# User's Guide

# DeviceNet™
# PC Interface Card

# DN-PC2
Rev. 1.0

HURON
NET
WORKS

771 Airport Boulevard, Suite 2, Ann Arbor, Michigan 48108 Phone: (313) 995-2637 Fax: (313) 995-2876

# 1. INTRODUCTION

The DN-PC2 hardware and DLL provide a convenient system development tool for DeviceNet™ based CAN networks. The DNPC2.DLL was designed to use the Allen-Bradley NetWdn16 API as the interface to the hardware.

The DLL interfaces to the DN-PC2 through an interrupt driven device driver. Commands are available to configure both the DN-PC2 specific interface as well as the CAN network parameters.

# 2. INSTALLING THE DN-PC2

The DN-PC2 may be installed in an ISA compatible Personal Computer. The module occupies 32 consecutive locations within the processor I/0 space. Prior to installation, the user must set both the base address and the interrupt level to be used by the adapter.

## 2.1. Port Addresses

The DN-PC2 may be configured to one of 4 separate base addresses using switch positions S9 and S10. Note that when setting the switch the down position is ON and the up position is OFF.

| SWITCH | | ADDRESS | |
|---|---|---|---|
| S10 | S9 | | |
| OFF | OFF | 0200H | |
| OFF | ON | 0280H | |
| ON | OFF | 0300H | (default) |
| ON | ON | 0380H | |

## 2.2. Interrupt Levels

The DN-PC2 supports 8 different interrupts through switch positions S1 through S8. Only 1 of these switches should be in the ON (Down) position.

| SWITCH | INTERRUPT | USAGE |
|---|---|---|
| S1 | IRQ15 | General I/O |
| S2 | IRQ12 | General I/O, PS2 mouse |
| S3 | IRQ11 | General I/O |
| S4 | IRQ10 | General I/O |
| S5 | IRQ7 | LPT1 |
| S6 | IRQ5 | LPT2 |
| S7 | IRQ4 | Com1 and Com3 |
| S8 | IRQ3 | Com2 and Com4 |

## 2.3. Connector Pin Out

The pin-out for the DB9 connector is as follows:

| PIN | FUNCTION | DeviceNet SIGNAL |
|---|---|---|
| 1 | N.C. | |
| 2 | CAN_L | CAN_L (blue) |
| 3 | GND | V- (black) |
| 4 | N.C. | |
| 5 | Shield (optional) | Shield (bare) |
| 6 | GND (optional) | |
| 7 | CAN_H | CAN_H (white) |
| 8 | N.C. | |
| 9 | V+ | V+ (red) |

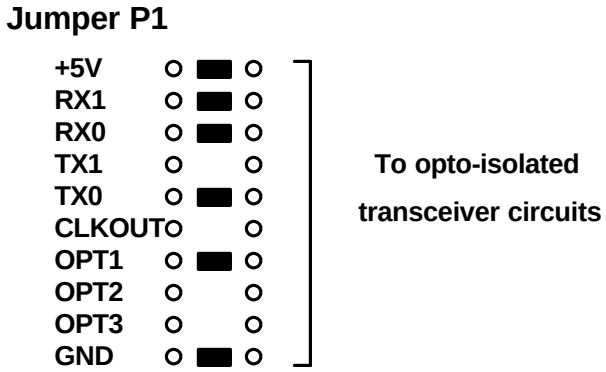
The above pin out complies with the IS-11898 standard used by CiA. For use with DeviceNet, signals should be connected as shown in the above table.

## 2.4. Network Adapter Jumpers

The DN-PC2 is provided with an optically coupled CAN transceiver. To support applications requiring alternate drivers, two sets of dual row 10 position jumpers are provided. Jumper group P1 carry signals from the CAN controller to the CAN transceiver circuit. Jumper group P2 carry signals from the transceiver to the DB9 connector. Jumper P2-10 provides a convenient option for enabling a 120 ohm network termination resistor. If the DN-PC2 card is to be enclosed in a PC chassis, the use of the onboard termination resistor is not recommended since it will be difficult to tell whether it is in or out. A three pin jumper group next to J2 is used to select the connection of the shield to the hybrid ground.

### 2.4.1. Jumper P1

Unless a custom network driver is installed the jumper locations on P1 containing white silk screened lines should be installed.




### 2.4.2. Jumper P2

Jumper group P2 connects the CAN transceiver circuit to the DB9 connector. Unless a custom network driver is installed, jumpers on P2 should be installed to connect the on-board transceiver. Some of the jumpers are used to select the power option for the transceiver.

To provide full isolation between the CAN network and the PC the transceiver circuit requires a separate power source. This is typically provided by a network wide power supply carried on the BUS + and BUS GND signals.

