SPP: A DATA BASE PROCESSOR DATA COMMUNICATIONS PROTOCOL

Paul A. Fishwick

Kentron Technical Center
Hampton, Virginia 23666

Contract NASI-16000
May 1983
SUMMARY

The design and implementation of a data communications protocol for the Intel Data Base Processor (DBP) is defined. The protocol is termed SPP (Service Port Protocol) since it enables data transfer between the host computer and the DBP service port. The protocol implementation is extensible in that it is explicitly layered and the protocol functionality is hierarchically organized. Extensive trace and performance capabilities have been supplied with the protocol software to permit optional efficient monitoring of the data transfer between the host and the Intel data base processor. Machine independence was considered to be an important attribute during the design and implementation of SPP. The protocol source is fully commented and is included in Appendix A of this report.

INTRODUCTION

SPP (Service Port Protocol) is defined to be the supporting first layer of the HILDA system. HILDA stands for "High Level Data Abstraction System" and is a three-layer system supporting the data abstraction features of the Intel Data Base Processor (DBP). The purpose of HILDA is the establishment of a flexible method for efficiently communicating with the Intel Data Base Processor. Each layer within HILDA plays a specific role during this communication. These roles may be seen in figures 1 and 2. The purpose of this report is to document the design and implementation of SPP in addition to the trace and performance diagnostic utilities available in the protocol package.

AN OVERVIEW OF SPP

SPP is an asynchronous data communications protocol that has been designed and implemented for use with the Intel Data Base Processor. The protocol permits complete usage of the DBP functionality. The physical environment in which the DBP operates is shown in figure 3 and consists of the host DEC VAX 11/780 with VMS level 3 operating system, the Intel Data Base Processor, and an RS-232 connection. At its most abstract interpretation, SPP is composed of the two procedures "Send Request" and "Receive Response." The SPP user may send a request (composed of a contiguous set of encoded commands) and receive a set of responses which may be in
the form of ASCII text or a more general binary form. "Send Request" and "Receive Response" activate a hierarchy of hand-shaking primitives which include error detection and correction capabilities using cyclic redundancy checking on both the host and DBP sides.

SPP may be viewed as a three-layer protocol. The "layer" within the protocol should not, however, be confused with the layers within HILDA (see fig. 4). The SPP layers may, therefore, be construed to be sub-layers of the HILDA data communications layer. The three layers of SPP are:

1. Application/Session:

The layer representing the highest level interface between the application software on the host computer and the DBP.

2. Data Link:

A middle protocol layer representing structured data transmission hand-shaking implemented with error detection and correction.

3. Physical Link:

The layer closest to the DBP, representing a primitive block I/O capability.

It is important to note that all procedures within the layers of the protocol operate strictly on the host computer. The Intel DBP has its own embedded set of protocol layers in firmware. Each of the SPP protocol layers will be separately discussed.

The Application/Session Layer

This is the protocol layer closest to the actual DBMS (Data Base Management System) application software accessing the data base machine. The application/session layer is composed primarily of two procedures, "SEND-REQUEST" and "RECV-RESPONSE" which perform as demonstrated below:
<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND-REQUEST</td>
<td>MODULE</td>
<td>Byte array to be sent</td>
</tr>
<tr>
<td></td>
<td>NBYTES-SENT</td>
<td>Number of bytes in 'MODULE'</td>
</tr>
<tr>
<td></td>
<td>PCBTYPE</td>
<td>Control or application PCB flag</td>
</tr>
<tr>
<td></td>
<td>APPLICATION-ID</td>
<td>A host-assigned id #</td>
</tr>
<tr>
<td></td>
<td>REQUEST-ID</td>
<td>Id # of the session making the request</td>
</tr>
<tr>
<td>RECV-RESPONSE</td>
<td>MODULE</td>
<td>Byte array received from DBP</td>
</tr>
<tr>
<td></td>
<td>NBYTES-RECV</td>
<td>Number of bytes received</td>
</tr>
<tr>
<td></td>
<td>PCBTYPE</td>
<td>Control or application PCB flag</td>
</tr>
<tr>
<td></td>
<td>MORE-TO-COME</td>
<td>Boolean flag representing when all DBP data has been received</td>
</tr>
</tbody>
</table>

Note that "PCB" stands for "Parameter Control Block" which is described further in the section on data structures. The "APPLICATION-ID" argument (in SEND-REQUEST) is a host-assigned number identifying the application program which will be sending the request to the DBP. "REQUEST-ID" (or session id) refers to the DBP-assigned number identifying the application program. A program that is sending a request to the control session must use a REQUEST-ID of zero, whereas programs sending application session requests may use the REQUEST-ID numbers 1 to 4 which are assigned by the DBP when the host creates application sessions. The request module contains "NBYTES-SENT" bytes of DBP machine code. It should be noted that the response module returned may be null (that is, NBYTES-RECV is zero) since many DBP operations do not yield a response. An example of the use of the above procedures may be shown in the form of the DBP conceptual command "REMARK <HOST> <HELLO>" which is performed after having started up the DBP with "DBPSTART:

```c
C
C FORTRAN EXAMPLE
C
BYTE MODULE(512)
PARAMETER APPLICATION = 1
DATA MODULE
X  /'3A'X,'01'X,'01'X,'05'X,
X  '/48'X,'45'X,'4C'X,'4C'X,'4F'X,'FF'X,'00'X/
```
CALL SEND-REQUEST (MODULE,11,APPLICATION,1,1)
CALL RECV-RESPONSE (MODULE,NBYTES-RECV,APPLICATION,MORE-TO-COME)

Figure 5 graphically depicts the general form of the Host-DBP interaction occurring during the SEND-REQUEST and RECV-RESPONSE procedures. Note that each DBP request module is prefixed by the "APPLICATION-ID" and "REQUEST-ID." This four-byte prefix is inserted by the SEND-REQUEST procedure. The prefix need not be placed within the request module itself. A list of the valid machine codes and formats for request and response modules may be found in the DBP Reference Manual (1).

The Data Link Layer

The data link layer is composed of the two operations "READ-BLOCK" and "WRITE-BLOCK." Data "blocks" may be viewed as the error-free transfer medium used during I/O with DBP. A Cyclic Redundancy Check (using the CRC16 polynomial) has been implemented so that the data within the block is re-transmitted if an error is detected during transmission (2). The format of the two data link procedures is shown below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-BLOCK</td>
<td>BLOCK, NBYTES, NBYTES-RECV, BASE, OFFSET</td>
<td>Block of data to read from DBP Number of bytes to read Number of actual bytes read (including header, data, and trailing bytes) Base address for I/O Offset from BASE</td>
</tr>
<tr>
<td>WRITE-BLOCK</td>
<td>BLOCK, NBYTES, BASE, OFFSET</td>
<td>Block of data to write to DBP Number of bytes to write Base address for I/O Offset from BASE</td>
</tr>
</tbody>
</table>
The Physical Link Layer

The physical link layer is the protocol layer closest to the DBP. It represents the actual serial I/O on the channel. At this level, there is no error correction. For correct operation it is imperative that the TTY port and channel be configured correctly, otherwise ambiguities are sure to occur. Figure 6 displays the appropriate communications parameters which need to be set for the VAX. The physical link layer is represented by two procedures "Q-INPUT" and "Q-OUTPUT" (The VMS operating system assigns queues to each port (3)). The following table summarizes the format for the "Q-INPUT" and "Q-OUTPUT" operations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q-INPUT</td>
<td>BYTES</td>
<td>Byte array received from DBP</td>
</tr>
<tr>
<td></td>
<td>NBYTES-RECV</td>
<td>Number of received bytes</td>
</tr>
<tr>
<td>Q-OUTPUT</td>
<td>BYTES</td>
<td>Byte array to be sent to the DBP</td>
</tr>
<tr>
<td></td>
<td>NBYTES</td>
<td>Number of bytes to be sent</td>
</tr>
</tbody>
</table>

THE SPP THREADED DATA STRUCTURE

The DBP Service Port Protocol uses a simple memory mapped I/O scheme to handle the DBMS control and application functions. The core of this scheme is represented as the PCB (Parameter Control Block) Vector. This vector contains pointers to the control and application address blocks. Depending on the type of DBMS function to be performed (control or application), the DBP commands are sent using the appropriate I/O addresses. All addresses are specified in a base:offset (4 bytes) format. Access to the data areas, whether the data is request or response data, is obtained by 'threading' through the PCB Vector and specific PCB (see fig. 7).

OPERATION OF SPP

This section defines the actual operation of SPP in the implementation. The protocol should be used at the application/session layer level, that is, using the
two session procedures "SEND-REQUEST" and "RECV-RESPONSE." The procedure for successfully communicating with the DBP is shown below:

<table>
<thead>
<tr>
<th>Program</th>
<th>Procedures Activated</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPSTART</td>
<td>INIT-COMM</td>
<td>Initialize communications</td>
</tr>
<tr>
<td></td>
<td>CREATE-CONTROL</td>
<td>Create control session</td>
</tr>
<tr>
<td></td>
<td>CREATE-APPLICATION</td>
<td>Create application session</td>
</tr>
<tr>
<td></td>
<td>...communicate...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INIT-COMM</td>
<td>Initialize communications</td>
</tr>
<tr>
<td></td>
<td>SEND-REQUEST</td>
<td>Send request module</td>
</tr>
<tr>
<td></td>
<td>RECV-RESPONSE</td>
<td>Receive response module</td>
</tr>
<tr>
<td>DBPSTOP</td>
<td>INIT-COMM</td>
<td>Initialize communications</td>
</tr>
<tr>
<td></td>
<td>DELETE-APPLICATION</td>
<td>Delete application session</td>
</tr>
<tr>
<td></td>
<td>DELETE-CONTROL</td>
<td>Delete control session</td>
</tr>
</tbody>
</table>

**SPP UTILITIES**

SPP contains two primary utilities which are useful in conjunction with the protocol operation. The two available utilities are tracing and performance monitoring. "Tracing" refers to a map containing detailed data transmission information including snapshots of the PCB Vector and Control/Application PCB's. The entire handshaking sequence within SPP may be studied with the aid of the trace utility. "Performance Monitoring" refers to the collection of certain execution statistics during host-DBP transmissions. By monitoring the DBP, the software analyst may study both the effect of SPP on VMS and the elapsed time during host-DBP requests and responses.

Both utilities may be used within any of the three SPP layers. The depth of trace and performance information may, therefore, be set by the analyst if only a subset of the SPP operations require monitoring.
The Trace Utility

A trace facility has been designed into SPP so that all Host-DBP communications may optionally be monitored. The trace output may be re-directed to any logical output unit including the terminal, if desired. Tracing may be accomplished by using the following two routines:

1. TRACE-START (UNIT) where UNIT = logical output file unit
2. TRACE-STOP

Snapshots of the PCB Vector and PCB are displayed on the trace output to aid the analyst. Appendix B displays all communications that transpire during the "CREATE-CONTROL" and "CREATE-APPLICATION" procedures (activated when DBP-START is called). For further information on interpreting the trace see the DBP Operations Manual (4).

The Performance Monitoring Utility

The analyst may wish to invoke the performance monitoring facility when using the other routines. The statistics that are currently monitored are listed below:

1. Elapsed Clock time
2. Elapsed VAX CPU time
3. Number of VMS buffered I/O requests
4. Number of VMS direct I/O requests
5. Number of VMS page faults

The following two routines may be used to obtain the above statistics:

1. PERFORM-START
2. PERFORM-STOP (CLOCK,CPU,BIO,DIO,PAGE) - where each argument directly corresponds to each item listed above.

