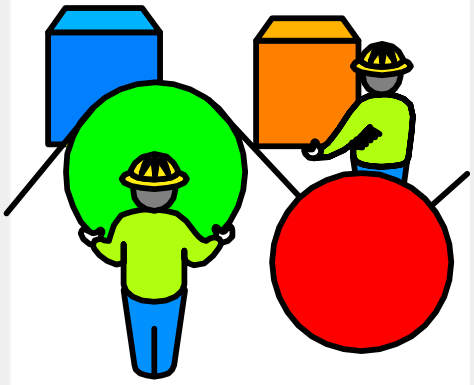
Allen-Bradley

DeviceNet RS-232 Interface Module Communication Protocol

(Cat. No. 1770-KFD)



Reference Manual

Important User Information

Because of the variety of uses for the products described in this publication, those responsible for the application and use of this control equipment must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes and standards.

The illustrations, charts, sample programs and layout examples shown in this guide are intended solely for example. Since there are many variables and requirements associated with any particular installation, Allen-Bradley does not assume responsibility or liability (to include intellectual property liability) for actual use based upon the examples shown in this publication.

Allen-Bradley publication SGI-1.1, “Safety Guidelines For The Application, Installation and Maintenance of Solid State Control” (available from your local Allen-Bradley office) describes some important differences between solid-state equipment and electromechanical devices which should be taken into consideration when applying products such as those described in this publication.

Throughout this manual we make notes to alert you to possible injury to people or damage to equipment under specific circumstances.

ATTENTION: Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss.

Attention helps you:

- Identify a hazard.
- Avoid the hazard.
- Recognize the consequences.

Important: Identifies information that is especially important for successful application and understanding of the product.

Important: We recommend you frequently backup your application programs on appropriate storage medium to avoid possible data loss.

Table of Contents

Using This Manual

Document Objectives

The purpose of this reference manual is to aid you in:

- writing applications that use the 1770-KFD interface module
- communicating with the 1770-KFD over the RS-232 serial-link
- preparing the 1770-KFD to be an interface module to the DeviceNet network

What to Expect

This manual explains how to communicate with the 1770-KFD interface module over an RS-232 serial-link. The protocol used to accomplish this includes:

- 1770-KFD specific, full-duplex DF1 protocol
- PCCC protocol
- DeviceNet protocol

Please note that this manual covers communication between your host computer and the 1770-KFD only. It does not explain how to communicate on the DeviceNet network.

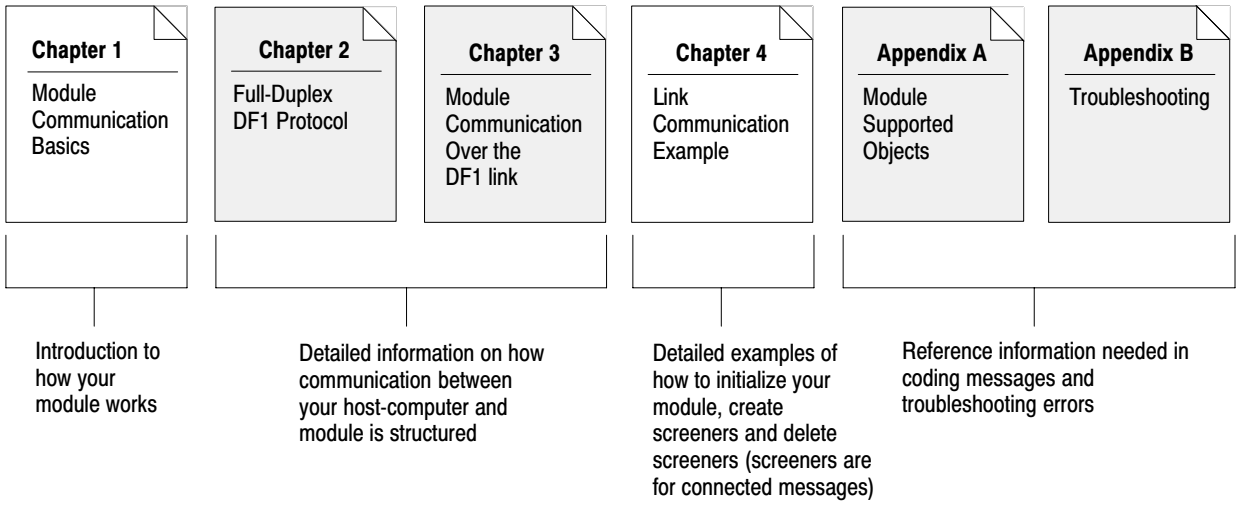
Assumptions About You

We assume that you are a product developer or technical user and:

- have a general familiarity with communication protocols
- have a knowledge of RS-232
- have access to the DeviceNet Specifications Volumes I and II
- have some knowledge of object-oriented modelling concepts

How to Use This Manual

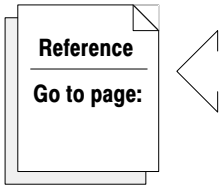
This manual can be divided into four main sections as shown below.



Document Conventions

Often you will see icons in the left margin of a page. These icons are designed to call your attention to more sources of information concerning the subject about which you are reading.

The *more* icon is placed beside paragraphs that refer you to other publications for additional information.



The *reference* icon gives you a page number within this document where more information about what you are reading can be found.

Manual Terminology

The table below lists terms, and their respective definitions, that are frequently used in this manual.

Frequently used term	Definition
BCC	block check character
hex	hexadecimal format
host	a host-computer it can be a desktop, laptop, or notebook personal computer
link-level	the physical communication layer between your host-computer and the 1770-KFD interface module
module	the 1770-KFD RS-232 interface module
network	in this document, the term *network* always refers to the DeviceNet network
PCCC	Programmable Controller Communication Commands an application-level command set that Allen-Bradley programmable controllers use to communicate across networks
serial-link	the RS-232 serial connection between your host-computer and the 1770-KFD interface module

Related Publications

Publication name	Publication number
DeviceNet RS-232 Interface Module Installation Instructions	1770-5.6
DeviceNet Specification - Volume I	available through ODVA*
- Volume II	available through ODVA*
Data Highway/Data Highway Plus™/DH-485 Communication Protocol and Command Set Reference Manual	1770-6.5.16

*ODVA is the Open DeviceNET Vendors Association. To obtain the specifications, contact:

ODVA
Attn: Bill Moss
8222 Wiles Road, Suite 287
Coral Springs, FL 33067
voice: (305) 340-5412
fax : (305) 340-5413
email: BillMoss@industry.net

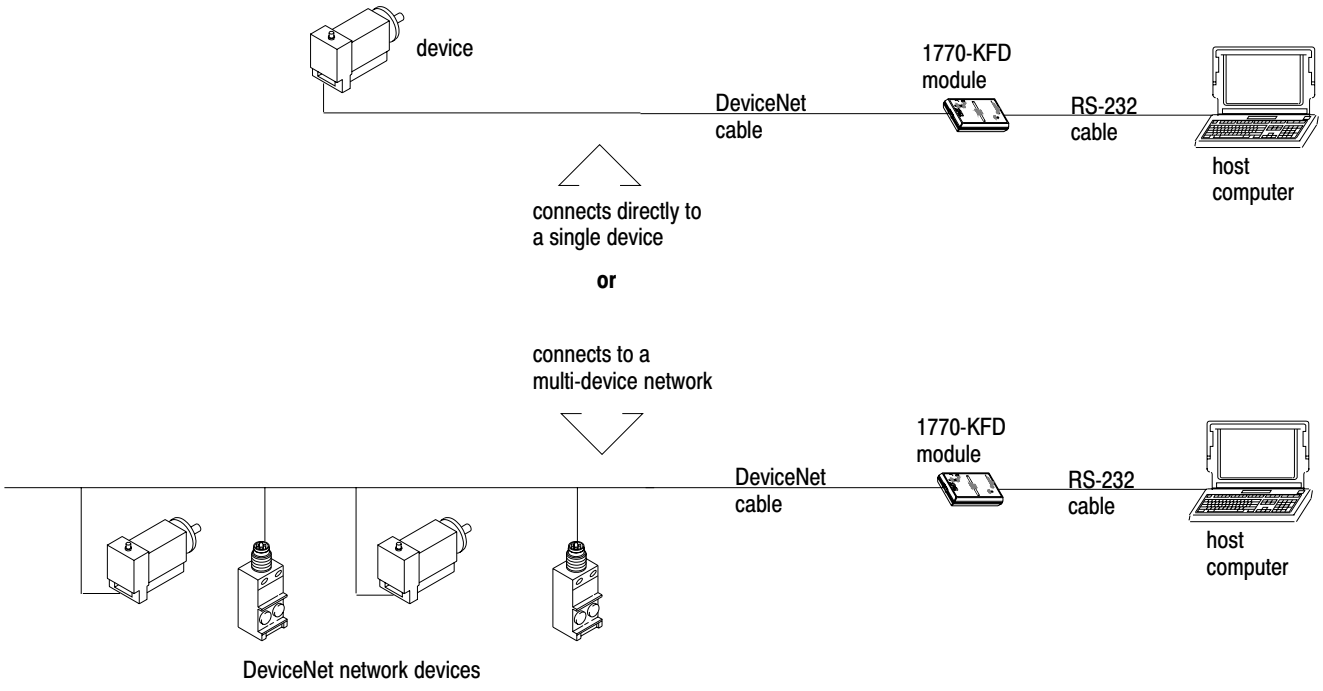
Module Communication Basics

Chapter Contents

This chapter introduces the 1770-KFD module as a RS-232 interface using the full-duplex DF1 communication-protocol. In addition, this chapter explains applicable DeviceNet objects.

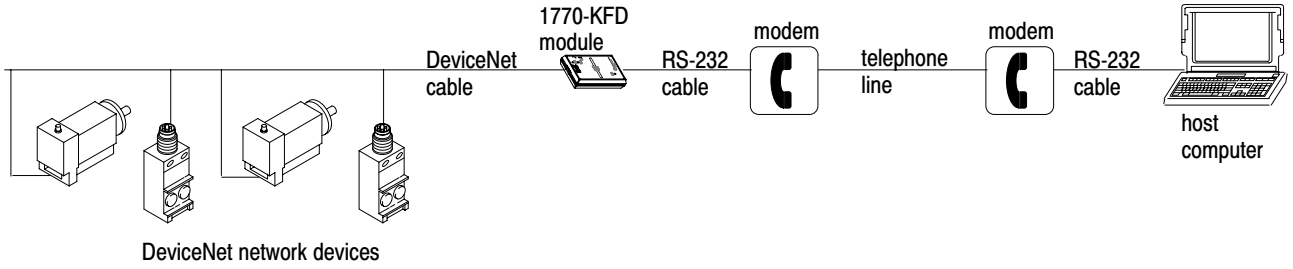
About the Module

The 1770-KFD module is an interface to the DeviceNet network for a host computer. The module connects to the host computer through a standard RS-232 serial communication-port and to the network via DeviceNet cable. Through the module, a host computer can connect to the network anywhere along its cable. This connection can be permanent or may be removed and reconnected as needed. In addition, the module can connect directly to a device via DeviceNet cable for a physical point-to-point connection.



The module can also provide access to a DeviceNet network remotely through a modem. You can connect the module to two types of standard modems:

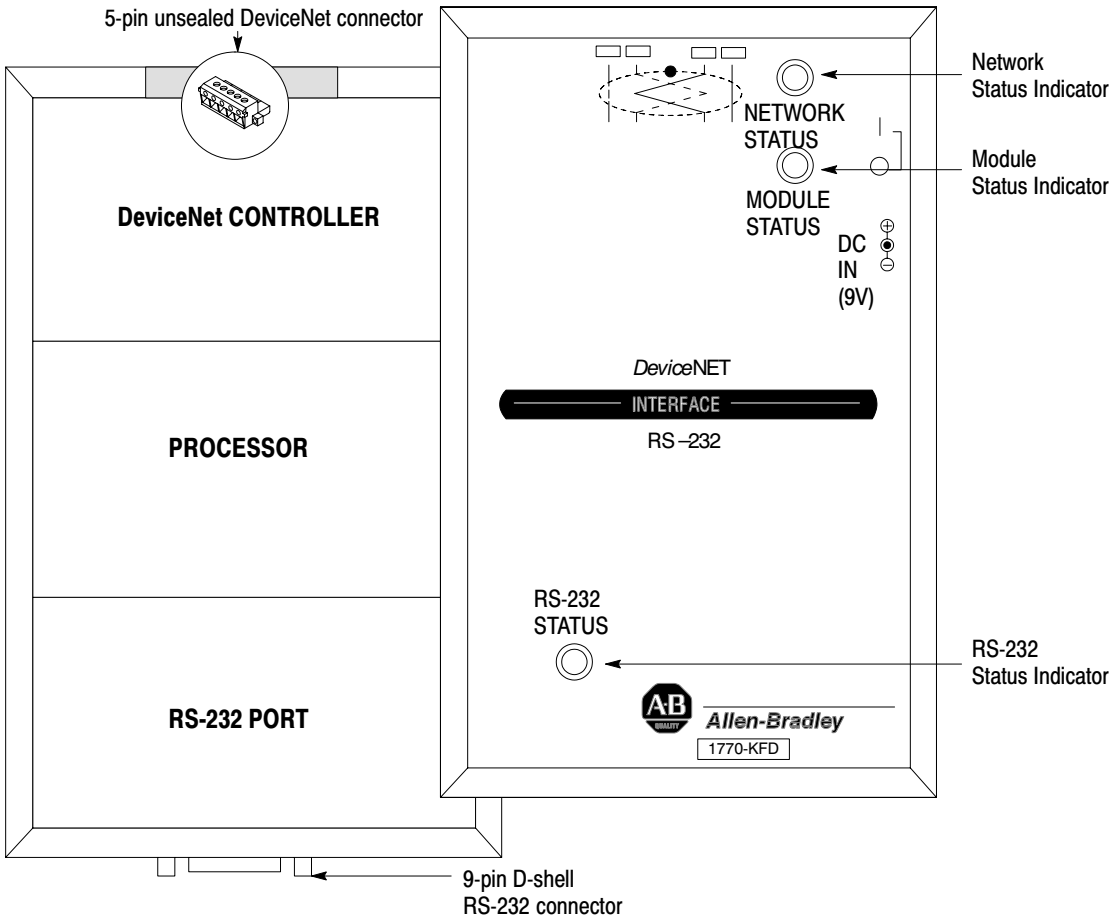
- DTE-controlled answer
- auto-answer



Detectable baud rates for the module include:

- 1200
- 2400
- 4800
- 9600
- 19200
- 38400
- 57600

As illustrated below, the module contains all the components needed to make it a stand-alone, portable interface.



