BiBaJa
Landscape Automation

USB12 TCL/TK DLL Reference

Version 1.1

INTRODUCTION

The USB12 DLL provides access to the USB12-01 USB to Power Line network interface from TCL/TK.

TCL/TK was chosen in part due to the simplicity and consistency of the command structure. TK also provided a simple means of constructing quick user interfaces for testing and demonstrations of the system under development.

Purpose of this Document

The purpose of this document is to describe how to install the USB12-01 driver and get the TCL/TK installed and connected to the USB12-01 adapter.

This document also provides a reference guide to the commands available in the USB12 DLL.

Finally a brief overview of LonWorks networking is provided with some pointers to other documents to assist you with learning more about the LonTalk protocol.

INSTALLATION

Successful installation of the requires 3 steps:

1) Installation of the FTDI D2XX Direct Driver for the USB interface
2) Installation of the TCL/TK software (Active State TCL/TK 8.4.1 release)
3) Copying BIBAJA.DLL and related *.TCL support files to the target run directory

FTDI D2XX Direct Driver Installation

The FTDI D2XX direct driver is required to communicate with the USB12-01 Power Line Network Interface. Before plugging in the USB12-01 adapter, make certain to read through this section and follow the directions carefully.

Locate the following directory on the CD:

```
\usb12-driver\D2XX10606
```

This is the direct driver for the FTDI FT232BM chip used in the USB12-01 adapter.

To install the driver, do the following:

1) Plug the Tamura supply into the wall and the USB12-01 adapter. The POWER LED should illuminate
2) Plug the USB cable into the USB12-01 adapter and the computer
3) You should see the "Found New Hardware" dialog
4) When prompted, specify the driver location from the CD given above

The FTD2XX direct driver should now be installed properly.

TCL/TK 8.4 Installation

The Active State TCL/TK 8.4.1 installation is provided in the following location:

```
\tcl-tk\ActiveTcl8.4.1.0-2-win32-ix86.exe
```

Launch the exe above and choose where you would like to install TCL/TK. Make certain the tcl\bin directory is in your path. Example TCL scripts will open with the "wish" TCL/TK application.

USB12 DLL and TCL/TK Example Application Copying

You may now copy the USB12 DLL and TCL/TK example application from the following location to your preferred location on the local hard disk:

\usb12-example\

Included in the usb12-example directory is the DLL BIBAJA.DLL and the supporting TCL scripts in *.TCL files.

If the FTD2XX direct driver and TCL are installed and in your path, you should be able to double-click on EXAMPLE.TCL to launch the example application.

Documentation

This document and other useful references are available on the CD in the folder:

\documentation

USB12-01 TCL/TK COMMAND REFERENCE

This section provides a reference for some of the commands available from the console command line when using the “BIBAJA.DLL” with TCL/TK and the “USB12.TCL” procedures. To load the DLL and procedures into TCL/TK, do the following:

1) Run “wish.exe” found typically in \tcl\bin
2) From the TCL/TK command prompt, type “load bibaja.dll”
3) From the TCL/TK command prompt, type “source USB12.TCL”

After these 3 steps, you will have TCL/TK initialized and ready to communicate with the USB12-01.

The following table gives a brief overview of the commands:

Command	**Page**	**Description**
ftdi_list	5	List available USB12 adapters
ftdi_open	5	Open USB12 adapter
ftdi_close	6	Close USB12 adapter
checksum	6	Recalculate the configuration and application checksum
clr_status	8	Clears the status counters in a device
download	8	Load a new application image into a device
dump_mem	9	Dump a range of memory from a device
dump_nv	10	Dump the values of a range of network variables from a device.
leave_domain	10	Invalidates domain structure 0 or 1 on a device
num_rcvd	11	Diagnostic routine to fetch number of messages addressed to a device
nv_cfg	12	Fetches a network variable’s configuration
nv_fetch	12	Fetches a network variable value from a device
nv_update	13	Updates an input network variable value
query_address	14	Fetch an NV address table entry from a device
query_domain	14	Fetch the domain structure 0 or 1 from a device
query_id	15	Find applicationless, unconfigured, or selected devices on the network
query_status	16	Fetch the status of a device
query_xcvr_status	17	Fetch the powerline transceiver status
read_mem	17	Read memory from a device
read_mem_sn	18	Subnet/Node address version of read_mem
reset_device	18	Reset a device on the network
residual	19	Runs a test to determine the residual error rate between the USB12 and a remote device
respond_to_query	19	Configure a device to respond or ignore query id requests
send_app	20	Diagnostic routine to send unacknowledged application

		layer messages to a device to determine error rate
send_msg	21	Sends a LonTalk message to the USB adapter or powerline network and returns a response
set_mode	22	Set the operating mode of a device
update_address	23	Update an NV address table entry on a device
update_domain	24	Update domain structure 0 or 1 on a device
update_key	25	Update the authentication key on a device
write_mem	26	Write memory of a device (CAUTION, see details)

BIBAJA.DLL Command Details

Loading the BIBAJA.DLL into TCL/TK will cause the prompt to change to "USB12> " and will provide a group of commands starting with the characters "ftdi". The chip used in the USB12-01 to provide the USB connectivity is the FT232BM from Future Technology Devices International Ltd. This is where the "ftdi" name for these commands originated.

ftdi_list

List the serial numbers of all available USB12-01 Power Line network interfaces. If no USB12-01 devices are detected, ftdi_list returns "-1". Otherwise a list of 8-byte ASCII serial numbers is returned for all available adapters.

Example:

```
USB12> ftdi_list
0027A7BE
```

In this example, one USB12-01 adapter is present with serial number "0027A7BE". Note the serial number of the USB12-01 corresponds to the middle 4 hexadecimal digits of the USB12-01's Neuron ID.

ftdi_open [serialNumber]

Open the USB12-01 interface, optionally by serial number. If no serial number is specified, then ftdi_open will open the first USB12-01 it finds attached to the system. This command returns "0" on success and "-1" on failure to open.

Example:

```
USB12> ftdi_list
0027A7BE
USB12> ftdi_open 0027A7BE
0
```

In this example, the "ftdi_list" command is used to find all available USB12-01 interfaces. The serial number returned by ftdi_list is then used to open the USB12-01 interface using the "ftdi_open" command.

ftdi_close

Close the USB12-01 interface. On success, this command returns “0”. If the close fails because the adapter wasn’t open or was unplugged, this command will return “-1”.

Example:

```
USB12> ftdi_close
0
USB12> ftdi_close
-1
```

In this example, the first ftdi_close closes the connection to the USB12-01. The subsequent ftdi_close fails and returns –1 because the USB12-01 was already closed.

