Verix eVo Volume II: Operating System and Communication

Programmers Manual

VeriFone Part Number DOC00302, Revision A

VeriFone, Inc.
2099 Gateway Place, Suite 600

San Jose, CA, 95110 USA.

1-800-VeriFone

www.verifone.com

VeriFone Part Number DOC00302, Revision A

CONTENTS

PREFACE 7

Organization 7
Target Audience 8
    Assumptions About the Reader 8
Conventions and Acronyms 8
    Acronyms 9
Related Documentation 13

CHAPTER 1
TCIP/IP Stack

Interface 15
Getting Started 15
    Simple Socket Application 15
Socket Functions 16
socket() 17
bind() 19
listen() 20
accept() 21
connect() 23
getpeername() 25
getsockname() 26
setsockopt() 27
getsockopt() 36
recv() 44
send() 46
recvfrom() 49
sendto() 51
shutdown() 53
socketclose() 54
socketerrno() 55
select() 56
socketset_owner() 58
socketioctl() 59
DnsGetHostByName() 60
gethostbyname() 61
blockingIO() 62
inet_addr() 63
DNS Resolver API 64
DnsSetServer() 65
DnsSetUserOption() 66
Ping API 67
PingOpenStart() 68
PingGetStats() 69
PingCloseStop() 70
Packet Capture 71

CHAPTER 2
Secure Sockets Layer (SSL)

OpenSSL API 73
- Include files 73
- Libraries 74
- Crypto functions 74
- SSL functions 74

Verix eVo SSL versus VeriFone SSL library 74

CHAPTER 3
Network Interface

GID1 CONFIG.SYS Configuration 75

CHAPTER 4
Configuration Management

Configuration File Format 78
Internal and External Configuration Files 78
- Verix eVo Configuration Files 78
- Delta Configuration Files 78
- Naming Convention and Location 79

Network Interface Configuration 80

CHAPTER 5
CommEngine Interface API (CEIF.lib)

Message Exchange mxAPI & Message Formatting mfAPI 101
ceAPI Concepts 102
- Device & Device Ownership 102
- Device Driver 102
- Network Interface (NWIF) 103
- Network Connection Process & States 104
- NWIF Events 106
- Multiple NWIFs 107

The ceAPI – Summary 108
- Registration 108
- Device Management 108
- Device Driver (DDI) Configuration 108
- Sending Commands to Device 109
- Communication Infrastructure Configuration, Management & Control 109
- Dial Interface Support 109
- Event Management 109
- Miscellany 110

ceRegister() 111
ceUnregister() 112
ceSetCommDevMgr() 113
ceRequestDevice() 114
ceReleaseDevice() 115
ceGetDDParamValue() 116
ceSetDDParamValue() 118
ceExCommand() 120
ceEnableEventNotification() 122
ceDisableEventNotification() 123
ceGetEventCount() 124
ceGetEvent() 125
ceSetSignalNotification() 127
ceIsEventNotificationEnabled() 129
ceGetNWIFCount() 130

ceGetNWIFInfo() 131
ceStartNWIF() 133
ceStopNWIF(). 135
ceSetNWIFStartMode() 137
ceGetNWIFStartMode() 138
ceSetNWParamValue() 139
ceGetNWParamValue() 141
ceStartDialIF() 142
ceStopDialIF() 144
ceActivateNCP(). 145
ceGetVersion() 146
Constants, Defines & Miscellany 147
List of CommEngine Events 147
Constants 149
List of Network Parameters 151
List of Error Codes 153
Application Developer Notes 157
Handling CommEngine Events. 157
CE_EVT_DDI_APPL for a GPRS / GSM driver 157
Sample code on ceSetNWParamValue() / ceGetNWParamValue(). 157
Sample code on ceSetDDParamValue() / ceGetDDParamValue() 157
Sample code on ceEXCommand() 158
Message Exchange mxAPI & Introduction 158
Payload Size. 158
Message Exchange mxAPI – Summary 158
mxCreatePipe() 159
mxClosePipe() 160
mxGetPipeHandle() 161
mxSend() 162
mxPending(). 163
mxRecv() 164
mxRecvEx() 165
mxAPI Error Codes 166
Message Formatting mfAPI 167
Tag, Length & Value (TLV) 167
TLV List. 167
Message Formatting mfAPI – Summary. 168
mfCreateHandle(). 170
mfDestroyHandle() 171
mfAddInit() 172
mfAddClose() 173
mfAddTLV() 174
mfFetchInit() 175
mfFetchClose(). 176
mfFetchReset(). 177
mfPeekNextTLV(). 178
mfFetchNextTLV() 179
mfFindTLV() 180
mfEstimate(). 181
mfAPI Error Codes 182

CHAPTER 6
Verix eVo Communication Engine Application
CommEngine Bootstrap Process 183
CommEngine Invocation 183
Conditionally Starting CommEngine 183
CommEngine Startup Operations 184
CommEngine Configuration and Metadata File 184
Network Configuration via Configuration Delta File 184
Network Configuration via GID1 CONFIG.SYS parameters 187
Configuration Precedence 187
Configuration Options 188
CommEngine Logging 188
Interface with Device Drivers 188
Interface with Treck TCPIP Library. 188
Application Interface 188
Application Events 189

CHAPTER 7
Network Control Panel (NCP)
Startup Operation 194
Starting NCP executable. 194
Interoperability 194
IP downloads from System Mode 197
Running under Single-Application Mode. 198
Running under VMAC Environment 198
Configuration Files 199
User Interface 202
Customization 202
Idle Screen 204

PREFACE

This communications manual supports the Development Toolkit (DTK) for the Verix eVo Volume II: Operating System and Communications 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 eVo terminal.

Verix eVo 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 eVo operating system. This manual is designed to help programmers develop those applications.

This manual also contains explicit information regarding the Application Programming Interfaces (APIs) within the Verix eVo 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 the network interface API.

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

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

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

Chapter 7, Network Control Panel (NCP) - discusses the functionilties of the Network Control Panel (NCP) in the Verix eVo system.

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 eVo-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 Verix eVo DTK on this machine.
- Has access to Verix eVo Volume II: Operating System and Communications 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
dBm	decibel meter
h	hexadecimal

Table 1 Conventions

Abbreviation	Definition
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
CDMA	Code Division Multiple Access
CE	Communication Engine

Table 2 Acronyms (continued)

Acronym	Definition
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
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
IPP	Internal Pin Pad
ISR	Interrupt Service Routine

Table 2 Acronyms (continued)

Acronym	Definition
LAN	Local Area Network
LCD	Liquid Crystal Display
LCP	Link Control Protocol
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
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
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
RFID	Radio Frequency Identification
RFU	Reserved for Future Use
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

Table 2 Acronyms (continued)

Acronym	Definition
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
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 eVo Volume II: Operating System and Communications Solutions, refer to the following set of documents:

- Verix eVo Multi-App Conductor Programmers Guide, VPN - DOC00306.
- Verix eVo Communications Server Instantiation Guide, VPN - DOC00308.
- Verix eVo Development Suite Getting Started Guide, VPN - DOC00309.
- Verix eVo Volume I: Operating System Programmers Manual, VPN - DOC00301.
- Verix eVo Volume III: Operating System Programming Tools Reference Manual, VPN - DOC00304.
- Verix eVo Porting Guide, VPN - DOC00305.
- Verix V Operating System Programmers Manual, VDN 23230.
- CommEngine DDI Integration Guide.
- Network Security with OpenSSL
- Cryptography for Secure Communications

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.

CHAPTER 1

TCIP/IP Stack

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

Interface

The first two items will be part of the Verix eVo 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 included with Verix eVo.

Getting Started

Tthe network device must first be opened then handed over to the stack. Once this has is, 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.

Simple Socket Application

Simple TCP application sample code.

Simple UDP application 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 out-of- 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 active 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 occurred, 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 standard 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 receive 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`. Use `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`.

errno	Description
`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.

Parameter	Description
`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 optionV`aluePtr`. 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_MAXRT`above. 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()). `sockfd90` 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 returnes 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 `fromPtr` 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()

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()
- DnsSetUserOption()
- DnsSetUserOption()

NOTE These functions will affect the whole system, not only the caller application. Please use extreme caution with using these functions.

DnsSetServer()

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

DnsSetUserOption()

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()

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`

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.

The lines between `<PCAP_B>` and `<PCAP_E>` must contain hex characters (0-9,A-F) which represent one PCAP record. This 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)

Verix eVo 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 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

Verix eVo SSL versus VeriFone SSL library

The Verix eVo 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.

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.

Configuration Management

New SW components are being introduced as part of Verix eVo 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.

This section describes the standardized configuration file format to use across all Verix eVo components requiring external configuration/customization.

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

Verix eVo components will be released with default settings. Customers will have the ability to overwrite some / all parameters during deployment via a delta file.

Configuration File Format

The configuration file can 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 CE Delta 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 ; (0x3B) 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.

View Sample Ethernet Driver Delta File

Internal and External Configuration Files

While delta files can be updated by just downloading a new delta file to GID 1. It is recommended that eVo users update Device Driver information via NCP or using ceAPIs `ceSetDDParamValue()` or `ceGetDDParamValue()`.

Verix eVo Configuration Files

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

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

Factory Default Settings

Every Verix eVo 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 eVo settings for specific components (i.e. device timeouts).

Following sample described on Sample CE Data File, following configuration file allows an Ethernet device to connect to Ethernet using static configuration with IP address 10.64.80.175. View Sample CE Factory Default Settings.

Metadata File

This file will be used by Verix eVo components to manipulate dynamic settings. Metadata files reside in GID 46 and are designed and generated by the Verix eVo Volume II team. This file states the limits on the parameters users want to set. View Sample Metadata.

Delta Configuration Files

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

Naming Convention and Location

Location	Filename	Description
N:46	<basename>.INI	Default configuration file. This file contains the name and value of configurable parameters.
I:1	<basename>.INI	User's configuration file. This file contains custom settings defined during deployment. Values on this file will overwrite those configured on N:46/<basename>.INI. 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 eVo processing. No run-time changes should happen to this file.

Network Interface Configuration

When starting NWIF users may want to set or get IP and PPP configurations, eVo users may use NCP to update these settings. eVo users may also use ceAPI ceSetNWParamValue() and ceGetNWParamValue().

eVo application developers may set or get the Device Driver parameters depending on the Driver that is available per terminal. eVo users may also use NCP to update these settings. eVo users may also use ceAPI ceSetDDParamValue() and ceGetDDParamValue().

eVo users can also use the macros defined in `vos_ddi_ini.h` and `vos_ioctl.h`.

Table 3 IOCTL Calls Available for all Device Drivers

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_GET_DRIVER_BUILD_DATE	None	NA	String **Ex:** Jan 4 201014:02:01	Response is the build date of the driver
IOCTL_GET_EXTENDED_ERROR	None	NA	String **Ex:**: Ethernet Port Open Failed	Response is the extended error encounter
IOCTL_GET_DRIVER_STATES	None	NA	String **Ex:** 10 31 21 41	Response is the network, power, communication, and connection states of the device

Ethernet Device Driver

Table 4 IOCTL Calls Available for Ethernet

Param Name (Type:string)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_GET_MAC	None	NA	String **Ex:** 00-0B-4F-30-31-C4	Response is the MAC address associated to the device
IOCTL_GET_LINK_SPEED	None	NA	String **Ex:**: 100 Mb/full	Response is the link speed

Table 5 Ethernet DDI DLM Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
DRIVER_VER	Driver Version number	NA	R	string	NA	Provides the DDI DLM version number
SUPP_AT	0	NA	R	boolean	NA	AT commands are supported by DDI DLM
SUPP_RSSI	0	NA	R	boolean	NA	RSSI values will be provided by DDI DLM
CONN_RET	1	NA	R	number	NA	Connection retries
CONNRET_TM	1000msecs	NA	R	number	NA	The time interval after which DDI DLM retries to connect to network again on failure.
CONNCHK_TM	0	NA	R	number	NA	The time interval at which DDI DLM runs network monitoring/management activity.
NO_NET_RET	9999	NA	R	number	NA	Number of retries for network to be up, before reporting error **Note:** When 9999 is the value of NO_NET_RET it indicates that retry is infinite
NET_RET_TM	0	NA	R	number	NA	Time interval after which network availability to be checked
ET_MULCAST	0	0-1	RW	number	ETI_MULCAST	If set to 1 allows reception of broadcast packets. **Note:** If ET_MULCAST is set to 1 in a 520 terminal
ETH_PORT	“/dev/eth1”	NA	R	string	NA	Comm Port to use.

