Huron Net Works, Inc.

DN-CMEM

DeviceNet™ Console Master Emulator

User's Guide

Revision	Date	Description
1.1	5/2/03	Original
1.2	5/14/03	Add offline, fault, claim, and disown. Add section on log files and macros
1.3	7/8/03	Change name on this document to DN-CMEM to reflect current catalog number. Original program name and internal error message tag remain "CDMEM"
1.4	3/26/04	Add dupmac and monitor keywords
1.5	6/8/05	Add ASCII keyword. Add I/O fragmentation for POLL request. Add I/O fragmentation for CAN literal notation.
1.6	5/7/06	Add keyword READ, WRITE, DIR, CD, and TYPE to support directory navigation, and small file editing.

Table of Contnts

1. GETTING STARTED

1.1. General Description

DeviceNet Console Master Emulator (DN-CMEM) is derived from the DN-MEM software that runs in conjunction with the DN-PC1 and DN-PC2 cards. These cards are now obsolete along with the ISA bus. Console Master emulator is a general-purpose tool used in the development and debugging of DeviceNet nodes. It can be used for other types of CAN networks because it does not directly implement the DeviceNet Protocol stack. It does implement certain DeviceNet specific features with respect to acknowledgement and fragmentation. DN-CMEM runs on the DeviceGate module and interfaces with several popular terminal emulators. It has the ability to monitor the DeviceNet bus for traffic. DN-CMEM displays incoming messages, with time stamps, on the screen, and can log them to a file for later analysis.

DN-CMEM 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, the UCMM, and the Offline Connection Set have defined keywords so that memorizing the exact format of common messages is not required. Sequences of messages and commands may be assigned to any of the twelve function keys and manually executed when the function key is depressed. Automated test suites can be placed in script files and executed with a single command. The script file may contain a **replay** command (See §2.6) to allow continuous testing. The script file may also contain a **play** (See §2.5), or a **chain** (See §2.24) command, which allows multiple files to be treated as a single file.

In the original design, all required files were supposed to reside in the root directory of the DeviceGate's solid state disk drive. There is a limitation of 32 files per folder on this simulated disk drive. Directory navigation on the Device Gate is accomplished with the **dir** (See §) command, and the **cd** (See §) command. In addition to directory navigation, it is now possible to edit small files directly within DN-CMEM. Small files can be read into a single key buffer with the **read** (See) command, and written from a key buffer with a **write** (See) command.

1.2. Invocation

Console Master Emulator runs on the DeviceGate Module and communicates with a terminal emulator. It is invoked by typing its name – cdmem – in response to the command prompt “A:\>” There are two optional parameters, which can be specified on the command line. These are baudrate and terminal port. 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 125. The default terminal port is zero, which corresponds to the EXT port on the DeviceGate. This port is accessible via the external DB-9 Male Connector. Valid terminal ports are 0,1,2, which correspond to ports EXT, COM, and TELNET on the DeviceGate. The COM port does not have a connector installed on the board. Baudrate can be conveniently specified as a command line parameter, or from the keyboard via the **baudrate** command (See §2.17), or in the **autoexec.scr** script file (See §2.5). Valid baudrates are 125, 250, and 500.

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

To exit Console Master Emulator use ^X. Hold the control key (Ctrl) and press the letter X. Alternatively depressing the Esc key twice, written as ESC-ESC, will also exit the program.

1.3. Screen Layout

Unlike the original Master Emulator, the split screen layout is no longer practical with an ANSI Terminal Emulator so some adjustments have been made. If the terminal emulator understands ANSI escape sequences, the main screen background color is cyan. Transmitted frames, keyboard input, and command results are in bold yellow. Received frames are shown in bold white. Informational text is in bold green.

All messages are shown on the main screen in hexadecimal form with the CAN identifier enclosed in square brackets -- [ ] -- and the CAN data field enclosed in angle brackets -- < > --. This notation is called CAN literal notation, which replaces the term “square bracket/angle bracket notation”. Packet counters and time stamps are shown at the beginning of a line enclosed in colons. Descriptions of transmitted and received messages are shown at the end of the line.

1.4. Keyboard

In the original version of Master Emulator (DMEM) keyboard input came directly from the console functions on the PC. Now it must come through a terminal emulator. Different terminal emulators have a multitude of methods for encoding and using the Function Keys (F1 through F12) and the Editing Keys (Insert, Delete, Home, End, Page Up, and Page Down). Console Master Emulator defines a set of Control Keys and Escape Sequences that it understands. Some terminal emulators can be configured to produce these sequences when certain keys are depressed. If you use a rudimentary terminal emulator it may take some practice to use two or three key sequences for the Function Keys. Two observations may help. The reverse tick (`) used to create the Shift/Control/Alt function key sequences is right under the ESC Key. The sequences for the individual keys correspond to the order of the keys on a standard typewriter keyboard with one exception. Alt-F12 uses the backslash as the third character of the sequence, which is actually up one row from the rest of the Alt-Function Keys.