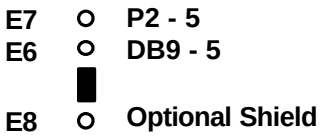
The following jumper options should be used when the BUS + and BUS GND signals are to be used to power the transceiver.

| Jumper P2 | DB9 Connector | Function |
|---|---|---|
| Xcvr GND | 3 | Bus GND |
| Unreg Xcvr Pwr | 1 | none |
| PC +12 | 8 | PC +12 (if installed) |
| Xcvr GND | 4 | PC GND |
| GND | 5 | 3 pin jumper |
| Xcvr GND | 6 | alternate Bus GND |
| Unreg Xcvr Pwr | 9 | Bus + |
| Xcvr CAN_H | 7 | CAN_H |
| Xcvr CAN_L | 2 | CAN_L |
| 120 resistor | none | 120 W termination resistor |

The following jumper options should be used when the PC +12 power is to be used to power the transceiver. Note that this configuration does not provide galvanic isolation between the CAN network and the PC. Installing the jumper shown in outline form will connect the PC +12 volt power to the DB9 connector. **Care should be exercised**.

| Jumper P2 | | DB9 Connector | Function |
|---|---|---|---|
| Xcvr GND |  | 3 | Bus GND |
| Unreg Xcvr Pwr |  | 1 | none |
| PC +12 |  | 8 | PC +12 (if installed) |
| Xcvr GND |  | 4 | PC GND |
| GND |  | 5 | 3 pin jumper |
| Xcvr GND |  | 6 | alternate Bus GND |
| Unreg Xcvr Pwr |  | 9 | Bus + |
| Xcvr CAN_H |  | 7 | CAN_H |
| Xcvr CAN_L |  | 2 | CAN_L |
| 120 resistor |  | none | 120 Ω termination resistor |

### 2.4.3. Shield Jumper Group




For all DeviceNet applications the 3 pin jumper should connect the shield to the DB9 connector, pin 5 (DB9-5).

## 3. INSTALLING SOFTWARE

The DN-PC2 DLL driver is provided on a 3.5 inch floppy diskette. This file is to be installed as described below, for operation with the NetWdn16 dirver dll. Typically, an initialization option within the application will bring up a dialog box with the Huron Net Works name in it that will allow the selection of the DN-PC2 I/O address and interrupt level.

The installation procedure calls for making an entry in a private .INI file that NetWdn16 parses to see what interfaces are available. The installation procedure is as follows:

1. Search for the file "windnet.ini" in the Windows directory (usually c:\windows). If it is not found, then create this file.

2. Search for a "drivers" section in windnet.ini. If it is not found, then create it by writing the string "[drivers]".

3. Add a description of your interface as the "entry" in the drivers section. The "string" associated with the "entry" must specify the fully qualified path of the Driver DLL. NetWdn16 passes this string directly to the LoadLibrary function.

4. Copy dnpc2.dll to the appropriate location(s) on the PC.

An example "windnet.ini" file:

[drivers]
Vendor X Driver Y = c:\vendx\drivey\devnet.dll
HNW DeviceNet DLL = c:\windows\dnpc2.dll

This example windnet.ini contains a drivers section (denoted by the "[drivers]" text) and two driver descriptions; "Vendor X Driver Y" and "HNW DeviceNet DLL". The driver descriptions are the entries that NetWdn16 will display in a List Box when a selection is necessary.

When the user makes a selection, NetWdn16 grabs the associated directory & file name string (to the right of the "=" sign) & passes this without modification to the ::LoadLibrary() function. For example, if the user selected "HNW DeviceNet DLL" from the List Box, then NetWdn16 would pass the string "c:\windows\dnpc2.dll" to the LoadLibrary() function.

The driver description string (to the left of the "=" sign) is limited to a maximum of 32 characters. The path string (to the right of the "=" sign) is limited to a maximum of 500 characters.

## 4. DN-PC2 SPECIFICATIONS

| | |
|---|---|
| Size | 8" x 4.5" (ISA BUS COMPATIBLE) |
| Power | typical. 100 mA @ 5 volts (ISA bus)<br>typical. 60 mA @ 11 to 25 volts (DB9 bus) |
| Bus Interface | optical isolation, ISO/IS 11898 |
| Bus Speed | up to 500 Kbit/sec, 64 nodes |
| Interrupt Levels | 3,4,5,7,10,11,12,15 (user selectable) |
| I/O Addresses | 200-21F, 280-29F, 300-31F, 380-39F |