Procedures Provided in USB12.TCL

This section documents the procedures provided in the file “USB12.TCL”. These procedures are used to build and send LonTalk messages to devices such as the IOPoint-PL and IVC24-04.

Procedures are documented using the following notation:

```
procedure_name required_arg1 required_arg2 [ optional_arg1 optional_arg2 ]
```

The following table defines some common required arguments:

Argument Name	Examples	Description
nid	“0x05 0x00 0x27 0xA7 0xBE 0x00” { 0x05 0x00 0x27 0xA7 0xBE 0x00 }	6-byte unique ID (Neuron ID) of a device on the network. The NID is specified as a 6-byte TCL list and may be enclosed in “” or {}.

checksum nid [checksum]

Recalculate the configuration checksum, or both the configuration and application checksums. This command is typically only used after downloading an application or when writing the configuration data using a write_mem command. It is not necessary to update the checksum unless configuration or application information has been changed. Note that writing configuration information or application code using write_mem while the node is running will cause a checksum error to be logged and the device state will be set to applicationless. See write_mem for more details.

“nid” is the 6-byte Neuron ID of the target device.

The default value for “checksum” is “both”. You may specify either “config” or “both” on the command line.

Example:

```
USB12> set nid “0x05 0x00 0x27 0xA7 0xBE 0x00”
USB12> checksum  $nid config
```

This example updates the configuration checksum only of a device on the network.

clr_status nid

Clear the status counters of a device. The status counters hold information such as last error logged, number of packets received by this node, number of CRC errors from the network, cause of last reset, etc. This command resets these counters to zero and clears status such as the cause of last reset.

“nid” is the 6-byte Neuron ID of the target device.
Example:

```
	USB12> clr_status $nid
```

This example clears the status of the device specified in the TCL variable $nid. See “checksum” command example for how to set the TCL variable $nid.

download nid imageFile

Download a new application image into a specified device. This command is typically used to upgrade or change the application in a device. When problems are found in the application code running on an IVC24-04, an update can be released and downloaded into the EEPROM of the IVC24-04 over the network. This command may be used to load a completely different application altogether, which is more common for Bibaja’s IOPoint-PL product line. IOPoints are generic points of I/O on the network that may be configured to perform different I/O operations depending on the hardware surrounding the IOPoint-PL module and the software loaded into the IOPoint-PL module. Various Ready-Made applications will be developed over time and available for free download to IOPoint-PL customers.

“nid” is the 6-byte Neuron ID of the target device.

“imageFile” is a filename of the file that contains the Intel Hex data to load into the device. The filename usually has the extension .NEI indicating it is a network downloadable image for an Echelon Neuron microcontroller.

Example:

```
	USB12> download $nid vc.nei
```

This example downloads the vc.nei (Valve Controller application) into the device specified by the TCL variable $nid. See “checksum” command example for example of setting the $nid variable.

dump_mem nid startAddress endAddress

Dump contents from a range in memory. This command is used to explore values in memory such as RAM and EEPROM during development. Some useful things to browse in EEPROM are the read-only data structure (documented in Appendix A of the Toshiba Neuron Databook). Appendix A also documents other useful structures worth exploring in the Neuron chip's memory.

The memory of the PL3120 Combination Neuron/Powerline transceiver embedded in the IVC24-04, IOPoint-PL, and the USB12-01 is split into the following major sections:

Start	End	Section
0x0000	0x5FFF	Firmware ROM
0x6000	0xE7FF	Unused (reads 0xFF)
0xE800	0xEFFF	RAM
0xF000	0xFBFF	EEPROM
0xFC00	0xFEFF	Unused (reads 0xFF)
0xFF00	0xFFFF	I/O Register Space

NOTE: Dumping memory in the I/O register space is not advised. There are test and set registers that will be set when read in this space. Unexpected setting of these registers from the application processor will cause a watchdog reset (since someone owns the lock and won't clear it.)

"nid" is the 6-byte Neuron ID of the target device.

"startAddress" is the 16-bit starting location to dump from.

"endAddress" is the 16-bit ending location to dump to.

Example:
```
  USB12> dump_mem $nid 0 0x1f
  0000: 0e 75 5f 2f 01 80 21 85 75 f0 26 58 ee a3 50 a6
.u_/..!.uð&Xî£P¦
  0010: 7f 58 ee a3 50 e6 7f e7 80 70 fd 31 f6 f6 a3 55
.Xî£Pæ.ç.pý1öö£U
```

This example dumps 32 bytes from the firmware ROM starting at location 0x0000 up through location 0x001F.

dump_nv nid startIndex endIndex

Dump the network variable value for a specified range of network variables. The nv_fetch command is called to retrieve the value of the network variable.

The valid range of network variables in the IVC24-04 device (as of this writing) is 0 to 19. For the IOPoint-PL module, the valid range of network variables depends on the Ready-Made or user developed application. Reading beyond the valid range will result in the device logging an error (invalid network variable index.) This error may be cleared using the "clr_status" command.

"nid" is the 6-byte Neuron ID of the target device.

"startIndex" is the starting index of the network variable to fetch.

"endIndex" is the ending index of the network variable to fetch.

Example:

```
USB12> dump_nv $nid 0 1
Index 0: Value 0xc8 0x01
Index 1: Value 0x00 0x00
```

This example dumps the value network variables 0 and 1 from a device.

leave_domain nid index

Invalidate one of the domain entries in the remote device. This command is used to unconfigure a device. Every Bibaja device has two domain entries, 0 and 1. Typically domain 0 will be the only configured entry in a device. The USB12-01 device uses domain 0 as the domain of the entire network, and domain 1 for the network management domain. One requirement for LonWorks networks is that network management is done using the zero-length domain. If you accidentally use leave_domain on your USB12-01 adapter, you will need to use the update_domain function to reconfigure domain entry 1 as a zero length domain before you will be able to access the network again.

"nid" is the 6-byte Neuron ID of the target device.

"index" is the index of the domain entry to invalidate, either 0 or 1 for most devices.

Example:

```
    USB12> leave_domain $nid 0
```

This example causes the device specified by TCL variable $nid to invalidate domain table entry 0. If no configured domain table entries remain, the device changes to the unconfigured state.

num_rcvd nid

Retrieve the number of messages received that were addressed to this device. This command is typically used in conjunction with the send_app command to perform error rate diagnostics. These diagnostics are useful in helping to track down problems in the powerline network.

See the command “residual” for a pre-packaged error rate test function.

“nid” is the 6-byte Neuron ID of the target device.

This command returns the number of packets received.