1.5. Program Input

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

- Keyboard Input
- Function Key Buffer(s)
- Script File(s)

Keyboard Input comes from the keyboard and is processed one character at a time into messages, commands or comments. Function Key Buffers are sequences of messages, commands and parameters, or comments, on multiple lines, that are saved and recalled with a single Function Key (F1 through F12). The maximum length of a Function Key Buffer is 640 characters. Longer sequences of messages, commands and parameters, and comments are contained in a Script File (*.scr). Console Master Emulator uses two other types of files: the Function Key Definition (*.key) file, and the Log (*.log) file. These files and their usage is discussed in **load**(§2.2)**, save**(§2.3)**,** and **log**(§2.4).

The input from any of the three sources consists of messages, commands, or comments. Commands consist of a keyword (§2) 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 (" "), a percent (%), or a newline (\n). This convention allows all of the files created by Console Master Emulator to be used as script files. This includes key definition (*.key) files and log (*.log) files. Old key files can be adapted to the new convention by inserting a newline('\n') after the "%D".

DeviceNet messages, in CAN literal notation, consist of an identifier and a message body, and 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. Optional spaces between message bytes improve readability. For example a message with the identifier 0x5E4 and two data bytes (0x55 and 0xAA) would be written as follows:

```
[5E4]<55 AA>
```

This CAN literal notation is designed to allow embedded End of Line characters so that a fragmented messages may actually cover several lines without the use of line continuation characters. This may lead to some confusion if the number of opening and closing brackets does not match. In these circumstances it may look as though the program is not responding. The bell character, or Control-G (^G) implements a forceful end of line abort condition with respect to unmatched brackets. After entering the bell character you can start over with a new command.

Without the split screen there are cases where input from one of the input sources will be overlapped with output to the screen. To avoid the ungainly appearance of this overlapping, Console Master Emulator implements a modified form of the XON/XOFF protocol. The normal default mode is “XON”. In this mode input characters are echoed to the screen as they are entered from the keyboard or read from a buffer or a file. The alternate mode is “XOFF”. In this mode input characters are echoed to the screen **only when there is a complete line**. Switching between these two modes is accomplished with the Control-Q (^Q) for XON and the Control-S (^S) for XOFF. When Console Master Emulator is in the XOFF mode you have to tolerate the lack of echoing as you type. If this is a serious problem then you have to put up with overlapping of keyboard input and screen output.

1.6. Fragmented Messages

1.6.1. Fragmented Explicit Messages

An Explicit Message with more than eight bytes specified in the message body will be sent out as a **fragmented explicit message**. The value of the identifier is used for this determination. All messages in Group 3 and the Explicit Request and Explicit Response in Group 2 are treated this way. 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 Console Master Emulator.

When receiving fragmented explicit responses; it is necessary to generate fragment acknowledgements. There are several commands that help control this process including **fragackid** (§2.18)**, fragokid**(§2.19)**, claim**(§2.35)**,** and **disown**(§2.36).

1.6.2. Fragmented I/O Messages

An I/O Message with more than eight bytes specified in the message body will be sent out as a **fragmented I/O message**. The value of the identifier is used for this determination. All messages in Group 1 and the Multicast Poll Request and the Poll Reques in Group 2 are treated this way. I/O fragments are not acknowledged and there is no need have a place holder for the fragmentation bytes.

1.6.3. Fragmented Other Messages

Any other type of message in Group 2 or Group 4 is assumed to be unfragmented and will be truncated and sent if the message body exceeds eight bytes.

1.7. Explicit Message Types

An Explicit Request Message can be constructed in two different ways. The first method uses the Predefined Master/Slave Connection Set. These explicit request messages have an Identifier in Group 2; Destination MAC ID, and Message Id four (4). The message header in the data field contains the Source MAC ID, which is the value of the Master MAC ID. The second method uses a presumed Group 3 or Group 1 identifier supposedly allocated by a UCMM (Unconnected Message Manager). 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. Switching between these two construction methods is accomplished with the **ucmm** (§2.32) command.

