Verix EOS Volume II

Communications Manual

VeriFone Part Number DOC00302, Revision A.4

Verix EOS Volume II Communications Manual

The information contained in this document is subject to change without notice. Although VeriFone has attempted to ensure the accuracy of the contents of this document, this document may include errors or omissions. The examples and sample programs are for illustration only and may not be suited for your purpose. You should verify the applicability of any example or sample program before placing the software into productive use. This document, including without limitation the examples and software programs, is supplied “As-Is.”

Comments? Please e-mail all comments on this document to your local VeriFone Support Team.

Acknowledgments

VeriFone, Inc.
2099 Gateway Place, Suite 600

San Jose, CA, 95110 USA.

1-800-VeriFone

www.verifone.com

VeriFone Part Number DOC00302, Revision A.4

CONTENTS

This communications manual supports the Development Toolkit (DTK) for the Verix EOS Volume II transaction terminals. This manual:

- Describes the programming tools for the communications environment,
- Provides descriptions of the CONFIG.SYS variables,
- Provides API descriptions and code examples,
- Provides discussion on system and communication devices,
- Provides descriptions of the security features,
- Describes working with the IPP (internal PIN pad), and
- Provides information on downloading applications into a Verix EOS Volume II terminal.

The Verix EOS Volume II Solutions terminals are designed to support many types of applications, especially in the point-of-sale environment. Applications are written in the C programming language and run in conjunction with the Verix EOS Volume II operating system. This manual is designed to help programmers develop those applications.

This manual also contains explicit information regarding the Application Programming Interface (API) with the Verix EOS Volume II operating system, and with optional peripheral or internal devices.

NOTE

Although this manual contains some operating instructions, please refer to the reference manual for your transaction terminal for complete operating instructions.

Organization

This document is organized as follows:

Chapter 1, TCIP/IP Stack - discusses the settings for the TCP/IP Stack.

Chapter 2, Secure Sockets Layer (SSL) - discusses the usage of Secure Sockets Layer (SSL).

Chapter 3, Network Interface - discusses network interface API.

Chapter 4, Configuration Management - discusses the standardized configuration file format to use across all Verix EOS Volume II components requiring external configuration/customization.

Chapter 5, CommEngine Interface API (CEIF.lib) - discusses the usage of CommEngine Interface API (CEIF.lib).

Chapter 6, Verix EOS Volume II Communication Engine Application (VXCE.out) - discusses the features and functionility of the Verix EOS Volume II Communication Engine Application (VXCE.out).

Chapter 7, Network Control Panel (NCP) - discusses the functionilties of the Network Control Panel (NCP) of Verix EOS Volume II .

Appendix A, External Parameters via CONFIG.sys - summarizes all CONFIG.sys parameters listed throughout the document.

Target Audience

This document is of interest to Application developers creating applications for use on Verix EOS Volume II -based terminals.

Assumptions About the Reader

It is assumed that the reader:

- Understands C programming concepts and terminology.
- Has access to a PC running Windows 2000 or Windows XP.
- Has installed the VVDTK on this machine.
- Has access to Verix EOS Volume II Solutions development terminal.

Conventions and Acronyms

The following conventions assist the reader to distinguish between different kinds of information.

- The `courier` typeface is used for code entries, filenames and extensions, and anything that requires typing at the DOS prompt or from the terminal keypad.
- The *italic* typeface indicates book title or emphasis.
- Text in blue indicates terms that are cross-referenced. When the pointer is placed over these references the pointer changes to the finger pointer, indicating a link. Click on the link to view the topic.

NOTE

Notes point out interesting and useful information.

CAUTION

Cautions point out potential programming problems.

The various conventions used throughout this manual are listed in Table 1.

Table 1 Conventions

Abbreviation	Definition
A	ampere
b	binary
bps	bits per second
dB	decibel

Table 1 Conventions

Abbreviation	Definition
dBm	decibel meter
h	hexadecimal
hr	hours
KB	kilobytes
kbps	kilobits per second
kHz	kilohertz
mA	milliampere
MAX	maximum (value)
MB	megabytes
MHz	megahertz
min	minutes
MIN	minimum (value)
ms	milliseconds
pps	pulse per second
Rx	Receive
s	seconds
Tx	Transmit
V	volts

Acronyms

The acronyms used in this manual are listed in Table 2.

Table 2 Acronyms

Acronym	Definition
ABNF	Augmented Backus-Naur Format
ACK	Acknowledge
ANSI	American National Standards Institute
APDU	Application Protocol Data Units
API	Application Program Interface
APN	Access Point Name
ARP	Address Resolution Protocol
ASCII	American Standard Code For Information Interchange
APACS	Association For Payment Clearing Services: Standards Setting Committee; A Member Of The European Committee For Banking Standards (Ecbs)
ATR	Answer To Reset
BCD	Binary Coded Decimal
BIOS	Basic Input Output System
BOOTP	Bootstrap Protocol UDP User Datagram Protocol
BRK	Break
BWT	Block Waiting Time
CDC	Communications Device Class

Table 2 Acronyms (continued)

Acronym	Definition
CDMA	Code Division Multiple Access
ceAPI	CommEngine Interface API
CEDM	CommEngine Device Management
CHAP	Challenge-Handshake Authentication Protocol
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
CRLF	Carriage Return Line Feed
CTS	Clear to Send
CVLR	Compressed Variable-length Record
CWT	Character Waiting Time
DDI	Device Driver Interface
DDL	Direct Download Utility
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamic Link Library
DNS	Domain Name System
DSR	Data Send Ready
DTK	Development Toolkit. See *Vvdtk*.
DTR	Data Terminal Ready
DUKPT	Derived Unique Key Per Transaction
EBS	European Banking Standard
EVDO	Evolution-Data Optimized
EEPROM	Electrically erasable programmable read-only memory
EMV	Europay Mastercard and Visa
EOF	End Of File
EOS	
EPP	External Pin Pad
FIFO	First In, First Out
FQDN	Fully Qualified Domain Name
GPRS	General packet radio service
HID	Human Interface Device
ICC	Integrated Circuit Card; Smart Card
IEEE	Institute Of Electrical And Electronics Engineers
IFD	Smart Card Interface Device
IFSC	Information Field Size Card
IFSD	Information Field Size Reader
IGMP	Internet Group Management Protocol
ILV	Identifier Length Value
ICMP	Internet Control Message Protocol
I/O	Input/Output
IPCP	Internet Protocol Control Protocol

Table 2 Acronyms (continued)

Acronym	Definition
IPP	Internal Pin Pad
ISR	Interrupt Service Routine
LAN	Local Area Network
LCD	Liquid Crystal Display
LCP	Link Control Protocol
LQM	
LRC	Longitudinal Redundancy Check
LQR	Link Quality Report
MAC	Message Authentication Code
MCU	Microcontroller
MDB	Multi Drop Bus
MIB	Management Information Block
MMU	Memory Management Unit
MPLA	Modem Profile Loading Application
MRU	
MSAM	Micromodule-Size Security Access Module
MSO	MorpoSmart™
MSR	Magnetic Stripe Reader
MTU	Maximum Transmission Unit
MUX	Multiplexor
NAK	No Acknowledgment
NI	Network Interface
NMI	Nonmaskable Interrupt
OS	Operating System
OTP	One-Time Password
PAP	Password Authentication Protocol
PCAP	
PCI PED	Payment Card Industry PIN Entry Devices
PED	PIN Entry Devices
PIN	Personal Identification Number
PKCS	Public Key Cryptography Standards
POS	Point-of-Sale
PPP	Point-to-Point Protocol
PSCR	Primary Smart Card Reader
PTID	Permanent Terminal Identification Number
PTS	Protocol Type Selection
RAM	Random Access Memory
RFC	
RFID	Radio Frequency Identification
RFU	Reserved for Future Use

Table 2 Acronyms (continued)

Acronym	Definition
ROM	Read-only Memory
RTC	Real-Time Clock
RTTTL	Ring Tone Text Transfer Language
RTS	Request To Send
SAM	Security Access Module
SCC	Serial Communication Control
SCC buffer	Storage Connecting Circuit Buffer
SCR	Swipe Card Reader
SDK	Software Development Kit
SDIO	Secure Digital Input/Output
SDLC	Synchronous Data Link Control
SLIP	Serial Line Internet Protocol
SMS	Small Message Service
SRAM	Static Random-access Memory
TCB	Task Control Block
TCP/IP	Transmission Control Protocol/Internet Protocol
TLV	Tag, Length, and Value
TTL	
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UPT	Unattended Payment Terminal
USB	Universal Serial Bus
VJ	Van Jacobson Header Compression
VLR	Variable-length Record
VPN	VeriFone Part Number
VSS	VeriShield Secure Script
VVDTK	Verix V Development Toolkit
WiFi	Wireless Fidelity
WTX	Workstation Technology Extended
WWAN	Wireless Wide Area Network

Related Documentation

To learn more about the Verix EOS Volume II Solutions, refer to the following set of documents:

- Multi-App Conductor for Verix V Programmers Guide, VPN - DOC00306.
- Getting Started with the Verix EOS Developers Suite, VPN - DOC00307.
- Verix V Communications Server Instantiation Guide, VPN - DOC00308.
- Verix EOS Volume I: Operating System Communications Guide, VPN - DOC00301.

- Verix EOS Volume III: Operating System Programming Tools Reference Guide, VPN - DOC00304.
- Verix EOS Porting Guide, VPN - DOC00305.
- V^{x}520 Software Engineering Requirement Specification, VPN - SPC252-001-01-A.
- V^{x}680 Software Engineering Requirement Specification, VPN - SPC268-004-01-A.
- Vx Extended OS Architecture Requirements & Description, VPN 28779.
- Application VCCESA.OUT, Engineering Requirements Specification, VPN 28810.
- Verix V Operating System Programmers Manual, VDN 23230.
- Vx Extended OS Architecture Requirements & Description, VPN 28779.
- VxEOS App VxEOS.OUT Engineering Requirements Specification, VPN 28780.
- VxEOS CommEngine Interface Library (CEIF.LIB), VPN 28781.
- VxEOS VMAC Compliant CommEngine Complaint Application (VCCESA.OUT), VPN 28810.
- VxEOS Configuration Management FRD, VPN xxxxx, (WIP).
- VxEOS Network Control Panel (VxEOS) FRD, VPN 28850.
- DDI Driver ERS, VPN xxxxx, Dated May 14, 2009
- CommEngine DDI Integration Guide.
- VxEOS Network ERS, VPN 28783.
- Vx Extended OS Architecture Requirements & Description, VPN 8779.
- Lib ceAPI.LIB ERS, VPN 28781.
- App VXCE.OUT FRD, VPN 28809.
- App VMACIF.OUT ERS, VPN 28810.
- DDI Driver ERS Version 09.
- Configuration Management FRD.
- OSDL FRD Version XX.
- HWID.lib FRD Version XX.
- Network Security with OpenSSL
- Cryptography for Secure Communications

  By John Viega, Matt Messier, Pravir Chandra

Detailed operating information can be found in the reference manual for your terminal. For equipment connection information refer to the reference manual or installation guides.

TCIP/IP Stack

This chapter discusses the settings for the TCP/IP Stack.

Interface

The first two items will be part of the Verix V SDK.

Parameter	Description
`svc_net.h`	Eventual home will be vrxsdk\include so "`#include <svc_net.h>`" will work
`svc_net.o`	Eventual home will be `vrxsdk\lib`.
`ipstack.bin`	To be include with Verix V EOS.

Getting Started

The first step is the network device must be opened then handed over to the stack. Once this has been done, applications may start using sockets. The code sequences below do not include any error checking to keep the code short and easy to understand. Production code should include error checking.

Minimal Ethernet Kickstart

Use the following sample code for Ethernet.

After using and modifying the sample code, you can have this or other tasks start using sockets.

Minimal COM1 Kickstart

Use the following minimal COM1 sample code.

Simple Socket Application

Simple TCP application sample code.

Simple UDP applicatopn sample code.

Socket Functions

Use the following socket functions:

- socket()
- bind()
- listen()
- accept()
- connect()
- getpeername()
- getsockname()
- setsockopt()
- getsockopt()
- recv()
- send()
- recvfrom()
- sendto()
- shutdown()
- socketclose()
- socketerrno()
- select()
- socketset_owner()
- socketioctl()
- DnsGetHostByName()
- gethostbyname()
- blockingIO()
- inet_addr()

socket()

Socket creates an endpoint for communication and returns a descriptor. The family parameter specifies a communications domain in which communication will take place; this selects the protocol family that should be used. The protocol family is generally the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `<svc_net.h>`. If protocol has been specified, but no exact match for the tuplet family, type, and protocol is found, then the first entry containing the specified family and type with zero for protocol will be used. The currently understood format is `PF_INET for ARPA` Internet protocols. The socket has the indicated type, which specifies the communication semantics.

Currently defined types are:

- `SOCK_STREAM`
- `SOCK_DGRAM`
- `SOCK_RAW`

A `SOCK_STREAM` type provides sequenced, reliable, two-way connection-based byte streams. An outof- band data transmission mechanism is supported. A `SOCK_DGRAM` socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length); a `SOCK_DGRAM` user is required to read an entire packet with each recv call or variation of recv call, otherwise an error code of EMSGSIZE is returned. protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case, a particular protocol must be specified in this manner.

The protocol number to use is particular to the "communication domain" in which communication is to take place. If the caller specifies a protocol, then it will be packaged into a socket level option request and sent to the underlying protocol layers. Sockets of type `SOCK_STREAM` are full-duplex byte streams. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with connect on the client side. On the server side, the server must call listen and then accept. Once connected, data may be transferred using recv and send calls or some variant of the send and recv calls. When a session has been completed, a close of the socket should be performed. The communications protocols used to implement a `SOCK_STREAM` ensure that data is not lost or duplicated.

If a piece of data (for which the peer protocol has buffer space) cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with (-1) return value and with `ETIMEDOUT` as the specific socket error. The TCP protocols optionally keep sockets "warm" by forcing transmissions roughly every two hours in the absence of other activity. An error is then indicated if no response can be elicited on an

otherwise idle connection for an extended period (for instance 5 minutes). SOCK_DGRAM or SOCK_RAW sockets allow datagrams to be sent to correspondents named in sendto calls. Datagrams are generally received with recvfrom which returns the next datagram with its return address. The operation of sockets is controlled by socket level options. These options are defined in the file <svc_net.h>. setsockopt and getsockopt are used to set and get options, respectively.

Prototype

```
int socket (int family, int type, int protocol);
```

Parameters

Parameter	Description
family	The protocol family to use for this socket (currently only PF_INET is used).
type	The type of socket.
protocol	The layer 4 protocol to use for this socket.

Family	Type
Protocol	Actual protocol
PF_INET	SOCK_DGRAM
IPPROTO_UDP	UDP
PF_INET	SOCK_STREAM
IPPROTO_TCP	TCP
PF_INET	SOCK_RAW
IPPROTO_ICMP	ICMP
PF_INET	SOCK_RAW
IPRPTOTO_IGMP	IGMP

Return Values

New Socket Descriptor or -1 on error. If an error occured, the socket error can be retrieved by calling socketerrno and using SOCKET_ERROR as the socket descriptor parameter.

The socket will fail if:

Parameter	Description
EMFILE	No more sockets are available
ENOBUFS	There was insufficient user memory available to complete the operation
EPROTONOSUPPORT	The specified protocol is not supported within this family
EPFNOSUPPORT	The Protocol family is not supported.

bind()

Bind assigns an address to an unnamed socket. When a socket is created with socket, it exists in an address family space but has no address assigned. bind requests that the address pointed to by addressPtr be assigned to the socket. Clients do not normally require that an address be assigned to a socket. However, servers usually require that the socket be bound to a "well known" address. The port number must be less than 32768 (`SOC_NO_INDEX`), or could be 0xFFFF (`WILD_PORT`). Binding to the `WILD_PORT` port number allows a server to listen for incoming connection requests on all the ports. Multiple sockets cannot bind to the same port with different IP addresses (as might be allowed in UNIX).

Prototype

```
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor to assign an IP address and port number to.
`addressPtr`	The pointer to the structure containing the address to assign.
`addressLength`	The length of the address structure.

Return Values

Value	Description
0	Success
-1	An error occurred

Bind can fail for any of the following reasons:

Value	Description
`EADDRINUSE`	The specified address is already in use.
`EBADF`	sockfd is not a valid descriptor.
`EINVAL`	One of the passed parameters is invalid or socket is already bound.

listen()

To accept connections, a socket is first created with socket a backlog for incoming connections is specified with listen and then the connections are accepted with accept. The listen call applies only to sockets of type `SOCK_STREAM`. The backLog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, and the underlying protocol supports retransmission, the connection request may be ignored so that retries may succeed. For `AF_INET` sockets, the TCP will retry the connection. If the backlog is not cleared by the time the TCP times out, connect will fail with `ETIMEDOUT`.

Prototype

```
int listen (int sockfd, int backlog);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor to listen on.
`backlog`	The maximum number of outstanding connections allowed on the socket.

Return Values

Value	Description
0	Success
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

Listen can fail for the following reason:

errno	Description
`EADDRINUSE`	The address is currently used by another socket.
`EBADF`	The socket descriptor is invalid.
`EOPNOTSUPP`	The socket is not of a type that supports the operation listen.

accept()

The argument sockfd is a socket that has been created with socket, bound to an address with bind, and that is listening for connections after a call to listen. accept extracts the first connection on the queue of pending connections, creates a new socket with the properties of sockfd, and allocates a new socket descriptor for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, accept returns an error as described below. The accepted socket is used to send and recv data to and from the socket that it is connected to. It is not used to accept more connections. The original socket remains open for accepting further connections. accept is used with connection-based socket types, currently with SOCK_STREAM. Using select (prior to calling accept):

Prototype

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Parameters

Parameter	Description
sockfd	The socket descriptor that was created with socket and bound to with bind and is listening for connections with listen.
addressPtr	The structure to write the incoming address into. If addressPtr and addressLengthPtr are equal to NULL, then no information about the remote address of the accepted socket is returned.
addressLengthPtr	Initially, it contains the amount of space pointed to by addressPtr. On return it contains the length in bytes of the address returned.

Returns

Value	Description
>0	New socket descriptor.
-1	Error

If accept fails, errno will be set to one of the following values:

errno	Description
EBADF	The socket descriptor is invalid.
EINVAL	addressPtr was a null pointer.
EINVAL	addressLengthPtr was a null pointer.
EINVAL	The value of addressLengthPtr was too small.
EPERM	Cannot call accept without calling listen first.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

connect()

The parameter sockfd is a socket. If it is of type SOCK_DGRAM, connect specifies the peer with which the socket is to be associated; this address is the address to which datagrams are to be sent if a receiver is not explicitly designated; it is the only address from which datagrams are to be received. If the socket sockfd is of type SOCK_STREAM, connect attempts to make a connection to another socket (either local or remote). The other socket is specified by addressPtr. addressPtr is a pointer to the IP address and port number of the remote or local socket. If sockfd is not bound, then it will be bound to an address selected by the underlying transport provider. Generally, stream sockets may successfully connect only once; datagram sockets may use connect multiple times to change their association. Datagram sockets may dissolve the association by connecting to a null address. Note that a non-blocking connect is allowed. In this case, if the connection has not been established, and not failed, connect will return SOCKET_ERROR, and socketerrno will return EINPROGRESS error code indicating that the connection is pending. connect should never be called more than once. Additional calls to connect will fail with EALREADY error code returned by socketerrno.

Prototype

```
int connect (int sockfd, const struct sockaddr *servaddr, socklen_t
addrlen);
```

Non-blocking connect and select

After issuing one non-blocking connect, the user can call select with the write mask set for that socket descriptor to check for connection completion. When select returns with the write mask set for that socket, the user can call getsockopt with the SO_ERROR option name. If the retrieved pending socket error is ENOERROR, then the connection has been established, otherwise an error occurred on the connection, as indicated by the retrieved pending socket error.

Non-blocking connect and polling

Alternatively, after the user issues a non-blocking connect call that returns SOCKET_ERROR, the user can poll for completion, by calling socketerrno until socketerrno no longer returns EINPROGRESS.

If connect fails, the socket is no longer usable, and must be closed. connect cannot be called again on the socket.

Parameters

Parameter	Description
sockfd	The socket descriptor to assign a name (port number) to.
addressPtr	The pointer to the structure containing the address to connect to for TCP. For UDP it is the default address to send to and the only address to receive from.
addressLength	The length of the address structure.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

Connect can fail for any of the following reasons:

errno	Description
EADDRINUSE	The socket address is already in use.
EADDRNOTAVAIL	The specified address is not available on the remote / local machine.
EPFNOSUPPORT	Addresses in the specified address family cannot be used with this socket
EINPROGRESS	The socket is non-blocking and the current connection attempt has not yet been completed.
EALREADY	Connect has already been called on the socket. Only one connect call is allowed on a socket.
EBADF	sockfd is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close the socket descriptor, and issue another socket call to obtain a new descriptor before attempting another connect call.
EPERM	Cannot call connect after listen call.
EINVAL	One of the parameters is invalid
EHOSTUNREACH	No route to the host we want to connect to. The calling program should close the socket descriptor, and should issue another socket call to obtain a new descriptor before attempting another connect call.
ETIMEDOUT	Connection establishment timed out, without establishing a connection. The calling program should close the socket descriptor, and issue another socket call to obtain a new descriptor before attempting another connect call.

getpeername()

This function returns the IP address / Port number of the remote system to which the socket is connected.

Prototype

```
int getpeername (int sockfd, struct sockaddr *localaddr, socklen_t
*addrlen);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor that we wish to obtain information about.
`fromAddressPtr`	A pointer to the address structure that we wish to store this information into.
`addressLengthPtr`	The length of the address structure.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	An error occurred

If the return value is -1, errno will be set to one of the following values. getpeername can fail for any of the following reasons:

errno	Description
`TM_EBADF`	`socketDescriptor` is not a valid descriptor.
`TM_ENOTCONN`	The socket is not connected.
`TM_EINVAL`	One of the passed parameters is not valid.

getsockname()

This function returns to the caller the Local IP Address/Port Number that we are using on a given socket.

Prototype

```
int getsockname (int sockfd, struct sockaddr *peeraddr,
socklen_t *addrlen);
```

Return Values

The function will return the following values.

Value	Description
0	Success
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

getsockname can fail for any of the following reasons:

errno	Description
`EBADF`	`sockfd` is not a valid descriptor.
`EINVAL`	One of the passed parameters is not valid.

Parameters

Parameter	Description
`sockfd`	The socket descriptor that we wish to inquire about.
`myAddressPtr`	The pointer to the address structure where the address information will be stored.
`addressLengthPtr`	The length of the address structure.

setsockopt()

setsockopt is used manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, protocolLevel is specified as `SOL_SOCKET`. To manipulate options at any other level, protocolLevel is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, protocolLevel is set to the TCP protocol number. The parameters optionValuePtr and optionlength are used to access option values for setsockopt. optionName and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file `<svc_net.h>` contains definitions for the options described below. Most socket-level options take an int pointer for option`ValuePtr`. For setsockopt, the integer value pointed to by the optionValuePtr parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. `SO_LINGER` uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in `<svc_net.h>`. struct linger contains the following members:

Parameter	Description
`l_onoff`	on = 1/off = 0
`l_linger`	linger time, in seconds

Prototype

```
int setsockopt (int sockfd, int level, int optname, const
void *optval, socklen_t optlen);
```

Return Values

SOL_SOCKET level

The following options are recognized at the socket level

protocolLevel options	Description
`SO_DONTROUTE`	Enable/disable routing bypass for outgoing messages. Default 0.
`SO_KEEPALIVE`	Enable/disable keep connections alive. Default 0.
`SO_LINGER`	Linger on close if data is present. Default is on with linger time of 60 seconds.
`SO_OOBINLINE`	Enable/disable reception of out-of-band data in band. Default 0.
`SO_REUSEADDR`	Enable this socket option to bind the same port number to multiple sockets using different local IP addresses. Note that to use this socket option, you also need to uncomment `USE_REUSEADDR_LIST` in trsystem.h. Default 0 (disable).

protocolLevel options	Description
SO_RCVLOWAT	The low water mark for receiving data.
SO_SNDLOWAT	The low water mark for sending data.
SO_RCVBUF	Set buffer size for input. Default 8192 bytes.
SO_SNDBUF	Set buffer size for output. Default 8192 bytes.
SO_RCVCOPYTCP	socket: fraction use of a receive buffer below which we try and append to a previous receive buffer in the socket receive queue. UDP socket: fraction use of a receive buffer below which we try and copy to a new receive buffer, if there is already at least a buffer in the receive queue. This is to avoid keeping large pre-allocated receive buffers, which the user has not received yet, in the socket receive queue. Default value is 4 (25%).
SO_SND_DGRAMS	The number of non-TCP datagrams that can be queued for send on a socket. Default 8 datagrams.
SO_RCV_DGRAMS	The number of non-TCP datagrams that can be queued for receive on a socket. Default 8 datagrams.
SO_SNDAPPEND	TCP socket only. Threshold in bytes of send buffer below, which we try and append, to previous send buffer in the TCP send queue. Only used with send. This is to try and regroup lots of partially empty small buffers in the TCP send queue waiting to be ACKED by the peer; otherwise we could run out of memory, since the remote TCP will delay sending ACKs.
SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_LINGER	Controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or decides it is unable to deliver the information. A timeout period, termed the linger interval, is specified in the setsockopt call when SO_LINGER is requested. If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible.

protocolLevel options	Description
SO_BROADCAST	requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv call without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high- volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).

IP_PROTOIP level

The following options are recognized at the IP level:

protocolLevel options	Description
IPO_HDRINCL	This is a toggle option used on Raw Sockets only. If the value is non-zero, it instructs the stack that the user is including the IP header when sending data. Default 0.
IPO_TOS	IP type of service. Default 0.
IPO_TTL	IP Time To Live in seconds. Default 64.
IPO_SRCADDR	Our IP source address. Default: first multi-home IP address on the outgoing interface.
IPO_MULTICAST_TTL	Change the default IP TTL for outgoing multicast datagrams
IPO_MULTICAST_IF	Specify a configured IP address that will uniquely identify the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket.
IPO_ADD_MEMBERSHIP	Add group multicast IP address to given interface (see struct ip_mreq data type below).
IPO_DROP_MEMBERSHIP	Delete group multicast IP address from given interface (see struct ip_mreq data type below).

ip_mreq structure definition:

```
struct ip_mreq
{
    struct in_addr      imr_multiaddr;
    struct in_addr      imr_interface
};
```

ip_mreq structure Members

Member	Description
`imr_multiaddr`	IP host group address that the user wants to join/leave
`imr_interface`	IP address of the local interface that the host group address is to be joined on, or is to leave from.

IP_PROTOTCP level

The following options are recognized at the TCP level. Options marked with an asterix can only be changed prior to establishing a TCP connection.

protocolLevel options	Description
`TCP_KEEPALIVE`	Sets the idle time in seconds for a TCP connection, before it starts sending keep alive probes. It cannot be set below the default value. Note that keep alive probes will be sent only if the `SO_KEEPALIVE` socket option is enabled. Default 7,200 seconds.
`TCP_MAXRT`	Sets the amount of time in seconds before the connection is broken, once TCP starts retransmitting, or probing a zero window, when the peer does not respond. A `TCP_MAXRT` value of 0 means to use the system default, and -1 means to retransmit forever. If a positive value is specified, it may be rounded-up to the connection next retransmission time. Note that unless the `TCP_MAXRT` value is -1 (wait forever), the connection can also be broken if the number of maximum retransmissions has been reached `TCP_MAX_REXMIT`). See `TCP_MAX_REXMIT` below. Default 0 (Which means use system default of `TCP_MAX_REXMIT` times network computed round trip time for an established connection; for a non established connection, since there is no computed round trip time yet, the connection can be broken when either 75 seconds, or when `TCP_MAX_REXMIT` times default network round trip time have elapsed, whichever occurs first).

protocolLevel options	Description
TCP_MAXSEG	Sets the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer., i.e. the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option) . The TCP_MAXSEG value can be decreased or increased prior to a connection establishment, but it is not recommended to set it to a value higher than the IP MTU minus 40 bytes (for example 1460 bytes on Ethernet), since this would cause fragmentation of TCP segments. Note: setting the TCP_MAXSEG option will inhibit the automatic computation of that value by the system based on the IP MTU (which avoids fragmentation), and will also inhibit Path Mtu Discovery. After the connection has started, this value cannot be changed. Note also that the TCP_MAXSEG value cannot be set below 64 bytes. Default value is IP MTU minus 40 bytes.
TCP_NODELAY	Set this option value to a non-zero value, to disable the Nagle algorithm that buffers the sent data inside the TCP. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). Default 0.
TCP_NOPUSH	Set this option value to a non-zero value, to force TCP to delay sending any TCP data until a full sized segment is buffered in the TCP buffers. Useful for applications that send continuous big chunks of data like FTP, and know that more data is coming. (Normally the TCP code sends a non full-sized segment, only if it empties the TCP buffer). Default 0
TCP_STDURG	Set this option value to a zero value, if the peer is a Berkeley system since Berkeley systems set the urgent data pointer to point to last byte of urgent data+1. Default 1 (urgent pointer points to last byte of urgent data as specified in RFC1122).
TCP_PACKET	Set this option value to a non-zero value to make TCP behave like a message-oriented protocol (i.e. respect packet boundaries) at the application level in both send and receive directions of data transfer. Note that for the receive direction to respect packet boundaries, the TCP peer which is sending must also implement similar functionality in its send direction. This is useful as a reliable alternative to UDP. Note that preserving packet boundaries with TCP will not work correctly if you use out-of-band data. USE_TCP_PACKET must be defined in trsystem.h to use the TCP_PACKET option. Default 0

protocolLevel options	Description
TCP_PEND_ACCEPT_RECV_WND	Specify the size (in bytes) of the listening socket's receive window. This size will override the default size or the size specified by setsockopt() with the SO_RCVBUF flag. Once accept() is called on the listening socket, the window size will return to the size specified by SO_RCVBUF (or the default). Note: This size may not be larger than the default window size to avoid shrinking of the receive window.
TCP_SEL_ACK	Set this option value to a non-zero value, to enable sending the TCP selective Acknowledgment option. Default 1
TCP_WND_SCALE	Set this option value to a non-zero value, to enable sending the TCP window scale option. Default 1
TCP_TS	Set this option value to a non-zero value, to enable sending the Time stamp option. Default 1
TCP_SLOW_START	Set this option value to zero, to disable the TCP slow start algorithm. Default 1
TCP_DELAY_ACK	Sets the TCP delay ack time in milliseconds. Default 200 milliseconds
TCP_MAX_REXMIT	Sets the maximum number of retransmissions without any response from the remote, before TCP gives up and aborts the connection. See also TCP_MAXRTabove. Default 12
TCP_KEEPALIVE_CNT	Sets the maximum numbers of keep alive probes without any response from the remote, before TCP gives up and aborts the connection. See also TCP_KEEPALIVE above. Default 8
TCP_FINWT2TIME	Sets the maximum amount of time TCP will wait for the remote side to close, after it initiated a close. Default 600 seconds
TCP_2MSLTIME	Sets the maximum amount of time TCP will wait in the TIME WAIT state, once it has initiated a close of the connection. Default 60 seconds
TCP_RTO_DEF	Sets the TCP default retransmission timeout value in milliseconds, used when no network round trip time has been computed yet. Default 3,000 milliseconds
TCP_RTO_MIN	Sets the minimum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TCP_RTO_MIN and TCP_RTO_MAX. Default 100 milliseconds
TCP_RTO_MAX	Sets the maximum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TCP_RTO_MIN and RTO_MAX. Default 64,000 milliseconds

protocolLevel options	Description
TCP_PROBE_MIN	Sets the minimum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TCP_PROBE_MIN and TCP_PROBE_MAX. Default 500 milliseconds
TCP_PROBE_MAX	Sets the maximum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TCP_PROBE_MIN and TCP_PROBE_MAX. Default 60,000 milliseconds
TCP_KEEPALIVE_INTV	Sets the interval between Keep Alive probes in seconds. See TCP_KEEPALIVE_CNT. This value cannot be changed after a connection is established, and cannot be bigger than 120 seconds. Default 75 seconds

Parameters

Parameter	Description
sockfd	The socket descriptor to set the options on.
protocolLevel	The protocol to set the option on. See below.
optionName	The name of the option to set. See below and above.
optionValuePtr	The pointer to a user variable from which the option value is set. User variable is of data type described below.
optionLength	The size of the user variable. It is the size of the option data type described below.

.Values for protocolLevel.

protocolLevel	Description
SOL_SOCKET	Socket level protocol.
IP_PROTOIP	IP level protocol.
IP_PROTOTCP	TCP level protocol

Values for optionName.

Protocol Level	Option Name	Option Data Type	Option Value
SOL_SOCKET	SO_DONTROUTE	int	0 or 1
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	0 or 1
	SO_OOBINLINE	int	
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1

Protocol Level	Option Name	Option Data Type	Option Value
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
	SO_RCVCOPY	unsigned int	
	SO_SNDAPPEND	unsigned int	
	SO_SND_DGRAMS	unsigned long	
	SO_RCV_DGRAMS	unsigned long	
	SO_UNPACKEDDATA	int	0 or 1
IP_PROTOIP	IPO_TOS	unsigned char	
	IPO_TTL	unsigned char	
	IPO_SRCADDR	ttUserIpAddress	
	IPO_MULTICAST_TTL	unsigned char	
	IPO_MULTICAST_IF	struct in_addr	
	IPO_ADD_MEMBERSHIP	struct ip_mreq	
	IPO_DROP_MEMBERSHIP	struct ip_mreq	
IP_PROTOTCP	TCP_KEEPALIVE	int	
	TCP_MAXRT	int	
	TCP_MAXSEG	int	
	TCP_NODELAY	int	0 or 1
	TCP_NOPUSH	int	0 or 1
	TCP_STDURG	int	0 or 1
	TCP_PACKET	int	0 or 1
	TCP_2MSLTIME	int	
	TCP_DELAY_ACK	int	
	TCP_FINWT2TIME	int	
	TCP_KEEPALIVE_CNT	int	
	TCP_KEEPALIVE_INTV	int	
	TCP_MAX_REXMIT	int	
	TCP_PROBE_MAX	unsigned long	
	TCP_PROBE_MIN	unsigned long	
	TCP_RTO_DEF	unsigned long	
	TCP_RTO_MAX	unsigned long	

Protocol Level	Option Name	Option Data Type	Option Value
	TCP_RTO_MIN	unsigned long	
	TCP_SEL_ACK	int	0 or 1
	TCP_SLOW_START	int	0 or 1
	TCP_TS	int	0 or 1
	TCP_WND_SCALE	int	0 or 1

Return Values

The function will return the following values:

Value	Description
0	Successful set of option
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

`Setsockopt` will fail if:

errno	Description
EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
ENOPROTOOPT	The option is unknown at the level indicated.
EPERM	Option cannot be set after the connection has been established.
EPERM	IPO_HDRINCL option cannot be set on non-raw sockets.
ENETDOWN	Specified interface not yet configured.
EADDRINUSE	Multicast host group already added to the interface.
ENOBUF	Not enough memory to add new multicast entry.
ENOENT	Attempted to delete a non-existent multicast entry on the specified interface.

getsockopt()

`getsockopt` is used retrieve options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, protocolLevel is specified as `SOL_SOCKET`. To manipulate options at any other level, protocolLevel is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, `protocolLevel` is set to the TCP protocol number. For `getsockopt`, the parameters `optionValuePtr` and `optionLengthPtr` identify a buffer in which the value(s) for the requested option(s) are to be returned. For getsockopt, optionLengthPtr is a value-result parameter, initially containing the size of the buffer pointed to by optionValuePtr, and modified on return to indicate the actual size of the value returned. optionName and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <svc_net.h> contains definitions for the options described below. Options vary in format and name. Most socket-level options take an int for optionValuePtr. `SO_LINGER` uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <svc_net.h>. struct linger contains the following members:

Parameter	Description
l_onoff	on = 1/off = 0
l_linger	linger time, in seconds

Prototype