SPP is currently fully operational using a 9600 baud physical link to the DBP service port. SPP is limited in that only one host may be used at any one time. It should be realized, however, that several host application sessions may be instantiated permitting multiple host simulation studies if desired.
In the future, Intel is planning on supporting the Ethernet link between multiple hosts and the DBP. The extensive host link protocol (5) (corresponding to the recent ISO protocol standard) will be used with Ethernet. The Ethernet implementation will permit fast DBP access which will be essential for multiple-user and embedded DBMS applications.

CONCLUDING REMARKS

SPP is to be used as the bottom layer of a stack of Data Base Processor (DBP) tools. The tools currently under development by the author to utilize the DBP functions are known collectively as the HILDA system. SPP has been implemented such that it may be separated from the HILDA system for use in another research effort.

The construction of a machine-independent protocol was considered important since the data base machine may be connected to a wide variety of hosts. The essential machine dependencies in SPP are clearly marked to aid the implementor in a non-DEC computer environment.

The functional, layered design of SPP supports the concept of extensibility so that an individual may easily make modifications and enhancements to the existing implementation.
REFERENCES

1. DBP DBMS Reference Manual. Intel Corporation, Austin, TX, Revision 001, Order No. 222100-001, August 1982.


5. DBP Host Link Reference Manual. Intel Corporation, Austin, TX, Revision 001, Order No. 222102-001, August 1982.
APPENDIX A - SPP Source

SPP has been implemented using VAX VMS FORTRAN 77. The 'SPP' program module specifies implementation notes which refer to certain computer dependencies of SPP. Subroutines which contain at least one source of VAX/VMS machine dependence are flagged with '*** MACHINE DEPENDENT ***' at the head of the routine.
PROGRAM SPP
C=============================================  
C PURPOSE :  
C 'SPP' IS A SERVICE PORT PROTOCOL TO BE USED IN  
C ACCESSING THE INTEL DBP  
C ARGUMENTS :  
C NONE  
C DIAGNOSTIC TRACE OPTION FOR PROTOCOL :  
C USE TRACE_START AND TRACE_STOP  
C PERFORMANCE MONITORING OPTION :  
C USE PERFORM_START AND PERFORM_STOP  
C SPP FUNCTIONAL COMPONENTS :  
C PROGRAMS  
C DBP_START - USED TO START I/O WITH THE DBP.  
C SPP - THIS PROGRAM IS JUST A SAMPLE PROGRAM  
C WRITTEN TO SHOW THE CORRECT FORM  
C FOR SPP OPERATION.  
C DBP_STOP - USED TO END I/O WITH THE DBP.  
C PROTOCOL SUBROUTINES :  
C INIT_COMM - INITIALIZE COMMUNICATIONS WITH DBP  
C END_COMM - END COMMUNICATIONS WITH DBP  
C CREATE_CONTROL - CREATE A DBP CONTROL SESSION  
C DELETE_CONTROL - DELETE THE DBP CONTROL SESSION  
C CREATE_APPLICATION - CREATE A DBP APPLICATION SESSION  
C DELETE_APPLICATION - DELETE THE DBP APPLICATION SESSION  
C RECVR_RESPONSE - RECEIVE A DBP RESPONSE  
C SEND_REQUEST - SEND A REQUEST TO THE DBP  
C READ_BLOCK - READ A DATA BLOCK FROM THE DBP  
C WRITE_BLOCK - WRITE A DATA BLOCK TO THE DBP  
C Q_INPUT - RECEIVE A BYTE BUFFER FROM THE DBP  
C Q_OUTPUT - SEND A BYTE BUFFER TO THE DBP  
C LOW16 - RETURN LOW ORDER BYTE FROM 16-BIT WORD  
C LOW32 - RETURN LOW ORDER BYTE FROM 32-BIT WORD  
C HIGH16 - RETURN HIGH ORDER BYTE FROM 16-BIT WORD  
C HIGH32 - RETURN HIGH ORDER BYTE FROM LOWER-HALF  
C OF 32-BIT WORD  
C GLUE - RETURN A 16-BIT WORD FORMED FROM 2 BYTES  
C UTILITY SUBROUTINES :  
C TRACK - IF TRACE MODE HAS BEEN ENABLED, DISPLAY THE  
C TWO DATA STRUCTURE FORMATS( PCB &  
C PCB VECTOR )  
C TRACE_START - ENABLE TRACE MODE  
C TRACE_STOP - DISABLE TRACE MODE
C PERFORM_START - ENABLE PERFORMANCE TRACING
C PERFORM_STOP  - DISABLE PERFORMANCE TRACING

C MACHINE DEPENDENCIES :
C
C THIS SOURCE TEXT REPRESENTS A TESTED VAX/VMS
C VERSION OF SPP.
C SPP HAS BEEN IMPLEMENTED SO THAT THE MACHINE
C DEPENDENCIES INHERENT WITHIN THE SOURCE TEXT
C ARE CLEARLY MARKED TO AID THE IMPLEMENTOR IN
C A NON-DEC COMPUTER ENVIRONMENT.
C
C THE FOLLOWING IS A LIST OF THINGS TO WATCH OUT FOR
C IF A NON-DEC MACHINE IS BEING USED :
C
1.) THE FOLLOWING ROUTINES CONTAIN VMS MACRO CALLS
   WHICH ARE USED MAINLY FOR TTY I/O PURPOSES :

   ROUTINE            DEPENDENCIES
   INIT_COMM          LIB$CRC_TABLE, SYS$ASSIGN
   Q_INPUT            SYS$QIOW
   Q_OUTPUT           SYS$QIOW
   READ_BLOCK         LIB$CRC
   WRITE_BLOCK        LIB$CRC
   END_COMM           SYS$DASSGN

   WHERE:

   LIB$CRC_TABLE - Initialize a table for further CRC16
              calculations
   LIB$CRC    - Calculate CRC16 for a given ASCII string
   SYS$ASSIGN - Assign an I/O channel
   SYS$DASSGN - De-assign an I/O channel
   SYS$QIOW   - Block I/O routine for serial I/O

   THE TYPES OF FUNCTIONS PRESENT WITHIN THESE
   ROUTINES IS USUALLY FOUND WITHIN MOST OPERATING
   SYSTEM SERVICE MANUALS.

2.) HEXADECIMAL VALUES FOR THE VAX ARE SPECIFIED AS
   FOLLOWS :

   'DE'X  'FF'X  etc.

   THIS REPRESENTATION MAY DIFFER ON ANOTHER COMPUTER.

3.) DATA TYPE 'BYTE' - ON THE VAX, THE MOST NATURAL WAY
   TO REPRESENT PURE BYTE STREAMS IS USING THE DATA TYPE
   'BYTE'. ON OTHER MACHINES, ONE MAY USE 'LOGICAL#1' OR
   'CHARACTER#1'. KEEP IN MIND, HOWEVER, THAT CHARACTER
   DATA IS GENERALLY STORE DIFFERENTLY(VMS CALLS THIS
   A DESCRIPTOR TYPE).

4.) IDENTIFIER LENGTHS - THE FORTRAN VARIABLE NAME LENGTHS ARE
LONGER THAN MAY BE SUPPORTED WITH SOME FORTRAN COMPILERS. 
THEY ARE LONG TO AID IN THE READING AND COMPREHENSION OF 
THE SOURCE.

5.) 'INCLUDE' STATEMENT - MOST FORTRANS SUPPORT A METHOD FOR 
INCLUDING/INSERTING A DISK FILE WITHIN THE SOURCE PRIOR 
TO COMPILATION.

DATE:

APRIL 12, 1983

AUTHOR:

PAUL A. FISHWICK
KENTRON TECHNICAL CENTER
3221 NORTH ARMISTEAD RD.
HAMPTON, VA. 23666
( 804 )-865-3195

NOTE: THIS IS AN EXAMPLE USE OF 'SPP', THE USER MUST 
HAVE STARTED COMMUNICATIONS BY ACTIVATING THE PROGRAM 
'DBPSTART' PRIOR TO THIS. THE FOLLOWING SET OF BYTES 
REPRESENTS THE CONCEPTUAL 'DEFINE DATABASE <TESTING>' 
DBP COMMAND. THE DIAGNOSTIC AND PERFORMANCE TRACING 
OPTIONS HAVE BEEN UTILIZED,

CALL TRACE_START( 9 )
CALL INIT_COMM 
MODULE(1) = '60'X 
MODULE(2) = '07'X 
MODULE(3) = '54'X 
MODULE(4) = '45'X 
MODULE(5) = '53'X 
MODULE(6) = '54'X 
MODULE(7) = '49'X 
MODULE(8) = '4E'X 
MODULE(9) = '47'X 
MODULE(10)= 'FF'X 
MODULE(11)= '00'X 
CALL PERFORM_START 
CALL SEND_REQUEST(MODULE,11,1,1,1) 
CALL RECV_RESPONSE(MODULE,NBYTES_RECV,1,MORE_TO_COME) 
CALL PERFORM_STOP( CLOCK,CPU,BIO,DIO,PAGEF ) 
CALL TRACE_STOP 
CALL EXIT 
END
COMMON FOR SPP (SERVICE PORT PROTOCOL)

BYTE Bytes(1024), Block(1024), Module(1024)
BYTE Module2(1024)
INTEGER BASE, OFFSET, IOSB(4), NBYTES, NBYTES_RECV
INTEGER TTY_CHANNEL
INTEGER Status, CRC_Table(16), CRC
CHARACTER STRING(512)
COMMON/CRC_COM/ CRC, CRC_Table
COMMON/COMM/ TTY_CHANNEL

LOGICAL MORE_TO_COME

SYSTEM SERVICE PARAMETERS

READ PARAMETERS
PARAETER IO$M_NOECHO = '00000040'X
PARAETER IO$M_PURGE = '00000800'X
PARAETER IO$M_TIMED = '00000880'X
PARAETER IO$TTYREADALL = '0000003A'X

STATUS INDICATORS
PARAETER SS$NORMAL = '00000001'X

WRITE PARAMETERS
PARAETER IO$WRITEVBLK = '00000030'X

DEBUG(TRACE) VARIABLES

INTEGER UNIT
LOGICAL DEBUG
COMMON/TRACE_COM/ DEBUG, UNIT
DATA DEBUG/ .FALSE. /
PROGRAM DBP_START
C======================================================
C
C PURPOSE :
C
C
C START OPERATIONS FOR THE DBP
C THIS INCLUDES ALLOCATING THE CHANNEL TO
C BE USED FOR HOST <-> DBP COMMUNICATIONS
C
C
C ARGUMENTS :
C
C NONE
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C APPLICATION
C
C DATE :
C
C APRIL 12, 1983
C
C======================================================
INCLUDE 'SPPCOM.TXT'
C
C SET UP COMMUNICATIONS
C
PRINT *, '** START DBP COMMUNICATIONS **'
CALL TRACE_START( 9 )
CALL INIT_COMM
C
C CREATE CONTROL APPLICATION SESSIONS
C
PRINT *, '** CREATING CONTROL SESSION **'
CALL CREATE_CONTROL
PRINT *, '** CREATING APPLICATION SESSION **'
CALL CREATE_APPLICATION
C
CALL TRACE_STOP
CALL EXIT
END
PROGRAM DBP_STOP
C=====================================================================
C
C PURPOSE:
C
C STOP OPERATIONS FOR THE DBP
C
C ARGUMENTS:
C
C NONE
C
C PROTOCOL:
C
SERVICE PORT
C
LAYER:
C
APPLICATION
C
C DATE:
C
APRIL 12, 1983
C
C=====================================================================
INCLUDE 'SPPCOM.TXT'
C
C SET UP COMMUNICATIONS
C
PRINT **, ** START DBP COMMUNICATIONS **'
CALL TRACE_START( 9 )
CALL INIT_COMM