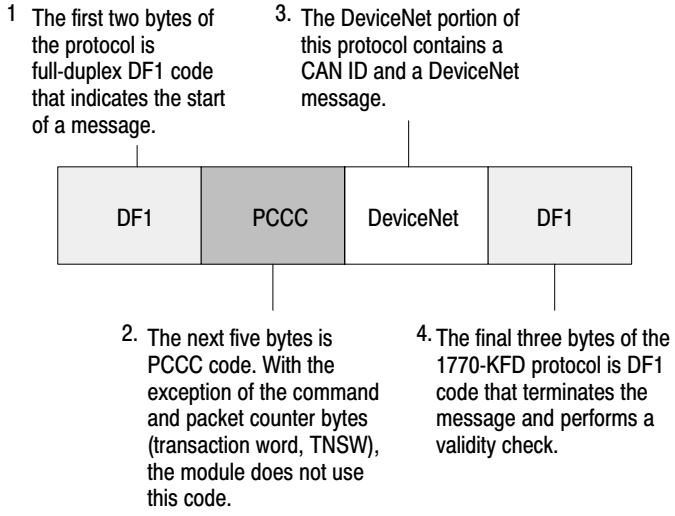
Serial Connection

The module can connect to any PC compatible desktop, laptop, and notebook computer that is equipped with an RS-232 serial communication-port. In addition, the module can connect to any RS-232 device, which can include devices such as embedded controls and RS-232 sensor products. During normal operation, a connection between the host computer and the module is continuously open.

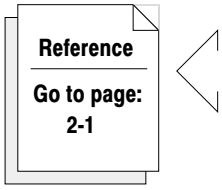
The RS-232 link is limited in bandwidth when compared to the DeviceNet network; however, the module has the ability to buffer incoming network data so that nothing is lost. The speed of data transfer between the host and module, or baud rate, is determined by the host; the module *autobauds* to match this baud rate.

Link-Level Communication

The module uses a layered link-protocol, illustrated below, to facilitate communication between your host and module. The protocol layers described in the following sections make up the required protocol between a host and module over the RS-232 serial-link.



DF1 Full-Duplex Protocol

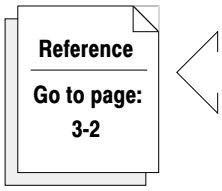


Full-duplex combines features of subcategories D1 (data transparency) and F1 (two-way simultaneous transmission with embedded responses) of ANSI x3.28. All communication between the host and the module uses this protocol.

PCCC Protocol

PCCC protocol is descriptive data after the DF1 segment, which is placed at the beginning of a DeviceNet message. Four of the five inserted fields do not change; none of the fields need to be manipulated. However, the fifth field is a packet counter that must be incremented with each message. Note that each field is one byte long except for this fifth field which is two bytes.

DeviceNet Protocol



The DeviceNet protocol pertains to the actual network data embedded inside a message exchanged between the module and host. This information is being sent or received on or from a DeviceNet network. In some cases, this information is destined for the module only.

When the module receives a DeviceNet message from the network, it attaches this packaging to the data before sending it to the host. Before transmitting a DeviceNet message onto the network, the module strips away the DF1 and PCCC packaging.

The Module's Message Format

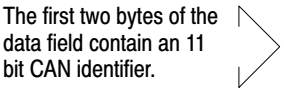
A basic DeviceNet message is packaged with DF1 and PCCC prefix data and DF1 suffix data as illustrated below. For message transmission between your host and module, the 1770-KFD protocol uses the following message format:

Layer	Name	Type	Description
DF1	DLE	USINT	DLE = 10_{hex}
	STX	USINT	STX = 02_{hex}
PCCC	DST	USINT	Destination = 0 (unused)
	SRC	USINT	Source = 0 (unused)
	CMD	USINT	Command = $0C_{hex}$ (DeviceNet message)
	STS	USINT	Status = 0 (unused)
	TNSW*	UINT	packet counter (incremented every message)
Data	CAN ID	UINT	CAN Indentifier (DeviceNet format)
	Data	array of USINT	CAN Data (DeviceNet format, 8 bytes maximum)
DF1	DLE	USINT	DLE = 10_{hex}
	ETX	USINT	ETX = 03_{hex}
	BCC	USINT	Block Check Character

Shading indicates values that do not change. They are listed for verification purposes only.

The TNSW (transaction word) is two bytes long.

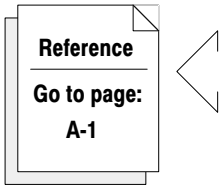
The first two bytes of the data field contain an 11 bit CAN identifier.



For more information about the 8 bit *Unsigned Short Integer* (USINT) and the 16 bit *Unsigned Integer* (UINT), refer to the Data Management portion of the DeviceNet Specification, Volume I.

Module-Supported Objects

There are several module-supported objects:



Object	Function
DeviceNet Object	used to provide the module's configuration and status data the DeviceNet object is always *created* but cannot be deleted
DF1 Object	used to provide diagnostic data about the DF1 link
Identity Object	used for general identification as a DeviceNet network device
Link Object	used to configure and maintain CAN identifier screeners which facilitate connected network-messaging
Power Management Object	used to provide information about and to configure the product power supply
RS-232 Object	used to provide information about and to configure the serial-link's baud rate

Important: Devices on the network cannot directly access these objects within the module. They can be accessed only through the host over its RS-232 port.

Full-Duplex DF1 Protocol

Chapter Contents

This chapter describes the data link-layer protocol your host and module use to communicate. In addition, this chapter provides full-duplex asynchronous link-protocol information as well as information on data link-layer message-packet fields.

Because you are connecting an asynchronous interface-module to a computer, you must program the computer to understand and to issue the proper protocol-character sequences. Use this information to determine how to program your host so that it can communicate with your module.

This chapter explains how to use DF1 protocol to communicate with the 1770-KFD interface module. However, it is not a comprehensive explanation. For more information about DF1 protocol, refer to the Data Highway/Data Highway Plus™/DH-485 Communication Protocol and Command Set reference manual.

About DF1 Protocol

DF1 protocol is an Allen-Bradley link protocol that combines features of subcategories D1 (data transparency) and F1 (two-way simultaneous transmission with embedded responses) of ANSI x3.28.

DF1 protocol:

- is used over a point-to-point link to facilitate two-way simultaneous transmission
- is intended for high performance applications that require the highest possible throughput from the available medium

Character Transmission

The module sends data serially over the RS-232-C/RS-422-A interface, one byte at a time. The transmission format conforms to ANSI x3.16, CCITT V.4, and ISO 1177. Your computer should conform to this mode of transmission:

- link protocol = full-duplex, DF1
- message type = asynchronous
- data size = eight bits
- parity = none
- stop bit = one
- validity check = block check character (BCC)

Transmission Symbols

Full-duplex protocol is character-oriented. It uses the ASCII control characters in the following table, extended to eight bits by adding a zero for bit 7. DF1 combines these characters into **control** and **data** symbols.

For the standard definition of these characters, refer to ANSI x3.4, CCITT V.3, or ISO 646.

Abbreviation	Hexadecimal value	Binary value
STX	02	0000 0010
ETX	03	0000 0011
ENQ	05	0000 0101
ACK	06	0000 0110
DLE	10	0001 0000
NAK	15	0000 1111

The following symbols are used for communication between your host and module:

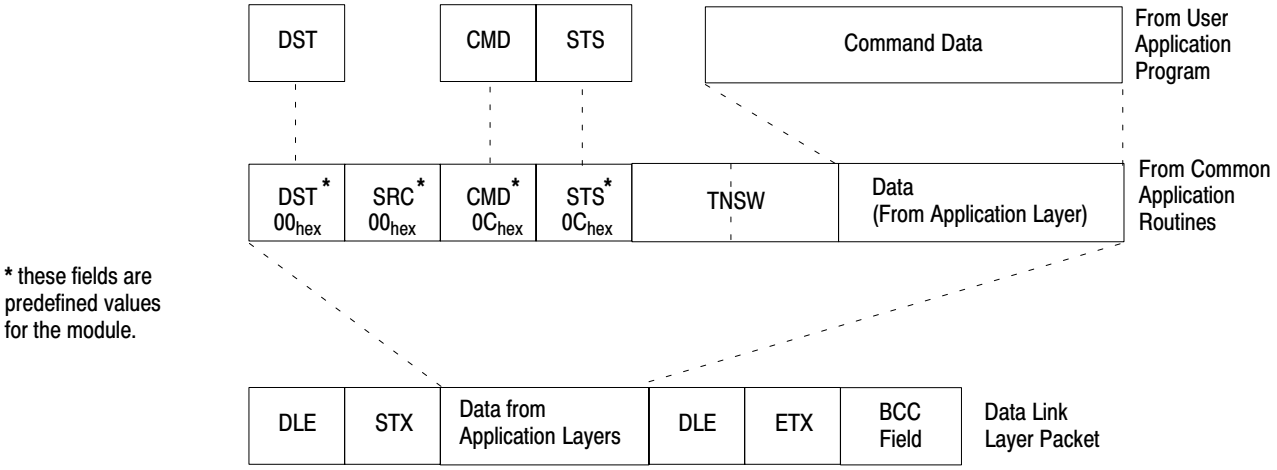
Symbol	Type	Meaning
DLE STX	control symbol	sender symbol that indicates the start of a message
DLE ETX BCC	control symbol	sender symbol that terminates a message
DLE ACK	control symbol	response symbol that signals when a message is successfully received
DLE NAK	control symbol	response symbol that signals when a message is not successfully received
DLE ENQ	control symbol	sender symbol that requests retransmission of a response symbol from the receiver
APP DATA	data symbol	single character data values between 00-0F and 11-FF. Includes data from application layer including user programs and common application routines
DLE DLE	data symbol	symbol that represents the data value 10_{hex}

Message Packet Fields

A data link-layer message packet starts with DLE STX and ends with DLE ETX BCC, with application layer data in between. Data symbols are only inside a message packet. Response symbols (DLE ACK and DLE NAK) can also be transmitted inside a message packet even though they are not considered part of it. Response symbols transmitted within a message packet are referred to as embedded responses.

Figure 2.1 shows the format of a full-duplex message packet and the layer at which each portion is implemented. Note the BCC field at the end of each message packet.

Figure 2.1
Packet Format for Full-Duplex Protocol



Block Check

The block-check character (BCC) is a way of checking each message-packet transmission's accuracy. It is the two's complement of the eight-bit sum (module-256 arithmetic sum) of all application layer data-bytes between a DLE STX and a DLE ETX BCC. It does not include response symbols.