```
int getsockopt (int sockfd, int level, int optname, void
*optval, socklen_t *optlen);
```

Return Values

SOL_SOCKET level

The following options are recognized at the socket level:

protocolLevel options	Description
`SO_ACCEPTCON`	Enable/disable listening for connections. listen turns on this option.
`SO_DONTROUTE`	Enable/disable routing bypass for outgoing messages. Default 0.

protocolLevel options	Description
SO_ERROR	When an error occurs on a socket, the stack internally sets the error code on the socket. It is called the pending error for the socket. If the user had called select for either readability or writability, select returns with either or both conditions set. The user can then retrieve the pending socket error, by calling getsockopt with this option name at the SOL_SOCKET level, and the stack will reset the internal socket error. Alternatively if the user is waiting for incoming data, read or other recv APIs can be called. If there is no data queued to the socket, the read/recv call returns SOCKET_ERROR, the stack resets the internal socket error, and the pending socket error can be returned if the user calls socketerrno (equivalent of errno). Note that the SO_ERROR option is useful when the user uses connect in non-blocking mode, and select.
SO_KEEPALIVE	Enable/disable keep connections alive. Default 0 (disable)
SO_OOBINLINE	Enable/disable reception of out-of-band data in band. Default is 0.
SO_REUSEADDR	Enable this socket option to bind the same port number to multiple sockets using different local IP addresses. Note that to use this socket option, you also need to uncomment USE_REUSEADDR_LIST in trsystem.h. Default 0 (disable).
SO_RCVLOWAT	The low water mark for receiving.
SO_SNDLOWAT	The low water mark for sending.
SO_RCVBUF	The buffer size for input. Default is 8192 bytes.
SO_SNDBUF	The buffer size for output. Default is 8192 bytes.
SO_RCVCOPY	TCP socket: fraction use of a receive buffer below which we try and append to a previous receive buffer in the socket receive queue. UDP socket: fraction use of a receive buffer below which we try and copy to a new receive buffer, if there is already at least a buffer in the receive queue. This is to avoid keeping large pre-allocated receive buffers, which the user has not received yet, in the socket receive queue.Default value is 4 (25%).
SO_SNDAPPEND	TCP socket only. Threshold in bytes of ‘send’ buffer below, which we try and append, to previous ‘send’ buffer in the TCP send queue. Only used with send. This is to try to regroup lots of partially empty small buffers in the TCP send queue waiting to be ACKED by the peer; otherwise we could run out of memory, since the remote TCP will delay sending ACKs. Default value is 128 bytes.
SO_SND_DGRAMS	The number of non-TCP datagrams that can be queued for send on a socket. Default 8 datagrams.

protocolLevel options	Description
SO_RCV_DGRAMS	The number of non-TCP datagrams that can be queued for receive on a socket. Default 8 datagrams.
SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages (every 2 hours) on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_LINGER	controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible. The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv call without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).

IP_PROTOIP level

The following options are recognized at the IP level:

protocolLevel options	Description
IPO_MULTICAST_IF	Get the configured IP address that uniquely identifies the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket.
IPO_MULTICAST_TTL	Get the default IP TTL for outgoing multicast datagrams.

protocolLevel options	Description
IPO_SRCADDR	Get the IP source address for the connection.
IPO_TOS	IP type of service. Default 0
IPO_TTL	IP Time To Live in seconds. Default 64

IP_PROTOTCP level

The following options are recognized at the TCP level. Options marked with an asterix can only be changed prior to establishing a TCP connection.

protocolLevel options	Description
TCP_KEEPALIVE	Get the idle time in seconds for a TCP connection before it starts sending keep alive probes. Note that keep alive probes will be sent only if the SO_KEEPALIVE socket option is enabled. Default 7,200 seconds.
TCP_MAXRT	Get the amount of time in seconds before the connection is broken once TCP starts retransmitting, or probing a zero window when the peer does not respond. A TCP_MAXRT value of 0 means the system default, and -1 means retransmit forever. Note that unless the TCP_MAXRT value is -1 (wait forever), the connection can also be broken if the number of maximum retransmission TCP_MAX_REXMIT has been reached. See TCP_MAX_REXMIT below. Default 0. (which means use system default of TCP_MAX_REXMIT times network computed round trip time for an established connection. For a non established connection, since there is no computed round trip time yet, the connection can be broken when either 75 seconds or when TCP_MAX_REXMIT times default network round trip time have elapsed, whichever occurs first).
TCP_MAXSEG	Get the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer. i.e. the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option). Default is IP MTU minus 40 bytes.
TCP_NODELAY	If this option value is non-zero, the Nagle algorithm that buffers the sent data inside the TCP is disabled. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). Default 0.
TCP_NOPUSH	If this option value is non-zero, then TCP delays sending any TCP data until a full sized segment is buffered in the TCP buffers. Useful for applications that send continuous big chunks of data and know that more data will be sent such as FTP. (Normally, the TCP code sends a non full-sized segment, only if it empties the TCP buffer). Default 0.

protocolLevel options	Description
TCP_STDURG	If this option value is zero, then the urgent data pointer points to the last bye of urgent data + 1, like in Berkeley systems. Default 1 (urgent pointer points to last byte of urgent data as specified in RFC1122).
TCP_PACKET	If this option value is non-zero, then TCP behaves like a message-oriented protocol (i.e. respects packet boundaries) at the application level in both send and receive directions of data transfer. Note that for the receive direction to respect packet boundaries, the TCP peer which is sending must also implement similar functionality in its send direction. This is useful as a reliable alternative to UDP. Note that preserving packet boundaries with TCP will not work correctly if you use out-of-band data. Default 0.
TCP_SEL_ACK	If this option value is zero, then TCP selective Acknowledgment options are disabled. Default 1.
TCPWND_SCALEI	f this option value is non-zero, then the TCP window scale option is enabled. Default 1.
TCP_TS	If this option value is non-zero, then the TCP time stamp option is enabled. Default 1.
TCP_SLOW_START	If this option value is non-zero, then the TCP slow start algorithm is enabled. Default 1.
TCPDELAY_ACK	Get the TCP delay ack time in milliseconds. Default 200 milliseconds.
TCPMAX_REXMIT	Get the maximum number of retransmissions without any response from the remote before TCP gives up and aborts the connection. See also TCP_MAXRT above. Default 12.
TCP_KEEPALIVE_CNT	Get the maximum number of keep alive probes without any response from the remote before TCP gives up and aborts the connection. See also TCP_KEEPALIVE above. Default 8.
TCPFINWT2TIME	Get the maximum amount of time TCP will wait for the remote side to close after it initiated a close. Default 600 seconds.
TCP2MSLTIME	Get the maximum amount of time TCP will wait in the TIME WAIT state once it has initiated a close of the connection. Default 60 seconds.
TCP_PEND_ACCEPT_RECV_WND	Specify the size (in bytes) of the listening socket's receive window. This size will override the default size or the size specified by setsockopt() with the SO_RCVBUF flag. Once accept() is called on the listening socket, the window size will return to the size specified by SO_RCVBUF (or the default). Note: This size may not be larger than the default window size to avoid shrinking of the receive window.

protocolLevel options	Description
TCP_RTO_DEF	Get the TCP default retransmission timeout value in milliseconds. Used when no network round trip time has been computed yet. Default 3,000 milliseconds.
TCP_RTO_MIN	Get the minimum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TCP_RTO_MIN and TCP_RTO_MAX. Default 100 milliseconds.
TCPRTO_MAX	Get the maximum retransmission timeout in milliseconds. The network computed retransmission timeout is bound by TCPRTO_MIN and RTO_MAX. Default 64,000 milliseconds.
TCPPROBE_MIN	Get the minimum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TCP_PROBE _MIN and TCP_PROBE _MAX. Default 500 milliseconds.
TCP_PROBE_MAX	Get the maximum window probe timeout interval in milliseconds. The network computed window probe timeout is bound by TCP_PROBE _MIN and TCP_PROBE _MAX. Default 60,000 milliseconds.
TCP_KEEPALIVE_INTV	Get the interval between Keep Alive probes in seconds. See TCP_KEEPALIVE_CNT. Default 75 seconds.

Parameters

Parameter	Description
sockfd	The socket descriptor to get the option from.
protocolLevel	The protocol to get the option from. See below.
optionName	The option to get. See above and below.
optionValuePtr	The pointer to a user variable into which the option value is returned. User variable is of data type described below.
optionLengthPtr	Pointer to the size of the user variable, which is the size of the option data type, described below. It is a value-result parameter, and the user should set the size prior to the call.

Values for protocolLevel.

Protocol Level	Description
SOL_SOCKET	Socket level protocol.
IP_PROTOIP	IP level protocol.
IP_PROTOTCP	TCP level protocol

Values for `optionName`

Protocol Level	Option Name	Option Data Type	Option Value
SOL_SOCKET	SO_ACCEPTCON	int	0 or 1
	SO_DONTROUTE	int	0 or 1
	SO_ERROR	int	
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	
	SO_OOBINLINE	int	0 or 1
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
	SO_RCVCOPY	unsigned int	
	SO_SND_DGRAMS	unsigned long	
	SO_SNDAPPEND	unsigned int	
	SO_UNPACKEDDATA	int	0 or 1
IP_PROTOIP	IPO_MULTICAST_IF	struct in_addr	
	IPO_MULTICAST_TTL	unsigned char	
	IPO_TOS	unsigned char	
	IPO_TTL	unsigned char	
	IPO_SRCADDR	ttUserIpAddress	
IP_PROTOTCP	TCP_KEEPALIVE	int	
	TCP_MAXRT	int	
	TCP_MAXSEG	int	
	TCP_NODELAY	int	0 or 1
	TCP_NOPUSH	int	0 or 1
	TCP_STDURG	int	0 or 1
	TCP_2MSLTIME	int	
	TCP_DELAY_ACK	int	
	TCP_FINWT2TIME	int	
	TCP_KEEPALIVE_CN	int	
	TCP_KEEPALIVE_IN	int	

Protocol Level	Option Name	Option Data Type	Option Value
	TCP_MAX_REXMIT	int	
	TCP_PACKET	int	0 or 1
	TCP_PROBE_MAX	unsigned long	
	TCP_PROBE_MIN	unsigned long	
	TCP_RTO_DEF	unsigned long	
	TCP_RTO_MAX	unsigned long	
	TCP_RTO_MIN	unsigned long	
	TCP_SEL_ACK	int	0 or 1
	TCP_SLOW_START	int	0 or 1
	TCP_TS	int	0 or 1
	TCP_WND_SCALE	int	0 or 1

Return Values

The function will return the following values:

Value	Description
0	Successful set of option
-1	An error occurred

If the return value is -1, errno will be set to one of the following values:

`getsockopt` will fail if:

errno	Description
EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
ENOPROTOOPT	The option is unknown at the level indicated

recv()

`recv` is used to receive messages from another socket. recv may be used only on a connected socket (see connect(), accept()). sockfd is a socket created with socket or accept. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket()). The length of the message returned could also be smaller than bufferLength (this is not an error). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the `MSG_DONTWAIT` flag is set in the flags parameter, in which case -1 is returned with socket error being set to `EWOULDBLOCK`.

Out-of-band data not in the stream (urgent data when the `SO_OOBINLINE` option is not set (default)) (TCP protocol only).

A single out-of-band data byte is provided with the TCP protocol when the `SO_OOBINLINE` option is not set. If an out-of-band data byte is present, recv with the MSG_OOB flag not set will not read past the position of the out-of-band data byte in a single recv request. That is, if there are 10 bytes from the current read position until the out-of-band byte, and if we execute a recv specifying a `bufferLength` of 20 bytes, and a flag value of 0, recv will only return 10 bytes. This forced stopping is to allow us to execute the `SOIOCATMARK` socketioctl to determine when we are at the out-of-band byte mark. Alternatively, `GetOobDataOffset` can be used instead of socketioctl to determine the offset of the out-of-band data byte.

Out-of-band data (when the `SO_OOBINLINE` option is set (see setsockopt()).

(TCP protocol only) If the `SO_OOBINLINE` option is enabled, the out-of-band data is left in the normal data stream and is read without specifying the `MSG_OOB`. More than one out-of-band data bytes can be in the stream at any given time. The out-of-band byte mark corresponds to the final byte of out-of-band data that was received. In this case, the `MSG_OOB` flag cannot be used with recv. The out-of-band data will be read in line with the other data. Again, `recv` will not read past the position of the out-of-band mark in a single recv request. Again, socketioctl with the `SOIOCATMARK`, or `GetOobDataOffset` can be used to determine where the last received out-of-band byte is in the stream.

select may be used to determine when more data arrives, or/and when out-of-band data arrives.

Prototype

```
int recv (int sockfd, void *buff, size_t nbytes, int flags);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor from which to receive data.

Parameter	Description
`bufferPtr`	The buffer into which the received data is put.
`bufferLength`	The length of the buffer area that bufferPtr points to.
`flags`	See below.

The flags parameter is formed by `ORing` one or more of the following:

Parameter	Description
`MSG_DONTWAIT`	Do not wait for data, but rather return immediately
`MSG_OOB`	Read any “out-of-band” data present on the socket rather than the regular “in-band” data
`MSG_PEEK`	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data

Returns

The function will return the following values.

Value	Description
>0	Number of bytes actually received from the socket.
0	EOF or remote host has closed the connection.
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

`recv` will fail if:

errno	Description
`EBADF`	The socket descriptor is invalid
`EMSGSIZE`	The socket requires that message be received atomically, and bufferLength was too small
`EWOULDBLOCK`	The socket is marked as non-blocking or the `MSG_DONTWAIT` flag is used and no data is available to be read, or the `MSG_OOB` flag is set and the out of band data has not arrived yet from the peer
`EINVAL`	One of the parameters is invalid, or the `MSG_OOB` flag is set and, either the `SO_OOBINLINE` option is set, or there is no out of band data to read or coming from the peer
`ENOTCONN`	Socket is not connected
`EHOSTUNREACH`	No route to the connected host

send()

`send` is used to transmit a message to another transport end-point. `send` may be used only when the socket is in a connected state. `sockfd` is a socket created with `socket`.

If the message is too long to pass automatically through the underlying protocol (non-TCP protocol), then the error `EMSGSIZE` is returned and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

Blocking socket send: if the socket does not have enough buffer space available to hold the message being sent, send blocks.

Non blocking stream (TCP) socket send: if the socket does not have enough buffer space available to hold the message being sent, the send call does not block. It can send as much data from the message as can fit in the TCP buffer and returns the length of the data sent. If none of the message data fits, then -1 is returned with socket error being set to `EWOULDBLOCK`.

Non blocking datagram socket send: if the socket does not have enough buffer space available to hold the message being sent, no data is being sent and -1 is returned with socket error being set to `EWOULDBLOCK`.

The `select` call may be used to determine when it is possible to send more data.

Prototype

```
int send (int sockfd, const void *buff, size_t nbytes, int
flags);
```

Return Values

Sending Out-of-Band Data

For example, if you have remote login application, and you want to interrupt with a ^C keystroke, at the socket level you want to be able to send the ^C flagged as special data (also called out-of-band data). You also want the TCP protocol to let the peer (or remote) TCP know as soon as possible that a special character is coming, and you want the peer (or remote) TCP to notify the peer (or remote) application as soon as possible.

At the TCP level, this mechanism is called TCP urgent data. At the socket level, the mechanism is called out-of-band data. Out-of-band data generated by the socket layer, is implemented at the TCP layer with the urgent data mechanism. The user application can send one or several out-of-band data bytes. With TCP you cannot send the out-of-band data ahead of the data that has already been buffered in the TCP send buffer, but you can let the other side know (with the urgent flag, i.e. the term urgent data) that out-of-band data is coming, and you can let the peer TCP know the offset of the current data to the last byte of out-of-band data.

So with TCP, the out-of-band data byte(s) are not sent ahead of the data stream, but the TCP protocol can notify the remote TCP ahead of time that some out-of-band data byte(s) exist. What TCP does, is mark the byte stream where urgent data ends, and set the Urgent flag bit in the TCP header flag field, as long as it is sending data before ,or up to, the last byte of out-of-band data.

In your application, you can send out-of-band data, by calling the `send` function with the `MSG_OOB` flag. All the bytes of data sent that way (using send with the `MSG_OOB` flag) are out-of-band data bytes. Note that if you call `send` several times with out-of-band data, TCP will always keep track of where the last out- of-band byte of data is in the byte data stream, and flag this byte as the last byte of urgent data. To receive out-of-band data, please see the `recv` section of this manual.

Parameters

Parameter	Description
`sockfd`	The socket descriptor to use to send data
`bufferPtr`	The buffer to send
`bufferLength`	The length of the buffer to send
`flags`	See below

The flags parameter is formed by `ORing` one or more of the following:

Parameter	Description
`MSG_DONTWAIT`	Do not wait for data send to complete, but rather return immediately.
`MSG_OOB`	Send “out-of-band” data on sockets that support this notion. The underlying protocol must also support “out-of-band” data. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.
`MSG_DONTROUTE`	The `SO_DONTROUTE` option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Returns

The function will return the following values:

Value	Description
>=0	Number of bytes actually sent on the socket
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

Send will fail if:

errno	Description
`EBADF`	The socket descriptor is invalid.

errno	Description
EINVAL	One of the parameters is invalid.the `bufferPtr` is NULL, the `bufferLength` is ? 0 or an unsupported flag is set.
ENOBUFS	There was insufficient user memory available to complete the operation.
EHOSTUNREACH	Non-TCP socket only. No route to destination host.
EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.
ENOTCONN	Socket is not connected.
ESHUTDOWN	User has issued a write shutdown or a `tfClose` call (TCP socket only).

recvfrom()

Use `recvfrom` to receive messages from another socket. `recvfrom` may be used to receive data on a socket whether it is in a connected state or not but not on a TCP socket. sockfd is a socket created with socket. If from`Ptr` is not a NULL pointer, the source address of the message is filled in. `fromLengthPtr` is a value-result parameter, initialized to the size of the buffer associated with fromPtr, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket()). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the `MSG_DONTWAIT` flag is set in the flags parameter, in which case -1 is returned with socket error being set to `EWOULDBLOCK`. select may be used to determine when more data arrives, or/and when out-of-band data arrives.

Prototype

```
int recvfrom (int sockfd, void * buff, size_t nbytes, int
flags, struct sockaddr *from, socklen_t *addrlen);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor to receive data from.
`bufferPtr`	The buffer to put the received data
`bufferLength`	The length of the buffer area that `bufferPtr` points to
`flags`	See Below.
`fromPtr`	The socket from which the data is (or to be) received.
`fromLengthPtr`	The length of the data area the `fromPtr` points to then upon return the actual length of the from data

The flags parameter is formed by `ORing` one or more of the following:

Value	Description
`MSG_DONTWAIT`	Do not wait for data, but rather return immediately
`MSG_PEEK`	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

Return Values

The function will return the following values.

Value	Description
>0	Number of bytes actually received from the socket.
0	EOF
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

recvfrom will fail if:

errno	Description
EBADF	The socket descriptor is invalid.
EINVAL	One of the parameters is invalid.
EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
EPROTOTYPE	TCP protocol requires usage of recv, not recvfrom.
EWOULDBLOCK	The socket is marked as non-blocking and no data is available

sendto()

`sendto` is used to transmit a message to another transport end-point. `sendto` may be used at any time (either in a connected or unconnected state), but not for a TCP socket. `sockfd` is a socket created with socket. The address of the target is given by to with `toLength` specifying its size.

If the message is too long to pass automatically through the underlying protocol, then -1 is returned with the socket error being set to `EMSGSIZE`, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

If the socket does not have enough buffer space available to hold the message being sent, and is in blocking mode, sendto blocks. If it is in non-blocking mode or the `MSG_DONTWAIT` flag has been set in the flags parameter, -1 is returned with the socket error being set to `EWOULDBLOCK`.

The select call may be used to determine when it is possible to send more data.

Prototype

```
int sendto (int sockfd, void * buff, size_t nbytes, int
flags, const struct sockaddr *to, socklen_t addrlen);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor to use to send data.
`bufferPtr`	The buffer to send.
`bufferLength`	The length of the buffer to send.
`toPtr`	The address to send the data to.
`toLength`	The length of the to area pointed to by toPtr.
`flags`	See below

The flags parameter is formed by ORing one or more of the following:

Value	Description
`MSG_DONTWAIT`	Don't wait for data send to complete, but rather return immediately.
`MSG_DONTROUTE`	The `SO_DONTROUTE` option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Return Values

The function will return the following values.

Value	Description
>=0	Number of bytes actually sent on the socket
-1	An error occurred

Value	Description
EHOSTDOWN	Destination host is down

If the return value is -1, errno will be set to one of the following values.

sendto will fail if:

errno	Description
EBADF	The socket descriptor is invalid.
ENOBUFS	There was insufficient user memory available to complete the operation.
EINVAL	One of the parameters is invalid: the `bufferPtr` is NULL, the `bufferLength` is ? 0, an unsupported flag is set, toPtr is NULL or toLength contains an invalid length.
EHOSTUNREACH	No route to destination host.
EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.
EPROTOTYPE	TCP protocol requires usage of send not sendto.
EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.

shutdown()

Shutdown is a socket in read, write, or both directions determined by the parameter howToShutdown.

Prototype

```
int shutdown (int sockfd, int howto);
```

Parameters

Parameter	Description
sockfd	The socket to shutdown
howToShutdown	Direction: 0 = Read 1 = Write 2 = Both

Return Values

The function will return the following values.

Value	Description
0	Success
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

shutdown will fail if:

errno	Description
EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
EOPNOTSUPP	Invalid socket type - can only shutdown TCP sockets.
ESHUTDOWN	Socket is already closed or is in the process of closing.

socketclose()

This function is used to close a socket. It is not called close to avoid confusion with an embedded kernel file system call.

Prototype

```
int socketclose (int sockfd);
```

Return Values

The function will return the following values.

Value	Description
0	Operation completed successfully
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

tfClose can fail for the following reasons:

errno	Description
`EBADF`	The socket descriptor is invalid.
`EALREAY`	A previous `tfClose` call is already in progress.
`ETIMEDOUT`	The linger option was on with a non-zero timeout value, and the linger timeout expired before the TCP close handshake with the remote host could complete (blocking TCP socket only).

Parameters

Parameter	Description
`sockfd`	The socket descriptor to close

socketerrno()

Function Description

This function is used when any socket call fails (SOCKET_ERROR), to get the error value back. This call has been added to allow for the lack of a per-process errno value that is lacking in most embedded realtime kernels.

Prototype

```
int socketerrno(int sockfd);
```

Return Values

The last errno value for a socket.

Parameters

Parameter	Description
`sockfd`	The socket descriptor to get the error on.

select()

`select` examines the socket descriptor sets whose addresses are passed in `readSocketsPtr`, `writeSocketsPtr`, and `exceptionSocketsPtr` to see if any of their socket descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. Out-of-band data is the only exceptional condition. The `numberSockets` argument specifies the number of socket descriptors to be tested. Its value is the maximum socket descriptor to be tested, plus one. The socket descriptors from 0 to `numberSockets` -1 in the socket descriptor sets are examined. On return, select replaces the given socket descriptor sets with subsets consisting of those socket descriptors that are ready for the requested operation. The return value from the call to select is the number of ready socket descriptors. The socket descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such file descriptor sets:

Value	Description
`FD_ZERO`	(`&fdset`);Initializes a socket descriptor set (`fdset`) to the null set.
`FD_SET`	(`fd, &fdset`);Includes a particular socket descriptor fd in `fdset`.
`FD_CLR`	(`fd, &fdset`);Removes fd from `fdset`.
`FD_ISSET`	(`fd, &fdset`);Is non-zero if fd is a member of `fdset`, zero otherwise.

NOTE

The term "fd" is used for BSD compatibility since select is used on both file systems and sockets under BSD Unix.

The timeout parameter specifies a length of time to wait for an event to occur before exiting this routine. struct timeval contains the following members:

Value	Description
`tv_sec`	Number of seconds to wait
`tv_usec`	Number of microseconds to wait

If the total time is less than one millisecond, select will return immediately to the user.

Prototype

```
int select(int maxfd, fd_set *in, fd_set *out, fd_set *ex,
struct timeval *timeout);
```

Parameters

Parameter	Description
`numberSockets`	Biggest socket descriptor to be tested, plus one.
`readSocketsPtr`	The pointer to a mask of sockets to check for a read condition.
`writeSocketsPtr`	The pointer to a mask of sockets to check for a write condition.
`exceptionSocketsPtr`	The pointer to a mask of sockets to check for an exception condition: Out of Band data.
`timeOutPtr`	The pointer to a structure containing the length of time to wait for an event before exiting.

Return Values

The function will return the following values.

Value	Description
>0	Number of sockets that are ready
0	Time limit exceeded
-1	An error occurred

If the return value is -1, errno will be set to one of the following values.

select will fail if:

errno	Description
EBADF	One of the socket descriptors is bad.

socketset_owner()

`socketset_owner` transfers ownership of an open socket to another task. Following this call the caller will not be able to access the socket. No changes to the socket state are made and buffers are not cleared. (The caller may wish to do this before transferring control.) Pending events for the socket are not transferred to the new task.

Prototype

```
int socketset_owner(int sockfd, int task_id);
```

Parameters

Parameter	Description
`sockfd`	Socket descriptor
`task_id`	Task ID

Return Values

The function will return the following values.

Value	Description
0	Success.
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EBADF`	Invalid handle, or caller does not own device
`EINVAL`	Invalid task number

socketioctl()

This function is used to set/clear non-blocking I/O, to get the number of bytes to read, or to check whether the specified socket's read pointer is currently at the out of band mark. It is not called ioctl to avoid confusion with an embedded kernel file system call.

Parameter	Description
`FIONBIO`	Set/clear nonbllocking I/O: if the int cell pointed to by `argumentPtr` contains a non-zero value, then the specified socket non-blocking flag is turned on. If it contains a zero value, then the specified socket non-blocking flag is turned off. See also `tfBlockingState`.
`FIONREAD`	Stores in the int cell pointed to by `argumentPtr` the number of bytes available to read from the socket descriptor. See also tfGetWaitingBytes.
`SIOCATMARK`	Stores in the int cell pointed to by `argumentPtr` a non-zero value if the specified socket's read pointer is currently at the out-of-band mark, zero otherwise. See revc call for a description of out-of-band data. See also tfGetOobDataOffset.

Prototype

```
int socketioctl(int sockfd, int cmd, int*arg);
```

Returns

The function will return the following values.

socketioctl can fail for the following reasons:

Value	Description
0	Success.
-1	An error has occurred.

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EBADF`	The socket descriptor is invalid.
`EINVAL`	Request is not one of `FIONBIO`, `FIONREAD`, or `SOIOCATMARK`.

Parameters

Parameter	Description
`sockfd`	The socket descriptor we want to perform the ioctl request on.
`request`	`FIONBIO`, `FIONREAD`, or `SIOCATMARK`
`argumentPtr`	A pointer to an int cell in which to store the request parameter or request result.

DnsGetHostByName()

This function retrieves the IP address associated with the given hostname.

Prototype

```
int DnsGetHostByName(const char *hostname, in_addr_t *ip);
```

Return Values

Value	Description
`TM_EINVAL`	Invalid host name string or IP address pointer.
`TM_EWOULDBLOCK`	DNS lookup in progress. The user should continue to call DnsGetHostName with the same parameters until it returns a value other than `TM_EWOULDBLOCK`. Only returned in non-blocking mode.
`TM_ENOERROR`	DNS lookup successful, IP address stored in `*ipAddressPtr`.

Parameters

Parameter	Description
`hostnameStr`	Hostname to resolve.
`ipAddressPtr`	Set to the IP address of the host.

gethostbyname()

Prototype `int gethostbyname(const char hostname, struct hostent he);`

blockingIO()

Prototype `int blockingIO(int sockfd);`

inet_addr()

This function converts an IP address from the decimal dotted notation to an unsigned long.