Example:

```
USB12> clr_status $nid
USB12> send_app $nid 100
USB12> set total_rcvd [ num_rcvd $nid ]
USB12> if { {$total_rcvd != 101 } {  # 101 for the 1 sent by num_rcvd
        puts “Lost [ expr 101 - $total_rcvd ] messages out of 100.”
}
USB12>
```

This example demonstrates a simple error rate test of a remote device specified by TCL variable $nid. First the status counters are cleared. Then 100 messages are sent to the device. The “num_rcvd” command is used to fetch the number of messages received by this device. Remember that the message used to retrieve the status will count also, so the total received will be 101 if no messages were lost. The TCL “if” statement examines the $total_rcvd variable which contains the output from “num_rcvd”, If the value isn’t 101, the number of messages lost is printed.

nv_cfg nid nvIndex

Read and display the raw configuration data for a network variable from a remote device. See Appendix B of the Neuron Toshiba Databook for the structure of the response data from this command. This command is typically used to determine the direction and the selector for a network variable. Nework variables are addressed by selectors for updates, and by index for fetches. The reason for this is documented in the brief introduction to LonWorks networking in this document.

“nid” is the 6-byte Neuron ID of the target device.

“nvIndex” is the index of the network variable configuration to fetch.

Example:

```
USB12> nv_cfg $nid 0
0x3f 0xff 0x0f
```

This example demonstrates fetching the configuration of the network variable at index 0 from the device specified by TCL variable $nid. The raw data indicates this is an input with selector 0x3FFF.

For more information on selectors, read the brief introduction to LonWorks in this document and the LonTalk 3.0 specification provided on the CD.

nv_fetch nid nvIndex [nvType]

Fetch the value of the network variable at the specified index. If nvType is specified, the return value is formatted for display. This command is used to poll the value of a network variable in the device. See the brief introduction to LonWorks in this document for more about network variables.

“nid” is the 6-byte Neuron ID of the target device.

“nvIndex” is the index of the network variable to fetch.

“nvType” is the format to apply to the fetch network variable value.

Current valid options for nvType are:

SNVT_switch
SNVT_freq_hz
SNVT_count
SNVT_amp_mil
SCPTmaxRcvTime

SNVT and SCPT are discussed in the brief introduction to LonWorks section of this document.

Example:

```
USB12> nv_fetch $nid 19 SCPTmaxRcvTime
60.0s
```

This example fetches the maximum receive timeout (network variable index 19) from the IVC24-04 specified by $nid. The maximum receive timeout determines how long an output remains active if an update isn't received. This is used to prevent flooding in the irrigation system by specifying that the nviOut1-nviOut4 network variable inputs must be updated every 60 seconds or the output controlled by those NVs will be shut off. This configuration property can be changed. The IVC24-04 factory default setting is 1 minute.

nv_update nid selector byte0 .. byten

Update an input network variable. This command is used to change an input network variable's value. Network variable inputs are used to control the outputs of the IVC24-04 device. When an update is received from the network, the device reacts by changing the output associated with that network variable. In the case of the IVC24-04, nviOut1-nviOut4 are network variable inputs mapped to output 1 through output 4.

"nid" is the 6-byte Neuron ID of the target device.

"selector" is the 16-bit selector value of the network variable to modify. This can be obtained by using the "nv_cfg" procedure.

"byte0" through "byten" represent the value in bytes used to set the network variable. SNVT_switch, for example, takes a value and a state, so two bytes are necessary to encode the value for the "nv_update" command.

Example:

```
USB12> nv_update $nid 0x3fff 1 1
```

This example sets the value of nvoOut1 to 1 and the state to 1 (0.5% and ON). Both "value" and "state" must be non-zero to turn on the output. For an IVC24-04, if another "nv_update" isn't received within 60 seconds (or the current SCPTmaxRcvTime value), the output will automatically turn off.

query_address nid index

Fetch an entry from the address table. The address table allows this device to implicitly address another device or group. Output network variables use the address table to communicate changes to a single node or a group of nodes.

“nid” is the 6-byte Neuron ID of the target device.

“index” selects one of 15 address table entries (0-14)
Example:

```
USB12> query_address $nid 0
0x00 0x00 0x00 0x00 0x00
```

This example reads address table entry 0. This entry is not initialized and is available for network binding. Bibaja’s irrigation network is master slave which means the computer responsible for scheduling irrigation periodically polls inputs and controls outputs. No network variables are bound in the current implementation. Code changes are being considered for the USB12-01 which would allow binding critical status outputs from devices on the network to the USB12-01 for immediate alarm notification (no polling required.) Implementing the network as master/slave simplifies the programming immensely.

query_domain nid domainIndex

Read the specified domain table entry. This command reads one of the two domain table entries (0 or 1) and reports the domain table data in hex format.

“nid” is the 6-byte Neuron ID of the target device.

“domainIndex” is either 0 or 1 to select one of the two domain table entries.

Typically only domain entry 0 is used in remote devices. The network interface device (USB12-01) uses domain entry 0 to match the domain of the network, and domain entry 1 as the zero-length network management domain. The zero-length domain is required to install and configure new nodes.

Example:

```
USB12> query_domain "0 0 0 0 0 0" 0
0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x81 0x00 0xff 0xff 0xff 0xff 0xff 0xff
USB12> query_domain "0 0 0 0 0 0" 1
0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x81 0x00 0xff 0xff 0xff 0xff 0xff 0xff
```

This example demonstrates query_domain used on the network interface device. The Neuron ID “0 0 0 0 0 0” is illegal and signals the software to perform the requested operation on the network interface device instead of going onto the network. Both domain entry 0 and 1 are the same in this example USB12-01 device. After decoding the domain table entry using the structure found in Appendix B of the Toshiba Neuron manual, we find the domain ID is set to zero length, the subnet is 1, the node is 1, and the 6-byte authentication key is all 0xFF.

query_id selector

Requests devices respond with their 6-byte Neuron ID and 8-byte Program ID information. This message may be used to find applicationless and unconfigured devices on the network. Typically applicationless devices will not make it out of the factory. All devices will be shipped in the unconfigured state. The unconfigured state, as the name implies, means the device does not have any addressing information assigned apart from the unique 6-byte Neuron ID which is assigned when the microcontroller is manufactured.