## 5. DLL HEADER FILE

The following is a sample header file defining the API for using the DNPC2.DLL with a custom windows application.

```
//  NetWdn16 to Driver API Functions
//  These are the functions that NetWdn16 can invoke within a DeviceNet
//  Driver DLL. NetWdn16 performs a ::LoadLibrary() to access each driver
//  DLL. These are the functions for which NetWdn16 will issue GetProcAddress() calls
//  and which  the Driver DLL exports. Parameters are also indicated.
//
//--------------NetWdn16 to Driver DLL Structure Definitions-----------------
//
//-----------------------------------------------------------------------------
// T_DRV_CONFIG
//
//   Use:        NetWdn16 passes a reference to this structure to an interface's
//                       DNetDrvConfig() function.
//
//   Memebers:
//                  appHwnd
//                       The window handle of the application that is registering
//                       with NetWdn16 and that is causing the invocation of the
//                       DNetDrvConfig() routine.  The interface can use this as the
//                       parent of any modal dialog boxes it needs to display during
//                       the config function.
//
//                  hNetWdn16
//                       The HINSTANCE associated with the NetWdn16 DLL.
//                       The Driver DLL can use this value to obtain the addresses
//                       of NetWdn16 functions it invokes at run-time using the
//                       GetProcAddress() function vs. getting these addresses at
//                       link time.  Obtaining the function addresses within
//                       NetWdn16 at run-time decouples the Driver DLL
//                       from updates to the NetWdn16 Client API.  If the addresses
//                       are obtained at link-time of the Driver DLL (through a
//                       IMPORTS section in the .DEF file or by linking with
//                       NetWdn16's LIB file), then updates to the Client API may
//                       require a re-build of the Driver DLL.
//
//                  appId
//                       An internal value NetWdn16 uses to identify the Client
//                       Application that is registering and is, thus, causing this
//                       routine to be invoked.  The driver DLL will pass this value
//                       back in the associated  DNetDrvConfigComplete() call.
//
//                  hdnAppHwnd
//                       The window handle of NetWdn16's hidden application.  The
//                       driver DLL may need to store this for use when hooking into the
//                       hidden application's message loop (e.g. it may need to be reported
```

```
//                      to another Windows App that will subsequently generate windows
//                      messages destined for the driver DLL).  The driver DLL CANNOT
//                      use this as the parent window for any dialog boxes that it may
//                      display. Instead, the appHwnd member must be utilized.
//
//              msgLoopHookId
//                      This parameter can be ignored.
//
//              isPassiveMon
//                      Indicates whether (TRUE) or not (FALSE) a passive monitoring
//                      Client Application is attempting to configure one of this
//                      driver's physical attachments.  The following characteristics
//                      apply to a physical attachment that is being selected for use by
//                      a passive monitoring application:
//
//                      - It cannot be currently SELECTED for any other use.  In
//                        other words, the driver DLL will ensure that a passive
//                        monitoring application obtains sole ownership of a
//                        previously UNSELECTED physical attachment.  Once a pysical
//                        attachment is SELECTED by a passive monitoring application,
//                        it cannot be chosen for use by any other application until
//                        it becomes UNSELECTED.
//
//                      - It can be used ONLY to receive messages, not to transmit.
//                       The physical attachment does not utilize a DeviceNet MAC ID and
//                       does not execute a Duplicate MAC ID check.
//
//                      -An acceptance filter may be specified via the
DnetDrvSetRxFilter()
//                       function. NetWdn16 only invokes this function for physical
//                       attachments being used by a passive monitoring application.  The
//                       driver must hand any message that passes thru the acceptance filter
//                       up to NetWdn16. In addition to acceptance filtering, passive
//                       monitoring applications may also implement "point" monitoring
//                       whereby a specific set of messages are desired.  The screening
//                        associated with specific points ("point  screeners") will be
delivered
//                       to the driver via calls to the  DNetDrvSetupRxScreener() function.
//                       A physical  attachment being used for passive  monitoring purposes
//                      can experience calls to both DNetDrvSetRxFilter() and
//                      DNetDrvSetupRxScreener() from NetWdn16. All messages that
//                      pass through either the current acceptance filter (configured by
//                      calling DNetDrvSetRxFilter()) OR the active point screener(s)
//                      (configured by calls to DNetDrvSetupRxScreener()) will  be
//                      delivered to NetWdn16.
//
//      pDrvSpecificInfo
//              A parameter whose use is defined by the driver DLL.
//              This member will be set to NULL.
//---------------------------------------------------------------------------
typedef struct
```

```
        {
    HWND                    appHwnd;
    HINSTANCE               hNetWdn16;
    unsigned long           appId;
    HWND                    hdnAppHwnd;
    unsigned short          msgLoopHookId;
    BOOL                    isPassiveMon;
    unsigned char FAR       *pDrvSpecificInfo;
        }T_DRV_CONFIG;

//---------------------------------------------------------------------------
// T_MSG_RX
//
//   Use:       NetWdn16 passes a reference to this structure to an interface's
//              DNetDrvPollDriver() function.  If a new message has been
//              received, then the interface will initialize the
//              structure members described below.
//
//   Memebers:
//              identifier