Prototype

```
in_addr_t inet_addr(char *strptr);
```

Return Values

Value	Description
-1	Error
Other	The IP Address in Network Byte Order

Parameters

Parameter	Description
`ipAddressDottedStringPtr`	The dotted string (i.e. "208.229.201.4")

DNS Resolver API

- DnsSetServer()
- DnsCacheInvalidate()
- DnsSetUserOption()

DnsSetServer()

Function Description

This function sets the address of the primary and secondary DNS server. To set the primary DNS server serverNumber should be set to `DNS_PRI_SERVER`; for the secondary server it should be set to `DNS_SEC_SERVER`. To remove a previously set entry, set `serverIpAddr` to zero.

Prototype

```
int DnsSetServer(ip_addr serverIpAddr, int serverNumber);
```

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	serverNumber is not `DNS_PRI_SERVER` or `DNS_SEC_SERVER`.
`ENOERROR`	DNS server set successfully.

Parameters

Parameter	Description
`serverIpAddr`	IP address of the DNS server
`serverNumber`	Primary or secondary server

DnsCacheInvalidate()

TBD

DnsSetUserOption()

Function Description

This function sets various DNS options which are outlined below:

Option Type	Type	Description
`DNS_OPTION_RETRIES`	(int)	Maximum number of times of retransmit a DNS request.
`DNS_OPTION_CACHE_SIZE`	(int)	Maximum number of entries in the DNS cache. Must be greater than zero.
`DNS_OPTION_TIMEOUT`	(int)	Amount of time (in seconds) to wait before retransmitting a DNS request.
`DNS_OPTION_CACHE_TTL`	(int)	The maximum amount of time to keep a DNS response in the cache. Note: This value only affects new entries as they are added to the cache. Existing entries will not be changed.

Prototype

```
int DnsSetUserOption(int optType, void *optValue, int optLen);
```

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	Invalid value for above option.
`ENOPROTOOPT`	Option not supported (not in above list).
`ENOERROR`	Option set successfully.

Parameters

Parameter	Description
`optionType`	See above
`optionValuePtr`	Pointer to the value for above option
`optionLen`	Length, in bytes, of the value pointed to by `optionValuePtr`

Ping API

Use the following APIs for Ping functions:

- PingOpenStart()
- PingGetStats()
- PingCloseStop()

PingOpenStart()

Function Description

This function opens an ICMP socket and starts sending PING echo requests to a remote host as specified by the `remoteHostName` parameter. PING echo requests are sent every pingInterval seconds. The PING length (not including IP and ICMP headers) is given by the pingDataLength parameter. To get the PING connection results and statistics, the user must call `PingGetStatistics`. To stop the system from sending PING echo requests and to close the ICMP socket, the user must call `PingClose`.

Prototype

```
int PingOpenStart(const char *remoteHost, int seconds, int datalen);
```

Return Values

The function will return the following values.

Value	Description
>0	Socket descriptor
-1	Error

If the return value is -1, errno will be set to one of the following values.

errnoDescription

errno	Description
`EINVAL`	`remoteHostNamePtr` was a null pointer
`EINVAL`	`pingInterval` was negative
`EINVAL`	`pingDataLength` was negative of bigger than 65595, maximum value allowed by the IP protocol.
`ENOBUFS`	There was insufficient user memory available to complete the operation.
`EMSGSIZE`	`pingDataLength` exceeds socket send queue limit, or `pingDataLength` exceeds the IP MTU, and fragmentation is not allowed.
`EHOSTUNREACH`	No route to remote host

Parameters

Parameter	Description
`remoteHostNamePtr`	Pointer to character array containing a dotted decimal IP address.
`pingInterval`	Interval in seconds between PING echo requests. If set to zero, defaults to 1 second.
`pingDataLength`	User Data length of the PING echo request. If set to zero, defaults to 56 bytes. If set to a value between 1, and 3, defaults to 4 bytes.

PingGetStats()

This function gets Ping statistics in the `ttPingInfo` structure that `pingInfoPtr` points to. `sockfd` should match the socket descriptor returned by a call to `PingOpenStart`. `pingInfoPtr` must point to a `ttPingInfo` structure allocated by the user.

Prototype

```
int PingGetStats(int sockfd, PingInfo *pingInfoptr);
```

Parameters

Parameter	Description
`sockfd`	The socket descriptor as returned by a previous call to `PingOpenStart`.
`pingInfoPtr`	The pointer to an empty structure where the results of the PING connection will be copied upon success of the call. (See below for details.)

`PingInfo` Data structure:

Field	Data Type	Description
`pgiTransmitted`	unsigned long	Number of transmitted PING echo request packets so far.
`pgiReceived`	unsigned long	Number of received PING echo reply packets so far (not including duplicates)
`pgiDuplicated`	unsigned long	Number of duplicated received PING echo reply packets so far.
`pgiLastRtt`	unsigned long	Round trip time in milliseconds of the last PING request/reply.
`pgiMaxRtt`	unsigned long	Maximum round trip time in milliseconds of the PING request/ reply packets.
`pgiMinRtt`	unsigned long	Minimum round trip time in milliseconds of the PING request/reply packets.
`pgiAvrRtt`	unsigned long	Average round trip time in milliseconds of the PING request/reply packets.
`pgiSumRtt`	unsigned long	Sum of all round trip times in milliseconds of the PING request/reply packets.
`pgiSendErrorCode`	int	PING send request error code if any.
`pgiRecvErrorCode`	int	PING recv error code if any (including ICMP error from the network).

PingCloseStop()

This function stops the sending of any PING echo requests and closes the ICMP socket that had been opened via `PingOpenStart`.

Prototype

```
int PingCloseStop(int sockfd);
```

Return Values

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`TM_EBADF`	`socketDescriptor` is not a valid descriptor

Parameters

Parameter	Description
`socketDescriptor`	An ICMP PING socket descriptor as returned by `PingOpenStart`

PPP API

Use the following APIs:

- GetPppDnsIpAddress()
- GetPppPeerIpAddress()
- PppSetOption()
- SetPpppPeerIpAddress()
- GetPppEvents()
- PppSetAuthPriority()

GetPppDnsIpAddress()

This function is used to return the DNS Addresses as negotiated by the remote PPP server. This function can only be used with PPP devices. If no DNS address is negotiated, the IP address returned will be 0.0.0.0

Prototype

```
int GetPppDnsIpAddress(int handle, ip_addr dnsIpAddressPtr,
int flag);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
EINVAL	One of the parameters is null or the device is a LAN device
ENETDOWN	Interface is not configured

Parameters

Parameter	Description
handle	The device handle to get the DNS IP address from.
dnsIpAddressPtr	The pointer to the buffer where the DNS IP address will be stored
flag	One of the following: DNS_PRIMARY or DNS_SECONDARY

GetPppPeerIpAddress()

Function Description

This function is used to get the PPP address that the remote PPP has used (respectively SLIP address of the remote SLIP). If a default gateway needs to be added for that interface, then the retrieved IP address should be used to add a default gateway through the corresponding interface. If a static route needs to be added for that interface, then the retrieved IP address should be used to add a static route through the corresponding interface.

Prototype

```
int GetPppPeerIpAddress(int handle, ip_addr ifIpAddress);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	One of the parameters is null, or the device is a LAN device
`ENETDOWN`	Interface is not configured

Parameters

Parameter	Description
`handle`	The device handle to get the Peer IP address from.
`ifIpAddressPtr`	The pointer to the buffer where the Peer PPP IP address will be stored into

PppSetOption()

Function Description

This function is used to set the PPP options that we wish to negotiate as well as those options that we will allow. This allows us to change the link away from the default parameters described in RFC1661.

Prototype

```
int PppSetOption(int handle, int protocolLevel, int
remoteLocalFlag, int optionName, const char *optionValuePtr,
int optionLength);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
ENOPROTOPT	protocolLevel or optionName is invalid
EINVAL	The option value or length is invalid

Parameters

Parameter	Description
handle	The device handle to set these options for
protocolLevel	The protocol which this option should be applied. Current supported protocols are: PPP_LCP_PROTOCOL PPP_IPCP_PROTOCOL PPP_PAP_PROTOCOL PPP_CHAP_PROTOCOL
remoteLocalFlag	This flag describes whether the option is for what we want to use for our side of the link (PPP_OPT_WANT) or if it what we will allow the remote side to use (PPP_OPT_ALLOW)
optionName	The name of the option (see below)
optionValuePtr	The value of the option (see below)
optionLength	The length of the option (see below)

LCP (PPP_LCP_PROTOCOL)

Option Name	Length	Meaning
LCP_ADDRCONTROL_COMP	1	A Boolean value specifying whether address field compression should be used. Default: OFF
LCP_PROTOCOL_COMP	1	A Boolean value specifying whether protocol field compression should be used. Default: OFF
LCP_MAGIC_NUMBER	1	A Boolean value indicating whether to specify a magic number. Default: OFF
LCP_MAX_FAILURES	1	Sets the maximum number of LCP configuration failures. This determines the maximum number of configuration NAKs that will be sent before we reject an option. Default: 5
LCP_MAX_RECV_UNIT	2	Specifies the largest MRU that we will allow and the default MRU that we want to use. Default: 1500
LCP_ACCM	4	Specifies the async control character map that we want to use, and if we want to allow the remote side to be able to set his ACCM. Default: 0xffffffff
LCP_AUTH_PROTOCOL	2	Use when authenticating to our peer, and vice versa (e.g. PAP or CHAP). Possible values are: PPP_PAP_PROTOCOL PPP_CHAP_PROTOCOL Default: No authentication
LCP_TERM_RETRY	1	Sets the maximum number of Terminate requests that the local peer will send (without receiving a Terminate Ack) before terminating the connection. Default: 3
LCP_CONFIG_RETRY	1	Sets the maximum number of LCP config requests that will be sent without receiving a LCP ack/nak/reject. remoteLocalFlag has no effect. Default: 10
LCP_TIMEOUT	1	Sets the LCP retransmission timeout in seconds. Default: 3 seconds

Option Name	Length	Meaning
LCP_QUALITY_PROTOCOL	4	Setting this option enables link quality monitoring. The option value is specified in hundredths of a second, and it configures the maximum time to delay (i.e. LQR timer period) between sending Link-Quality-Report messages (refer to RFC-1989, Reporting- Period field of the Quality-Protocol Configuration Option). This option can be set for either the local or the remote end of the link, however the direction in which it applies is the opposite of what one would expect: when `remoteLocalFlag` is set to `PPP_OPT_WANT`, this specifies an option value that we want the remote end of the link to use, and when `remoteLocalFlag` is set to `PPP_OPT_ALLOW`, this specifies an option value that we will allow the remote end to configure us to use. If a non-zero option value is specified, the LQR timer is started with the specified timeout period to pace the sending of Link-Quality-Report messages, and this timer is restarted whenever a Link- Quality-Report message is sent. If the specified option value is 0, no LQR timer is used, but instead a Link-Quality-Report message is sent as a response every time one is received from the peer. At least one side of the link must use the LQR timer to pace the sending of Link-Quality-Report messages when link quality monitoring is enabled, therefore this option value should not be set to 0 for both ends of the link.

LQM (LCP_QUALITY_PROTOCOL)

Option Name	Length	Meaning
LCP_QUALITY_PROTOCOL	4	Setting this option enables link quality monitoring. The option value is specified in hundredths of a second, and it configures the maximum time to delay (i.e. LQR timer period) between sending Link-Quality-Report messages (refer to RFC-1989, Reporting- Period field of the Quality-Protocol Configuration Option). This option can be set for either the local or the remote end of the link, however the direction in which it applies is the opposite of what one would expect: when remoteLocalFlag is set to PPP_OPT_WANT, this specifies an option value that we want the remote end of the link to use, and when remoteLocalFlag is set to PPP_OPT_ALLOW, this specifies an option value that we will allow the remote end to configure us to use. If a non-zero option value is specified, the LQR timer is started with the specified timeout period to pace the sending of Link-Quality-Report messages, and this timer is restarted whenever a Link-Quality-Report message is sent. If the specified option value is 0, no LQR timer is used, but instead a Link-Quality-Report message is sent as a response every time one is received from the peer. At least one side of the link must use the LQR timer to pace the sending of Link-Quality-Report messages when link quality monitoring is enabled, therefore this option value should not be set to 0 for both ends of the link.

IPCP (PPP_IPCP_PROTOCOL)

Option Name	Length	Meaning
IPCP_COMP_PROTOCOL	2	Specifies the type of compression to use over the link (optional). PPP_COMP_TCP_PROTOCOL selects Van Jacobson header compression.
IPCP_MAX_FAILURES	1	Sets the maximum number of IPCP configuration failures. This determines the maximum number of configuration NAKs that will be sent before we reject an option. Default: 5

Option Name	Length	Meaning
IPCP_VJ_SLOTS	1	The number of slots used to store state information for each side of a VJ compressed link. This value is determined by the maximum number of concurrent TCP sessions that you will have. Default: 1 slot.
IPCP_IP_ADDRESS	4	Specifies if we want to allow the remote to set their IP address. Please see “setting a peer PPP IP address” below for explanations. Default: Don’t Allow
IPCP_DNS_PRI	4	Specifies the IP addresses of the Primary DNS Server we will allow the remote to use or the Primary DNS server that we ant to us. Se the section setting a PPP IP Address Default: Don’t Allow
IPCP_DNS_SEC	4	Specifies the IP Address of the Secondary DNS server we will allow the remote to use or the Secondary DNS server that we want to use. See the section setting a PPP IP Address. Default: Don’t Allow

PPP (PPP_PROTOCOL)

Option Name	Length	Meaning
PPP_SEND_BUFFER_SIZE	2	Length of data buffered by the PPP link layer (but not beyond the end of a packet) before the device driver send function is called. Default: 1 byte
IPCP_RETRY	1	Sets the maximum number of IPCP config requests that will be sent without receiving a IPCP nak/ack/reject. remoteLocalFlag has no effect. Default: 10
IPCP_TIMEOUT	1	Sets the IPCP retransmission timeout value (in seconds). remoteLocalFlag has no effect. Default: 1 Second

PPP_PROTOCOL

Option Name	Length	Meaning
PPP_SEND_BUFFER_SIZE	2	Length of data buffered by the PPP link layer (but not beyond the end of a packet) before the device driver send function is called. Default: 1 byte

Setting a PPP IP address

The following applies to all of the IP Address optionNames `IPCP_IP_ADDRESS`, `IPCP_DNS_PRI` and `IPCP_DNS_SEC`.

- If `PppSetOption` is not used with the IP Address optionName (default) then the remote will not be allowed to request its IP and/or DNS IP addresses.
- If `PppSetOption` is called with the IP Address `optionNames`, `remoteLocalFlag` `PPP_OPT_ALLOW`, and `optionValuePtr` points to an IP address whose value is 0.0.0.0, then the remote will be allowed to request that its IP/DNS IP address be set to anything except 0.0.0.0.
  sample code
- If `PppSetOption` is called with an IP Address `optionName`, `remoteLocalFlag` `PPP_OPT_ALLOW`, and `optionValuePtr` points to an IP address whose value is not 0.0.0.0, two situations may occur. The remote will be allowed to set its IP/DNS IP address to this value, or will be returned this value, if it requests 0.0.0.0.

PAP (PPP_PAP_PROTOCOL)

Option Name	Length	Meaning
PAP_USERNAME	Any	Sets the username to use with PAP Client Default: NONE
PAP_PASSWORD	Any	Sets the password to use with PAP Default: NONE
PAP_RETRY	1	Sets the maximum number of PAP authentication requests that will be sent without receiving an ACK/NAK remoteLocalFlag has no effect. Default: 10
PAP_TIMEOUT	1	Sets the PAP retransmission timeout value in seconds. remoteLocalFlag has no effect. Default: 3 Seconds

CHAP (PPP_CHAP_PROTOCOL)

Option Name	Length	Meaning
CHAP_USERNAME	Any	Sets the username to use with CHAP Client Default: NONE
CHAP_SECRET	Any	Sets the secret to use with CHAP Default: NONE
CHAP_MSSECRET	Any	Sets the secret to use with MS-CHAPv1 Default: NONE
CHAP_RETRY	1	Sets the maximum number of CHAP challenges that will be sent without receiving a CHAP response remoteLocalFlag has no effect. Default: 10
CHAP_TIMEOUT	1	Sets the CHAP retransmission timeout value in seconds. remoteLocalFlag has no effect. Default: 3 Seconds
CHAP_ALG_ADD	1	Add a CHAP algorithm. You do not need to add CHAP_MD5, because it is automatically added when PPP_CHAP_PROTOCOL is used. You may add CHAP_MSV1 in order to support MS_CHAP version 1.
CHAP_ALG_DEL	1	Delete a CHAP algorithm. You may delete CHAP_MD5 (standard CHAP) or CHAP_MSV1 (MS-CHAP version 1)

optionLength

`optionLength` should be the size of the data type, `optionValuePtr` is pointing to, except for the `PAP_USERNAME, PAP_PASSWORD, CHAP_USERNAME`, and `CHAP_SECRET` options, where `optionValuePtr` points to the first byte of an array, and where `optionLength` is the size of the array `optionValuePtr` is pointing to.

The data types are as follows:

Protocol Level	Option Name	Data Type
PPP_LCP_PROTOCOL	LCP_ADDRCONTROL_COMP	unsigned char
	LCP_PROTOCOL_COMP	unsigned char
	LCP_MAGIC_NUMBER	unsigned char
	LCP_MAX_RECV_UNIT	unsigned short
	LCP_ACCM	unsigned long
	LCP_AUTH_PROTOCOL	unsigned short
	LCP_TERM_RETRY	unsigned char
	LCP_CONFIG_RETRY	unsigned char
	LCP_TIMEOUT	unsigned char
PPP_IPCP_PROTOCOL	IPCP_COMP_PROTOCOL	unsigned short
	IPCP_VJ_SLOTS	unsigned char
	IPCP_IP_ADDRESS	unsigned long
	IPCP_RETRY	unsigned char
	IPCP_TIMEOUT	unsigned char
PPP_PAP_PROTOCOL	PAP_USERNAME	char
	PAP_PASSWORD	char
	PAP_RETRY	unsigned char
	PAP_TIMEOUT	unsigned char
PPP_CHAP_PROTOCOL	CHAP_USERNAME	char
	CHAP_SECRET	char
	CHAP_RETRY	unsigned char
	CHAP_TIMEOUT	unsigned char
PPP_PROTOCOL	PPPSEND_BUFFER_SIZE	unsigned short

SetPppPeerIpAddress()

Function Description

This function is used to set a default remote PPP/SLIP IPaddress. This IP address will be used as the default remote point to point IP address, in case no remote IP address is negotiated with PPP (see PppSetOption()), or for SLIP. If no IP address is set with this function, (and no IP address is negotiated with the remote PPP for PPP), then the local IP address + 1 will be used as the default IP address for the remote PPP (or SLIP) for routing purposes (see GetPppPeerIpAddress()).

Prototype

```
int SetPppPeerIpAddress(int handle, ip_addr ifIpAddress);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	The handle is null, or the device is a LAN device
`EISCONN`	PPP connection is already established

Parameters

Parameter	Description
`handle`	The device handle to update the Peer IP address in
`ifIpAddress`	The IP address to use for routing purposes for the remote PPP system

GetPppEvents()

Function Description

Return bit mapped PPP events.

Prototype

```
unsigned long GetPppEvents(int handle);
```

Return Values

The value returned is a bit mapped value with one or more of the following bits set.

Value	Description
`EINVAL`	The handle is null, or the device is a LAN device
`LL_OPEN_COMPLETE`	PPP Device is ready to accept data from the user.
`LL_OPEN_FAILED`	PPP Device open failed.
`LL_CLOSE_STARTED`	PPP Device has started to close this link.
`LL_CLOSE_COMPLETE`	PPP Device has closed
`LL_LCP_UP`	LCP negotiation has completed.
`LL_PAP_UP`	PAP authentication has completed.
`LL_CHAP_UP`	CHAP authentication has completed.
`LL_LQM_UP`	LQM is enabled on the link.
`LL_LQM_DISABLED`	LQM is disabled on the link.
`LL_LQM_LINK_BAD`	Link quality is bad, user recovery should be attempted.

These events may be used to monitor the status of the PPP connection. The PPP connection should not be used before a `LL_OPEN_COMPLETE` event is received, and the device should not be restarted (after a close) before a `LL_CLOSE_COMPLETE` event is received.

From these events, it is also possible to determine why PPP negotiation failed. For instance, if authentication fails, a `LL_LCP_UP` event is first received indicating that the physical link has been negotiated.

However, if authentication fails, the next event will be `LL_CLOSE_STARTED` and then `LL_CLOSE_COMPLETE` since the link must be closed if negotiation fails. If authentication is successful the events that are received are `LL_LCP_UP, LL_PAP_UP or LL_CHAP_UP` and then `LL_OPEN_COMPLETE`.

Parameters

Parameter	Description
`handle`	The device handle to get PPP events from.

PppSetAuthPriority()

User calls this routine to set priority value for PPP authentication methods. The priority value can be any integer between 1 (with highest priority) and 15 (lowest priority), inclusive. The authenticator will try to negotiate authentication method with the highest priority. If that authentication method is NAK-ed by the peer, it will choose the second one according to the priority value. The less the priority value, the higher priority to use.

The default priority value for the following authentication method is:

Option	Value
`PPP_AUTHMETHOD_MSCHAP_V2`	1
`PPP_AUTHMETHOD_CHAP`	2
`PPP_AUTHMETHOD_PAP`	3
`PPP_AUTHMETHOD_MSCHAP_V1`	4

NOTE

Whenever user calls `PppSetAuthPriority`, the default priority value is gone. User must call `tfPppSetAuthPriority` for all authentication methods if they want to change the default priority sequence.

For example, if user prefers the following priority sequence:`MSCHAPv1-CHAP-PAP`, then user can call this sample code.

Prototype

```
int PppSetAuthPriority(int authMethod, int priorityValue);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	invalid parameter

DHCP/bootp Interface

- ConfGetBootEntry()
- UseBootp()
- UseDhcp()
- UserSetFqdn()
- DhcpConfSet()

ConfGetBootEntry()

Get DHCP/BOOTP configuration parameters.

`struct bootEntry`

This sample code contains values returned from the DHCP server.

Prototype

```
int ConfGetBootEntry(int devhdl, bootEntryPtr bootEntry);
```

Return Values

Value	Description
0	Success

Parameters

Parameter	Description
`devhdl`	Device handle returned by Verix `open()`.
`bootEntry`	Pointer to DHCP/BOOTP configuration structure.

UseBootp()

Use BOOTP on the specified device which must have an Ethernet like device. Call this after `AddInterface` and before `OpenInterface`.

Prototype

```
int UseBootp(int devhdl);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	The interface handle parameter is invalid.
`EMFILE`	Not enough sockets to open the BOOTP client UDP socket.
`ADDRINUSE`	Another socket is already bound to the BOOTP client UDP port.

Parameters

Parameter	Description
`devhdl`	Device handle returned by Verix open().

UseDhcp()

Use DHCP on the specified device which must have be an Ethernet like device. Call this after `AddInterface` and before `OpenInterface`.

Prototype

```
int UseDhcp(int devhdl);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	The interface handle parameter is invalid.
`EMFILE`	Not enough sockets to open the BOOTP client UDP socket
`ADDRINUSE`	Another socket is already bound to the BOOTP client UDP port.

Parameters

Parameter	Description
`devhdl`	Device handle returned by Verix `open()`.

UserSetFqdn()

This function is called by the user application to set the default system-wide FQDN domain name. If FQDN is not set using either `DhcpConfSet` or `DhcpUserSet`, the value configured by this function will be used.

Prototype

```
int UserSetFqdn(const char *fqdnPtr, fqdnLen, int flags);
```

Return Values

Value	Description
0	Success
-1	Failure, see errno

errno	Description
`EINVAL`	Invalid FQDN domain name
`ENOBUFS`	Out of memory

Parameters

Parameter	Description
`fqdnPtr`	Pointer to the domain name char string.
`fqdnLen`	The length of the domain name character string (cannot be greater than 255).
`flags`	Set it to `TM_DHCPF_FQDN_PARTIAL` if the domain name is partial.

DhcpConfSet()

Allows the user to set the DHCP initial state (`INIT`, or `INIT_REBOOT`) prior to the user calling `OpenInterface`. Called by the user when the user wants to specify his/her own Client ID, or suppress the Client ID option, or if the user wants to start in `INIT_REBOOT` state, or if the user wants to specify an IP address in the DISCOVERY phase.

- If user specifies `INIT_REBOOT` state, the Requested IP address needs to be specified as well.
- If user specifies INIT state (i.e. `TM_DHCPF_INIT_REBOOT` not set), optionally the user can specify the IP address, and/or the CLIENT ID option.
- If CLIENT ID is not specified, and not suppressed, the stack will pick a unique CLIENT ID that will be the same across reboots provided that the user uses the same type of configuration and same index.

If the user specifies `TM_DHCPF_FQDN_ENABLE` in flags, then clientIdPtr is used to indicate the DHCP FQDN option domain name. If flags is set to `TM_DHCPF_FQDN_ENABLE` and `clientIdPtr` is NULL, the global FQDN (configured by `UserSetFqdn`) will be used instead. Note that the `TM_DHCPF_FQDN_ENABLE` option cannot be set at the same time as the other flags (except `TM_DHCP_FQDN_PARTIAL`).

Prototype

```
int DhcpConfSet(int devhdl, int flags, ip_addr requestedIp,
unsigned char *clientId, int clientIdLen);
```

Return Values

Value	Description
0	Success

Parameters

Parameter	Description
`devhdl`	Ethernet interface handle
`flags`	0, or a combination of `TM_DHCPF_INIT_REBOOT`, `TM_DHCPF_REQUESTED_IP_ADDRESS`, `TM_DHCPF_SUPPRESS_CLIENT_ID`, `TM_DHCPF_FQDN_ENABLE`, `TM_DHCPF_FQDN_PARTIAL` (`TM_DHCPF_FQDN_PARTIAL` is ignored if clientIdPtr is NULL). The FQDN option must be set separately from other DHCP options (The only other flag allowed with `TM_DHCPF_FQDN_ENABLE` is `TM_DHCPF_FQDN_PARTIAL`).
`requestedIpAddress`	User requested IP address in network byte order

Parameter	Description
`clientIdPtr`	Pointer to client ID. When flags is set to `TM_DHCPF_FQDN_ENABLE`, the FQDN domain name will be stored in this parameter, a character pointer. If it is NULL the global FQDN (in `tvFqdnStruct`) will be used instead.
`clientIdLength`	Length of client ID. When flag is set to `TM_DHCPF_FQDN_ENABLE`, FQDN domain name length.

ARP/Routing Table API

The functions in this section is restricted to applications running GID1 or GID46. The functions have affect the entire stack so these functions should only be used by network configuration applications responsible for the terminal networking.

- AddArpEntry()
- AddDefaultGateway()
- AddProxyArpEntry()
- AddStaticRoute()
- ArpFlush()
- DelArpEntryByIpAddr()
- DelArpEntryByPhysAddr()
- DelDefaultGateway()
- DelProxyArpEntry()
- DelStaticRoute()
- DisablePathMtuDisc()
- GetArpEntryByIpAddr()
- GetArpEntryByPhysAddr()
- GetDefaultGateway()
- RtDestExists()

AddArpEntry()

This function is used add an entry to the ARP cache. This function will allow the user to manipulate the ARP cache beyond standard means. Normally the TCP/IP stack maintains the ARP cache.

Prototype

```
int AddArpEntry(ip_addr arpIpAddress, char *physAddrPtr, int
physAddrLength);
```

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	Bad parameter.

Parameters

Parameter	Description
`arpIpAddress`	The IP address to add to the ARP cache
`physAddrPtr`	A pointer to the character array that contains the physical address
`physAddrLength`	The length of the physical address

AddDefaultGateway()

This function is used to add the system default gateway for all interfaces.

Prototype

```
int AddDefaultGateway(int handle, ip_addr gatewayIpAddress);
```

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
ENOBUFS	Not enough buffer to allocate a routing entry
EALREADY	A default gateway is already in the routing table
EHOSTUNREACH	The gateway is not directly accessible
EINVAL	One of the parameters is bad: the gatewayIpAddress is zero

Parameters

Parameter	Description
handle	Interface handle
gatewayIpAddress	The default gateway IP address in Network Byte Order

AddProxyArpEntry()

Add an entry to the Proxy ARP table for the given IP address. `arpIpAddress` is expected to be in network byte order.

Prototype

```
int AddProxyArpEntry(ip_addr arpIpAddress);
```

Parameters

Parameter	Description
`arpIpAddrss`	IP address on behalf of which the system will reply to ARP requests.

Return Values

Value	Description
0	Success
-1	Error

errno	Description
`ENOERROR`	Success
`EINVAL`	Bad parameter (0 IP address parameter)
`EALREADY`	Entry already in PROXY ARP table
`ENOBUFS`	Couldn't allocate proxy ARP entry

AddStaticRoute()

This function is used to add a route for the interface. It allows packets for a different network to be routed to the interface.

Prototype

```
int AddStaticRoute(int handle, ip_addr destIpAddress,
ip_addr destNetMask, ip_addr gateway, int hops);
```

Parameters

Parameter	Description
`handle`	The interface ID to use to add this routing entry
`destIpAddress`	The IP address to add the route for
`destNetMask`	The net mask for the route
`gateway`	IP address of the gateway for this route.
`hops`	Number of routers between this host and the route.

Return Values

Value	Description
0	Success
-1	Error

errno	Description
`ENOBUFS`	Not enough buffer to allocate a routing entry
`EALREADY`	The route is already in the routing table.
`EHOSTUNREACH`	The gateway is not directly accessible.
`EINVAL`	One of the first 4 parameters is null or 0.

ArpFlush()

TBD

DelArpEntryByIpAddr()

This function is used delete an entry in the ARP cache. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Prototype

```
int DelArpEntryByIpAddr(ip_addr arpIpAddress);
```

Parameters

Parameter	Description
`arpIpAddress`	The IP address to delete in the ARP cache

Returns Values

Value	Description
0	Success
-1	Error

errno	Description
`EINVAL`	Bad parameter.

DelArpEntryByPhysAddr()

This function is used to delete an entry in the ARP cache by looking up the entry by the Physical Address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Prototype

```
int DelArpEntryByPhysAddr(char *physAddrPtr, int
physAddrLength);
```

Parameters

Parameter	Description
`physAddrPtr`	The Physical Address to delete in the ARP cache
`physAddrLength`	The length of the physical address

Return Values

The function will return the following values.

Value	Description
`0`	Success
`-1`	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	Bad parameter.

DelDefaultGateway()

This function is used to delete the system default gateway for all interfaces.

Prototype

```
int DelDefaultGateway(ip_addr gatewayIpAddress);
```

Parameters

Parameter	Description
`gatewayIpAddress`	The default gateway IP address in Network Byte Order

Return Values

The function will return the following values.

Value	Description
`0`	Success
`-1`	Error

Return Values

errno	Description
`EINVAL`	Parameter is 0
`ENOENT`	No default gateway was found

DelProxyArpEntry()

This function deletes an entry from the Proxy ARP table for the given IP address. `arpIpAddress` is expected to be in network byte order.

Prototype

```
int DelProxyArpEntry(ip_addr arpIpAddress);
```

Parameters

Parameter	Description
`arpIpAddrss`	IP address on behalf of which the system will stop replying to ARP requests.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`ENOERROR`	Success
`EINVAL`	Bad parameter (0 IP address parameter)
`ENOENT`	Entry was not in PROXY ARP table.

DelStaticRoute()

This function is used to delete a route from the interface.

Prototype

```
int DelStaticRoute(ip_addr destIpAddress, ip_addr
destNetMask);
```

Parameters

Parameter	Description
`destIpAddress`	The IP Address to add the route for
`destNetMask`	The net mask for the route

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	One of the parameter is 0
`ENOENT`	No routing enry for this route in the routing table.
`EPERM`	Cannot delete an ARP entry with DelStaticRoute

DisablePathMtuDisc()

This function is used to disable path MTU discovery for a given route. If pathMtu is zero, or bigger than the outgoing device IP MTU, then we will default the route IP MTU to the outgoing device IP MTU; otherwise we will set the route IP MTU with the passed parameter value.

Prototype

