User's Guide

DeviceNet™
Master Emulator

DN-MEM

Rev. 1.12

HURON
NET
WORKS

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

Pub # 2200002 Revision 1.4 1 July 1998

1. GETTING STARTED

1.1. General Description

The Master Emulator Software is a general purpose tool used in the development and debugging of DeviceNet nodes. It has the ability to monitor the DeviceNet bus for traffic. It displays incoming messages on the screen and optionally logs them to a file for later analysis. It provides a general purpose message construction feature which can be used to send specific messages to a DeviceNet node under development. Certain DeviceNet operations from the predefined Master/Slave Connection Set and the UCMM have their own keywords defined so that the exact format of common messages need not be committed to memory. Sequences of messages and commands may be assigned to any of the twelve function keys and manually executed when the function key is depressed. In order to run automated test suites there is a script file capability. The script file may contain a **replay** command (Sec 2.6) to allow continuous testing. The script file may also contain a **play**, or a **chain** command which allows multiple files to be treated as a single file.

1.2. Installation

Copy the contents of the diskette onto the hard drive in any convenient sub-directory. There are two executable files -- DMEM.EXE, and DMEM286.EXE. Use DMEM.EXE on a 386 or higher machine, use DMEM286.EXE on a 286 or higher machine. There are several other files included as example scripts (*.scr) and function key definitions (*.key).

The Master Emulator is design to be used with the DN-PC1, DeviceNet™ PC Interface Card. Refer to the board manual for proper installation of the hardware. The DIP Switches will select a board address and an interrupt level. The board may be set at address 0x200, 0x280, 0x300, or 0x380, and the IRQ must be set for one of the following interrupt levels 3, 4, 5, 7, 10, 11, 12, or 15. The software defaults are as follows:

Board Address	=	0x300
IRQ	=	IRQ5

The Master Emulator Software has three optional parameters which can be specified on the command line. These are board address, interrupt level, and baudrate. They may be specified in any order since only the values are significant. If a parameter is not specified on the command line it takes a default value. The default for baudrate is 0 which causes the CAN chip initialization to be skipped. Without a valid baudrate specification, the initialization of the CAN chip could disrupt a network on which there was traffic present.

Baudrate can be conveniently specified as a command line parameter, or from the keyboard via the **baudrate** command (see section 2.15), or in the **autoexec.scr** script file (see section 2.5). Valid baudrates are 125, 250, and 500.

For example if the DN-PC1 is located at address 0x280 and uses interrupt level 10 and the baudrate is to be 250 Kbaud, then enter the following:

```
C:\DMEM>DMEM   280   10   250    or
C:\DMEM>DMEM   250   10   280    or
C:\DMEM>DMEM   10   280   250
```

Master Emulator does not care about the order of the three arguments: only the values.

After being loaded, Master Emulator will look for a file named **autoexec.scr**. If the file exists, the file will automatically be played (See Section 2.5).

To exit the Master Emulator use **^X**. Hold the control key (Ctrl) and press the letter X.

1.3. Screen Layout

The screen is divided into four colored sections called Banner, Traffic, Command, and Help. The Banner Section is two lines of red text on a light gray background. The software version and compilation date are shown here. The Help Section at the bottom of the screen is a single line of red text on a light gray background. The Help Section displays the board address, the interrupt level, the baudrate, the state of the DeviceNet bus power supply, and a prompt indicating that **Control-x** will exit the program.

The remainder of the screen is divided into two halves. The top half of the screen, with the black text on a cyan background, is the Traffic Section which monitors incoming traffic. The bottom half of the screen, with the yellow text on a blue background, is the Command Section, which is used for entering commands and outgoing messages. Actual messages, sent on the DeviceNet wire, are shown in blue text on a green background. This color scheme is also used to display a directory of filenames. (See the **load, play, and chain** commands in Sections 2.2, 2.5 and 2.23).

In the traffic section all of the received messages are shown in hexadecimal form with the CAN identifier enclosed in square brackets -- [ ] -- and the CAN data field enclosed in angle brackets -- < > --. Where possible a textual description of the message is given on the same line.