A message packet containing 08, 09, 06, 00, 02, 04, and 03 (decimal), appears as:

10	**02**	**08 09 06 00 02 04 03**	**10**	**03**	**E0**
DLE	**STX**	**APP DATA**	**DLE**	**ETX**	**BCC**

The sum of the application data bytes in this message packet is 32 decimal or 20_{hex}. The BCC is the 2's complement of this sum, or $E0_{hex}$ as shown in the binary calculation below:

0010 0000	**20_{hex}**
1101 1111	**1's compliment**
+1	

1110 0000	**2's compliment ($E0_{hex}$)**

To quickly determine a BCC value, add up the application-layer byte's hex values. If the total is greater than 100_{hex}, drop the most significant digit. Then, subtract the result from 100_{hex}. This should give you the BCC. For example, the sum of 20_{hex}, then:

100_{hex}
-20_{hex}

$E0_{hex}$

Important: To transmit the value 10_{hex}, you must use the data symbol DLE DLE. However, only **one** of these DLE data bytes is included in the BCC sum. For example, to transmit the values 08, 09, 06, 00, 10, 04, and 03_{hex}, you would use the following message symbols:

10	**02**	**08 09 06 00 10 10 04 03**	**10**	**03**	**D2**
DLE	**STX**	**APP DATA (DLE DLE)**	**DLE**	**EXT**	**BCC**

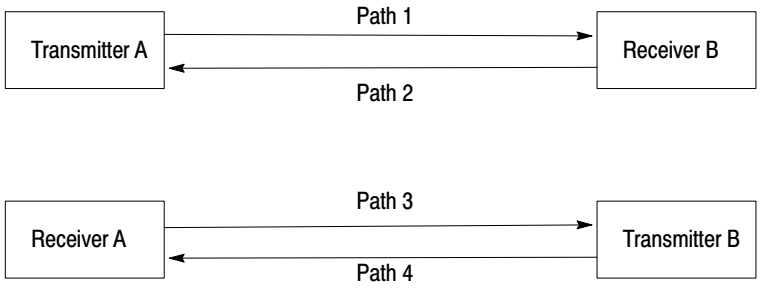
In this case, the sum of the application-layer data bytes is $2E_{hex}$ because only one DLE byte is included in the BCC; therefore, the BCC is $D2_{hex}$.

Important: The BCC algorithm provides only a medium level of data security. It cannot detect transposition of bytes during transmission of a packet. It also cannot detect the insertion or deletion of the value zero within a packet.

Two-Way Simultaneous Operation

To communicate with full-duplex protocol, the network uses two physical circuits (cable systems) for two-way simultaneous message transmission. Figure 2.2 shows an example of two-way simultaneous operation.

Figure 2.2
Data Paths for Two-Way Simultaneous Operation



On the first circuit, transmitter A sends messages to receiver B (data path 1) and receiver A sends response control symbols (DLE ACK, DLE NAK) to transmitter B (data path 3).

On the second circuit, transmitter B sends messages to receiver A (data path 4) and receiver B sends response control symbols (DLE ACK, DLE NAK) to transmitter A (data path 2).

All messages and symbols on the first circuit are traveling in the same direction (node A to node B) and all messages and symbols on the second circuit are traveling in the opposite direction (B to A).

To implement four data paths with only two physical circuits, a software multiplexer is needed to combine the message symbols with the response symbols going in the same direction.

At the other end of the link, a software separator divides the message symbols from the response symbols. The internal software sends the message symbols to the appropriate receiver and the response symbols to the appropriate transmitter.

Environment Definition

To fully define the environment's protocol, the transmitter needs to know where to get the message it sends, and the receiver must have a means of disposing of messages. These are implementation-dependent functions which are respectively called the *message source* and the *message sink*.

We assume that the message source:

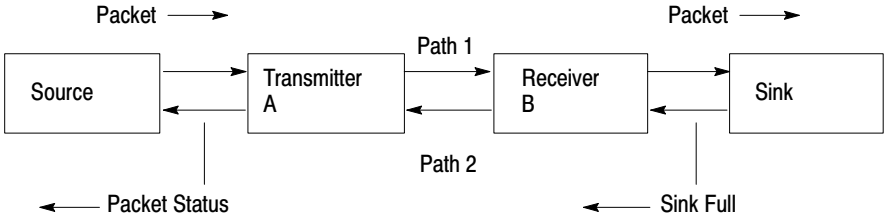
- supplies one message at a time upon request from the transmitter
- requires notification of the success or failure of the transfer before supplying the next message

When the message source is empty, the transmitter waits in an inactive state until a message is available.

Whenever the receiver has received a message successfully, it attempts to give it to the message sink. If the message sink is full, the receiver must be notified.

Figure 2.3 shows the protocol environment for message symbols from transmitter A to receiver B (path 1) and response codes from receiver B to transmitter A (path 2).

Figure 2.3
Protocol Environment



How the Transmitter Operates

Whenever the message source can supply a message and the transmitter is not busy, it sends a message packet. It then starts a timeout, and waits for a response symbol.

When a DLE ACK is received, the message has been successfully transferred. After signaling the message source that the message was successfully transmitted, the transmitter proceeds with the next message.

If a DLE NAK is received, the message is retransmitted. The transmitter restarts the timeout and waits again for a response. This can be repeated several times. You can set a limit to the number of times a message can be retransmitted for each module. If this limit is exceeded, the message source is informed of the failure and the transmitter proceeds with the next message.

If the timeout expires before a response is received, the transmitter sends a DLE ENQ to request a retransmission of the last response. It restarts the timeout and waits for a response.

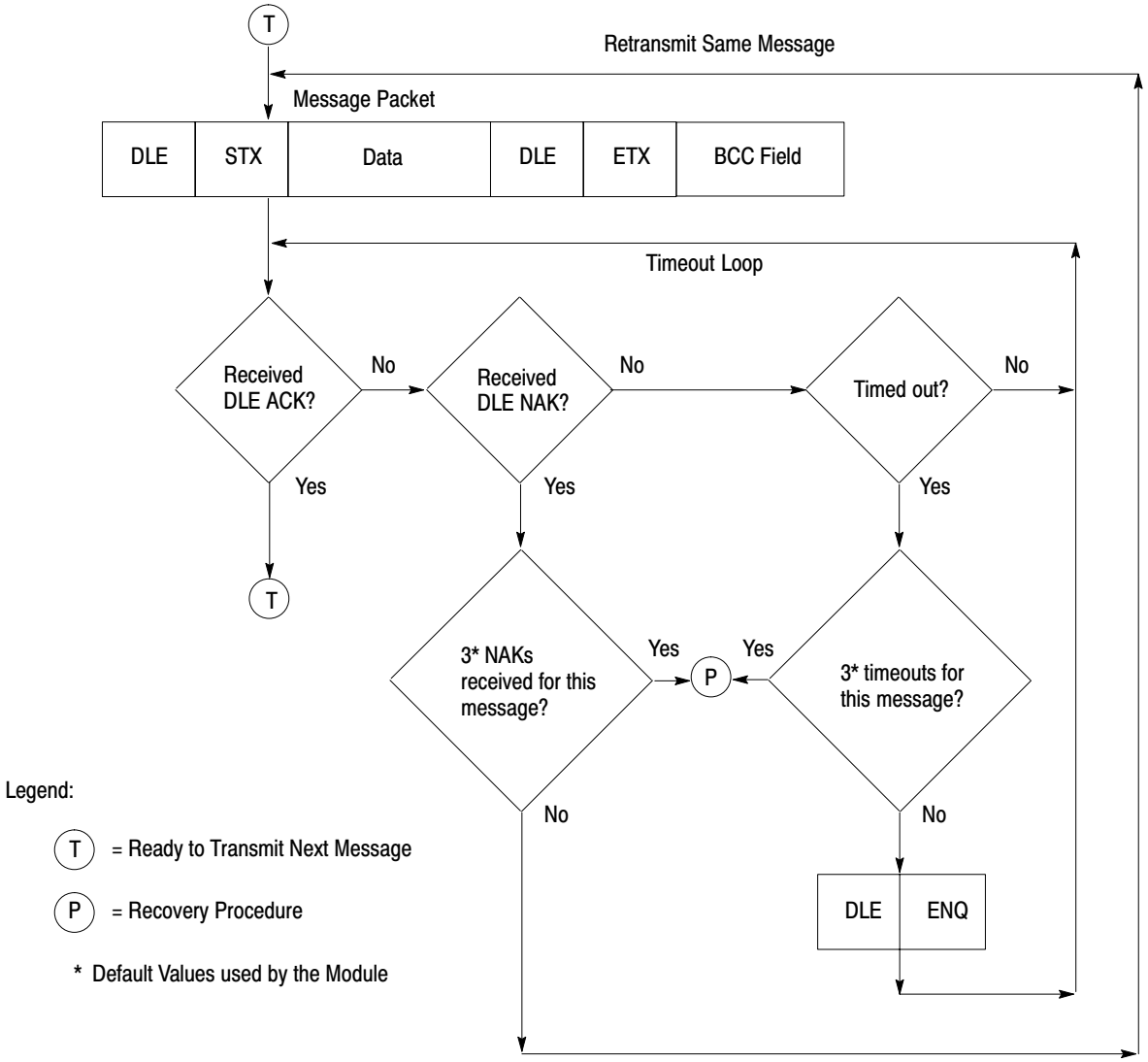
You can also set a limit on the number of timeouts that are allowed per message. If the enquiry (ENQ) limit is exceeded, the transmitter signals the message source that the transmission has failed, and the transmitter proceeds to the next message.

Since there are only two response symbols defined, all other symbols are undefined or invalid. If an invalid or undefinedresponse symbol is received, the transmitter ignores it. A more precise and detailed description of the actions of the transmitter appears in the following structured English procedure.

```
TRANSMITTER is defined as
    loop
        Message=GET-MESSAGE-TO-SEND
        Status=TRANSFER(Message)
        SIGNAL-RESULTS(Status)
    end loop
TRANSFER (Message) is defined as
    initialize nak-limit and enq-limit
    SEND(Message)
    start timeout
loop
    WAIT for response on path 2 or timeout.
    if received DLE ACK then return SUCCESS
    else if received DLE NAK then
        if nak-limit is exceeded then return FAILURE
        else
            begin
                count NAK re-tries;
                SEND-MESSAGE(message);
                start timeout
            end
    else if timeout
        if enq-limit is exceeded then return FAILURE
        else
            begin
                count ENQ re-tries;
                send DLE ENQ on path 1;
                start timeout
            end
end loop
SEND (message) is defined as
    begin
        BCC = 0
        send DLE STX on path 1
        for every byte in the message do
            begin
                add the byte to the BCC;
                send the corresponding data symbol
        on path 1
            end
        send DLE ETX BCC on path 1
    end
GET-MESSAGE-TO-SEND
    This is an operating-system-dependent interface
    routine that waits and allows the rest of the
    system to run until the message source has supplied
     a message to be sent.
SIGNAL-RESULTS
    This is an implementation-dependent routine that
    gives the message source the results of the
    attempted message transfer.
WAIT
    This is an operating-system-dependent routine
    that waits for any of several events to occur
    while allowing other parts of the system to run.
```

Figure 2.4 is a flowchart of the software logic for implementing the transmitter.

Figure 2.4
Software Logic for Implementing Transmitter



Important: Depending on data-traffic and saturation level, you may need to wait for a reply from the remote node before transmitting the next message. You should implement an option that lets the user choose the maximum amount of outstanding messages that can exist at one time. We suggest a selectable range of one to three messages.

How the Receiver Operates

The receiver must be capable of responding to many adverse situations. Many problems can arise.

Some of the problems that can occur are:

- the message sink may be full, leaving the receiver with nowhere to put a message
- a message may contain a parity error
- the BCC may be invalid
- the DLE STX or DLE ETX BCC may be missing
- the message may be too long or too short
- a spurious control or data symbol may occur outside a message
- a spurious control symbol may occur inside a message
- the DLE ACK response may be lost, causing the transmitter to send a duplicate copy of a message already passed to the message sink

The receiver keeps a record of the last response sent to the transmitter. The value of this response is either ACK or NAK. It is initialized to NAK. When a DLE ENQ (enquiry) is received from the transmitter, the receiver sends the value of the last response.

The receiver ignores all input until receiving a DLE STX or a DLE ENQ. If anything other than a DLE STX or DLE ENQ is received on path one, the receiver sets the last response variable to a NAK.

If	Then
a DLE ENQ is received from the transmitter	the last response is sent and the receiver continues waiting for input
a DLE ACK is received	the BCC and message buffer are reset – and the receiver starts building the message

While building a message, all data symbols are stored in the message buffer and added to the BCC. If the buffer overflows, the receiver continues summing the BCC, but the data is discarded.