```
int DisablePathMtuDisc(ip_addr destIpAddress, unsigned short pathMtu);
```

Parameters

Parameter	Description
`destIpAddress`	The IP address destination on which route we want to disable path MTU discovery.
`pathMtu`	New fixed IP MTU. If zero, we default to the device IP MTU.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	DestIpAddress parameter is zero.
`EPERM`	Route is direct. No path MTU discovery is ever going to take place.
`EHOSTUNREACH`	No route to destination IP address.
`ENOBUFS`	Not enough memory to allocate new routing entry.

GetArpEntryByIpAddr()

This function is used retrieve an entry from the ARP cache by looking up the entry by the IP address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally the TCP/IP stack maintains the ARP cache.

Prototype

```
int GetArpEntryByIpAddr(ip_addr arpIpAddress, char
*physAddrPtr, int physAddrLength);
```

Parameters

Parameter	Description
`arpIpAddress`	The IP address to use to lookup the entry by
`physAddrPtr`	The Pointer to the buffer where to store the physical address
`physAddrLength`	The length of the physical address buffer

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	Bad parameter
`ENOENT`	No ARP entry found with this IP address

GetArpEntryByPhysAddr()

This function is used retrieve an entry from the ARP cache by looking up the entry by the Physical Address. This function will allow the user to manipulate the ARP cache beyond standard means. Normally, the TCP/IP stack maintains the ARP cache.

Prototype

```
int GetArpEntryByPhysAddr(char *physAddrPtr, int
physAddrLen, ip_addr *arpIpAddressPtr);
```

Parameters

Parameter	Description
`physAddrPtr`	The Physical Address to lookup the entry in the ARP cache.
`physAddrLen`	The length of the physical address
`arpIpAddressPtr`	The location where to store the IP address of the matching physical entry

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EINVAL`	Bad parameter
`ENOENT`	No ARP entry found with this physical address

GetDefaultGateway()

This function is called from the socket interface to get the default gateway IP address. The default gateway IP address will be stored in network byte order.

Prototype

```
int GetDefaultGateway(ip_addr *gwayIpAddPtr);
```

Parameters

Parameter	Description
`gwayIpAddrPtr`	Pointer to store gateway IP address into

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`ENOERROR`	Success
`EINVAL`	Bad parameter
`ENOENT`	No default gateway

RtDestExists()

Find out whether a route to a destination, given by the pair destination IP address and destination IP network mask, exists.

Prototype

```
int RtDestExists(ip_addr destIpAddress, ip_addr
destNetMask);
```

Parameters

Parameter	Description
`destIpAddress`	Destination IP address
`destNetMask`	Destination IP Network Mask

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
`EHOSTUNREACH`	No route to destination.

Network Interface API

The functions in this section is restricted to applications running GID1 or GID46. The functions have affect the entire stack so these functions should only be used by network configuration applications responsible for the terminal networking.

- net_addif()
- net_delif()
- net_stopif()
- net_startif()
- openSockets()
- openaux()
- closeaux()
- AddInterface()
- CloseInterface()
- GetBroadcastAddress()
- GetIpAddress()
- GetNetMask()
- InterfaceSetOptions()
- OpenInterface()
- SetIfMtu()

net_addif()

This function returns a network interface handle given a Verix device handle and a task/thread ID. The task/thread specified will own the network interface handle.

The following functions among other take the interface handle as a parameter: `net_delif`, `AddInterface`, `OpenInterface`, and `CloseInterface`.

Prototype

```
int net_addif(int devhdl, int task_id);
```

Parameters

Parameter	Description
`devhdl`	Verix device handle such as the handle returns by open("/DEV/COM3",0).
`task_id`	Task or thread id returned from `run()` or `run_thread()`.

Return Values

Value	Description
>= 0	Network interface handle
-1	An error occurred.

errno	Description
`EBADF`	Invalid handle
`EINVAL`	Invalid task ID

net_delif()

This function deallocates the specified interface handle.

Prototype

```
int net_delif(int inthdl);
```

-1 `errno` set to EPERM: Session key not set or set by a different task.

Parameters

Parameter	Description
`inthdl`	Network interface handle returned by net_addif().

Return Values

Value	Description
0	Success
-1	An error occurred.

errno	Description
`EINVAL`	Invalid handle

net_stopif()

This function stops the flow of packets on this interface.

Socket functions such as send, recv, etc. will return -1 and set errno to `EINTDISABLED` if there is no active network interface.

Prototype

```
int net_stopif(int inthdl);
```

Parameters

Parameter	Description
`inthdl`	Network interface handle returned by `net_addif()`.

Return Values

The function will return the following values.

Value	Description
0	Success.
-1	An error occurred.

If the return value is -1, errno will be set to one of the following values.

errno	Description
EACCES	Caller not in GID1 or GID46.
EBADF	Invalid interface handle.

net_startif()

This functions starts the flow of packets on this interface. This reverses the effect of calling `net_stopif()`. When an interface is created by `net_addif()`, it is always enabled so it is not normally necessary to call `net_startif()`.

Prototype

```
int net_startif(int inthdl);
```

Parameters

Parameter	Description
`inthdl`	Network interface handle returned by `net_addif()`.

Return Values

The function will return the following values.

Value	Description
0	Success.
-1	An error occurred.

If the return value is -1, errno will be set to one of the following values.

errno	Description
EACCES	Caller not in GID1 or GID46.
EBADF	Invalid interface handle.

openSockets()

Returns the number of open sockets.

Prototype `int openSockets(void);`

Parameters None.

Return Values The function returns the number of open sockets.

openaux()

This function returns a Verix device handle for a limited device to be used for link monitoring. The parameter is the handle of the primary network device. The caller must own `devhdl`.

The only functions that use this handle are: `closeaux` and `get_port_status` for COM devices and `get_enet_status` for USB Ethernet and WiFi devices.

For example,

```
com3hdl = open("/DEV/COM3", 0);
com3stat = openaux(com3hdl);
```

Prototype

```
int openaux(int devhdl);
```

Parameters

Parameter	Description
`devhdl`	Verix device handle returned by call to open().

Return Values

Value	Description
>0	Success. The return value is a device handle.
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
EBUSY	The caller does not own the device. Possibly owned by another task.

closeaux()

This function closes the device status device specified by the handle. The call must own the handle. The handle is the value returned by calling `openaux()`.

Prototype

```
int closeaux(int stshdl);
```

Parameters

Parameter	Description
`stshdl`	Status handle of device. The handle is returned from `openaux()`.

Return Values

Value	Description
0	Sucess
-1	Error

If the return value is -1, errno will be set to one of the following values.

errno	Description
EBADF	The caller does not own the device.

AddInterface()

Bind Verix device to IP stack. This must be done before setting link layer options such as DHCP. The stack will take control of reading and writing this device. Link status changes will be reported by the specified Verix event bit. The interface handle is the value returned by `net_addif()`.

Prototype

```
int AddInterface(int inthdl, int linktype, long event);
```

Parameters

Parameter	Description
`inthdl`	Device handle returned by Verix open().
`linktype`	Link layer type. LL_ETHERNET or LL_PPP.
`event`	Verix event bit

Values for the linktype parameter.

Value	Description
`LL_ETHERNET`	Ethernet or Ethernet-like link layer
`LL_PPP`	PPP link layer

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	One of the parameters is invalid
`EALREADY`	Device has already been added.
`ENOBUFS`	Not enough buffers to allocate a device entry.

CloseInterface()

Unbind Verix device from IP stack.

Prototype

```
int CloseInterface(int inthdl);
```

Parameters

Parameter	Description
`inthdl`	Device handle returned by Verix `open()`.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	Parameter is invalid
`EALREADY`	The device is already closed
`EINPROGRESS`	If the connection is in the process of closing the connection (as in PPP). The user does not need to do anything, and will be notified by the PPP link layer when the interface is actually closed.

GetBroadcastAddress()

This function is used to retrieve the broadcast address for an interface. The broadcast address is automatically calculated from the IP address and netmask combination

Prototype

```
int GetBroadcastAddress(int inthdl, ip_addr *broadcastAddr);
```

Parameters

Parameter	Description
`inthdl`	The device driver handle return by Verix `open()`.
`broadcastAddr`	The pointer to the area that the function will store the broadcast address into.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
EINVAL	Bad parameter.
EACCES	Invalid buffer pointer.

GetIpAddress()

This function is used to get the IP address of an interface.

Prototype

```
int GetIpAddress(int inthdl, ip_addr *ipAddr);
```

Parameters

Parameter	Description
`inthdl`	The device driver handle return by Verix `open()`.
`ipAddr`	The pointer to the area that the function will store the IP address into.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	Bad parameter.
`EACCES`	Invalid buffer pointer.

GetNetMask()

This function is used to get the Net Mask from a given interface.

Prototype

```
int GetNetMask(int inthdl, ip_addr *netmask);
```

Parameters

Parameter	Description
`inthdl`	The device driver handle return by Verix `open()`.
`netMaskPtr`	A pointer to a location where to store the Net Mask upon function completion.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	Bad parameter
`EACCES`	Invalid buffer pointer.

InterfaceSetOptions()

Configure interface options. `optValue` points to a variable of type as described below. `optLen` contains the size of that variable.

Option Name	Description
`DEV_OPTIONS_BOOT_TIMEOUT`	Base number of seconds for a BOOTP/DHCP request timeout. BOOTP/DHCP timeouts increase with each retransmission, so if this value is set to two seconds, the first timeout will be two seconds, the second will be four seconds, the third will be eight seconds, etc. Default: 4 seconds Data Type unsigned char
`DEV_OPTIONS_BOOT_RETRIES`	Total number of BOOTP/DHCP requests to send without receiving a response from a BOOTP/ DHCP server. Default: 6. Data Type unsigned char
`DEV_OPTIONS_NO_DHCP_RELEASE`	When set to 1, this option disables DHCP release messages when the specified interface or DHCP is stopped. When set to 0 (default), DHCP release messages are enabled for these events. The data type for this option is unsigned char.
`DEV_OPTIONS_BOOT_ARP_RETRIES`	This specifies the numer of ARP probe retries before configuring a DHCP/BOOTP address. When set to -1, ARP probes are disabled. When set to 0 (default), the default of `TM_MAX_PROBE` is used. The data type for this option is int.
`DEV_OPTIONS_BOOT_ARP_INTVL`	This specifies the interval, in seconds, between ARP probes (when enabled) prior to configuring a DHCP/BOOTP address. When set to 0 (default), the default value of `TM_PROBE_INTERVAL` / `TM_UL`(1000) is used. The data type for this option is unsigned char.
`DEV_OPTIONS_BOOT_ARP_TIMEOUT`	This specifies the number of seconds to wait after sending the first ARP probe / ARP request before finishing DHCP/BOOTP address configuration. When set to 0 (default), the default value is such that the timeout will occur after the last probe has been sent after the interval time has elapsed. The data type for this option is unsigned char.

Option Name	Description
DEV_OPTIONS_BOOT_PK_HOST_NM	When set to 1, this option instructs the DHCP client to use the host name as provided by the DHCP server when building the host name option in the DHCP request. If the server did not send a host name option or if this option is set to 0 (default), then the DHCP client will use the host name as provided by the user instead. Default value for this option is 0.The data type for this option is unsigned char.
DEV_OPTIONS_BOOT_PK_DOMAIN_NM	When set to 1, this option instructs the DHCP client to use the domain name as provided by the DHCP server when building the FQDN option in the DHCP request. In that case the host name in the FQDN option will be picked according to the TM_DEV_OPTIONS_BOOT_PK_HOST_NM option (see above). If the server did not send a domain name option or if this option is set to 0 (default), then the DHCP client will use the FQDN as provided by the user instead. Default value for this option is 0.The data type for this option is unsigned char.
DEV_OPTIONS_FORWARDING	When set to 1, IP forwarding is enabled on the specified interface. When set to 0, IP forwarding is disabled on the specified interface. The data type for this option is unsigned char.
DEV_OPTIONS_FILTER	Determines whether Filtering is enabled or disabled for the given interface. Default: 0 (disabled) Data Type: unsigned char
DEV_OPTIONS_NO_GRAT_ARP	Determines whether a gratuitous ARP is sent when the given interface is opened. Must be set before the interface is opened. Default: 0 (gratuitous ARP is sent) Data Type: unsigned char

Prototype

```
int InterfaceSetOptions(int inthdl, int optName, void
*optVal, int optLen);
```

Parameters

Parameter	Description
inthdl	The device driver handle return by Verix open().
optName	The option to set. See above.
optVal	The pointer to a user variable into which the option value is set. User variable is of data type described above.
optLen	Size of the user variable, which is the size of the option data type.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	Invalid `optionName`, or invalid option length for option, or invalid option value for option.
`EACCES`	Invalid buffer pointer.

OpenInterface()

This function is used to configure an interface with an IP address, and net mask (either supernet or subnet). It must be called before the interface can be used.

Note `UseDhcp()` must be called before this function if DHCP is desired. Also the `ipAddr` and `netmask` must be set to 0. Similary for `UseBootp()`.

Prototype

```
int OpenInterface(int inthdl, ip_addr ipAddr, ip_addr, netmask);
```

Parameters

Parameter	Description
`inthdl`	The device driver handle return by Verix open().
`ipAddr`	IP address.
`netmask`	Network mask.

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EADDRNOTAVAIL`	Attempt to configure the device with a broadcast address.
`EINPROGRESS`	OpenInterface call has not completed. This error will be returned for a DHCP or BOOTP configuration for example.
`ENOBUFS`	Not enough memory to complete operation
`EINVAL`	Bad parameter. Note that a zero IP address is allowed for Ethernet if the BOOTP or DHCP flag is on, or for PPP, otherwise a EINVAL errorCode is returned.
`EALREADY`	A previous call to `tfOpenInterface` has not yet completed.
`EPERM`	User attempted to configure an IP address via DHCP (respectively BOOTP) without having called `UseDhcp` (respectively `UseBootp`) successfully first.
`EMFILE`	Not enough sockets to open the BOOTP client UDP socket (BOOTP or DHCP configurations only.)
`ADDRINUSE`	Another socket is already bound to the BOOTP client UDP port. (BOOTP or DHCP configurations only.)

Value	Description
`ETIMEDOUT`	DHCP or BOOTP request timed out
`EAGAIN`	A PPP session is currently closing. Call `OpenInterface` again after receiving notification that the previous session has ended.

SetIfMtu()

This function is used to set the Maximum Transmission Unit (MTU) for a device. For Ethernet and PPP, this is typically set to 1500 bytes. The link MTU is always the size of the largest IP packet which can be sent unfragmented over the link, which is the maximum link-layer frame size minus any link-layer header and trailer overhead. For PPP and Ethernet, this value can be changed via Path MTU Discovery to allow larger frames and to prevent IP datagram fragmentation.

Prototype

```
int SetIfMtu(int inthdl, int mtu);
```

Parameters

Parameter	Description
`inthdl`	The device driver handle return by Verix `open()`.
`mtu`	The Maximum Transmission Unit for a device

Return Values

The function will return the following values.

Value	Description
0	Success
-1	Error

If the return value is -1, errno will be set to one of the following values.

Value	Description
`EINVAL`	One of the parameter is null or 0.

Verix Device Interface API

Verix device drivers for network devices have various features that are useful for managing the device.

Ethernet Link Layer

Ethernet devices include Ethernet and WiFi devices. WiFi devices include wireless channels and wireless authentication protocols which are not needed for wired Ethernet.

USB Ethernet

The Verix V AX88772 device driver is access via the device name "/DEV/ETH1".

- open()
- close()
- read()
- write()
- get_enet_MAC()
- get_enet_status()
- set_enet_rx_control()
- int get_enet_event()

open()

This is the standard Verix V `open()` with the usual return codes. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details. There are no stack related changes.

Prototype

```
int open(const char *pathname, int flags);
```

close()

This is the standard Verix V `close()` with the usual return codes. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details. There are no stack related changes.

Prototype

```
int close(int handle)
```

read()

This is the standard Verix V `read()` with the usual return codes. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details.

When the device is handed over to the stack, the stack will read from the device.

Prototype

```
int read(int handle, char *buffer, int length);
```

write()

This is the standard Verix V `write()` with the usual return codes. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details.

When the device is handed over to the stack, the stack will read from the device.

Prototype

```
int write(int handle, char *buffer, int length);
```

get_enet_MAC()

This function exists today for Ethernet and WiFi devices. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details.

Prototype

```
int get_enet_MAC(int handle, char *MACbuf6);
```

get_enet_status()

This function exists today for Ethernet devices. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details.

Prototype

```
int get_enet_status(int handle, char *status4);
```

set_enet_rx_control()

This function exists today for Ethernet devices. See the Getting Started with the Verix EOS Developers Suite (VPN - DOC00307) for details.

Prototype

```
int set_enet_rx_control(int handle, int rx_ctrl);
```

int get_enet_event()

This function returns Ethernet link status changes and network DHCP status changes. If the return value is 0, there is no valid data in the event4 array. If the return value is 4, there is.

Prototype

```
int get_enet_event(int handle, char *event4);
```

Parameters

Parameter	Description
`handle`	Value returned by call to Verix V `open()`.
`event4`	Array of 4 bytes. The first byte is the type of event.

Byte 0	Byte 1	Byte 2	Byte3	Description
0	NA	NA	NA	Link down
1	link speed	NA	NA	Link up
2	DHCP event	NA	NA	DHCP event

Link Speed	Description
0	10 Mbits/half duplex
1	100 Mbits/half duplex
2	10 Mbits/full duplex
3	100 Mbits/full duplex

DHCP Event	Description
0	Success
`EHOSTDOWN`	Renewal failed because the DHCP renewal retrans mechanism timed out (and the server never responded).
`EPERM`	Renewal failed because a NAK was received.
`ETIMEDOUT`	Rebind failed and the address expired before the rebind completed.

Return Values

Value	Description
0	No event. event4 does not contain valid data.
4	An event is in event4.

USB WiFi

PPP Link Layer

Serial port COM1/ COM2

Dial modem COM3

GPRS modem on COM2

USB EVDO

Packet Capture

The stack will output to the system log network packets in packet capture (PCAP) format. Since the system log is designed for printable ASCII data and PCAP is binary data, the PCAP data is converted to ASCII hex characters. A program must post-process the syslog data to extract the PCAP data into a binary file for WireShark to read.

Sample output in the syslog. The lines between <PCAP_B> and <PCAP_E> must contain hex characters (0-9,A-F) which represent one PCAP record. There may contain more than one line, if needed.

```
<PCAP_B>
001122334455…
DDEEFF…
<PCAP_E>
```

To eliminate the risk of accidentally revealing card data, this feature will limit the length of each packet to the first 100 bytes. This should include enough of the network packet headers for troubleshooting and debugging.

The GID 1 CONFIG.SYS variable *PCAP will control this feature.

*PCAP=1 The stack writes PCAP records to the system log.

If *PCAP is another other value or not present, the stack does not write PCAP records to the system log.

Secure Sockets Layer (SSL)

VxEOS SSL will be a port of OpenSSL 0.9.8k with changes required to interface with Verix V security features such as hardware random number generator. For Trident, hardware crypto accelerators for AES and RSA will be interfaced to OpenSSL.

While the primary use for OpenSSL is for SSL over TCP for transactions, it will also be used for IP downloads. WPA/WPA2 Enterprise supplicants require SSL over Ethernet (no IP, no TCP) support also so the library must also support this mode of operation.

OpenSSL API

Rarely used, proprietary, or flawed crypto algorithms such as blowfish, Camellia, CAST, Diffie-Hellman, DSA, elliptic curve, IDEA, Kerberos, mdc2, ripemd, rc5, ssl1, and ssl2 are not included.

Include files

Crypto functions

aes.h	evp.h	pkcs7.h
asn1.h	hmac.h	rand.h
asn1_mac.h	md2.h	rc2.h
asn1t.h	md4.h	rc4.h
bio.h	md5.h	rsa.h
crypto.h	ocsp.h	sha.h
des.h	pem2.h	x509.h
engine.h	pem.h	x509v3.h
err.h	pkcs12.h	x509_vfy.h

SSL functions

The following is from the ssl man page.

Currently the OpenSSL ssl library provides the following C header files containing the prototypes for the data structures and and functions:

`ssl.h`

That's the common header file for the SSL/TLS API. Include it into your program to make the API of the ssl library available. It internally includes both more private SSL headers and headers from the crypto library. Whenever you need hard-core details on the internals of the SSL API, look inside this header file.

`ssl3.h`

That's the sub header file dealing with the SSLv3 protocol only. Usually you don't have to include it explicitly because it's already included by `ssl.h`.

`ssl23.h`

That's the sub header file dealing with the combined use of the SSLv2 and SSLv3 protocols. Usually you don't have to include it explicitly because it's already included by `ssl.h`.

`tls1.h`

That's the sub header file dealing with the TLSv1 protocol only. Usually you don't have to include it explicitly because it's already included by `ssl.h`.

Libraries

The crypto and SSL libraries will be built as Verix V shared libraries.

`libcrypto.lib` — crypto shared library

`libssl.lib` — ssl shared library

`libcrypto.o` — stub functions. Applications link with this file.

`libssl.o` — stub functions. Applications link with this file.

Crypto functions

See the OpenSSL crypto library man pages for details.

http://www.openssl.org/docs/crypto/crypto.html

SSL functions

See the OpenSSL SSL man page for details.

http://www.openssl.org/docs/ssl/ssl.html

VxEOS SSL versus VeriFone SSL library

The VxEOS SSL library is a relatively straight forward port of OpenSSL to Verix V. The IP stack socket functions do not interface to the SSL library so the SSL library must be called to use SSL. OpenSSL is designed to operate in this fashion where the SSL library calls the socket functions as needed rather than the socket functions calling OpenSSL.

Test Results

Various tests using a Vx570 running Treck IP/VxEOS SSL over USB Ethernet.

These are not transactions times, only SSL connection establishment times.

Connect to dummy local server

The test ssl server in the HNL office is running WinXP with stunnel and echo server. Using a local test server versus Vital eliminates host and network delays. The dummy certificates use RSA 1024 bit keys.

There are two SSL client programs that connect to the test SSL server.

Lab test program using library IP/SSL

client1.c program using the VxEOS IP/OpenSSL

Times are from TCP SYN to Change Cipher.

Lab test program using library IP/SSL w/o fast malloc:

859 milliseconds

Lab test program using library IP/SSL w/ fast malloc:

405 milliseconds

client1.c test using Treck IP/openSSL w/ fast malloc:

249 milliseconds

Connect to ssl2.vitalps.net

Vital certs

root cert: 2048 bit RSA key

inter cert: 1024 bit RSA key

server cert: 1024 bit RSA key

vital422b.pcap

WinXP PC in HNL connecting to ssl2.vitalps.net:5003. From TCP SYN to Change Cipher Spec is 460 ms. Since a PC can do crypto very fast, assume 460 msis mostly host and network delays.

vital422570.pcap

Vx570 in HNL running VxEOS stack and SSL connecting to ssl2.vitalps.net:5003 using fast malloc. From TCP SYN to Change Cipher Spec is 979 ms. The 519 ms increase of for 570 processing of the three certificates.

Network Interface

GID1 CONFIG.SYS Configuration

*SOCKET

*SOCKET allocates the maximum number of open sockets in the IP stack. The allocation takes place only on restart.

```
Min 4, default 8, Max 32
```

*HEAP

*HEAP allocates the maximum amount of system heap (not application heap) for miscellaneous IP stack variables and USB stack variables.

net_addif()

This function returns a network interface handle given a Verix device handle and a task/thread ID. The task/thread specified will own the network interface handle.

The following functions take the interface handle as a parameters:

- `AddInterface`
- `OpenInterface`
- `CloseInterface`

Prototype

```
int net_addif(int devhdl, int task_id);
```

Parameters

`devhdl`	”Verix device handle such as the handle returns by open(“”/DEV/COM3””,0).”
`task_id`	Task or thread id returned from `run()` or `run_thread()`.

Returns

`>= 0`	Network interface handle
`-1`	An error occurred.

net_delif()

This function deallocates the specified interface handle.

Prototype

```
int net_delif(int inthdl);
```

Parameters

`inthdl`	Network interface handle returned by `net_addif()`.

Returns

`0`	Success
`-1`	An error occurred.

errno	Description
`EINVAL`	Invalid handle

openaux()

This function returns a Verix device handle for a limited device to be used for link monitoring. The parameter is the handle of the primary network device. The caller must own devhdl.

The only functions that use this handle are: closeaux and get_port_status for COM devices and get_enet_status for USB Ethernet and WiFi devices.

For example,

```
com3hdl = open("/DEV/COM3", 0);
com3stat = openaux(com3hdl);
```

Prototype

```
int openaux(int devhdl);
```

Parameters

`devhdl`	Verix device handle returned by call to `open()`.

Returns

`0`	Sucess
`-1`	Error

errno	Description
`>0`	Device status handle.
`EBUSY`	The caller does not own the device. Possibly owned by another task.

closeaux()

This function closes the device status device specified by the handle. The call must own the handle. The handle is the value returned by calling `openaux()`.

Prototype

```
int closeaux(int stshdl);
```

Parameters

`stshdl`	Status handle of device. The handle is returned from `openaux()`.

Returns

`0`	Sucess
`-1`	Error

errno	Description
EBADF	The caller does not own the device.

Configuration Management

New SW components are being introduced as part of Verix EOS Volume II SW Architecture. Having separate modules with specific responsibilities, moving from static dependencies to loading modules at run-time, all facilities future enhancements with minimal impact to SW certifications.

While implementing new components enhances current SW architecture, it adds overhead when trying to handle configuration files and parameters for each specific module. To simplify this procedure, parameters will be consolidated based on their use case. Internally Verix EOS Volume II components will retrieve specific parameters from the consolidated set.

This document describes the standardized “Configuration File Format” to use across all Verix EOS Volume II components requiring external configuration/ customization.

Customers will be able to setup all parameters on a single file (i.e. GPRS parameters). In some cases, same parameters will take different values based on specific criteria (i.e. based on carriers available). The format described below provides abstraction to design same parameters driven on specific conditions.

Internally Verix EOS Volume II components are responsible for retrieving those parameters and taking proper actions. Editing fields on the UI require some rules regarding type, size limits and verifications. Each parameter is meant to affect specific component(s). All criteria regarding how Verix EOS Volume II manipulates customer’s parameters will be encapsulated within Verix EOS Volume II space (GID 46).

Files editable by customers and internal Verix EOS Volume II files follow similar format, but they are designed to handle different details of the same information. This document presents a common and generic template to use for both types of files. Content and usage within Verix EOS Volume II is outside the scope of this document.

Finally, another consideration within API noted below, is the concept of “Factory Default” parameters. Verix EOS Volume II components will be released with default settings. Customers will have the ability to overwrite some / all parameters during deployment.

Configuration File Format

The configuration file can be described be described as a collection of one or more tables and each table consisting of one or more records. Each record may optionally consist of one or more attribute.

View Sample Configuration File

A configuration file may contain comment and blank lines to make the file self documenting and readable. A comment line starts with the character # (0x23) may be followed by any character and terminated by end of line (CRLF 0x0D0A). Similarly a blank line is any number (zero or more) tabs (0x09) and / or space (0x20) characters terminated with an end of line.

The configuration file supports the notion of environment variables which are evaluated at file load, i.e., when the file is parsed. The environment variable is matched against a regular expression and if there is a match the records specified under the expression are evaluated.

View Regular Expresison Sample

NOTE

Regular expression restrictions:

- Conditional segments cannot be nested.
- Conditional segments can enclose:
  - Any set of parameter definitions outside a table.
  - A complete table.
  - A complete record.
  - Any set of attributes defined within a table record.

Regular Expression Support

The expression support in the configuration file is in the following syntax: View Regular Expression Support Syntax Sample.

Formal Specification of Configuration File Format

The configuration file formal specification in ABNF specifies the configuration file format using Augmented Backus-Naur Format (ABNF). See **RFC 5234** for specification of ABNF.

View the Configuration File Formal Specification in ABNF.

Internal and External Configuration Files

Samples provided above, are good references to differentiate which information must be provided by Verix EOS Volume II customers and which information should be encapsulated within Verix EOS Volume II .

Details inside “wifi_params” (See Sample Configuration File) should be configured during deployment; that information should be provided via configuration file external to Verix EOS Volume II .

Details inside “wifi_params_display” (See Sample Configuration File) are for Verix EOS Volume II internal consumption only; that information should be predefined via configuration file but private within Verix EOS Volume II space.

Verix EOS Volume II Configuration Files

These files are designed and generated by Verix EOS Volume II team. They will be included as part of the default Verix EOS Volume II installation package and will reside on GID46, visible to Verix EOS Volume II components only.

Because total space precedes file being readable, for all Verix EOS Volume II setup files concise notation should be used. Short and meaningful comments are still encouraged.

Factory Default Settings

EveryVerix EOS Volume II component relying on Configuration Files will include its 'factory default' settings. These settings allow every component to operate even if no custom settings are provided. These values also become the recommended Verix EOS Volume II settings for specific components (i.e. device timeouts).

Following sample described on Sample Configuration File, following configuration file allows a Wi-Fi device to connect to any open/unsecure network. View Sample Factory Default Settings.

Metadata File

This file will be used by Verix EOS Volume II components to manipulate dynamic settings. This sample shows how to build an UI to edit parameters noted on Sample Factory Default Settings, also API required to set this value with the corresponding Device Driver. View Sample Metadata.

External Configuration Files

This file allows customers to provide specific configuration values. Following filename, extension and format defined by Verix EOS Volume II , this file will be provided during installation on GID1. Verix EOS Volume II will read its default settings and will overwrite them with the customer's settings. View Sample Customer's Configuration File.

Naming Convention and Location

Location	Filename	Description
N:46	<basename>.CFG	Default configuration file. This file contains the name and value of configurable parameters.
I:1	<basename>.CFG	User's configuration file. This file contains custom settings defined during deployment. Values on this file will overwrite those configured on N:46/<basename>.CFG. Changes via API "update" will be reflected on this file.
N:46	<basename>.MTD	This file contains the name and editable information to configure parameters. This file is primarily for Verix EOS Volume II processing. No run-time changes should happen to this file.

Configuration API

Single API library will be available for all Verix EOS Volume II SW components to use. INI Processing Class will be available, by including `DDI_Tools.h` header file and linking static library `INI_TOOLS.a`.

INI Parser Class Definition

The `DDI_INI_TOOLS` class provides a set of methods for loading and parsing a Configuration File. See INI Parser Code.

DDI_INI_TOOLS::DDI_INI_TOOLS()

Prototype `void DDI_INI_TOOLS(char * INI_file_path, GENERAL_TOOLS * general);`

Description Constructor: Loads the INI file into memory.

Parameters

In	`INI_file_path`	TBD
In	`general`	TBD

Return Values

Not Applicable

NOTE

This API is not applicable to CommEngine.

DDI_INI_TOOLS::DDI_INI_TOOLS()

Prototype

```
void DDI_INI_TOOLS (    char * INI_file_name,
                        char * INI_Delta_File_name )
```

Description

Constructor: Loads the INI file into memory. The constructor does not have to provide the GENERAL_TOOLS instance.

Parameters

In	INI_file_name	The full INI path is provided (including the drive ID).
In	INI_Delta_File_name	Delta file path is provided for INI updates.

Return Values

Not Applicable

DDI_INI_TOOLS::DDI_INI_TOOLS()

Prototype

```
void DDI_INI_TOOLS(char * INI_Delta_File_name )
```

Description

Constructor: Loads the INI file into memory. If no base INI file is specified, then all updates are carried out on the delta file.

Parameters

In	`INI_Delta_File_name`	Delta file path is provided for INI updates.

Return Values

Not Applicable

NOTE

Refer to References for more information.

DDI_INI_TOOLS::load_parse_ini_file()

Prototype

```
int load_parse_ini_file( char paramList[][30], int totalParams )
```

Description

Parses the INI file parameters. All parameters are parsed for performance reasons.

Parameters

In	`paramList`	The list of parameters supported by the driver is passed in the 2nd array.
In	`totalParam`	TBD

Return Vaues

Int	`paramList`	Parser error value as noted in INI Parser Class Definition.
In	`totalParam`	TBD

NOTE

A parameter is referenced by its index within the 2nd array. This ensures acceptable performance when referencing INI parameters.

This API is not applicable to CommEngine.

DDI_INI_TOOLS::load_parse_ini_file()

Prototype