//                      The CAN Identifier Field associated with the received message
//                         is placed in this member.
//
//              msgLength
//                      The number of CAN Data bytes in the received message is placed
//                      in this member.  If this is set to zero, then an "identifier only"
//                      message has been received & the msgData member is ignored.
//
//              msgData
//                      The actual CAN Data Field bytes in the received message are
//                      copied into this member.
//
//              numMs
//                      This member is only used when a physical attachment is being
//                      used for passive monitoring purposes.  If the physical attachment
//                      is not being used for passive monitoring purposes, then this
//                      member is not accessed by NetWdn16.
//
//                      A physical attachment that is being used for passive
//                      monitoring purposes may implement the ability to "timestamp"
//                      message received events.  If this is the case, then this member
//                      should be initialized with a snapshot of the interface's
//                      "millisecond time tick" when this message was received.
//                      If the interface does not support a "millisecond time tick" or
//                      does not support time-stamping, then this member must
//                      be set to zero (0).
//
//              numUs
//                      This member is only used when a physical attachment is being
//                      used for passive monitoring purposes.  If the physical attachment
//                      is not being used for passive monitoring purposes, then this
```

```
//                      member is not accessed by NetWdn16.
//
//                      A physical attachment that is being used for passive
//                      monitoring purposes may implement the ability to "timestamp"
//                      message received events.  If this is the case, then this member
//                      should be initialized with a snapshot of the interface's
//                      "microsecond time tick" when this message was received.
//                      If the interface does not support a "microsecond time tick" or
//                      does not support time-stamping, then this member must
//                      be set to zero (0).
//-------------------------------------------------------------------------
typedef struct
        {
        unsigned short          identifier;
        unsigned char           msgLength;
        unsigned char           msgData[8];
        unsigned long           numMs;
        unsigned long           numUs;
        }T_MSG_RX;

//.........................................................................
//
//      void WINAPI EXPORT DNetDrvConfig(T_DRV_CONFIG FAR *pDrvConfig)
//
//      NetWdn16 invokes this function when a new application
//      within the PC wants to talk on DeviceNet through the interface.  The
//      interface is responsible for configuring itself and, if necessary,
//       executing the Duplicate MAC ID State Machine.  Note that an interface that
//      is not being used for passive monitor purposes is responsible for capturing and
//      processing all Duplicate MAC ID check messages
//
//      Once the driver DLL determines the specific physical attachment desired
//      by the Client App, it immediately places that physical attachment in the
//      SELECTED state.  The physical attachment will remain in the SELECTED state
//      until there are no Client Applications making use of the physical
//      attachment (see DNetDrvUnselect()). Note that the interface will not
//      reconfigure itself if it is in the SELECTED state because other applications
//      may be actively communicating using the current configuration.
//
//      When the configuration is complete (which may include the execution of the
//      Duplicate MAC ID Check), NetWdn16's DNetDrvConfigComplete() routine
//      will be invoked by the driver DLL.
//
//      Multiple client applications may request use of the same interface
//      simultaneously .  In this case NetWdn16 will issue "back to back" calls to
//      DNetDrvConfig() possibly before any of the associated
//      DNetDrvConfigComplete() calls are made.
//
//      Parameters:
//
//      pDrvConfig -    References a structure containing various configuration
```

```
//                      parameters.  See the description of the T_DRV_CONFIG structure
//                      for a detailed definition of all structure members.
//
//.....................................................................
//
//  void WINAPI EXPORT DNetDrvUnselect(int driverId)
//
//      NetWdn16 invokes this routine when there are no longer any applications
//      within the PC referencing the physical attachment specified by "driverId".
//      The "driverId" value is handed to the driver DLL as the return value from
//      DNetDrvConfigComplete(). The interface can now mark this physical attachment
//      as UNSELECTED.
//
//      When there are no more applications within the
//      PC that reference any physical attachments being managed by an interface, the
//      associated Driver DLL is removed from memory by NetWdn16 via a call to
//      FreeLibrary(). A Driver DLL associated with an interface that
//      manages a single physical attachment to DeviceNet will experience the following
//      sequence when no more PC applications are referencing it:
//
//              1.  The DNetDrvUnselect() routine will be invoked.
//
//              2.  The driver's "WEP" or "ExitInstance" routine will be invoked due
//                      to the fact that it is being removed from memory.
//
//      DLLs associated with interfaces that manage multiple physical attachments
//      will be removed from memory via the FreeLibrary() call when no more
//      applications are referencing any of the associated physical attachments.  In this
//      scenario, each previously SELECTED physical attachment will have received a
//      DNetDrvUnselect() call.
//
//      If NetWdn16 determines that the Driver DLL needs to be handed an "unselect"
//      indication prior to a previous call to DNetDrvConfig() being completed (& thus
//      prior to the driver being assigned a driver ID), NetWdn16 will still invoke
//      DNetDrvUnselect() with a value of -1.
//
//      A physical attachment can only reconfigure itself when it is in the UNSELECTED
//      state.
//
//      Parameters:
//
//              driverId