1.4. Program Input

Input to the program comes from one of three sources. These sources are:

- Keyboard
- Function Key Buffer
- Script File

The input from any of the three sources consists of either messages, commands, or comments. Commands consist of a keyword and an optional list of parameters separated by one or more spaces. A comment is any line beginning with a pound sign(#), a colon(:), a space (" ") or a newline(\n). DeviceNet messages consisting of an identifier and a message body are represented with the following syntax:

[identifier]<xx xx xx>

The **eleven (11) bit CAN identifier**, in hexadecimal notation, is enclosed in square brackets. The message body consists of one or more bytes, in hexadecimal notation, enclosed in angle brackets. Spaces used between message bytes to improve readability are optional. For example a message with the identifier 0x5E4 and two data bytes (0x55 and 0xAA) would be written as follows:

[5E4]<55 AA>

If more than eight bytes are specified in the message body then the message will be sent out as an explicit fragmented message. In this case the first and second bytes of the message should conform to the DeviceNet fragmentation protocol. The first byte should have the fragmentation bit set and the second byte should be 0 to indicate the first fragment. In all other cases when using this form of message construction, there is no error checking or other protocol related activity on the part of the Master Emulator.

1.5. Explicit Message Types

An explicit message can be constructed in two different ways. The first method uses the Predefined Master/Slave Connection Set. These explicit messages have an identifier in Group 2, message Id four(4), and destination Mac Id. The message header in the data field contains the value of the Master Mac Id. the second method uses the UCMM to allocate an identifier. This identifier will have the Master Mac Id in it. The Slave Mac Id is placed in the message header byte in the data field.

1.6. Message Body Format

Message Body Format is related to the sizes of Class Identifier and Instance Identifier, used in the construction of explicit messages. The DeviceNet specification defines four choices for how to handle this value. See Vol. I, Chapter 4, page 4-7 of the DeviceNet Specification.

2. COMMANDS

Chapter Preview

2.1	*help*	*2.12*	*release*	*2.23*	*chain*
2.2	*load*	*2.13*	*slaveid*	*2.24*	*mask*
2.3	*save*	*2.14*	*masterid*	*2.25*	*match*
2.4	*log*	*2.15*	*baudrate*	*2.26*	*reset*
2.5	*play*	*2.16*	*fragackid*	*2.27*	*cosack*
2.6	*replay*	*2.17*	*fragokid*	*2.28*	*open*
2.7	*delay*	*2.18*	*message*	*2.29*	*close*
2.8	*allocate*	*2.19*	*class*	*2.30*	*bodyformat*
2.9	*setepr*	*2.20*	*instance*	*2.31*	*ucmm*
2.10	*poll*	*2.21*	*get*		
2.11	*strobe*	*2.22*	*set*		

2.1. help

The **help** command causes a list of commands to be displayed. The list is displayed one page at a time. Depressing any key causes the next page to appear.

2.2. load

The **load** command, a space and a filename causes a file of previously saved Function Key Definitions, to be loaded into the function key buffers. For example to load key definitions from the file **"startup.key"** enter the following:

load startup.key Then press Enter.

If the filename parameter to the **load** command does not have a filename extension then the extension ".key" will be appended to the name. Master Emulator will try to open the file, load the key buffers and print a message saying how many key buffers it loaded. If the file cannot be found a message will be displayed.

To obtain a list of function key definition files, type load, without a filename, then, press enter. The display will change to a green background with blue text. The text will consist of a list of *.key files from the directory in which Master Emulator is currently being run. Press any key to return to the command screen.

2.3. save

The **save** command is the opposite of the load command. It saves the current definitions for the Function Keys into a file. For example to save the Function Key Definitions to a file called **"newfuncs.key"** type the following:

save newfuncs.key Then press Enter.

As with the **load** command, if no extension is provided for the filename parameter then an extension of ".key" will be appended. Master Emulator will open the file and save all the Function Key Definitions and display a message saying how many definitions it saved. Note that the definition files are saved in ASCII text format and may be viewed and manipulated with standard word processors and editors. The syntax is trivial and will not be further explained in this manual.

2.4. log

The **log** command has two optional parameters. The first is a filename and the second is an access control. A filename without an extension will have the extension ".log" appended. If no file name is specified then logging is turned off. If a filename is specified then any active logfile is closed and a new one is opened. The access control can be either the letter 'a' or the letter 'w'. If the specified filename exists, then the 'a' or 'w' tells Master Emulator whether to append ('a') new information to the end of the existing file, or to overwrite ('w') the old data. If the access control parameter is not specified then 'a' for append is the default.