Conexant Device Driver

Table 6 IOCTL Calls Available for Dial

Param Name (Type:string)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_GET_CONNECTION_BAUD	None	NA	String	Get the connection baud rate

Table 7 Dial DDI DLM Configuration Parameters

INI Parameter Name	Default Values	Attribute R= read only RW= read/ write	Data Type	Value (Number)/ Length (String)		Config.sys	Description
				Min	Max		
DRIVER_VER	Driver Version Number	R	string	1	21	NA	Version Number of the Dial DDI DLM
SUPP_AT	1	R	boolean	NA	NA	NA	AT Command Supported
SUPP_RSSI	0	R	boolean	NA	NA	NA	RSSI Check not supported
CONN_RET	1	RW	number	1	10	CONN_RET	Maximum connection retries
CONNRET_TM	1000	RW	number	500	60 000	CONNRET_TM	Connection retry timeout in milliseconds
NET_RET_T	1000	RW	number	500	60 000	NET_RET_T	Network retry timeout in milliseconds
NO_NET_RET	2	RW	number	1	10	NO_NET_RET	Maximum retries during no network coverage
PR_NET_RET	2	RW	number	1	10	PR_NET_RET	Maximum retries during poor network coverage
DL_START	Conexant: AT&FE0Q0V0 Silabs: ATFE0Q0V0	RW	string	2	50	DL_START	Start up string
DL_HANGUP	ATH0	RW	string	2	50	DL_HANGUP	Hang up string

Table 7 Dial DDI DLM Configuration Parameters

INI Parameter Name	Default Values	Attribute R= read only RW= read/ write	Data Type	Value (Number)/ Length (String)		Config.sys	Description
				Min	Max		
DL_DIAL	ATDTW	RW	string	2	50	DL_DIAL	AT command string to dial
DL_INITRET	10	RW	number	1	10	DL_INITRET	Number of times in sending the start up string until an "OK" is returned
DL_ATDELAY	250	RW	number	250	100 0	DL_ATDELAY	Number of millisenconds to wait between sending AT commands
DL_INIT_1	ATS0=2	RW	string	2	50	DL_INIT_1	Init String
DL_INIT_2	ATS0=2	RW	string	2	50	DL_INIT_1	Init String **Note:** Applicable only for Silabs DDI Driver
DL_BAUD	9	RW	number	0	12	DL_BAUD	baudrate
DL_FORMAT	4	RW	number	0	4	DL_FORMAT	format
DL_ATSNDTO	20000	RW	number	500	60 000	DL_ATSNDTO	Timeout to send the AT command
DL_HW_FCTL	true	RW	boolean	NA	NA	DL_HW_FCTL	Hardware flow control
DL_INIT_TO	20000	RW	number	500	60 000	DL_INIT_TO	Response timeout used for all init AT commands
DL_CONN_TO	120000	RW	number	500	60 000	DL_CONN_TO	Data connection timeout in milliseconds
DL_DISC_TO	1400	RW	number	500	60 000	DL_DISC_TO	Data disconnect timeout in milliseconds
DL_ICHAR_TO	500	RW	number	500	60 000	DL_ICHAR_TO	Intecharacter timeout for all AT commands
DL_PHO_PRI	0	RW	string	1	15	DL_PHO_PRI	Primary phone number to dial
DL_PHO_SEC	0	RW	string	1	15	DL_PHO_SEC	Secondary phone number to dial
DL_SDLC_ON	false	RW	boolean	NA	NA	DL_SDLC_ON	Enable/Disable SDLC protocol

Table 7 Dial DDI DLM Configuration Parameters

INI Parameter Name	Default Values	Attribute R= read only RW= read/ write	Data Type	Value (Number)/ Length (String)		Config.sys	Description
				Min	Max		
DL_SDLCINT	ATW2X4S25=1&D2%C0\N0+A8E=,,,1	RW	string	2	50	DL_SDLCINT	SDLC Init String
DL_SDLC_MOD	AT+MS=V22,0,300,1200,300,1200	RW	string	2	50	DL_SDLC_MOD	AT Command string to set the modulation
DL_FAST_CO	true	RW	boolean	NA	NA	DL_FAST_CO	Enable/Disable Fast connect

GPRS Device Driver

Table 8 IOCTL Calls Available for GPRS

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOTCL_GET_RSSI	None	NA	String **Ex:** 100 (means a signal strength of approximately -51 dBm)	Response is a percentage equivalent of the RSSI value retrieved from the modem.
IOTCL_GET_RSSI_ABS	None	NA	String **Ex:** 30 (means a signal strength of approximately -53 dBm)	Response is a range indicator of the RSSI signal, measured in dBm. Please see AT Commands document for Cinterion MC55i.
IOTCL_GET_RSSI_DBM	None	NA	String **Ex:** -59 (means a signal strength percentage of 85%)	Response is a dBm approximate of the RSSI value retrieved from the modem.
IOTCL_GET_FW_VERSION	None	NA	String **Ex:** 01.03	Response is the firmware version of the Cinterion MC55i modem.
IOTCL_GET_BER	None	NA	String **Ex:** 99 (means an unknown bit error rate)	Response is the bit error rate associated to the RSSI.
IOTCL_GET_LINK STATUS	None	NA	String **Ex:** 1 (means the data link is up)	Response can be either: • 1 (data link is up) • 0 (data link is not up)
IOTCL_GET_IMEI	None	NA	String	Response is the International Mobile Equipment Identity of the modem
IOTCL_GET_ICCID	None	NA	String	Response is the SIM card identification number

Table 8 IOCTL Calls Available for GPRS

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOTCL_GET_IMSI	None	NA	String	Response is the International Mobile Subscriber Identity
IOTCL_SET_SERVICE_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets network registration notification if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOTCL_SET_BER_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets signal quality notification (in terms of bit error rate) if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOTCL_SET_ROAM_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets roaming status notification if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications

Table 8 IOCTL Calls Available for GPRS

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOTCL_SET_SMS_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets unread short message notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOTCL_SET_CALL_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets call status notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOTCL_SET_SMS_FULL_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets `full SMS memory` notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications

Table 9 GPRS DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_DRIVER_VERSION	Driver Version number	NA	R	string	NA	Provides the DDI DLL version number
INI_SUPPORTS_AT	1	NA	R	boolean	NA	If AT commands are supported by DDI DLL: • 1 (supported) • 0 (not supported)
INI_SUPPORTS_RSSI	1	NA	R	boolean	NA	If RSSI values will be provided by DDI DLL • 1 (supported) • 0 (not supported)
INI_MAX_CONNECTION_RETRIES	5	2-9999	RW	Integer	GS_CONN_RET	Maximum attempts to connect to the network during `DDI_MANAGER::ddi_open()`
INI_POOR_NETWORK_RETRIES	5	2-9999	RW	Integer	GS_PR_NET_RET	Number of retries for signal strength to be above threshold, before reporting error
INI_NO_NETWORK_RETRIES	5	2-9999	RW	Integer	GS_NO_NET_RET	Number of retries for network to be up, before reporting error
INI_NETWORK_RETRY_TIMEOUT	1000	500-120000	RW	Integer	GS_NET_RET_TM	Time interval after which network availability to be checked (in milliseconds)
INI_CONNECTION_RETRY_TIMEOUT	1000	500-120000	RW	Integer	GS_CONNRET_TM	The time interval after which DDI DLL retries to connect to network again on failure (in milliseconds)
INI_GPRS_APN	None	1-64	RW	string	GS_CONNCHK_TM	GPRS access point name
INI_GPRS_PRIMARY	None	4-128	RW	string	GS_PRIMARY	GPRS dial number
INI_GPRS_MONITOR	0	NA	RW	boolean	GS_SIM_PIN	If the driver needs to check GPRS availability before connection. • 0 (do not monitor GPRS) • 1 (monitor GPRS)

Table 9 GPRS DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GPRS_SIM_PIN	None	0-5	RW	Integer	GS_SIM_PUK	GPRS SIM PIN
INI_GPRS_SIM_PUK	None	0-8	RW	Integer	GS_MINRSSI	GPRS SIM PUK
INI_GPRS_MIN_RSSI	10	0-99	RW	Integer	GS_SEND_TO	Minimum allowable RSSI percentage for the driver to connect
INI_GPRS_AT_SEND_TIMEOUT	20000	500-120000	RW	Integer	GP_SEND_TO	Timeout in milliseconds when sending an AT command to the modem
INI_GPRS_AT_RESPONSE_TIMEOUT	20000	500-120000	RW	Integer	GP_RESP_TO	Timeout in milliseconds when receiving a response from an AT command is sent to the modem
INI_GPRS_CONNECT_TIMEOUT	40000	500-120000	RW	Integer	GP_CONN_TO	Data link connection timeout in milliseconds
INI_GPRS_DISCONNECT_TIMEOUT	20000	500-120000	RW	Integer	GP_DISC_TO	Data link disconnection timeout in milliseconds
INI_GPRS_INTERCHAR_TIMEOUT	2000	500-120000	RW	Integer	GP_CHAR_TO	Intercharacter timeout in milliseconds
INI_GPRS_AT_DELAY	2000	250-2000	RW	Integer	GP_ATDELAY	Delay in between retries in sending initialization strings to the modem (milliseconds)
INI_GPRS_ATTACH_TIMEOUT	300000	500-300000	RW	Integer	GP_ATCH_TO	GPRS network attach timeout in milliseconds
INI_GPRS_INIT_RETRIES	3	2-9999	RW	Integer	GP_INIT_RT	Retries in sending the initialization strings in the modem
INI_GPRS_POWER_UP_RETRIES	3	2-9999	RW	Integer	GP_POWR_RT	Retries in powering up the GPRS modem
INI_GPRS_LOW_BATTERY_LIMIT	10	10-99	RW	Integer	GP_LOWBATT	Minimum percentage of remaining battery charge
INI_GPRS_DETACH	false	NA	RW	Boolean	GP_DETACH	Set to true if the driver needs to detach before attaching to the network

Table 9 GPRS DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GPRS_SERVICE_NOTIFICATION	false	NA	RW	Boolean	GP_SERV_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.
INI_GPRS_BER_NOTIFICATION	false	NA	RW	Boolean	GP_BER_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.
INI_GPRS_ROAM_NOTIFICATION	false	NA	RW	Boolean	GP_ROAMSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.
INI_GPRS_SMS_NOTIFICATION	false	NA	RW	Boolean	GP_SMS_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.

Table 9 GPRS DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GPRS_CALL_NOTIFICATION	false	NA	RW	Boolean	GP_CALLSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.
INI_GPRS_SMS_FULL_NOTIFICATION	false	NA	RW	Boolean	GP_SMSFSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.

GSM Device Driver

Table 10 IOCTL Calls Available for GSM

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_GET_RSSI	None	NA	String **Ex:** 100 (means a signal strength of approximately -51 dBm)	Response is a percentage equivalent of the RSSI value retrieved from the modem.
IOCTL_GET_RSSI_ABS	None	NA	String **Ex:** 30 (means a signal strength of approximately -53 dBm)	Response is a range indicator of the RSSI signal, measured in dBm. Please see AT Commands document for Cinterion MC55i.
IOCTL_GET_RSSI_DBM	None	NA	String **Ex:** -59 (means a signal strength percentage of 85%)	Response is a dBm approximate of the RSSI value retrieved from the modem.
IOCTL_GET_FW_VERSION	None	NA	String **Ex:** 1.03	Response is the firmware version of the Cinterion MC55i modem.
IOCTL_GET_CONNECTION_BAUD	None	NA	String **Ex:** 9600 (means the connection is running on 9600 bits per second)	Response is the connection baud rate established on the GSM network
IOCTL_GET_BER	None	NA	String **Ex:** 99 (means an unknown bit error rate)	Response is the bit error rate associated to the RSSI

Table 10 IOCTL Calls Available for GSM

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_GET_LINK STATUS	None	NA	String **Ex:** 1 (means the data link is up)	Response can be either: • 1 (data link is up) • 0 (data link is not up)
IOCTL_GET_IMEI	None	NA	String	Response is the International Mobile Equipment Identity of the modem
IOCTL_GET_ICCID	None	NA	String	Response is the SIM card identification number
IOCTL_GET_IMSI	None	NA	String	Response is the International Mobile Subscriber Identity
IOCTL_SET_SERVI CE_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets network registration notification if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOCTL_SET_BER_N OTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets signal quality notification (in terms of bit error rate) if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications

Table 10 IOCTL Calls Available for GSM

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_SET_ROAM_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets roaming status notification if the application wants to have notifications or not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOCTL_SET_SMS_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets unread short message notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications

Table 10 IOCTL Calls Available for GSM

Param Name (Type:macro)	Data Accepted	Input Data Type	Output Parameter Data Type	Description
IOCTL_SET_CALL_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets call status notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications
IOCTL_SET_SMS_FULL_NOTIFICATION	Pointer to a string that contains either `1` or `0`.	String	None	Sets `full SMS memory` notification if the application wants to have notifications of not. Valid input data values are: 1 - turn on notifications 0 - turn off notifications

Table 11 GSM DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_DRIVER_VERSION	Driver Version Number	NA	R	String	NA	Provides the DDI DLL version number
INI_SUPPORTS_AT	1	NA	R	Boolean	NA	If AT commands are supported by DDI DLL: • 1 (supported) • 0 (not supported)
INI_SUPPORTS_RSSI	1	NA	R	Boolean	NA	If RSSI values will be provided by DDI DLL • 1 (supported) • 0 (not supported)

Table 11 GSM DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_MAX_CONNECTION_RETRIES	5	2-9999	RW	Integer	GS_CONN_RET	Maximum attempts to connect to the network during `DDI_MANAGER::ddi_open()`
INI_POOR_NETWORK_RETRIES	5	2-9999	RW	Integer	GS_PR_NET_RET	Number of retries for signal strength to be above threshold, before reporting error
INI_NO_NETWORK_RETRIES	5	2-9999	RW	Integer	GS_NO_NET_RET	Number of retries for network to be up, before reporting error
INI_NETWORK_RETRY_TIMEOUT	1000	500-120000	RW	Integer	GS_NET_RET_TM	Time interval after which network availability to be checked (in milliseconds)
INI_CONNECTION_RETRY_TIMEOUT	1000	500-120000	RW	Integer	GS_CONNRET_TM	The time interval after which DDI DLL retries to connect to network again on failure (in milliseconds)
INI_MAX_CONNECTION_RETRIES	500	500-120000	RW	Integer	GS_CONNCHK_TM	The time interval at which DDI DLL runs network monitoring/ management activity (in milliseconds)
INI_GSM_PRIMARY	None	4-128	RW	String	GS_PRIMARY	GSM dial number
INI_GSM_SIM_PIN	None	0-5	RW	Integer	GS_SIM_PIN	GSM SIM PIN
INI_GSM_SIM_PUK	None	0-5	RW	Integer	GS_SIM_PUK	GSM SIM PUK
INI_GSM_MINRSSI	10	0-99	RW	Integer	GS_MINRSSI	Minimum RSSI
INI_GSM_AT_SEND_TIMEOUT	20000	500-120000	RW	Integer	GS_SEND_TO	Timeout in milliseconds when sending an AT command to the modem

Table 11 GSM DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GSM_AT_RESPONSE_TIMEOUT	20000	500-120000	RW	Integer	GS_RESP_TO	Timeout in milliseconds when receiving a response from an AT command is sent to the modem
INI_GSM_CONNECT_TIMEOUT	40000	500-120000	RW	Integer	GS_CONN_TO	Data link connection timeout in milliseconds
INI_GSM_DISCONNECT_TIMEOUT	20000	500-120000	RW	Integer	GS_DISC_TO	Data link disconnection timeout in milliseconds
INI_GSM_INTERCHAR_TIMEOUT	2000	500-120000	RW	Integer	GS_CHAR_TO	Interchar timeout in milliseconds
INI_GSM_AT_DELAY	2000	250-2000	RW	Integer	GS_ATDELAY	Delay in between retries in sending initialization strings to the modem (milliseconds)
INI_GSM_INIT_RETRIES	3	2-9999	RW	Integer	GS_INIT_RT	Retries in sending the initialization strings in the modem
INI_GSM_POWER_UP_RETRIES	3	2-9999	RW	Integer	GS_POWR_RT	Retries in powering up the GSM modem
INI_GSM_LOWBATTERY_LIMIT	10	10-99	RW	Integer	GS_LOWBATT	Minimum percentage of remaining battery charge
INI_GSM_SERVICE_NOTIFICATION	false	NA	RW	Boolean	GS_SERV_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding “set” IOCTL parameter.