1.8. 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 Adaptation of CIP Specification defines four choices for how to handle this value. See Vol. 3,Edition 1.0, p. 2-27. The command **bodyformat** (§2.31) with a numeric argument is used to select one of the four choices. The default value, zero(0), is to construct Explicit Request Messages with an eight (8)-bit Class Identifier, and an eight (8)-bit Instance Identifier.

1.9. Log Files

Console Master Emulator has the ability to create log files on the DeviceGate's Flash Disk. Sadly the maximum size of this Flash Disk is only 256K bytes, or about the same size as an original low density floppy disk. Room on the Flash Disk is needed for programs and their data files, so we recommend that log files be created and maintained by the terminal emulator on the PC platform. If this is not practical for some reason, an alternate approach is to periodically transfer the log files on the DeviceGate, using FTP, to files on the PC platform supporting the terminal emulator.

2. COMMANDS

Chapter Preview

2.1. help

The **help** command causes a list of commands to be displayed. The list is displayed on one page. If your terminal emulator is not big enough you may have to scroll backwards to see the entire screen. The terminal emulator should allow a setting for the number of lines in a window.

2.2. load

The **load** command, with 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. Console 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 directory of function key definition files, type "load", without a filename, then, press enter. The text will consist of a list of *.key files from the directory in which Master Emulator is currently being run. In addition to the *.key files, any folders at the current level are displayed in a separate list.

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. In addition to the text there are three separator tags which are described in the table below:

Separator Tag	Meaning
%L	Text up to the next % is a key buffer label
%D	Text up to the next % is key buffer data
%E	End of File

To obtain a directory of function key definition files, type "save", without a filename, then, press enter. The text will consist of a list of *.key files from the directory in which Master Emulator is currently being run. In addition to the *.key files, any folders at the current level are displayed in a separate list.

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 filename 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 Console 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 §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. 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 (See §2.6) with no parameter as the last line of the file.

To obtain a directory of script files, use a **play** command without a filename. Then, press enter. The text will consist of a list of *.scr files from the directory in which Console Master Emulator is currently being run. In addition to the *.scr files, any folders at the current level are displayed in a separate list.

When input is being taken from a script file, or a key buffer, the dollar sign ($) character may be used as a temporary keyboard escape. All keyboard input up to the next return goes in place of the dollar sign in the currently executing script file or key buffer. 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 §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 predefined poll and strobe connections on one of the slave devices on the network must have been allocated. See §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 §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, Release 2.0, Chapter 7). The single parameter for the **allocate** command is a hexadecimal encoded bit mask which is, described in the DeviceNet Specification Vol. I, Release 2.0, p.5-57, and constructed as follows:

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

NACK	Acknowledge Suppression
CYC	Cyclic Connection
COS	Change of State Connection
MCP	Multicast Poll 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, a Multicast Poll 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
allocate 50    To open a COS connection with Ack Suppression
```

*** Note:**
It is not possible to allocate just the I/O Connections without the Explicit Messaging Connection. It is possible to allocate the desired 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 (Vol. 1 CIP Common Specification, Edition 2.0, Chapter 3, pp. 28-29). 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
5	Multicast Poll 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 the node'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 any amount of data to the destination slave device. If there are more than eight bytes to send they will be sent as multiple I/O fragments. 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 §2.6 for an example program.

2.11. mcpoll

The **mcpoll** command is used to send packets of date to a group of slave devices using the Master’s Multicast Poll command. [Group 2, Multicast Maid, MsgId 1]. The **mcpoll** command may send any amount of data to the destination slave device. If there are more than eight bytes to send they will be sent as multiple I/O fragments.

The implementation of this command has not been extensively tested since after at least five years of being defined in the specification there are no identifiable, commercially available implementations of multicast poll.

2.12. 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 is normally eight bytes and each allocated slave device is assigned one bit out of the sixty-four, corresponding to its MAC ID, to be used as it wishes. Each allocated slave device receiving a bit strobe from its master produces a bit strobe response. Console Master Emulator also supports the use of the zero length strobe. In this case, just the command with no data is entered.

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.
```

By definition, neither the strobe request, nor the strobe response can be fragmented.

2.13. 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** command.

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

CYC	Cyclic Connection
COS	Change of State Connection
MCP	Multicast Poll 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 8      To release the Multicast Poll Connection
release 10     To release the COS connection
release 20     To release the Cyclic Connection
release 1F     To release the COS and all other
Connections
release 2F     To release the CYCLIC and all other
               Connections
```

2.14. 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, mcpoll, 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).

2.15. 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, mcpoll, 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).

2.16. multicastid

The **multicastid** command takes a decimal number in the range [0..63] and establishes an internal value to use with the **mcpoll** command. To set the multicast MAC ID to 11, type the following:

```
multicastid 11     Then press Enter.
```

Once set the multicastid remains in effect until it is changed. The default value is one (1), which corresponds to the default masterid.

2.17. 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.
baudrate 250       to set the baudrate to 250 Kbaud.
baudrate 500       to set the baudrate to 500 Kbaud.
```

On a network which is running this change may upset any nodes which do not track this change including forcing them or the DeviceGate module to a **Bus Off** condition.