In operation, the logfile records all commands going out and all traffic coming in. For example to start logging to a file called test1.log use the following command:

log test1.log Then press Enter.

Typing **log** again without the filename will cause test1.log to be closed, and a message printed saying logging has been turned off.

In the logfile all incoming packets begin with a colon(:) which is interpreted as a comment. This allows a logfile to be "played"(See Section 2.5)

2.5. play

The **play** command takes a filename as a parameter. If the filename has no extension then the extension ".scr" is appended. It runs the file as a script, exactly as if it had been entered from the keyboard. In DMEM version 1.09 and greater, the functionality of this

command has been extended. The original functionality of this command is retained in the **chain** command (section 2.23). Up to eleven open "play" files will be maintained on a stack. If a play file contains a **play** command then input will come from the new file and will return to the original file when end of file is reached. For example to play a file called **"count.scr"**, type the following command:

play count.scr Then press Enter.

The file **"count.scr"** will be opened, read and interpreted exactly as if the characters had been entered from the keyboard. When the end of file is reached input will return to the keyboard. If a file is to be played continuously then use a **replay** command (section 2.6) with no parameter as the last line of the file.

To obtain a list of script files, use a **play** command without a filename. Then, press enter. The display will change to a green background with blue text. The text will consist of a list of *.scr files from the directory in which Master Emulator is currently being run. Press any key to return to the command screen.

When input is being taken from a script file, the dollar sign($) character may be used as a temporary keyboard escape. All keyboard input up to the next return will go in place of the dollar sign in the currently executing script file. This may be used for example to program unique vendor serial numbers into a DeviceNet node. A canned sequence of steps may be placed in a script file to "unlock" the change serial number mechanism. At a strategic point the dollar sign($) could be placed in the script file, and the serial number could be entered. See section 4.4 for an example.

2.6. replay

The **replay** command takes an optional decimal parameter which specifies the number of times the file is to be read from the beginning. If the parameter is missing it causes the file that is being read to loop continuously until the process is aborted by pressing any key. The **replay** command must be placed at the end of the actual test script itself. Here is a short example test script using the replay command:

```
poll  01  00
delay 100
poll  02  00
delay 50
strobe  0 1 2 3 4 5 6 7
replay
```

In order for this script to produce a response, the three predefined connections on one of the slave devices on the network must have been allocated. See section 2.8.

If the example above should be performed ten times then the file would appear as follows:

```
poll  01  00
delay 100
poll  02  00
delay 50
strobe  0 1 2 3 4 5 6 7
replay  10
```

2.7. delay

The **delay** command adds a delay in milliseconds when inserted into a sequence of commands. For example to insert a delay of 150 milliseconds type the following:

delay 150 Then press Enter.

See the example program in Section 2.6.

Small delays on the order of several milliseconds may not be timed with extreme accuracy, due to the processing delay of the script file interpretation.

2.8. allocate

The **allocate** command is used to establish one or more of the connections in the Predefined Master/Slave Connection Set (DeviceNet Specification Vol. I, Rev 1.3, Chapter 7). The single parameter for the **allocate** command is a hexadecimal encoded bit mask which is, described in the DeviceNet Specification Vol. I, Rel 2.0, p5-57, and constructed as follows:

7	6	5	4	3	2	1	0
*	NACK	CYC	COS	*	STB	POLL	EM

NACK	Acknowledge Suppression
CYC	Cyclic Connection
COS	Change of State Connection
STB	Bit Strobe Connection
POLL	Poll Connection
EM	Explicit Message Connection
*	Reserved - should always be set to zero (0).

The available connections are an Explicit Messaging Connection, a Poll Connection, a Bit-Strobe Connection, and a Change of State/Cyclic Connection. To establish the connections between the Master and the Slave, set the Slave's MAC ID with the **slaveid** command to that of the desired Slave device. For example to open the various connections with device number 62 use the following sequence of commands:

slaveid 62	Then press Enter
allocate 1	To open an Explicit Messaging Connection
	OR
allocate 3	To open an Explicit Messaging and Poll Connection
	OR
allocate 7	To open an Explicit Messaging, a Poll, and, a Strobe Connection

*** Note:**
It is not possible to allocate just the I/O Connections without the Explicit Messaging Connection. It is possible to allocate all connections and then release the Explicit Messaging Connection.