Customers installing new devices can therefore choose from several options:

1) Find unconfigured devices (Works provided no close neighbors have unconfigured devices)
2) Enter the Neuron ID from the barcode on the device
3) Press the service pin (magnetically activated button on the IVC24-04, pushbutton on IOPoint-EVB) which will send the Neuron ID and Program ID on the network

"selector" selects the type of device to locate. Choose from one of the following:

0 – Find all unconfigured or applicationless devices
1 – Find all selected devices
2 – Find all selected unconfigured devices

Example:

```
USB12> respond_to_query $nid 1
0x22
USB12> query_id 1
0x05 0x00 0x27 0xa7 0xbe 0x00 0x9f 0xfe 0x61 0x07 0x00 0x06 0x11 0x01
USB12> respond_to_query $nid 0
0x22
```

In this example, a device is selected to respond to the "query_id" command. The "query_id" command is issued asking for selected devices (selector=1). Our selected device responds with the 6-byte Neuron ID and the 8-byte Program ID. Finally the selection is cleared using the respond_to_query command again.

When doing device discovery, a more typical scenario would be:

1) Broadcast on the zero length domain a "respond_to_query" set message
2) Send "query_id" with the selector=2 (find unconfigured selected nodes)
3) If we received a response, record the response and clear the "selected" flag on that node, go to 2) until no response received

4) No response was received in 3, broadcast a “respond_to_query” clear message

At the end of this process, we should have a list of unconfigured nodes built in step 3. The user may choose to install nodes from this list.

query_status nid

Retrieve the status of a device on the network. This command is very useful for diagnostics and checking the health of a device. CRC errors on the network do not necessarily indicate problems with the network. Due to the implementation of the powerline algorithms, there will be a low level of false packets received by the device. These false packets are discarded and a CRC error is recorded. It is quite typical for a device to record anywhere from 1000 to 4000 CRC errors a week.

The reset cause records the cause of the last reset for a device. Unless a device has been reconfigured, told to reset, or had it’s status cleared, the typical reset cause will be power-up reset. Watchdog resets may indicate a device is malfunctioning. Self-induced software resets not caused by configuration changes may also include aberrant device behavior such as corrupted EEPROM.

The node state will typically be Configured and online. New nodes from the factory will show up as unconfigured and online. If a node somehow escaped final test in the factory or has corrupted itself, the state may be set the applicationless online meaning it has no application loaded.

The error log shows the last error logged. One error you may run into while tinkering with the device is invalid network variable index. This error indicates you tried to access a network variable index beyond the end of the network variable table. The current version of the IVC24-04 has 20 entries (0-19).

The “clr_status” procedure may be used to clear this information.

“nid” is the 6-byte Neuron ID of the target device.

Example:
```
     USB12> query_status “0 0 0 0 0 0”
     Query Status Passed
     CRC Errors:                 112
     Transaction Timeouts:       3
     Receive Transaction Full:   0
     Lost Messages:              0
     Missed Messages:            0
     Reset Cause:                Power-up reset
     Node State:                 Configured, online
     Version Number:             0x0e
     Error Log:                  No error
```

```
    Model Number:           0x0f
    0x31
```

This example shows how a typical "query_status" might look for the network interface (USB12-01). This device has had 112 CRC errors. Three (3) transactions have timed out. In this case, the timeouts were due to "query_id" messages that did not receive a response. The version number indicates this is Firmware Version 14 and the model number indicates this is an Echelon PL3120 Powerline Smart Transceiver (Neuron/Transceiver combination IC.)

query_xcvr_status nid

Retrieve the status from the powerline transceiver. This message is really not that useful without documentation. Unfortunately for customers, Echelon has chosen not to document what these values mean. Due to my contract with Echelon, I am not at liberty to disclose the meanings of these status registers.

"nid" is the 6-byte Neuron ID of the target device.

Example:

```
    USB12> query_xcvr_status $nid
    0x54 0x00 0xb5 0xfe 0xb6 0x00 0xb5
```

In this example we queried the transceiver status of a device and got back some very useful data. Unfortunately we have no idea what it means.

read_mem nid address count

Read memory from a device. This command is used to read up to 16 bytes at a time from a device on the network. Read data is returned in the standard TCL way and is easily used by other commands. If you know the location of variables in RAM, for example, the read memory command may be used to monitor the device state for debugging.

"nid" is the 6-byte Neuron ID of the target device.

"address" is the 16-bit address in memory to read from

"count" is the number of bytes, up to 16, to read from memory

Example:

```
    USB12> read_mem $nid 0xf000 16
    0x05 0x00 0x27 0xa7 0xbe 0x00 0x0f 0xf8 0xf6 0xc8 0x14 0xf7 0x39 0x9f 0xfe
0x61
```

This example shows how to read the first 16 bytes of the read-only data structure. The read-only data structure is documented in Appendix A of the Toshiba Neuron databook included on your CD.

read_mem_sn subnet node address count

Same as read_mem, but uses subnet/node addressing. This command is used exactly as read_mem, but provides an example of how to perform subnet/node addressing. All of the other procedures included in routine.tcl use Neuron ID addressing. Neuron ID addressing was chose for the procedures so we would not have to implement network management to assign and keep track of subnet/node numbering for the devices. Using subnet/node addressing saves at least 4 bytes in the packet, or just over 8ms of time in the transmission.

For the primary carrier, packet length on the network is calculated as follows:

1 bit = 182.4 microseconds
24 bits of preamble
11 bits of synchronization
11 bits per byte of packet data
22 bits for end of packet (2 11-bit EOP characters.)

“subnet” is the subnet of the target device

“node” is the node address of the target device

“address” is the 16-bit address to read from in device memory (see dump_mem for map)

“count” is the number of bytes, up to 16, to read from memory

Example:

```
USB12> read_mem_sn 2 5 0 1
0x0e
```

This example reads 1 byte of memory from address 0x0000 on a device with subnet/node address 2/5.

reset_device nid

Reset a device. This command is used to tell a remote device to issue a software reset and restart. One typical use is to force a device to reload configuration changes.

“nid” is the 6-byte Neuron ID of the target device.

Example:

```
USB12> reset_device $nid
```

This example resets the device who Neuron ID is stored in the TCL variable $nid.

residual nid count

Perform a residual error rate test on a target device. This diagnostic test uses the procedures clr_status, send_app, and num_rcvd to perform a simple communication test between the USB12-01 and a remote device such as an IOPoint-PL.

“nid” is the 6-byte Neuron ID of the target device.

“count” is the number of packets to send to the target device.

Example:

```
USB12> residual $nid 100
Rcvd: 101   Err Rate: 0.0%
101 0.0
```

Thjs example sends 100 packets to the target device. Then it sends the “num_rcvd” query to the device which accounts for packet 101. The error rate is calculated as (101-num_rcvd)/101. The results are printed to the console using “puts” and are returned as a TCL result list.