2.18. fragackid

The **fragackid** command is used to tell Console Master Emulator what identifier to use when acknowledging a fragmented response. This response would be from an explicit message request. For Explicit Messaging Connections, which do not use the predefined Master Slave Connection Set, failure to specify this parameter before receiving a fragmented response may cause the slave device to hang up. For example to tell Console 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. If this command is given with no parameter then the current value is displayed.

2.19. fragokid

The **fragokid** command is used to tell Console Master Emulator which message identifiers to check for the fragmentation bit being set. For Explicit Messaging Connections, which do not use the predefined Master Slave Connection Set, failure to set this parameter may cause 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. If this command is given with no parameter then the current value is displayed.

2.20. class

The **class** command, followed by a hexadecimal parameter, is used to select a DeviceNet Class Identifier for a subsequent **get** (attribute single), **set** (attribute single), or **reset** commands. 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 DN-CMEM 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 CIP specification in decimal.**

2.21. 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 DN-CMEM 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.

2.22. 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 Chapter 5 of the CIP Common Specification we see that Vendor ID 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 next line and look something like:

```
[5FB] <01 8E 14 00>
```

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

2.23. set

The **set** command is the compliment 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].

2.24. chain

The **chain** command takes a filename as a parameter. This command preserves the functionality of the **play** command (See §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 §2.5 on play for the new description of its functionality. To obtain a directory of script files, type “chain”, without a filename, then, press enter. The text will consist of a list of *.scr files from the directory in which Master Emulator is currently being run. In addition to the *.scr files, any folders at the current level are displayed in a separate list.

2.25. mask

The **mask** command takes up to four (4) hexadecimal arguments. These four bytes map to the Acceptance Mask Registers of the SJA-1000 CAN Chip. The Acceptance Mask Registers in conjunction with Acceptance Code Registers form a pair of hardware screeners. These hardware screeners in the SJA-1000 CAN chip on the **DeviceGate** can be programmed to accept certain messages and ignore all other traffic on the network. In constructing the mask values place a zero (0) in each position 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". 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.

Both hardware screeners in the SJA-1000 CAN Controller chip are active. The first one operates on the eleven-bit identifier field, the RTR bit (always zero for DeviceNet) and the first byte of the data field. The second screener works on the eleven-bit identifier field and the RTR bit. Both screeners may be set to pass the same frames if only one screener is required.

This command will redisplay the four bytes entered in a more convenient notation which shows the eleven bit identifier mask, the RTR mask, and the data mask for mask1, followed by the eleven bit identifier mask and the RTR mask for Mask2. See the example in the following section.

2.26. match

The **match** command takes up to four (4) hexadecimal arguments. These four bytes map to the Acceptance Code Registers of the SJA-1000 CAN Chip. The Acceptance Code Registers in conjunction with the Acceptance Mask Registers form a pair of hardware screeners. These hardware screeners in the SJA-1000 chip on the **DeviceGate** can be programmed to accept certain messages and ignore all other traffic on the network. In constructing the match values place a zero (0) in each position that is to be a zero (0). Place a one (1) in each position which is to be a one (1). For positions, which are “don’t care” according to the **mask** place either a zero (0) or a one (1). 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.

Both hardware screeners in the SJA-1000 CAN Controller chip are active. The first one operates on the eleven-bit identifier field, the RTR bit (always zero for DeviceNet) and the first byte of the data field. The second screener works on the eleven-bit identifier field and the RTR bit. Both screeners may be set to pass the same frames if only one screener is required.

This command will redisplay the four bytes entered in a more convenient notation which shows the eleven bit identifier match, the RTR match, and the data match for match1, followed by the eleven bit match and the RTR match for Match2.