Table 11 GSM DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GSM_BER_NOTIFICATION	false	NA	RW	Boolean	GS_BER_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding “set” IOCTL parameter.
INI_GSM_ROAM_NOTIFICATION	false	NA	RW	Boolean	GS_ROAMSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding “set” IOCTL parameter.
INI_GSM_SMS_NOTIFICATION	false	NA	RW	Boolean	GS_SMS_STS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding “set” IOCTL parameter.

Table 11 GSM DDI DLL Driver Configuration Parameters

INI Parameter Name	Default Values	Min-Max Values	Attribute R= read only RW= read/ write	Data Type	Config.sys	Description
INI_GSM_CALL_NOTIFICATION	false	NA	RW	Boolean	GS_CALLSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.
INI_GSM_SMS_FULL_NOTIFICATION	false	NA	RW	Boolean	GS_SMSFSTS	This should be set to true before the driver is loaded if the application wants to get notified with network registration status. This configuration can be modified at run-time using its corresponding "set" IOCTL parameter.

NOTE The INI and IOCTL parameters are defined in `vos_ddi_ini.h` and `vos_ioctl.h` respectively. Applications using these parameters should include the relevant header file.

CommEngine Interface API (CEIF.lib)

The Verix eVo 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 section.

The ceAPI is a shared library and this library is referred as the Communication Engine Interface library or CEIF.LIB residing in Verix eVo. 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 Verix eVo. 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 eVo, a 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. Once the task is complete, it closes the device and ownership ceases. The device is then 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 name and it is owned by the task that opens it. Device ownership can be transferred by the device that currently owns it.

VMACIF is a required download in the VMAC environment.

Device Driver

In Verix eVo, 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. A device driver supports 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.

A 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., a device driver 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 from each other. 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 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 a terminal from the network by managing the network interface. The application using ceAPI can query and modify the NWIF configuration, and 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 Verix eVo 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 advantages as the link layer need not be re-established and is cost saving as the PPP session is not idle. 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.

NOTE

It is not uncommon for ISPs and Wireless Service Providers to terminate the link layer when the PPP session is terminated. In such situations, the application making the request will receive additional events that it should manage. It is also possible to incrementally bring up the connection. However, the same may not be true for bringing the connection down as this depends on the service provider.

Figure 1 and Figure 2 below depict this connection and disconnection process.

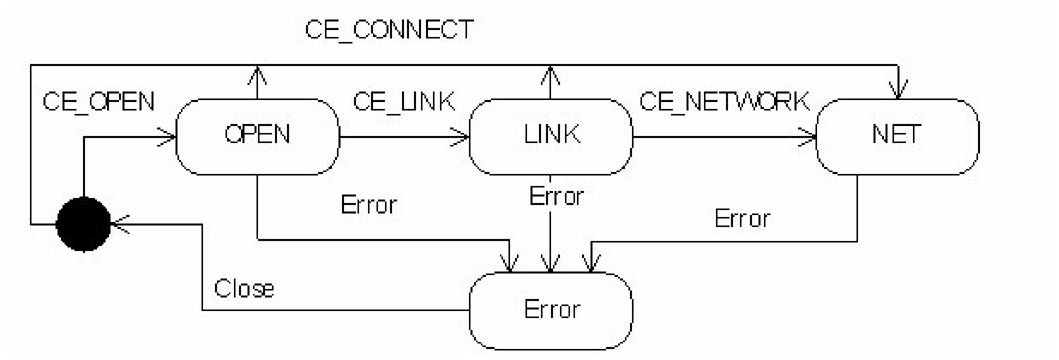


Figure 1 **Connection-up state transition diagram (Left to Right)**

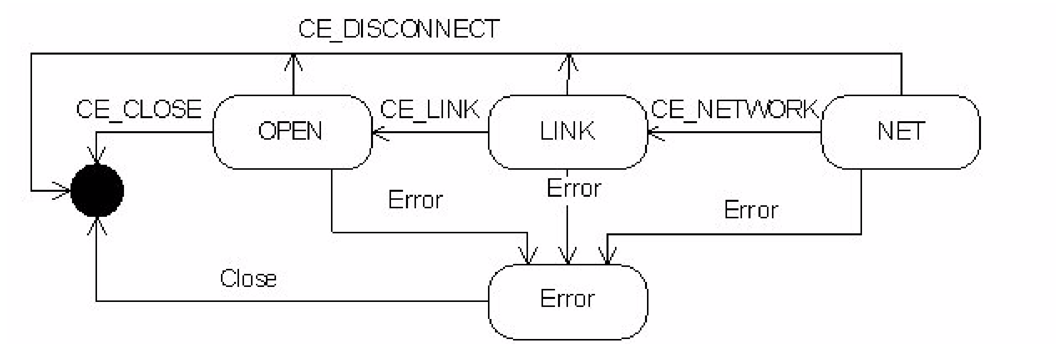


Figure 2 **Connection-down state transition diagram (Right to Left)**

The connection process takes time and applications need to know when the network connection is up so they can start processing transactions. When the connection reaches an end (terminal) state, it broadcasts an event to all registered applications notifying the applications of the state of the connection. This notification is sent out when the network connection is established, or torn down, or reaches an error state. Applications can also query the status of the connection using ceAPI.

The Handling CommEngine Events section expands on the concept of connection management. An understanding of the ceAPI is recommended prior to reviewing topics in the Application Developer Notes section.

Network Connection State Descriptions

Network Connection State	Description
OPEN	In an OPEN state, this means that CommEngine has successfully initialized the device driver. Once the device driver is initialized, any queries to it will now be allowed. In this state, any AT commands and IOCTL requests will now be possible.
LINK	In a LINK state, the device driver is successfully connected to the network. In a PPP connection, a data link has been established. In the case of a GPRS device driver, this state includes attaching to the network and setting and activating the PDP context.
NET	In a NET state, TCPIP stack is now up and running. Socket functionalities can now be used. In a PPP connection, this means that the PPP session has been established. For Ethernet DHCP, the IP address has been obtained.
ERROR	In the event that an error occurs while CommEngine is trying to establish the target state requested, CommEngine will bring the connection down until CommEngine’s initial state. There is no way to recover the previous target state if an error occurs. Applications will have to manually set the previous state.

Once CommEngine has successfully reached the target state requested, it maintains the responsibility of maintain the current target state. For example, an application requests CommEngine to establish a connection until NET state but in the middle of the connection, the network goes down. CommEngine will try to re-establish the connection until the previous target state is reached again or until re-establishing the network is no longer possible.

If re-establishing the connection is no longer possible, CommEngine brings the connection back to its initial state. While CommEngine is trying to re-establish the network, changing the target state is not allowed since CommEngine is busy.

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 a possibility (not a 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 the 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.

Multiple NWIFs

CommEngine supports running multiple NWIFs at the same time, as long as they can work independently of each other. An example of this is running Ethernet and PPP Dial NWIFs on the Vx520 at the same time. However, on the Vx680, both GSM and GPRS NWIFs cannot be run at the same time. Both GSM and GPRS NWIF’s share the same physical device limiting it only to one NWIF running at a time.

When two or more NWIFs are running at the same time, how the TCP/IP stack is configured (more details below) is affected thus applications should take this into consideration.

NWIF’s have implicit hierarchy and when more than one NWIF is running, the TCP/IP stack is configured with gateway and DNS of the highest running NWIF.

On the Vx510/Vx520 the hierarchy is:

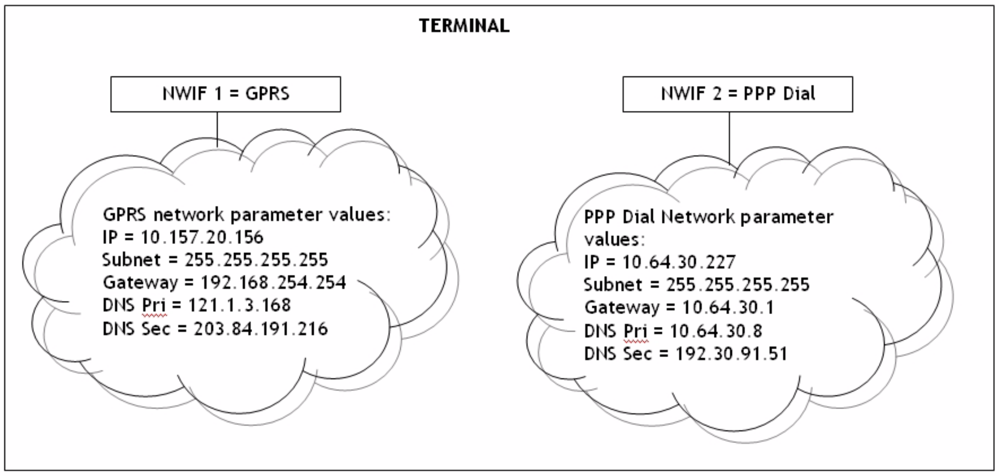
- Ethernet
- PPP/Dial

In this case, Ethernet is higher on the hierarchy. When both are running, Ethernet takes precedence. Consider the situation when both are running and the Ethernet NWIF is stopped. Since PPP/Dial is the only NWIF running, the TCP/IP stack is now configured with its parameters. When Ethernet is restarted, the TCP/IP stack is reconfigured as Ethernet ranks higher in the hierarchy over PPP/Dial.

This set-up will be dependent on the network cloud that each NWIF is using and how the network is set-up. Also, as only one active Default Gateway is allowed, the hierarchy of which gateway per NWIF will be using is automatically done by CommEngine. Primary and secondary DNS that will be used will also be done automatically.

Refer to the `ReadMe.txt` document packaged along with the software for a list of terminals supported and their hierarchy.

Below is a sample scenario on how CommEngine will handle multiple running NWIFs.



If both interfaces are up and the application would want to PING the secondary DNS of the PPP Dial Network interface, this would fail. This is because the packet would pass thru the default gateway of the GPRS network, as GPRS is the primary NWIF. If applications want to make sure that the packet would go thru the default gateway of the PPP Dial network, the GPRS should be brought down. This would force the packet to use the default gateway for the PPP Dial Network.

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.
`ceReleaseDevice()`	Releases the device 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.

Dial Interface Support

API	Description
`ceStartDialIF()`	Starts the dial interface and provides the handle to device.
`ceStopDialIF()`	Stops the dial interface.

Event Management

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.
`ceGetEventCount()`	Returns the number of pending CommEngine Events.
`ceGetEvent()`	Returns first pending CommEngine notification event.
`ceIsEventNotificationEnabled()`	Returns the current state of event notification.

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.
`ECE_REGAGAIN`	Application is attempting to register again to CommEngine even though application is already registered.
`ECE_CREATEPIPE`	CommEngine failed to create a pipe for the application.
`ECE_REGFAILED`	Attempt to register to CommEngine failed as CommEngine is not running.
`< 0`	Error. See List of Error Codes for other return values.

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.

For a multi-threaded application, only one thread needs to register to CE and all the other threads will be able to use any ceAPI.

ceUnregister()

Cancels the application’s prior successful registration with CommEngine.

Prototype

`int ceUnregister(void)`

Return Values

`0`	Successfully unregistered. Prior registration successfully cancelled.
`ECE_NOTREG`	Application is not registered with CommEngine.
`ECE_CANNOT_UNREG`	Application is not allowed to un-register with CommEngine. Possible reasons are: • application was registered as a device management application • application has devices checked-out or has pending requests for devices
`< 0`	Error. See List of Error Codes.

Programming Notes

API `ceRegister()` creates a pipe, registers 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.
`ECE_NOTREG`	Application has not successfully registered with CommEngine.
`ECE_NOCDM`	No device management was required.
`ECE_CDMAPP`	Possible reasons are: • a device management application has already been set • application is attempting to register as the device management application twice
`< 0`	Error. See List of Error Codes.