2.9. setepr

The **setepr** command sets the Expected Packet Rate (EPR) attribute (attribute # 9) of a connection instance(DeviceNet Specification Vol. I, Rev 1.3, page 5-8). This command takes two decimal arguments. The first is the connection instance and the second is the EPR value. The units of the EPR value are milliseconds. For the Predefined Master/Slave connection set, the connection identifiers are as follows:

Connection Instance Number	Connection Type
1	Explicit Messaging
2	Poll Connection
3	Bit-Strobe Connection
4	COS/Cyclic Connection

To set the EPR of the explicit messaging connection to 0 use the following:

setepr 1 0 Then press Enter.

To set the EPR of the poll connection to 100(milliseconds), use the following:

setepr 2 100 Then press Enter.

Setting the EPR of other Connection Instances is possible depending on Slave's implementation.

2.10. poll

The **poll** command is used to send packets of data to the slave device using the Master's Poll command [Group2, Destination Mac Id, MsgId 5]. The Poll command may send up to

eight bytes non-fragmented to the destination slave device. For example to send the data bytes 0x55 and 0xE7 to a slave device type the following command:

poll 55 E7 Then press Enter.

See Section 2.6 for an example program.

In the present version of DMEM there is no support for fragmented IO messages. This is due to the requirement that produced and consumed connection sizes are required on both ends of an I/O connection to determine whether or not a message is being fragmented. Since DMEM does not implement a full connection manager, which would interfere with its use as a development tool, fragmented I/O messages are not supported. The individual fragments can be created using the square bracket/angle bracket notation and placed in a Function Key Buffer or a script file. Note that fragmented Explicit Messages are supported for both requests and responses.

2.11. strobe

The **strobe** command [Group2, Source Mac Id, MsgId 0] sends one bit of output data to each allocated slave, using the Master's Bit Strobe Command Message. The message length would be eight bytes and each allocated slave on the network is assigned one bit out of the sixty-four, corresponding to its MAC address, to be used as it wishes. Each allocated slave receiving a bit strobe from its master will produce a bit strobe response. Master Emulator also supports the use of the zero length strobe. In this case, just the command with no data is required.

Example #1

Send a strobe bit of 1 to device 60 and zero to the rest.

strobe 0 0 0 0 0 0 0 10 Then press Enter.

To send a zero length strobe to the slaves allocated to the Master Emulator, type the following:

strobe Then press Enter.

2.12. release

The **release** command informs the Slave that it is no longer under the Master's control. The parameter to the release is a hexadecimal encoded bit mask, which follows the same format as for the **allocate**.

7	6	5	4	3	2	1	0
*	*	CYC	COS	*	STB	POLL	EM

CYC	Cyclic Connection
COS	Change of State Connection
STB	Bit Strobe Connection
POLL	Poll Connection
EM	Explicit Message Connection
*	Reserved - should always be set to zero (0).

To release a connection, type:

release 1	To release the Explicit Messaging Connection
release 2	To release the Poll Connection
release 4	To release the Bit Strobe Connection
release 10	To release the COS connection
release 20	To release the Cyclic Connection
release 37	To release all Connections

2.13. slaveid

The **slaveid** command takes a decimal number in the range [0..63] and establishes an internal value to be used by the commands **allocate, setepr, release, poll, strobe, get, set, cosack, open, and close**. For example to set the Slave MAC ID to 57 type the following:

slaveid 57 Then press Enter.

Once set the slaveid remains in effect until it is changed. The default value is sixty-three (63).

2.14. masterid

The **masterid** command takes a decimal number in the range [0..63] and establishes an internal value to use in the commands **allocate, setepr, release, poll, strobe, get, set, cosack, open, and close** for the master's MAC ID. For example to set the master's MAC ID to 1 use the following command:

masterid 1 Then press Enter.
Once set the masterid remains in effect until it is changed. The default value is one (1).

2.15. baudrate

The **baudrate** command sets the baudrate to one of the three values allowed by DeviceNet. To change the baudrate type the following:

baudrate 125	to set the baudrate to 125 Kbaud.
	OR
baudrate 250	to set the baudrate to 250 Kbaud.
	OR
baudrate 500	to set the baudrate to 500 Kbaud.

If the baudrate is changed by this command then the value displayed on the Help line will also change. On a network which is running this change may upset any nodes which do not track this change including forcing them or the DN-PC1 card to a Bus Off condition.