```
int load_parse_ini_file( void )
```

Description

Parses the INI file parameters. The following method parses tables from an INI file without having to provide the list of INI text parameters. INI tables are used to describe the INI parameters that are updateable via separate API.

Parameters

None

Return Values

Int `paramList` Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::getTableRecord()

Prototype

```
char * getTableRecord( char * tableName, int recordNumber )
```

Description

Retrieves a name of a record from an INI table with from a table named in parameter 'tableName'.

Parameters

In	`tableName`	Name of the table
In	`recordNumber`	Specifies which record in the table to retrieve

Return Values

`NULL`	If the record number is invalid.
`not NULL`	Pointer to a string containing the name of the record.

DDI_INI_TOOLS::update_ini_field()

Prototype

```
int update_ini_field( char * paramName, char * paramText )
```

Description

Updates a value of an INI parameter. This only updates the parameter in memory and does not save it.

Parameters

In	`paramName`	Name of the INI parameter. Case sensitive.
In	`paramText`	Text to set the field to.

Return Values

`int`	Parser error value as noted in INI Parser Class Definition.

NOTE

If `paramName` field format contains a '.', i.e. `NEW_TABLE.RECORD_NAME` then the table name 'NEW_TABLE' will be added to the list of tables, and `RECORD_NAME` added to the list of records for that table.

DDI_INI_TOOLS::commit_updates()

Prototype

```
int commit_updates( void )
```

Description

Saves all the updates to an INI file made via the '`update_ini_field`' method.

Parameters

Not Applicable

Return Values

int	Parser error value as noted in INI Parser Class Definition.

NOTE

Update is to be atomic. Creates a new update file, so that there can be a rollback to the previous configuration change, if the configuration update fails.

DDI_INI_TOOLS::clear_updates()

Prototype

```
void commit_updates( void )
```

Description

Clear the last set of INI updates (i.e. user hits cancel key)

Paramters

Not Applicable

Return Values

None

DDI_INI_TOOLS::reset_defaults()

Prototype

```
int reset_defaults( void )
```

Description

Clear the last set of INI updates (i.e. user hits cancel key)

Parameters

Not Applicable

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::set_ini_environment_text()

Prototype

```
int set_ini_environment_text( char * variableName, char *
variableText)
```

Description

Sets INI environment text. INI environment variables are used to determine whether conditional INI parameters apply, i.e. Network specific parameters for CDMA.

Parameters

In	`variableName`	TBD
In	`variableText`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::set_ini_environment_integer()

Prototype

```
int set_ini_environment_integer( char * variableName, unsigned
long variableInt )
```

Description

Sets INI environment integer value.

Parameters

In	`variableName`	TBD
In	`variableText`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::read_ini_param_text()

Prototype

```
int read_ini_param_text( char * paramName, char * paramText, int
* paramTextLen )
```

Description

Retrieves a text string associated with an INI parameter name.

Parameters

In	`paramName`	TBD
In	`paramText`	TBD
In	`paramTextLen`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::read_ini_param_integer()

Prototype

```
int read_ini_param_integer( char * paramName, unsigned long *
paramInt )
```

Description

Retrieves an integer value associated with an INI parameter name.

Parameters

In	`variableName`	TBD
In	`variableInt`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::read_ini_param_text()

Prototype

```
int read_ini_param_text( int paramID, char * paramText, int *
paramTextLen )
```

Description

Retrieves a text string associated with an INI parameter ID.

Parameters

In	`paramID`	TBD
In	`paramText`	TBD
In	`paramTextLen`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

NOTE

Parameter ID's are used by the DDI driver instead of parameter names for enhanced performance.

DDI_INI_TOOLS::read_ini_param_integer()

Prototype

```
int read_ini_param_integer( int paramID, unsigned long *
paramInt )
```

Description

Retrieves an integer value associated with an INI parameter ID.

Parameters

In	`paramID`	TBD
In	`variableInt`	TBD

Return Values

int	Parser error value as noted in INI Parser Class Definition.

NOTE

Parameter ID's are used by the DDI driver instead of parameter names for enhanced performance.

DDI_INI_TOOLS::getParserErrorLine( )

Prototype

```
int getParserErrorLine( void )
```

Description

Retrieves the line number where an error occurred after `INI_PARSE_FAILURE` error.

Parameters

Not Applicable

Return Values

int	Parser error value as noted in INI Parser Class Definition.

DDI_INI_TOOLS::~DDI_INI_TOOLS()

Prototype

```
void ~DDI_INI_TOOLS(void )
```

Description

INI Tools Class destructor. Frees all memory associated with the parsed INI parameters.

Parameters

Not Applicable

Return Values

Not Applicable

Making Table Updates

A table is updated by making one or more calls to the 'update_ini_field()' method, followed by a call to the `commit_updates()` method.

Making multiple calls to the `update_ini_field()` method prior to `commit_updates()` will result in better performance than calling `commit_updates()` after each update. This is because a table update can be considered a binary operation. The binary file update follows these steps:

1. Rename the existing CFG delta update file to a temporary file name.
2. Recreate the CFG delta file including the new parameter updates.
3. If the new CFG Delta file creation was successful, then delete the temporary backup.
4. If the new delta file creation fails, then the original delta contents are restored, by renaming the temporary file to the working delta file.

Update Restrictions

- Update on the delta CFG file will be made in a trailing section of the Delta CFG file.
- If the `CFG` delta file does not exist, the file will be created using the path name specified.
- If the delta file already exists the updates will be appended to the specified delta file. The original contents of the delta `INI` file will not be disturbed.
- Updates to the `CFG` Delta file are independent of any of the parameters or tables previously defined in the main INI file or the delta file. In other words, new tables or parameters can be added without requireing prior definition.
- Conditional sections (defined using the 'exp' keyword) have no bearing on `CFG` updates.

Treck TCP/IP Stack Parameters

Treck has a notion of network interface that must be configured before it is started. The configuration may be is specific to the communication technology. For example setting the IP address (in case of static IP) is common across all communication technologies. For PPP there are a suite of configuration parameters.

CommEngine is the Verix EOS Volume II component that manages the Treck TCIPIP stack. CommEngine sets the Treck configuration parameters. The list of configuration parameters is a known set. In future this list is subject to change with new versions of Treck API. Such changes will affect CommEngine as it requires knowledge of the new configuration parameter.

To "future proof" CommEngine a configuration parameter driven approach is proposed. This requires a two new Treck API that CommEngine can use to set and get a configuration parameter. In addition the list of Treck configuration parameters and corresponding Ids must be created.

New Treck API for Configuration Management

Use the following APIs for Treck Configuration Management:

- set_config()
- get_config()

set_config()

Description

Fetch the configuration parameter value associated with parameter Id.

Prototype

```
int set_config(int handle, int paramId, char *paramValue, int
paramValueSize)
```

Parameters

In	`handle`	Device handle associated with Network Interface.
In	`paramId`	Treck Parameter ID
In	`paramValue`	Pointer to `paramValue` that
In	`paramValueSize`	Size of `paramValue` in bytes.

Return Values

0	Successful
-1	Failed. errno will be set.

get_config()

Description

Get the configuration parameter value associated with parameter ID.

Prototype

```
int get_config(int handle, int paramId, char *paramValue, int
paramValueSize, *paramValueLeng)
```

Parameters

In	`handle`	Device handle associated with Network Interface.
In	`paramId`	Treck Parameter ID
Out	`paramValue`	Pointer to `paramValue` that
In	`paramValueSize`	Size of `paramValue` in bytes.
Out	`paramValueLeng`	Length of `paramValue`. `paramValueLeng` is less than or equal to `paramValueSize`.

Return Values

0	Successful
-1	Failed. errno will be set.

Treck Parameter ID

The content of this table is illustrative of a sampling of Treck parameters. This is not an exhaustive list.

Parameter Name	Parameter ID	Data Type	Size
IP Address	0x0001	String (null terminated)	16
MTU	0x0002	Integer	4
PPP Username	0x0003	String (null terminated)	32
PPP Password	0x0004	String (null terminated)	64

Network Interface Configuration

WORK IN PROGRESS

File Location and Structure

Metadata File Location and Structure

Owner / Scope of Content

Mechanism for Reading and Updating

DDI Communicating with CE on Table Selected/Used

DDI Configuration

Parameter File Locations and Structure

Metadata File Location and Structure

Mechanism for Reading and Updating File (IOCTL Calls)

References

CommEngine-DDI_Integration_guide_v4

CommEngine Interface API (CEIF.lib)

The VxEOS Communication Engine or CommEngine (VXCE.OUT) provides services to applications. Applications can query, for example, the IP Address of the terminal, start and stop a network connection, etc. These services are provided by an API referred as the CommEngine Interface API (ceAPI). The complete scope and list of application services is the purpose of this document.

The ceAPI is a shared library and this library is referred as the Communication Engine Interface library or CEIF.LIB residing in VxEOS. All applications will have access to this shared library avail of the services of the CommEngine via ceAPI. Applications must first register using ceAPI before they can benefit from the services.

ceAPI covers a broad range of services. In addition applications may optionally choose to subscribe to unsolicited messages (or events) from CommEngine on the state of the network connection. The ceAPI application services are categorized as:

- Registration
- Device Management
- Communication infrastructure control
- Broadcast Messages
- IP Configuration
- Device Driver (DDI) configuration
- Connection status and configuration query

Message Exchange mxAPI & Message Formatting mfAPI

`ceAPI` under the covers operates by exchanging messages with `CommEngine`. `ceAPI`, to provide application services, constructs a message, referred as the “Request Message” and delivers it to CommEngine. CommEngine reads this message and in response sends the “Response Message”. `ceAPI` reads this response from CommEngine, analyzes it and provides the received information to the application.

Messages exchanged with CommEngine have a specific format and follow certain simple rules. The messages are a sequence of tag, length and value (TLV). Applications need to create the Request message and have the ability to read and parse the Response message both of which are in TLV format.

To summarize, `ceAPI` to provide application services must be able to:

- Send and receive messages with CommEngine

- The messages exchanged are in the TLV format that CommEngine understands. Similarly ceAPI must be able to under the TLV formatted message sent by CommEngine in response.

`ceAPI` takes advantage of two general purpose API to exchanges and format messages:

- `mxAPI` – Message eXchange API. Send and receive messages.
- `mfAPI` – Message Formatting API. Create and parse TLV messages.

NOTE

Both `mxAPI` and `mfAPI` libraries are part of CEIF.LIB which is a shared library and residing in VxEOS. It is important to note that both `mxAPI` and `mfAPI` are general purpose API and can be used by any application. Application developers are encouraged to use both these libraries in their applications. Using these two APIs it is possible to for any two applications to communicate and exchange messages and data.

ceAPI Concepts

The `ceAPI` provides applications the capability and flexibility to configure and manage its network interfaces. This section provides the background and core concepts behind the design of `ceAPI`.

Device & Device Ownership

In Verix V device is physical entity such as a modem. Each device has name such as /DEV/COM1 and referred as the device name.

A device is owned by the process or by the task that opens the device. When it is done with and closes the device, the ownership ceases and the device is available for any other device to use. Device ownership can be explicitly transferred from one task to another. Once the device is transferred no operation can be performed by the task that owned the device.

The key points are a device has a device name and is owned by the task that opens it. Device ownership can be transferred by the device that currently owns it.

Device Driver

In VxEOS a device driver is a software component that manages a device (described in the Device & Device Ownership section). This device driver is distinct and different from the OS device driver. Device deriver support a published interface referred as device driver interface or DDI. It is not uncommon to refer to this device driver as DDI driver in common usage.

Each device driver has a configuration file and applications using ceAPI can alter this configuration file. Device drivers read this configuration file at start up and configure themselves. The rest of this section provides additional information on the device driver concept and at the end revisit how applications can obtain / alter device driver configuration.

The device driver supports a specific communication technology such as Ethernet, WiFi, GPRS, CDMA, etc. More specifically it supports the combination of the device (modem) and the communication technology. For example a device driver supports GPRS on the Vx610 and the device name is /DEV/COM2.

It is possible to have another driver that supports GSM on /DEV/COM2. However since both drivers share the same device, both cannot be simultaneously supported. So if the device driver for GPRS is running it cannot run the driver for GSM and vice versa. The device driver to device is a one-to-one relationship, i.e., it is associated with one and only one device. The relationship between a device and the device driver is a one-to-many relationship.

In the previous example the two drivers for GPRS and GSM share the same device and their operation is mutually exclusive. Other factors also contribute to two device drivers operating mutually exclusive. If two devices are muxed on the same serial communications port (COM port) then only one device is accessible at a time. Alternately two devices cannot operate at the same time due to radio interference or power requirements or any other number of reasons. When two (or more) drivers have a mutually exclusive operation relationship then it is necessary to turn off a driver if the other needs to be turned on.

As described earlier in this section, each device driver has a configuration file. This file contains a list of parameters and corresponding values. At start up the driver reads the configuration parameters and configures itself. Applications using ceAPI can query and modify device driver configuration. In addition to configuration parameters, the configuration file contains “radio” parameters. When an application requests the value of a radio parameter, the value is fetched from the device. The wireless signal strength is an example of a “radio” parameter. Applications can query and obtain the current value but cannot it change it.

To summarize a device driver supports DDI (device driver interface) and is associated with a specific device and communication technology. Each driver has a configuration file with parameters and values. Applications may query and modify the parameter value. For the driver to function, the device must be available and owned by the task.

Network Interface (NWIF)

A Network Interface or NWIF for short is an entity by which a network connection can be managed. A terminal (or network device) can either be connected or not connected to the network. When connected the terminal has a network connection. Similarly, when the terminal is disconnected it has not network connection.

An application using ceAPI can connect a terminal to the network or disconnect the terminal from the network by managing the network interface. The application using ceAPI can query and modify the NWIF configuration, manage the connection or disconnection processes.

The network interface consists of the following elements:

- NWIF handle, a mechanism to refer to a network interface.
- Device, as described in Device & Device Ownership.
- Device driver, as described in Device Driver.
- Communication technology identifier. This is a property of driver and NWIF inherits it.
- NWIF configuration.
- NWIF connection state and error.

The NWIF works with two VxEOS components – device driver and the TCPIP stack. The device driver is started first and after this component is successfully established, the TCIPIP stack is initialized. When both these components are up and functioning, the network connection is established. Similarly when the network connection is torn down, both the components (driver and stack) must be closed before the network connection is closed.

Network Connection Process & States

CommEngine's objective is to bring up the network interfaces and notify applications where there a change in state. It notifies application via events. This section describes the process of bringing and tearing down the connection. The events are described in the NWIF Events section.

Bringing up the connection is a multi-step process and is depicted in Figure 1. These steps are transparent to applications. Similarly bringing down the connection is depicted in Figure 2. Using ceAPI an application can start or stop the network interface.

Some applications have requirements to manage the connection process and using ceAPI it is possible to incrementally bring up or down the connection. For example a CDMA or GPRS connection first brings up the link by dialing to the service provider then establishes the PPP session. In certain geographical regions the duration of the PPP connection is billed and there is incentive to tear down the connection when inactive. A practice has evolved to tear down the PPP session but to maintain the link layer up. This approach has performance advantage as the link layer need not be re-established and is cost saving as the PPP session is not idling. Re-establishing the PPP session takes a few seconds.

To accomplish this, applications must have the capability to control the connection process to pause at intermediate states and proceed in either direction, to establish the connection or tear it down. Figure 1 and Figure 2 below depict this connection and disconnection process.

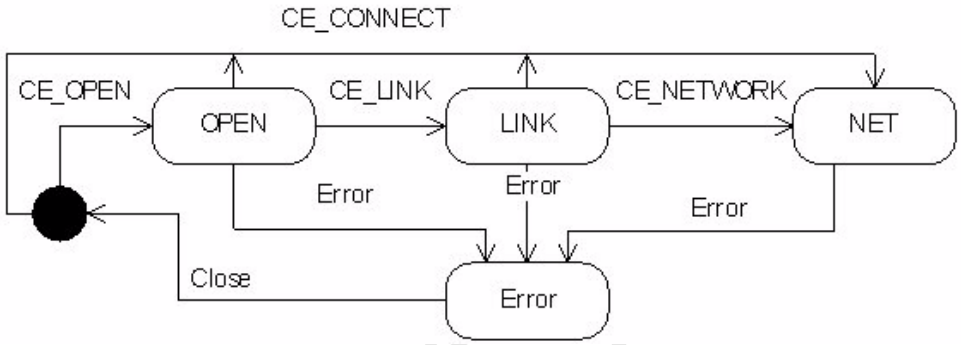


Figure 1 Connection-up state transition diagram (Left to Right)

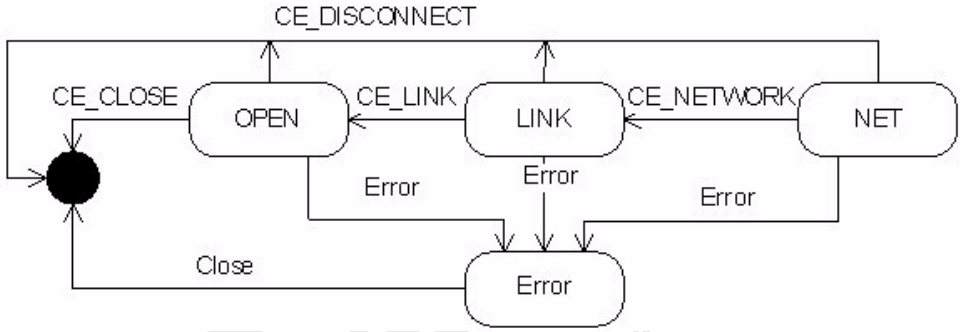


Figure 2 Connection-down state transition diagram (Right to Left)

NWIF Events

Events are posted to registered applications by CommEngine when interesting network “events” occur. For example when the network connection is established, an event is broadcasted to all applications that have registered for events. This lets applications know that network connectivity is established and can start processing transactions or any other network activity. Similarly if the network connection is lost, an event is posted indicating the network connection is down.

Network connection being down is distinct from a failed network connection. When the connection goes down there is possibility (not certainty) that the connection may recover back in the future. A WiFi terminal (e.g. Vx670 WiFi) going out and coming back in range illustrates this situation. When a network connection fails, it usually is configuration that is in compatible with the current environment. If the GPRS APN (Access Point Name) is incorrectly specified, this leads to a failed network connection. The APN value must be corrected before attempting to connect again. See List of CommEngine Events section for additional details.

As described in the Network Connection Process & States section, the connection or disconnection can be incremental. The application moves the connection from one state to the other using `ceAPI`. Here, states refer to `OPEN`, `LINK` and `NET` as shown in Figure 1 and Figure 2. The state transition process involves multiple steps – first the application calls `ceAPI` either `ceStartNWIF()` or `ceStopNWIF()` to move the connection to the next step. These API calls are non-blocking, i.e., they return the operation is not complete just that the request has been accepted and the request is being processed. When the operation is complete, CommEngine posts an event notifying the application that the requested operation is completed.

The List of CommEngine Events section provides the list of events generated by CommEngine. The Handling section provides sample on how application can take advantage of it.

The ceAPI – Summary

The `ceAPI` provides applications services to all registered applications. There are no pre-conditions to registration. The API is categorized based on the type of service it provides. They are categorized in the sub-sections below.

The API names are hyperlinked. Clicking on the API will take you to its detailed description.

Registration

API	Description
`ceRegister()`	Registers application with CommEngine.
`ceUnregister()`	Cancels prior successful registration with CommEngine.
`ceSetCommDevMgr()`	Register application for device management.

Device Management

API	Description
`ceRequestDevice()`	Request device from CommEngine.
`ceRequestDeviceNotify()`	Request the device from CommEngine. CommEngine will notify the application via user event when the device is available.
`ceReleaseDevice()`	Releases the device to CommEngine.
`ceCancelDevice()`	Cancels a previous device Request issued to CommEngine.

Device Driver (DDI) Configuration

API	Description
`ceGetDDParamValue()`	Fetch device driver (DD) parameter.
`ceSetDDParamValue()`	Set device driver (DD) parameter.

Sending Commands to Device

API	Description
`ceExCommand()`	Execute a command on the device and obtain response string.

Communication Infrastructure Configuration, Management & Control

API	Description
`ceGetNWIFCount()`	Obtain number of network interfaces (NWIF) supported.
`ceGetNWIFInfo()`	Obtain Network Interface information.
`ceStartNWIF()`	Starts all enabled network interfaces.
`ceStopNWIF()`	Stops all enabled network interfaces.
`ceSetNWIFStartMode()`	Specifies if the network interfaces are started at start up (auto mode) or need to be explicitly started (manual mode).
`ceGetNWIFStartMode()`	Obtain the current network interfaces start mode.
`ceSetNWParamValue()`	Specify IP and PPP configuration parameter value.
`ceGetNWParamValue()`	Fetch current IP and PPP configuration parameter value.
`ceControlParamLock()`	Lock the parameter files (network and device driver) associated with a network interface.

Event Notification

API	Description
`ceEnableEventNotification()`	Enables notification of events from CommEngine. Application also registers callback function to be called when an event is posted.
`ceDisableEventNotification()`	Disables notification of events from CommEngine. After this call the application will no longer receive events.
`ceIsEventNotificationEnabled()`	Returns the current state of event notification.

Log Operations

API	Description
`ceControlLog()`	Start and stop CommEngine logging.
`ceFetchLog()`	Creates a copy of the log file as a text file at target location.

Miscellany

API	Description
`ceActivateNCP()`	”Activates application VxNCP. On successful execution of this API, VxNCP has the focus and ownership of the console.”

`ceGetVersion()`	Obtain version information for CommEngine Interface library and CommEngine.

ceRegister()

Registers application with CommEngine.

Prototype

```
int ceRegister(void)
```

Return Values

`0`	Registration successful.
`< 0`	Error. See List of Error Codes. ”Of interest are errors ECE_REGAGAIN, ECE_CREATEPIPE and ECE_REGFAILED.”

Programming Notes

This is the first API that must be called by an Application. A pipe is created and message is sent to CommEngine to register. CommEngine responds either to confirm or deny the registration.

`ceRegister()` must be called just once. Calling it multiple times has no affect and results in an error. CommEngine must be running for successful registration.

ceUnregister()

Cancels the application's prior successful registration with CommEngine.

Prototype

```
int ceUnregister(void)
```

Return Values

`0`	Successfully unregistered. Prior registration successfully cancelled.
`< 0`	Error. See List of Error Codes. Of interest is error ECE_NOTREG.

Programming Notes

API `ceRegister()` creates a pipe, resisters with CommEngine and allocates necessary resources (memory). `ceUnregister()` does the reverse, it cancels the registration with CommEngine, closes the pipe and frees any other allocated resources.

Calling any other API after `ceUnregister()` will result in failure as the application is no longer registered. The application may register again by calling `API ceRegister()`.

Note: If this application is the CommEngine communication device management application and has successfully registered itself via API `ceSetCommDevMgr()`, it should not unregister. Attempts to unregister will fail.

ceSetCommDevMgr()

Registers the application for communication device management.

Prototype

```
int ceSetCommDevMgr(void)
```

Return Values

`0`	Application successfully registered as CommEngine communication device management application.
`< 0`	Error. See List of Error Codes. Of interest are errors ECE_NOTREG, ECE_CDMAPP and ECE_NOCDM.

Programming Notes

An application may optionally register itself as CommEngine's communication device management application.

See App VCCESA.OUT Engineering Requirements Specification VDN: 28810 Revision A.

ceRequestDevice()

Request device from CommEngine.

Prototype

```
int ceRequestDevice(const char deviceName[], const unsigned short
timeoutSec)
```

Parameters

In:	`deviceName`	VerixV device name. This is the device being request from CommEngine. Device information is returned by API `ceGetNWIFInfo` in structure element `niDeviceName` of parameter stArray.
In	`timeoutSec`	This API is a blocking call and will wait for `timeoutSec` duration for a response from CommEngine. The can be in the range from 30 to 600 seconds (10min). If it is outside this range the default, 30 seconds is assumed.

Return Values

`0`	Successful. Application is now device owner and may open the device.
`< 0`	Error. See List of Error Codes. Of interest are errors `ECE_NOTREG`, `ECE_DEVNAME`, `ECE_DEVOWNER` and `ECE_DEVBUSY`.

Programming Notes

The calling application is requesting CommEngine for the device `deviceName`. CommEngine transfers the ownership of device to the requesting application. The network interface associated with this device is torn down before the device is released to the application.

It is important to distinguish between device and network interface. Usually there is a one to one relationship between network interface and device but there are exceptions. The most common example is that of GPRS device, usually these devices also support GSM and it is possible to support PPP over a GSM call. A device may support more than one network interfaces but only one network interfaces may be active at a time.

This API is a blocking call, i.e., it returns when the device is available or timeout occurs. For CommEngine to relinquish the device it needs to tear down the network interface if one is up. It verifies that there are no open and active sockets prior to that. If the sockets are active then the device request is denied.

Use this API if the device is needed within the timeout limit. Alternately to use a non-blocking call use API `ceRequestDeviceNotify`.

ceRequestDeviceNotify()

Request the device from CommEngine. CommEngine will notify the application via user event when the device is available.

Prototype

```
int ceRequestDeviceNotify(const char deviceName[], const int taskId, const int userEvent)
```

Parameters

In:	`deviceName`	VerixV device name. This is the device being requested from CommEngine. Device information is returned by API `ceGetNWIFInfo` in structure element `niDeviceName of` parameter `stArray`.
In:	`taskId`	CommEngine will notify the application by posting a user event `userEvent` to the process with task Id `taskId`.
In:	`userEvent`	CommEngine will notify the application by posting a user event `userEvent` to the process with task Id `taskId`.

Return Values

`0`	Successful. CommEngine has accepted the request and will notify the application when the device is available.
`< 0`	Failed. Application must register before attempting this API or deviceName is unknown or CommEngine does not own the device. The request is denied. Error. See List of Error Codes. Of interest are errors ECE_NOTREG, ECE_DEVNAME, ECE_DEVOWNER, ECE_NOTASKID and ECE_DEVBUSY.

Programming Notes

See Programming Notes associated with `ceAPI ceRequestDevice()`.

This API is non-blocking. On return CommEngine has accepted the request and promises to notify the application when the device is available. There is no commitment when the device will be available.

Use this API if the device is needed with no time limit. Alternately use API `ceRequestDevice()`.

To cancel a device request created by this API use `ceCancelDevice()`.

ceReleaseDevice()

Releases the device to CommEngine.

Prototype

```
int ceReleaseDevice(const char deviceName[])
```

Parameters

In:	`deviceName`	VerixV device name. This is the device being returned to CommEngine. For Device information refer to API `ceGetNWIFInfo` in structure element `niDeviceName` of parameter stArray.

Return Values

`0`	Successful. CommEngine has accepted the device.
`< 0`	Failed. Application must register before attempting this API or `deviceName` is unknown or is a device not required by CommEngine. The release is rejected. Error. See List of Error Codes. Of interest are errors ECE_NOTREG, ECE_DEVNAME, and ECE_DEVOWNER.

Programming Notes

This is the mechanism for an application to “return” a device back to CommEngine.

CommEngine will bring up the network interface associated with this device to its prior state. If it was running then CommEngine start the network interface.

ceCancelDevice()

Cancels a previous device Request issued to CommEngine.

Prototype

```
int ceCancelDevice(const char deviceName[])
```

Parameters

In:	`deviceName`	VerixV device name. This is the device being returned to CommEngine. For Device information refer to API `ceGetNWIFInfo` in structure element `niDeviceName` of parameter stArray.

Return Values

`0`	Successful. CommEngine has cancelled the device request.
`< 0`	Failed. Application must register before attempting this API or deviceName is unknown or is prior device request does not exist. The request for cancelling the device request is ignored. Error. See List of Error Codes. Of interest are errors ECE_NOTREG, ECE_DEVNAME, ECE_DEVOWNER and ECE_NODEVREQ.

Programming Notes

This API is for cancelling a request issued by API `ceRequestDeviceNotify()`.

ceGetDDParamValue()

Fetch device driver (DD) parameter.

Prototype

```
int ceGetDDParamValue(const unsigned short niHandle, const char
paramStr[],char paramValue[], const unsigned short paramValueSize,
unsigned short *paramValueLen)
```

Parameters

`In:`	`niHandle`	Handle to network interface. Returned by API `ceGetNWIFInfo` in structure element `niHandle` of parameter `stArray`.
`In:`	`paramStr`	To fetch value associated with parameter `paramStr`. `paramStr` is a null terminated string. This parameter cannot be NULL[1].
`Out:`	`paramValue`	The value associated with `paramStr` is returned in `paramValue`. This parameter cannot be NULL.
`In:`	`paramValueSize`	Size of buffer `paramValue`. Its value must be greater than zero
`Out:`	`paramValueLen`	Number of bytes containing data in `paramValue`. This value is returned and is always less than or equal to `paramValueSize`.

[1] The list of parameters is published for each communication device separately along with each DDI driver.

Return Values

`0`	Parameter value obtained.
`< 0`	Failed. See List of Error Codes.

Programming Notes

This API is for fetching the value associated with parameter `paramStr`. Device driver configuration parameters can be obtained (such as timeouts, waiting times, etc) even if the device is unavailable or network interface is disabled or not up.

Real time parameters, such as signal strength, wireless carrier, etc. can be obtained from the device/modem. This is possible only if the network interface is started and running.

ceSetDDParamValue()

Set device driver (DD) parameter.

Prototype

```
int ceSetDDParamValue(const unsigned short niHandle, const char
paramStr[], const char paramValue[], const unsigned short paramValueLen)
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	paramStr	To set value associated with parameter paramStr. paramStr is a null terminated string. This parameter cannot be NULL.[2]
In:	paramValue	The value associated with paramStr. This parameter cannot be NULL.
In:	paramValueLen	Number of bytes containing data in paramValue.

[2] The list of parameters is published for each communication device separately along with each DDI driver. See footnote 1.

Return Values

0	Parameter value set.
< 0	Failed. See List of Error Codes.

Programming Notes

This API is for assigning a value associated with parameter paramStr. Device driver configuration parameters can be set (such as timeouts, waiting times, etc) even if the device is unavailable or network interface is not up.

ceExCommand()

Execute a command on the device and obtain response string.