Mask & Match Example #1

Suppose that we want to monitor only the **poll** responses from a particular slave node (Node 60), with the data byte equal to 0x3X with the first screener, and suppose we want to monitor all UCMM messages with the second screener. See Appendix B for the details of the mapping of Acceptance Mask and Acceptance Code bit onto the bits of a CAN Frame.
We construct the screener mask & match conditions as follows:

```
Screener #1
     ID BIT    10   = 0       Group 1 Message
     ID BITS   9-6  = 1111    PollResponse Message ID
     ID BITS   5-0  = 111100  MAC ID 60
     RTR BIT        = 0
     DATA BITS 7-4  = 0011
     DATA BITS 3-0  = XXXX

Screener #2
     ID BITS   10-9 = 11      Group 3 Message
     ID BITS   8-6  = 11X     UCMM Request/Response
     ID BITS   5-0  = XXXXXX  Any Source Maid
     RTR BIT        = 0
```

From the values for the screeners we construct the register contents as follows.

```
mask      00 00 0F EF
CDMEM: Mask1 = 0x000,0;0x0F
CDMEM: Mask2 = 0x07F,0;--

match     7F 83 F0 00
CDMEM: Match1 = 0x3FC,0;0x30
CDMEM: Match2 = 0x780,0;--
```

```
    baudrate  125
```

The default for mask is all ones, which lets everything through, and the default match is all zeros.

2.27. 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.

2.28. cosack

The **cosack** command will take up to eight hexadecimal parameters and save this data. When a COS message arrives from the slaveid device DN-CMEM will 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. In most cases the COS Acknowledge message should have no data and this is the default condition.

2.29. 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, Release 2.0, pp. 4-7 thru 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.

2.30. 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, Release 2.0, pp. 4-17 thru 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

2.31. 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

2.32. 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.

2.33. offline

The **offline** command constructs and sends an Offline Ownership Request Message. The data field may contain up to eight arbitrary data bytes. There should be no response to this message unless another master on the network has already claimed ownership of the offline connection set.

2.34. fault

The **fault** command constructs and sends a Communication Faulted Request Message. The data field may contain up to eight arbitrary data bytes. There may or may not be responses to these messages depending on the presence or absence of faulted nodes on a given network.

One form of the Communication Faulted Request Message is used to change the MAC ID of a faulted node without doing a power cycle on the node.

2.35. claim

The **claim** command takes one or more node addresses separated by spaces and adds them to a list of nodes for which Console Master Emulator will be responsible for acknowledging COS and Cyclic production, and acknowledging fragmented explicit production. This responsibility is a great deal less than full mastership but does avoid having to set this process up manually. By default Console Master Emulator will take responsibility for the default Slave MAC ID which is 63.

Input arguments greater than 63 are ignored. The present **masterid** value cannot be claimed, and is ignored. The node being claimed must either have no owner or be owned by the present master. All other values are ignored.

The final result of the claiming process is that a table of identifiers is produced which trigger appropriate productions when those identifiers are consumed. Normally these identifiers are from the predefined connection set. When using a ucmm created explicit connection make sure that **fragackid** and **fragokid** are set correctly for a node <u>before</u> it is claimed.

2.36. disown

The **disown** command takes one or more node addresses and deletes them from the list of nodes for which Console Master Emulator will acknowledge COS and Cyclic productions and fragmented explicit productions. For Console Master Emulator to be a true monitor all nodes should be disowned, or the **masterid** should be changed to an address, which is not responsible for acknowledgement.

Input arguments greater than 63 are ignored. The present **masterid** value cannot be claimed and is ignored. To disown a node the present master must have previously claimed it. All other values are ignored.

Be careful about disowning a node and expecting proper behavior when it produces a COS/Cyclic message or an explicit fragmented response.

2.37. dupmac

The **dupmac** command takes zero to eight bytes of hexadecimal data. The first data byte is a node address. The remaining seven bytes are the contents of a dupmac message. Unspecified bytes are set to zero. Sending a Dup Mac Check Request to node #1 would look like

```
dupmac 1
[40F]<00 00 00 00 00 00 00>
```

Sending a Dup Mac Check Response to node #1 would look like

```
dupmac 1 80
[40F]<80 00 00 00 00 00 00>
```

Finally sending a Dup Mac Check Request to node #1, port 0, Vendor Id 20 and Serial Number 0x00007090 would look like

```
dupmac 1 00 14 00 90 70 00 00
[40F]<00 14 00 90 70 00 00>
```

2.38. monitor

The **monitor** command is used to change the mode of the CAN Controller from normal mode to listen only mode. In listen only mode the CAN Controller on the DeviceGate does not acknowledge received frames. In conjunction with the baudrate command the listen only mode can be used to determine the baudrate of a network. The procedure would be:

1. Start DN-CMEM with the DeviceGate powered, but disconnected from the network.
2. Use the **monitor** command to go into listen only mode.
3. Attach the DeviceGate to the network.
4. if no messages are displayed, use the baudrate command to change baudrates until received messages are displayed.
5. When the baudrate is established, use the **monitor** command again to exit the listen only mode.