This command may also be used to measure performance. The following example demonstrates about how many packets the network interface can send per minute. Divide this number by a little more than 2 and you have the number of transactions that may be completed every minute. This performance number is useful to know when determining how much traffic you can schedule for datalogging and control every minute:

```
USB12> time { residual $nid 521 }
Rcvd: 522   Err Rate: 0.0%
59365684 microseconds per iteration
```

respond_to_query nid respond

Set or clear the “respond to query” flag inside of a device. This procedure will set or clear the flag in a device that is already known (since this message uses Neuron ID addressing.) Typical use of this procedure is to clear the “respond to query” flag in a device that was just discovered. This allows subsequent “query_id” to find a new device. During discovery, the software will repeat the “query_id”, “respond_to_query off” procedure until the query_id command does not receive a response from the network.

“nid” is the 6-byte Neuron ID of the target device.

“respond” is a Boolean flag, 1 means repond, 0 means do not respond.

Example:

```
    USB12> set myNid [ lrange [ query_id 2 ] 0 5 ]
    USB12> respond_to_query $myNid 0
```

This example demonstrates finding a selected unconfigured device (selector=2 for query_id command). The TCL variable $myNid is set to the list range (lrange) 0 to 5. This range is the Neuron ID in the respond from query_id. Note that this example assumes the response was successful. Once the value is stored in $myNid, the respond_to_query command is used to turn off the respond to query flag in device $myNid.

send_app nid count

Send an unacknowledged message to a device repeatedly to determine error rate statistics. This command is used by the “residual” command to perform the packet error rate testing. “send_app” builds a respond_to_query off command and configures the message service to be unacknowledged and the number of retries is zero. This gives each message one try to get to the remote device on the primary carrier frequency.

“nid” is the 6-byte Neuron ID of the target device.

“count” is the number of messages to send to the target device.

Example:

```
    USB12> clr_status $nid
    USB12> send_app $nid 100
    USB12> set myResult [ num_rcvd $nid ]
    USB12> set myError [ expr  ( 100 – ( $myResult – 1)) ]%
```

This example demonstrates using “clr_status”, “send_app”, and “num_rcvd” to implement a simple error rate test. This is essentially the same procedure used in the “residual” procedure. Status counters are cleared using “clr_status”. “send_app” sends 100 messages to the device with Neuron ID $nid. The number of messages is retrieved from the device using “num_rcvd”. Since 100 messages we sent, the error rate in percent is simply the number lost. The 1 message sent using “num_rcvd” is subtracted from $myResult to leave the total number of messages received by the target device. Subtracting this from 100 gives the number lost, which is the same as the %. error rate.

send_msg svc_type data_count msg_addr msg_code msg_data

Build and send a message on the network. This command may be used to build and send a LonTalk message using different service types and different addressing modes. Most of the pre-packaged message commands included in routine.tcl use Neuron ID addressing. New procedures may be developed that use “send_msg” to implement subnet node addressed versions of the procedures.

“svc_type” is one of:

- 0 – Acknowledged
- 1 – Unacknowledged repeated
- 2 – Unacknowledged
- 3 – Request/Response

“data_count” is the length of the “msg_data” field, can easily be found using “llength $msg_data”.

“msg_addr” is the 11 byte address structure. See basic introduction to LonWorks in this document for more about the address structure.

“msg_code” tells the receiving node what function to perform and how to interpret the msg_data.

“msg_data” is the data payload of the message and the format depends on the “msg_code”.

The “send_msg” procedure will return the response from the USB12-01 device as a TCL list of hexadecimal values.

Here is an example of the “send_msg” procedure embedded within the procedure “query_domain”. This may be found in the file “routine.tcl” included with your distribution:

```
proc query_domain { nid domain } {
   set svc_type 3
   set retry_count 3
   set msg_addr "0x02 0x80 0x9$retry_count 0x09 0x01 $nid"
   set msg_code "0x6a"
   set msg_data $domain
   set data_count [ llength $msg_data ]
   set myResult [ lrange [ send_msg $svc_type $data_count $msg_addr
$msg_code $msg_data ] 14 end ]
   if { [ lindex $myResult 0 ] != 0x2a } {
      return "query_domain failed"
   } else {
      return [ lrange $myResult 1 end ]
```

```
    }
}
```

set_mode nid mode [state]

Set the mode or state of the target device. This command is used during installation to take a device offline before making configuration changes. When a device is being reloaded, the device state will be altered to applicationless to prevent the Neuron microcontroller from interfering with the EEPROM during the network download.

The current operate mode may be determined using the "query_status" procedure.

"nid" is the 6-byte Neuron ID of the target device.

"mode" is one of offline, online, reset, or state

"state" is only required when "mode" is set to state. The valid states are unconfigured, applicationless, configured, and offline. The difference between "mode" offline and "state" offline is that the offline "state" is persistant across resets. "mode" offline is a soft offline and will revert to online after a reset.
Example:

```
USB12> query_status $nid
Query Status Passed
CRC Errors:               3
Transaction Timeouts:     0
Receive Transaction Full: 0
Lost Messages:            0
Missed Messages:          0
Reset Cause:              Software reset
Node State:               Configured, online
Version Number:           0x0e
Error Log:                No error
Model Number:             0x0f
0x31
USB12> set_mode $nid offline
Setting mode to 0
USB12> query_status $nid
Query Status Passed
CRC Errors:               3
Transaction Timeouts:     0
Receive Transaction Full: 0
Lost Messages:            0
Missed Messages:          0
Reset Cause:              Software reset
Node State:               Configured, soft offline
Version Number:           0x0e
Error Log:                No error
```

```
Model Number:             0x0f
0x31
USB12> reset_device $nid
0x10
USB12> query_status $nid
Query Status Passed
CRC Errors:               0
Transaction Timeouts:     0
Receive Transaction Full: 0
Lost Messages:            0
Missed Messages:          0
Reset Cause:              Software reset
Node State:               Configured, online
Version Number:           0x0e
Error Log:                No error
Model Number:             0x0f
0x31
```

This example examines the initial state of a device, changes the device mode to soft offline using “set_mode”, confirms the device is offline using query_status, resets the device to clear the offline mode, and confirms the device has gone online again using query_status.

update_address nid index type domain member rpt_timer retry_count rcv_timer tx_timer group_or_subnet

Update an address table entry. This command is used to configure an address table entry to use for binding network variables together or for configuring a device as part of a group of devices. Binding network variables allows peer-to-peer event driven traffic on the network. Typically the Bibaja system uses nv_fetch and nv_update to get and set values in devices across the network. We are considering adding network variable support in the USB12-01 adapter which will allow binding alarm outputs from all nodes into alarm inputs on the USB12-01 for immediate event driven notification of faults.

“nid” is the 6-byte Neuron ID of the target device.

“index” is the address table index.