C DELETE THE APPLICATION SESSION
C AND CONTROL SESSION
C ( TERMINATE DBP )
C
PRINT **, ** DELETING THE APPLICATION SESSION **'
CALL DELETE_APPLICATION
C
PRINT **, ** DELETING THE CONTROL SESSION **'
CALL DELETE_CONTROL
C

CALL TRACE_STOP
CALL EXIT
END
SUBROUTINE INIT_COMM

C======================================================
C
C *** MACHINE DEPENDENT ***
C
PURPOSE :
C
INITIALIZE COMMUNICATIONS PARAMETERS PRIOR TO ACTUALLY
TRANSMITTING DATA BACK AND FORTH
C
ARGUMENTS :
C
NONE
C
PROTOCOL :
C
SERVICE PORT
C
LAYER :
C
APPLICATION
C
DATE :
C
APRIL 12, 1983
C

C======================================================
C
INCLUDE 'SPPCOM.TXT'
INTEGER*4 SYSSASSIGN

IF( DEBUG ) WRITE( UNIT,5 )
  FORMAT( ' ** Initialize iDBP Communications **' )
C
INITIALIZE A CRC-16 TABLE FOR ERROR DETECTION
( THE VAX 'CRC' MACHINE INSTRUCTION IS USED )
C
CALL LIB$CRC_TABLE( '120001'O,CRC_TABLE )
C
ASSIGN AN I/O CHANNEL USING A TTY PORT
C
STATUS = SYSSASSIGN( 'REMOTE',TTY_CHANNEL, )
IF( STATUS.NE.SS$NORMAL ) THEN
  WRITE( UNIT,300 ) STATUS
  FORMAT( 'Error, unable to assign the DBP I/O Channel/',
         ' Status is ',z8,' See : INIT_COMM' )
ENDIF
C
SEND A CONTROL-C TO FLUSH THE TYPE-AHEAD BUFFER
C AND INITIALIZE DBP COMMUNICATIONS
C
BYTES(1) = '03'X
CALL Q_OUTPUT( BYTES,1 )
NBYTES_RECV = 16
CALL Q_INPUT( BYTES,NBYTES_RECV )
RETURN
END
SUBROUTINE END_COMM
C============================================================
C
C *** MACHINE DEPENDENT ***
C
C PURPOSE :
C
C END COMMUNICATIONS TO THE DBP. DEASSIGN CHANNEL.
C
C ARGUMENTS :
C
C NONE
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C APPLICATION
C
C DATE :
C
C APRIL 12, 1983
C
C============================================================
C
INCLUDE 'SPPCOM.TXT'
INTEGER*4 SYS$DASSGN
C
C DEASSIGN THE PREVIOUSLY ASSIGNED CHANNEL
C
STATUS = SYS$DASSGN(TTY_CHANNEL)
IF( STATUS.NE.SS$NORMAL ) THEN
   WRITE( UNIT,100 ) STATUS
100   FORMAT(' Error, unble to de-assgin the DBP Channel/',
 X       ' Status is ',Z8,' See: END_COMM' )
ENDIF
RETURN
END
SUBROUTINE CREATE_CONTROL

C=====================================================================
C
C PURPOSE:
C
C CREATE A CONTROL SESSION
C NOTE: THIS IS THE FIRST FUNCTION TO BE PERFORMED
C TO ACCESS THE DBP, THE 'MONITOR' BUTTON MUST
C BE PUSHED PRIOR TO CALLING THIS ROUTINE.
C
C ARGUMENTS:
C
C NONE
C
C PROTOCOL:
C
C SERVICE PORT
C
C LAYER:
C
C APPLICATION
C
C DATE:
C
C APRIL 12, 1983
C
C=====================================================================

INCLUDE 'SPPCOM.TXT'
INTEGER*2 BASE_CTRL, OFFSET_CTRL

IF ( DEBUG ) WRITE ( UNIT, 5 )
5 FORMAT ( ' ** Create Control Session ** ', X 'DBP Status is ', X 'z2'h' )

C READ THE PCB ADDRESS VECTOR
C
BASE = 'EE0C'X
OFFSET = 0
CALL READ_BLOCK ( BLOCK, 10, NBYTES_RECV, BASE, OFFSET )
IF ( DEBUG ) CALL TRACK ( BLOCK, 0 )
CALL GLUE ( BLOCK(3), BLOCK(4), OFFSET_CTRL )
CALL GLUE ( BLOCK(5), BLOCK(6), BASE_CTRL )

C READ THE CONTROL SESSION PCB
C
CALL READ_BLOCK ( BLOCK, 43, NBYTES_RECV, BASE_CTRL, OFFSET_CTRL )
IF ( DEBUG ) CALL TRACK ( BLOCK, 1 )
IF ( BLOCK(15).EQ.4 ) THEN
C HOST TO SEND 'ENABLE SERVICE PORT'
BLOCK(16) = '11'X
CALL WRITE_BLOCK ( BLOCK, 43, BASE_CTRL, OFFSET_CTRL )
IF ( DEBUG ) CALL TRACK ( BLOCK, 1 )
ELSE
WRITE ( UNIT, 100 ) BLOCK(15)
100 format ( ' Error, DBP's Wait on Enable is not set, '/,
X ' DBP Status is ', X 'z2'h' )
ENDIF
C
C RETURN CONTROL TO THE DBHS
C
BYTES(1) = '47'X
BYTES(2) = '0D'X
CALL Q_OUTPUT( BYTES,2 )
NBYTES_RECV = 29
CALL Q_INPUT( BYTES,NBYTES_RECV )
RETURN
END
SUBROUTINE CREATE_APPLICATION
C======================================================================
C
C PURPOSE :
C
C CREATE AN APPLICATION SESSION
C
C ARGUMENTS :
C
C NONE
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C APPLICATION
C
C DATE :
C
C APRIL 12, 1983
C======================================================================
C
INCLUDE 'SPPCOM.TXT'
C
IF( DEBUG ) WRITE( UNIT, 5 )
5
FORMAT( ' ** Create Application Session **' )
C
PERFORM 'CREATE APPLICATION SESSION'
C
MODULE(1) = 'E4'X
MODULE(2) = '01'X
MODULE(3) = 'FE'X
MODULE(4) = 'FF'X
MODULE(5) = '00'X
C
CALL SEND_REQUEST( MODULE, 5, 0, 1, 0 )
C
RECEIVE THE APPLICATION #
C
CALL RECEV_RESPONSE( MODULE, NBYTES_RECV, 0, MORE_TO_COME )
IF( DEBUG ) THEN
   WRITE( UNIT, 200 ) ( MODULE(I), I=1,NBYTES_RECV )
200
   FORMAT( ' ** Create Application Response **'/,
                     16(I1X,Z2.2) )
ENDIF
C
RETURN CONTROL TO THE DBMS
C
BYTES(1) = '47'X
BYTES(2) = '0D'X
CALL Q_OUTPUT( BYTES, 2 )
NBYTES_RECV = 29
CALL Q_INPUT( BYTES, NBYTES_RECV )
C
RETURN
END
SUBROUTINE DELETE_CONTROL
C===================================================================
C PURPOSE :
C DELETE A CONTROL SESSION
C NOTE : THIS IS THE LAST FUNCTION TO BE PERFORMED
C WHEN THE DBP IS TO BE STOPPED
C ARGUMENTS :
C NONE
C PROTOCOL :
C SERVICE PORT
C LAYER :
C APPLICATION
C DATE :
C APRIL 12, 1983
C===================================================================
C INCLUDE 'SPPCOM.TXT'
C IF ( DEBUG ) WRITE ( UNIT ,5)
5 FORMAT ( ' ** Delete Control Session **' )
C PERFORM 'TERMINATE DBP'
C MODULE (1)='ED'X
MODULE (2)='FF'X
MODULE (3)='00'X
C CALL SEND_REQUEST ( MODULE ,3 ,0 ,1 ,0 )
C RECEIVE THE 'TERMINATE DBP' RESPONSE
C CALL RECV_RESPONSE ( MODULE ,NBYTES_RECV ,O ,MORE_TO_COME )
C RETURN
END
SUBROUTINE DELETE_APPLICATION
C==============================================
C
C PURPOSE :
C
C DELETE AN APPLICATION SESSION
C
C ARGUMENTS :
C
C NONE
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C APPLICATION
C
C DATE :
C
C APRIL 12, 1983
C
C==============================================
INCLUDEl 'SPPCOM.TXT'
INTEGER*2 BASE_APP, OFFSET_APP
C
IF (DEBUG) WRITE (UNIT,5)
5 FORMAT( ' ** Delete Application Session **'
C
C READ THE PCB ADDRESS VECTOR
C
BASE = 'EEOC'X
OFFSET = 0
CALL READ_BLOCK ( BLOCK, 10, NBYTES_RECV, BASE, OFFSET )
IF (DEBUG) CALL TRACK ( BLOCK, 0 )
CALL GLUE ( BLOCK (7), BLOCK (8), OFFSET_APP )
CALL GLUE ( BLOCK (9), BLOCK (10), BASE_APP )
C
C CHECK THE INDEX FIELD FOR POSSIBLE ERRORS
C
IF ( (BLOCK (1), GE, 'A0'X), AND,
X (BLOCK (1), LE, 'DF'X ) ) THEN
WRITE (UNIT, 100 ) BLOCK (1)
100 FORMAT ( ' Error, Couldn't Delete Application Session',/
X ' Index Field(low) is ', z2,'h' )
RETURN
ENDIF
C
C READ THE APPLICATION SESSION PCB
C
CALL READ_BLOCK ( BLOCK, 43, NBYTES_RECV, BASE_APP, OFFSET_APP )
IF (DEBUG) CALL TRACK ( BLOCK, 1 )
IF ( BLOCK (15), EQ, 7 ) THEN
C HOST SEND 'OK FIN'
BLOCK (16) = '05'X
CALL WRITE_BLOCK ( BLOCK, 43, BASE_APP, OFFSET_APP )
IF (DEBUG) CALL TRACK ( BLOCK, 1 )
ELSE
WRITE (UNIT, 200 ) BLOCK (15)
200 FORMAT ( ' Error, Application Session cannot be deleted',/,
C X 'DBP Status is 'z2'h' )
ENDIF
C RETURN CONTROL TO THE DBMS
C
BYTES(1) = '47'X
BYTES(2) = '0D'X
CALL Q_OUTPUT( BYTES,2)
NBYTES_RECV = 29
CALL Q_INPUT( BYTES,NBYTES_RECV)
RETURN
END
SUBROUTINE RECV_RESPONSE( MODULE, TOTAL_BYTES, PCBTYPE, X MORE_TO_COME )
C==============================================================================
C PURPOSE :
C RECEIVE RESPONSE MODULE FROM THE DBP
C ARGUMENTS :
C MODULE - RECEIVED RESPONSE MODULE
C NBYTES - # OF BYTES IN RESPONSE MODULE RECEIVED
C PCBTYPE - TYPE OF PCB TO RECEIVE RESPONSE MODULE
C = 0 --> CONTROL PCB
C = 1 --> APPLICATION PCB
C MORE_TO_COME - TRUE, IF THERE IS MORE DATA TO BE RECEIVED
C AFTER THIS ROUTINE HAS BEEN CALLED
C - FALSE, IF ALL DATA HAS BEEN RECEIVED FROM
C THE DBP
C
C PROTOCOL :
C SERVICE PORT
C LAYER :
C APPLICATION
C DATE :
C APRIL 12,1983
C==============================================================================
C INCLUDE 'SPPCOM.TXT'
INTEGER*2 NSEGMENTS,BUFFER1_LENGTH,BUFFER2_LENGTH,PCBTYPE
INTEGER*2 BUFFER1_BASE,BUFFER2_BASE,BUFFER1_OFFSET
INTEGER*2 BUFFER2_OFFSET,TOTAL_BYTES
C
C GET PCB ADDRESS VECTOR
C IF( DEBUG ) WRITE( UNIT,5 )
5 FORMAT(' ** Receive Response **')
50 TOTAL_BYTES = 0
60 BASE = 'EEOC'X
OFFSET = 0
CALL READ_BLOCK( BLOCK,10,NBYTES_RECV,BASE,OFFSET )
IF( DEBUG ) CALL TRACK( BLOCK,0 )
C
C LOOK AT THE INDEX FIELD
C IF( (BLOCK(1),GE,'A0'X),AND,
X (BLOCK(1),LE,'DF'X) THEN
WRITE( UNIT,100 ) BLOCK(1)
100 FORMAT(' Error in RECV_RESPONSE, Index(low) is ',Z2,'h')
RETURN
ELSE IF ( (BLOCK(1).EQ.'FF')AND, 
X (BLOCK(2).EQ.'FF') ) THEN
  IF( DEBUG ) WRITE( UNIT,125 )
125 FORMAT(' Error, iDBP is suspended. Index is FFFFh' )
  GO TO 9999
ENDIF