Programming Notes

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

CEDM must be set to 1 if device management is required.

ceRequestDevice()

Request device from CommEngine.

Prototype `int ceRequestDevice(const char deviceName[])`

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. This should be a null terminated ASCII string.

Return Values

`0`	Successful. Application may open the device.
`ECE_DEVNAMELEN`	Length of device name is above the maximum length for a device name. Maximum length of a device is 31.
`ECE_DEVNAME`	Unknown device name.
`ECE_DEVOWNER`	Device is being used by another application or is unavailable with CE.
`ECE_DEVBUSY`	Device is currently busy as it is being used by CommEngine.
`< 0`	Error. See List of Error Codes.

Programming Notes The calling application is requesting CommEngine for the device `deviceName`. CommEngine transfers the ownership of device to the requesting application. The application is expected to bring down the active network associated with this device before the device is requested.

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 devices. 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 interface may be active at a time.

This API is a blocking call and will return `ECE_SUCCESS` when the requested device is available.

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.
`ECE_DEVNAME`	Unknown device name.
`ECE_NOTDEVOWNER`	Possible reasons are: • application did not check out this device • application is attempting to check in a device that it did not check out
`< 0`	Failed. See List of Error Codes.

Programming Notes

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

If an application is trying to release a device it does not have ownership on, `ECE_NOTDEVOWNER` is returned. `ECE_DEVNAME` is returned if the device name is unknown.

CommEngine will bring up the network interface associated with this device if its start mode is `CE_SM_AUTO`.

ceGetDDParamValue()

Fetch device driver (DD) parameter.

Prototype

```
int ceGetDDParamValue(const unsigned int niHandle, const char paramNameStr
[], void *paramValue, const unsigned int paramValueSize, unsigned int
*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. The type of `paramValue` is dependent on the data type of `paramNameStr`. If it is a numeric type, an integer type of `paramValue` should be passed. In the case of a type Boolean, this will be handled as an integer, this is to be able to support C/C++ applications.
In:	`paramValueSiz e`	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`. In the case of a string type, the NULL terminator is also counted as part of the `paramValueLen`.

Return Values

0	Parameter value obtained.
ECE_PARAM_INVALID	Possible reasons are: • `iHandle` passed is out of range • `paramValueSize` passed is not valid • IOCTL query for parameter failed • `paramValue` is out of range or not within the minimum or maximum value or length allowed • string is not NULL terminated • `paramValue` contains non-valid character/s
ECE_DD_NOT_READY	Parameter associated with `paramNameStr` might be a real time parameter. Such queries will only be possible if NWIF is ready for an IOCTL call. Applications can start NWIF to at least an OPEN state for this query to be successful.

`ECE_PARAM_NOT_FOUND`	Parameter associated with `paramNameStr` can not be found in the Driver's metadata file. This parameter may not be supported by device driver corresponding to `iHandle` passed.
`< 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 running.

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.

A list of Device Driver parameters and their definitions can be seen in the Configuration Management section.

ceSetDDParamValue()

Set device driver (DD) parameter.

Prototype

```
int ceSetDDParamValue(const unsigned int niHandle, const char
paramNameStr[], const void *paramValue, const unsigned int 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`. In the case of a string type, The NULL terminator should be included in the count of `paramValueLen`. In the case of a type Boolean, this will be handled as an integer, this is to be able to support C/C++ applications.

Return Values

`0`	Parameter value set.
`ECE_PARAM_INVALID`	Possible reasons are: • `iHandle` passed is out of range • `paramValueSize` passed is not valid • IOCTL query for parameter failed • `paramValue` is out of range or not within the minimum or maximum value or length allowed • string is not NULL terminated • `paramValue` contains non-valid character/s
`ECE_DD_NOT_READY`	Parameter associated with `paramNameStr` might be a real time parameter. Such queries will only be possible if NWIF is ready for an IOCTL call. Applications can start NWIF to at least an OPEN state for this query to be successful.
`ECE_PARAM_NOT_FOUND`	Parameter associated with `paramNameStr` can not be found in the Driver's metadata file. This parameter may not be supported by device driver corresponding to `iHandle` passed.
`ECE_PARAM_READ_ONLY`	Parameter associated with `paramNameStr` is a read-only parameter.
`< 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. However, some parameters can only be set if a network interface is up.

A list of Device Driver parameters and their definitions can be seen in the Configuration Management section.

ceExCommand()

Execute a command on the device and obtain response string.

Prototype

```
int ceExCommand(const unsigned int niHandle, const void *cmdStr, const
unsigned int cmdStrLen, const unsigned int cmdRespSize, void *cmdResp,
unsigned int *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 until a response is obtained.

Return Values

`0`	Command executed successfully.
`ECE_EX_CMD_FAILED`	IOCTL for specified command failed.
`ECE_NOTREG`	Application must register before attempting this API.
`ECE_TIMEOUT`	Timeout occurred while waiting for response
`ECE_DD_NOT_READY`	Command will only be possible if NWIF is ready for an IOCTL call. Applications can start NWIF to at least an OPEN state for this query to be successful.
`ECE_NOT_SUPPORTED`	Device Driver does not support sending AT commands.
`< 0`	Failed. See List of Error Codes for other return values.

Programming Notes

This API can be successfully executed if network interface associated with `niHandle` is enabled and running. There are certain device driver states where running of this command is not permissible/advisable.

NOTE

This API should only be used as a last resort. Using of this API is strongly discouraged.

ceEnableEventNotification()

Subscribe to notification events from CommEngine.

Prototype

`int ceEnableEventNotification(void);`

Parameters

Return Values

0	Successful.
`ECE_EVENTS_AGAIN`	Application is trying to enable notification events from CommEngine again.
< 0	Failed. See List of Error Codes.

Programming Notes

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

On calling this API, a pipe is created internally for CommEngine to post notification events. The application receives a pipe event whenever CommEngine posts a notification event. The application can fetch the notification event via ceAPI `ceGetEvent()`.

There are two events – the first is the CommEngine notification event which the application registered for and the second, the OS pipe event. Though both are referred as events, the delivery mechanism for both is different. The OS pipe event lets the application know that a CommEngine notification event is available which can be fetched by using API `ceGetEvent()`.

An application may have multiple pipes open and on receiving a pipe event it needs to determine the pipe that caused the OS pipe event to be posted. The application may use ceAPI `ceGetEventCount()` to determine the number of CommEngine notification events pending. If the value returned by API `ceGetEventCount()` is zero, the application may choose to check other pipes.

See API ceGetEvent() for details on event structure.

ceDisableEventNotification()

Unsubscribe to CommEngine notification events. After this call, the application will no longer receive CommEngine notification 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 API `ceEnableEventNotification()` to subscribe to CommEngine notification events.

ceGetEventCount()

Returns the number of pending CommEngine Events.

Prototype

```
int ceGetEventCount(void);
```

Return Values

`>= 0`	Number of pending messages

Programming Notes

This API returns the number of pending CommEngine notification events. If there are none or the application has not enabled notification events, the value returned is zero.

ceGetEvent()

Returns first pending CommEngine notification event.

Prototype

```
int ceGetEvent(stceNWEvt *ceEvt, const int applExtEvtDataSz, void
*applExtEvtData, int *applExtEvtDataLen);
```

Parameters

In:	ceEvt	Pointer to CommEngine notification event structure. See CommEngine Notification Event Structure for details of the structure.
In:	applExtEvtDataSz	Size of buffer pointed by `applEvtData`. Set this zero if no extended application events are expected or desired by the application.
Out:	applExtEvtData	Extended application event data will be loaded in this buffer. Set this parameter to NULL if no application events are expected or desired by the application.
Out:	applExtEvtDataLen	Length of buffer populated with extended application event data. Set this parameter to NULL if no application events are expected or desired.

Return Values

0	Success
ECE_NO_CE_EVENT	No pending CommEngine notification events are present.
< 0	Failed. See List of Error Codes.

Programming Notes

Use API `ceGetEvent()` to fetch CommEngine notification events. CommEngine Notification Event Structure describes structure `stceNWEvt` where the event information is returned. List of CommEngine Events contains the list of all events returned by CommEngine.

CommEngine works with device drivers and some of these drivers generate extended application events (`CE_EVT_DDI_APPL`). The extended application event data is returned in parameter `applExtEvtData`. The size of buffer `applExtEvtData` is specified in parameter `applExtEvtDataSz`. The actual length populated is returned in `applExtEvtDataLen`.

In the case of a `CE_EVT_NET_FAILED` event, CommEngine also sends out extended errors so that applications can determine why a `CE_EVT_NET_FAILED` was sent by CommEngine. The extended error string is returned in parameter `applExtEvtData`. Handling of the extended error string on a `CE_EVT_NET_FAILED` will just be like handling a `CE_EVT_DDI_APPL`.

CommEngine Notification Event Structure

```
typedef struct
{
   unsigned int niHandle;   //  Handle to Network Interface
   unsigned int neEvt;      //  Network Interface event, see List of
                            //  CommEngine Events.
   int neParam1;            //  Populated where applicable. Zero
                            //  otherwise. Used with CE_EVT_SIGNAL
                            //  and CE_EVT_NET_FAILED
                            //  (ECE_PPP_AUTH_FAILURE).
   int neParam2;            //  Populated where applicable. Zero
                            //  otherwise. Used with CE_EVT_SIGNAL
                            //  (signal in dBm).
   int neParam3;                Populated where applicable. Zero
                                otherwise. Used with CE_EVT_SIGNAL
                                (signal in RSSI).
} stceNWEvt;                //
```

ceSetSignalNotification()

Sets the frequency of signal strength notification events from CommEngine to application.

Prototype `int ceSetSignalNotification(const unsigned int freq);`

Parameters

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

Return Values

`0`	Successful.
`ECE_PARAM_INVALID`	Parameter freq passed is not a valid frequency value.
`< 0`	Failed. See List of Error Codes.

Programming Notes

For an application to receive signal strength notifications, it must call this API with the required frequency. 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 supports wireless and signal strength makes sense. In addition to this, the network interface associated with wireless must be initialized.

CommEngine will send signal strength updates at frequency chosen by the application. For example, if the application chooses `ON (CE_SF_ON)` then it is sent. CommEngine starts sending signal strength notifications once device driver has been initialized, and will stop signal strength notifications once driver has been released.

Signal strength notifications will only be received when :

- Driver has just initialized
- If there are changes to the signal strength
- If any application turns their signal strength notifications on while the driver has initialized. This is to ensure that if any applications turned on their signal strength notification, they will immediately receive an update.

If applications were not able to process the signal strength notifications immediately, it can query current signal level by calling API `ceGetNWParamValue()` and passing `SIGNAL_LEVEL` as the `paramStr`.

When the application is preparing to go on power saving mode, it is recommended that all notifications be disabled by calling API `ceDisableEventNotification()`.

To obtain the current frequency of signal strength notification setting, use API `ceIsEventNotificationEnabled()`. Refer to Signal Event Notification for details of the event structure.

Signal Event Notification

The above section describes the process of enabling (and disabling) signal strength notification events. The signal event, similar to all events, is obtained by calling ceAPI `ceGetEvent()`. The event structure is described in the CommEngine Notification Event Structure section and is populated with signal data (see Table 12). The extended event data will hold the actual RSSI value with the data type of an integer. The actual RSSI value will be in the range of 0 to 31. If the value is 99, it implies the signal value is unavailable.

Table 12 Signal Event

Structure Element	Description
`unsigned int niHandle`	Handle to Network Interface
`unsigned int neEvt`	`CE_EVT_SIGNAL`
`int neParam1`	Signal strength as percentage. Values are in the range 0 to 100%. If the value is -1, it implies the signal value is unavailable.
`int neParam2`	Signal strength in dBm. Usually in the range of -51dBm (Excellent) to -113dBm (Poor). These are typical values may change depending on the geography and wireless medium (GPRS / CDMA / WiFi, etc). If the value is 99, it implies the signal strength is unavailable or unknown.
`int neParam3`	Actual RSSI value. Usually in the range of 0 to 31. This is the actual value that the GPRS modem is giving. If the value is 99, it implies the signal strength is unavailable or unknown.

ceIsEventNotificationEnabled()

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

Prototype

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

Parameters

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

Return Values

`1`	Event notification enabled.
`0`	Event notification disabled.
`ECE_PARAM_NULL`	Pointer freq passed in NULL.
`ECE_NOTREG`	Application needs to register to use this API.
`< 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 can signal strength notifications be sent at the chosen frequency.

ceGetNWIFCount()

Obtain number of network interfaces (NWIF) supported.

Prototype

```
int ceGetNWIFCount(void);
```

Return Values

`count`	Number of network interfaces
`< 0`	Failed. See List of Error Codes.

Programming Notes

Terminals support multiple communication devices (or modems) referred to 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 this API be called prior to API `ceGetNWIFInfo()`. The count provided by this API is 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 int stArraySize,
unsigned int *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 the 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 int niHandle;`	//	Handle to Network Interface
`char niCommTech[16];`	//	Null terminted str. See Constants section for complete list.
`char niDeviceName[32];`	//	/DEV/COMx /DEV/WLN1 /DEV/ETH1
`unsigned int niRunState;`	//	1—Running, 2—Ready, 3-Not Ready, 0-Failed to run
`int niErrorCode`	//	Associated error code
`unsigned int niStartUpMode;`	//	0—Auto startup, 1—Manual startup
`char niDeviceDriverName[18+1];`	//	`Format` `[2 char media type] [2 char vendor ID][4 char device id]>.lib`

```
    int                         //    1-Ethernet, 2-PPP, 3-Dial
    niDeviceDriverType;
} stNIInfo;
```

ceStartNWIF()

Starts network interface associated with the handle.

Prototype

```
int ceStartNWIF(const unsigned int niHandle, const int 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.
`ECE_START_STOP_BUSY`	`ceStartNWIF()` is still busy processing a previous `ceStartNWIF()` or `ceStopNWIF()` request. Possible scenarios are: • CommEngine is still bringing up or tearing down the network • network is out (cable was unplugged or no signal) and CommEngine is busy re-establishing the network • In the middle of the connection, the battery was removed so CommEngine is busy tearing down the network
`ECE_EVENTS_AGAIN`	Enabling of events is required when using `ceStartNWIF()`. Not enabling events will produce this error code.
`ECE_STATE_ERR`	Application passed an invalid state. Possible scenarios are: • CommEngine state has already reached target connection state • target connection state is greater than CommEngine's current state
`ECE_PARAM_INVALID`	Application passed an invalid network interface such as a dial only network interface.
`ECE_DEVOWNER`	CommEngine is currently not the owner of this device. Another application might have requested this device and has not yet released it to CommEngine.
`ECE_DEVBUSY`	Device is currently busy processing another request. If the device is a shared device, it most likely is being used by another NWIF.