“type” is the addressing type, either 0 for unbound, 1 for subnet/node, 3 for broadcast, or 0x80 |group size for group addressing.

“domain” is the domain entry to use when sending messages using this address entry, either 0 or 1.

“member” is either this device’s group member number for group addressing or the node address for subnet/node addressing.

“rpt_timer” specifies the delay between message repeats for unacknowledged repeated service.

“retry_count” specifies the number of retries for this address. Total number of tries is initial try plus retry_count, so a retry_count of 3 means 4 total tries.

“rcv_timer” is used in group addressing to determine the maximum interval between received messages to distinguish retries from new messages. See Appendix A page A-23 for a clearer description.

“tx_timer” determines the repeat delay when using acknowledged or request/response service.

“group_or_subnet” specifies the group number for group addressing. If subnet/node addressing is used, this byte is the subnet ID for this address table entry.

update_domain nid domain_index domain_id subnet node len [key]

Update one of the two domain table entries. This command is used to configure the domain, subnet, and node address of the target device. The network management device uses domain 0 to join the network domain, and domain 1 is used for network management (zero length domain). Devices on the network generally only use domain entry 0 to join the network domain.

“nid” is the 6-byte Neuron ID of the target device.

“domain_index” is either 0 or 1 to select one of the two domain table entries.

“domain_id” is a 6-byte domain value. Even if you use the zero length domain, “domain_id” should be set to something, typically “0 0 0 0 0 0”.

“ subnet” is the subnet number for this device.

“node” is the node address for this device.

“len” is the length of the domain_id. Valid lengths are 0, 1, 3, and 6.

“key” is an optional argument that specifies the 6-byte authentication key. Authentication prevents your neighbor from turning on your sprinklers. The default is all 0xFF.

Example:

```
USB12> update_domain "0 0 0 0 0 0" 1 "0 0 0 0 0 0" 2 1 0
USB12> query_domain "0 0 0 0 0 0" 1
0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x81 0x00 0xff 0xff 0xff 0xff 0xff 0xff
```

This example configured domain entry 1 of the USB12-01 to the zero-length domain. The subnet is chosen to be 2, and the node ID is 1. The Authentication key takes the default value of all 0xFF as shown in the query_domain response.

update_key nid index keyIncrement

Modify the authentication key of a device using the specified 6-byte key increment. This command provides a way to modify the authentication key using an increment. This prevents the authentication key from being broadcast across the network and possibly snooped. Changing the key in the USB12-01 device should always be done using update_domain since local network management commands do not travel across the network. A remote or secondary USB12-01 may be modified from the network using this command.

"nid" is the 6-byte Neuron ID of the target device.

"index" is the domain table index to modify.

"keyIncrement" is a 6 signed-byte increment to apply to the existing key. newKey = oldKey + keyIncrement (on a byte by byte basis.)

Example:

```
    USB12> query_domain $nid 0
    0x14 0x00 0x00 0x00 0x00 0x00 0x02 0x85 0x01 0xff 0xff 0xff 0xff 0xff 0xff
    USB12> update_key $nid 0 "0xff 0x01 0xff 0x01 0xff 0x01"
    0x25
    USB12> query_domain $nid 0
    0x14 0x00 0x00 0x00 0x00 0x00 0x02 0x85 0x01 0xfe 0x00 0xfe 0x00 0xfe 0x00
```

This example shows the initial value of the authentication key in domain table entry 0 is all 0xFF. The "update_key" function is used to increment and decrement alternating bytes in the authentication key by 1. The result is shown in the subsequent query_domain response.

write_mem nid address data0 [.. datan]

Write from 1 to 16 bytes of data into memory. This procedure may be used to update values in EEPROM.

CAUTION: Do not randomly write values to the EEPROM. This will corrupt the application and configuration areas and will result in a node that no longer functions properly. Writing to RAM may cause the node to crash if you hit just the right spot, but will likely not cause a persistent failure.

When writing EEPROM, the node should either be offline or in the applicationless state. This allows writing a range of EEPROM, updating the checksum, and then changing the node state back to configured. If EEPROM is written while the application is running, the device will likely detect the EEPROM checksum error and will log an error and go to the applicationless state.

If the RAM address of application variables are known, they may be remotely poked using the write_mem procedure for debugging purposes. Data may be directly peeked and poked from devices using read_mem and write_mem, but the preferred method for exchanging data between devices is through the use of network variables.

Example:

```
    USB12> write_mem $nid 0xE800 1
    0x2e
    USB12> after 1000
    USB12> write_mem $nid 0xE800 0
```

```
        0x2e
```

This example writes to the location “activate_service_led” (0xE800) in device RAM. If the device has a service LED (the IVC24-04 and USB12-01 do not, the IOPoint-EVB does), then it will illuminate for approximately 1 second. (after 1000 is a TCL command to wait for 1000ms.)

BRIEF INTRODUCTION TO LONWORKS

This section will give a brief introduction to the LonWorks network. For a more in-depth description of the LonWorks networking protocol, you must purchase the LonTalk Protocol Specification (EIA 709.1) available online from “http://global.ihs.com”.

Powerline Physical Layer

Bibaja’s products utilize Echelon’s power line communications physical layer. This is a 7V peak-to-peak communications signal. The signal consists of two carriers known as the primary and secondary carrier. The secondary is only used as a backup path if communication fails on the primary carrier when using acknowledged packets. If you are using unacknowledged repeated messages, the message will be repeated on alternating carrier frequencies to increase the chance of getting the message to the remote device.

The primary carrier is around 131.579kHz. The raw data rate is 1 bit every 182.4us or 5482.45614bps. Every data byte is encoded as an 11-bit word with two polarity bits and one parity bit. The packet is preceded by a 24-bit preamble and an 11-bit wordsync pattern. The packet is followed by two 11-bit end of packet patterns.

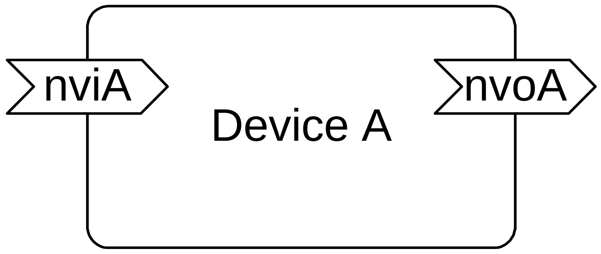
Typical LonTalk messages are about 13 bytes. The protocol includes a 16-bit CRC that is appended to the LonTalk message for error detection. Therefore, the total length of a typical packet on the powerline primary channel would be:

> 24-bit preamble
> 11-bit wordsync
> 11 * (13 byte packet + 2 byte crc) = 165 bits
> 11 * 2 EOP patterns = 22 bits
> Total: 222 bits
>
> 222 bits * 182.4us per bit = 40.4928ms

There is a minimum inter-packet gap as well as randomizing slots between packets that also contribute to overall network performance. It is safe to assume the gap between packets is about 15ms on average. This could be reduced by eliminating the priority slots on the standard channel. A network management tool can change the communications parameters during device commissioning to eliminate priority slots. Bibaja’s devices do not use priority messages.

