponds to the appropriate bit in the set. This conversion is handled by a routine called SET.TO.INT.

PRINT.FAULTS

This subroutine is not actually used in the program. It is provided so you can convert the diagnostic register data back to its original form and display it on the MotionPRO™ screen.

Host Computer Diagnostic Display

The purpose of making the diagnostic data available to the MODBUS master is so the host computer can display GEN-III fault information to the operator. There is quite a lot of information in the diagnostic registers and it is important to display it in a way that does not confuse the operator.

The most important information is the MotionBASIC® error code, register 1. This code always tells you the reason the program entered the error handler and is the most useful in figuring out what happened. All other faults and/or axis faults occurred after or as a consequence of the original error.

If the error code is 1911 the problem was an axis fault. If more than one axis indicates fault, AXIS.FLT1@, register 5, points to the axis that caused the original problem. The other axes faulted after or as a result of the original fault.

The ESTOP.STATE routine will open the NO-FAULT relay if the actual fault did not already do so. This process will set the Machine Fault bit \{8\} in the FAULT@ variable, register 4. Unless your program uses user defined faults, you may want to ignore bit 8 of register 4 in your display.
## Diagnostic Registers

The following table shows the diagnostic register assignments:

<table>
<thead>
<tr>
<th>Register Number</th>
<th>MotionBASIC® Variable</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ERR</td>
<td>MotionBASIC® Error Code Number</td>
<td>Integer number</td>
</tr>
<tr>
<td>2</td>
<td>ERL</td>
<td>Current line number when error occurred</td>
<td>Low word, modulo 10000</td>
</tr>
<tr>
<td>3</td>
<td>ERL</td>
<td></td>
<td>High word, modulo 10000</td>
</tr>
<tr>
<td>4</td>
<td>FAULT@</td>
<td>Set of all Controller Faults</td>
<td>Integer bit image</td>
</tr>
<tr>
<td>5</td>
<td>AXIS.FLT1@</td>
<td>First axis to fault</td>
<td>Integer bit image</td>
</tr>
<tr>
<td>6</td>
<td>AXIS.FAULT@</td>
<td>Set of all faulted axes</td>
<td>Integer bit image</td>
</tr>
<tr>
<td>7</td>
<td>AFAULT@1(1)</td>
<td>Axis 1 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>8</td>
<td>ALARM@1(1)</td>
<td>Axis 1 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>9</td>
<td>AFAULT@2(2)</td>
<td>Axis 2 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>10</td>
<td>ALARM@2(2)</td>
<td>Axis 2 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>11</td>
<td>AFAULT@3(3)</td>
<td>Axis 3 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>12</td>
<td>ALARM@3(3)</td>
<td>Axis 3 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>13</td>
<td>AFAULT@4(4)</td>
<td>Axis 4 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>14</td>
<td>ALARM@4(4)</td>
<td>Axis 4 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>15</td>
<td>AFAULT@6(6)</td>
<td>Axis 6 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>16</td>
<td>ALARM@6(6)</td>
<td>Axis 6 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>17</td>
<td>AFAULT@7(7)</td>
<td>Axis 7 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>18</td>
<td>ALARM@7(7)</td>
<td>Axis 7 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>19</td>
<td>AFAULT@8(8)</td>
<td>Axis 8 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>20</td>
<td>ALARM@8(8)</td>
<td>Axis 8 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
<tr>
<td>21</td>
<td>AFAULT@9(9)</td>
<td>Axis 9 fault code</td>
<td>Integer number</td>
</tr>
<tr>
<td>22</td>
<td>ALARM@9(9)</td>
<td>Axis 9 Servodrive alarm code</td>
<td>Integer number</td>
</tr>
</tbody>
</table>
Program Listing

Module: TN012.BAS
Routine name: POWERUP
Abstract: Program entry point
Routines called: *MP.CONFIG, CLEAR.FAULTS, ERROR.HDLR
ESTOP.STATE, INIT.MODBUS, INIT.VARIABLES
Variables used: None

POWERUP:
MP.CONFIG
INIT.VARIABLES
INIT.MODBUS
CLEAR.FAULTS
ON ERROR GOTO ERROR.HDLR
ESTOP.STATE ‘put the machine in ESTOP state
END

Module: TN012.BAS
Routine name: INIT.VARIABLES
Abstract: Initialize any program variables
Routines called: None
Variables used: DIAG()

INIT.VARIABLES:
ERASE DIAG :DIM DIAG(22) ‘dimension the diagnostic register array
RETURN

Module: TN012.BAS
Routine name: INIT.MODBUS
Abstract: Initialize MODBUS communications
Routines called: None
Variables used: DIAG(), MAP, MOD.CFG, MOD.INIT, TMP

INIT.MODBUS:
MOD.CFG 0,,0,7 ‘MotionNet port, 9600 baud, no parity, 7 data bits
MOD.INIT 13,0,1 ‘station 13, slave, ASCII mode
MAP ERASE
FOR TMP=1 TO 22
MAP TMP TO DIAG(TMP) ‘map registers to the diagnostic array
NEXT TMP
RETURN