Prototype

```
int ceExCommand(const unsigned short niHandle, const char *cmdStr, const
unsigned short cmdStrLen, const unsigned short cmdRespSize, char *cmdResp,
unsigned short *cmdRespLen, const unsigned long timeoutMS)
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	cmdStr	cmdStr contains the command string to execute. cmdStr is of length cmdStrLen. This parameter cannot be NULL.
In:	cmdStrLen	Length of cmdStr.
In:	cmdRespSize	Size of buffer cmdResp in bytes. Its value must be greater than zero.
Out:	cmdResp	The response in consequent of executing cmdStr.
Out:	cmdRespLen	Number of bytes containing response data in cmdResp. This value is returned and is always less than or equal to cmdRespSize.
In:	timeoutMS	This command is executed by the device driver. Parameter timeoutMS is the duration in milliseconds the device driver will wait for a response before it gives up. This timeout value should be reasonable and long enough to obtain a response. If this parameter is zero, the device driver will wait indefinitely till a response is obtained.

Return Values

0	Command executed successfully.
< 0	Failed. Application must register before attempting this API or parameter name is unknown or timeout occurred waiting for response. See List of Error Codes.

Programming Notes

This API can be successfully executed if network interface associated with niHandle is enabled and running. There may certain states the device driver may be in and running of this command is either not permissible or prudent.

This API is provided as option for applications where in running a specific command is necessary and should be used as a last resort. Usage of this API is strongly discouraged.

ceEnableEventNotification()

Subscribe to notification events from CommEngine.

Prototype

```
int ceEnableEventNotification(const int taskId, const int userEvent,
fifo_t *pF)
```

Parameters

In:	`taskId`	When application receives an event from CommEngine, the application is notified by posting a user event `userEvent` to the process with task Id `taskId`.
In:	`userEvent`	When application receives an event from CommEngine, the application is notified by posting a user event `userEvent` to the process with task Id `taskId`
In:	`pF`	Pointer to FIFO structure. The event from CommEngine and event data will be posted here.

Return Values

0	Successful.
< 0	Failed. See List of Error Codes.

Programming Notes

This API is non-blocking and sets the environment for handling events from CommEngine.

There are two events – the first is the CommEngine event which the application registered for and the second, the user event, is used as notification mechanism. Though both are referred as events, the delivery mechanism for both is different.

The CommEngine event is delivered internally. This event and associated event data are placed in the FIFO pointed by parameter `pF`. After this `userEvent` (second parameter) is posted to the task whose task Id is `taskId` (first parameter).

The application reacts to this event and then reads the user event by calling OS API `read_user_event()`. At this point the application is aware that a CommEngine event is available and reads the FIFO for the CommEngine event and related event data.

In the FIFO the data is structured as follows:

1. CommEngine event – 4bytes (int)
2. Event data length – 2bytes (short)
3. Event data of size length – length bytes (var)

The last field is not present if length is zero.

ceDisableEventNotification()

Unsubscribe to CommEngine notification events. After this call the application will no longer receive events.

Prototype

```
int ceDisableEventNotification(void)
```

Return Values

`0`	Successful.
`< 0`	Failed. Error. See List of Error Codes.

Programming Notes

Event notification is disabled by default. Use this API if event notification was enabled via API Error! Reference source not found.

ceSetSignalNotificationFreq()

Set frequency of signal strength notification events from CommEngine to application.

Prototype

```
int ceSetSignalNotification(const unsigned short freq)
```

Parameters

In:	`freq`	Specify the frequency at which notifications are sent. The possible frequency values are `CE_SF_HIGH`, `CE_SF_MED`, `CE_SF_LOW` or `CE_SF_OFF`. Frequency values are defined in the Constants section.

Return Values

`0`	Successful.
`< 0`	Failed. See List of Error Codes.

Programming Notes

An application to receive signal strength notifications, is must call this API with the required frequency and Error! Reference source not found. This API enables delivery of all network events to the application including signal strength. Signal strength events are sent by CommEngine only if the underlying hardware support wireless and signal strength makes sense. In addition, the Network interface associated with wireless must be up and running.

CommEngine will send signal strength updates at frequency chosen by the application. For example if the application chooses high (`CE_SF_HIGH`) then it is sent every 5 seconds. The time interval 5 seconds is the default value and it can be changed by altering parameter `CE_SIG_FREQ_BASE` (see List of CommEngine Parameters). The scope of this parameter is CommEngine so it affects all applications.

The frequency can be reduced by changing parameter freq to either CE_SF_MED (every 5*2=10 seconds) or CE_SF_LOW (5*4=20seconds). Signal strength notifications can be turned off completely by setting the frequency to CE_SF_OFF.

When the application has focus (console and keyboard) the frequency of notifications may be set to CE_SF_HIGH to update the screen as frequently as possible. However when there is no focus, the frequency may be reduced to CE_SF_LOW or turned off. When the application is preparing to go power save, it is recommended that it disable all notifications by calling API `ceDisableEventNotification()`.

To obtain the current frequency of signal strength notification setting, use API `ceIsEventNotificationEnabled()`.

ceIsEventNotificationEnabled()

Returns the current state of event notification and frequency of signal strength notification.

Prototype

```
int ceIsEventNotificationEnabled(unsigned short *freq)
```

Parameters

In:	freq	Specify the frequency at which notifications are sent. The possible frequency values are CE_SF_HIGH, CE_SF_MED, CE_SF_LOW or CE_SF_OFF. Frequency values are defined in the Constants section.

Return Values

1	Event notification enabled.
0	Event notification disabled.
< 0	Failed. See List of Error Codes.

Programming Notes

When event notification is disabled no signal strength notifications are sent regardless of frequency value. Only when event notifications are enabled are signal strength notifications sent at the chosen frequency.

ceGetNWIFCount()

Obtain number of network interfaces (NWIF) supported.

Prototype

```
int ceGetNWIFCount(void)
```

Return Values

`0`	Event notification disabled.
`< 0`	Failed. See List of Error Codes.

Programming Notes

Terminals support multiple communication devices (or modems) and referred as network interfaces. This API returns the total number of network interfaces supported. For example on the Vx570 the two network interfaces are Ethernet and the Eisenhower modem (over PPP).

It is recommended that API be called prior to API `ceGetNWIFInfo()`. The count provided by this API required as input to API `ceGetNWIFInfo()`.

ceGetNWIFInfo()

Obtain Network Interface information. See description of structure `stNIInfo` for details.

Prototype

```
int ceGetNWIFInfo(stNIInfo stArray[], const unsigned short stArraySize,
unsigned short *arrayCount)
```

Parameters

In / Out:	stArray	Pointer to array of stArraySize elements of stNIINfo.
In:	stArraySize	The number of elements in parameter stArray. Its value should be the value returned by API ceGetNWIFCount()
Out:	arrayCount	Number of elements in stArray populated. The value of parameter arrayCount is same as stArraySize if stArraySize is same as the value returned by API ceGetNWIFCount().

Return Values

0	Successful.
< 0	Failed. See List of Error Codes.

Programming Notes

This API returns the information associated with each network interface in structure `stNIInfo`.

Configuration Structure stNIInfo

```
typedef struct
{
  unsigned short niHandle;          //  Handle to Network Interface
  char niCommTech[16];              //  Null terminted str. Refer to section 5.2
                                        for complete list.
  char niDeviceName[32];            //  /DEV/COMx /DEV/WLN1 /DEV/ETH1
  unsigned short niRunState         //  1—Running, 2—Ready,
                                        3-Not Ready, 0-Failed to run
  unsigned short                    //  Error code associted with
niErrorCode
  unsigned short                    //  1—Auto startup, 0—Manual startup
niStartUpMode
  char                              //  Format
niDeviceDriverName[12+1]                <MediaTag><VendorTag>[3]
} stNIInfo;
```

[3] See VxEOS Architecture document Section 5.2.2.9.2 for details.

ceStartNWIF()

Starts network interface associated with the handle.

Prototype

```
int ceStartNWIF(const unsigned short niHandle, const short cl);
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	cl	Specify the extent to which the connection is established. To connect all the way, set parameter cl to CE_CONNECT. To perform incremental connection set cl to CE_OPEN, CE_LINK_ or CE_NETWORK. For definitions see the Constants section.

Return Values

0	Successful.
< 0	Failed. See List of Error Codes.

Programming Notes

The network interface can either start the connection incrementally or completely. See Network Connection Process & States for details on bringing up the connection.

This API is non-blocking. When it returns successfully, the request has been accepted by CommEngine. When the network interface is started (or moves to the next state), CommEngine posts an event. If the interface cannot be started for any reason, CommEngine posts an event. See List of CommEngine Parameters.

ceStopNWIF()

Stops network interface associated with handle.

Prototype

```
int ceStopNWIF(const unsigned short niHandle, const short cl);
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	cl	Specify the extent to which the connection is established. To connect all the way, set parameter cl to CE_CONNECT. To perform incremental connection set cl to CE_OPEN, CE_LINK_ or CE_NETWORK. For definitions see the Constants section.

Return Values

0	Successful.
< 0	Failed. See List of Error Codes.

Programming Notes

The network interface can either start the connection incrementally or completely. See Network Connection Process & States for details on bringing up the connection.

This API is non-blocking. When it returns successfully, the request has been accepted by CommEngine. When the network interface is started (or moves to the next state), CommEngine posts an event. If the interface cannot be started for any reason, CommEngine posts an event. See List of CommEngine Parameters.

ceSetNWIFStartMode()

Specifies if the network interfaces are started at start up (auto mode) or need to be explicitly started (manual mode).

Prototype

```
int ceSetNWIFStartMode(const unsigned short niHandle, const unsigned short startmode)
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	startmode	0 – manual mode (CE_SM_MANUAL). Network interfaces must be explicitly started. 1 – auto mode (CE_SM_AUTO). At power up or restart, this network interfaces will be started.

Return Values

0	Successful.
< 0	Failed. See List of Error Codes.

Programming Notes

Setting the auto mode affects the next time the terminal is either cold or warm booted. There is no immediate effect.

In auto mode the connection is established completely, i.e., it is equivalent to ceStartNWIF (handle, CE_CONNECT). For starting incrementally, set the start mode to manual (CE_SM_MANUAL).

ceGetNWIFStartMode()

Obtain the current network interfaces start mode.

Prototype

```
int ceGetNWIFStartMode(void)
```

Return Values

`CE_SM_AUTO`	Auto mode. At power up or restart, all enabled network interfaces are started.
`CE_SM_MANUAL`	Manual mode. Network interfaces must be explicitly started by using API `ceStartNWIF()`.
`< 0`	Failed. See List of Error Codes.

ceSetNWParamValue()

Set network (NW) parameter.

Prototype

```
int ceSetNWParamValue(const unsigned short niHandle, const char
paramStr[], const char paramValue[], const unsigned short paramValueLen)
```

Parameters

In:	`niHandle`	Handle to network interface. Returned by API `ceGetNWIFInfo` in structure element `niHandle` of parameter stArray.
In:	`paramStr`	To set value associated with parameter `paramStr`. `paramStr` is a null terminated string. This parameter cannot be NULL. The list of parameters is available in the List of Network Parameters section.
In:	`paramValue`	The value associated with `paramStr`. This parameter cannot be NULL.
In:	`paramValueLen`	Number of bytes containing data in `paramValue`.

Return Values

`0`	Parameter value set.
`< 0`	Failed. Application must register before attempting this API or parameter name is unknown. See List of Error Codes.

Programming Notes

This API is for assigning a value associated with parameter `paramStr`. Certain network configuration parameters can be even if the device is unavailable or network interface is disabled or not up.

ceGetNWParamValue()

Fetch network (NW) parameter value.

Prototype

```
int ceGetNWParamValue(const unsigned short niHandle, const char
paramStr[],char paramValue[], const unsigned short paramValueSize,
unsigned short *paramValueLen)
```

Parameters

`In:`	`niHandle`	Handle to network interface. Returned by API `ceGetNWIFInfo` in structure element niHandle of parameter stArray.
`In:`	`paramStr`	”To fetch value associated with parameter paramStr. paramStr is a null terminated string. This parameter cannot be NULL. The list of parameters is available in the List of Network Parameters section.”
`Out:`	`paramValue`	The value associated with `paramStr` is returned in `paramValue`. This parameter cannot be NULL.
`In:`	`paramValueSize`	Size of buffer `paramValue`. Its value must be greater than zero.

Return Values

`0`	Parameter value obtained.
`< 0`	Failed. See List of Error Codes.

Programming Notes

This API fetches the value associated with parameter `paramStr`. Certain network configuration parameters may be obtained even if the device is unavailable or network interface not up.

Certain parameters, such as DHCP lease time are meaningful only when network interface is up.

ceControlParamLock()

Applications can optionally lock the parameter files (network and device driver) associated with a network interface. Same API may be used to unlock and check status.

Prototype

```
int ceParamLockControl(const unsigned short niHandle, const unsigned short lockAction)
```

Parameters

In:	niHandle	Handle to network interface. Returned by API ceGetNWIFInfo in structure element niHandle of parameter stArray.
In:	lockAction	Lock operation to perform – CE_LA_LOCK, CE_LA_UNLOCK or CE_LA_STATUS. See Constants for list and description of lock actions.

Return Values

0	Success for lock action CE_LA_LOCK and CE_LA_UNLOCK. For CE_LA_STATUS indicates the parameters files are unlocked.
1 or 2	For CE_LA_STATUS, a return value of 1 indicates the file is locked and the calling application owns the lock. A return value of 2 indicates the file is locked and the application is not the owner of the lock.
< 0	Input error. See List of Error Codes.

Programming Notes

Applications sometime require exclusive write access to parameters files. This prevents other application from making changes while the one application is planning on making a suite of changes. The lock is write only, i.e., it prevents other applications from writing but it does not prevent it from reading. Other applications can determine the file lock status and be cautioned prior to performing a read.

The application when it acquires a lock, it is for a fixed duration (default 5min). This is the lock recovery mechanism. The lock duration is automatically extended for the same duration after a call to either ceSetDDParamValue() or ceSetNWParamValue(). The default lock duration can be altered and is described below.

When the lock duration expires, CommEngine sends a CE_EVT_LOCK_EXP event to the application. After the application receives this event, it expected to take remedial action either by calling API ceSetDDParamValue() or ceSetNWParamValue() which extends the duration of the lock. Alternately it can release the lock, i.e., unlock it.

If the application takes receives CE_EVT_LOCK_EXP, the application is quasi owner of the lock. Should it take remedial action as described above, it has either full ownership or relinquished it.

If the application takes no remedial action after it receives `CE_EVT_LOCK_EXP`, should any other application request for the lock it will receive the lock and an event `CE_EVT_LOCK_REL` is posted to the application. This approach ensures that there is due diligence on CommEngine's part should it encounter a lackadaisical application.

Since CommEngine posts events for remedial action, it is necessary for the application to enable events for the duration the lock is active. ceAPI **Error! Reference source not found.** must be called prior to locking and ceAPI `ceDisableEventNotification()` may optionally be called only after the lock is relinquished.

As mentioned earlier, the default lock duration is 5 min and can be altered by changing CommEngine parameter `CE_LOCK_DURATION`. See List of CommEngine Parameters for additional details.

ceControlLog()

Start and stop CommEngine logging.

Prototype

```
int ceControlLog(const unsigned short operation)
```

Parameters

In:	operation	Operation to perform – start, stop or purge. See Log Operations.

Return Values

1 / 0	For CE_LOG_STATE, 1 – logging enabled, 0 – logging disabled.
0	Success. For all other operations.
< 0	Failed. See List of Error Codes.

Programming Notes

Application must be registered to perform log control operations.

ceFetchLog()

Creates a copy of the log file as a text file at target location

Prototype

```
int ceFetchLog (const char *fileName)
```

Parameters

`In:`	`fileName`	Target log filename. A copy of the current log file is copied to file with filename.

Return Values

`0`	Success. Log file copied at target location.
`< 0`	Failed. See List of Error Codes.

Programming Notes

The target file is created if one is not present. If the file is present it is overwritten. The log file is a text file and each entry is prefixed with a date and time stamp and delimited by a CRLF (0x0D0A).

ceActivateNCP()

Activates application VxNCP. On successful execution of this API, VxNCP has the focus and ownership of the console.

Prototype

```
int ceActivateNCP(void)
```

Return Values

`0`	Successful. VxEOS application VxNCP activated.
`< 0`	Failed. Possible reasons are: 1. Application not registered with CommEngine. 2. Application currently does not own the console. 3. Application VxNCP is not running. See List of Error Codes.

Programming Notes

Any application that owns the console (/dev/console) can activate VxNCP by calling this application. When user can exits VxNCP, it returns back to this application, i.e., the application that activated it.

Refer to documentation titled VCCESA.OUT, See References section.

ceGetVersion()

Obtain version information for CommEngine Interface library (CEIF.LIB) and CommEngine (VxCE.OUT).

Prototype

```
int ceGetVersion(const unsigned short component, const unsigned short
verStrSize, char *verStr)
```

Parameters

In:	component	VxEOS component to obtain version information. See Constants for list of values.
In:	verStrSize	Size of buffer verStr.
Out:	verStr	Buffer where the version string is returned. A null terminated string is returned provided the buffer size verStrSize is sufficient.

Return Values

0	Value returned in parameter verStr.
< 0	Input error. See List of Error Codes.

Programming Notes

Application must register to obtain version information.

The version information is returned in the form a.b.c where a, b and c are numerical digits. For example “1.3.5” is valid version string.

Constants, Defines & Miscellany

Use the following information to determine constants, defines and other variables.

List of CommEngine Events

The table contains the list of CommEngine events that an application is expected to receive. The event may optionally contain event data. Details of event data will be published in a future version of this document.

Event ID	Event **Value**	Dist. Scope	Description
CE_EVT_NET_UP	0x0001	Broad-cast[1]	Sent when network is up and running.
CE_EVT_NET_DN	0x0002	Broad-cast	Sent when the network is torn down.
CE_EVT_NET_FAILED	0x0003	Broad-cast	"Sent when attempting to establish the network, the connection failed. The event data contains the error code, the reason for failure."
CE_EVT_NET_OUT	0x0004	Broad-cast	The network is up but not active. This event is sent when there is either no coverage (cellular) or out of range (WiFi) or cable has been removed (Ethernet). This is a temporary outage
CE_EVT_NET_RES	0x0005	Broad-cast	The network is restored and is running. This event usually follows CE_EVT_NET_OUT. This event occurs after the outage has been fixed.
CE_EVT_SIGNAL	0x0006	Subs-cribed	The event data contains the signal strength as percentage and the signal strength in dBm.
CE_EVT_START_OPEN	0x0007	Single App	Event posted to requesting application after successful ceStartNWIF(CE_OPEN).
CE_EVT_START_LINK	0x0008	Single App	Event posted to requesting application after successful ceStartNWIF(CE_LINK).
CE_EVT_START_NW	0x0009	Single App	Event posted to requesting application after successful ceStartNWIF(CE_NETWORK).

CE_EVT_START_FAIL	0x000A	Single App	Event posted to requesting application after failed ceStartNWIF(). This is an incremental start.
CE_EVT_STOP_NW	0x000B	Single App	Event posted to requesting application after successful ceStopNWIF(CE_NETWORK).
CE_EVT_STOP_LINK	0x000C	Single App	Event posted to requesting application after successful ceStopNWIF(CE_LINK).
CE_EVT_STOP_CLOSE	0x000D	Single App	Event posted to requesting application after successful ceStopNWIF(CE_CLOSE).
CE_EVT_STOP_FAIL	0x000E	Single App	Event posted to requesting application after failed ceStopNWIF(). This is an incremental stop.
CE_EVT_LOCK_EXP	0x000F	Single App	Lock duration has expired. Either renew the lock by performing parameter update operation or relinquish the lock. Only application that has successfully locked using API ceControlParamLock() will receive this event.
CE_EVT_LOCK_REL	0x0010	Single App	CommEngine has notified the application that it is no longer the lock owner. This event is sent after CE_EVT_LOCK_EXP has been sent and the application has taken no remedial action.

[1] Broadcast to all applications that have registered for events via ceAPI Error! Reference source not found..

Constants

Constant	Value	Description
Comm. Tech		Communication Technology Descriptor String

	Ethernet	Ethernet
	WiFi	Wireless Fidelity
	GPRS	General Packet Radio Service
	CDMA	Code Division Multiple Access
	Dial/PPP	Dial / Point to Point Protocol

Component Ver.		Component Name for Version Information
CE_VER_CEIF	1	CommEngine Interface Library
CE_VER_VXCE	2	CommEngine

Logging Operation		Operations performed on CommEngine Log
CE_LOG_START	1	Start logging
CE_LOG_STOP	2	Stop logging
CE_LOG_PURGE	3	Purge log file
CE_LOG_STATE	4	Returns if the logging has started or stopped.

Starting the connection		Start connection states.
CE_CONNECT	10	Establish the full connection (normal operation).
CE_OPEN	12	Open and prep the communication device.
CE_LINK	13	On PPP devices the data connection is established.
CE_NETWORK	14	Establish the network connection.

Stopping the connection		Stop connection states
CE_DISCONNECT	11	Disconnect and close the communication device (normal operation)
CE_NETWORK	14	Close the network connection.
CE_LINK	13	On PPP devices the data connection is torn down
CE_CLOSE	15	Close the device.

NWIF Start Mode		Network Interface Start Mode
CE_SM_AUTO	0	Start Mode - Automatic
CE_SM_MANUAL	1	Start Mode - Manual
Signal Notification Frequency		Frequency at which signal strength notifications are sent to applications by CommEngine. The base interval can be altered CommEngine configuration variable CE_SIG_FREQ_BASE
CE_SF_HIGH	3	Signal Strength notification is sent once every CE_SIG_FREQ_BASE seconds.
CE_SF_MED	2	Signal Strength notification is sent once every 2*CE_SIG_FREQ_BASE seconds.
CE_SF_LOW	1	Signal Strength notification is sent once every 4*CE_SIG_FREQ_BASE seconds.
CE_SF_OFF	0	No notifications are sent.
Lock Action		Parameter file lock operation
CE_LA_LOCK	1	On a successful lock operation the application gets an exclusive write lock on the parameters files.
CE_LA_UNLOCK	2	Application releases the exclusive write lock on the parameter files.
CE_LA_STATUS	3	Determine the lock status.

List of Network Parameters

The table below lists the configuration parameters that can be either set or fetched using ceAPI ceSetNWParamValue() or ceGetNWParamValue() respectively.

Parameter	Data Type	Size	Get	Set	Description
IP_CONFIG	stNI_IPConfig	-	✓	✓	See Configuration Structure stNI_IPConfig
PPP_CONFIG	stNI_PPPConfig	-	✓	✓	See Configuration Structure stNI_PPPConfig

Configuration Structure stNI_IPConfig

```
typedef struct
{
  unsigned long ncIPAddr;    //    0-DHCP or Static IP otherwise.
  unsigned long ncSubnet;    //    Mandatory if Static IP
```

```
  unsigned long ncGateway;          //  Mandatory if Static IP
  unsigned long ncDNS1;             //  Optional but usually required
  unsigned long ncDNS2;             //  Optional
  char dhcpLeaseStartTime[14];      //  YYYYMMDDHHMMSS format (read only)
  char dhcpLeaseEndTime[14];        //  YYYYMMDDHHMMSS format (read only)
} stNI_IPConfig;
```

Configuration Structure stNI_PPPConfig

```
typedef struct
{
  pppAuthType ncAuthType;           //  Type of PPP authentication
  char ncUsername[MAX_USERNAME];    //  PPP username, optional
  char ncPassword[MAX_PASSWORD];    //  PPP password, optional
} stNI_PPPConfig;
```

List of CommEngine Parameters

The table below lists the CommEngine configuration parameters. CommEngine configuration parameters can be changed only via VxNCP.

Parameter	Data Type	Size	Get	Set	Description
CE_SIG_FREQ_BASE	short	2	✓	✓	The interval at which signal strength is sampled and its value is sent to applications. Changes to its value goes into affect immediately. Units: Seconds. Default: 5seconds. Min: 5, Max: 60
CE_LOCK_DURATION	short	2	✓	✓	The duration the lock is active before remedial action is taken by CommEngine due to an inactive application.

Units: Seconds

Default: 5min.

Min: 1min, Max: 10min.

NOTE

The list of CommEngine parameters is illustrative. This list will be expanded.

List of Error Codes

Error codes returned by ceAPI are negative (< 0). `ceAPI` are designed to return a non-negative return value (>=0) if the operation is successful. The return values are documented with each API. Negative return values (< 0) are listed here.

`ceAPI Error Codes`

Error ID	Error Value	Description
`ECE_REGAGAIN`	-1001	Application is attempting to register again after a successful registration. If necessary unregister by calling ceAPI `ceUnregister()` and register again using API ceRegister().
`ECE_REGFAILED`	-1002	Registration failed as CommEngine is not running.
`ECE_CREATEPIPE`	-1003	Application creates a pipe as part of the registration process. This error is returned if the API has failed to create a pipe.
`ECE_NOTREG`	-1004	Application has not successfully registered with CommEngine. This error returned if an application is attempting a ceAPI prior to successful registration.
`ECE_CDMAPP`	-1005	Application has failed to register as CommEngine's communication device management application. Either another application has successful registered or this application is attempting more than once. See API `ceSetCommDevMgr()` for additional details.
`ECE_NOCDM`	-1006	Application is attempting to register itself as CommEngine's communication device management application when none is required. See API `ceSetCommDevMgr()` for additional details.
`ECE_DEVNAME`	-1007	Unknown device name. This is not a communication device that CommEngine is aware of.
`ECE_DEVOWNER`	-1008	CommEngine is not the owner of the device. This is a communication device that CommEngine is aware of but currently not is its possession.
`ECE_DEVBUSY`	-1009	CommEngine is currently the owner of this device. CommEngine is unable to release this device as it is busy. This error is returned if there are open sockets.

ECE_NOTASKID	-1010	Task Id provided as parameter does not exist.
ECE_NODEVREQ	-1011	No prior device request was made. Nothing to cancel.

NOTE

The list of ceAPI Error codes is illustrative. This list will be expanded.

Application Developer Notes

This section covers two topics of interest to Application Developers. Handling CommEngine Events describes how applications can manage events coming from CommEngine. Managing and Controlling the Connection describes the starting and stopping the connection either incrementally or in one go.

Handling CommEngine Events

This sample code illustrates how an application enables events using ceAPI `ceEnableEventNotification()` and manages them.

sample_handling_commengine_events.txt

Managing and Controlling the Connection

In this sample the application starts the CommEngine incrementally and to notify it after the network interface is in OPEN state, `ceStartNWIF(CE_OPEN)`. CommEngine notifies the application by sending event `CE_EVT_START_OPEN`. At this point the application sends a command to the device to obtain its ICCID (assuming it is a GPRS device) via API ceExCommand(). It then starts the network by API `ceStartNWIF(CE_CONNECT)`.

Function `process_CEEvent()` illustrates how the application can be managed using the events from CommEngine. The signal strength event is used to update the display. References to code snippets in Handling CommEngine Events are used in this section.

sample_managing_controlling_connection.txt

Message Exchange mxAPI & Introduction

The Message Exchange API is designed for applications to communicate via pipes. Any set applications intending to communicate with each other can take advantage of this API. For example if App-A and App-B wish to communicate with each other, each application creates a pipe and communicates with each other by sending and receiving messages.

Data Layout

The message that is exchanged between two applications consists of a two fields

- Header and
- Payload

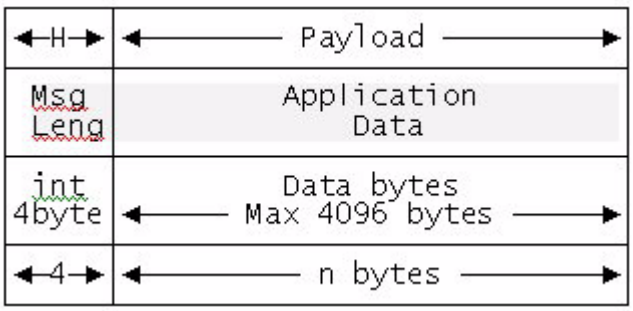


Figure 3 Message Fields

The Payload field is the application data that is sent by the sending application to the receiving application. The contents and size are provided by the sending application. This field is considered as a sequence of bytes of size n, where n is the size of the payload. The maximum payload is 4096bytes.

The Payload field is appended to the Header field before the message is sent to the recipient. The Header field consists of one field:

Field name	Data type	Size	Description
MsgLeng	int	sizeof(int) 4 bytes	Size of the Message - computed by adding the size of the header (4 bytes) to the size of the Payload. If the Payload is 500 bytes then this field has the value of 504.
	Total	4 bytes	

Message Exchange mxAPI – Summary

The Message Exchange API consists of the functions listed in the table below. The following section provides detailed description of each API. The API names are hyperlinked to the description.

API	Description
`mxCreatePipe()`	Creates a message pipe
`mxClosePipe()`	Closes the pipe created by API `mxCreatePipe()`
`mxGetPipeHandle()`	Obtain pipe handle associated with pipe name
`mxSend()`	Send message to destination pipe
`mxPending()`	Determines number of messages in the queue that are pending read
`mxRecv()`	Reads pending message from queue

mxCreatePipe()

Creates a VerixV message pipe and returns handle to pipe.

Prototype

```
int mxCreatePipe(const char *pipeName, const short messageDepth)
```

Parameters

In:	`pipeName`	Name of pipe to create. `pipeName` is up to 8 characters long. Pipe name be NULL or empty string (“”) for anonymous pipe.
In:	`messageDepth`	Maximum number of incoming messages buffered before they are read. `messageDepth` must be greater than 0 and less than or equal to 10.

Return Values

`0`	Success. Pipe created and handle returned
`< 0`	See mxAPI Error Codes.

mxClosePipe()

Close Verix V message pipe created via `mxCreatePipe()`.

Prototype

```
int mxClosePipe(const int pipeHandle)
```

Parameters

`In:`	`pipeHandle`	Pipe handle returned by mxCreatePipe().

Return Values

`0`	Returns zero when the pipe is close successfully. Returns zero when the pipe handle is not a valid pipe handle. Returns zero in all circumstances.

mxGetPipeHandle()

Obtains pipe handle associated with pipe name.

Prototype

```
int mxGetPipeHandle(const char *pipeName)
```

Parameters

In:	pipeName	Target pipe name whose handle to obtain. Only the first 8 characters are considered in the comparison. The pipe name cannot be NULL or empty string (“”).

Return Values

> = 0	Pipe handle. API `mxGetPipeHandle()` fetches the handle associated with a pipe created with name pipeName.
< 0	`pipeName` does not match currently open pipes. See mxAPI Error Codes.

Programming Notes

Use this API to obtain the handle of a named pipe. API `mxSend()` requires the handle of the destination pipe.

mxSend()

Sends message to destination pipe.

Prototype

```
int mxSend(const int pHSrc, const int pHDest, const char *dataPayload,
const short dataLength)
```

Parameters

In:	pHSrc	Source pipe handle. Created using `mxCreatePipe()`.
In:	pHDest	Destination pipe handle. Obtained handle via `mxGetPipeHandle()` or provided by other means.
In:	dataPayload	Data to send to destination pipe.
In:	dataLength	Size of data in `dataPayload`. Size cannot exceed 4096 bytes.

Return Values

`> 0`	Returns `dataLength`. Returns positive non-zero value on successful send.
`< 0`	See mxAPI Error Codes.

mxPending()

Determines number of messages in the queue that are pending read.

Prototype

```
int mxPending(const int pH)
```

Parameters

In:	pH	Pipe handle. Created using `mxCreatePipe()`.

Return Values

`> = 0`	Count of number of unread / pending messages. Will return zero if no messages are pending.
`< 0`	If the handle is not valid. See mxAPI Error Codes.

Programming Notes

Use this API to obtain the handle of a named pipe. API `mxSend()` requires the handle of the destination pipe.

mxRecv()

Reads pending message from queue.

Prototype

```
int mxRecv(const int pH, int *pHFrom, char *dataPayload, const short
*dataLength)
```

Parameters

In:	pH	Pipe handle. Created using mxCreatePipe().
Out:	pHFrom	Sender pipe handle.
Out:	dataPayload	Pointer to buffer to receive payload data from sender.
Out:	dataLength	Length of data in payload buffer.

Return Values

> 0	Message read and number of bytes in payload returned.
= 0	Message read and has no payload.
< 0	Read error or no pending message. See mxAPI Error Codes.

Programming Notes

API `mxRecv()` does not wait for incoming messages. It checks the queue and if one is present reads it and returns. This is a non-blocking call. Applications can wait on pipe event and then read the waiting message using this API.

mxAPI Error Codes

mxAPI errors are negative and returned by the API. The list presented here is not exhaustive and is subject to change.

Error ID	Error Value	Description
EMX_PIPE_NAME	-2001	Pipe name too long. May not exceed 8 characters.
		Pipe name has invalid non-ASCII characters.
EMX_MSG_DEPTH	-2002	The message depth parameter should be in the range 1 to 10. Any value outside this range will result in this error.
EMX_PIPE_NOMEM	-2003	No memory to create pipe.
EMX_PIPE_MATCH	-2004	Pipe name does not match currently open named pipe.
EMX_PIPE_HANDLE	-2005	Pipe handle not valid.
EMX_PAYLOAD_SIZE	-2006	Payload size should positive non-zero value.

Message Formatting mfAPI

Applications must have common formats in order to exchange data. Applications using Message Exchange API (mfAPI) format data as a TLV list during the exchange process.

The data exchanged between applications usually consists of a list of name value pairs. This combination of name and value will be referred as TLV or its expansion Tag, Length and Value. TLV is also referred as Type, Length and Value. The name is different but makes no difference to the concept.