`ECE_INIT_FAILURE`	Device failed to initialize. The device driver library file does not exist.
`< 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 Events.

In the case of having static IP configurations, CommEngine will always post `CE_EVT_NET_UP`. It is up to the user to ensure validity of the set environment.

Enabling of events is required when using `ceStartNWIF()`. If events are not enabled an error code of `ECE_NO_CE_EVENT` will be returned. Passing a Dial only network interface handle will result in an error code of `ECE_PARAM_INVALID`.

An error code of `ECE_START_STOP_BUSY` will be returned if CommEngine is still processing a previous `ceStartNWIF()` or `ceStopNWIF()` call.

In the case of a PPP Authentication failure, CommEngine will try to re-establish the network three (3) times before finally tearing the network down. Applications can change the PPP configurations at this time, and no call to `ceStartNWIF()` will be needed to re-connect. If changes to PPP configurations were done, after the three retries have been exhausted, `ceStartNWIF()` should be called again.

If battery was removed in the middle on a connection, CommEngine will tear the network down. It is up to the application to start the network again once the battery is replaced.

ceStopNWIF()

Stops network interface associated with handle.

Prototype

```
int ceStopNWIF(const unsigned int niHandle, const int 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 disconnect all the way, set parameter cl to `CE_DISCONNECT`. To perform incremental disconnect set `cl` to `CE_NETWORK`, `CE_LINK_` or `CE_NETWORK`. For definitions see the Constants section.

Return Values

`0`	Successful.
`ECE_START_STOP_BUSY`	`ceStartNWIF()` is still busy processing a previous `ceStartNWIF()` or `ceStopNWIF()` request. Possible scenarios are: • CommEngine is still bringing up or tearing down the network • network is out (cable was unplugged or no signal) and CommEngine is busy re-establishing the network • In the middle of the connection battery was removed so CommEngine is busy tearing down the network
`ECE_EVENTS_AGAIN`	Enabling of events is required when using `ceStartNWIF()`. Not enabling events will produce this error code.
`ECE_STATE_ERR`	Application passed an invalid state. Possible scenarios are: • CommEngine state has already reached target connection state • target connection state is greater than CommEngine's current state
`ECE_PARAM_INVALID`	Application passed an invalid network interface such as a dial only network interface.
`ECE_DEVOWNER`	CommEngine is currently not the owner of this device. Another application might have requested this device and has not yet released it to CommEngine.
`ECE_DEVBUSY`	Device is currently busy processing another request. If the device is a shared device, it most likely is being used by another NWIF.

`< 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 stopped (or moves to the next state), CommEngine posts an event. See List of CommEngine Events.

ceSetNWIFStartMode()

Specifies if a network interface is automatically initialized during start up (auto mode) or explicitly started (manual mode).

Prototype

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

Parameters

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

Return Values

`0`	Successful.
`ECE_PARAM_INVALID`	Application passed an invalid startmode or `niHandle` passed is out of range.
`< 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 (it is equivalent to `ceStartNWIF` (handle, `CE_CONNECT`)). For starting incrementally, set the start mode to manual (`CE_SM_MANUAL`).

In the case of a Dial NWIF, setting of start mode does not apply. Any start mode value set to Dial NWIF will be ignored.

ceGetNWIFStartMode()

Obtain the network interface start mode.

Prototype

```
int ceGetNWIFStartMode(const unsigned int niHandle);
```

Parameters

`In:`	`niHandle`	Handle to network interface. Returned by API `ceGetNWIFInfo()` in structure element `niHandle` of parameter `stArray`.

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()`.
`ECE_PARAM_INVALID`	Application niHandle passed is out of range.
`< 0`	Failed. See List of Error Codes.

Programming Notes

In the case of a Dial NWIF, start mode will always return `CE_SM_MANUAL`.

ceSetNWParamValue()

Set network (NW) parameter.

Prototype

```
int ceSetNWParamValue(const unsigned int niHandle, const char paramStr[],
const void *paramValue, const unsigned int 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.
`ECE_NOTREG`	Application must register before attempting this API.
`ECE_PARAM_INVALID`	Possible reasons are: • application passed an a dial only network interface • application passed an Ethernet driver type for a `PPP_CONFIG` • `niHandle` passed is out of range
`ECE_PARAM_NULL`	`paramValue` passed is missing a required field. In the case of an `IP_CONFIG`, IP address, Subnet Mask and Gateway are required.
`ECE_SUBNET_MASK_INVALID`	Subnet Mask passed is invalid.
`ECE_NOT_SUPPORTED`	`niHandle` passed does not support `paramStr`.
`< 0`	Failed. 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 running.

In the case of setting a ncSubnet, an error code of `ECE_SUBNET_MASK_INVALID` will be returned if it is invalid. An error code of `ECE_PARAM_NULL` will be returned if a required parameter is not set. When setting ncDNS1, ncDNS2 may be NULL. However, the reverse is not allowed. If ncDNS2 is not NULL and ncDNS1 is NULL, ncDNS2 and ncDNS1 will be reversed internally by CE.

When setting for a static IP configuration, make sure to set the variable ncDHCP to 0. As for using DHCP configuration, make sure to set the variable ncDHCP to 1.

ceGetNWParamValue()

Fetch network (NW) parameter value.

Prototype

```
int ceGetNWParamValue(const unsigned int niHandle, const char
paramStr[],void *paramValue, const unsigned int paramValueSize, unsigned
int *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.
`Out:`	`paramValueLen`	Number of bytes containing data in `paramValue`. This value is returned and is always less than or equal to `paramValueSize`.

Return Values

`0`	Parameter value obtained.
`ECE_NOTREG`	Application must register before attempting this API.
`ECE_PARAM_INVALID`	Possible reasons are: • application passed a dial only network interface • application passed an Ethernet driver type for a `PPP_CONFIG` • `niHandle` passed is out of range
`ECE_SIGNAL_FAILURE`	Query of signal level to DDI Driver failed.
`< 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 the network interface is not running.

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

ceStartDialIF()

Starts the dial interface and provides the handle to device.

Prototype

```
int ceStartDialIF(const unsigned int diHandle, const unsigned int deSize,
char *dialErrorStr);
```

Parameters

`In:`	`diHandle`	Handle to dial interface. Returned by API `ceGetNWIFInfo` in structure element `niHandle` of parameter stArray.
`In:`	`deSize`	Size of buffer `dialError`.
`Out`	`dialErrorStr`	Null terminated error string provided by the dial device driver.

Return Values

`> 0`	Dial device handle
`ECE_DIAL_ONLY_FAILED`	Dial interface failed. See `dialErrorStr` for error details.
`ECE_PARAM_INVALID`	diHandle passed may not be a Dial driver type or is out of range
`ECE_DD_NOT_READY`	Device driver is currently not in the proper state to process request.
`ECE_DEVBUSY`	Device is currently busy processing another request. If the device is a shared device, it most likely is being used by another NWIF.
`ECE_DEVOWNER`	CommEngine is currently not the owner of this device. Another application might have requested this device and has not yet released it to CommEngine.
`ECE_START_STOP_BUSY`	CommEngine is tearing down the connection. Possible causes may be: • In the middle of the connection battery was removed • Connection was lost
`< 0`	Failed. See List of Error Codes.

Programming Notes

After the dial interface is successfully started, application uses the device handle returned to perform `read()` and `write()` operations. Even if the applications share the dial device handle that was returned by `ceStartDialIF()`, only the application that called `ceStartDialIF()` will have a successful `read() / write()`.

When complete, close the interface using ceAPI `ceStopDialIF()`.

If a `Connection Lost` occurs in the middle of a `DIAL ONLY` or a `GSM ONLY` connection, CommEngine will revoke the ownership of the application on the device handle and will automatically tear down the connection. Applications will experience write / read error when this happens.

ceStopDialIF()

Stops the previously started dial interface.

Prototype

```
int ceStopDialIF(const int short diHandle);
```

Parameters

`In:`	`diHandle`	Handle to dial interface. Returned by API `ceGetNWIFInfo` in structure element `niHandle` of parameter `stArray`. This is the same handle provided in `ceStartDialIF()`.

Return Values

`0`	Interface successfully closed
`ECE_PARAM_INVALID`	`diHandle` passed may not be a Dial driver type or is out of range
`ECE_DD_NOT_READY`	Device driver is currently not in the proper state to process request.
`ECE_NOTASKID`	`ceStopDialIF()` is called by a different application instead of the on that called `ceStartDialIF()`.
`ECE_START_STOP_BUSY`	CommEngine is tearing down the connection. Possible causes may be: • In the middle of the connection battery was removed • Connection was lost
`< 0`	Failed. See List of Error Codes.

Programming Notes

Use ceAPI `ceStopDialIF()` to close the dial interface that was previously started using `ceStartDialIF()`.

The first application that called `ceStartDialfIF()` successfully will be the only application that can call `ceStopDialIF()` successfully. No other application will be allowed to bring down the PPP connection.

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. Verix eVo application VxNCP activated.
`ECE_VXNCP_CONSOLE`	Application does not own console. Applications must own console before NCP can be activated.
`ECE_VXNCP_PIPE`	VxNCP is not running.
`ECE_VXNCP_PIPECR`	Unable to create pipe to communicate with NCP.
`< 0`	Failed. See List of Error Codes.

Programming Notes

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

VxNCP sets printer font to default. If application wants to use previous settings of font, application must set the printer font again after VxNCP exits.

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`	Verix eVo 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	Success. Value returned in parameter `verStr`.
ECE_VER_UNAVAIL	CommEngine version is unavailable.
ECE_PARAM_INVALID	Parameter component passed is invalid.
< 0	Failed. See List of Error Codes.

Programming Notes

Application must register to obtain version information.

The version information is returned in the form “mm.nn.pp.bb” where:

- mm – major number
- nn – minor number
- pp – patch number
- bb – build number

For example “01.08.04.07” is valid version string.

In this example, the size of this version string is 11. Using a buffer of reasonable size (ex. 32) to account for future changes or expansions is recommended.

Constants, Defines & Miscellany

Use the following information to determine constants, definitions, 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.

Event ID	Event Value	Dist. Scope	Description
`CE_EVT_NET_UP`	0x0001	Broad-cast[5]	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. An error code of `ECE_PPP_AUTH_FAILURE` stored in `neParam1`, will be received if PPP Authentication failed. Extended errors can also be extracted from `applExtEvtData`.
`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 in percentage stored in `neParam1`. Signal strengths are sent once device driver has been initialized and will stop once device driver has been released.
`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)` or ceStartNWIF(CE_CONNECT).
`CE_EVT_START_FAIL`	0x000A	Single App	Event posted to requesting application after failed `ceStartNWIF()`.
`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) or ceStopNWIF(CE_DISCONNECT).` This can also be received by the requesting application if there are errors encountered during `ceStartNWIF()` or `ceStopNWIF().`
`CE_EVT_STOP_FAIL`	0x000E	Single App	Event posted to requesting application after failed `ceStopNWIF()`.

`CE_EVT_DDI_APPL`	0x000F	Subscribed	The extended event data contains the device driver application event data. For a GSM / GPRS driver, this can also contain reasons for a failed start-up. The application event is stored in the event structure in the variable `neParam1`.

Constants

Constant	Value	Description
Comm. Tech		Communication Technology Descriptor String
`CE_COMM_TECH_ETH`	Ethernet	Ethernet
`CE_COMM_TECH_WIFI`	WiFi	Wireless Fidelity
`CE_COMM_TECH_GPRS`	GPRS	General Packet Radio Service
`CE_COMM_TECH_CDMA`	CDMA	Code Division Multiple Access
`CE_COMM_TECH_PPPDIAL`	PPP/Dial	Point to Point Protocol over Dial
`CE_COMM_TECH_DIALONLY`	Dial	Dial over phone line (POTS)
`CE_COMM_TECH_PPPGSM`	PPP/GSM	Point to Point Protocol over GSM
`CE_COMM_TECH_GSMONLY`	GSM	Global system for mobile communications
Component Ver.		Component Name for Version Information
`CE_VER_CEIF`	1	CommEngine Interface Library
`CE_VER_VXCE`	2	CommEngine
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.
`CE_SF_ON`	1	Signal Strength notification are sent.
`CE_SF_OFF`	0	No notifications are sent.
Run State		The run state of the network interface.
`CE_RUN_STATE_FAILED`	0	NWIF failed to start
`CE_RUN_STATE_RUNNING`	1	NWIF is successfully running
`CE_RUN_STATE_READY`	2	NWIF is not running and is pending start
`CE_RUN_STATE_NOT_READY`	3	NWIF cannot run as device is unavailable
Device Driver Type		The driver type of network interface.
`CE_DRV_TYPE_ETHERNET`	1	Driver type is Ethernet
`CE_DRV_TYPE_PPP`	2	Driver type is PPP
`CE_DRV_TYPE_DIAL`	3	Driver type is Dial

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
`SIGNAL_LEVEL`	`stSignalLevel`	-	✓	-	See Configuration Structure stSignalLevel

NOTE The list of network parameters is not exhaustive and will expand based on requirement. Treck API provides a high degree of configurability. However not all options are applicable.

Configuration Structure stNI_IPConfig