TN012a -4- April 2, 1993
Customer Support Engineering Tech Note #12

Module: TN012.BAS
Routine name: ESTOP.STATE
Abstract: Wait here for transition of ESTOP.OK@ from false to true to clear faults and restart
Routines called: CLEAR.FAULTS, FAKE.A.FAULT, RESTART
Variables used: ESTOP.FLAG, ESTOP.OK@, FAULT@

ESTOP.STATE:
IF FAULT@<>{} THEN
  IF MBVER$>="MB2.1a" THEN FAULT@=TRUE ELSE FAKE.A.FAULT ENDIF
  ESTOP.FLAG=NOT ESTOP.OK@
  WHILE NOT (ESTOP.FLAG AND ESTOP.OK@)
    IF NOT ESTOP.OK@ THEN ESTOP.FLAG=TRUE
  WEND
  CLEAR.FAULTS
  STACK CLEAR
  RESTART 'restart your main program
END

Module: TN012.BAS
Routine name: RESTART
Abstract: Application restart location. This is where your program takes over.
Routines called: None
Variables used: None

RESTART:
  WHILE TRUE :WEND
END

Module: TN012.BAS
Routine name: ERROR.HDLR
Abstract: React to a fault or error
Routines called: ESTOP, ESTOP.STATE, GET.FAULTS
Variables used: MODBUS@

ERROR.HDLR:
  IF ERR=1805 THEN MODBUS@=OFF: ON ERROR GOTO 0
  GET.FAULTS
  ESTOP
  RESUME ESTOP.STATE
END

TN012a -5- April 2, 1993
Routine name: ESTOP

Abstract: Emergency Stop

Routines called: None

Variables used: AXIS.LIST@, DSP.DONE@(), FAULT@, MODE@()

```
ESTOP:
    HALT AXIS.LIST@
    WAIT UNTIL DSP.DONE@(AXIS.LIST@) OR FAULT@<>{}
    MODE@(AXIS.LIST@)=0
RETURN
```

Routine name: CLEAR.FAULTS

Abstract: Attempt to clear faults

Routines called: GET.FAULTS

Variables used: AFAULT@, DIAG(), FAULT@, OTL.FWD@, OTL.REV@

```
CLEAR.FAULTS:
    OTL.FWD@=0 :OTL.REV@=0 :AFAULT@=0 :FAULT@=0 :WAIT 300
    GET.FAULTS :DIAG(1)=0 :DIAG(2)=0 :DIAG(3)=0
RETURN
```

Routine name: FAKE.A.FAULT

Abstract: Create a fake fault to make sure NO FAULT relay opens up

Routines called: TEMP.ROUTINE

Variables used: None

```
FAKE.A.FAULT:
    ON ERROR GOTO TEMP.ROUTINE 'set up a temporary error handler
    ERROR 1901 'cause a machine fault
RETURN
```

Routine name: TEMP.ROUTINE

Abstract: Temporary error handler used by FAKE.A.FAULT

Routines called: ERROR.HDLR

Variables used: None

```
TEMP.ROUTINE:
    ON ERROR GOTO ERROR.HDLR 'restore the normal error handler
    RESUME NEXT
```
Abstract: Transfer fault variables to registers

Variables used: AFAULT@(), ALARM@(), AXIS.FAULT@, AXIS.FLT1@
AXIS.LIST@, DIAG(), FAULT@, INT.LO, SET~, TMP

GET.FAULTS:

unit diagnostic codes

DIAG(1)=ERR 'program error code
DIAG(2)=ERL MOD 10000 'low word (modulo 10000) of error line number
DIAG(3)=FIX(ERL\10000) 'high word (modulo 10000) of error line number

SET~=FAULT@ :SET.TO.INT :DIAG(4)=INT.LO 'fault@ binary word
SET~=AXIS.FLT1@ :SET.TO.INT :DIAG(5)=INT.LO 'axis.flt1@ binary word
SET~=AXIS.FAULT@ :SET.TO.INT :DIAG(6)=INT.LO 'axis.fault@ binary word

axis diagnostic codes

FOR TMP~ WITHIN AXIS.LIST@
TMP=TMP~
IF TMP>4 THEN TMP=(TMP*2)+4 ELSE TMP=(TMP*2)+5
DIAG(TMP)=AFAULT@(TMP~) 'axis faults
DIAG(TMP+1)=ALARM@(TMP~) 'servo drive alarms
NEXT TMP~

RETURN

SET.TO.INT:

INT.LO=0
INT.HI=0
FOR TMP=0 TO 15
TMP~=(TMP+1)
TMP1~=(TMP+16)
IF TMP~*SET~ THEN INT.LO=INT.LO+2^TMP
IF TMP1~*SET~ THEN INT.HI=INT.HI+2^TMP
NEXT TMP

RETURN
Module: TN012.BAS

Routine name: PRINT.FAULTS

Abstract: Print fault data from registers, this is useful for debugging

Routines called: INT.TO.SET

Variables used: DIAG(), INT.HI, INT.LO, SET~, TMP, TMP1

PRINT.FAULTS:

CLS
PRINT "Error code"DIAG(1)":"ERR$(DIAG(1))
PRINT "at line"DIAG(2)+DIAG(3)*10000
INT.HI=0 :INT.LO=DIAG(4) :INT.TO.SET :PRINT "FAULT@="SET~
INT.HI=0 :INT.LO=DIAG(5) :INT.TO.SET :PRINT "AXIS.FLT1@="SET~
INT.HI=0 :INT.LO=DIAG(6) :INT.TO.SET :PRINT "AXIS.FAULT@="SET~
PRINT "Axis","AFAULT@","ALARM@"
FOR TMP=1 TO 8
  IF TMP<5 THEN TMP1=TMP ELSE TMP1=TMP+1
  PRINT TMP1,DIAG((TMP*2)+5),DIAG((TMP*2)+6)
NEXT TMP
RETURN

Module: TN012.BAS

Routine name: INT.TO.SET

Abstract: Convert INT.LO and INT.HI to a set variable

Routines called: None

Variables used: INT.HI, INT.LO, SET~, TMP, TMP&

INT.TO.SET:

SET~={}
FOR TMP=0 TO 15
  TMP&=2^TMP
  IF TMP& AND INT.LO THEN SET~+=SET~+(TMP+1)
  IF TMP& AND INT.HI THEN SET~+=SET~+(TMP+17)
NEXT TMP
RETURN
Traverse Winder Application

Abstract

Many applications require a set of motions to be executed successively, one immediately following the other. These motions can be either time based or slaved to the motion of another axis. The REPEAT command can be used with both the MOVE FOR and GEAR FOR commands to achieve a continuous sequence of motions. A continuous motion Traverse Winder application is used to demonstrate the use of the REPEAT GEAR command.

Description

A Traverse Winder system is used to evenly wrap material around a core with a width greater than that of the material being wound (examples: fishing line, spooled wire). A system of this type consists of two axes, the Winder and the Traverse.

The Winder axis wraps the material around the core, and the Traverse axis guides the material back and forth along the core's length. In order for the material to be evenly wrapped around the core a fixed relationship must exist between the motions of the Winder and Traverse axes. It is also important that the Traverse axis change it's direction of motion at the end of it's travel at points in the Winder's rotation that are offset from each other..

For our example the both axes will be servo controlled motors, however, the Winder axis could be a pacer encoder.

Implementation

Assume that the user units have been configured such that the Winder position units are degrees, and the Traverse position units are inches.

The following code is a table of operator configurable parameters which define the operation of the system:

```
OPERATOR.CONFIG:
  CYCLE.OFFSET   =120   'offset per cycle in master degrees
  TRAVERSE.ACCEL =45    'master degrees for acceleration
  TRAVERSE.DIST! =1.000 'Traverse travel in inches
  WINDER.REVS    =1     'numbers of Winder revs per Traverse
  CYCLES         =10    'cycle = forward index & reverse index
RETURN
```

Each cycle of the Traverse axis consists of two passes, forward and reverse, between 0” 1.000”. At the end of each Traverse axis cycle the Winder axis is 120 degrees out of phase from the end of the previous cycle.

The following code is the calculations required for the Traverse axis motion. NOTE: The position User Units are scaled up by 1000 for the Traverse axis, and 10 for the Winder axis, for better resolution.
CALCULATIONS:

INDEX& = TRAVERSE.DIST! * 1000

'index distance scaled for user units

WINDER.DIST& = (WINDER.REVS * 360 + CYCLE.OFFSET / 2) * 10

'total Winder distance for Traverse motion

ACCEL.DIST& = TRAVERSE.ACCEL * 10

'Winder distance for Traverse acceleration

RETURN

The following code is the program that is executed to operate the system:

MAIN:

WINDER = 1
TRAVERSE = 2

MP.CONFIG ' configure controller parameters
OPERATOR.CONFIG ' configure system
CALCULATIONS ' calculate motion parameters

' clear faults and enable the axes

AXIS.SET@ = AXIS.LIST@
AFAULT@ = 0 :FAULT@ = 0 :WAIT 300 :MODE@ = 5 :WAIT 500

AXIS.SET@ = TRAVERSE

'REPEAT GEAR FOR INDEX& IN WINDER.DIST&, ACCEL.DIST&
'REPEAT GEAR FOR -INDEX& IN WINDER.DIST&, ACCEL.DIST&

'MOVE WINDER AT 60 IN 250

This part of the program keeps track of where the axis is in its motion so that the queue is interrupted at the desired point in the index sequence.

CYCLE.COUNT = 0 ' count the number of cycles completed

WHILE CYCLE.COUNT < CYCLES

WAIT UNTIL POS.ACT@ > INDEX& / 2
WAIT UNTIL POS.ACT@ < INDEX& / 2

CYCLE.COUNT = CYCLE.COUNT + 1

WEND

In order to interrupt the motion queue a GEAR or MOVE for 0 distance must be commanded. This will stop the motion sequence after the motion in progress.

GEAR FOR 0 IN 10 ' interrupt the DSP motion queue

WAIT UNTIL DSP.DONE@
HALT WINDER IN 250

RETURN

END

Performance Considerations

The motions in the queue are initiated one immediately after the other, without missing a DSP tick.