2.16. fragackid

The **fragackid** command is used to tell Master Emulator what identifier to use when acknowledging a fragmented response. This response would be from an explicit message request. Failure to specify this parameter before receiving a fragmented response will probably cause the slave device to hang up. For example to tell Master Emulator to use identifier 0x601 to acknowledge fragmented responses, type the following:

fragackid 601

The default value of fragackid is derived from the value of **slaveid** (60 for example) and is computed as follows:

#define	GROUP2	(2<<9)
#define	EXP_REQ	4
#define	EXP_RSP	3

fragackid = GROUP2 + (**slaveid** <<3) + EXP_REQ ;
= (0x400) + (0x1E0) + (0x004) ;
= 0x5E4 ;

That is, **fragackid** is a group 2 Explicit Request with the slave MAC ID shifted left three places.

2.17. fragokid

The **fragokid** command is used to tell Master Emulator which message identifiers to check for the fragmentation bit being set. Failure to set this parameter will result in erroneous behavior when fragmented messages come in. To set the fragokid to 0x606, type the following:

fragokid 606

The default value of **fragokid** is derived from the value of **slaveid** (60 for example) and is computed as follows:

```
#define     GROUP2      (2<<9)
#define     EXP_REQ     4
#define     EXP_RSP     3

fragokid    = GROUP2 + (slaveid <<3) + EXP_RSP
            = (0x400) + (0x1E0) + (0x003) ;
            = 0x5E3 ;
```

That is fragokid is a group 2 Explicit Response with the slave MAC ID shifted left three places.

2.18. message

The **message** command provides an easy way to become familiar with the format of an explicit message because Master Emulator prompts for each byte of a non-fragmented Explicit Message. To invoke the message command just type message, then press Enter.

Prompts for each field are displayed. Master Emulator then waits for a hexadecimal value to be entered for each field. If there is no more data for the message the process is terminated by pressing Enter.

2.19. class

The **class** command followed by a hexadecimal parameter is used to select a DeviceNet Class for a subsequent invocation of **get** (attribute single), **set** (attribute single), or **reset**. Once a value is assigned to **class** it retains that value until the next **class** command is encountered. The value defaults to 1 (Identity Class) when DMEM is invoked. A message is displayed confirming the current value of **class**. If no parameter is given then the present value is displayed. The **class** parameter is entered and displayed in hexadecimal to be consistent with the DeviceNet specification. Instance and attribute numbers are documented in the DeviceNet specification in decimal.

2.20. instance

The **instance** command followed by a decimal parameter is used to select an instance of a **class** for a subsequent **get, set, or reset** command. Once **instance** is assigned a value it retains that value until the next **instance** command. The value defaults to 1 (First Instance) of a **class** when DMEM is invoked. A message is displayed confirming the current value of **instance**. If no parameter is given then the present value is displayed. The **instance** parameter is entered and displayed in decimal to be consistent with the usage in the DeviceNet specification. Instance may be set to 0 to refer to the separate list of attributes belonging to the class as a whole.

2.21. get

The **get** command followed by a decimal parameter is used to send a get attribute single service to the slave device over the explicit messaging connection. This command uses the information provided by **slaveid, masterid, class, instance, bodyformat** and the parameter to the **get** command to construct the message. As an example suppose we want to get the vendor identification from a device. In Volume II p.6-4 of the DeviceNet specification we see that Vendor is attribute 1 of instance 1 of the Identity Class. Let us suppose that we have as MACID #1 (**masterid 1**) established a connection to MACID #63 (**slaveid 63)** and we enter the command:

get 1

The transmitted message will be constructed as follows:

```
#define     GROUP2                  2
#define     EXP_REQ                 4
#define     GET_ATTRIBUTE_SINGLE    14
```

- Identifier = (GROUP2<<9) + ((**slaveid)** <<3)+ EXP_REQ ;
- = (0x400) + (0x1F8) + (0x004) ;
- = 0x5FC ;
- DATA[0] = **masterid** ; // Frag = 0, and Xid = 0
- = 0x01 ;
- DATA[1] = GET_ATTRIBUTE_SINGLE ;
- = 0x0E ;
- DATA[2] = **class** ; //default = 1
- = 0x01 ;
- DATA[3] = **instance** ; //default = 1
- = 0x01 ;
- DATA[4] = attribute // parameter of **get** command

- = 0x01

Using the syntax for a general purpose message, the above lines would cause the following message to be sent:

[5FC] <01 0E 01 01 01>

The response from the slave would show up on the receive screen and look something like:
[5FB] <01 8E 14 00>

This would identify the vendor of the slave device as Huron Net Works!