Network Variables

Network variables provide a simple method for sharing information between devices on the network. To the programmer, network variables appear as standard C types and structures. Network variables are typically represented in diagrams as follows:



The main box represents a device or a functional block within a device. The arrow labeled “nviA” represents an input network variable. This variable is a network variable input (nvi) from the network and likely controls a physical output on the device. The arrow labeled “nvoA” is a network variable output (nvo) from the device to the network and likely comes from a physical input on the device. The tag “nvi” is a programming convention used to identify network variable inputs. Likewise, the tag “nvo” is a programming convention to identify network variable outputs. Most devices follow this programming convention when naming network variables.

Network variable outputs may be monitored by polling or may propagate on an event driven basis through bound connections. If a network variable output is not bound, it will not send any information on the network.

Polling is used in a master/slave system such as Bibaja’s. Device outputs are polled periodically and either stored to a datalog or acted upon when read. One disadvantage to polling is the delay between when a fault condition occurs and when it is detected. Network variable outputs that indicate critical faults, such as a break in a pipe or an overcurrent condition in a valve solenoid, are monitored frequently to minimize the time between detection and corrective action.

Bound connections allow network variable inputs to be updated as soon as a network variable output changes. There are several advantages to peer to peer binding in LonWorks networks. One advantage is a fault condition may be detected and acted upon by the network without the need for a polling master to fetch the fault, make a decision, and act upon it. In the case of a leak, the flow meter could be bound to the master valve control. As soon as the meter detected flow and no irrigation cycle was in progress, the master valve could be shut down. This would allow the network to self regulate and respond to fault conditions in the fastest way possible. One disadvantage is the network management tool then needs to keep a list of bindings between nodes and present then to the user in an easy to understand fashion. Users must also be aware of how to bind the fault sensors to the actuators in a way that will allow for autonomous peer-to-peer control. This means the end user must be able to do a bit of distributed programming using logic and intelligent bindings in the network. Unfortunately distributed programming is difficult, so most LonWorks networks revert to a hierarchical control structure similar to polling.

Bibaja chose polling by a central master to simplify the system. Flow sensors, for example, are associated with the master valve in the irrigation system when the user installs them. The user does not have to make any clever bindings to turn off the master valve if flow is detected when the irrigation is not running. The central master knows when irrigation cycles are running and when they are not running. If the central master detects flow and an irrigation cycle is not in progress, it will turn off the master valve and alert the maintenance staff or the homeowner that there is trouble.

Network Variable Index

The network variable index is a static reference to a network variable in a node. The index was assigned at application compile time based on the order the network variable was declared in the Neuron C source code. The network variable index may be used to fetch the configuration information for the network variable. The index may also be used to fetch the current value.

Part of the configuration is the network variable selector. The selector must be used to update an inputs network variable. The index is used to fetch the value of either an input or an output across the network.

Network Variable Selector

The network variable selector is a number from 0 to 0x3FFF. To update an input network variable, you must use the selector. You may be asking yourself why not simply used the index of the variable? Why make this so complicated? Well, here’s why.

Let’s say you have one light switch and you want to control 4 light controllers in a big conference room. This LonWorks enabled light switch has a network variable output, nvoSwitch. The 4 light controllers are from different manufacturers and have inputs named nviLight, nviSpotLight, nviFlambee, and nviHeatLamp. Since different manufacturers developed these devices, the network variable index, assigned at compile time, will likely be different. However, the selector is programmable so you can use one handy number to address different network variables in different devices.

So, to bind these together, we would modify the selector on these devices to be a common unique selector among this group of nodes. This same selector value would be written to the NV configuration table entry of the light switch and into the NV configuration of the 4 different light controllers. Note also that we would be using group addressing and probably would use unacknowledged repeated service to reduce traffic on the network.

If we were forced to use the network variable index, we would have been required to construct 4 different messages from the light switch to control the 4 different light controllers. It would have consumed 4 address table entries on the light switch. Address table entries consume EEPROM which is a limited resource in the Neuron microcontroller. We would also not have the option of reducing traffic using group addressing since a unique index is required for each update.

Selectors offer value in a polled system like Bibaja’s as well. At some point in the future, we may manufacture devices that need to have accurate time. The network master using the USB12-01 could broadcast the time using group addressing to a network variable input on all nodes subscribed to the time service. We would modify the selector to match the time service group on all subscribers. Then a single unacknowledged repeated message to the time service group is all that is required to keep the devices in sync.

NVs vs. Memory Mapped Variables

Another question that might arise is why use NVs when you can simply read and write data directly between the memory of network devices. Poking and peeking data directly in memory is similar to the MODBUS networking scheme. While it has been shown to work, there are a few reasons why Bibaja chose not to use memory-mapped variables.

Pros of NVs

- Very well supported on the LonWorks network
- Easy discovery of available inputs and outputs on a device
- Discovery of input or output type possible
- Code to support NVs is embedded in the firmware in ROM, consuming no additional code space in EEPROM
- Index won't change as long as order in code is retained

Cons of NVs

- Limited to 63 NVs per Neuron based device
- Dealing with selectors and binding and address table entries (if needed)
- Large complex types aren't easy to implement

Pros of Memory Mapped

- Large complex types possible
- Limited only to amount of memory space available (Can have greater than 63 variables)
- No selectors

Cons of Memory Mapped

- No simple way to propagate changes between devices (requires code support in the application to form a memory write)
- Requires additional code support in some instances
- No way to control the allocated addresses in the compiler (address may change from code release to another)
- Cannot discover inputs and outputs without external documentation
- Cannot discover types without external documentation

Given these options, it seemed that network variables offered the most robust and stable way to exchange information between devices on the network.

Standard Network Variable Types (SNVT)

Standard Network Variable Types, or SNVT (snivvet) for short, are used to encourage interoperability between devices on the network. Take the light controller example again. The nvoSwitch output is of the type SNVT_switch. The same is true for all of the nvi's on the light controllers. This means the nvo may be bound to the nvi's because they are a compatible type. There is no need to translate the information. All of these devices speak the same language, in this case, SNVT_switch. For more documentation on SNVT's and standard types, see http://types.lonmark.org/.