C RECEIVE RESPONSE USING CONTROL OR APPLICATION PCB ?
C
  IF( PCBTYPE.EQ.0 ) THEN
    CALL GLUE( BLOCK(3),BLOCK(4),OFFSET )
    CALL GLUE( BLOCK(5),BLOCK(6),BASE )
    ELSE
      CALL GLUE( BLOCK(7),BLOCK(8),OFFSET )
      CALL GLUE( BLOCK(9),BLOCK(10),BASE )
    ENDIF
    CALL READ_BLOCK( BLOCK,43,NBYTES_RECV,BASE,OFFSET )
    IF( DEBUG ) CALL TRACK( BLOCK,1 )
  C
  C TEST THE DBP STATUS FIELD, FIRST
C
  IF( BLOCK(15).EQ.7 ) THEN
    C
    C UPDATE THE PCB
C
    BLOCK( 16 ) = 1
    CALL WRITE_BLOCK( BLOCK,43,BASE,OFFSET )
    IF( DEBUG ) CALL TRACK( BLOCK,1 )
  C
  C RETURN CONTROL TO DBMS
C
    BYTES(1) = '47'X
    BYTES(2) = '00'X
    CALL O_OUTPUT( BYTES,2 )
    NBYTES_RECV = 29
    CALL O_INPUT( BYTES,NBYTES_RECV )
    TOTAL_BYTES = 0
    MORE_TO_COME = .FALSE.
    GO TO 9999
  ENDIF
C
  C READY TO RECEIVE SEGMENT(S)
C
  NSEGMENTS = BLOCK( 31 )
C
  C RECEIVE THE FIRST BUFFER( SEGMENT )
    CALL GLUE( BLOCK(32),BLOCK(33),BUFFER1_OFFSET )
    CALL GLUE( BLOCK(34),BLOCK(35),BUFFER1_BASE )
    CALL GLUE( BLOCK(36),BLOCK(37),BUFFER1_LENGTH )
    TOTAL_BYTES = TOTAL_BYTES + BUFFER1_LENGTH
    CALL READ_BLOCK( MODULE(1),BUFFER1_LENGTH,NBYTES_RECV, 
X BUFFER1_BASE,BUFFER1_OFFSET )
  C
  C RECEIVE THE SECOND BUFFER( SEGMENT ), IF ANY
    IF( NSEGMENTS.LT.2 ) GO TO 200
    CALL GLUE( BLOCK(38),BLOCK(39),BUFFER2_OFFSET )
    CALL GLUE( BLOCK(40),BLOCK(41),BUFFER2_BASE )
    CALL GLUE( BLOCK(42),BLOCK(43),BUFFER2_LENGTH )
    IF( BUFFER2_LENGTH.GT.0 ) CALL READ_BLOCK( MODULE(TOTAL_BYTES+1), 
X BUFFER2_LENGTH,NBYTES_RECV,BUFFER2_BASE,BUFFER2_OFFSET )
    TOTAL_BYTES = TOTAL_BYTES + BUFFER2_LENGTH
  C
  C UPDATE THE PCB
C
26
BLOCK(16) = 1
CALL WRITE_BLOCK(BLOCK,43,BASE,OFFSET)
IF(DEBUG) CALL TRACK(BLOCK,1)

RETURN CONTROL TO DBMS SOFTWARE

BYTES(1) = '47'X
BYTES(2) = '0D'X
CALL Q_OUTPUT(BYTES,2)
NBYTES_RECV = 29
CALL Q_INPUT(BYTES,NBYTES_RECV)

C ARE ALL MODULES READ FROM THE DBP?
C IF NOT, FLAG THE CALLER

IF(BLOCK(15).EQ.6) THEN
  MORE_TO_COME = .FALSE.
  IF(DEBUG) WRITE(UNIT,300)
  300 FORMAT(' ** All data has been received **')
ELSE
  MORE_TO_COME = .TRUE.
  IF(DEBUG) WRITE(UNIT,400)
  400 FORMAT(' ** There is more data to come **')
ENDIF

DONE READING ALL RESPONSES

ADJUST THE MODULE ARRAY(RETURNED RESPONSE)
TO GET RID OF THE HEADER BYTES

DO 500 I = 1,TOTAL_BYTES
  500 MODULE(I) = MODULE(I+4)
  TOTAL_BYTES = TOTAL_BYTES - 4

RETURN
END
SUBROUTINE SEND_REQUEST( MODULE,NBYTES_SENT,PCBTYPE,
X
APPLICATION_ID,REQUEST_ID )
C======================================================================
C PURPOSE :
C
C SEND REQUEST MODULE TO THE DBP
C
C ARGUMENTS :
C
C MODULE - REQUEST MODULE
C NBYTES - # OF BYTES IN REQUEST MODULE TO SEND
C PCBTYPE - TYPE OF PCB TO RECEIVE RESPONSE MODULE
C
C = 0 --> CONTROL PCB
C = 1 --> APPLICATION PCB
C
C APPLICATION_ID - ARBITRARILY ASSIGNED HOST APPLICATION ID
C
C REQUEST_ID - THIS IS THE ID # OF THE SESSION MAKING
C THE REQUEST. WHEN AN APPLICATION IS FIRST
C CREATED, THE CONTROL SESSION( = 0 ) ID IS
C THE 'REQUEST_ID'. AFTER THAT, THE APPLICATION
C ID ISSUING THE REQUEST IS THE 'REQUEST_ID'.
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C APPLICATION
C
C DATE :
C
C APRIL 12,1983
C
C======================================================================
C
INCLUDE 'SPPCOM.TXT'
BYTE BUFFER1( 512 ),BUFFER2( 512 ),TEMP_BYTE
INTEGER PCBTYPE,HSEGMENTS,BUFFER1_LENGTH,BUFFER2_LENGTH
INTEGER BUFFER1_BASE,BUFFER1_OFFSET
INTEGER BUFFER2_BASE,BUFFER2_OFFSET
INTEGER TOTAL_SENT,NBYTES_SENT
INTEGER APPLICATION_ID,REQUEST_ID
C
C STICK IN HOST APPLICATION ID & SESSION ID
C
50 NBYTES = NBYTES_SENT
LEFTOVER_BYTES = .FALSE.,
DO 2 I = NBYTES,1,-1
2 MODULE(I+4) = MODULE(I)
CALL LOW32( APPLICATION_ID,TEMP_BYTE )
MODULE(1) = TEMP_BYTE
MODULE(2) = '00'X
CALL LOW32( REQUEST_ID,TEMP_BYTE )
MODULE(3) = TEMP_BYTE
MODULE(4) = '00'X
NBYTES = NBYTES + 4
C
C GET PCB ADDRESS VECTOR
C
70 IF ( DEBUG ) WRITE( UNIT,80 )
80 FORMAT( ' ** Send Request ** ')
BASE = 'EEOC'X
OFFSET = 0
CALL READ_BLOCK( BLOCK,10,NBYTES_RECV,BASE,OFFSET,MORE_TO_COME )
IF ( DEBUG ) CALL TRACK( BLOCK,0 )
C
C LOOK AT THE INDEX FIELD
C
IF ( ( BLOCK(1).GE.'AO'X).AND.
X ( BLOCK(1).LE.'DF'X ) THEN
WRITE( UNIT,100 ) BLOCK(1)
100 FORMAT( ' Error in SEND_REQUEST, Index(low) is ',z2,'h' )
RETURN
ELSE IF ( ( BLOCK(1).EQ.'FF'X).AND.
X ( BLOCK(2).EQ.'FF'X ) ) THEN
BYTES(1) = '03'X
CALL Q_OUTPUT( BYTES,1 )
NBYTES_RECV = 16
CALL Q_INPUT( BYTES,NBYTES_RECV )
GO TO 50
ENDIF
C
C SEND REQUEST USING CONTROL OR APPLICATION PCB ?
C
IF ( PCBTYPE.EQ.0 ) THEN
CALL GLUE( BLOCK(3),BLOCK(4),OFFSET )
CALL GLUE( BLOCK(5),BLOCK(6),BASE )
ELSE
CALL GLUE( BLOCK(7),BLOCK(8),OFFSET )
CALL GLUE( BLOCK(9),BLOCK(10),BASE )
ENDIF
CALL READ_BLOCK( BLOCK,43,NBYTES_RECV,BASE,OFFSET )
IF ( DEBUG ) CALL TRACK( BLOCK,1 )
C
C TEST THE DBP STATUS FIELD, FIRST
C
140 IF ( ( BLOCK(15).EQ.5).OR.( BLOCK(15).EQ.6) ) THEN
150 IF ( DEBUG ) WRITE( UNIT,150 ) BLOCK(15)
150 FORMAT( ' ** Warning ** ,
X ' ** Had to receive a response during ' ,
X ' this SEND_REQUEST/ ,
X ' iDBP Status is ',z2,'h' )
CALL RECV_RESPONSE( MODULE2,NBYTES_RECV,PCBTYPE,MORE_TO_COME )
IF ( MORE_TO_COME ) GO TO 140
ENDIF
C
C CAN SEND THE MODULE
C
NSEGMENTS = BLOCK( 31 )
C
C GO AHEAD AND TAKE CARE OF THE FIRST BUFFER
C
CALL GLUE( BLOCK(32),BLOCK(33),BUFFER1_OFFSET )
CALL GLUE( BLOCK(34),BLOCK(35),BUFFER1_BASE )
CALL GLUE( BLOCK(36),BLOCK(37),BUFFER1_LENGTH )
IF ( NBYTES.LT.BUFFER1_LENGTH ) THEN
LENGTH = NBYTES
ELSE
LENGTH = BUFFER1_LENGTH
ENDIF
LEFTOVER = NBYTES - LENGTH
DO 200 I = 1,LENGTH
200 BUFFER1(I) = MODULE(I)
C WRITE THE FIRST BUFFER
CALL WRITE_BLOCK( BUFFER1,LENGTH,BUFFER1_BASE,
                    BUFFER1_OFFSET )
TOTAL_SENT = LENGTH
C IF TWO SEGMENTS ARE REQUESTED, SEND THE OTHER BUFFER
C IF( NSEGMENTS.EQ.2 ) THEN
CALL GLUE( BLOCK(38),BLOCK(39),BUFFER2_OFFSET )
CALL GLUE( BLOCK(40),BLOCK(41),BUFFER2_BASE )
CALL GLUE( BLOCK(42),BLOCK(43),BUFFER2_LENGTH )
DO 300 I = 1,LEFTOVER
300 BUFFER2(I) = MODULE(I + LENGTH)
C WRITE THE SECOND BUFFER
CALL WRITE_BLOCK( BUFFER2,LEFTOVER,BUFFER2_BASE,
                    BUFFER2_OFFSET )
TOTAL_SENT = TOTAL_SENT + LEFTOVER
LEFTOVER = 0
ENDIF
C UPDATE THE PCB &
C SET 'REQUEST LENGTH' FIELD
C IF( LEFTOVER.GT.0 ) THEN
C SEND REQUEST
C BUFFER THIS REQUEST UNTIL THE REST OF THE
C REQUEST DATA CAN BE SENT
C BLOCK( 16 ) = 1
ELSE
C SEND REQUEST WITH EOM
C I.E. THE COMPLETED REQUEST IS SENT
C BLOCK( 16 ) = 3
ENDIF
CALL LOW16( TOTAL_SENT,BLOCK(29) )
CALL HIGH16( TOTAL_SENT,BLOCK(30) )
CALL WRITE_BLOCK( BLOCK,43,BASE,OFFSET )
IF( DEBUG ) CALL TRACK( BLOCK,1 )
C RETURN CONTROL TO DBMS SOFTWARE
C BYTES( 1 ) = '47'X
BYTES( 2 ) = '00'X
CALL Q_OUTPUT( BYTES,2 )
NBYTES_RECV = 29
CALL Q_INPUT( BYTES,NBYTES_RECV )
C CHECK IF THE HOST NEEDS TO SEND ANY
C LEFTOVER BYTES
C IF( LEFTOVER.GT.0 ) THEN
IF ( DEBUG ) WRITE ( UNIT, 600 ) NBYTES-LENGTH
   600   FORMAT (/ ** Process ',13,' leftover bytes **'/)
      DO 750 I = LENGTH+1, NBYTES
      750   MODULE ( I - LENGTH ) = MODULE ( I )
         NBYTES = NBYTES - LENGTH
         GO TO 70
   ENDFI