2.22. set

The **set** command is the opposite of the **get**. The first parameter is in decimal and is the attribute number of the **class** and **instance** that is to be set. After the attribute number comes one or more bytes of attribute value. While maybe not the most convenient for word or longer attributes it is simple and general. Note that word parameters are specified in little-endian format with the low order byte coming first. If we do a **set 1** without changing anything from the above example we should get an error message since Vendor is not a settable attribute of the Identity Class. The coding would be the same as the example above except the GET_ATTRIBUTE_SINGLE would be replaced by a SET_ATTRIBUTE_SINGLE (0x10) and two data bytes would be added as DATA[5] and DATA[6].

2.23. chain

The **chain** command takes a filename as a parameter. This command preserves the functionality of the **play** command (section 2.5) from DMEM Version 1.08 and earlier. Any file being read for commands is closed. The file specified in the parameter is opened and input continues from the beginning of the new file. See Section 2.5 on play for the new description of its functionality.

2.24. mask

The **mask** command takes a hexadecimal argument as a parameter. In conjunction with the **match** command the hardware screener in the 82C200 chip on the **DN-PC1** can be programmed to accept certain messages and ignore other traffic on the network. In constructing a mask place a zero(0) in each position of the eleven bit CAN Identifier field that is significant for determining whether a message is to be accepted. Place a one (1) in each position which is a "don't care". Bit 10 (MSB) of the CAN Identifier field corresponds to bit 10 of the mask, while bit 0 (LSB) of the CAN Identifier field corresponds to bit 0 of the mask. This command will **not** take effect until the next

occurrence of the **baudrate** command. If the same baudrate which is in effect is used then the new values of **mask** and **match** will take effect and the baudrate will remain unchanged.

The hardware screener in the 82C200 CAN Controller chip works on the most significant eight bits of the eleven bit CAN Identifier field. Arguments to the **mask** and **match** commands assume that all eleven bits of the CAN Identifier field participate in the process. **DMEM** will adjust these values before writing them to the CAN Controller.

See the example in the following section.

2.25. match

The **match** command takes a hexadecimal argument as a parameter. In conjunction with the **mask** command the hardware screener in the 82C200 chip on the **DN-PC1** can be programmed to accept certain messages and ignore other traffic on the network. In constructing the match place a zero(0) in each position of the eleven bit CAN Identifier field that is to be a zero (0). Place a one (1) in each position which is to be a one (1).
Bit 10 (MSB) of the CAN Identifier field corresponds to bit 10 of the match, while bit 0 (LSB) of the CAN Identifier field corresponds to bit 0 of the match. This command will **not** take effect until the next occurrence of the **baudrate** command. If the same baudrate which is in effect is used then the new values of **mask** and **match** will take effect and the baudrate will remain unchanged.

The hardware screener in the 82C200 CAN Controller chip works on the most significant eight bits of the eleven bit CAN Identifier field. Arguments to the **mask** and **match** commands assume that all eleven bits of the CAN Identifier field participate in the process. **DMEM** will adjust these values before writing them to the CAN Controller.

Mask & Match Example #1

Suppose that we want to monitor only the **poll** responses from a particular slave node (Node 60). We construct the mask condition as follows:

BIT	10	=	0	Group 1 Message
BITS	9-6	=	1111	Poll Response Message ID
BITS	5-0	=	111100	MAC ID 60

mask 0
match 3FC
baudrate 125

Since only the most significant eight bits take part in the hardware screening process, all poll responses from nodes 56 to 63 will be received.

Mask & Match Example #2

Suppose we want to monitor all Group 3 messages. the mask and match commands would be as follows:

mask 1FF
match 600
baudrate 125

Only bits 10 and 9 of the CAN Identifier field are significant, and they must both be one.

2.26. reset

The **reset** command uses the internal variables **class** and **instance** to build an explicit message whose service code is a reset service (0x05). As with other explicit messages the internal variables **slaveid**, **masterid, and bodyformat** are used to construct the identifier, the message header, and the message body. Not all objects support a reset service, so error responses are to be expected in many cases. Two DeviceNet objects which typically support the reset service are the identity object and the connection object.