2.39. ascii

The **ascii** command is used to change the value of an internal variable which controls the printing of the body of a CAN frame as an ASCII string. The default for this variable is off. The ASCII string is enclosed in curly braces ("{…}") after the hexidecimal string and before the description of the frame. It may be necessary to extend the line length in the terminal emulator to produce lines that do not wrap. Non-printable characters are displayed as periods (".").

2.40. dir

The **dir** command is used to display a directory of all the files at the current level. In addition each of the subfolders at the current level is displayed in a separate list. This command takes an optional argument, with wildcards, which can be used to display a restricted set of files. For example

```
dir *.exe
```

might display the following lines.

```
  Directory of *.exe files
    CDMEM.EXE      DAYTIME.EXE    DNDRV.EXE

  Other Folders
    ZED            SCR
```

and

```
dir
```

might display the following lines

```
  Directory of *.* files
    ERASE.SCR      CDMEM.EXE      WRITEFF.SCR    SERIAL6E.SCR   CHIP.INI
    DAYTIME.EXE    ZED            XX.SCR         DNDRV.EXE      XX.TXT
    CHIP.TST       YY.SCR         SCR

  Other Folders
    ZED            SCR
```

2.41. cd

The **cd** command is used to change the current directory. There is a required parameter which is the name of the subdirectory or folder that is to become the new working directory. The shorthand names period "." and double period ".." refer to the current directory and the parent directory respectively. If the parameter is missing or does not correspond to a subdirectory name then the error message is

```
CDMEM: Unable to go there
```

2.42. type

The **type** command is used to quickly view the contents of a text file. If the name corresponds to a valid file then the file is opened and the contents are written to the display. If the parameter is missing then a list of all files is displayed. Any subfolders in the current directory are displayed in a separate list. For example, to display the contents of the CHIP.INI file use the following:

```
type chip.ini
[IP]
DHCP=0
ADDRESS=192.168.2.87
NETMASK=255.255.255.0
GATEWAY=192.168.2.9
```

```
[STDIO]
STDIN=COM EXT TELNET
STDOUT=COM EXT TELNET
[SERIAL]
EXT_BAUD=115200
```

2.43. read

The **read** command reads a small file into a key buffer. It takes two optional parameters, a filename and a buffer number in the range [1..12]. If the buffer number is omitted then the next free buffer is used. If both parameters are omitted then a directory of all files is diplayed, and a separate list of subdirectories is displayed. If the command is successfule then a message will be displayed with the number of characters read and which of the key buffers they were placed in.

2.44. write

The **write** command writes a key buffer into a file. It takes two parameters, a buffer number and a file name. If the file name is omitted there is an error message. If the buffer number is omitted then the current buffer is written.

3. <u>FUNCTION KEYS</u>

3.1. Terminal Emulators, Displays, and Keyboards

The original Master Emulator used console I/O to display information and accept input from the keyboard. Programs running on the DeviceGate module do not have access to a PC display and keyboard. This sort of input and output must come over a serial port or over the Ethernet. Console Master Emulator uses a Terminal Emulator running on a PC as the display, and keyboard devices.

A partial standard called “ANSI Escape Sequences” exists and provides some foundation for creating screen-based applications to work in conjunction with so-called “dumb terminals” or “glass teletypes”. There is no identifiable and consistent standard for representing the function keys, or the editing keys on a PC keyboard in terms of either escape sequences or characters on a serial port.

Console Master Emulator has therefore created a set of Escape Sequences and Control Characters, which it understands and can map to various functions. For the Function Keys, these sequences use the ESC key and the numbers along the top row of the keyboard. The zero (0) stands for F10 (ESC 0), the minus stands for F11, and the equal sign (=) for F12. The shifted versions of the function keys use the ESC key and the back-tick (lower case tilde) followed by a number key along the top row including minus and equal to stand for SHIFT-F1 through SHIFT-F12. The tables in Appendices A-1 and A-2 show how the method extends to Control and Alt Function Keys.

Terminal Emulator programs have a wide variety of methods and capabilities for mapping keys on the keyboard to various sequences of characters. <u>In the discussion that follows any reference to a key on the standard PC keyboard shall be understood to mean “that key” or an equivalent sequence of keys that maps to the desired function.</u>

This means that if you can program or configure a terminal emulator to map keyboard keys into the standard DN-CMEM sequences, then you can avoid learning a new technique for using function keys and editing keys.