C
RETURN
END
SUBROUTINE READ_BLOCK( BLOCK, NBYTES, NBYTES_RECV, BASE, OFFSET )
C=============================================================================
C *** MACHINE DEPENDENT ***
C PURPOSE :
C READS DATA FROM THE DBP
C ARGUMENTS :
C BLOCK - DATA READ FROM DBP
C NBYTES - # OF BYTES READ FROM THE DBP
C BASE - BASE PART OF I/O ADDRESS
C OFFSET - OFFSET PART OF I/O ADDRESS
C PROTOCOL :
C SERVICE PORT
C LAYER :
C DATA LINK
C DATE :
C APRIL 12, 1983
C=============================================================================
C INCLUDE 'SPPCOM.TXT'
INTEGER*2 COUNT
BYTE LOWBYTE, HIGHBYTE
BYTE INIT(3)
DATA INITI '55'X,'52'X,'OD'X /
C INITIATE READ
C 50 IF( DEBUG ) WRITE( UNIT,55 )
55 FORMAT( '** Initiate a READ_BLOCK **' )
DO 75 I = 1,3
75 BYTES( I ) = INIT( I )
C SEND COUNT, OFFSET, AND BASE
C CALL LOW16( NBYTES, BYTES(4) )
CALL HIGH16( NBYTES, BYTES(5) )
CALL LOW16( OFFSET, BYTES(6) )
CALL HIGH16( OFFSET, BYTES(7) )
CALL LOW16( BASE, BYTES(8) )
CALL HIGH16( BASE, BYTES(9) )
WRITE( STRING,110 ) (BYTES(I),I=4,9 )
110 FORMAT( 6A1 )
CRC = LIB$CRC( CRC_TABLE,0,STRING(I:6) )
CALL LOW32( CRC, BYTES(10) )
CALL HIGH32( CRC, BYTES(11) )
C SEND THE BYTES
C CALL Q_OUTPUT( BYTES,11 )
C 32
C RECEIVE RESPONSE

NBYTES_RECV = NBYTES + 15
CALL Q_INPUT( BYTES,NBYTES_RECV )
IF( (BYTES(1),NE.,'55'X),OR,
  X (BYTES(2),NE.,'52'X),OR,
  X (BYTES(3),NE.,'OD'X),OR,
  X (BYTES(4),NE.,'0A'X) ) THEN
    IF(DEBUG) WRITE( UNIT,125 ) (BYTES(I),I=1,4)
125 FORMAT(' Error, Expected to find 55h,52h,0Dh,0Ah./
  ' Instead found 'z2',h',3('z2',h') )
    GO TO 50
ENDIF

C CHECK THE REMAINDER OF THE DATA BYTES

WRITE( STRING,150 ) (BYTES(I),I=5,10)
150 FORMAT( 6A1 )
CRC = LIB$CRC( CRC_TABLE,0,STRING(1:6) )

C CHECK CRC-1

CALL LOW32( CRC,LOWBYTE )
CALL HIGH32( CRC,HIGHBYTE )
IF( (BYTES(11),NE.,LOWBYTE),OR,
  X (BYTES(12),NE.,HIGHBYTE) ) THEN
  CRC'S DO NOT HATCH
    IF( DEBUG ) WRITE( UNIT,200 ) HIGHBYTE,LOWBYTE,
      X (BYTES(12)),BYTES(11)
200 FORMAT(' Error, CRC16 : /
  ' Host CRC( High,Low ) : 'z2',1x,z2/,
  ' DBP CRC( High,Low ) : 'z2',1x,z2/ )
    GO TO 50
ENDIF

C PROCESS REST OF DATA

CALL GLUE( BYTES(5),BYTES(6),NBYTES_RECV )
DO 400 I = 1,NBYTES_RECV+3
400 BLOCK(I) = BYTES(I+12)
WRITE( STRING,410 ) (BLOCK(I),I=1,NBYTES_RECV)
410 FORMAT( <NBYTES_RECV>A1 )
CRC = LIB$CRC( CRC_TABLE,0,STRING(1:NBYTES_RECV) )

C CHECK CRC-2

CALL LOW32( CRC,LOWBYTE )
CALL HIGH32( CRC,HIGHBYTE )
IF( (BLOCK(NBYTES_RECV+1),NE.,LOWBYTE),OR,
  X (BLOCK(NBYTES_RECV+2),NE.,HIGHBYTE) ) THEN
  IF( DEBUG ) WRITE( UNIT,500 ) HIGHBYTE,LOWBYTE,
    X BLOCK(NBYTES_RECV+2),BLOCK(NBYTES_RECV+1)
500 FORMAT(' Error, CRC16 : /
  ' Host CRC( High,Low ) : 'z2',1x,z2/,
  ' DBP CRC( High,Low ) : 'z2',1x,z2/ )
    GO TO 50
ENDIF

C SUCCESSFUL READ_BLOCK OPERATION
C
RETURN
END
SUBROUTINE WRITE_BLOCK( BLOCK, NBYTES, BASE, OFFSET )

C===================================================~==
C
C *** MACHINE DEPENDENT ***
C
C PURPOSE :
C WRITES DATA FROM THE HOST TO THE DBP
C
C ARGUMENTS :
C BLOCK - DATA TO BE WRITTEN TO THE DBP
C NBYTES - # OF BYTES IN 'BLOCK' TO BE SENT
C BASE - BASE PART OF I/O ADDRESS
C OFFSET - OFFSET PART OF I/O ADDRESS
C
C PROTOCOL :
C SERVICE PORT
C LAYER :
C DATA LINK
C
C DATE :
C APRIL 12, 1983
C
C======================================================
C
INCLUDE 'SPPCOM.TXT'
INTEGER*2 COUNT
BYTE INIT( 3 )
DATA INIT/ '55', '57', '0D' /
C
C INITIATE WRITE
C
50 IF( DEBUG ) WRITE( UNIT, 60 )
60 FORMAT( '** Initiate a WRITE_BLOCK **' )
DO 75 I = 1, 3
75 BYTES( I ) = INIT( I )
C
C SEND COUNT, OFFSET, AND BASE
C
CALL LOW16( NBYTES, BYTES(4) )
CALL HIGH16( NBYTES, BYTES(5) )
CALL LOW16( OFFSET, BYTES(6) )
CALL HIGH16( OFFSET, BYTES(7) )
CALL LOW16( BASE, BYTES(8) )
CALL HIGH16( BASE, BYTES(9) )
WRITE( STRING, 100 ) ( BYTES(I), I=4, 9 )
100 FORMAT( 6A1 )
CRC = LIB*CRC( CRC_TABLE, 0, STRING(1:6) )
CALL LOW32( CRC, BYTES(10) )
CALL HIGH32( CRC, BYTES(11) )
C
C SEND THE BYTES
C
CALL Q_OUTPUT( BYTES, 11 )
C
C RECEIVE ACKNOWLEDGMENT
C
WRITE_BLK.FOR; 25 5-JUL-1983 12:20

NBYTES_RECV = NBYTES + 15
CALL Q_INPUT( BYTES,NBYTES_RECV )
IF( (BYTES(1),EQ.,'55'X).AND.
X (BYTES(2),EQ.,'57'X).AND.
X (BYTES(3),EQ.,'0D'X).AND.
X (BYTES(4),EQ.,'0A'X) ) THEN
IF( BYTES(5),NE.,'06'X ) THEN
IF(DEBUG) WRITE( UNIT,200 ) BYTES(5)
200 FORMAT( 'Error, WRITE_BLOCK 1st Receive Ack./',
' Expecting to find 06h, instead found ',z2,'h' )
GO TO 50
ENDIF
ELSE
IF(DEBUG) WRITE( UNIT,300 ) (BYTES(I),I=1,4)
300 FORMAT( 'Error, WRITE_BLOCK 1st Receive Ack./',
' Expecting to find 55h,57h,0Dh,0Ah. ',
' Instead found ',z2,'h',3(',',z2,'h') )
GO TO 50
ENDIF
C
C SEND DATA
C
CRC = 0
IF( NBYTES.EQ.0 ) GO TO 650
C
C BUFFER THE CRC
C
WRITE( STRING,625 ) (BLOCK(I),I=1,NBYTES)
625 FORMAT( NBYTES )A1
CRC = LIB.CRC( CRC_TABLE,0,STRING(1:NBYTES) )
650 CALL LOW32( CRC,BLOCK(NBYTES+1) )
CALL HIGH32( CRC,BLOCK(NBYTES+2) )
CALL Q_OUTPUT( BLOCK,NBYTES+2 )
C
C RECEIVE ACKNOWLEDGEMENT
C
NBYTES_RECV = 2
CALL Q_INPUT( BYTES,NBYTES_RECV )
IF( BYTES(1),NE.,'06'X ) THEN
IF(DEBUG) WRITE( UNIT,700 ) BYTES(1)
700 FORMAT( 'Error, in WRITE_BLOCK 2nd Receive Ack./',
' Expecting 06h, instead found ',z2,'h' )
GO TO 50
ENDIF
C
C SUCCESSFUL WRITE_BLOCK OPERATION
C
RETURN
END
SUBROUTINE Q_INPUT( BYTES,NBYTES_RECV )
c======================================================
c*** MACHINE DEPENDENT ***

PURPOSE:

QUEUE A SEQUENCE OF BYTES TO THE INPUT CHANNEL
'Q_INPUT' WAITS UNTIL DATA APPEARS ON THE CHANNEL

ARGUMENTS:

BYTES - THE ARRAY SEQUENCE OF BYTES RECEIVED
NBYTES_RECV - THE NUMBER OF BYTES TO RECEIVE &
THE NUMBER OF ACTUAL BYTES RECEIVED

NOTE:

Q_INPUT WAITS FOR THE DBP TO SEND 'NBYTES_RECV' BYTES.
IF 'NBYTES_RECV' BYTES HAVE NOT BEEN SENT BY THE TIME
THAT THE TIME-OUT VALUE CURRENTLY 5 SECONDS ) HAS
OCCURRED, THE ROUTINE EXITS WITH THE DATA THAT WAS
RECEIVED.