The Message Formatting API provides the necessary functionality to create a TLV list and the reverse, i.e., convert a TLV List to individual TLVs. Consider two applications, a client and server application. The client application obtains service from the server application by sending a request message and the server application responds by sending the response message. The client application creates a request message as a TLV list and dispatches it to the server application. The server sends a response message as a TLV list and the client application converts it to individual TLVs. The Message Formatting API provides API for constructing the request message as a TLV list and to analyze the response message as individual TLVs.

Tag, Length & Value (TLV)

A TLV consists of a fixed size Tag name size field followed by a variable length Tag name field. This pair of fields is followed by a fixed size Value length field and terminated by the variable length Value field. Representing it as a sequence of fields:

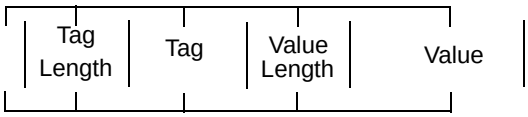


Figure 4 Fixed Length TLV

The Tag Length and Value Len fields are fixed sizes while the Tag and Value fields are variable size. For example if Tag name length is 9 the size of the Tag is 9. Taking this example further, the string “CITY_NAME” is 9 characters long.

The Value field is a variable length field of data whose size is dictated by the Length field.

Consider this example:

```
#define TAG_CITY_NAME      “CITY_NAME” // Tag for city name, 9bytes
```

The city name tag is “CITY_NAME” and its value is "BOSTON" which is 6 characters long and the length is 6. The TLV for this example represented as:

```
09 | CITY_NAME | 06 | BOSTON
```

The '|' separator and spaces are only for readability.

Variable length tags are similar and are represented with an additional Tag length field. This is depicted in the figure below:

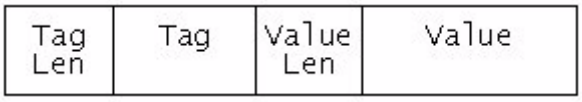


Figure 5 Variable length TLV

The Tag Length field is fixed length field (1 byte) and specifies the size of the subsequent Tag field. The size of the Tag field is variable and is equal to the length as specified in the Tag Length field.

The Value Length and Value fields are identical as described in Fixed Length TLVs.

TLV List

This section describes the message format. The TLV list is a sequence of TLVs preceded by a Header (Figure 6), or a collection of Variable Length TLVs (Figure 7). The Tag Size field in the Header determines if the Tag List is a collection of either Fixed or Variable length TLVs. If the Tag Size field is non-zero then it is a Fixed Length TLV List otherwise it a Variable Length TLV List.

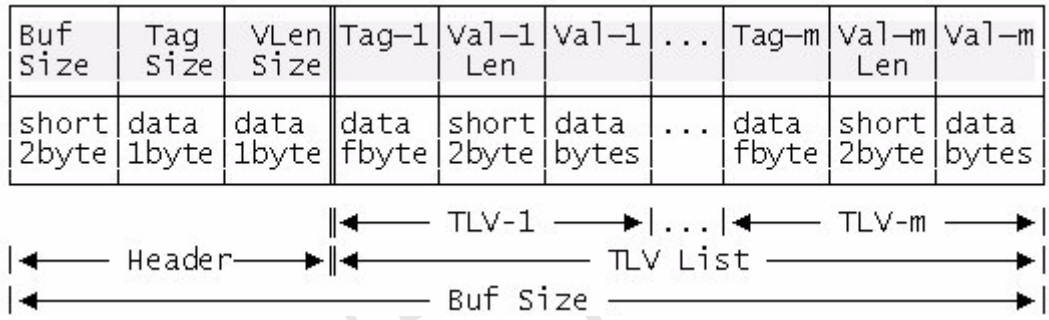


Figure 6 Fixed length TLV list

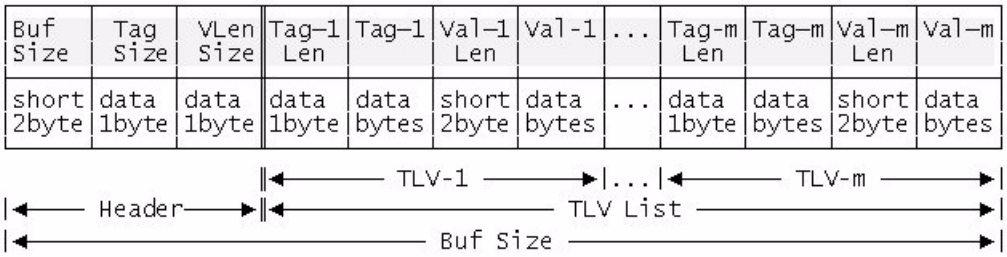


Figure 7 Variable length TLV list

The header consists of one field:

Field Name	Data Type	Size	Description
Buffer Size	short	sizeof(short) 2 bytes	Size of the Message - computed by adding the size of the header (2 bytes) to the size of the TLV List. For example, if the size of TLV List is 97 bytes then the value of this field is 99 bytes.
	byte	2 bytes	

Message Formatting mfAPI – Summary

The Message Formatting API consists of the functions listed in the table below. The following section has detailed description of each API. The API names are hyperlinked to the description.

API	Description
Handle Management	
`mfCreateHandle`	Obtain a new handle
`mfDestroyHandle`	Release a handle previously created by `mfCreateHandle`.
Add Operations	
`mfAddInit()`	Initializes the Message buffer for Add operations

`mfAddClose()`	Closes the session and returns length of Message buffer.
`Error! Reference source not found.`	
`mfAddTLV()`	Add a fixed length TLV to the Message buffer.
`mfDelTLV()`	Removes fixed length TLV from Message buffer if Tag name matches.
Var Length Tags	
`mfAddVarTLV ()`	Add a variable length TLV to the Message buffer.
`mfDelVarTLV()`	Removes variable length TLV from Message buffer if Tag name matches
Fetch Operations	
`mfFetchInit()`	Initializes the Message buffer for Fetch (read) operations and returns the session handle.
`mfFetchClose()`	Closes the session for *fetch* operations.
`mfFetchReset()mfFetchReset()`	Repositions the “Next” pointer to the top of the TLV list.
`mfPeekNextTLV()`	
`mfFetchNextTLV()mfFindTLV()`	Fetches the next TLV in sequence from list. Moves the “Next” pointer to the next TLV in the list.
`mfFindTLV()mfFetchNextTLV()`	Fetch the first TLV from list that matches tag.
`HELPER FUNCTIONmfPeekNextTLV()`	
`mfEstimate()`	Compute memory requirement for a set of tags.
`mfFindVarTLV()`	Fetches the first TLV from list that matches tag.
`mfFetchNextVarTLV()`	Fetches the next TLV in sequence from list. Moves the “Next” pointer to the next TLV tag in the list.
`mfPeekNextVarTLV()`	Obtains the next Tag length and the size of its Value. The “Next” pointer is not advanced.

mfCreateHandle()

Obtain a new handle.

Prototype

```
void *mfCreateHandle(void);
```

Parameters

Return Values

`hF (non-zero)`	Operation successful. Pointer to session handle.
`0`	Operation failed.

Programming Notes

This must be the first API. All other API require session handle (hF). Use this handle for `mfAddInit()` or `mfFetchInit()`. Release the handle using `mfDestroyHandle()`.

mfDestroyHandle()

Release a handle previously created by `mfCreateHandle()`.

Prototype

```
void mfDestroyHandle(void *hF);
```

Parameters

Return Values

`hF`	Session Handle created by `mfCreateHandle()`.
`None`	Operation successful. Pointer to session handle.
`0`	Operation failed.

Programming Notes

This must be the first API. All other API require session handle (hF). Release the handle using `mfDestroyHandle()`.

mfAddInit()

Initializes the handle and input buffer for Add operations.

Prototype

```
int mfAddInit(const void *hF, char *tlvBuffer, const unsigned short
tlvBufferSize);
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()`.
In:	*tlvBuffer	Pointer to TLV buffer. TLVs are added in this buffer. This may be NULL.
In:	tlvBufferSize	Size of tlvBuffer in bytes.

Return Values

= 0	Success. This handle and buffer are initialized for add operation.
< 0	Invalid session handle or invalid operation or invalid parameters. See mfAPI Error Codes.

Programming Notes

This must be the first API before calling `mfAddTLV()`. The session must be closed with `mfAddClose()`.

mfAddClose()

Closes the Session Handle and returns the length of tlvBuffer. The tlvBuffer is ready for further processing after mfAddClose.

Prototype

```
int mfAddClose(const void *hP)
```

Parameters

In:	hp	Session Handle returned by `mfCreateHandle()` and initialized for *Add* operations by `mfAddInit()`.

Return Values

> = 0	Successful closure. Returns length of `tlvBuffer`.
< 0	Invalid handle or invalid operation. See mfAPI Error Codes.

Programming Notes

After a successful `mfAddClose()`, the handle is no longer available for Add operations. To reuse the handle, it should be initialized again with either `mfFetchInif()` or `mfAddInit()`.

mfAddTLV()

Add TLV to buffer.

Prototype

```
int mfVarTLV(const void *hP, const char *tag, unsigned short tagLen, const
char *value, const unsigned short valueLen);
```

Parameters

In:	hP	Add operation Session Handle returned by `mfAddInit()`.
In:	*tag	Pointer to Tag field. The size of this fixed size tag is specified via parameter tagSize in API `mfAddInit()`. This parameter may not be NULL.
In:	valueLen	Length of value field. Its value must be greater than zero.
In:	*value	Pointer to value field. Its length must be as specified in parameter valueLen.

Return Values

`0`	Success. Tag added to `tlvBuffer`.
`< 0`	Invalid session handle or parameter valueLen exceeds limits or no space available in `tlvBuffer`. See mfAPI Error Codes.

Programming Notes

This API must be called after `mfAddInit()`. Successful execution results in adding the TLV in to the tlvBuffer. `mfAddTLV` does not check for duplicates and does not overwrite an existing Tag with the same name.

`mfAddInit()` must be initialized with `tagSize > 0` to support fixed length tags.

mfDelTLV()

Removes first instance of TLV if tag present.

Prototype

```
int mfDelTLV(const int hP, const char *tag)
```

Parameters

In:	hP	Session Handle returned by mfCreateHandle() and initialized for Add operations by `mfAddInit()`.
In:	*tag	Pointer to Tag field. The size of this fixed size tag is specified via parameter tagSize in API `mfAddInit()`.

Return Values

`0`	Success. Tag removed if present.
< 0	Unknown session handle. See mfAPI Error Codes.

Programming Notes

`mfAddInit()` must be initialized with `tagSize > 0` to support fixed length tags.

mfAddVarTLV()

Add variable length TLV.

Prototype

```
int mfAddVarTLV(const int hP, const char *tag, const unsigned short
tagLen, const unsigned short valueLen, const char *value)
```

Parameters

In:	hP	Add operation Session Handle returned by `mfAddInit()`.
In:	*tag	Pointer to Tag field. The size of this fixed size tag is specified via parameter tagSize in API `mfAddInit()`.
In:	tagLen	Size of field tag.
In:	*tag	Pointer to Tag field. Its size must be as specified in parameter tagLen. May not be NULL.
In:	valueLen	Size of value field.
In:	*value	Pointer to Value field. Its size must be as specified in parameter valueSize. May not be NULL.

Return Values

0	Success. Tag added.
< 0	Invalid session handle or parameter valueLen exceeds limits or no space available in tlvBuffer or invalid operation. See mfAPI Error Codes.

Programming Notes

This API must be called after `mfAddInit()`. Successful execution results in adding the TLV to tlvBuffer. `mfAddTLV()` does not check for duplicates and does not overwrite an existing Tag with the same name.

mfDelVarTLV()

Removes first instance of tag if present.

Prototype

```
int mfDelVarTLV(const int hP, const char *tag, const unsigned short tagLen)
```

Parameters

In:	hP	Add operation Session Handle returned by `mfAddInit()`.
In:	*tag	Pointer to Tag field. The size of this fixed size tag is specified via parameter `tagSize` in API `mfAddInit()`.
In:	tagLen	Size of field tag

Return Values

`0`	Success. Tag removed if present.
`< 0`	Unknown Session Handle or tagLen exceeds 255. See mfAPI Error Codes.

Programming Notes

`mfAddInit()` must be initialized with `tagSize=0` to support variable length tags.

mfFetchInit()

Initializes the handle and tlvBuffer for Fetch operations. It positions the next pointer at the top of the TLV list.

Prototype

```
int mfFetchInit(const void *hF, const char *tlvBuffer)
```

Parameters

In:	hF	Session Handle returned by mfCreateHandle().
In:	tagSize	Pointer to message buffer. TLVs are fetched from this buffer. This parameter cannot be NULL.

Return Values

= 0	Success. The handle is initialized for Fetch operations.
< 0	Invalid parameters or operation not supported. See mfAPI Error Codes.

Programming Notes

This must be the first API for fetch operations – `mfFetchClose()`, `fmFetchReset()`, `mfPeekNextTLV()`, `mfFetchNextTLV()`, `mfFindTLV()`. The Fetch session must be closed with `mfFetchClose()`.

mfFetchClose()

Closes the Session Handle for fetch operations.

Prototype

```
int mfFetchClose(const void *hF)
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()` and initialized for *Fetch* operations by `mfFetchInit()`.

Return Values

0	Successful closure.
< 0	Invalid handle or operation. See section 12 for list of error codes. See mfAPI Error Codes.

Programming Notes

After successful `mfFetchClose()` call, the handle should be initialized again for with either `mfFetchInif()` or `mfAddInit()`.

mfFetchReset()

Repositions the “Next” pointer to the top of the TLV list.

Prototype

```
int mfFetchReset(const void *hF)
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()` and initialized for *Fetch* operations by `mfFetchInit()`.

Return Values

`0`	Successful closure.
`< 0`	Invalid handle or no operation. See mfAPI Error Codes.

mfPeekNextTLV()

Obtains the next Tag length and the size of its Value. The “Next” pointer is not advanced.

Prototype

```
int mfPeekNextTLV(const void *hF, unsigned short *tagLen, unsigned short
*valueLen)
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()` and initialized for *Fetch* operations by `mfFetchInit()`.
Out:	tagLen	Pointer to `tagLen`. The length of tag is set on return. This parameter must not be NULL.
Out:	valueLen	Pointer to `valueLen`. The length of value field is set on return. This parameter must not be NULL.

Return Values

`0`	Successful operation.
`< 0`	End of TLV list, no TLV returned or incorrect parameters or invalid operation. See mfAPI Error Codes.
`0`	Successful match.
`< 0`	No match. No TLV returned. See mfAPI Error Codes.

Programming Notes

mfFindTLV searches for tags from the top of the TLV list and returns on the first successful match. The “Next” pointer is not moved.

mfFetchNextTLV()

Fetches the next TLV from the list. Moves the “Next” pointer to the next TLV tag.

Prototype

```
int mfFetchNextTLV(const void *hF, const unsigned short tagSize, char
*tag, unsigned short *tagLen, const unsigned short valueSize, char *value,
unsigned short *valueLen);
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()` and initialized for *Fetch* operations by `mfFetchInit()`.
In:	tagSize	Size of buffer tag in bytes. Its value must be greater than zero.
Out:	tag	Pointer to tag populated by `mfFetchNextTLV`. The size of this tag is returned by parameter tagLen. This parameter must not be NULL.
In:	valueSize	Size of buffer value in bytes. Its value must be greater than zero.
Out:	value	Value of tag. `mfFindVarTLV` sets this parameter when a match is found. Its length is as returned by valueLen. This parameter must not be NULL.
Out:	valueLen	Size of value in bytes. `mfFindVarTLV` sets this parameter when a match is found. This parameter must not be NULL. Its value does not exceed valueSize.

Return Values

`0`	Successful match and tag returned.
`< 0`	No match. no TLV returned. See mfAPI Error Codes.

Programming Notes

mfFindVarTLV searches for tags from the top of the TLV list and returns on the first successful match. The tag lengths are compared first and if equal, the tags are matched.

mfFetchNextVarTLV()

Fetches the next TLV from list. Moves the “Next” pointer to the next TLV tag in the list. Use this API for variable length tags.

Prototype

```
int mfFetchNextVarTLV(const int hF, char *tag, unsigned short *tagLen,
const unsigned short valueSize, unsigned short *valueLen, char *value)
```

Parameters

In:	hF	Fetch operation Session Handle returned by `mfFetchInit()`.
Out:	tag	Pointer to tag populated by `mfFetchNextVarTLV`. The size of this tag is returned by parameter tagLen. This parameter must not be NULL.
Out:	tagLen	Pointer to `tagLen`. The size of parameter tag is returned. This parameter must not be NULL.
In:	valueSize	Size of buffer value in bytes. Its value must be greater than zero.
Out:	value	Size of value in bytes. `mfFetchNextTLV` sets this parameter when a match is found. This parameter must not be NULL.
Out:	valueLen	Pointer to valueLen set by `mfFetchNextTLV`. This parameter must not be NULL. . Its value does not exceed `valueSize`.

Return Values

`0`	Successful fetch. Tag and value returned.
`< 0`	End of TLV list, no TLV returned or invalid TLV operation. See mfAPI Error Codes.

Programming Notes

The “Next” pointer moves to the next TLV in the TLV List.

mfFindTLV()

Find and fetch the first TLV from list that matches tag.

Prototype

```
int mfFindTLV(const void *hF, const char *tag, const unsigned short
tagLen, const unsigned short valueSize, char *value, unsigned short
*valueLen,)
```

Parameters

In:	hF	Session Handle returned by `mfCreateHandle()` and initialized for *Fetch* operations by `mfFetchInit()`.
In:	tag	Pointer to tag to fetch. `mfFindTLV` will search for tag that matches the one pointed by parameter tag. This parameter must not be NULL.
In:	tagLen	Length of field tag. This field must be great than zero.
In:	valueSize	Size of buffer value in bytes. Its value must be greater than zero.
Out:	value	Value of tag. `mfFindTLV` sets this parameter when a match is found. Its length is as returned by valueLen. This parameter must not be NULL.or incorrect parameters. See mfAPI Error Codes.
Out:	valueLen	Length of value in bytes. `mfFindTLV` sets this parameter when a match is found. This parameter must not be NULL. The returned value will not exceed `valueSize`.

Return Values

`0`	Successful match and tag returned.
`< 0`	No match or invalid operation. No TLV returned. See mfAPI Error Codes.

Programming Notes

mfFindTLV searches for tags from the top of the TLV list and returns on the first successful match. The tag lengths are compared first and if equal, the tags are matched.

mfEstimate()

Compute memory requirement for a set of tags.

Prototype

```
int mfEstimate(const unsigned short tagCount, const unsigned short
sigmaTag, const unsigned short sigmaValue);
```

Parameters

In:	tagCount	Number of tags.
In:	sigmaTag	Size of all the tag fields.
In:	sigmaValue	Size of all the value fields.

Return Values

> 0	Estimate of memory buffer to accommodate all the tags.

Programming Notes

`mfAPI mfAddInit()` requires a buffer (tlvBuffer) of size (tlvBufferSize). The size of the buffer is dependent on the number of tags, their cumulative size and the cumulative size of the value fields. The API provides the size of the TLV buffer required to accommodate all tags.

mfAPI Error Codes

mfAPI errors are negative and returned by the API. The list presented here is not exhaustive and is subject to change.

Error ID	Error Value	Description
EMF_TAG_SIZE	-3001	Tag size should be greater than zero.
EMF_VALUE_SIZE	-3002	Value size should be greater than zero.
EMF_PARAM_NULL	-3003	Parameter is null when one is not expected.
EMF_PARAM_INVALID	-3004	Parameter is invalid or out of range
EMF_TLV_BUF_SIZE	-3005	The buffer size is less than minimum required size
EMF_UNKWN_HANDLE	-3006	Unknown session handle
EMF_BUFFER_FULL	-3007	No space in buffer to add tag.
EMF_OP_NOT_SUPP	-3008	Operation not supported.
EMF_TAG_NO_MATCH	-3009	No matching tag was found.
EMF_TAG_EOL	-3010	End of list. No more tags in list.

Verix EOS Volume II Communication Engine Application (VXCE.out)

The Verix EOS Volume II Communication Engine or CommEngine (VXCE.OUT) is the core component of the communication infrastructure. The boot strap application VXEOS.OUT starts VXCE.OUT which in turn starts the rest of the components that constitute the communication infrastructure consisting of device drivers and the TCPIP stack. Both these components have defined interfaces that CommEngine uses.

Prior to starting the device drivers, CommEngine needs to determine the right drivers to load. It identifies the communication devices present on the terminal and then loads the right device driver. One of the key objectives of the Verix EOS Volume II is to minimize the change when new communication devices are introduced. The purpose of this identification apart from the obvious purpose is to provide a level of abstraction to CommEngine and insulate it from future changes to new device introduction.

CommEngine also provides application services, i.e. it applications can obtain network events and status, configure and query device drivers, manage the network connection, etc. Applications must link with the CommEngine Interface Library (CEIF.LIB) for these services.

CommEngine Bootstrap Process

Use the following sections to employ the CommEngine bootstrap process.

CommEngine Invocation

On start up (both cold and warm boot) the OS starts Verix EOS Volume II . Verix EOS Volume II starts CommEngine and the executable files specified in the Verix EOS Volume II manifest file. The manifest contains VxNCP which is the application with the User Interface (UI) and provides configuration and management services to Verix EOS Volume II components. CommEngine after start up brings up the device driver and the Treck TCPIP stack. A point to note here is that the CommEngine boot straps the communication infrastructure. VxNCP though part of the communication infrastructure is started independently by Verix EOS Volume II via the manifest file.

Conditionally Starting CommEngine

On startup Verix EOS Volume II looks for configuration variable *VCE in GID1. If this variable is not present or its value is non-zero, CommEngine is started. If *VCE is set to 0, CommEngine is not started. Note that *VCE is a Verix EOS Volume II configuration parameter and is referred here as it affects CommEngine.

To reiterate, CommEngine may be conditionally started but VxNCP is always started. VxNCP is sensitive to this fact and will continue to run even if CommEngine is not running.

CommEngine on Predator platform

On the Predator platform space (RAM and FLASH) is a premium and using it conservatively takes precedence over clean and elegant design nuances. The scope of Verix EOS Volume II on the Predator platform is limited to the communication infrastructure. Since there are no plans to add more components to the Verix EOS Volume II infrastructure the sole purpose of starting Verix EOS Volume II is limited to starting CommEngine (VxCE).

Since space is a premium, Verix EOS Volume II will be dispensed with and the OS will directly start CommEngine (VXCE.OUT). CommEngine on start up will start VxNCP. CommEngine will examine parameter *VXC and if disabled will exit. Refer to Conditionally Starting CommEngine for details on parameter *VXC.

CommEngine Startup Operations

CommEngine Device Management

CommServer on start up grabs the communication device and starts the necessary device drivers and the TCPIP stack. In very rare cases the communication device needs to be exchanged with other applications that are VMAC complaint. In such situations, CommEngine needs to know that it should not grab the communication device and wait for a notification before it opens the communication device.

On start up CommEngine looks at variable *CEDM (CommEngine Device Management) in GID1. If not present or set to zero (*CEDM=0) then CommEngine will consider this as normal operation and open the communication devices. If *CEDM is non-zero then CommEngine waits for devices to be provided to it

For complete details on device management refer to App VCCESA.OUT FRD, see Related Documentation.

CommEngine Pipe – CEIF

Applications use API provided by CEIF.LIB to obtain CommEngine services. This API is designed to exchange pipe messages with CommEngine. CommEngine is listening on a named pipe – CEIF.

CommEngine creates named pipe CEIF at start up.

CommEngine Configuration Files

CommEngine maintains its configuration in two configuration files:

Configuration Filename	Description
VXCEC.CFG	Configuration data file. This file contains the name and value of the configuration variable. A change to configuration value updates this file.
VXCEM.MTD	Configuration metadata file. This file contains the metadata associated with a configuration variable. This file will primarily used by VxNCP.

The structure of the configuration file and its access mechanism is described document titled Configuration Management FRD, see Related Documentation.

CommEngine & Download Support

In the Verix EOS Volume II architecture document, when a system mode download is initiated, OS System Mode starts CommEngine which in turn starts VxNCP.

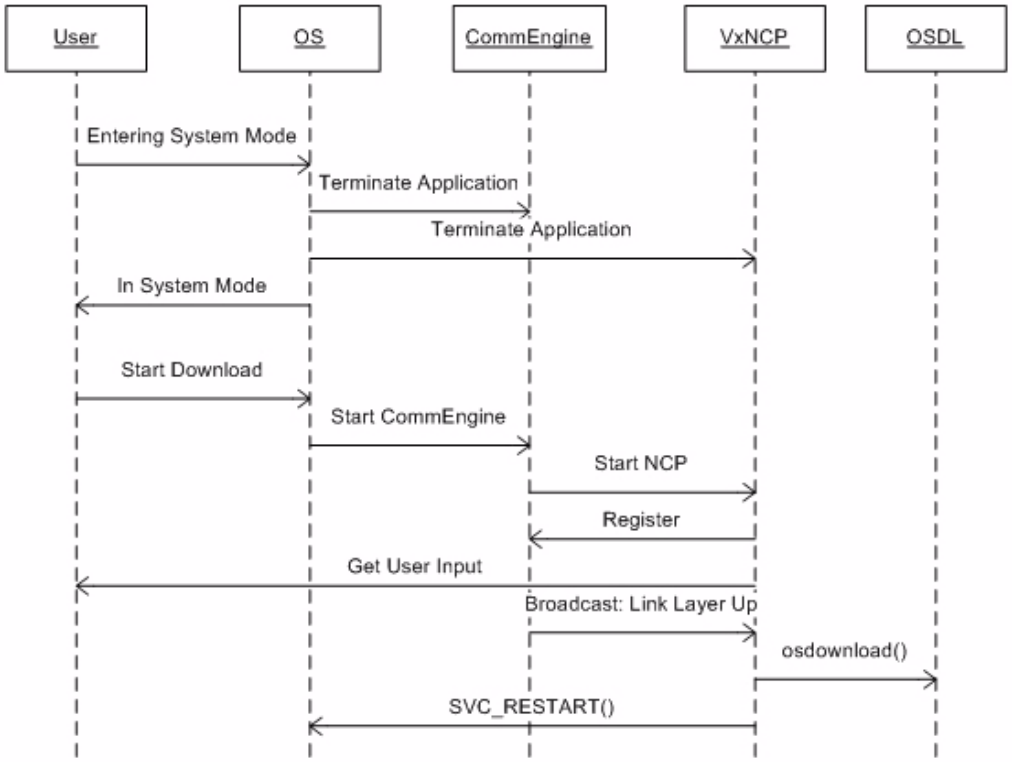


Figure 8 Flow for Download Support

OS System Mode

OS System Mode current provides the option for TCPIP downloads.

The user selects TCPIP downloads by selecting TCPIP option. On selecting this option, the application specified in *ZTCP is executed. If parameter *ZTCP is not specified then CommEngine is initiated with the same parameters:

```
run("N:VXCE.OUT", "arg1 arg2");
```

Where:

- arg1 – download type: either "F" for full, or "P" for partial.
- arg2 – download group: for example, "01" for group 1. The group will always be two digits to simplify parsing the user application.

CommEngine & VxNCP

Seeing arguments CommEngine assumes it is a download and consequently it runs VxNCP with the exact same arguments.