```
typedef struct
{
 unsigned int ncDHCP;          //   1 = DHCP, 0 = Static
 unsigned long ncIPAddr;       //   Mandatory if Static IP
 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                          //   YYYYMMDDHHMMSS format (read only)
dhcpLeaseStartTime[14];
 char                          //   YYYYMMDDHHMMSS format (read only)
dhcpLeaseEndTime[14];
 unsigned int                  //   Yes or No (read only)
ncIsConnected;
} stNI_IPConfig;
```

NOTE

Variable `ncIsConnected` shows if an IP address was successfully obtained by CommEngine. It does not reflect CommEngine's current state. Its value may differ on a `CE_EVT_NET_OUT` event depending on the device driver type.

Note the following scenarios:

- For an Ethernet type of driver:

  Once the cable is unplugged, `ncIsConnected` will become false as the network connection is lost. CommEngine will try to reestablish the connection once the cable is plugged back in and the IP will be reacquired as a different LAN cable may have been plugged.

- For a GPRS device driver:

  Once the driver detects that the signal has been lost, `CE_EVT_NET_OUT` will be sent out by CommEngine but `ncIsConnected` will still be true as the network was not torn down.

Configuration Structure stNI_PPPConfig

```
typedef struct
{
  pppAuthType ncAuthType;          //    Type of PPP authentication
  char                             //    PPP username, optional
ncUsername[MAX_USERNAME];
```

```
  char                              //    PPP password, optional
ncPassword[MAX_PASSWORD];

} stNI_PPPConfig;
```

Configuration Structure stSignalLevel

```
typedef struct

{

 int iSignal1;                      //    signal in percentage

 int iSignal2;                      //    signal in dBm

 int iSignal3;                      //    signal in RSSI

} stSignalLevel;
```

List of Error Codes

Error codes returned by ceAPI are negative (< 0). `ceAPIs` 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_SUCCESS`	0	Success return code
`ECE_NO_MEMORY`	-1000	No memory
`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_CREATEMON`	-1004	Application creates a monitoring thread as part of the registration. This error occurs if the monitor task has failed.
`ECE_SENDCEREQ`	-1005	Application unable to send request event to CommEngine.
`ECE_RESPFAIL`	-1006	Application failed to receive a response.
`ECE_TIMEOUT`	-1007	No response, timeout error.

`ECE_NOTREG`	-1008	Application has not successfully registered with CommEngine. This error returned if an application is attempting a ceAPI prior to successful registration.
`ECE_DEVNAMELEN`	-1009	Device name length is beyond limits. Verify device name is accurate and null terminated.
`ECE_PARAM_NULL`	-1010	Parameter is null when one is not expected.
`ECE_PARAM_INVALID`	-1011	Parameter is invalid or out of range.
`ECE_VXNCP_PIPE`	-1012	Pipe VxNCP is not running.
`ECE_VXNCP_PIPECR`	-1013	Unable to create temp pipe to create with VxNCP.
`ECE_VXNCP_CONSOLE`	-1014	Application is not console owner.
`ECE_VER_UNAVAIL`	-1015	CommEngine version information is unavailable.
`ECE_CDMAPP`	-1016	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`	-1017	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`	-1018	Unknown device name. This is not a communication device that CommEngine is aware of.
`ECE_DEVOWNER`	-1019	CommEngine is not the owner of the device. This is a communication device that CommEngine is aware of but currently not in its possession.
`ECE_DEVBUSY`	-1020	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`	-1021	Task Id provided as parameter does not exist.
`ECE_NODEVREQ`	-1022	No prior device request was made. Nothing to cancel.
`ECE_CANNOT_UNREG`	-1023	Application cannot unregister for one of the following reasons: • Application has registered itself as CommEngine's device management application • This application has devices checked out or has pending requests for devices. Check in all devices or cancel all requests before attempting to unregister again.

`ECE_NOT_SUPPORTED`	-1024	The feature requested is not supported. Check version information and releases notes of ceAPI and CommEngine.
`ECE_NWIF_BUSY`	-1025	The configuration cannot be changed as the network interface is in a run state. To change configuration -- stop, change configuration and restart.
`ECE_DD NOT_READY`	-1026	The Device Driver is not in a state to provide response to parameter requests. In the case of an `ioctl()` command, CommEngine is expecting the nwif's state to at least be in OPEN before it proceeds.
`ECE_NO_CE_EVENT`	-1027	No CommEngine Notification Event in queue or Event Notification was not enabled.
`ECE_EX_CMD_FAILED`	-1028	Execute Command Failed.
`ECE_NOTDEVOWNER`	-1029	Application did not check out this device or application is attempting to check if the device did not check out previously.
`ECE_PARAM_NOT_FOUND`	-1030	Parameter requested is not supported.
`ECE_PARAM_READ_ONLY`	-1031	Parameter is a read only. Parameter can not be set.
`ECE_PARAM_SAVE_FAIL`	-1032	Saving to Delta file failed.
`ECE_STATE_ERR`	-1033	`ceStart/StopNWIF()` state error. Target mode passed might be less than CommEngine's current state.
`ECE_EVENTS_AGAIN`	-1034	Application is attempting to enable events again after a successful event enabling.
`ECE_DIAL_ONLY_FAILED`	-1035	`ceStartDialIF()` failed.
`ECE_SUBNET_MASK_INVALID`	-1036	Invalid subnet mask was passed.
`ECE_START_STOP_BUSY`	-1037	A previous call to `ceStartNWIF()` or `ceStopNWIF()` is still being processed.
`ECE_PPP_AUTH_FAILURE`	-1038	PPP Authentication failure. This is returned in as a value of `neParam1` in the structure `stceNWEvt`.
`ECE_INIT_FAILURE`	-1039	This is returned when ddi driver can not be initialized.
`ECE_SIGNAL_FAILURE`	-1040	This is returned when query of signal level failed.

NOTE

The list of ceAPI Error codes is not exhausive. This will be revised as necessary.

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.

Handling CommEngine Events

Managing and Controlling the Connection

In this sample, the application starts the CommEngine incrementally and notifies 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.

This sample code illustrates how an application enables events using ceAPI `ceEnableEventNotification()` and manages them.

sample_handling_commengine_events.txt
(see PDF embedded file attachment)

CE_EVT_DDI_APPL for a GPRS / GSM driver

Some device drivers may send application events directly to the application. In the case of a GPRS / GSM driver, it sends strings to application stating reasons for failure. The sample code below shows an example on how applications can extract the raw data sent with `CE_EVT_DDI_APPL`.

CE_EVT_DDI_APPL_GPRS_GSM.txt
(see PDF embedded file attachment)

Sample code on ceSetNWParamValue() / ceGetNWParamValue()

The sample code below illustrates how applications can set network parameters using `ceSetNWParamValue()` and get network parameters using `ceGetNWParamValue()`.

Function `SetParametersEthernet()` illustrates how applications can set static ip configurations.

Function `SetParametersPPP()` illustrates how applications can set username, password and authentication type for PPP connections.

sample_code_ceSetNWParamValue_ceGetNWParamValue.txt
(see PDF embedded file attachment)

Sample code on ceSetDDParamValue() / ceGetDDParamValue()

The sample code below illustrates how applications can set device driver parameters using `ceSetDDParamValue()` and get device driver parameters using `ceGetDDParamValue()`.

sample_code_ceSetDDParamValue_ceGetDDParamValue.txt
(see PDF embedded file attachment)

Sample code on ceEXCommand()

Sample code below illustrates how applications can send `ioctl()` calls directly to device driver using `ceExCommand()`.

sample_code_ceEXCommand.txt

(see PDF embedded file attachment)

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, for application A and application B to communicate with each other, each application creates a pipe and communicates with each other by sending and receiving messages.

Payload Size

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 n bytes, where n is the size of the payload. The maximum payload is 4096bytes.

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.

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
`mxRecvEx()`	Reads pending message and extended pipe header information from queue

mxCreatePipe()

Creates a Verix eVo 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. The pipe name must have the P: prefix. Pipe name may be “P:” 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 eVo 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 closed successfully. Returns zero when the pipe handle is not a valid pipe handle. Returns zero in all other circumstances.

mxGetPipeHandle()

Obtains pipe handle associated with pipe name.

Prototype

`int mxGetPipeHandle(const char *pipeName)`

Parameters

`In:`	`pipeName`	Target pipe name whose handle to be obtained. Only the first 8 characters are considered in the comparison. The pipe name cannot be NULL or an 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 length 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.

mxRecv()

Reads pending message from queue.

Prototype

```
int mxRecv(const int pH, int *pHFrom, void *dataPayload, const unsigned
int dataPayloadSz, unsigned int *dataPayloadLen))
```

Parameters

`In:`	`pH`	Pipe handle. Created using `mxCreatePipe()`
`Out:`	`pHFrom`	Sender pipe handle
`Out:`	`dataPayload`	Pointer to buffer to receive payload data from sender
`In:`	`dataPayloadSz`	Size of buffer `dataPayload` in bytes
`Out:`	`dataLoadLen`	Length of data in payload buffer

Return Values

`> 0`	Message read and number of bytes in payload
`= 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.

mxRecvEx()

Reads pending message from queue.

Prototype