//                      Identifies the specific physical attachment which is no
//                      longer being referenced.  This is a value that was previuosly
//                      returned from the DNetDrvConfigComplete() call.
//
//

//.....................................................................
//
//      void WINAPI EXPORT DNetDrvSetupRxScreener(int driverId, unsigned short
```

```
//                  canIdField)
//
//        This function is used to define a specific message that is to be delivered
//        to NetWdn16.  This is referred to as the configuration of a "point screener"
//        within the driver.
//
//         With respect to physical attachments NOT being used for passive monitoring
//         purposes, NetWdn16 utilizes this function as described below:
//
//        A SELECTED physical attachment that IS NOT being used for passive monitor
//        purposes (see the T_DRV_CONFIG structure) will pass ALL messages
//        whose CAN Identifier Field indicates a UCMM Message (Message Group 3,
//        Message ID 5 or 6) to NetWdn16 via the DNetDrvPollDriver() function.  This is
//        true regardless of the MAC ID specified in the CAN Data Field.  NetWdn16 will
//        not set up point screeners for UCMM messages but the interface will
//        hand all UCMM messages to NetWdn16.  NetWdn16 will handle screening in the
//        Data Field.
//
//        A SELECTED physical attachment will pass all messages whose
//        CAN Identifier Fields are equal to values NetWdn16 has previously specified in
//        DNetDrvSetupRxScreener() calls.  So, this function is used to indicate messages
//        other than UCMM Messages that must be passed up to NetWdn16.  A physical
//        attachment that is not being used for passive monitoring purposes can discard any
//        other message.
//
//********************** IMPORTANT ************************************
//        The interface is responsible for processing all Duplicate MAC ID
//         Check messages and issuing any required response.  Duplicate MAC ID Check
//        messages received across non-passive monitoring physical attachments
//         are NEVER passed up to NetWdn16. The interface hardware will execute the
//        Duplicate MAC ID state machine.
//*************************************************************************
//
//
//         With respect to physical attachments that ARE being used for passive monitoring
//         purposes, NetWdn16 utilizes this function as described below:
//
//        A physical attachment SELECTED for passive monitoring may experience calls to
//        both this routine AND the DNetDrvSetRxFilter() routine to define the messages
//        that are to be received.  NetWdn16 invokes this routine when a passive monitor
//        application requests a "point" monitor of a specific identifier.  The driver
//        will pass all messages whose CAN Identifier Field matches a value
//        NetWdn16 previously specified in a call to this routine AND/OR whose CAN
//        Identifier Field passes through the active acceptance filter that NetWdn16
//        configured by invoking DNetDrvSetRxFilter().  This includes Duplicate MAC ID
//        Check  related messages.
//
//        Parameters:
//
//                  driverId
//                           The identification value that NetWdn16 previously reported
```

```
//                      to the interface as the return value from
//                      DNetDrvConfigComplete().
//                      This is mainly useful if an interface manages multiple physical
//                      attachments to DeviceNet.  In this case, this value identifies
//                      the specific physical attachment for which a screener is to be
//                      configured. Interfaces that manage only a single physical
//                      attachment ignore this parameter.
//
//              canIdField
//                      Specifies the CAN Identifier Field associated with a message that
//                      is to be handed up to NetWdn16.
//
//.....................................................................
//
//      void WINAPI EXPORT DNetDrvCancelRxScreener(int driverId,
//                              unsigned short canIdField)
//
//      Used to cancel screening for a specific identifier previously submitted
//       via the call to DNetDrvSetupRxScreener().
//
//      A non-passive monitor physical attachment (isPassiveMon = FALSE in the
//    T_DRV_CONFIG structure) will discontinue sending messages up to NetWdn16
//      whose CAN Identifier matches the "canIdField" parameter.  This cancels the
//      action caused by a previuous DNetDrvSetupRxScreener() call.
//
//    With respect to physical attachments being used for passive monitoring purposes
//      (isPassiveMon = TRUE in the T_DRV_CONFIG structure), this call will cancel
//      a previously configured "point monitor".  It is important to note, though, that
//      the interface may still need to deliver messages whose CAN Identifier matches the
//      "canIdField" parameter based on the current acceptance filter configuration
//      (see DNetDrvSetRxFilter()).
//
//      Parameters:
//              driverId
//                      The identification value that NetWdn16 previously reported
//                      to the interface as the return value from
//                      DNetDrvConfigComplete().
//                      This is mainly useful if an interface manages multiple physical
//                      attachments to DeviceNet.  In this case, this value identifies
//                      the specific physical attachment for which a screener is to be
//                      canceled
//                      Interfaces that manage only a single physical attachment will ignore
//                      this parameter.
//
//              canIdField
//                      Specifies the CAN Identifier Field whose screening
//                      is to be discontinued.
//
//.....................................................................
//
// void WINAPI EXPORT DNetDrvSetMonRxFilter(int driverId,
```

```
//              unsigned short filterCode,
//              unsigned short filterMask);
//
//      This function is used by NetWdn16 to define an "acceptance filter" for
//      physical attachments that are being used for passive monitoring
//      purposes (isPassiveMon = TRUE in the T_DRV_CONFIG structure).
//      NetWdn16 only invokes this function for physical attachments being used for
//      passive monitoring purposes.
//
//      The passive monitoring physical attachment  utilizes the filterCode and
//      filterMask arguments to determine which DeviceNet messages pass through