2.27. cosack

The **cosack** command will take up to eight hexadecimal parameters and build a Change of State/Cyclic Acknowledge Message. This is a group 2 message with destination (**slaveid**) MAC address and message ID two(2). It operates the same as a **poll** except for the change of message ID. It is hard to imagine how this might be used in a script file, but it is included for completeness.

Example

Send four(4) bytes of data as a Change of State Acknowledge

cosack 14 00 19 26 Then press Enter

2.28. open

The **open** command takes three decimal parameters, and constructs a UCMM Open Explicit Request Message. This message is a group 3 message, with a message ID of six(6), and the Source MAC ID (**masterid)** The three parameters are the requested body format, the group select, and the source message id in that order. No error checking is done on the parameters so many possible combinations will result in error responses or other unexpected conditions. Valid ranges for each of the three parameters is [0..15]. The data field of the message consists of the message header, the service code (0x4B), and the three parameters packed into two bytes. See DeviceNet Specification Vol. I, Rel 2.0, pp. 4-7 to 4-10.

Example

Construct a UCMM Open, asking for DeviceNet 8/8, on Group 3 with Message ID two(2).

open 0 3 2 Then press Enter

The resulting message might look like

[781]<3F 4B 00 32>

Example

Construct a UCMM Open asking for DeviceNet 16/16, on Group 1 with Message ID 10

open 2 1 10 Then press Enter

The resulting message might look like

[781]<7F 4B 02 1A>

The **open** command sets the value of an internal variable called **ucmm** to the value one(1). It can be toggled between one(1) and zero(0) with the ucmm command.

2.29. close

The **close** command takes a decimal parameter which is the instance number of the connection to close. It constructs a UCMM Close Connection Request which is a group 3 message, with message ID six(6) and source MAC ID (**masterid**). The data field consists of the message header, the service code (0x4C) and the connection instance number. See the DeviceNet Specification Vol. I, Rel 2.0, pp. 4-17 to 4-19.

Example

Close connection instance #3 on the slave device

close 3 Then press Enter

The resulting message might look like

[781]<3F 4C 03 00>

The internal variable **ucmm** is set to zero

2.30. bodyformat

The **bodyformat** command takes a decimal parameter which may be a in the range zero(0) to three(3). The value is saved in an internal variable; it is then used to construct Explicit Request Messages in any of the acceptable formats. The following table shows the

correspondence between the values of **bodyformat** and the format of a corresponding Explicit Request.

Value	Meaning
0	DeviceNet(8/8) Class= 8 bits, Instance= 8 bits
1	DeviceNet(8/16) Class= 8 bits, Instance = 16 bits
2	DeviceNet(16/16) Class= 16 bits, Instance = 16 bits
3	DeviceNet(16/8) Class= 16 bits, Instance = 8 bits

2.31. ucmm

The **ucmm** command toggles an internal variable which selects either the Group 2 predefined Explicit Request or the Explicit Request created by the UCMM. After both a UCMM connection (see **open**) and the predefined Explicit (see **allocate**) have been created, this toggle will allow a choice of which method to use. If using the UCMM Explicit Connection make sure to set the **fragackok**, and **fragidok** variables.

3. FUNCTION KEYS

3.1. Use of Function Keys

Function Keys are used to store common sequences of commands and messages in a single memory buffer. Then by pressing a single key the entire sequence may be executed. For example, instead of typing:

```
allocate 7
setepr 1 0          This may all be stored in
setepr 2 0          the Function Key F1.
setepr 3 0
```

If, during debugging, the slave device needs to be reset, then connections may be reestablished by pressing just the F1 key. Function Keys F1 through F12 may be used, and may be edited, saved, and recalled.

3.2. Defining a Function Key

If a Function Key has no definition, then the first time it is depressed an editing screen will appear and a definition may be entered. Once a key is defined subsequent depressions will cause the definition to be replayed as if the characters had been entered from the keyboard again. After a function key is defined it may be edited by holding down the shift key and depressing the function key. Depressing Alt and a Function Key will cause the buffer associated with that function key to be cleared. Depressing Ctrl and a Function Key performs no function, however an acknowledgment that the key was seen is displayed.

To enter commands into a Function Key, just type as if the commands were being entered at the keyboard. Once the desired commands are entered, press the F6 key to exit the Function Key Menu. You have now defined a Function Key.

3.3. Function Key Editor

The Function Key Editor consists of editing area and a listing of the Function Key definitions while in the Function Key Editor. While in the Function Key Editor, the F1 through F6 keys have the following functions.