```
int mxRecvEx(const int pH, pipe_extension_t *extPI, void *dataPayload,
const unsigned int dataPayloadSize, unsigned int *dataPayloadLen)
```

Parameters

`In:`	`pH`	Pipe handle. Created using `mxCreatePipe()`
`Out:`	`pHFrom`	Sender pipe handle
`Out:`	`dataPayload`	Pointer to buffer to receive payload data from sender
`In:`	`dataPayloadSz`	Size of buffer `dataPayload` in bytes
`Out:`	`dataLoadLen`	Length of data in payload buffer

Return Values

`> 0`	Message read and number of bytes in payload
`= 0`	Message read and has no payload
`< 0`	Read error or no pending message. See mxAPI Error Codes

Programming Notes

Structure `pipe_extension_t` is defined in VFSDK file `svc.h`.

Refer to mxRecv() for additional details.

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_UNKNOWN`	-2000	Undetermined error
`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(s)
`EMX_PIPE_HANDLE`	-2005	Pipe handle not valid
`EMX_PAYLOAD_SIZE`	-2006	Payload size should be positive non-zero value and less than limit
`EMX_NULL_PARAM`	-2007	Input parameter is unexpected null
`EMX_DEST_PIPE`	-2008	Target pipe for `mxSend()` does not exist or not configured for messages
`EMX_PIPE_FULL`	-2009	The destination pipe is full and the message cannot be sent

Message Formatting mfAPI

Applications need to exchange data. To successfully do so, participating applications must have a common format. Applications use Message Exchange API (mfAPI) format data as a TLV list for exchange.

The data exchanged between applications usually consists of a list of name value pairs. This combination of name and value is referred to as TLV or its expansion Tag, Length and Value. TLV is somtimes referred to as *Type*, Length and Value.

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 *mf*API provides API for constructing the request message as a TLV list and to extract individual TLVs from the response message.

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:

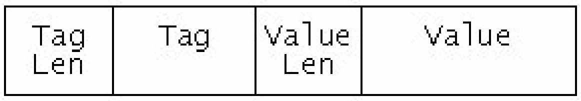


Figure 3 Fixed Length TLV

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.

TLV List

This section describes the message format. The TLV list is a sequence of TLVs preceded by a Header (Figure 4).

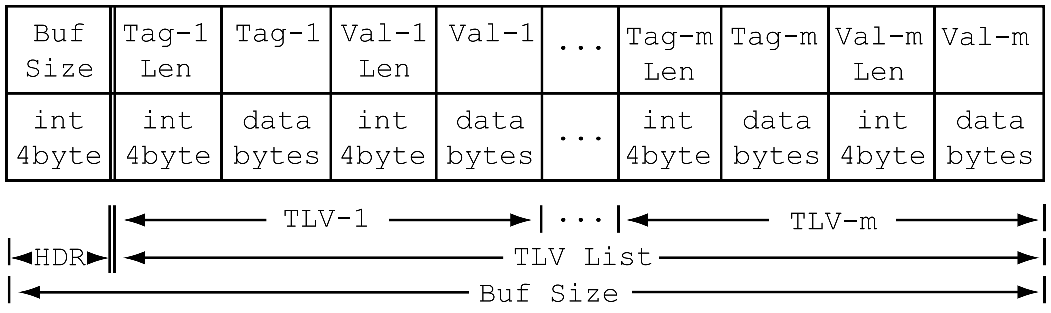


Figure 4 Fixed length TLV list

The header consists of one field:

Field Name	Data Type	Size	Description
Buffer Size	int	sizeof(int) 4bytes	Size of the Message - computed by adding the size of the header (4 bytes) to the size of the TLV List. For example, if the size of TLV List is 96 bytes then the value of this field is 100bytes.
	Total	**4 bytes**	

Message Formatting mfAPI – Summary

The Message Formatting API consists of the functions listed in the table below. The following section has detailed descriptions of each API.

API	Description
Handle Management	
`mfCreateHandle`	Obtains a new handle
`mfDestroyHandle`	Releases 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.
`mfAddTLV()`	Adds TLV to the buffer.
Fetch Operations	
`mfFetchInit()`	Initializes the Message buffer for Fetch (read) operations and returns the session handle.
`mfFetchClose()`	Closes the session for Fetch operations.
`mfFetchReset()`	Repositions the “Next” pointer to the top of the TLV list.
`mfPeekNextTLV()`	Obtains the next Tag length and the size of its Value. The “Next” pointer is not advanced.
`mfFetchNextTLV()`	Fetches the next TLV in sequence from list. Moves the “Next” pointer to the next TLV in the list.

`mfFindTLV()`	Fetches the first TLV from list that matches tag.
Helper Function	
`mfEstimate()`	Computes memory requirement for a set of tags.

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

Release the handle using `mfDestroyHandle()`. This is the last API to be called as the handle is relinquished.

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`	Session handle returned by `mfCreateHandle()` and initialized for add operations by `mfAddInit()`.
`In:`	`tagLen`	Size of field tag
`In:`	`*tag`	Pointer to Tag field. Its size must be as specified in `tagLen`. This parameter may not be NULL.
`In:`	`valueLen`	Size of value field
`In:`	`*value`	Pointer to value field. Its length must be as specified in parameter `valueLen`. This parameter 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`. 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.

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:`	`tlvBuffer`	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 operation.
`< 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.

Programming Notes

Applications using this API can foresee the next tag without “fetching” it. This API does not advance the “Next” pointer. Applications can plan for space requirements by knowing the size. This call is usually followed by `mfFetchNextTLV()`.

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.
`Out:`	`tegLen`	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. `mfFindVarTLV` 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 returnedor invalid operation. See mfAPI Error Codes.

Programming Notes

The “Next” pointer moves to the next TLV in the 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.
`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.

CHAPTER 6

Verix eVo Communication Engine Application

The Verix eVo Communication Engine or CommEngine (`VXCE.OUT`) is the core component of the communication infrastructure. The boot strap application `VXeVo.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 Verix eVo 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 eVo. Verix eVo starts the executable files specified in the Verix eVo manifest file. The manifest contains CommEngine and VxNCP, which is the application with the User Interface (UI) and provides configuration and management services to Verix eVo components. CommEngine after start up brings up the device driver and the Treck TCPIP stack.

Conditionally Starting CommEngine

On startup Verix eVo looks for configuration variable `*NO.VXCE` in GID1. If this variable is not present or its value is zero, CommEngine is started. If `*NO.VXCE` is set to 1, CommEngine is not started. Note that `*NO.VXCE` is a Verix eVo configuration parameter and is referenced here as it affects CommEngine.

VxNCP also has a conditional start-up variable `*NO.VXNCP`. The same rules apply to `*NO.VXCE`.

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-compliant. 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

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 and Metadata File

CommEngine maintains its configuration in one configuration file:

Configuration Filename	Description
`VXCEHW.INI`	Configuration *data* file. This file contains the name and value of the configuration variable. A change to configuration value updates this f
`VXCEHW.MTD`	Metadata file. This file contains the list of config.sys variables that CommEngine uses.

Network Configuration via Configuration Delta File

CommEngine configurations can be updated via downloading a configuration delta file. The configuration delta file can be described as a file containing a section name and having keys that the application users can define. Applications may download their own CommEngine Configuration Delta File named `VXCEHW.INI` in GID 1.

A configuration file may contain comment and blank lines to make the file self documenting and readable. A comment line starts with the semi-colon (;), is 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.

Table 13 List of CommEngine Configuration Section Names

Configuration Section Name	Description
ETH	Ethernet
PPPDIAL	PPP over Dial
DIAL	Dial only (no PPP)
CDMA	CDMA
GPRS	GPRS
PPPGSM	PPP over GSM
GSM	GSM only (no PPP)

Table 14 List of CommEngine Configuration Key Names

Configuration Key Name	Description
startmode	This states if the NWIF is Auto or Manual
dhcp	This states if DHCP (1) or Static (0). Keys ipaddr, subnet, gateway, dns1 and dns2 will be ignored if dhcp is set to 1.
ipaddr	IP address to be used.
subnet	Subnet Mask to be used.
gateway	Gateway IP Address.
dns1	Primary DNS IP Address.
dns2	Secondary DNS IP Address.
username	Username, applicable for PPP connection.
password	Password, applicable for PPP connection. **Note:** Password is case sensitive
authtype	Authentication type, applicable for PPP connection. • Values can be 1 (NONE), 2 (PAP), 3 (CHAP), and 4 (MSCHAP). • CommEngine uses NONE as a default if this is not specified.

A sample CommEngine Configuration Delta File is provided below:

```
[ETH]
startmode = Auto
dhcp = 0
ipaddr = 10.64.80.175
subnet = 255.255.252.0
gateway = 10.64.80.1
dns1 = 10.64.80.35
dns2 = 10.64.128.227

[PPPDIAL]
startmode = Manual
dhcp = 1
ipaddr =
subnet =
gateway =
dns1 =
dns2 =
username = verifone
;Note password is case sensitive. In this example the
letters are lower case
password = c4h9oh
authtype =
```

Network Configuration via GID1 CONFIG.SYS parameters

CommEngine configuration can be specified via `CONFIG.SYS` parameters in GID1. This is an alternate approach for users to configure CommEngine.

The configuration parameter is a combination of the section name (see Table 13) and key name (see Table 14). Taking the example of `PPPDIAL` username the corresponding configuration parameter name is: `PPPDIAL.USERNAME`. There is a period (.) between the section name and key name. In general, the configuration parameter is:

```
<SectionName>.<KeyName> = <KeyValue>
```

Where `<SectionName>` is defined in Table 13 and `<KeyName>` is defined in Table 14. The key value, `<KeyValue>` is same as specified in the configuration delta file (see Network Configuration via Configuration Delta File).

Here is another example where the ETH configuration parameters and its values are listed.