PROTOCOL:

SERVICE PORT

LAYER:

PHYSICAL

DATE:

APRIL 12,1983

c======================================================

INCLUDE 'SPPCOM.TXT'
INTEGER*4 SYS$QOW,TERMINATOR(2),TIME_OUT
BYTE PRBYTES( 1024 ), MASK( 6 )

TIME_OUT = 1

SET UP THE TERMINATOR BYTES

TERMINATOR(1) = 0
TERMINATOR(2) = 0

INITIATE THE INPUT OPERATION
(WAIT FOR THE DBP TO SPEAK )

IOSB(2) = 0
STATUS = SYS$QOW( ,%VAL( TTY_CHANNEL ),
X %VAL(IO$_TTYREADALL+IO$M_NOECHO+IO$M_TIMED),
X IOSB(2),BYTES(1),%VAL(NBYTES_RECV),%VAL(5),TERMINATOR,,)
IF( STATUS.NE.SS$_NORMAL ) THEN
WRITE( UNIT,10 ) STATUS
INPUT.FOR;74  5-JUL-1983 12:20  Page 2

10 FORMAT( ' Error, Q_INPUT failure, '/,
X   ' Return Status is ',Z8 )
ENDIF
NBYTES_RECV = IOSB(2)
IF( NBYTES_RECV.EQ.0 ) THEN
  IF( TIME_OUT.EQ.10 ) THEN
    IF( DEBUG ) WRITE( UNIT,18 )
    FORMAT( ' --- Max Time Out 's Encountered ---' )
    RETURN
  ELSE
    RETURN
ENDIF
ENDIF

C RETURN TO GET INPUT ONCE MORE
C
IF( DEBUG ) WRITE( UNIT,20 ) TIME_OUT
20 FORMAT( ' --- Time Out # ',Z2, ' ---' )
TIME_OUT = TIME_OUT + 1
GO TO 5
ENDIF
ENDIF

C SET UP ASCII BYTES
C NOTE: NON-PRINTABLE CHARACTERS ARE DENOTED
C WITH A PERIOD( '2E'X )
C
DO 50 I = 1,NBYTES_RECV
  IF( (BYTES(I).LT.'20'X).OR.
X   (BYTES(I).GT.'7E'X)) THEN
    PRBYTES(I) = '2E'X
  ELSE
    PRBYTES(I) = BYTES(I)
  ENDIF
CONTINUE
WRITE( UNIT,100 ) NBYTES_RECV
100 FORMAT( ' == Q_INPUT == '//' # of bytes is ',I5,
X   '/, ' Byte Stream:' )
MULTIPLE16 = ( NBYTES_RECV/16 )*16
LEFTOVER = NBYTES_RECV - MULTIPLE16
IF( MULTIPLE16.GT.0 ) THEN
  DO 200 I = 1,MULTIPLE16,16
  WRITE( UNIT,150 ) (BYTES(I1),I1=I,I+15),(PRBYTES(I2),I2=I,I+15)
150 FORMAT(16(1X,Z2.2),2X,16A1)
CONTINUE
ENDIF
IF( LEFTOVER.GT.0 ) THEN
  WRITE( UNIT,250 ) (BYTES(I1),I1=MULTIPLE16+1,
X   MULTIPLE16+LEFTOVER),(PRBYTES(I2),I2=MULTIPLE16+1,
X   MULTIPLE16+LEFTOVER)
250 FORMAT( <LEFTOVER>(1X,Z2.2),<16-LEFTOVER>(3X),2X,
X   <LEFTOVER>A1 )
ENDIF
WRITE( UNIT,400 )
400 FORMAT(/)
ENDIF
RETURN
END
SUBROUTINE Q_OUTPUT( BYTES, NBYTES )
C======================================================
C
C *** MACHINE DEPENDENT ***
C
C PURPOSE :
C
C QUEUE A SEQUENCE OF BYTES TO THE OUTPUT TTY CHANNEL
C
C ARGUMENTS :
C
C BYTES - THE ARRAY( SEQUENCE ) OF BYTES TO BE TRANSFERRED
C NBYTES - # OF BYTES TO BE TRANSFERRED IN ARRAY 'BYTES'
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C DATA LINK
C
C DATE :
C
C APRIL 12, 1983
C
C======================================================
C
C INCLUDE 'SPPCOM.TXT'
INTEGER*4 SYSSQIOW
BYTE PRBYTES( 1024 )

C INITIATE THE OUTPUT OPERATION
C ( TALK TO THE DBP )
C
C IF( DEBUG ) THEN
C
C SET UP ASCII BYTES
C
DO 50 I = 1, NBYTES
IF( (BYTES(I),L.T., '20'X), OR,
X (BYTES(I),G.T., '7E'X)) THEN
PRBYTES(I) = '2E'X
ELSE
PRBYTES(I) = BYTES(I)
ENDIF
50 CONTINUE
C
C MULTIPLE16 = (NBYTES/16)*16
LEFTOVER = NBYTES - MULTIPLE16
IF( MULTIPLE16.GT.0 ) THEN
DO 200 I = 1, MULTIPLE16, 16
WRITE( UNIT, 90 ) (BYTES(II),II=I,I+15),(PRBYTES(I2),I2=I,I+15)
150 FORMAT( '16(1X,Z2.2),2X,16A1')
200 CONTINUE
ENDIF
IF( LEFTOVER.GT.0 ) THEN
WRITE( UNIT, 250 ) (BYTES(I1),I1=I, I+15), (PRBYTES(I2),I2=I, I+15)
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER, (PRBYTES(I2), I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
X MULTIPLE16+LEFTOVER), (PRBYTES(I2),I2=MULTIPLE16+1,
**QOUTPUT.FOR:22**  
5-JUL-1983 12:21

```
X MULTIPLE16+LEFTOVER
250   FORMAT(<LEFTOVER>(1X,Z2.2),<16-LEFTOVER>(3X),2X,
X   <LEFTOVER>32 )
ENDIF
WRITE( UNIT,300 )
300   FORMAT(/)
C
ENDIF
STATUS = SYS$QIOW( , ZVAL(TTY_CHANNEL),
X   ZVAL(IO$_WRITEVBLK),IOSB,,
X   BYTES(1),ZVAL(NBYTES),,ZVAL(0),, )
IF( STATUS.NE.SS$_NORMAL ) THEN
   WRITE( UNIT,400 ) STATUS
400   FORMAT( ' Error, Q_OUTPUT failure, ',/,
X   ' Return Status is ',ZB )
ENDIF
RETURN
END
```
SUBROUTINE LOW16( WORD16, LOWBYTE )
C======================================================================
C
C PURPOSE :
C
C RETURN LOW ORDER BYTE FROM 16 BIT WORD
C
C ARGUMENTS :
C
C WORD16 - 16 BIT WORD
C LOWBYTE - LOW ORDER 8 BITS
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C DATA LINK
C
C DATE :
C
C APRIL 12, 1983
C
C======================================================================
C
BYTE LOWBYTE, WORD16(2)
LOWBYTE = WORD16(1)
RETURN
END
SUBROUTINE LOW32( WORD32, LOWBYTE )
C======================================================
C PURPOSE :
C RETURN LOW ORDER BYTE FROM 32 BIT WORD
C ARGUMENTS :
C WORD32 - 32 BIT WORD
C LOWBYTE - LOW ORDER 8 BITS
C PROTOCOL :
C SERVICE PORT
C LAYER :
C DATA LINK
C DATE :
C APRIL 12, 1983
C======================================================
C
BYTE LOWBYTE, WORD32(4)
LOWBYTE = WORD32(1)
RETURN
END
SUBROUTINE HIGH16( WORD16,HIGHBYTE )
C=======================================================
C
C PURPOSE : 
C
C RETURN HIGH ORDER BYTE FROM 16 BIT WORD
C
C ARGUMENTS :
C
C WORD16 - 16 BIT WORD
C HIGHBYTE - HIGH ORDER 8 BITS
C
C PROTOCOL :
C
C SERVICE PORT
C LAYER :
C DATA LINK
C DATE :
C
C APRIL 12,1983
C
C=======================================================
C
BYTE HIGHBYTE,WORD16(2)
HIGHBYTE = WORD16(2)
RETURN
END
SUBROUTINE HIGH32( WORD32, HIGHBYTE )

C==================================
C
C PURPOSE :
C
C RETURN HIGH ORDER BYTE FROM LOWER HALF OF A
C 32-BIT WORD
C
C ARGUMENTS :
C
C WORD32 - 32 BIT WORD
C HIGHBYTE - HIGH ORDER 8 BITS
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C DATA LINK
C
C DATE :
C
C APRIL 12, 1983
C
C==================================
C
BYTE HIGHBYTE, WORD32(4)
HIGHBYTE = WORD32(2)
RETURN
END
SUBROUTINE GLUE( LOWBYTE, HIGHBYTE, GLUED )

C======================================================
C
C PURPOSE :
C
C   GLUE TWO BYTES TOGETHER TO FORM A 16-BIT WORD
C
C ARGUMENTS :
C
C   LOWBYTE   - LOW ORDER 8 BITS
C   HIGHBYTE  - HIGH ORDER 8 BITS
C   GLUED     - 16-BIT WORD
C
C PROTOCOL :
C
C   SERVICE PORT
C
C LAYER :
C
C   ALL
C
C DATE :
C
C   APRIL 12, 1983
C
C======================================================
BYTE LOWBYTE, HIGHBYTE, GLUED(2)
GLUED(1) = LOWBYTE
GLUED(2) = HIGHBYTE
RETURN
END
SUBROUTINE TRACE_START( TRACE_UNIT )