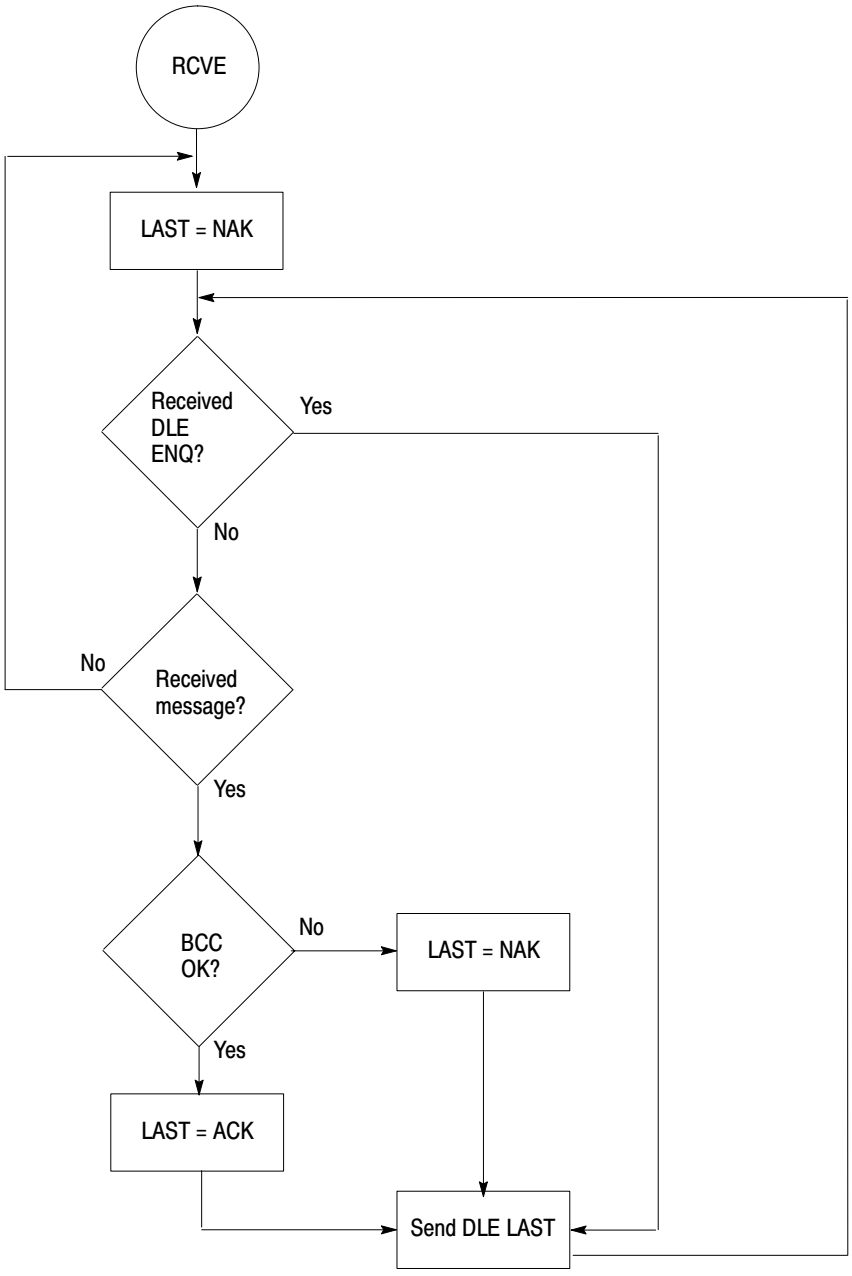
If	Then
any control symbols other than a DLE ETX BCC are received	the message is aborted and a DLE NAK is sent to the transmitter
a DLE ETX BCC is received	the error flag (the BCC), the message size, and the address (optional) are all checked
the BCC, message size, or address test fails	a DLE NAK is sent
duplicate message detection is enabled, the message is valid, and its header is the same as the previous message	the duplicate message is discarded without being executed and a DLE ACK is sent
duplicate message detection is enabled, the message is valid, but the header is not the same as the previous message	the message-sink's state is tested
if the message sink is full	a DLE NAK is sent
if the message sink is not full	the message is forwarded to the message sink, the header information is saved for the duplicate message detector, and a DLE ACK is sent

GET-CHAR is defined as an implementation-dependant function that returns one byte of data from the link interface hardware

```
RECEIVER is defined as
    variables
        LAST-HEADER is 4 bytes copied out of the last good message
        RESPONSE is the value of the last ACK or NAK sent
        BCC is an 8-bit block check accumulator
    LAST-HEADER = invalid
    LAST RESPONSE = NAK
    loop
        reset parity error flag
        GET-SYMBOL
        if DLE STX then
            begin
                BCC = 0
                GET-SYMBOL
                while it is a data symbol
                    begin
                        if buffer is not overflowed put
                        data in buffer
                        GET-SYMBOL
                    end
                if the control symbol is not a DLE ETX then send DLE
                NAK
                else if error flag is set then send DLE NAK
                else if BCC is not zero then send DLE NAK
                else if message is too small then send DLE NAK
                else if message is too large then send DLE NAK
                else if header is same as last message send a DLE ACK
                else if message sink is full send DLE NAK
                else
                    begin
                        send message to message sink
                        send a DLE ACK
                        save a last header
                    end
                end
        else if DLE ENQ then send LAST-RESPONSE
        else LAST-RESPONSE = NAK
    end loop
GET-SYMBOL is defined as
    loop
        GET-CHAR
        if char is not DLE
            begin
                add char to BCC
                return the char and data flag
            end
        else
            begin
                GET-CHAR
                if char is a DLE
                    begin
                        add char to BCC
                        return DLE and data flag
                    end
                else if char is an ACK or NAK send it to the transmitter
                else if char is an ETX
                    begin
                        GET-CHAR
                        add char to BCC
                        return ETX with a control flag
                    end
                else return char with a control flag
            end
    end loop
GET-CHAR is defined as an implementation-dependant function that returns one byte
of data from the link interface hardware
```

Figure 2.5 is a flowchart of the software logic for implementing the receiver.

Figure 2.5
Receiver for Full-Duplex Protocol



Host/Module Message Transmission Examples

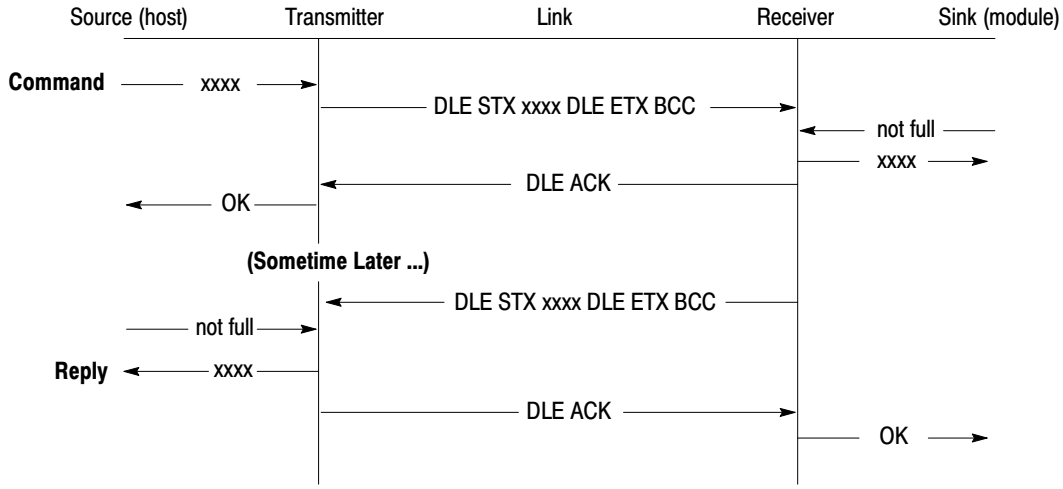
The following sections illustrate basic message transmissions and responses between a host and its module. The purpose of these examples is to illustrate how DF1 protocol is used; therefore, the DeviceNet message within is unimportant at this point. Chapter 4 illustrates the nested DeviceNet messages that are used to access the module's local objects.

Important: PCCC protocol is not illustrated in the following figures. However, it resides in each of these local messages (between the DF1 prefix data and the DeviceNet message).

Normal Message Transmission

An example of a normal message transfer, shown below in Figure 2.6, illustrates a DeviceNet message wrapped in DF1 protocol.

Figure 2.6
Normal Message Transfer



Note: xxxx = DeviceNet message

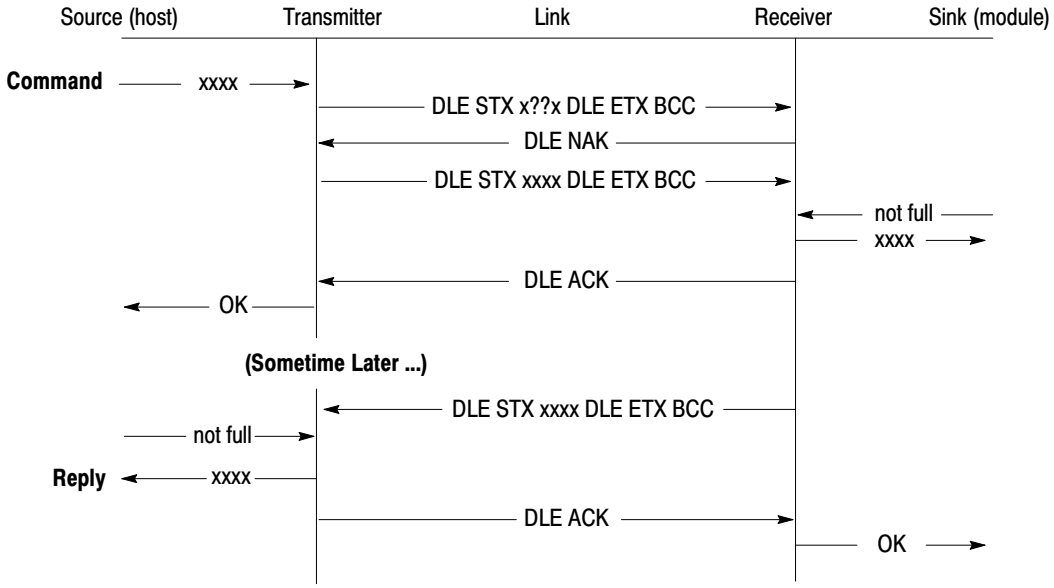
In this figure:

- the transmitter sends the message to the receiver
- the sink sends a "not full" message
- the receiver sends the message to the sink and sends a DLE ACK to the transmitter
- the transmitter tells the source that the message was delivered
- reply is successfully returned from the network

When a Message Transfer Fails

Figure 2.7 illustrates what happens when the module receives corrupted data, shown as "??."

Figure 2.7
Message Transfer with NAK



Note: xxxx = DeviceNet message

In this figure:

- the transmitter sends a corrupted message to the receiver and the receiver responds with a DLE NAK
- the transmitter retransmits; the transmission is successful
- reply is successfully returned from the network

In addition to the example above, a **DLE NAK** can occur if:

- a reply is corrupted
- the receiving device's buffers are full

Both the host and module use a DLE NAK. If the module sends your host an invalid message, your host will reply with a DLE NAK.

When a Retransmission is Requested

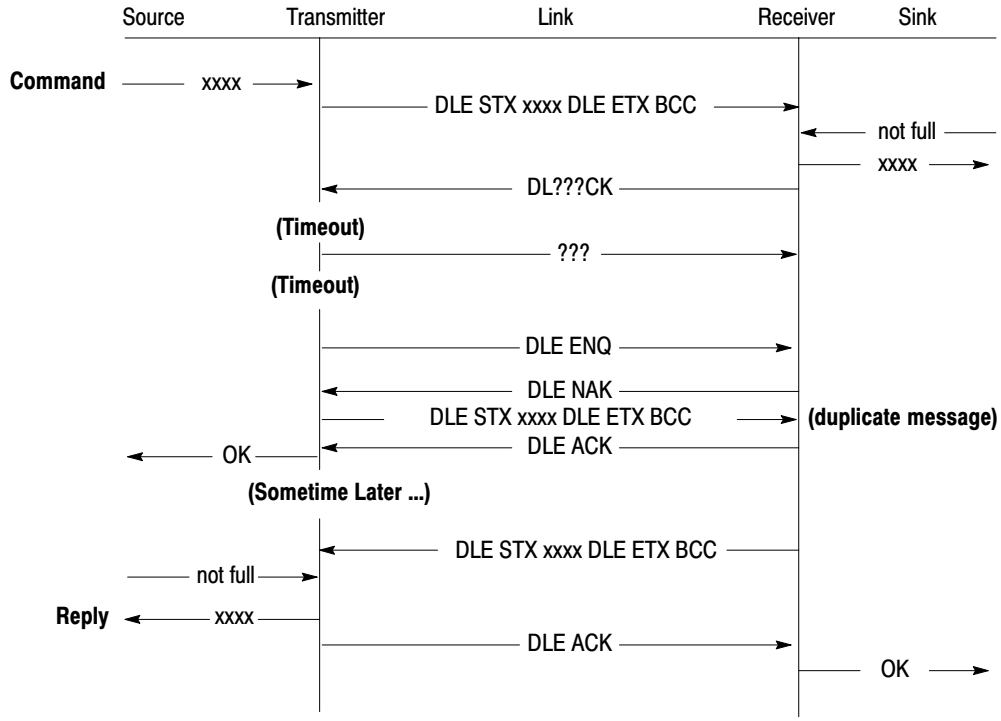
In some cases, as illustrated in Figure 2.8, the host or module can request a message retransmission.

Figure 2.8
Message Transfer with Retransmission

Note: xxxx = DeviceNet message

In this figure:

- Noise destroys the DLE ACK
- The transmitter times-out waiting for the response and sends a DLE ENQ which is also destroyed by noise.
- because of the invalid characters, the receiver changes its last response to a DLE NAK
- since the DLE ACK was destroyed, the transmitter sends a DLE ENQ (enquiry), and the receiver returns the DLE NAK
- the transmitter retransmits the message and the receiver sends an ACK
- the receiver discards the duplicate message (if duplicate message detection is enabled on your module)
- reply is successfully returned from the network



Module Communication Over the DF1 Link

Chapter Contents

This chapter explains each of the components needed for communication between the host and module over the DF1 protocol link.