The declaration for SNVT_switch is:

```
typedef struct {
        unsigned short value;
        unsigned short state;
} SNVT_switch;
```

Value is a number from 0 to 200 which represents the output from 0 to 100% in steps of 0.5%. State is either 0 or 1 for OFF or ON respectively.

You will see there are a number of SNVTs to choose from. If you cannot find what you need, however, don't fret. You may create your own types (called UNVTs). You just have to be certain to document the structure or data format for your user defined network variable type so customers know how to use it.

Standard Configuration Property Types (SCPT)

Standard Configuration Property types, or SCPTs (skippets), are used to determine configuration properties that affect the way a node behaves.

Configuration properties may be implemented as non-volatile network variables, as we have implemented them in the IVC24-04's nciMaxRcvTime. They may also be implemented as files of configuration properties managed using the file transfer protocol. The advantage to FTP is the configuration properties do not consume network variable space. The disadvantage is that you must implement the file transfer protocol in the limited code space in EEPROM.

Since the IVC24-04 isn't limited by network variable space, we chose to implement the configuration properties as network variables.

Now, what does a configuration property do? In the case of nciMaxRcvTime, this configuration property determines how long to wait between updates if network variable inputs before reverting to the default state. In this case, nciMaxRcvTime applies to the 4 network variable inputs nviOut1, nviOut2, nviOut3, and nviOut4. These 4 inputs from the network control the 24VAC valve outputs on the IVC24-04 devices. If, for some reason, the master controller does not continue to tell the output to remain on, it will automatically turn off. This is a failsafe to prevent an output from sticking on in a bad state if the main controller goes offline. This is also called a heartbeat in the LonWorks network. Heartbeats are simply keep-alive timers. If you don't update the network variable within a certain period of time, it will revert to a safe state.

There are many other configuration property types. These are standardized so people will know what they mean simply by looking at them.

Address Types in LonTalk

There are 4 address types used in LonTalk:

- Neuron ID
- Subnet/Node
- Group
- Broadcast

Neuron ID addressing is typically used for installation and commissioning of a new device. New devices always have a 6-byte Neuron ID that may be retrieved using the service pin of a device, through the query ID message, or by a printed barcode or number on a device. This unique ID allows a network management device to communicate with and configure new devices.

Subnet/Node addressing is a simple two-byte address format. This addressing mode is commonly used in configured networks to reduce the size of the packet. Neuron ID addressing consumes at least 6 bytes vs. 2 bytes for subnet/node addressing.

Group addressing is used to allow one node to communicate with a small to large group of nodes. It provides an efficient means of sharing information between a group of devices instead of addressing each device individually or broadcasting the message to the entire network and consuming every device's resources to determine if they want the update or not.

Broadcast messages are sent subnet or domain wide and are used for things like service pin messages, new device discovery, and perhaps sharing things like time stamps across the entire network.

msg_addr in the USB12-01

The format of the msg_addr is a union of the 4 possible address types listed above. In addition to basic addressing information, the address structure of the Neuron chip also includes transaction timers, retry timers and retry count information.   The following typedef shows the union for msg_addr:

```
typedef union msg_addr {
        group_struct;
        snode_struct;
        nrnid_struct;
```

    bcast_struct;
} msg_addr;

These structures will be listed in the following 4 sections.

group_struct

The following typedef represents the group addressing format:

```
typedef struct group_struct {
        unsigned type     : 1;  // 1 => group
        unsigned size     : 7;  // group size (0 => huge group)
        unsigned domain   : 1;  // domain index
        unsigned member   : 7;  // member num (if huge group, only use
0)
        unsigned rpt_timer: 4;  // unackd_rpt timer
        unsigned retry    : 4;  // retry count
        unsigned rcv_timer: 4;  // receive timer index
        unsigned tx_timer : 4;  // transmit timer index
        unsigned group;         // group ID
} group_struct;
```

For details of the encoding of the timers, see Appendix A of the Toshiba Neuron databook included on the CD.

Bits in the Neuron microcontroller are packed from the most significant bit down to least significant, so the group type is equal to (0x80 | size) where size is 0x00 to 0x7F.

Bytes 6-11 are not used for the group_struct and should be 0.

snode_struct

The following typedef shows the subnet/node addressing format:

```
typedef struct snode_struct {
        addr_type type;         // SUBNET_NODE == 1
        unsigned domain   : 1;  // domain index
        unsigned node     : 7;  // node number
        unsigned rpt_timer: 4;  // unackd_rpt timer
        unsigned retry    : 4;  // retry count
        unsigned          : 4;
        unsigned tx_timer : 4;  // transmit timer index
        unsigned subnet;        // subnet ID
} snode_struct;
```

For timer encoding details, see Appendix A of the Toshiba Neuron databook included on the CD.

Bits in the Neuron are packed most significant bit down to least signficant, so the domain/node byte is equal to ((domain * 0x80) | node).

Bytes 6-11 are not used for the snode_struct and should be 0.

nrnid_struct

The following typedef shows the Neuron ID addressing format:

```
typedef struct nrnid_struct {
        addr_type       type;
        unsigned        domain    : 1;
        unsigned                  : 7;
        unsigned        rpt_timer : 4;
        unsigned        retry     : 4;
        unsigned                  : 4;
        unsigned        tx_timer  : 4;
        unsigned        subnet;
        unsigned        nid [NEURON_ID_LEN];
} nrnid_struct;
```

Bits in the Neuron are packed from most significant bit down to least significant, therefore the domain bit above will either be 0x80 or 0x00 (domain * 0x80). Blank fields above are reserved and should be left 0x00.

See Appendix A of the Toshiba Neuron databook included on the CD for timer field encoding.

This structure consumes all 11 bytes of the msg_addr union.

bcast_struct

The following typedef shows the broadcast addressing format:

```
typedef struct bcast_struct {
        addr_type type;         // BROADCAST
        unsigned domain   : 1;  // domain index
        unsigned          : 1;
        unsigned backlog  : 6;  // backlog override value
        unsigned rpt_timer: 4;  // unackd_rpt timer
        unsigned retry    : 4;  // retry count
        unsigned          : 4;
        unsigned tx_timer : 4;  // transmit timer index
        unsigned subnet;        // subnet ID (0 => domain broadcast)
} bcast_struct;
```

Bits in the Neuron are packed from most significant down to least significant, therefore the domain bit above will either be 0x80 or 0x00. Byte 2 will therefore be equal to (domain * 0x80) | backlog. Byte 3 will be (rpt_timer * 0x10 | retry).

See Appendix A of the Toshiba Neuron databook included on the CD for timer field encoding.

Bytes 6-11 are not used for the bcast_struct and should be 0.