//      the acceptance filter and are, thus, to be delivered to NetWdn16.
//      This routine results in either the initial configuration or re-configuration of
//      acceptance filtering logic associated with a passive monitor physical attachment.
//
//      Passive monitor physical attachments may also process calls
//      to DNetDrvSetupRxScreener() to define point screeners.  Point screeners
//      denote specific CAN Identifiers that are also to be sent to NetWdn16 in addition
//      to messages that flow through the acceptance filter.  It is possible for both an
//      acceptance filter and point screeners to be active simultaneously.  Drivers/interface
//      cards that are performing screening logic treat the acceptance filter
//      and point screeners separately as indicated below:
//
//              1.  Message received by a passive monitor physical attachment
//
//              2.  If it passes through the acceptance filter, deliver it to NetWdn16.
//
//              3.  If a point screener has been configured for this  message, deliver
//                              it to NetWdn16.
//
//              4.  If neither #2 or #3 is TRUE, discard the message.
//
//
//      Parameters:
//
//              driverId
//                      The identification value that NetWdn16 previously reported to
//                      the interface as the return value from DNetDrvConfigComplete().
//                      This is mainly useful if an interface manages multiple physical
//                      attachments to DeviceNet.  In this case, this value identifies
//                      the specific physical attachment for which a monitor acceptance
//                      filter is being configured.
//
//              filterCode
//                      Specifies a component of the acceptance filter through which
//                      received messages must pass prior to being fully received
//                      and processed by NetWdn16. This member and the CAN Identifier
//                      Field of the DeviceNet message must be equal within those bit
//                      positions marked as relevant by the filterMaskBits member.
//                      Only the 11 least significant bits of this
//                      member are utilized within acceptance filtering logic.  Note
```

```
//                      that the upper 5 bits of this member are not utilized.
//
//              filterMask
//                      Specifies a component of the acceptance filter through which
//                      received messages must pass prior to being fully received
//                      and processed by NetWdn16.  This member indicates which of the
//                      corresponding bits in the "filterCode" member are "relevant"
//                      or "don't care" when performing acceptance filtering.  If
//                      a bit position is marked as "relevant", then the bit value
//                      within the CAN Identifier of the received message must be equal
//                      to the corresponding bit within the filterCode to pass through
//                      the filter.  If all "relevant" bits are equal, then the
//                      message passes through the filter and is received/processed
//                      by NetWdn16.  The following values are defined:
//
//                              0 - 0x7ff  - Specifies the filter mask
//
//                              0xffff     - A special value used to cancel all acceptance
//                                           filtering.  This indicates that the acceptance
//                                           filter is to be closed such that no messages will
//                                           pass through it.  Note that point screeners may
//                                           still be active and, as such, messages may still
//                                           need to be delivered to NetWdn16.
//
//                      An example of how the filterCode and filterMask members are used
//                      is presented below (the "x" indicates these bits are not involved
//                      in acceptance filtering):
//
//                              filterCode          = xxxx x000 0000 1000 = 0x004A
//                              filterMask          = xxxx x001 1111 0110 = 0x00F0
//                              Relevant Bits   =     @@     @  @ = bits 0,3,9,10
//
//                      In the example above, a received CAN Identifier would have to
//                      have the following characteristics to pass through the filter:
//                              Bit 0 = 0, Bit 3 = 1, Bit 9 = 0, Bit 10 = 0.
//
//                      To specify an acceptance filter that receives all DeviceNet
//                      Message Group 3 traffic, the following values would be plugged
//                      into the filterCode and filterMask.
//                              filterCode = xxxx x110 0000 0000
//                              filterMask = xxxx x001 1111 1111
//
//                      To specify an acceptance filter through which all messages on the
//                      network would pass, all bits within the filterMask would be
//                      set to 1 (0x7FF) and the filterCode value would be "don't care".
//
//.....................................................................
//
//      void WINAPI EXPORT DNetDrvTransmitMessage(int driverId,
//          unsigned short identifier, unsigned char FAR *pMsg, int msgLen)
//
```

```
//      NetWdn16 invokes this to transmit a CAN Data Frame.  The interface will
//      execute whatever set of steps are necessary to transmit the
//      message on DeviceNet.
//
//      Parameters:
//              driverId
//                      The identification value that NetWdn16 previously reported  to
//                      the interface as the return value from DNetDrvConfigComplete().
//                      This is mainly useful if an interface manages multiple physical
//                      attachments to DeviceNet.  In this case, this value identifies the
//                      specific physical attachment across which the message is to be sent.
//                      Interfaces that manage only a single physical attachment can ignore
//                      this parameter.
//
//              identifier - The value to place in the CAN Identifier Field
//
//              pMsg - References the data to place in the CAN Data Field.
//                        This may be a temporary memory reference and if it
//                        is not immediately used its contents must be copied.