Module Serial-Link Autobaud

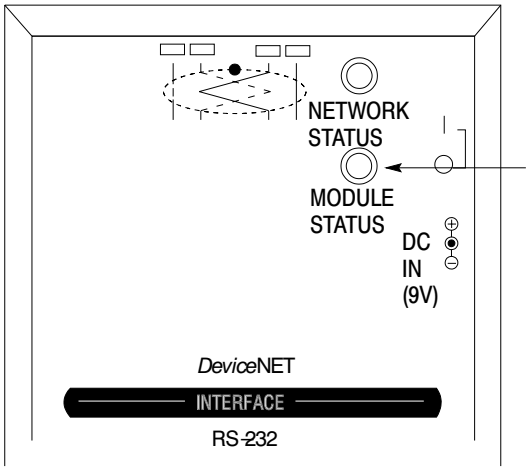
Your module needs to detect the baud rate between it and the host upon power-up or when the *heartbeat* between your host and module stops. To accomplish this, your host must send the following message until the module responds.

hexadecifmal value	**10**	**05**
symbol	**DLE**	**ENQ**

hexadecifmal value	**10**	**05**
symbol	**DLE**	**ENQ**

The module returns the following message to your host when serial-link autobaud is complete. This message indicates to your host that the module is ready to communicate.

hexadecifmal value	**10**	**15**
symbol	**DLE**	**NAK**



The Module Status Indicator flashes green when your module is in the serial-link autobaud detection-state and turns solid green when the baud rate is acquired.

The RS-232 Heartbeat

Since there is no DF1-level heartbeat, the DCD signal at the module's RS-232 port is used to indicate that the host is active. If the DCD is lost for 500 milliseconds or more, the module:

- drops DTR for 1 second
- disables the CAN network
- deletes all connection links
- returns to serial-link autobaud state

Host and Module Local Connections

To communicate with the module over a predefined local-connection, the host wraps DeviceNet protocol with DF1 and PCCC protocol and uses the invalid CAN ID:

$FFFF_{hex}$

The message's body format is set to DeviceNet 8/8:

Class = 8 bit integer Instance ID = 8 bit integer

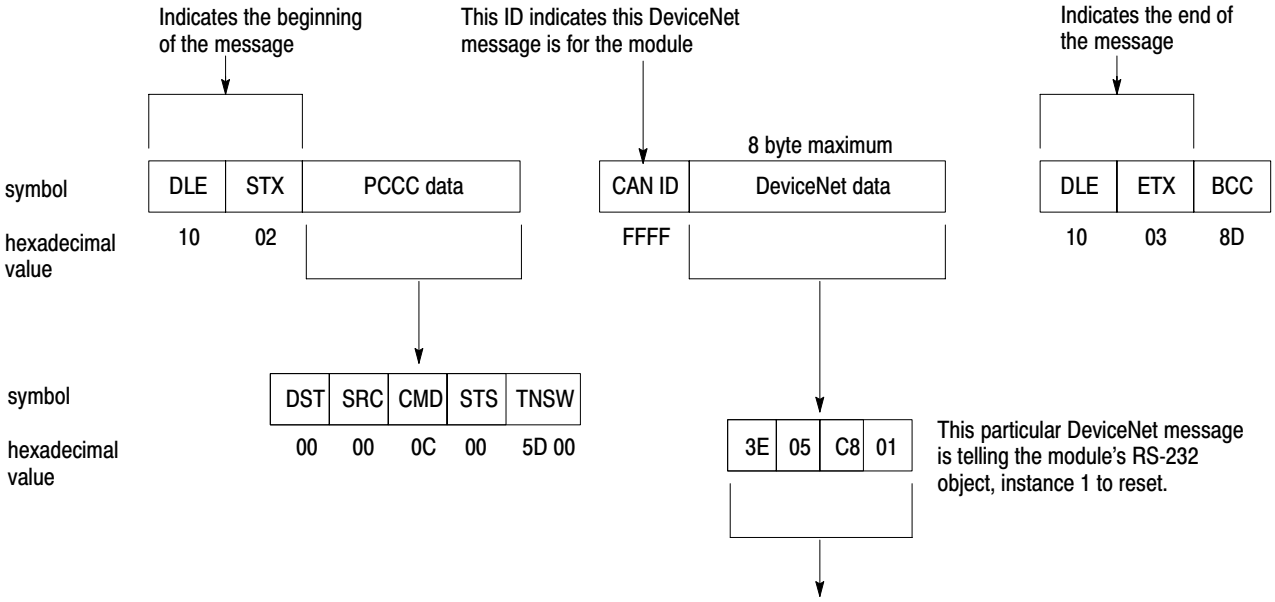
Layer	Name	Description
DF1	DLE	DLE = 10_{hex}
	STX	STX = 02_{hex}
PCCC	DST	Destination = 0 (unused)
	SRC	Source = 0 (unused)
	CMD	Command = 0C hex (DeviceNet message)
	STS	Status = 0 (unused)
	TNSW	Packet Counter
$FFFF_{hex}$ → Data	CAN ID	CAN Indentifier (DeviceNet format)
	Data	CAN Data (DeviceNet format, 8 bytes maximum)
DF1	DLE	DLE = 10_{hex}
	ETX	ETX = 03_{hex}
	BCC	Block Check Character

For more information about DeviceNet protocol, refer to the DeviceNet Specification, Volume 1.

Accessing Local Objects

The purpose of using FFFF to address the module is to access objects residing within the module. Each of the module-supported objects, listed in chapter 1, have a specific function. These functions range from maintaining the host/module local connection to facilitating communication on the DeviceNet network. Objects make it possible for the module to perform its duties as the host's interface to the network. The code used to manipulate these objects is inside the DeviceNet message.

The example below illustrates how the host accesses local objects within the module. In this particular example, the host is resetting the module's RS-232 object.



hex value	description
3E	This is the node address. 3E in decimal is 62. This value is arbitrary.
05	This is the service code. 05 means Reset.
C8	This is the class code. It identifies which object the host is trying to access. C8 is the RS-232 object.
01	This value is indicating an instance of the selected object. 01 is instance 1.

Inside the DeviceNet Message

When the FFFF address is used, the module reads the DeviceNet message to follow; it does not pass the message to the network. The following list is the structure of a DeviceNet message sent from a host to a module.

- the module's node address
- service-code (the command you give to the specified object)
- class-code of the object for which the service-code is sent
- the object's instance for which the service-code is sent
- the instance's attribute for which the service-code is sent (only when applicable)
- the value you need to apply to the specified attribute (only when applicable)

Important: All code in any message between the host and module is hexadecimal, including the values within the DeviceNet message.

The following table is a listing of hexadecimal common-service codes that are used to manipulate your module's objects.

Service name	Hexadecimal service code
get_attribute_single	0E
set_attribute_single	10
reset	05
start	06
stop	07
create	08
delete	09

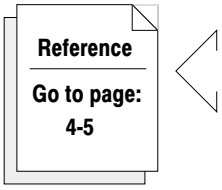
Before Network Communication

Before you can use the module to communicate on the network, you must first access the appropriate objects within the module to:

- stop the module if you need to re-initialize
- configure node address and baud rate
- start the module

Stop Service

If your module has already been activated, we recommend that you send a **stop service** to the module's DeviceNet object. This makes sure that your module is not actively receiving and sending messages and is ready to be configured.



Configure Node Address and Baud Rate

Before your module can communicate on a DeviceNet network, you must configure the node address and baud rate. This is done by sending the *set_attribute_service* to the module's DeviceNet object (Class DeviceNet, Instance 1, Attributes 1 and 2).

Important: The module uses two baud rates. The first is the rate at which data is exchanged between it and the host; this is where the serial-link autobaud function is applied. The second is the rate at which data is sent to and received from the devices on the DeviceNet network; this is where the *set_attribute_service* to the DeviceNet object is applied.

Start Service

To activate DeviceNet network communication you must send a **start service** to the module's DeviceNet object. Upon receiving this message, the module:

- validates the node address and baud rate
- initializes and starts the CAN chip
- performs a duplicate node address check

When the Network-Access State Machine is completed the module returns a response. The duration between sending the start service message and receiving the start response is arbitrary; however, this time is never less than two seconds.

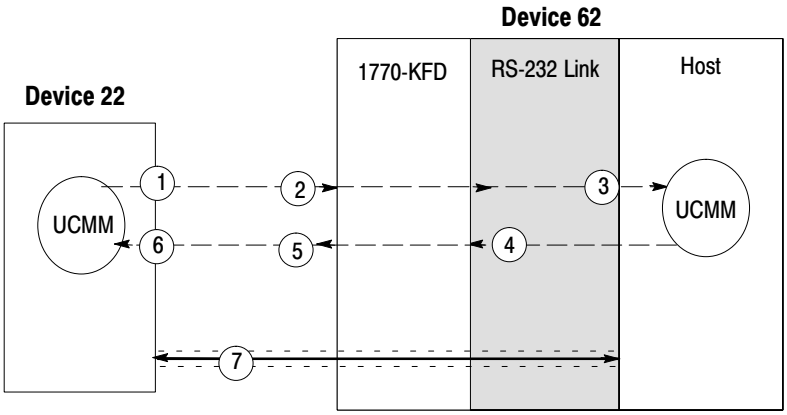
For more information about the Network-Access State Machine, refer to the DeviceNet Specification, Volume 1.

Communication On The DeviceNet Network

As an interface for your host, the module sends messages to and receives messages from devices on the network. It passes messages from your host to the network and determines which DeviceNet messages to accept from the wire. The module also supports unconnected messages by passing them through to the host's Unconnected Message Manager (UCMM).

Sending and Receiving Unconnected DeviceNet Messages

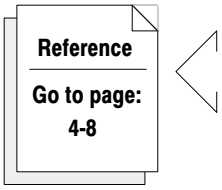
Before a connection has been established, a device may contact the host through its UCMM. A DeviceNet device, when online, always has two unconnected message ports available, one for receiving and one for sending UCMM messages. A UCMM object resides in the host and not in the module; therefore, any unconnected message the module receives is passed directly to the host. It is through this unconnected messaging that a connection is established between the host and a DeviceNet device.



1. Device 22 needs to set up a connection with the host (device 62). Device 22's UCMM sends an unconnected message (open-connection request) via the DeviceNet network.
2. The unconnected message is accepted through the module's UCMM receive port.
3. The unconnected message travels through the module, over the RS-232 serial-link, and to the host's UCMM object.
4. The host sends a connection-request response via the module.
5. Through the module's UCMM transmit port, the response is sent over the network to the requesting device.
6. Device 22 receives the open-connection response from the host through its UCMM.
7. A connection is created. This connection will remain as long as Device 22 and the module are online. All consequent communication between Device 22 and the host will occur over this connection unless either the module or device goes offline. If either component goes offline , the connection must be renegotiated through the UCMM function.

Sending DeviceNet Messages

The module transmits onto the network any DeviceNet message it is given by the host, provided the CAN ID is valid. Messages bound for the network do not pass through screeners or any other similar objects in the module.



Receiving Connected DeviceNet Messages

After a connection has been established, the host must create screeners in the module. Screeners are created for every device connection from which you need to receive messages, other than unconnected messages. A different screener is created for each of these device connections. Each screener is responsible for recognizing a particular CAN ID; they pass any message with a matching CAN ID to the host. Screeners can be created and deleted as needed.

To create a screener, the host sends a *create service* message to the module's link object, class code = CB_{hex}, instance 00_{hex}. The CAN ID that you want screened becomes the first part of the create service. Screeners stay in place until deleted by the host using the *delete service* or when the module goes offline.

Link Communication Example

Chapter Contents

This chapter presents full-duplex, DF1-communication examples between a host and module. There is an example of each major task you perform to enable network communication via the 1770-KFD interface module. These include:

- initializing your module
- creating screeners
- deleting screeners

This chapter's examples were created with the following assumptions in mind.

- The hexadecimal value, 3e (decimal value = 62) is your host's node address.

Important: Your host's node address is completely arbitrary. You can set its ID to whatever value fits your structure and the DeviceNet network's specifications.

- PCCC protocol, which is not included in these examples, is present during communication between a host and module.
- The PCCC protocol packet-counter (TNSW), though not shown, is incremented in each message.
- The DF1 block check character, though not shown as a true value in these examples, is calculated for each message exchanged between a host and module.

Initializing Your Module

To initialize your module, perform the processes listed below.

Important: If you are *re-initializing* your module, you must begin by resetting your module. If you are initializing your module for the first time, skip the module reset and begin with the serial-link autobaud function.

To re-initialize your module, begin here.