```
run("N:VXNCP.OUT", "arg1 arg2");
```

VxNCP seeing arguments will run in download mode. This implies that it provides seamless user experience from OS System Mode. VxNCP will display the screens to enter the download parameters *ZA, *ZT and *ZP if not present.

Verix EOS Volume II supports SSL and consequently SSL downloads. VxNCP will display a screen for the user to select SSL download.

A new download configuration parameter is proposed *ZSEC (for security) and will take these values:

*ZSEC	Description
1	TCPIP download
2	SSL download

VxNCP will look for the *ZSEC and proceed if the value is present and understood failing which it will display a screen seeking user input.

Interface with Device Drivers.

CommEngine starts and manages the device drivers. All device drivers work under the aegis of the Device Driver Manager. This is described in the document titled DDI Driver ERS, see Related Documentation.

The Device Driver Manager implements the Device Driver Interface (DDI). CommEngine uses the DDI to manage it. This is described in detail in the CommEngine DDI Integration Guide, see Related Documentation.

Interface with Treck TCPIP Library.

Similar to the DDI, CommEngine uses the Treck TCPIP library to manage it. Management involves add /remove, configure and monitor network interfaces.

This API is described in Verix EOS Volume II Network ERS, see Related Documentation.

Application Interface

Application services are provided by CommEngine via CEIF.LIB. Applications link with CEIF.LIB which under the covers interfaces with CommEngine. The complete API and detailed description of CEIF.LIB is described in document titled Verix EOS Volume II CommEngine Interface Library,see Related Documentation.

Packaging

Verix EOS Volume II Filename	Verix EOS Volume II Location	Download Filename with Suffix	Download Location	File Type
VXCE.OUT	N:	VXCE.OUT{!	F:1	Program
VXCE.P7S	I:46	VXCE.OUT{!.P7S	I:1	Signature
VXCEM.MTD	N:	VXCEM.MTD{!	F:1	Configuration Metadata file
VXCEM.P7S	I:46	VXCEM.MTD{!.P7S	I:1	Signature
VXCEC.CFG	I:46	VXCEC.CFG{!	I:1	Configuration Data file
VXCEC.P7S	I:46	VXCEC.CFG{!.P7S	I:1	Signature

Network Control Panel (NCP)

VxEOS is designed to be self contained, including user interface for administration, monitoring, diagnostics, configuration and setup tasks. This approach is implemented providing Network Control Panel as the default user interface for users to interface with different VxEOS components.

The Network Control Panel consists of multiple functional modules collectively known as the Services Layer. This layer interacts directly with the components in VxEOS. The user interface modules that will be present in the VxEOS are:

- Configuration, Status & Management
- Software Downloads
- Network Diagnostics and Logging
- Device Drivers Configuration

The following diagram represents how Network Control Panel integrates VxEOS components to implement the functionality listed above.

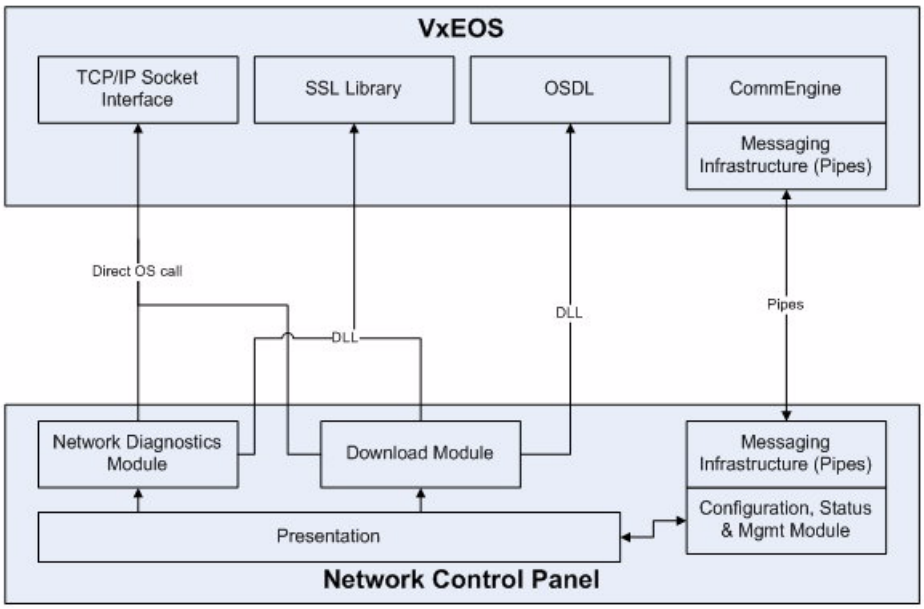


Figure 9 Network Control Panel

Startup Operation

By default VxEOS will start NCP executable. As later described on this document, specific customizations may be implemented overriding the default behavior of NCP, or completely replacing it with a new implementation.

Because default NCP components are bundled within VxEOS, clearing the whole application space and CONFIG.sys variables only erases the custom NCP; upon restart VxEOS will revert back to its default NCP solution. Full details will be covered in the next sections.

NCP is provided as default UI for all VxEOS components. Alternative solutions may require implementing different features and functionality for specific markets; with this in mind VxEOS' NCP execution may be turned off.

Following sections summarize how VxEOS starts NCP, also how NCP will be available on different scenarios.

Starting NCP executable

On startup OS starts VxEOS master application. This application is responsible for starting all other VxEOS components, including NCP.

For those implementations where NCP will be replaced or turned off, CONFIG.sys variable *VXNCP on GID1 should be used.

On startup NCP will look for configuration variable *VXNCP on GID1. If this variable is set to 0 (zero), NCP will exit immediately. If *VXNCP is not present or set to a non-zero value, NCP will run normally.

On Startup, NCP will create all communication channels to operate with other EOS components.

- Registers with CommEngine via ceAPI.
- Creates named PIPE "VXNCP" which purpose is detailed below on section Invoking and Exiting NCP
- Registration with VMAC is outside NCP responsibility and will be handled by VMACIF, as described on Running under VMAC environment section.

Once NCP has initialized and completed registration/connection with other components, it waits until it gets activated as described on Error! Reference source not found. section.

Interoperability

The succeeding sections describe how NCP communicates and interacts with other components on the system.

CommEngine

On Startup, NCP will:

- Register with CommEngine is made via ceAPI.
- Create named PIPE "VXNCP" which purpose is discussed on the Invoking and Exiting NCP section.

In case registration with CommEngine fails, NCP will limit its functionality to those operations not requiring communication with CommEngine or the Device Drivers.

Device Ownership

I. Console ownership

As described on Invoking and Exiting NCP section, it is important for NCP to return the CONSOLE to the same application that triggered its activation. For this purpose, NCP will keep ownership of the CONSOLE device until the user explicitly selects “Exit” from the UI.

To prevent user to abruptly switch application, NCP will disable hotkey on activation using OS API disable_hot_key(). When user selects “Exit” or “Cancel Key” on IDLE, hotkey functionality will be restored by calling OS API enable_hot_key().

II. Printer ownership

Normal NCP operations only require the CONSOLE, while for some menu options user has the possibility to print the same information shown on the display. This directly implies getting control over the PRINTER.

NCP will not prevent its activation until printer becomes available, instead it will only attempt opening the PRINTER once the user selects the “Print” option. If open fails user will be properly notified and NCP will continue its regular operations.

Application invoking NCP is responsible for making PRINTER device available for NCP. Under no circumstances, NCP will retry obtaining the printer neither will retry the print operation. User will have to manually retry the “Print” option from the UI.

For VMAC environments, PRINTER will be mandatory requirement for NCP activation and this will be handled by an external application. PRINTER will be listed on the ACTIVATE event of VMACIF application, who will be responsible for activating NCP. More details are detailed on the specifications VDN 28810 listed on the References section.

Invoking and Exiting NCP

I. Invoking NCP

The invoking application only needs to call ceAPI – `ceActivateNCP()` to activate NCP. Upon return from this API, caller application does not own the CONSOLE.

- Identifying NCP’s Task ID to assign CONSOLE ownership.
- Communicating Task ID of the Invoking Application; required to return CONSOLE to the same originator.
- Transporting caller Task ID to NCP via named PIPE “VXNCP”.

II. Returning to Invoking Application

When user selects “Exit” or presses “Cancel Key” on Idle Screen, NCP will return CONSOLE ownership to the original invoking application. Task ID of the originator was received during activation process and received via named PIPE.

III. Control Flow between invoking application and NCP

The figure below depicts the flow between the Invoking Application and NCP in both directions, i.e., when NCP acquires control and when it relinquishes control. It follows this sequence of steps:

1. Invoking Application calls ceAPI ceActivateNCP().
2. API ceActivate()
   a. Disables hot key by calling OS API `disable_hot_key()`.
   b. Obtains task Id of named pipe “VXNCP” via OS API `get_owner()`
   c. Sends message to name pipe “VXNCP”. This notifies NCP the task Id of the invoking application.
   d. Calls OS API `activate_task()`.
      i. OS posts event `EVT_DEACTIVATE` to invoking application.
      ii. OS posts event *EVT_ACTIVATE* to NCP
   e. API ceActiviate() returns to calling application.
3. User is ready to exit NCP.
4. NCP enables hot key by calling OS API `enable_hot_key()`.
5. NCP calls OS API `activate_task()`.
   a. OS posts event `EVT_DEACTIVATE` to NCP.
   b. OS posts event `EVT_ACTIVATE` to invoking application

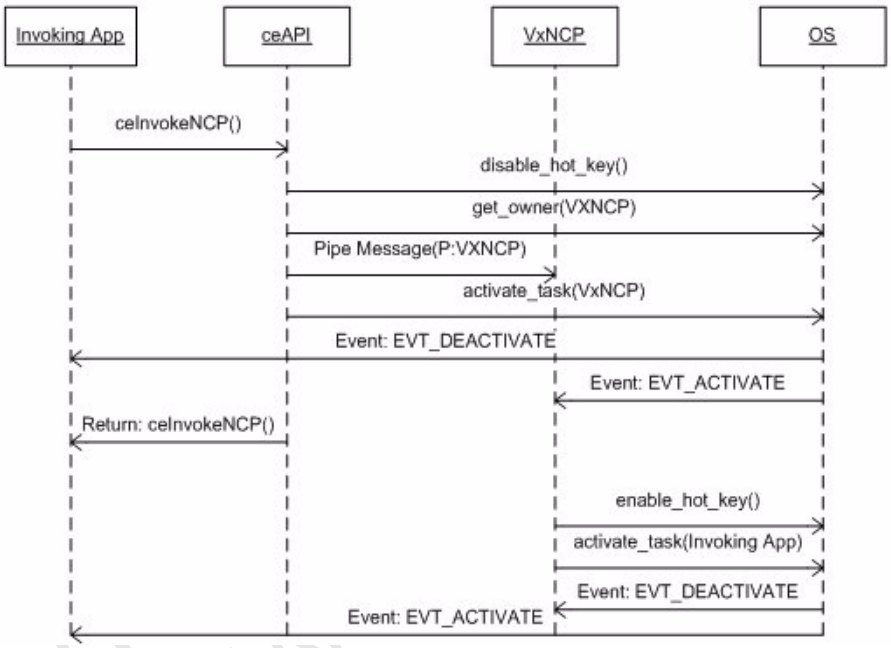


Figure 10 Control Flow – Invoking and Exiting VxNCP

IP downloads from System Mode

An empty terminal needs the ability to download applications. OS will continue to handle downloads via dial, while VxEOS becomes the engine responsible to handle IP downloads.

When user selects TCPIP downloads in System Mode, if *ZTCP is not configured, OS will run CommEngine (responsible for initializing Device Driver and IP Stack) and CommEngine will initiate NCP (responsible for running download protocol over TCP/SSL connection)

OS communicates two parameters when invoking the application to handle IP downloads; CommEngine will communicate same parameters to NCP

- arg1 – download type: either “F” for full, or “P” for partial.
- arg2 – download group: for example, “01” for group 1. The group will always be two digits to simplify parsing the user application.

NCP seeing arguments will run in download mode. Similar to Download, NCP will display the screens to enter the download parameters *ZA, *ZT and *ZP if not present.

Additionally, IP downloads from VeriCentre may be secured with SSL or not. NCP will display a screen for the user to select SSL download. Similar to other variables, user selection will be saved for future download operations. New download configuration parameter *ZSEC (for security) will be used. If *ZSEC already exists, future downloads will not prompt user again.

The following diagram depicts flow to handle IP downloads from System Mode.

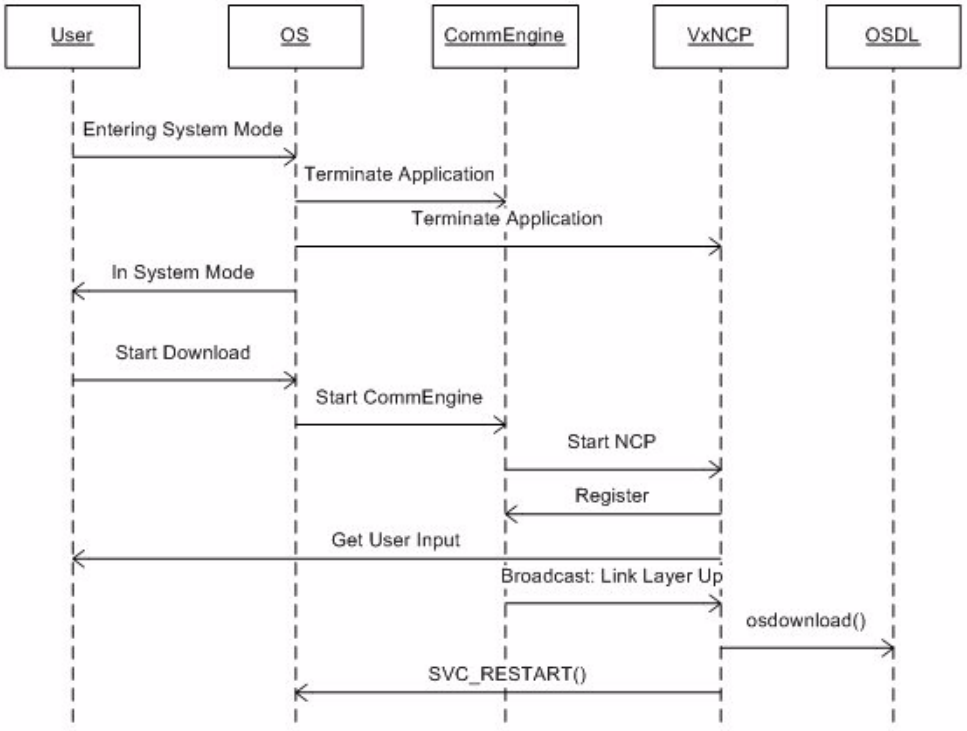


Figure 11 IP downloads from System Mode

Running under Single-Application Mode

No specific action is required. By default NCP will be running in the background waiting to be activated as described above.

Running under VMAC Environment

The expectation is that NCP will be invoked via VMAC menu, but since NCP runs before VMAC, NCP cannot register with VMAC/IMM. Direct implication of this sequence of execution is that NCP is not aware of VMAC.

A surrogated application VMAC compliant will be created to interface with other applications on NCP's behalf. Whenever user selects the surrogated application from VMAC menu it will activate NCP immediately.

Since VMAC is an optional component for Vx solutions, surrogated application will not be installed as part of the default VxEOS bundle. Instead it must be installed during deployment and customization for specific solutions.

Full details about NCP requirements under VMAC environment are detailed on the specifications VDN 28810 listed on References section.

User Interface

The default language for all UI prompts will be English. To facilitate portability and use on different markets, following considerations will be revised during development:

- Same font file will be used across all prompts and menus.
- All prompts will be external to the binary executable. Replacement prompts may be configured.

- To reduce the number of prompts, when possible icons will be used to represent the operation to perform. Icons will not be replaceable.

NOTE

Prompts will be limited to ASCII set supported by default OS fonts. No Unicode support.

TBD - Prompts file format

Customization

VxEOS installation will include its own fonts to display and print the default prompts in English. To customize NCP, “font files” and “prompts file” may be provided during deployment.

Prompts

On startup NCP will look for configuration variable `*VXNCPPROMPTS` on GID1 to retrieve the custom prompts file path. In order for the file to become the default source of prompts, all the following verifications must be passed:

- Variable defined
- File name points to an existing file

If any of these verifications fail, `NCPprompts.dat` on GID46 is selected as current prompts file.

NOTE

TBD: Depending on the prompt file format selected, verification steps will include making sure the required number of prompts are available via custom prompts file; mainly to avoid unexpected/garbage strings on the UI.

Display Font

On startup NCP will look for configuration variable `*VXNCPDSPFONT` on GID1 to retrieve the custom font filename to use for Display operations. In order for the file to become the default display font, all the following verifications must be passed:

- Variable defined
- File name points to an existing file
- Character size matches the same used by NCP’s default font
- OS call to `set_font()` succeeds

If any of these verifications fail, NCPDSPfont.fon on GID46 is selected as current font file.

NOTE

Character size flexibility will be revised during development.

Printer Font

On startup NCP will look for configuration variables *VXNCPPRNFONT and *VXNCPPRNTABLE on GID1 to retrieve the custom font filename to use for Printer operations and table ID to load it. In order for the file to become the default printer font, all the following verifications must be passed:

- Both variables defined
- File name points to an existing file
- Character size matches the same used by NCP's default font

If any of these verifications fail, NCPDSPfont.fon on GID46 is selected as current font file.

NOTE

Support for Printer Country Code.

Prompts and Fonts verification

If any of the configuration variables *VXNCPPROMPTS, *VXNCPDSPFONT, *VXNCPPRNFONT, *VXNCPPRNTABLE is defined but either one fails the verification sequence, ALL VARIABLES will be ignored and NCP will revert to its default English prompts and fonts.

Time Format

Time will be configurable via configuration variable *VXNCPTIMEFORMAT where the only valid strings are noted below.

Valid options	Examples
"12"	01:00 p.m.
"24"	13:00

Date Format

Date will be configurable via configuration variables *VXNCPDATEFORMAT and *VXNCPDATESEP. *VXNCDDATEFORMAT specifies order to show Day-Month-Year while *VXNCPDATESEP is used as separator character.

Valid values for *VXNCPDATESEP are '/' (forward slash), '.' (dot) or '-' (dash). If variable is not defined or its value does not match any of the valid strings the default separator '/' will be used.

Valid values for *VXNCPDATEFORMAT are MMDDYYYY, DDMMYYYY, MMMDDYYYY, DDMMMYYY; where month in the format "MM" represents the numeric value while "MMM" represents the first three letters of the month's name. If variable is not defined or its value does not match any of the valid strings the default MMDDYYYY format will be used.

The following table represents the possible combinations based on the acceptable formats noted above.

*VXNCPDATEFORMAT Valid options	Example for Jan 11th 2009based on *VXNCPDATESEP values /	-	.
MMDDYYYY	11.01.2009	11.01.2009	11.01.2009
DDMMYYYY	01.11.2009	01.11.2009	01.11.2009
MMMDDYYYY	Jan/11/2009	Jan-11-2009	Jan.1.2009
DDMMMYYYY	11.Jan.09	11.Jan.09	11.Jan.09

The configuration variable *VXNCPMONTHNAME allows defining the abbreviated string name to use as ‘month’ for the formats MMMDDYYY & DDMMMYYYY. This variable must provide twelve (12) strings, each one up to three (3) characters long, separated by comma.

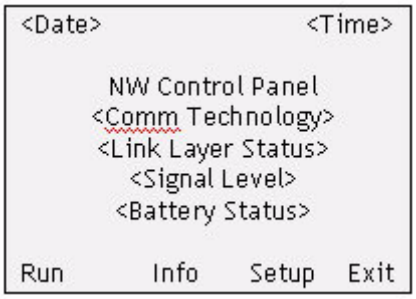
*VXNCPMONTHNAME allows replacing the month names without having to download the whole prompts file. The following sample shows how to use this variable to download month names in Spanish

```
*VXNCPMONTHNAME=Ene,Feb,Mar,Abr,May,Jun,Jul,Ago,Sep,Oct,Nov,Dic
```

*VXNCPMONTHNAME variable is only verified if *VXNCPDATEFORMAT is configured to MMMDDYYY or DDMMMYYYY. If *VXNCPMONTHNAME variable is not defined or it does not contain exactly 12 strings, NCP will revert to its default English names.

Idle Screen

The following information displayed as result of activating NCP. Menu options are organized following latest OS menu distribution.



- Date & Time (following formats documented above)
- “NW Control Panel” as product name
- Primary Communication Technology
- Link Layer Connection Status
- Signal level on wireless devices
- Battery status on portable devices
- Menu selections for
  i. Run
  ii. Terminal Info
  iii. Setup
  iv. Exit

Information will be represented via icon when possible. Based on the final design, position on the screen may differ from the current presentation. Menu entries will be represented at the bottom on the screen and will be associated with the purple keys (function keys a-thru-d)

Once any menu option is selected, all sub-menus will be text-only and options will be selectable via function keys e-thru-h.

NOTE

TBD: For devices without function keys (i.e. Vx 700) alternate approach will be revised and decided during development.

Menu Hierarchy

Main Menu Idle Screen	Sub-menu	Description
Run	Diagnostics	All
		Ping IP address / URL (single or continuous)

		Ping Gateway
		Ping DNS Server
		DNS Lookup
		TCP Socket connect
		SSL Socket connect
	Network Maintenance	Select Network Interface
		Restart communication
		Stop communication
		Start communication
		Network Interface Events Log
		View
		Print
		Purge
	Download	Download application(s) from VeriCentre server(s)
Terminal Info	IP addresses status	Current IP setting / DHCP Lease (if applicable)
	Communication Technology	Specific details depending on the network interfaces available
		Ethernet
		Wi-Fi
		CDMA
		GPRS
		GSM with PPP
		Land-Line with PPP
	No Network Interface for “Land-Line” or “GSM” (without PPP) are supported by VxEOS Versions	Version for VxEOS components
Setup	Primary Network Interface	Default Network Interface for specific operations

Communication Technology	Specific parameters depending on the network interface selected Specific parameters depending on the network interface selected Ethernet Wi-Fi CDMA GPRS GSM with PPP Land-Line with PPP
Default settings may be defined via configuration files. Run-time edition of those files will be available via NCP. Configuration files will follow the format described on the document “Configuration Management” listed on References section. As described in the “Configuration Management” documentation, setup files may have several sets of the parameters listed above. If that is the case, NCP first will prompt for the section to review and next will list the parameters. Additionally, user will have the ability to create a “New Section” and subsequently will be prompted to enter all the parameters (based on the “metadata” information).	IP address /URL for Ping IP:PORT for TCP Socket connect IP:PORT for SSL Socket connect
Diagnostics	
CommEngine	Change CommEngine parameters from setup file

	Default IP address or URL to use from Diagnostics menu, SSL option. IV. CommEngine NCP as default UI for all VxEOS components serves as UI Engine to dynamically modify CommEngine's settings. This menu option is dynamically generated based on the configuration files VXCEC.CFG and VXCEM.MTD, as described on VDN 28809 listed on References section. Device Driver (s)	Change specific settings for all DDI drivers available
Menu: Exit Cancel key	(no sub-menu)	Returns CONSOLE

Menu: Run

This menu groups all menu options requiring operations, either locally to the network interface(s) available or externally over network interface(s) services.

I. Diagnostics

For all following menu options, IP addresses for those operations are previously configured. Refer to Configuration Files and Menu: Setup.

i. All

When selected, it runs all tests listed below automatically. Ping IP address will do a single attempt. After completion, if printer is available, user will also have the option to print the results as shown on the display.

ii. Ping IP address / URL (single or continuous)

When menu option is manually selected, choices to select "Single" vs. "Continuous" will be displayed.

Single

Input	Preconfigured IP address / URL
Action	PING IP address
Result	If PING operation succeeds UI will display total RTT (Round Trip Time) in milliseconds. If PING operation fails UI will show a failure message. Note: Server must support PING feature

Continuous

Input	Preconfigured IP address / URL
Action	Continuously PING IP address until user press “Cancel” Key
Result	Summary screen will show accumulated results
	Successful PINGs / Attempts
	Average RTT (Round Trip Time) in milliseconds
	Accumulated RTT (Round Trip Time) in milliseconds
	Note: Server must support PING feature

iii. Ping Gateway

Input	Gateway IP address from Primary NWIF
Action	PING Gateway’s IP address
Result	If PING operation succeeds, UI will display total RTT (Round Trip Time) in milliseconds.
	If PING operation fails, UI will show a failure message.

iv. Ping DNS Server

Input	Primary DNS Server IP address from Primary NWIF
Action	PING DNS Server’s IP address
Result	If PING operation succeeds, UI will display total RTT (Round Trip Time) in milliseconds.
	If PING operation fails, UI will show a failure message.

v. DNS Lookup

Input	Preconfigured URL
Action	Convert URL to dotted IP address using DNS service
Result	IP address (or addresses) returned by DNS server
	Total resolution time in milliseconds
	If operation fails, UI will show a failure message.

vi. TCP Socket connect

Input	Preconfigured IP address / URL and PORT
Action	Perform TCP socket connect and TCP socket disconnect to the IP:PORT address specified
Result	Total time in milliseconds.
	If operation fails, UI will show a failure message.

vii. SSL Socket connect

Input	Preconfigured IP address / URL and PORT
Action	Perform TCP socket, connect SSL handshake and TCP socket disconnect to the IP:PORT address specified
Result	Total time in milliseconds. If operation fails, UI will show a failure message.

II. Network Maintenance

NCP will detect all Communication Technologies available on the device and will list all names for user to select the Network Interface (NWIF from CommEngine).

Once user selects the network interface to operate, the following options will be listed:

i. Restart communication

`Input`	`Network Interface selected by user`
Action	Use ceAPI to “Stop” and “Start” the specific Network Interface
Result	Total time in milliseconds. If operation fails, UI will show a failure message.

ii. Stop communication

`Input`	`Network Interface selected by user`
Action	Use ceAPI to “Stop” specific Network Interface
Result	Total time in milliseconds. If operation fails, UI will show a failure message.

iii. Start communication

`Input`	`Network Interface selected by user`
Action	Use ceAPI to “Start” specific Network Interface
Result	Total time in milliseconds. If operation fails, UI will show a failure message.

iv. Network Interface Events

CommEngine will record different events generated by the different DDI drivers and TCP stack. NCP provides this menu to read the contents of the LOG file.

- View
- Print
- Purge Log

NCP will use ceAPI to obtain filename of the log file. This file will be used to handle “View” and “Print” operations. Contents of this file will not be manipulated or reformatted by NCP; information will go directly to CONSOLE or DISPLAY. Different ceAPI will be used purge CommEngine’s log file.

III. Download

Currently VxDL is the default application for Software Downloads for most of the latest products. Moving forward to VxEOS, NCP becomes the default application for Software Downloads from System Mode.

This menu option will be available to download applications into the device. The following information is required:

- Target GID
- Selecting “Full” vs. “Partial” download
- Selecting “TCP” vs. “SSL”
- How to reach VeriCentre Server
  - IP address
  - PORT number
- Application ID
- Terminal ID

Summary screen will summarize all information for confirmation or editing before starting the download.

Once user confirms data entered, similar to IP downloads from System Mode download starts.

NOTE

TBD: Initial NCP development will mimic VxDL features for secured (SSL) downloads. Currently VxDL does not provide SSL Server/Client Authentication. Further investigation and design required on this area.

Menu: Terminal Information

Selecting this menu option automatically presents several screens with current information from the device, drivers and connections.

The following sections will be presented in consecutive pages. User will have control when to scroll pages. As described on Printer ownership section, if PRINTER device is available, on any page user will also have the option to print all pages.

I. IP addresses status

This menu option will be listed separate from the menu option Communication Technology, but in reality IP address information is directly related to the Network Interface. This menu is available to allow user direct access to the IP address information.

NCP will detect all Communication Technologies available on the device and will list all names for user to select the Network Interface (NWIF from CommEngine).

- DHCP Enable vs. Static IP
- If Static IP
  - IP address
  - Subnet Mask
  - Gateway IP address
  - Primary DNS Server IP address
  - Secondary DNS Server IP address
- If applicable
  - DHCP Lease Start Time
  - DHCP Lease End Time

II. Communication Technology

Based on the Communication Technologies available on the device, the following screens will be generated

i. Ethernet Status

- Device Name
- Device Driver name
- Connection Status
- MAC address

ii. Wi-Fi Status

- Device Name
- Device Handler
- Network name (SSID)
- Encryption (None, WEP64, WEP128, WPA-PSK)
- Key Index
- Connection Status
- Signal Quality
- Link Quality
- Model
- Firmware version
- MAC address
- AP MAC address (if available)

iii. CDMA Status

- Device Name
- Device Handler
- Phone Number
- Username
- Password
- Connection Status
- RSSI
- Signal Quality
- Model
- Firmware version
- PRL (Preferred Roaming List) version
- ESN (Electronic Security Number)
- SID (System Identification Number)
- MDN
- MIN

iv. GPRS Status

- Device Name
- Device Handler
- APN
- Phone Number
- Username
- Password
- Connection Status
- RSSI
- Signal Quality
- Model
- Firmware version
- ICC ID
- IMSI
- IMEI

v. GSM with PPP

- Device Name
- Device Handler
- Phone Number
- Username
- Password
- Connection Status

vi. Land-Line with PPP

- Device Name
- Device Handler
- Phone Number
- Username
- Password
- Connection Status

NOTE

No Network Interface for “Land-Line” or “GSM” (without PPP) are supported by VxEOS

III. Versions

- VxNCP version
- CommEngine version
- ceAPI version
- Device Driver Software (DDI) version

Menu: Setup

NCP will detect all Communication Technologies available on the device and will list all names for user to select the Network Interface (NWIF from CommEngine).

I. Primary Network Interface

Generally, NCP’s menu options detect all Network Interfaces from CommEngine via ceAPI. Under specific circumstances, defining a ‘Default Network Interface’ simplifies and improves the response time. This setting applies for the following options:

- Idle Screen
- Diagnostics: Ping Gateway
- Diagnostics: Ping DNS Server

II. Communication Technology

Changes to any Network Interface require manually restarting the corresponding interface via menu option Network Maintenance.

Based on the technologies available, following settings will be configurable via NCP.

i. Ethernet

- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address
    - Primary DNS Server IP address
    - Secondary DNS Server IP address

ii. Wi-Fi

- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address
    - Primary DNS Server IP address
    - Secondary DNS Server IP address
- Network name (SSID)
- Encryption (None, WEP64, WEP128, WPA-PSK, WPA2)
- Key Index
- Key Value

NOTE

For security reasons, current “Key Value” never will be displayed in clear. Masked string will represent the value is present. If customer selects to modify it, then new value being entered will be edited in clear. Once input completes (pressing enter) value will be masked on the display again.

iii. CDMA

- Username
- Password
- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address
    - Primary DNS Server IP address
    - Secondary DNS Server IP address

iv. GPRS

- APN
- Phone Number
- Username
- Password
- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address
    - Primary DNS Server IP address
    - Secondary DNS Server IP address

v. GSM with PPP

- Phone Number
- Username
- Password
- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address

- Primary DNS Server IP address
- Secondary DNS Server IP address

vi. Land-Line with PPP

- Phone Number
- Username
- Password
- IP Parameters
  - DHCP Enable vs. Static IP
  - If Static IP
    - IP address
    - Subnet Mask
    - Gateway IP address
    - Primary DNS Server IP address
    - Secondary DNS Server IP address

NOTE

No Network Interface for “Land-Line” or “GSM” (without PPP) are supported by VxEOS

TBD: More parameters will be added during VxEOS development. NCP dynamically will retrieve those new parameters and will be listed subsequently to the items listed above.

III. Diagnostics

i. IP address / URL for Ping

Default IP address or URL to use from Diagnostics menu, PING option.

ii. IP:PORT for TCP Socket connect

Default IP address or URL to use from Diagnostics menu, TCP option.

iii. IP:PORT for SSL Socket connect

Default IP address or URL to use from Diagnostics menu, SSL option.

V. CommEngine

NCP as default UI for all VxEOS components serves as UI Engine to dynamically modify CommEngine’s settings.

This menu option is dynamically generated based on the configuration files `VXCEC.CFG` and `VXCEM.MTD`, as described on VDN 28809 listed on References section.

VI. Device Driver (s)

NCP as default UI for all VxEOS components serves as UI Engine to dynamically modify settings for all Device Drivers available on the device.

NCP will query CommEngine via ceAPI for all Network Interfaces and the corresponding Device Driver loaded. All Device Driver names will be listed for user to select which one to setup. Once user selects a specific Device Driver, NCP will use the Device Driver Name returned by ceAPI to load the corresponding ".CFG" and ".MTD" configuration files.

Following steps demonstrate how NCP will dynamically build the UI based on the specific Device Driver configurable parameters.

- Device Driver name retrieved via ceAPI
- Configurable data is defined on the specific ".MTD" file
- Current values will be dynamically queried via ceAPI
- Based on the ".MTD" setup and values via ceAPI, user will be able to change setup
- Updated values will be saved on the specific ".CFG" file

Configuration files ".CFG" and ".MTD" will follow the format described on the document "Configuration Management" listed on References section.

Menu: Exit

This menu option will automatically return the CONSOLE device to the application that activated NCP. Full details covered in the section "Error! Reference source not found.".

Pressing CANCEL (Red X) key from IDLE SCREEN will have the same effect as selecting EXIT.

Configuration Files

NCP default settings for "Diagnostic" operations will be pre-defined via configuration files.

Location	FIlename	Description
N:46	VXNCP.CFG	Default configuration file. This file contains the name and value of configurable NCP parameters.
I:1	VXNCP.CFG	User's NCP configuration file. This file contains NCP's settings defined during deployment. Values on this file will overwrite those configured on N:46/VXNCP.CFG. Changes via NCP's UI will be reflected on this file.
N:46	VXNCP.MTD	This file contains the name and editable information to configure NCP parameters. This file is primarily for NCP processing. No run-time changes should happen to this file.

Packaging – Filenames & locations

NCP will be part of the standard VxEOS installation package.

VxEOS Filename	VxEOS Location	Download Filename	Download Location	File Type
VxNCP.out	N:46	VxNCP.out{!	F:1	Program
VxNCP.p7s	I:46	VxNCP.p7s	I:1	Signature
VxNCP.cfg	N:46	VxNCP.cfg{!	F:1	Data
VxNCP.cfg.p7s	I:46	VxNCP.cfg.p7s	I:1	Signature
VxNCP.mtd	N:46	VxNCP.mtd{!	F:1	Data
VxNCP.mtd.p7s	I:46	VxNCP.mtd.p7s	I:1	Signature
NCPDSPfont.fon	N:46	NCPDSPfont.fon{!	F:1	Display Font
NCPDSPfont.p7s	I:46	NCPDSPfont.p7s	I:1	Signature
NCPDSPfont.fon	N:46	NCPDSPfont.fon{!	F:1	Display Font
NCPDSPfont.p7s	I:46	NCPDSPfont.p7s	I:1	Signature
NCPPRNfont.fon	N:46	NCPPRNfont.fon{!	F:1	Printer Font
NCPPRNfont.p7s	I:46	NCPPRNfont.p7s	I:1	Signature

NOTE

NCPDSPfont.fon and NCPPRNfont.pft may not be implemented on the final Release version. Only noted here as placeholders.

External Parameters via CONFIG.sys

Setting External Parameters

This section summarizes all `CONFIG.sys` parameters listed throughout the document, including assumptions if not present. Note all `CONFIG.sys` variables must be provided on GID1.

Configuration via CONFIG.sys

CONFIG.sys	Valid values	Description
*VXNCP	0	If 0 (zero) NCP will abort its execution, otherwise runs normally
*VXNCPPROMPTS	[GID][Drive]filename	Custom prompts file
*VXNCPDSPFONT	[GID][Drive]filename	Custom font file for DISPLAY
*VXNCPPRNFONT	[GID][Drive]filename	Custom font file for PRINTER and TableID to load it on the printer memory
*VXNCPPRNTABLE	[1 - 64]	
*VXNCPDATESEP	/ (slash) - (dash) . (dot)	Changes default separator when displaying or printing date information. Separator character used between day month and year values.
*VXNCPDATEFORMAT	MMDDYYYY DDMMYYYY MMMDDYYYY DDMMMYYYY	Changes default format when displaying or printing date information. Specifies whether to use month followed by day or day followed by month. Also to represent month by its numeric value or string abbreviation.
*VXNCPMONTHNAME	12 strings, 3 characters, long, comma separated	Abbreviated string names for ‘month’, setting only valid with formats MMMDDYYY or DDMMMYYYY.
*VXNCPTIMEFORMAT	“12”	Changes default time format. 1pm will be represented as 01:00 PM

	“24”	Changes default time format. 1pm will be represented as 13:00
*ZSEC	“1”	TCP/IP (non secured) download
	“2”	SSL (secured) download

VeriFone, Inc.
2099 Gateway Place, Suite 600

San Jose, CA, 95110 USA.

1-800-VeriFone

www.verifone.com

Verix EOS Volume II

Communications Manual

VeriFone Part Number DOC00302, Revision A.4