C=================================================================
C
C PURPOSE:
C
C INITIALIZE A FILE FOR DIAGNOSTIC TRACE OUTPUT
C
C ARGUMENTS:
C
C TRACE_UNIT - LOGICAL OUTPUT UNIT FOR TRACE INFORMATION
C
C PROTOCOL:
C
C SERVICE PORT
C
C LAYER:
C
C ALL : TRACE UTILITY
C
C DATE:
C
C APRIL 12, 1983
C
C=================================================================
C
INCLUDE 'SPPCOM.TXT'
C
OPEN A DEBUG FILE, IF WE ARE NOT TALKING
C TO THE TERMINAL
C
IF( UNIT.NE.6 ) OPEN( Unit=TRACE_UNIT,File='TRACE.DBP',
X STATUS='NEW' )
UNIT = TRACE_UNIT
DEBUG = .TRUE.
C
RETURN
END
SUBROUTINE TRACE_STOP
C=======================================================================
C
C PURPOSE :
C
C STOP THE TRACE OUTPUT
C
C ARGUMENTS :
C
C NONE
C
C PROTOCOL :
C
C SERVICE PORT
C
C LAYER :
C
C ALL : TRACE UTILITY
C
C DATE :
C
C APRIL 12, 1983
C
C=======================================================================
C
C     INCLUDE 'SPPCOM.TXT'
C
C     DEBUG = .FALSE.
C
C     RETURN
END
SUBROUTINE PERFORM_START
C================================:=====:c=============
C
C *** MACHINE DEPENDENT ***
C PURPOSE :
C START TRACKING THE FOLLOWING PERFORMANCE STATISTICS :
C 1. VAX CPU TIME ELAPSED
C 2. VAX CLOCK TIME ELAPSED
C 3. VAX BUFFERED I/O
C 4. VAX DIRECT I/O
C 5. VAX PAGE FAULT COUNT
C
C ARGUMENTS :
C NONE
C PROTOCOL :
C SERVICE PORT
C LAYER :
C ALL : PERFORMANCE UTILITY
C DATE :
C APRIL 12, 1983
C================================:=====:c=============
C
INTEGER*4 BUFIO,CPUTIME,DIO,PAGEF
INTEGER*4 BUFIO_ADR,CPUTIME_ADR,DIO_ADR,PAGEF_ADR
INTEGER*4 ZERO1,ZERO2,ZERO3,ZERO4,ZERO5
INTEGER*4 SYS$GETJP1,STATUS
INTEGER*2 LENGTH1,LENGTH2,LENGTH3,LENGTH4
COMMON/STATCOM/ CSETUP,CLOCK_TIME,BUFIO,CPUTIME,DIO,PAGEF
COMMON/JPICOM/ LENGTH1,BUFIO_CODE,BUFIO_ADR,ZERO1,
               X LENGTH2,CPUTIME_CODE,CPUTIME_ADR,ZERO2,
               X LENGTH3,DIO_CODE,DIO_ADR,ZERO3,
               X LENGTH4,PAGEF_CODE,PAGEF_ADR,ZERO4,ZERO5
DATA BUFIO_CODE/ 1036 /
DATA CPUTIME_CODE/ 1031 /
DATA DIO_CODE/ 1035 /
DATA PAGEF_CODE/ 1034 /
DATA LENGTH1,LENGTH2,LENGTH3,LENGTH4/4,4,4,4/
C INITIALIZE THE STATISTIC VARIABLES
C
CLOCK_TIME = SECNDS( 0.0 )
BUFIO_ADR = %LOC( BUFIO )
CPUTIME_ADR= %LOC( CPUTIME )
DIO_ADR = %LOC( DIO )
PAGEF_ADR = %LOC( PAGEF )
C GET THE PROCESS INFORMATION
C
STATUS = SYS$GETJPI(' ',LENGTH1,' ')
IF( STATUS.NE.1 ) WRITE( 6,100 ) STATUS
100 FORMAT( 'Error with SYS$GETJPI, status is ',Z8,'h' )
C
RETURN
END
SUBROUTINE PERFORM_STOP( NEW_CLOCK, NEW_CPU, NEW_BUFF, NEW_DIRECT, NEW_PAGE )

C======================================================
C
C *** MACHINE DEPENDENT ***
C
C PURPOSE:
C STOP THE TRACKING OF THE PERFORMANCE STATISTICS
C AND RETURN THE VALUES
C
C ARGUMENTS:
C CLOCK  - VAX CLOCK TIME ELAPSED
C CPU    - VAX CPU TIME ELAPSED
C BUFFERED - VAX BUFFERED I/O
C DIRECT - VAX DIRECT I/O
C PAGE   - VAX PAGE FAULT COUNT
C
C PROTOCOL:
C SERVICE PORT
C
C LAYER:
C ALL : PERFORMANCE UTILITY
C
C DATE:
C APRIL 12, 1983
C
C======================================================
C
INTEGER*4 BUFIO, CPUTIME, DIO, PAGEF
INTEGER*4 BUFFERED, CPU_INT, DIRECT, PAGE
INTEGER*4 NEW_BUFF, NEW_DIRECT, NEW_PAGE
REAL NEW_CLOCK, NEW_CPU
INTEGER*4 BUFIO_ADR, CPUTIME_ADR, DIO_ADR, PAGEF_ADR
INTEGER*4 ZERO1, ZERO2, ZERO3, ZERO4, ZERO5
INTEGER*4 SYS$GETJPI, STATUS

INTEGER*2 LENGTH1, LENGTH2, LENGTH3, LENGTH4, LENGTH5
INTEGER*2 BUFIO_CODE, CPUTIME_CODE, DIO_CODE, PAGEF_CODE

COMMON/STATCOM/ CLOCK_TIME, BUFIO, CPUTIME, DIO, PAGEF
COMMON/JPICOM/ LENGTH1, BUFIO_CODE, BUFIO_ADR, ZERO1,
               LENGTH2, CPUTIME_CODE, CPUTIME_ADR, ZERO2,
               LENGTH3, DIO_CODE, DIO_ADR, ZERO3,
               LENGTH4, PAGEF_CODE, PAGEF_ADR, ZERO4, ZERO5
DATA BUFIO_CODE/ 1036 /
DATA CPUTIME_CODE/ 1031 /
DATA DIO_CODE/ 1035 /
DATA PAGEF_CODE/ 1034 /
DATA LENGTH1, LENGTH2, LENGTH3, LENGTH4/4,4,4,4/

C DETERMINE THE STATISTICS
C
BUFIO_ADR = %LOC( BUFFERED )
CPUTIME_ADR= %LOC( CPU_INT )
DIO_ADR = %LOC( DIRECT )
PAGEF_ADR = %LOC( PAGE )

C
STATUS = SYS$GETJPI( ''LENGTH1'', )
IF( STATUS.NE.1 ) WRITE( 6,100 ) STATUS
100 FORMAT(' Error, SYS$GETJPI, status is 'Z8','')
C
RETURN THE APPROPRIATE STATISTICS
C
NEW_CLOCK = SECONDS( CLOCK_TIME )
NEW_CPU = ( CPU_INT - CPUPTIME )/100.0
NEW_DIRECT = DIRECT - DIO
NEW_PAGE = PAGE - PAGEF
NEW_BUFF = BUFFERED - BUFIO
C
RETURN
END
SUBROUTINE TRACK( BLOCK, DATA_TYPE)

C=============================================~======
C
C PURPOSE:
C
C DISPLAY THE FORMAT OF THE REQUESTED DATA STRUCTURE
C
C TWO DATA STRUCTURES ARE DISPLAYED -
C
1.) PCB VECTOR
C
2.) PCB
C
C ARGUMENTS:
C
BLOCK - THE ARRAY CONTAINING THE DATA
DATA_TYPE - THE DATA STRUCTURE TYPE

= 0 IF PCB VECTOR
= 1 IF PCB
C
C PROTOCOL:
C
SERVICE PORT
C
LAYER:
C
ALL
C
DATE:
C
APRIL 12, 1983
C
C=============================================================================
C
INCLUDE 'SPPCOM.TXT'
INTEGER*2 REQUEST_LENGTH
INTEGER*2 BUFFER1_LENGTH, BUFFER2_LENGTH
INTEGER*4 DBP_STATUS(4), HOST_STATUS(6), DATA_TYPE
CHARACTER*40 DBP_MESSAGE(4), HOST_MESSAGE(6), DBP+HOST
DATA DBP_STATUS
X /4,5,6,7/
DATA HOST_STATUS
X /0,1,2,3,5,17/
DATA DBP_MESSAGE
X /'WAIT ON ENABLE',
X /'READ RESPONSE',
X /'READ RESPONSE WITH EOM',
X /'WRITE REQUEST' /
DATA HOST_MESSAGE
X /'SUSPEND SESSION',
X /'READ/WRITE OK',
X /'ERROR ENCOUNTERED',
X /'WRITE OK WITH EOM',
X /'OK FIN',
X /'ENABLE SERVICE PORT' /
C
Determine the necessary decimal values
C
CALL_GLUE( BLOCK(29), BLOCK(30), REQUEST_LENGTH )
CALL GLUE( BLOCK(36),BLOCK(37),BUFFER1_LENGTH )
CALL GLUE( BLOCK(42),BLOCK(43),BUFFER2_LENGTH )

C
C OUTPUT THE PCB VECTOR OR PCB
C
IF( DATA_TYPE.EQ.1 ) THEN
C
PROCESS A PCB DATA STRUCTURE
C
DO 50 I = 1,4
50 IF( BLOCK(15).EQ,DBP_STATUS(I)) GO TO 75
DBP = 'UNKNOWN DBP STATUS'
GO TO 80
75 DBP = DBP_MESSAGE(I)
80 DO 100 I = 1,6
100 IF( BLOCK(16).EQ,HOST_STATUS(I)) GO TO 125
HOST = 'UNKNOWN HOST STATUS'
GO TO 130
125 HOST = HOST_MESSAGE(I)
130 WRITE( UNIT,200 ) (BLOCK(I1),I1=1,14),(BLOCK(I5),DBP),
X BLOCK(16),HOST,(BLOCK(I2),I2=17,28),REQUEST_LENGTH,
X BLOCK(I3),(BLOCK(I3),I3=35,32,-1),
X BUFFER1_LENGTH,(BLOCK(I4),I4=41,38,-1),
X BUFFER2_LENGTH
200 FORMAT( ' +---------------------+/',
X ' | PCB |/',
X ' +---------------------+/',
X ' | RESERVED|T25,I4(Z2,2,1X)|/',
X ' | DBP STATUS|T25,Z2,2,1X,A|/',
X ' | HOST STATUS|T25,Z2,2,1X,A|/',
X ' | RESERVED|T25,I2(Z2,2,1X)|/',
X ' | REQUEST LENGTH|T25,I4|/',
X ' | NUMBER OF SEGMENTS|T25,I1|/',
X ' | BUFFER 1 PTR|T25,4(Z2,2)|/',
X ' | BUFFER 1 LENGTH|T25,I4|/',
X ' | BUFFER 2 PTR|T25,4(Z2,2)|/',
X ' | BUFFER 2 LENGTH|T25,I4|/'
ELSE
C
PROCESS A PCB VECTOR DATA STRUCTURE
C
WRITE( UNIT,300 ) (BLOCK(I1),I1=2,1,-1),
X (BLOCK(I2),I2=6,3,-1),(BLOCK(I3),I3=10,7,-1)
300 FORMAT( ' +---------------------+/',
X ' | PCB VECTOR |/',
X ' +---------------------+/',
X ' | INDEX |T30,Z2,2|/',
X ' | CONTROL PCB ADDRESS|T30,4Z2,2|/',
X ' | APPLICATION PCB ADDRESS|T30,4Z2,2|')
ENDIF
RETURN
END
$ !
$ ! THIS IS THE COMMAND FILE USED TO RUN PROGRAM 'SPP'
$ ! THE VMS TTY PORT 'TTB0:' IS USED FOR COMMUNICATIONS
$ !
$ DBPTERM := _TTB0:
$ ! ALLOCATE THE PORT FOR ACCESS
$ ALLOCATE 'DBPTERM'
$ SET PROTECTION=(WRW)/DEVICE 'DBPTERM'
$ !
$ ! SET TERMINAL CHARACTERISTICS FOR TTBO:
$ ! SEE FIGURE 2 OF THIS REPORT
$ !
$ SET TERMINAL 'DBPTERM'/NOWRAP/WIDTH=80/SPEED=9600/PASSALL/EIGHT_BIT/PERM
$ ASSIGN/USER 'DBPTERM' REMOTE
$ ASSIGN/USER TT: SYS$INPUT
$ RUN [INTEL.SPP]SPP
$ ! DEALLOCATE TTBO:
$ DEALLOCATE 'DBPTERM'
APPENDIX B