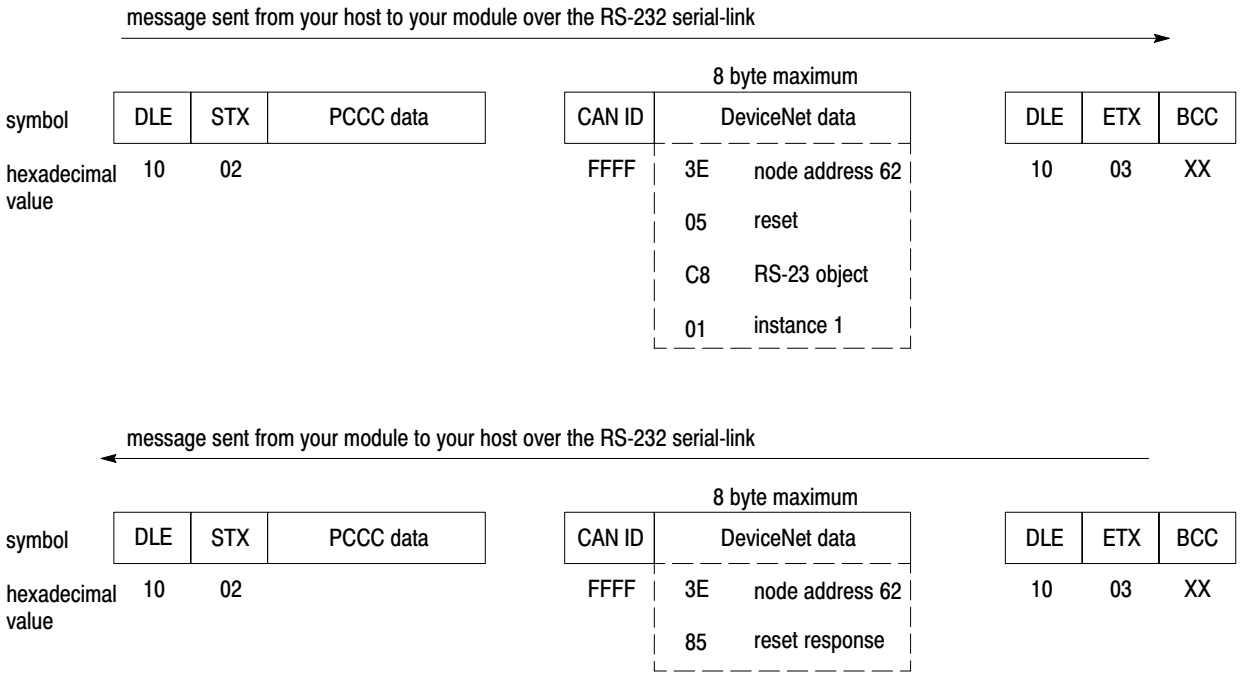
If you are initializing your module for the first time, begin here.

Process	Command	Hexadecimal value	Corresponding local object	Object's class code (in hex)
module reset	reset	05	RS-232	C8
serial-link autobaud	DLE ENQ	10 05	N/A	N/A
stop service	stop	07	DeviceNet	03
set node address	set_attribute_ single	10	DeviceNet	03
set baud rate	set_attribute_ single	10	DeviceNet	03
start service	start	06	DeviceNet	03

This step is for re-initialization only.

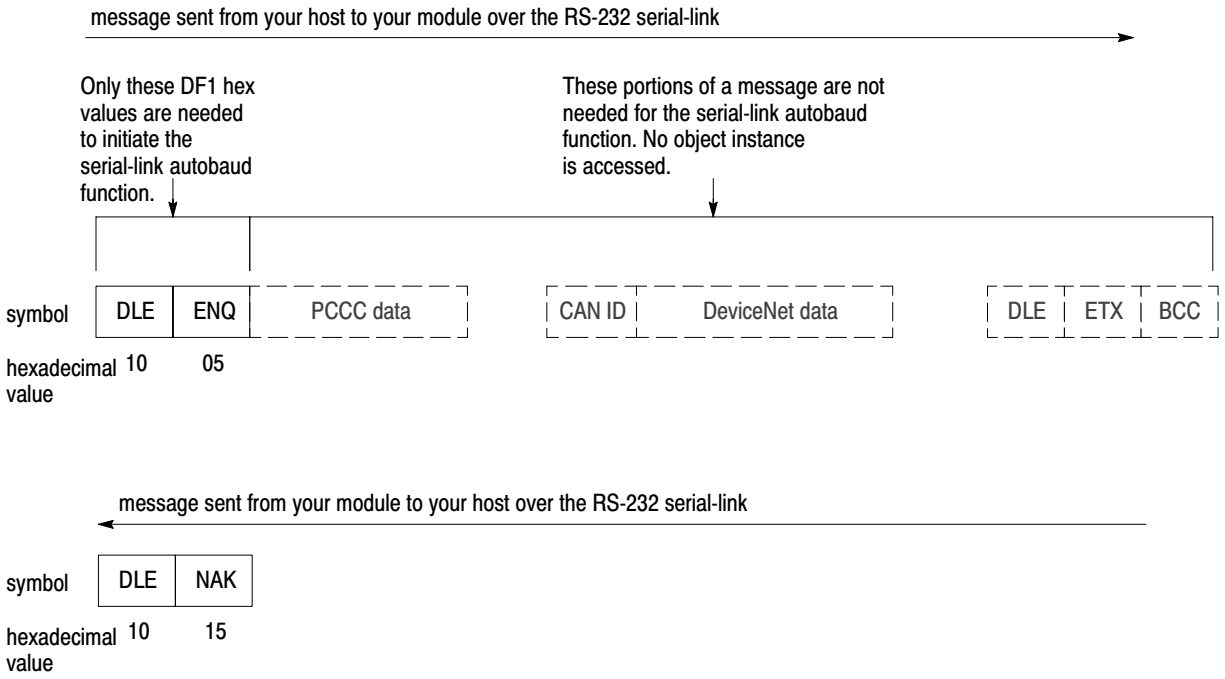
Module Reset

Since you will be configuring your module, you need to clear the module by resetting it. To reset your module, send the *reset* command to the module's RS-232 object, instance 1.



Serial-Link Autobaud

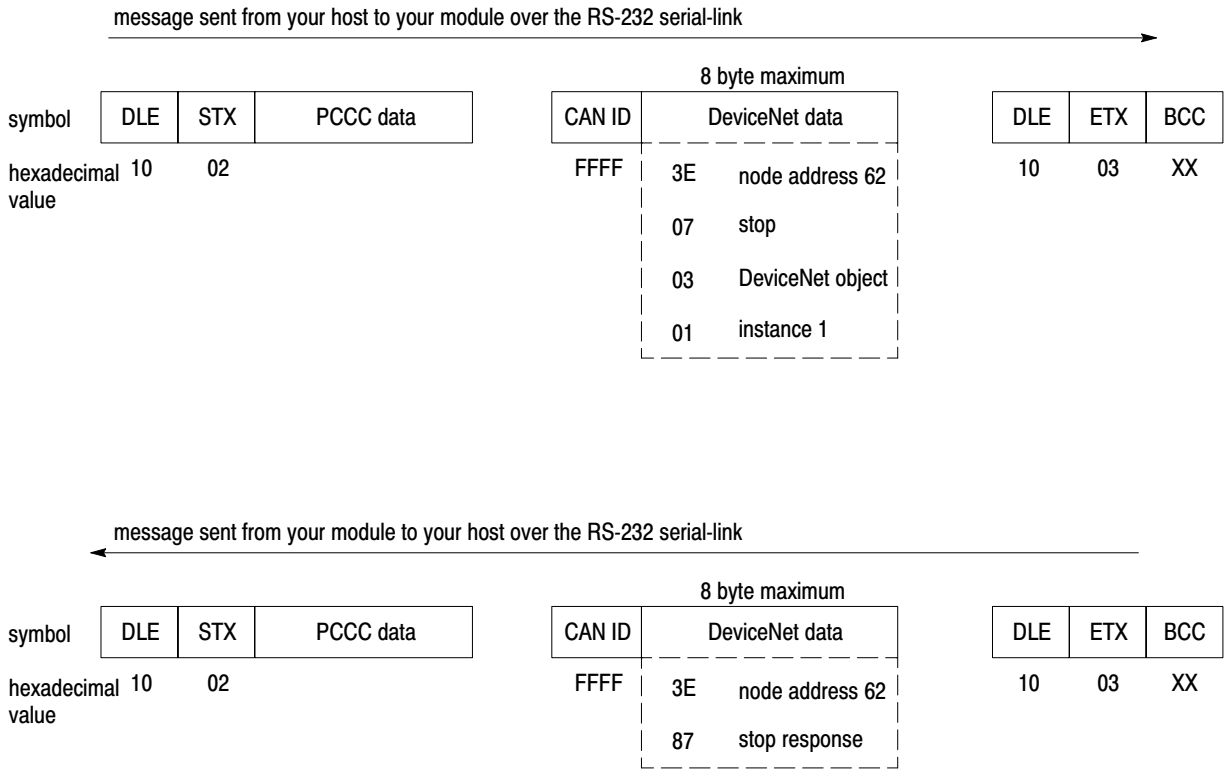
So that your host and module can communicate properly, the module must set its serial-link baud rate to match your host. To accomplish this, repeatedly send a response-retransmission request (10 05_{hex}) to the module. You will know the module has acquired the proper baud rate when it returns a message non-acknowledgment (10 15_{hex}) and its module-status indicator is solid green.



Important: After transmitting the 10 05_{hex}, your host needs to wait a reasonable amount of time for the module to respond before your host retransmits this message. In a "worst-case-scenario," this amount of time could range from 50 to 100 milliseconds.

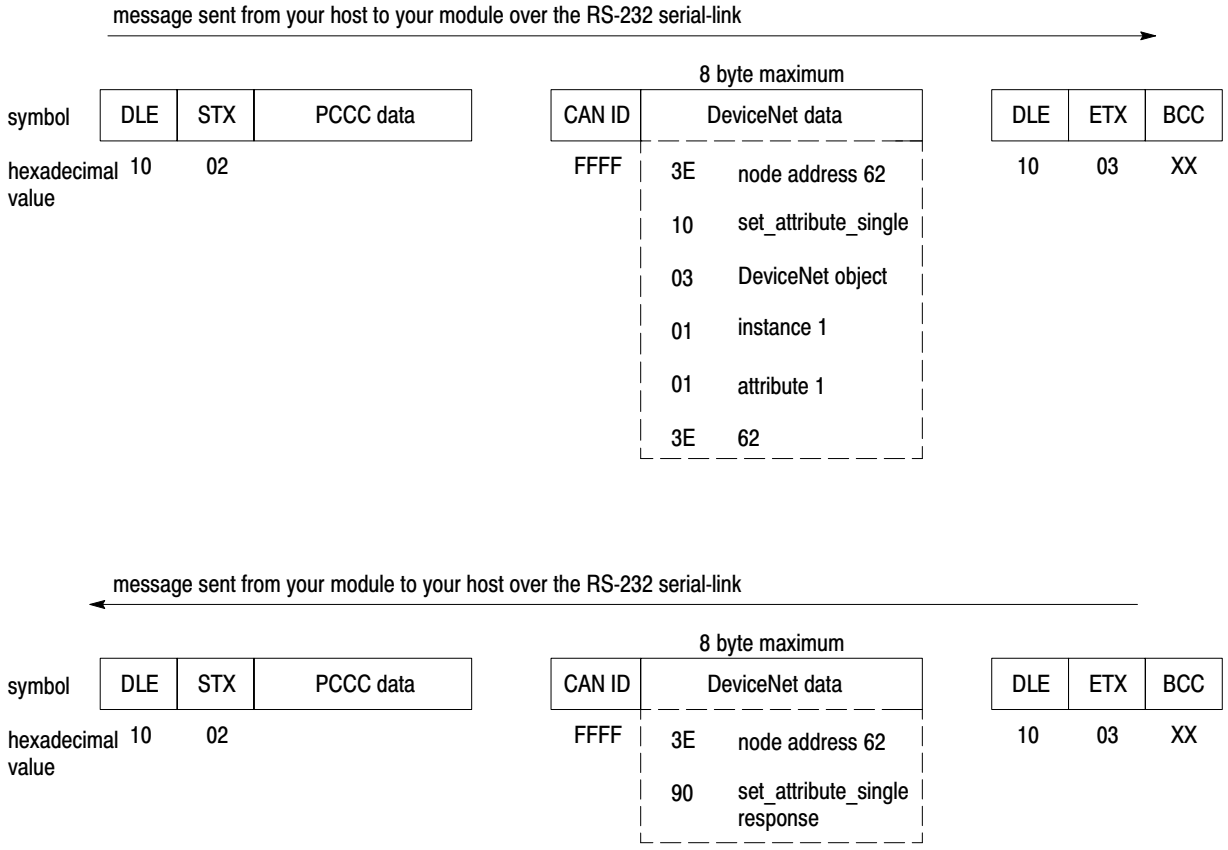
Stop Service

When you send a stop message to your module, the module ceases all network activities; it prepares the module for configuration. To stop your module, send a stop message to its DeviceNet object's instance 1.