3.2. Use of Function Keys

Function Keys are used to store common sequences of commands and messages in a single memory buffer. Then, pressing a single key the entire sequence is 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, cleared, saved, and recalled.

3.3. Defining a Function Key

If a Function Key has no definition, then the first time it is pressed an editing screen will appear and a definition may be entered. Once a key is defined subsequent keystrokes will cause the definition to be replayed as if the characters had been entered from the keyboard again. After a function key is defined, Shift-Fxx will allow editing of the definition. Alt-Fxx will clear the buffer associated with that function key. Ctrl-Fxx performs no function, however a message shows the key was processed.

To enter commands into a Function Key Buffer, just type, as if the commands were being entered at the keyboard. Once the desired commands are entered, press the ESC key twice, the Control-X key, or the F6 key, to exit the Function Key Editor. You have now defined a Function Key.

Exiting the Function Key Editor with the F6 key is retained for historical purposes, but its use for this purpose is discouraged. Use the more consistent double ESC, or the convenient control-X.

3.4. Function Key Editor

The Function Key Editor consists of an editing area, a label line, a prompt line, and a cursor information line. While in the Function Key Editor, the F1 through F5 keys have the following functions.

Function Key	Description
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 80 characters read in from a file, fit the 80 characters per line length of the display. This function is seldom necessary when using terminal emulators with a virtual screen size

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 (MAC IDs 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 Console 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 $. The keyboard escape now works from a Function Key Buffer as well as from a script file.

APPENDIX A.1

DN-CMEM -- Control Key Mapping

Code	Key	Token	Comment
0	^@	I	Illegal
1	^A	I	Illegal
2	^B	END	END or “BOTTOM”
3	^C	I	Illegal
4	^D	PGDN	Page Dn or “DOWN”
5	^E	END	END or “END”
6	^F	I	Illegal – OS Change FOCUS
7	^G	I	Illegal
8	^H	BS	Backspace
9	^I	TAB	Tab
10	^J	EOL	End of Line – linefeed or newline
11	^K	I	Illegal
12	^L	I	Illegal – if we echo this the screen is cleared
13	^M	EOL	End of Line – Carriage Return
14	^N	I	Illegal
15	^O	I	Illegal
16	^P	I	Illegal
17	^Q	XON	Keyboard Input overlaps Display Output
18	^R	I	Illegal
19	^S	XOFF	Keyboard Input echo delayed until End of Line
20	^T	HOME	HOME or “TOP”
21	^U	PGUP	Page Up or “UP”
22	^V	I	Illegal
23	^W	I	Illegal
24	^X	EXIT	EXIT Editor or DN-CMEM
25	^Y	I	Illegal
26	^Z	I	Illegal
27	^[	ESC	Escape Key
28	^\	DEL	DEL – replaces Delete Key
29	^]	I	Illegal
30	^^	INS	INS – replaces the Insert Key
31	^_	I	Illegal

Token is related to the function performed internally by DN-CMEM. These control characters can be used with terminal emulators that have no capability to map editing keys on the PC keyboard

APPENDIX A.2
DN-CMEM – Edit/Function Key Mapping

Escape Sequence	Key	Comment
ESC [	EP1	Escape Prefix 1 – ANSI Sequences
ESC O	EP2	Escape Prefix 2 – ANSI Sequences
ESC `	EP3	Escape Prefix 3 – DN-CMEM Function Keys
ESC [A	UP	Cursor Up One Line
ESC [B	DOWN	Cursor Down One Line
ESC [C	RIGHT	Cursor Right One Character
ESC [D	LEFT	Cursor Left One Character
ESC [H	HOME	The Home Key
ESC [K	END	The End Key
ESC O P	F1	HyperTerm’s default F1 Key
ESC O Q	F2	HyperTerm’s default F2 Key
ESC O R	F3	HyperTerm’s default F3 Key
ESC O S	F4	HyperTerms default F4 Key
ESC 1	F1	DN-CMEM Definition
ESC 2	F2	DN-CMEM Definition
ESC 3	F3	DN-CMEM Definition
ESC 4	F4	DN-CMEM Definition
ESC 5	F5	DN-CMEM Definition
ESC 6	F6	DN-CMEM Definition
ESC 7	F7	DN-CMEM Definition
ESC 8	F8	DN-CMEM Definition
ESC 9	F9	DN-CMEM Definition
ESC 0	F10	DN-CMEM Definition
ESC -	F11	DN-CMEM Definition
ESC =	F12	DN-CMEM Definition
ESC ` 1	SF1	Shift-F1, DN-CMEM Definition
ESC ` 2	SF2	Shift-F2, DN-CMEM Definition
ESC ` 3	SF3	Shift-F3, DN-CMEM Definition
ESC ` 4	SF4	Shift-F4, DN-CMEM Definition
ESC ` 5	SF5	Shift-F5, DN-CMEM Definition
ESC ` 6	SF6	Shift-F6, DN-CMEM Definition
ESC ` 7	SF7	Shift-F7, DN-CMEM Definition
ESC ` 8	SF8	Shift-F8, DN-CMEM Definition
ESC ` 9	SF9	Shift-F9, DN-CMEM Definition
ESC ` 0	SF10	Shift-F10, DN-CMEM Definition
ESC ` -	SF11	Shift-F11, DN-CMEM Definition
ESC ` =	SF12	Shift-F12, DN-CMEM Definition