A sample transmission trace
** Initialize iDBP Communications **

== Q_OUTPUT ==
# of bytes is 1
Byte Stream:

03

== Q_INPUT ==
# of bytes is 16
Byte Stream:

AO 0A 2A 43 6F 6E 74 72 6C 20 43 2A 0A 2E 01 2E

** Create Control Session **
** Initiate a READ_BLOCK **

== Q_OUTPUT ==
# of bytes is 11
Byte Stream:

55 52 0D 0A 00 00 00 0C EE 85 E6

== Q_INPUT ==
# of bytes is 25
Byte Stream:

55 52 0D 0A 0A 00 00 00 0C EE 85 E6 00 00 00 00
4D 98 00 00 00 00 01 2E 01 2E

+---------------------+
| PCB VECTOR |
+---------------------+

INDEX
CONTROL PCB ADDRESS 9B4D0000
APPLICATION PCB ADDRESS 00000000

** Initiate a READ_BLOCK **

== Q_OUTPUT ==
# of bytes is 11
Byte Stream:

55 52 0D 2B 00 00 00 4D 98 32 E1

== Q_INPUT ==
# of bytes is 58
Byte Stream:

55 52 0D 0A 2B 00 00 00 0D 98 32 E1 00 00 00 00
00 00 00 00 00 00 00 00 00 00 04 00 00 02 00 00
00 00 00 00 04 00 00 00 00 00 01 BD 1F 03 00 00
00 FF FF FF 00 00 00 83 51 2E

+---------------------+
| PCB |
| 56 |
** Initiate a WRITE_BLOCK **

** Initiate a WRITE_BLOCK **

== Q_OUTPUT ==
# of bytes is 11
Byte Stream:

55 57 0D 28 00 00 00 40 98 32 E1

** Initiate a WRITE_BLOCK **

== Q_OUTPUT ==
# of bytes is 5
Byte Stream:

55 57 0D 0A 06

== Q_INPUT ==
# of bytes is 45
Byte Stream:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 11 11
00 02 00 00 00 00 00 00 04 00 00 00 00 00 01 BD 11
1F 03 00 80 00 FF FF FF 00 00 00 9E 51 11

** Initiate a WRITE_BLOCK **

== Q_INPUT ==
# of bytes is 2
Byte Stream:

06 2E

** Initiate a WRITE_BLOCK **

== Q_OUTPUT ==
# of bytes is 11
Byte Stream:

55 57 0D 28 00 00 00 40 98 32 E1

== Q_OUTPUT ==
# of bytes is 5
Byte Stream:

55 57 0D 0A 06

== Q_INPUT ==
# of bytes is 45
Byte Stream:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 11 11
00 02 00 00 00 00 00 00 04 00 00 00 00 00 01 BD 11
1F 03 00 80 00 FF FF FF 00 00 00 9E 51 11

** Initiate a WRITE_BLOCK **

== Q_INPUT ==
# of bytes is 2
Byte Stream:

06 2E
** Create Application Session **
** Send Request **
** Initiate a READ_BLOCK **

** Initiate a READ_BLOCK **
** Send Request **
** Create Application Session **
** Initiate a WRITE_BLOCK **

== Q_OUTPUT ==

# of bytes is 11  
byte stream:

```
55 57 0D 09 0B 1F 00 30 14 17
```

== Q_INPUT ==

# of bytes is 5  
byte stream:

```
55 57 0D 0A 06
```

== Q_OUTPUT ==

# of bytes is 11  
byte stream:

```
01 00 00 00 E4 01 FE FF 00 5C 7A
```

== Q_INPUT ==

# of bytes is 2  
byte stream:

```
06 2E
```

** Initiate a WRITE_BLOCK **

== Q_OUTPUT ==

# of bytes is 11  
byte stream:

```
55 57 0D 2B 00 00 00 4D 98 32 E1
```

== Q_INPUT ==

# of bytes is 5  
byte stream:

```
55 57 0D 0A 06
```

== Q_OUTPUT ==

# of bytes is 45
TRACE.DBP;55  24-MAY-1983 14:26  Page 5

Byte Stream:

00 00 00 00 00 00 00 00 07 03  
00 02 00 00 00 00 00 00 04 00 00 09 00 01 B9  
1f 00 30 84 00 FF FF 00 F0 00 00 58 80  

== Q_INPUT ==

# of bytes is 2
Byte Stream:

06 2E 

+---------------------+
<table>
<thead>
<tr>
<th>PCB</th>
</tr>
</thead>
</table>

RESERVED 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
iDBP STATUS 07 WRITE REQUEST
HOST STATUS 03 WRITE OK WITH EOM
RESERVED 00 02 00 00 00 00 00 00 00 04 00 00
REQUEST LENGTH
NUMBER OF SEGMENTS 1
BUFFER 1 PTR 30001FB9
BUFFER 1 LENGTH 132
BUFFER 2 PTR F00FFFF
BUFFER 2 LENGTH 0

== Q_OUTPUT ==

# of bytes is 2
Byte Stream:

47 0D 

== Q_INPUT ==

# of bytes is 29
Byte Stream:

47 0D 0A 0D 0A 2A 42 52 45 41 4B 2A 20 61 74 20 65,*BREAK* at 33 30 41 36 3A 30 33 42 41 20 0D 0A 2E 30A6:03BA ... 

** Receive Response **
** Initiate a READ_BLOCK **

== Q_OUTPUT ==

# of bytes is 11
Byte Stream:

55 52 0D 0A 00 00 00 0C EE 85 E6 

== Q_INPUT ==

# of bytes is 25
Byte Stream:

55 52 0D 0A 0A 00 00 00 0C EE 85 E6 00 00 00 00 00 4D 98 00 00 C5 68 7C BF 2E 

60
** Initiate a READ_BLOCK **

== Q_OUTPUT ==

爀 # of bytes is 11

Byte Stream :

55 52 0D 2B 00 00 00 4D 98 32 E1

== Q_INPUT ==

爀 # of bytes is 58

Byte Stream :

55 52 0D 0A 2B 00 00 00 4D 98 32 E1 00 00 00 00 UR...+...M.2.

00 00 00 00 00 00 00 00 00 6E 00 00 02 00 00 ........................

00 00 00 00 00 04 00 00 09 00 01 B9 IF 00 30 0B ........................0.

00 FF FF 00 F0 00 00 D6 61 2E

** Initiate a READ_BLOCK **

== Q_OUTPUT ==

爀 # of bytes is 11

Byte Stream :

55 52 0D 0B 00 B9 1F 00 30 15 F5

== Q_INPUT ==

爀 # of bytes is 26

Byte Stream :

55 52 0D 0A 0B 00 B9 1F 00 30 15 F5 01 00 00 00 UR....0........

FC 03 E4 01 00 FF 00 08 DA 2E

** Initiate a WRITE_BLOCK **
== Q_OUTPUT ==
# of bytes is 11
Byte Stream :
55 57 OD 2B 00 00 00 4D 98 32 E1

== Q_INPUT ==
# of bytes is 5
Byte Stream :
55 57 OD 0A 06

== Q_OUTPUT ==
# of bytes is 45
Byte Stream :
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 01
00 02 00 00 00 00 00 00 00 00 00 00 04 00 00 09 00 01 B9
1F 00 30 0B 00 FF FF 00 F0 00 00 D5 E1

== Q_INPUT ==
# of bytes is 2
Byte Stream :
06 2E

+---------------------+
| PCB |
+---------------------+
RESERVED 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IDBP STATUS 06 READ RESPONSE WITH ED
HOST STATUS 01 READ/WRITE OK
RESERVED 00 02 00 00 00 00 00 00 00 00 04 00 00
REQUEST LENGTH 9
NUMBER OF SEGMENTS 1
BUFFER 1 PTR 30001FB9
BUFFER 1 LENGTH 11
BUFFER 2 PTR F000FFFF
BUFFER 2 LENGTH 0

== Q_OUTPUT ==
# of bytes is 2
Byte Stream :
47 OD

== Q_INPUT ==
# of bytes is 29
Byte Stream :
47 0D 0A 0D 0A 2A 42 52 45 41 48 2A 20 61 74 20 G....*BREAK* at
33 30 41 36 3A 30 33 42 41 20 0D 0A 2E 30A6103BA ...
** All data has been received **
** Create Application Response **

FC 03 E4 01 00 FF 00
== Q_OUTPUT ==
# of bytes is 2
Byte Stream :
 47 0D

== Q_INPUT ==
# of bytes is 29
Byte Stream :
 47 0D 0A 0D 0A 2A 42 52 45 41 4B 2A 20 61 74 20 6...*BREAK* at
33 30 41 36 3A 30 33 42 41 20 0D 0A 2E 30A6:03BA ...
Figure 1. - A flowchart describing HILDA.
**HILDA: A SAMPLE QUERY**

<table>
<thead>
<tr>
<th>HOST</th>
<th>DBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAYER 3: DBPOL</td>
<td>LAYER 2: DBPSSL</td>
</tr>
<tr>
<td>DATA BASE PROCESSOR QUERY LANGUAGE</td>
<td>DATA BASE PROCESSOR SEMANTICS SPECIFICATION LANGUAGE</td>
</tr>
</tbody>
</table>

```
"CREATE DATABASE TEST"
START('60'X):
ASC('TEST', 4):
SEND:
```

```
SEND REQUEST
60
04
45
53
54
FF
00
```

```
DBP RESPONSE
RECV(RESPONSE, NBYTES MORE):
IF MORE THEN RECEIVE MORE DATA FROM THE DBP
```

```
RECEIVE RESPONSE
```

Figure 2. - A sample query within HILDA.
Figure 3. - The physical DBP environment.
Figure 4. - Layers within HILDA and SPP.
If the host sends the following request module to iDBP:

[IDID]-----Command-1-----|-----Command-2-----|Command-3|

Then the host and iDBP will transmit the following segments:
(values are for example only)

1. 

HOST 

+--------------------------+
| ID | ID | 4-byte header |
+--------------------------+

iDBP 

Number of Segments = 1,
Buffer 1 Length = 4

2. 

HOST 

+--------------------------+
| Command-1-a | 512 bytes |
+--------------------------+

iDBP 

Number of Segments = 1,
Buffer 1 Length = 512

3. 

HOST 

+--------------------------+
| Command-1-b Command-2-a | 512 bytes |
+--------------------------+

iDBP 

Number of Segments = 1,
Buffer 1 Length = 512

4. 

HOST 

+--------------------------+
| Command-1-b Command-2-a | 512 bytes |
+--------------------------+

iDBP 

Number of Segments = 2,
Buffer 1 Length = 4,
Buffer 2 Length = 512

4-byte ID | ID | header

512 bytes Response-A

Figure 5. General Form for Host-DBP Interaction.
Figure 6. - VAX Asynchronous Communications Parameters.
Figure 7. - Threaded Data Structure of SPP.
The design and implementation of a data communications protocol for the Intel Data Base Processor (DBP) is defined. The protocol is termed SPP (Service Port Protocol) since it enables data transfer between the host computer and the DBP service port. The protocol implementation is extensible in that it is explicitly layered and the protocol functionality is hierarchically organized. Extensive trace and performance capabilities have been supplied with the protocol software to permit optional efficient monitoring of the data transfer between the host and the Intel data base processor. Machine independence was considered to be an important attribute during the design and implementation of SPP. The protocol source is fully commented and is included in Appendix A of this report.
End of Document