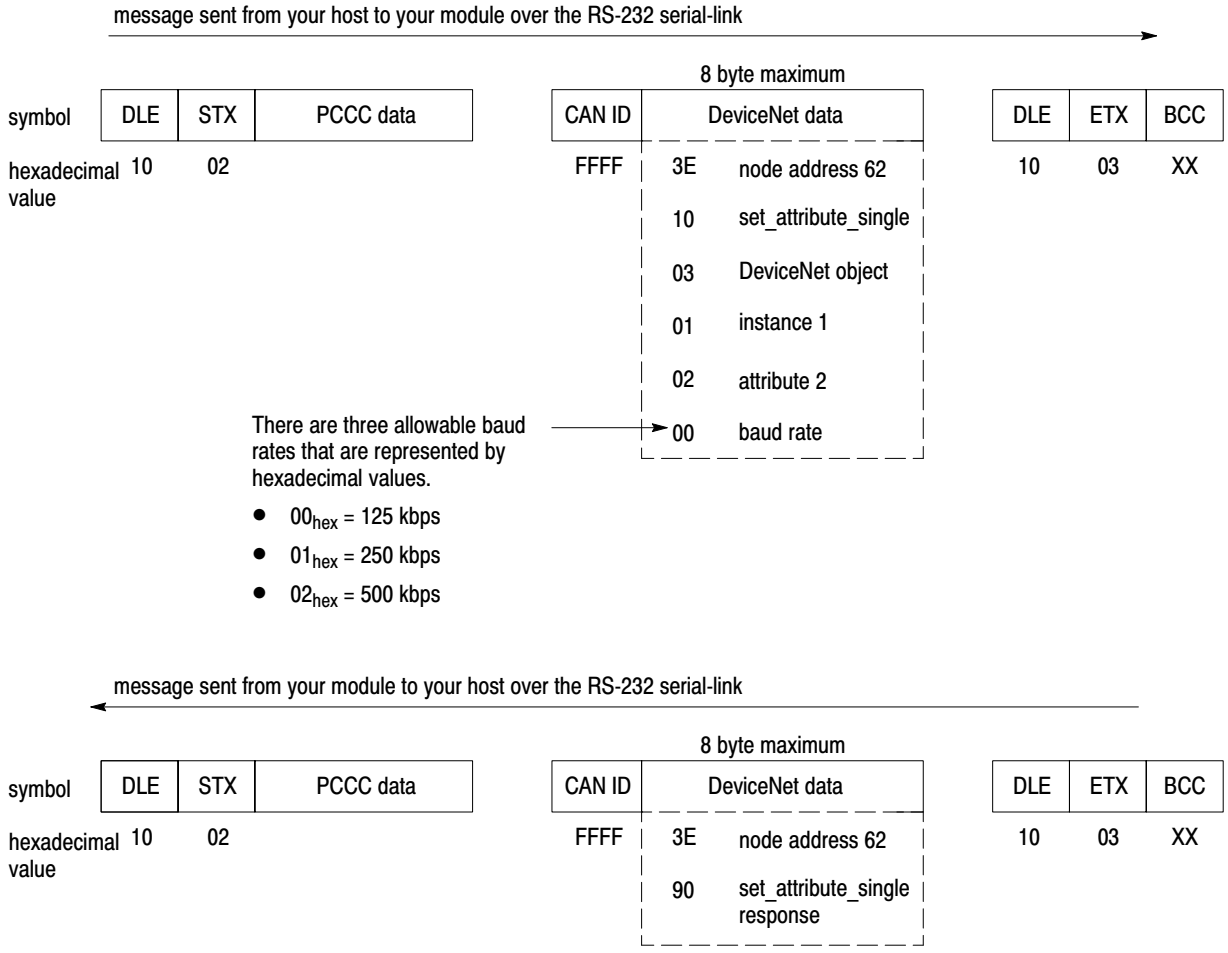
Set Node Address

Since the module is one part of the overall device (the host and module together constitute one device) you set the module's node address to match your host. To set your module's node address, send a *set_attribute_single* with the ID value to the DeviceNet object's instance 1, attribute 1.



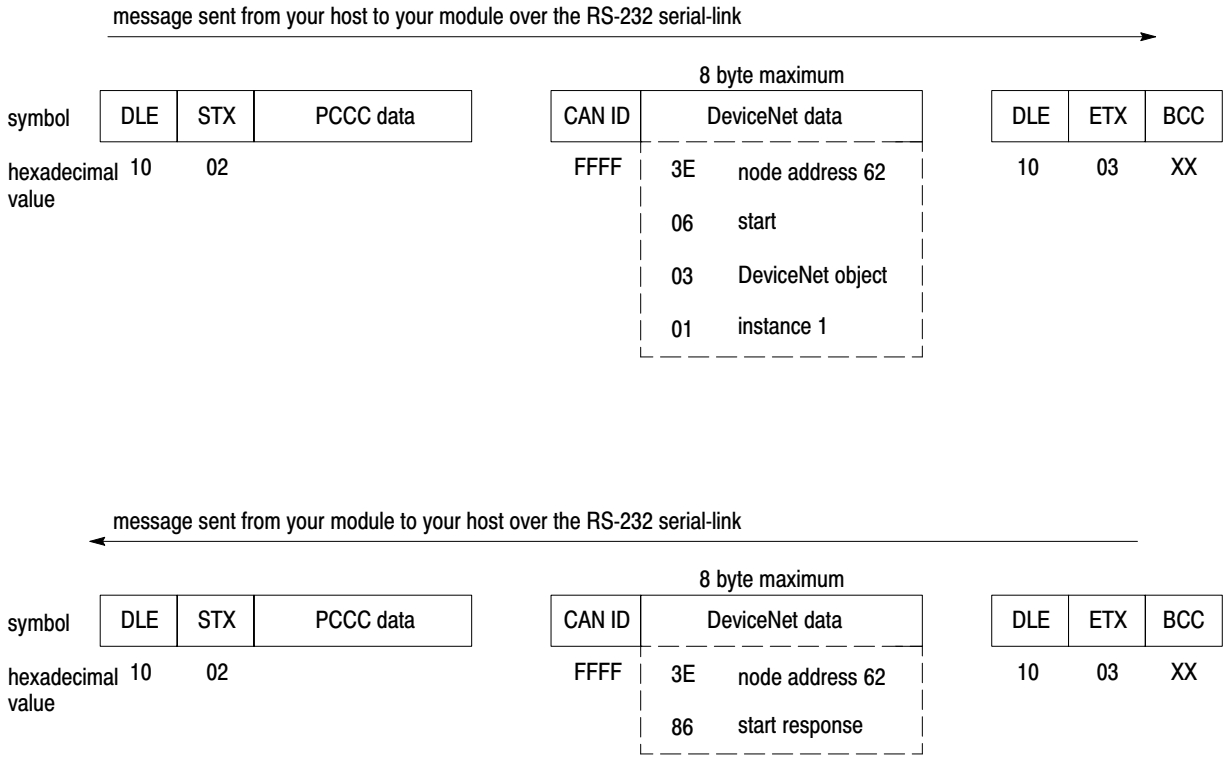
Set Baud Rate

You set the baud rate at which you would like to communicate on the DeviceNet network via the DeviceNet object. To set the baud rate at which your module will communicate on the network, send a *set_attribute_single* with the baud rate value to the DeviceNet object's instance 1, attribute 2.



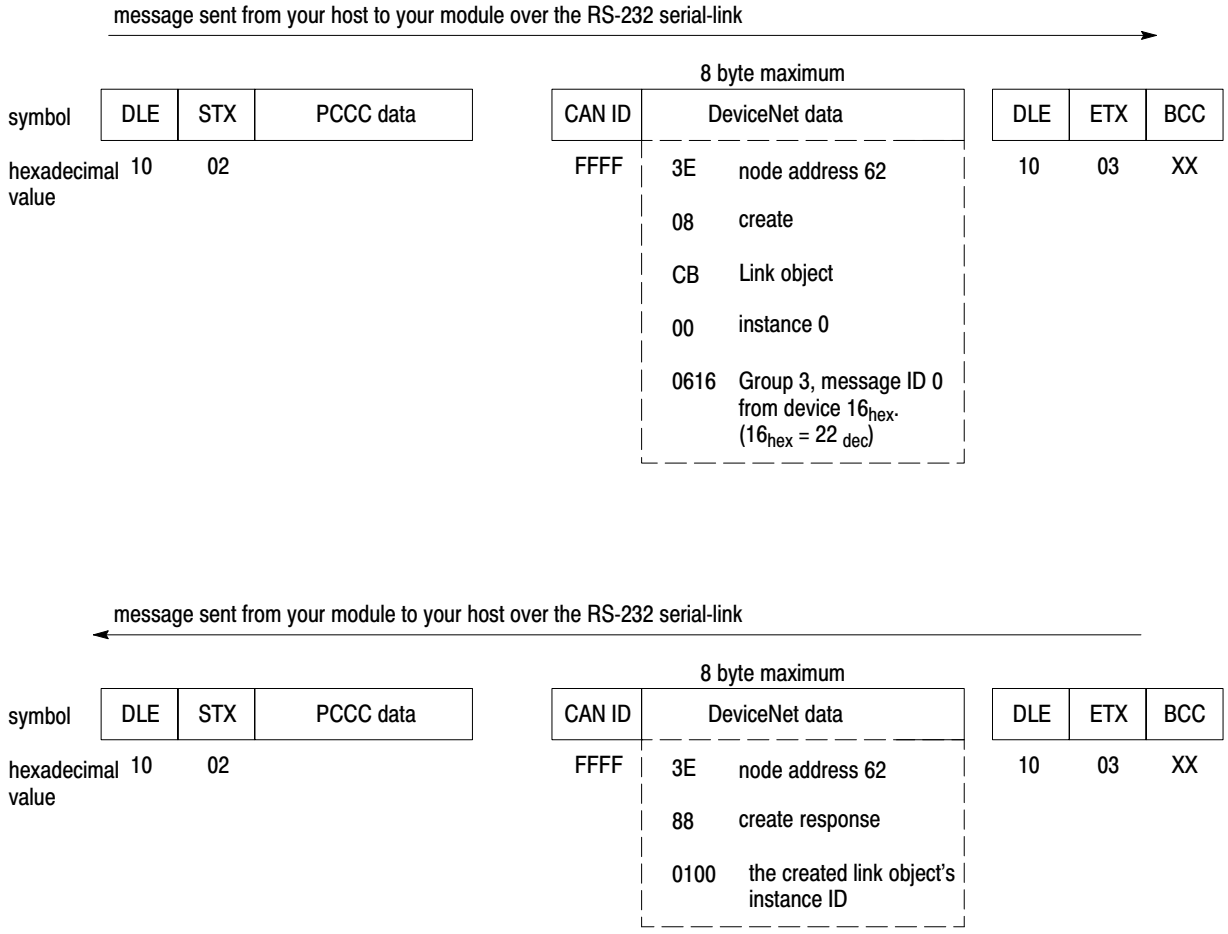
Start Service

The final step of initialization is to start the module. You start the module by sending a start service-code (06_{hex}) to its DeviceNet object's instance 1.



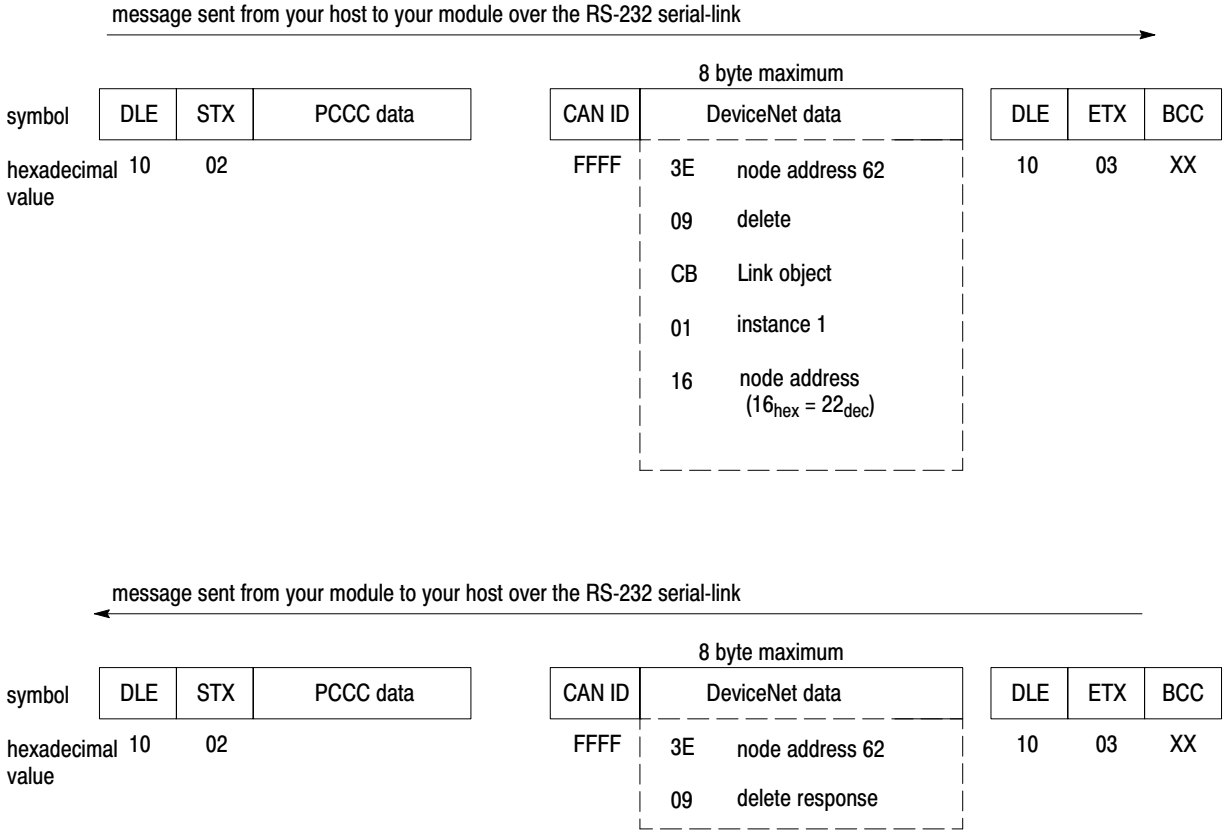
Creating Screeners

Screeners facilitate connected message transfers between your host and devices on the network. To communicate with a network device via connected messages, you must create a screener for the device within the Link object. This is accomplished by sending a create service-code to the Link class. In this example, we are creating a screener for a network device with the node address 22.