Escape Sequence	Key	Comment
ESC ` q	CF1	Control-F1< DN-CMEM Definition
ESC ` w	CF2	Control-F2, DN-CMEM Definition
ESC ` e	CF3	Control-F3, DN-CMEM Definition
ESC ` r	CF4	Control-F4, DN-CMEM Definition
ESC ` t	CF5	Control-F5, DN-CMEM Definition
ESC ` y	CF6	Control-F6, DN-CMEM Definition
ESC ` u	CF7	Control-F7, DN-CMEM Definition
ESC ` i	CF8	Control-F8, DN-CMEM Definition
ESC ` o	CF9	Control-F9, DN-CMEM Definition
ESC ` p	CF10	Control-F10, DN-CMEM Definition
ESC ` [	CF11	Control-F11, DN-CMEM Definition
ESC `]	CF12	Control-F12, DN-CMEM Definition
ESC ` a	AF1	Alt-F1, DN-CMEM Definition
ESC ` s	AF2	Alt-F2, DN-CMEM Definition
ESC ` d	AF3	Alt-F3, DN-CMEM Definition
ESC ` f	AF4	Alt-F4, DN-CMEM Definition
ESC ` g	AF5	Alt-F5, DN-CMEM Definition
ESC ` h	AF6	Alt-F6, DN-CMEM Definition
ESC ` j	AF7	Alt-F7, DN-CMEM Definition
ESC ` k	AF8	Alt-F8, DN-CMEM Definition
ESC ` l	AF9	Alt-F9, DN-CMEM Definition
ESC ` ;	AF10	Alt-F10, DN-CMEM Definition
ESC ` ‘	AF11	Alt-F11, DN-CMEM Definition
ESC ` \	AF12	Alt-F12, DN-CMEM Definition

APPENDIX B

SJA-1000 Mask & Match Register Bit Mapping

CAN Message Bit	Mask1 Bit	Match1 Bit	Mask2 Bit	Match2 Bit
ID.10	AMR0.7	ACR0.7	AMR2.7	ACR2.7
ID.9	AMR0.6	ACR0.6	AMR2.6	ACR2.6
ID.8	AMR0.5	ACR0.5	AMR2.5	ACR2.5
ID.7	AMR0.4	ACR0.4	AMR2.4	ACR2.4
ID.6	AMR0.3	ACR0.3	AMR2.3	ACR2.3
ID.5	AMR0.2	ACR0.2	AMR2.2	ACR2.2
ID.4	AMR0.1	ACR0.1	AME2.1	ACR2.1
ID.3	AMR0.0	ACR0.0	AMR2.0	ACR2.0
ID.2	AMR1.7	ACR1.7	AMR3.7	ACR3.7
ID.1	AMR1.6	ACR1.6	AMR3.6	ACR3.6
ID.0	AMR1.5	ACR1.5	AMR3.5	ACR3.5
RTR	AMR1.4	ACR1.4	AMR3.4	ACR3.4
DATA0.7	AMR1.3	ACR1.3	-	-
DATA0.6	AMR1.2	ACR1.2	-	-
DATA0.5	AMR1.1	ACR1.1	-	-
DATA0.4	AMR1.0	ACR1.0	-	-
DATA0.3	AMR3.3	ACR3.3	-	-
DATA0.2	AMR3.2	ACR3.2	-	-
DATA0.1	AMR3.1	ACR3.1	-	-
DATA0.0	AMR3.0	ACR3.0	-	-

Mask and Match Command Mapping Example §2.26

mask	0x00	0x00	0x0F	0xEF
	AMR0	**AMR1**	**AMR2**	**AMR3**
match	0x7F	0x83	0xF0	0x00
	ACR0	**ACR1**	**ACR2**	**ACR3**