Function	Description

Key

F1 - Displays the contents of the next Function Key, increasing from F1 to F12 and wrapping around to F1 again.

F2 - Displays the contents of the previous Function Key, decreasing from F12 to F1 and wrapping around to F12 again.

F3 - Clears all data in the displayed Function Key buffer.

F4 - Allows entry of a 12 character label for the Function Key.

F5 - Used to reformat lines longer than 64 characters read in from a file, fit the 64 characters per line length of the display.

F6 - Exits the Function Key Editor, and saves the data in memory.

4. APPLICATION HINTS

4.1. Allocating Connections

To debug various kinds of messages and objects on a slave device it is necessary to establish at least an Explicit Messaging Connection. Since each connection has associated with it an expected packet rate (EPR) which causes the connection to time out after four(4) times the EPR has elapsed with no message; it is useful to set the EPRs of all connections to zero. This can be accomplished with either a function key definition or a script file with the following information:

```
masterid  1
slaveid  60
allocate  7
setepr 1 0
setepr 2 0
setepr 3 0
```

If this is assigned to a function key buffer and saved then it can be used to establish the connections after resetting the slave device.

4.2. Simple Scanner

The functions of a simple scanner (MACID = 1) with two slave devices (MACIDs 52 and 54) can be implemented with one function key definition and one script file.

The function key definition would look like:

```
masterid  1
slaveid  52
allocate  3
setepr  1  0
setepr  2  50
slaveid  54
allocate  3
setepr  1 0
setepr  2 50
```

The script file would look like

```
slaveid  52
poll  07 03 05 09
```

```
slaveid 54
poll  08 04 06 0A
replay
```

After executing the function key once and receiving a success response to the allocate and set EPR messages, type "**play**" and the name of the script file to start a continuous scan of the two slave devices.

4.3. Nested Play Files

One of the uses of master emulator is automating a checkout procedure. Each DeviceNet device is required to implement certain object classes. To get the value of each attribute of each object class and to verify that illegal values produce the expected error response a sequence of nested play files may be created. In the following scripts the notation **(eof)** represents the end of file. It should not be placed literally in the file.

The top level play file called **getall.scr** might look like the following:

```
#       getall.scr
#       Do the Identity Object
play    idobj.scr
#       Do the DeviceNet Object
play    dnetobj.scr
#       Do the predefined connection objects
class  5
instance  1
play    cnxn.scr
instance 2
play    cnxn.scr
instance 3
play    cnxn.scr
(eof)
```

The file idobj.scr might look like the following:

```
#       idobj.scr
class   1
instance  1
get 1
get 2
get 3
...
get  7
#       Verify error on attribute #8
get  8
```

(eof)

The file dnetobj.scr might look like the following

```
#       dnetobj.scr
class  3
instance  1
get  1
get  2
get  3
get  4
get  5
#       Verify error on attribute #6
get  6
(eof)
```

The file cnxn.scr might look like the following:

```
#       cnxn.scr
get  1
get  2
...
#       Verify errors on attribute #10 and #11 per DeviceNet Spec. Vol. I, Rev 2.0, p5-7
get  10
get  11
...
get  16
(eof)
```

The whole process is invoked from the keyboard by entering the command
play getall.scr Then press return

If a record of the transactions is required a **log** command may be inserted at strategic points.

4.4. Keyboard Escape

In this example we presume that a group of slave devices need to be given their serial numbers to be stored in some non-volatile memory. We further assume that some sequence of DeviceNet explicit messages is used to provide an unlock mechanism for programming the serial number. We want to construct a play file that will automate the procedure. In this example the **masterid** is 1 and the **slaveid** is 63. Also the Function Key F1 is presumed to contain an **allocate** command and the appropriate **setepr** commands. To unlock the serial number we must do a series of **get** commands to various classes, instances, and attributes: followed by a set to the serial number attribute of the identity object. An example script might look like the following:

```
#       serial.scr
#       Unlock the set serial number lock
class   20
instance 45
get  7
get  100
get  1
#
#       set the serial number
#       Must be a fragmented request
#       Enter four hexadecimal bytes in place of the dollar sign ($)
#
[5E4]<81 00 10 01 01 06 $>
```
(eof)

The $ will interrupt the processing flow and allow the parameters of the set command to be entered from the keyboard replacing the $