Deleting Screeners

You can delete screeners for devices with which you no longer need to communicate via connected messages. To delete a screener, send a delete service-code to the Link object. In this example, we are deleting the screener we created in the previous example.



Module Supported Objects

Appendix Contents

This appendix describes each module-supported object, covering only those attributes and services specific to the module.

Important: Except for the Link object, the module does not support class-level attributes or services of its objects.

Because this appendix is specific to the 1770-KFD interface-module, much detail on the objects described here is omitted. For more information about object definitions, refer to the DeviceNet Specification, Volumes I and II.

DeviceNet Object

Class ID Code: 03_{hex}

The DeviceNet object provides the configuration and status for a physical attachment to the DeviceNet network. Each physical network-attachment in a device (there can be more than one per device) has one and only one DeviceNet object. This is a required object for all DeviceNet devices.

Instance Services

$Code_{hex}$	Name	Description
0E	get_attribute_single	used to read a DeviceNet Object attribute value
10	set_attribute_single	used to modify a DeviceNet Object attribute value

Instance Attributes

ID	Access Rule	Name	Data Type	Description	Value
1	get/set	MAC ID	USINT	node address	range 0-63
2	get/set	baud rate	USINT	baud rate	range 0-2

MAC ID:

The host and the module together constitute one device. The MAC ID for this device is held in the module's DeviceNet object.

To modify the MAC ID, you must delete all connection objects and re-execute the Network Access State Machine. To accomplish this, you must:

1. Send a stop service.
2. Reconfigure.
3. Send a start service (restart).

Baud Rate:

The baud rate attribute indicates the selected rate as listed below:

Value	Meaning
00	125 kbps
01	250 kbps
02	500 kbps

To modify the baud rate you must delete all connection objects and re-execute the Network Access State Machine. To accomplish this, you must:

1. Send a stop service.
2. Reconfigure.
3. Send a start service (restart).

1770-KFD DF1 Object

Class ID Code: $C9_{hex}$

The host uses the module's DF1 object to track communication on the RS-232 serial-link. It logs all DF1 communication on this link and can be accessed by sending a *get_attribute_single*. Different bytes carry specific counters. These bytes are what the host reads to obtain the desired data, such as how many ACKs were received.

Instance Services

$Code_{hex}$	Name	Description
0E	get_attribute_single	retrieves information stored in a DF1 counter

Instance Attributes

ID	Access Rule	Name	Data Type	Description	Value
5	get	DF1 error counters	USINT[34]	tracks serial- link communication	see Table A.A

Table A.A
Module-Specific Counter Bytes

Counter byte	Counts
35, 36	number of times the node attempted to send a message
37, 38	messages that were successfully transmitted and ACKed
39, 40	ACKs that were received
42	NAKs received
45	ENQs sent
46	messages that could not be successfully sent
48, 49	messages received
50, 51	ACKs sent
52	NAKs sent
53	ENQs received
55	STXs received
57	messages that were aborted by receipt of DLE ENQ
58	messages that were aborted by the receipt of an unexpected control code other than DLE ENQ
60	times DLE NAK was sent because there was no buffer
67	times that DCD was lost

Identity Object

Class Code: 01_{hex}

The identity object provides general information about its device. This object *must* be present in all DeviceNet products.

Class Services

$Code_{hex}$	Name	Description
0E	get_attribute_single	returns the contents of the specified attribute
05	reset	invokes the Reset service for the device.

Instance Attributes

ID	Access Rule	Name	Data Type	Description
1	get	vendor	UINT	identification of each vendor by number
2	get	device type	UINT	indication of general type of product
3	get	product code	UINT	identification of a particular product of an individual vendor
4	get	revision	STRUCT of	revision of the item the identity object represents
		major revision	USINT	
		minor revision	USINT	
5	get	status	WORD	summary status of device
6	get	serial number	UDINT	serial number of device
7	get	product name	SHORT STRING	human readable identification
8	get	state	USINT	present state of the device

Vendor

If the returned attribute equals zero, it means “Unknown.”

Device Type

Every DeviceNet vendor uses the same list of device types to:

- register assembly-object instance definitions
- provide a scope for the product-code numbers

If the returned attribute equals zero, it means “Generic Device.” An explanation of the generic device must be supplied in the vendor documentation.

For a listing of the presently defined Device Types, refer to the DeviceNet Specification, Volume II.

Product Code

The product code identifies a product among a particular device type. Each vendor assigns this code to each of its products. The product code should map to a catalog/bulletin number.

The value zero is reserved to mean “Unassigned.” If the returned attribute equals zero, it means “Generic Vendor Product.”

Revision

The revision attribute, which consists of *major* and *minor* revisions, identifies the *revision* of the item this object represents.

If the returned attribute equals zero, it means “Unknown.”

The **major** and **minor** attributes, referred to above, are sometimes called a *series* and *revision*, respectively. The major-revision attribute is limited to seven bits. The eighth bit is reserved by the DeviceNet network and must have a default value of zero.

Status

This attribute represents the current status of the entire device. Its value changes as the state of the device changes.

Serial Number

This attribute is a number used in conjunction with the vendor number to form a unique identifier for each device on a DeviceNet. network

State

This attribute is an indication of the present state of the device.

Link Object

Class Code: CB_{hex}

You create CAN ID screeners in the link object. These screeners make it possible for the module to receive messages from selected network devices and pass that message on to your host. Screeners are similar to a list of passwords. When the module detects a message on the network, it checks its CAN ID and compares it to its screeners in the link object. If the message has a matching *password* (CAN ID) then it is accepted and passed on the host.

Class Services

$Code_{hex}$	Name	Description
08	create	configures ID screener
09	delete	deletes all screeners

Instance Services

$Code_{hex}$	Name	Description
09	delete	deletes a specified, single sreener

Instance Attributes

ID	Access Rule	Name	Data Type	Description	Value
1	get/set	screened ID	WORD	sets CAN ID to screen on	11 bit identifier

There can be a maximum of 128 screeners created.

Power Management Object

The power management object holds information about the module's network power supplying function. In addition, it facilitates terminating resistor configuration.

Class Code: CA_{hex}

Instance Services

$Code_{hex}$	Name	Description
0E	get_attribute_single	used to access data indicating status of the power settings
10	set_attribute_single	modifies the attribute responsible for the terminating resistor

Instance Attributes

ID	Access Rule	Name	Data type	Description
1	get	supply network power	USINT	indicates whether or not the module is supply network power
2	get	network power	USINT	indicates whether or not network power is present
3	get/set	terminating resistor	USINT	indicates if the terminating resistor is on. Toggles the resistor on or off

Values for Attribute 1	Definition
0	network
1	power supply
2	batteries

RS-232 Object

Class Code: $C8_{hex}$

The RS-232 object maintains the module's serial-link autobaud function before and after acquiring that rate. However, it only maintains the rate of data exchange between the host and module. Do not confuse this object's function with the DeviceNet object. The DeviceNet object acquires and maintains the DeviceNet network baud rate. In addition, you access the RS-232 object before initializing the module; you send a *reset* command to this object to clear the module before configuring it, which includes "serial-link autobauding."

Instance Services

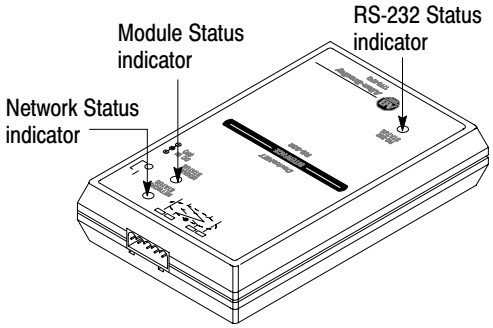
$Code_{hex}$	Name	Description
05	reset	clears the instance in preparation for the serial-link autobaud function

Troubleshooting

Appendix Contents

This appendix lists problems you may encounter when working with your 1770-KFD module on the link layer. It covers the most commonly encountered errors.

Status-indicator lights on the module can point to several types of errors. The lights may appear in a number of different states to indicate different problems or processes.



RS-232 Status Indicator	
condition	status
off	no activity, link OK
flickering green	activity, link OK
solid red	link failed (critical fault)
flashing red	link failed (non-critical fault)
Network Status Indicator	
condition	status
off	offline
flashing green	online
solid red	link failed (critical fault)
solid green	online, communicating
Module Status Indicator	
condition	status
off	no power
solid green	device OK
flashing green	not configured
solid red	critical fault
flashing red	non-critical fault

Troubleshooting the Module

The following table lists common errors encountered when communicating with your module on the link layer. The first column in each category describes a "symptom" that is then followed by a list of status-indicator states, probable causes, and recommendations on how to correct the error.

Important: Due to the wide range of hardware and software combinations used by developers, this troubleshooting section cannot be a comprehensive list of errors. However, we recommend that when you encounter an error, check here before calling technical support.

Error Category	Where To Look	Possible Cause	Recommendation
No communication occurring between your PC and the module.	RS-232 status indicator is flashing red.	An incorrect baud rate or parity error has been detected.	Be sure that your PC's baud rate is module supported (see page 1-2). Be sure there is no parity used and that the character size is 8 bits.
	Module status indicator is flashing green.	No baud rate has been detected.	Initiate autobaud.
	Module status indicator is solid red.	The module has an unrecoverable fault.	Replace your 1770-KFD module.
	the RS-232 cable	▪ Cable is not connected properly to the PC or module. ▪ Cable does not have the proper internal-wire cross-over.	▪ Be sure to securely attach the cable to the PC and module's respective RS-232 serial ports. ▪ Replace cable with an Allen-Bradley approved RS-232 serial cable.
	your software	Your software is not asserting DTR (data-terminal-ready).	Refer to your particular software application's user manual or programming instruction guide.

Error Category	Where To Look	Possible Cause	Recommendation
No communication occurring between the module and the DeviceNet network.	Network status indicator is off.	The module is offline because its DeviceNet cable may not be properly connected	▪ Activate your module on the DeviceNet network by sending the start service. ▪ Check the network and its devices to be sure they have been properly installed. ▪ Be sure the DeviceNet cable connecting the module to the network is properly installed, including the 5-position connector at the module's DeviceNet port.
	Network status indicator is flashing green.	The module has not been configured for connected messages (no screeners set up to receive connected DeviceNet messages).	Create screeners within the module's link object (see page 4-8).
	Network status indicator is solid red.	The module has detected an error that is prohibiting communication on the link. This could include detection of a duplicate node address or network configuration error.	Be sure that all of your devices are set to the correct baud rate. In addition, be sure that no two nodes are set to the same node address (MAC ID).
Error response received from start service (06_{hex}). Note that a good response is 86_{hex} while an error response is 94_{hex}.	94_{hex} response after you send start service.	The module is set at an incorrect node address (MAC ID).	Access the DeviceNet object to set the appropriate node address for the module (see page 4-5).
		The module is set at an incorrect network baud rate.	Access the DeviceNet object to set the appropriate baud rate for the module (see page 4-6).
	There is no response to the start service.	The module is not properly connected to the network via the DeviceNet cable.	Be sure the DeviceNet cable connecting the module to the network is properly installed, including the 5-position connector at the module's DeviceNet port.
Not receiving connected DeviceNet network messages (this excludes any unconnected messages).	Link object	The module has not been configured for connected messages (no screeners set up to receive connected DeviceNet messages).	Create screeners for each CAN ID from which you want to receive messages (see page 4-8)
Create link failed	The number of links created	There can only be a maximum of 128 links.	Delete unnecessary links.

Numbers

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V