//                        If this is NULL, then a zero-length CAN Data Field
//                        message (Identifier Only) is to be transmitted.
//
//              msgLen - The number of bytes referenced by pMsg that are to
//                                be placed in the CAN Data Field.
//
//.....................................................................

//.....................................................................
//
//      int WINAPI EXPORT DNetDrvPollDriver(int driverId,
//              T_MSG_RX FAR *pMsgRx);
//
//      NetWdn16 invokes this function to poll the driver for a new event.  Currently,
//      an "event" includes a new message received or an indication that the
//      physical attachment has gone "off-line".
//
//      Parameters:
//              driverId - The identification value that NetWdn16 previously reported to
//                      the interface as the return value from DNetDrvConfigComplete().
//                      This is mainly useful if an interface manages multiple physical
//                      attachments to DeviceNet.  In this case, this value identifies
//                      the specific physical attachment from which a received message
//                      is to be obtained.
//
//              pMsgRx - If a message that has yet to be delivered to NetWdn16 has been
//                       received by the driver, then the driver is responsible for
//                       initializing this structure with the received message data.
//
//              Return Value:
//
//              0   - Nothing to report. No new events have occurred.
```

```
//
//              1   - A new message has been received and the pMsgRx structure has been
//                          updated accordingly.
//
//             -1  - The physical attachment identified by the driverId parameter
//               went offline.  After delivering this indication, the interface
//               will treat this physical attachment as if it were UNSELECTED.
//                In other words, if a subsequent call to DNetDrvConfig() is made &
//                this physical attachment is selected, the interface will function as
//                if the physical attachment had never been previuosly SELECTED.
//                The interface will hold the specified physical attachment in the
//                off-line state until further instructed by NetWdn16.  The "off-line"
//                state is described in Chapter 6 of Volume I of the DeviceNet
//                Specification.
//
//               In certain cases, it may be possible for NetWdn16 to invoke one of the
//               functions exported by an interface for a physical attachment that has
//               previously reported itself as "off-line".  In this case the interface
//               will ignore the call.
//.....................................................................................
//
//      void WINAPI EXPORT DNetDrvTimeTick(void)
//
//      When NetWdn16 gets loaded into memory it spawns a hidden application
//      that performs a variety of functions.  One of these functions is to
//      deliver a "time tick" to NetWdn16 every 55 milliseconds. NetWdn16 will invoke
//      this routine after it processes the "time tick" event from the hidden application.
//
//



//.....................................................................................
//
//      void WINAPI EXPORT DNetDrvDisplayConfig(int driverId, HWND parentWin)
//
//      This routine is invoked to command the display of the current
//      configuration associated with the physical attachment identified
//      by the driverId parameter.  The HWND passed down is used
//      as the parent window of a dialog box that is displayed.
//
//      Parameters:
//
//              driverId - Identifies the specific physical attachent whose
//                      current configuration information is to be displayed.
//                      This was previously reported to the interface as the return
//                      value from DNetDrvConfigComplete().
//
//              parentWin - Contains the window handle associated with the window
//                      that is to be used as the parent window for the display
//                      that presents the requested configuration.
//
```

```
//.....................................................................
//
//      void WINAPI EXPORT DNetDrvGetTimeStampInfo( int driverId,
//                              unsigned long  FAR *pNumMsMax,
//                              unsigned long  FAR *pNumUsMax );
//
//      This function is available for reading time stamp values.
//
//      Parameters:
//              driverId - Identifies the specific physical attachent whose
//                      current timestamp information is being queried.
//                      This was previously reported to the interface as the return
//                      value from DNetDrvConfigComplete().
//
//              pNumMsMax- references the millisecond timestamp
//
//              pNumUsMax –references the microsecond timestamp
//
//……
//
//---------------------------------------------------------------------------
// Driver To NetWdn16 API Functions
//
// These are exported by NetWdn16 and are invoked by a Driver DLL as described
//  in the comment header blocks
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// T_DRV_INFO
//  Use:        Passed to NetWdn16 by a driver in the DNetDrvConfigComplete() call.
//              This structure contains information that NetWdn16 uses to fill out
//              a DeviceNet and Identity Object for the physical attachment
//              that was configured.
//
//  Memebers:
//              ifaceRev
//                      Indicates the "interface revision" that the driver supports.
//                      If updates are made to the NetWdn16/Driver Interface after
//                      a release has occurred, it will be necessary for NetWdn16 to
//                      provide both the "older" and "newer" interfaces.  This
//                      member tells NetWdn16 the revision level that this particular
//                      driver supports.  The following values are defined:
//                              0 - Invalid/not used
//                              1 - Indicates the driver supports the first release of
//                                 the interface.  Currently, this is the only valid
//                                      value for this member.
//
//              macId
//                      The DeviceNet MAC ID configured for use over the physical
//                      attachment.  This must be a value between 0 - 63 decimal.
//                      Physical attachments being used for passive monitoring
```

```
//                      are not required to utilize a DeviceNet MAC ID.  In this
//                      case, the value 0 can be loaded into this member.
//
//              netBaud
//                      Indicates the DeviceNet Baud Rate that the physical attachment
//                      is using.  This contains the value for the Baud Rate attribute
//                      of the DeviceNet Object. See section 5.5 in Volume I of the
//                      DeviceNet Specification for details.
//
//              vendorCode
//                      Identifies the vendor of the driver/interface card.  This
//                      contains the value for the Vendor attribute of the Identity
//                      Object.  See the description of the Identity Object in Volume
//                      II of the DeviceNet Specification for more details.
//
//              deviceType
//                      This contains the value for the Device Type attribute of the Identity
//                      Object.  See the description of the Identity Object in Volume
//                      II of the DeviceNet Specification for more details.
//
//              productCode
//                      This contains the value for the Product Code attribute of
//                      the Identity Object.  See the description of the Identity Object in
//                      Volume II of the DeviceNet Specification for more details.
//
//              majorRevision
//                      This contains the value for the Major Revision portion of the
//                      Identity Object's Revision attribute. See the description of the
//                      Identity Object in Volume II of the DeviceNet Specification for
//                      more details.
//
//              minorRevision
//                      This contains the value for the Minor Revision portion of the
//                      Identity Object's Revision attribute. See the description of the
//                      Identity Object in Volume II of the DeviceNet Specification for
//                      more details.
//
//              serialNumber
//                      This contains the value for the Serial Number attribute of
//                      the Identity Object.  See the description of the Identity Object in
//                      Volume II of the DeviceNet Specification for more details.
//
//              productName;
//                      This contains the value for the Product Name attribute of
//                      the Identity Object.  See the description of the Identity Object in
//                      Volume II of the DeviceNet Specification for more details.
//
//------------------------------------------------------------------------
typedef struct
        {
        unsigned short          ifaceRev;
```

```
        unsigned char           macId;
        unsigned char           netBaud;
        unsigned short          vendorCode;
        unsigned short          deviceType;
        unsigned short          productCode;
        unsigned char           majorRevision;
        unsigned char           minorRevision;
        unsigned long           serialNumber;
        char FAR                        *productName;
        }T_DRV_INFO;

//---------------------------------------------------------------------------
//
//  int WINAPI EXPORT DNetDrvConfigComplete( BOOL configSuccess,
//                                           char FAR *pErrString,
//                                           int driverId,
//                                           unsigned long appId,
//                                           T_DRV_INFO FAR *pDrvInfo )
//
//              See the description of the DNetDrvConfig() function for details
//              concerning when this function is invoked.  If the configuration was
//              successful, then the physical attachment must remain in the SELECTED
//              state until all Client Applications have discontinued their use (see
//              DNetDrvUnselect()).  If the configuration was not successful, then
//              a previously UNSELECTED physical attachment was chosen & an error
//              was encountered during its configuration step.  If this is the case,
//              then that physical attachment can be marked as UNSELECTED.
//
//      Parameters:
//
//              configSuccess
//                      Indicates whether (TRUE) or not (FALSE) the configuration was
//                              successful.
//
//              pErrString
//                      If this is non-NULL, then it references a character
//                      string describing the error that occurred.
//
//              driverId
//                      This member contains a -1 if this is the first
//                      DNetDrvConfigComplete() call being issued for the physical
//                      attachment that was selected/configured.
//                      If a DNetDrvConfigComplete() call has already been issued for this
//                      physical attachment, then this member contains the "driver ID"
//                      value previously returned by DNetDrvConfigComplete().
//
//              appId
//                      Contains the value sent down in the appId parameter of the
//                      associated DNetDrvConfig() call.
//
//              pDrvInfo
```

```
//                      If this is the first DNetDrvConfigComplete() call for the
//                      physical attachment that was selected/configured (driverId parm = -
//                      1), then this parameter references a T_DRV_INFO structure.
//                      Otherwise, this parameter must be set to NULL.  If the
//                      configSuccess  parameter indicates an error was encountered
//                      (configSuccess = FALSE), then this parameter is ignored by
//                      NetWdn16 and can be set to NULL.
//
//              Return value:
//                      NetWdn16 returns the integer identification value it has allocated
//                      and assigned to the DeviceNet physical attachment that was
//                      selected and successfully configured.  The interface will store this
//                      value for subsequent use when NetWdn16 invokes the
//                      DNetDrvPollDriver() function.  If the physical attachment that was
//                      already SELECTED, then NetWdn16 will return the same
//                      identification value that it previously returned from
//                      DNetDrvConfigComplete().
//
//------------------------------------------------------------------------
```