```
ETH.STARTMODE = AUTO
ETH.DHCP = 0
ETH.IPADDR = 10.64.80.175
ETH.SUBNET = 255.255.252.0
ETH.GATEWAY = 10.64.80.1
ETH.DNS1 = 10.64.80.35
ETH.DNS2 = 10.64.128.227
```

NOTE Only the first letter is processed for `STARTMODE. ETH.STARTMODE = A`, will be the same as `ETH.STARTMODE = AB`. Both will be taken in as `AUTO`. Any invalid values will result to `MANUAL`.

Configuration Precedence

All configuration parameters have a default value that CommEngine uses if none is specified. To override the default values, the user has the option to specify the value in the configuration delta file (.INI, see Network Configuration via Configuration Delta File). By specifying a `CONFIG.SYS` parameter, it overrides the value in the delta file.

Table 15 Configuration Precedence

Location	Precedence
Default value	Lowest
Configuration value in delta file (Network Configuration via Configuration Delta File)	
Configuration value in CONFIG.SYS (Network Configuration via GID1 CONFIG.SYS parameters)	Highest

CommEngine at start up reads the `CONFIG.SYS` values and updates the configuration delta file. After the value is updated, the `CONFIG.SYS` value is deleted.

When a value is accessed, CommEngine looks for it in the delta file and if not present uses the default value.

Configuration Options

To override the default value users have two options – the delta file and `CONFIG.SYS` . The two options are not mutually exclusive and using one does not preclude the other. However, when both options are used, the value specified via `CONFIG.SYS` takes precedence.

This gives users the option to specify the configuration via the delta file and is usually applicable to its entire terminal base. However should a few terminals require a different configuration, it can be altered by specifying `CONFIG.SYS` parameters rather than having a different delta file.

CommEngine Logging

By default, CommEngine logs are disabled, but applications can choose to enable CommEngine logs by defining configuration variable CELOG to C (CELOG=C). CommEngine logs, if enabled, by default only shows CommEngine major logs. CommEngine filter can be changed by defining configuration variable CEFIL to a value depending on how much details the application would like to see. Refer to the ReadMe.txt document packaged along with the software for the list of filter values.

Interface with Device Drivers

CommEngine starts and manages the device drivers. All device drivers work under the aegis of the Device Driver Manager.

The Device Driver Manager implements the Device Driver Interface (DDI). CommEngine uses the DDI to manage it.

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.

Application Interface

Application services are provided by CommEngine via `CEIF.LIB`. Applications link with `CEIF.LIB` which under the covers interfaces with CommEngine.

Application Events

CommEngine can forward application event and data from the driver. This is commonly used for notification purposes in case the application needs to do further processing when certain events occur. Below are the events that the application can receive:

Table 16 Common Application Events

Param Name (Type:macro)	Event Value	Event Data	Event Data Type	When Sent
NO_BATTERY	200	`No battery is attached or recognized`	Null-terminated String	When battery is removed from the terminal
NO_SIM_CARD	201	`SIM card not present or not connected`	Null-terminated String	When there's no SIM card present

SIM_PIN_RETRY	202	Can be any of the following:
		`SIM PIN: [attempts]`
		`SIM PIN: UNKNOWN`
		`SIM PUK: [attempts]`
		`SIM PUK: UNKNOWN`
		`UNKNOWN`
		Where:
		`[attempts]` – the number of remaining attempts
		Example:
		`SIM PIN: 10`
		- There are 10 remaining SIM PIN attempts before the SIM card asks for SIM PUK.
			Null-terminated String	When SIM PIN or PUK verification fails

Table 17 GPRS Application Events

Param Name (Type:macro)	Event Value	Event Data	Event Data Type	When Sent
GPRS_SIGNAL_QUALITY_STATUS	300	String that contains a any integer from 0 to 7, or 99.	Null-terminated String	When a signal quality URC is received from the modem
GPRS_ROAM_STATUS	301	Can be any of the following: `0` - Registered to home network or not registered `1`- Registered to other network	Null-terminated String	When a roaming status URC is received from the modem
GPRS_MESSAGE_STATUS	302	Can be any of the following: `0` - No unread SMS `1`- There is an unread SMS	Null-terminated String	When a message URC is received from the modem
GPRS_CALL_STATUS	303	Can be any of the following: `0` - No active calls `1`- There is an active call	Null-terminated String	When a call status URC is received from the modem
GPRS_SMSFULL_STATUS	304	Can be any of the following: `0` - SMS memory not full `1`- SMS memory full	Null-terminated String	When an SMS full memory URC is received from the modem
GPRS_SERVICE_STATUS	305	Can be any of the following: `0` - Not registered to any network `1`- Registered to any network	Null-terminated String	When a network registration status URC is received from the modem

NOTE

These events are defined in `vos_ddi_app_evt.h`. Applications registering to CE should include this header file if it wants to process GPRS application events.

Table 18 GSMApplication Events

Param Name (Type:macro)	Event Value	Event Data	Event Data Type	When Sent
GSM_SIGNAL_QUALITY_STATUS	300	String that contains a any integer from 0 to 7, or 99.	Null-terminated String	When a signal quality URC is received from the modem
GSM_ROAM_STATUS	301	Can be any of the following: `0` - Registered to home network or not registered `1`- Registered to other network	Null-terminated String	When a roaming status URC is received from the modem
GSM_MESSAGE_STATUS	302	Can be any of the following: `0` - No unread SMS `1`- There is an unread SMS	Null-terminated String	When a message URC is received from the modem
GSM_CALL_STATUS	303	Can be any of the following: `0` - No active calls `1`- There is an active call	Null-terminated String	When a call status URC is received from the modem
GSM_SMSFULL_STATUS	304	Can be any of the following: `0` - SMS memory not full `1`- SMS memory full	Null-terminated String	When an SMS full memory URC is received from the modem
GSM_SERVICE_STATUS	305	Can be any of the following: `0` - Not registered to any network `1`- Registered to any network	Null-terminated String	When a network registration status URC is received from the modem

NOTE

These events are defined in `vos_ddi_app_evt.h`. Applications registering to CE should include this header file if it wants to process GSM application events.

Network Control Panel (NCP)

Verix eVo 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 Verix eVo components.

The Network Control Panel consists of multiple functional modules collectively known as the Services Layer. This layer interacts directly with the components in Verix eVo. The user interface modules that will be present in the Verix eVo are:

- Configuration, Status & Management
- Software Downloads
- Network Diagnostics and Logging
- Device Drivers Configuration

The following diagram represents how Network Control Panel integrates Verix eVo components to implement the functionality listed above.

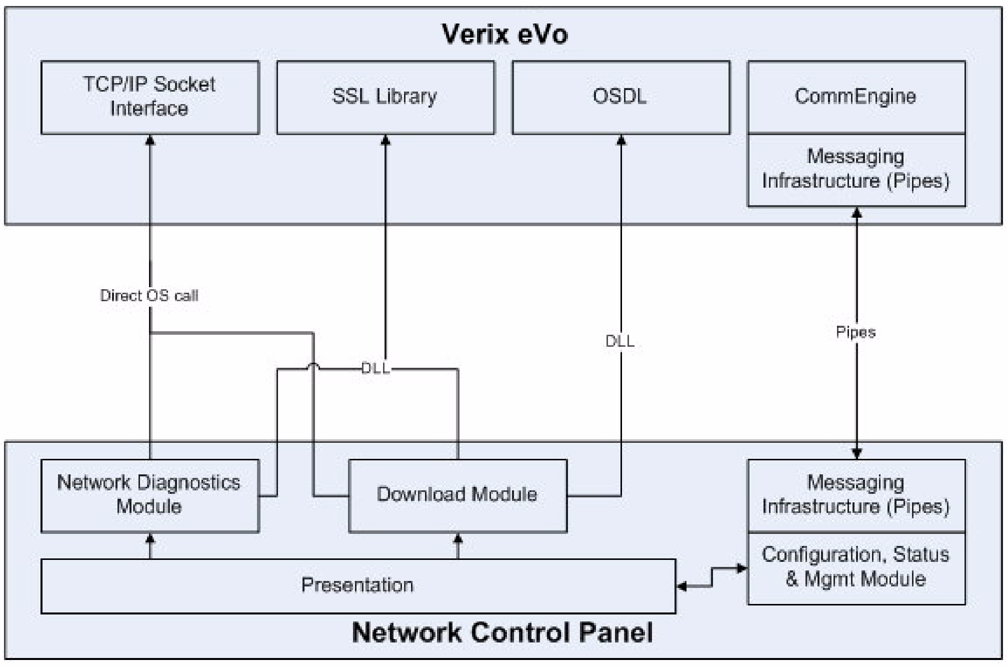


Figure 5 Network Control Panel

Startup Operation

By default Verix eVo will start NCP executable. 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 Verix eVo, clearing the whole application space and CONFIG.sys variables only erases the custom NCP; upon restart Verix eVo 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 Verix eVo components. Alternative solutions may require implementing different features and functionality for specific markets; with this in mind Verix eVo NCP execution may be turned off.

Following sections summarize how Verix eVo starts NCP and how NCP will be available under different scenarios.

Starting NCP executable

On startup, OS starts Verix eVo master application. This application is responsible for starting all other Verix eVo components, including NCP.

For those implementations where NCP will be replaced or turned off, `CONFIG.sys` variable `*NO.VXNCP` on GID1 should be used.

On startup, `VxeVo.out` will look for configuration variable `*NO.VXNCP=1` on GID1. If `*NO.VXNCP` is set to 1 (one), NCP will not run. If `*NO.VXNCP` is not present or set to a different value, NCP will run normally.

On Startup, NCP will create all communication channels to operate with other eVo components.

- Registers with CommEngine via ceAPI.
- Creates named PIPE “VXNCP” which purpose is detailed below in Invoking and Exiting NCP
- Registration with VMAC is outside NCP responsibility and will be handled by VMACIF, as described in Running under VMAC Environment.

Once NCP has initialized and completed registration/connection with other components, it waits until it gets activated.

Interoperability

The following sections describe how NCP communicates and interacts with other components on the system.

CommEngine

On Startup, NCP will:

- Register with CommEngine via ceAPI.
- Create named PIPE “VXNCP” which purpose is discussed under Invoking and Exiting NCP.

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

Console ownership

As described in Invoking and Exiting NCP, 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` on IDLE, hotkey functionality will be restored by calling OS API `enable_hot_key()`.

Printer ownership

Normal NCP operations only require the CONSOLE while some menu options, the user has the option 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 to open 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 will NCP retry obtaining the printer nor retry the print operation. Users 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.

Invoking and Exiting NCP

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".

Returning to Invoking Application

When user selects `Exit` on Idle Screen, NCP will return CONSOLE ownership to the original invoking application. Task ID of the originator is received during activation process via named PIPE.

Control Flow between invoking application and NCP

Figure 6 depicts the flow between the Invoking Application and NCP in both directions such as when NCP acquires control and when it relinquishes control. It follows this sequence of steps:

1. Invoking Application calls `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

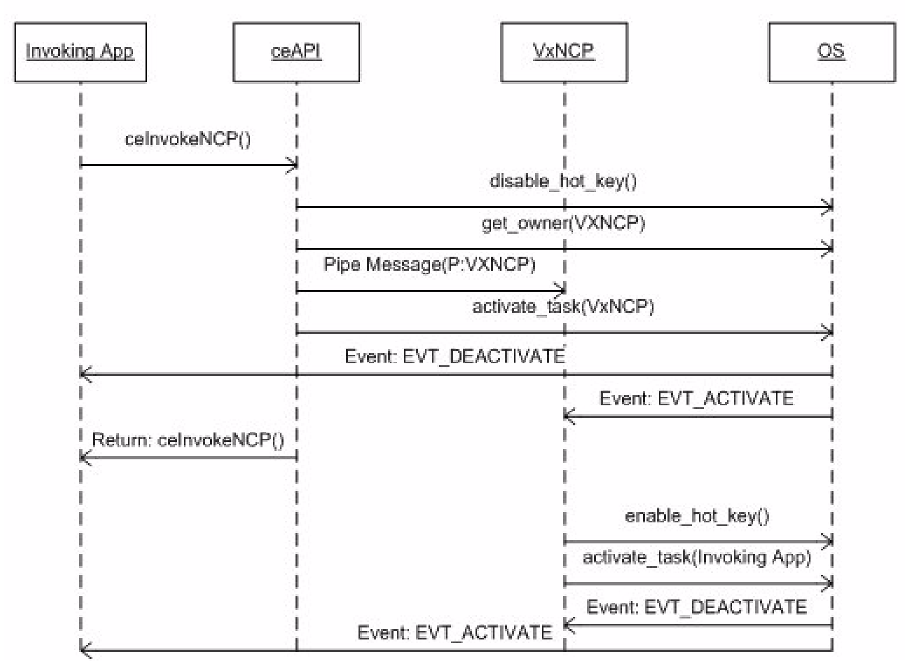


Figure 6 Control Flow – Invoking and Exiting VxNCP

IP downloads from System Mode

An empty terminal needs the ability to download applications. OS will continue to handle downloads via dial, while Verix eVo becomes the engine responsible to handle IP downloads.

When user selects TCP/IP downloads in System Mode, OS will run `VxeVo.out` indicating the special startup sequence and will also run the application configured via `*ZTCP` on GID1. When NCP starts, it will verify if `*ZTCP` is not configured to continue operation; if it is configured to any value NCP exits without doing anything.

OS communicates two parameters when invoking the application to handle IP downloads; Verix eVo 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 passes arguments to run in download mode. Similar to Download, NCP will display the screens to confirm/enter the download parameters required by `SVC_ZONTALK_NET *ZA, *ZT, *ZN and *ZSSL`.

The following diagram depicts flow to handle IP downloads from System Mode.

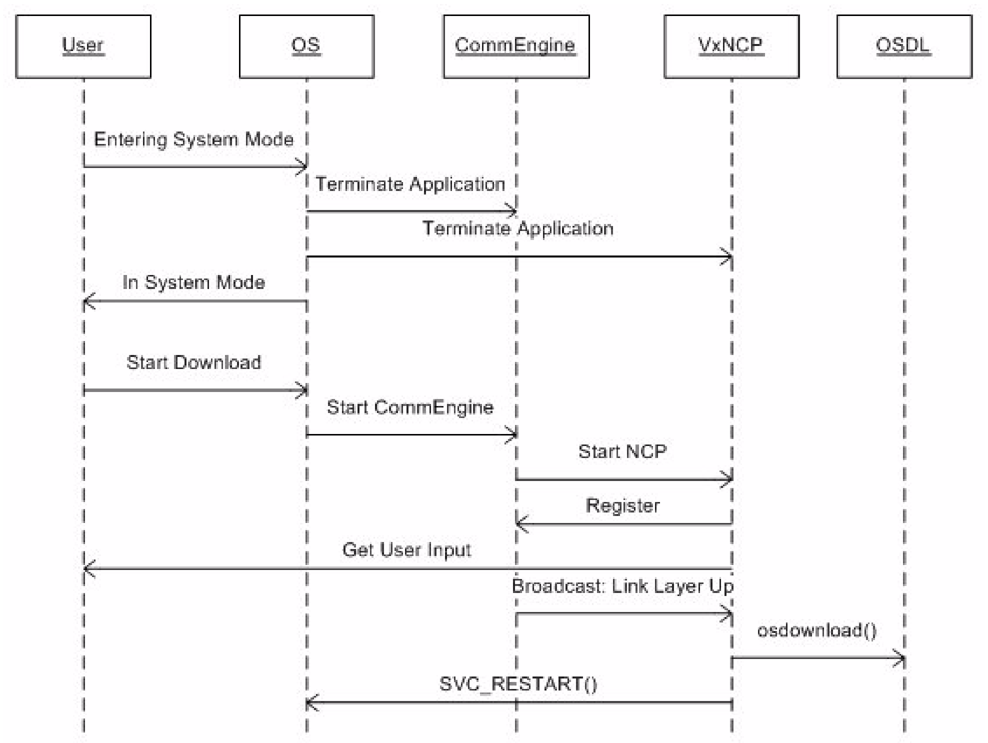


Figure 7 IP downloads from System Mode

Running under Single-Application Mode

No specific action is required. By default, NCP will run 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 that is 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 Verix eVo solutions, surrogated application will not be installed as part of the default Verix eVo bundle. Instead it will be installed during deployment and customization for specific solutions.

Configuration Files

NCP will be released with *factory* default settings via .INI file on GID46. Customers will be able to overwrite some or all settings via .INI file on GID1. File locations will be as follow:

Table 19 NCP Configuration Files

Location	Filename	Description
N:46	VXNCP.INI	Default configuration file. This file contains the name and value of configurable NCP parameters.
I:1	VXNCP.INI	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.INI`. Changes via NCP's UI will be reflected on this file.

NCP's configuration file will follow the standard .INI file format. Following is the current *factory* file.

```
[ Diagnostic-PING ]
  URL = Google.com

[ Diagnostic-DNS ]
  URL = VeriFone.com

[ Diagnostic-TCP ]
URL  = time-a.nist.gov
PORT = 13
  send_data = 0            ; boolean value
  receive_data = 1         ; boolean value
  send_size = 100          ; packet size to send
  delay_after_open = 0     ; delays in miliseconds, calls to
                           SVC_WAIT
  delay_before_close = 0   ; delays in miliseconds, calls to
                           SVC_WAIT
  send_timeout = 0         ; timeout in miliseconds
  receive_timeout = 0      ; timeout in miliseconds
```

```
[ UI ]
  Prompts  = N:prompts.dat

  ; Use absolute paths for FONT files, meaning GID must be
    explicitly noted
  ; UI navigation may require switching GID and fonts must be
    accessible
  Font6x8  = default
  Font8x16 = N:46/asc4x16.vft

  ; enable/disable arrows to navigate menus
  arrows4menus = 1

  ; Timeout values in miliseconds
  timeout_screens= 90000
  timeout_menus= 45000
  timeout_input= 30000
  timeout_confirm= 15000

  keyscan_up= 97            ; Default menu browsing UP key
  keyscan_down= 98          ; Default menu browsing DOWN key

  Vx700_key_up   = 109      ; Different keyscan codes for Vx700
  Vx700_key_down = 108      ; No function/ATM keys available

  ; DATE format on the display
  ; Example for Jan 11th 2009
  ; date separator on the 1st row
  ; date format    on the 1st column
  ;
  ;                 /           -           .
  ;   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/2009 11-Jan-2009 11.Jan.2009
  date_format    = mmmddyyyy
  date_separator = /
```

```
 ; TIME format on the display
  ;   12                   01:00 PM
  ;   24                   13:00
  time_format = 12

[ DefaultNWIF ]
  ; Priority list to set Default NWIF on IDLE.
  ; First NWIF available becomes the default
  ; Key values will be ignored.
    Ethernet=
    GPRS=
    Dial/PPP=

[ DownloadOptions ]
 ; Read-only section. Cannot be modified via GID1
  options = FPpRr

[ DownloadPrompts ]
upperF = Full
upperP = Partial - Restart
lowerP = Partial - Resume
upperR = Erase - Download
```

User Interface

The default language for all UI prompts will be English. To facilitate portability and use on different markets, the following considerations will be taken 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.

Prompts file is generated using TXOFILE standard utility.

Customization

Verix eVo installation will include its own fonts to display the default prompts in English.

Prompts

```
[ UI ]
  Prompts   = F:1/MyPrompts.dat
```

Section	UI
Key Name	Prompts
Description	Custom prompts file

Display Font

```
[ UI ]
  Font6x8   = default
  Font8x16 = N:46/asc4x16.vft
```

Section	UI
Key Name	Font6x8
Description	Custom font file for small (6x8) strings

Section	UI
Key Name	Font8x16
Description	Custom font file for medium (8x16) strings

Time Format

Time will be configurable where the only valid strings are noted below:

`Valid options`	`Examples`
"12"	01:00 p.m.
"24"	13:00

```
[ UI ]
  time format = 12
```

Section	UI
Key Name	Time format
Description	Default time format for display and printed information

Date Format

Date will be configurable. `Date_format` specifies order to show Day-Month-Year while `date_separator` is used as separator character.

Valid values for `date_separator` are '/' (forward slash), '.' (dot) or '-' (dash). If key is not defined or its value does not match any of the valid strings the default separator '/' will be used.

Valid values for `Date_format` 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.

`Date_format` Valid options	**Example for Jan 11th 2009 based on `date_separator` 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 `Month_names` allows defining the abbreviated string name to use as 'month' for the formats `MMMDDYYYY` & `DDMMMYYYY`. This variable provides twelve (12) strings, each one up to three (3) characters long, separated by comma.

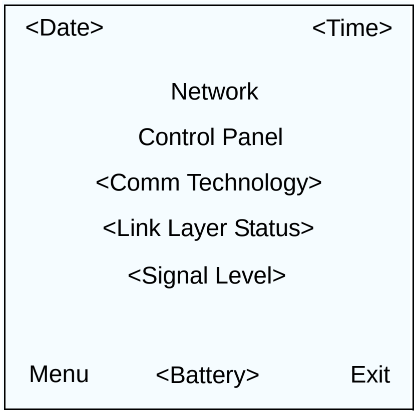
`Month_names` allows replacing the month names without having to download the whole prompts file. The following sample shows how to use this variable to download month names in Spanish

```
Month_names=Ene,Feb,Mar,Abr,May,Jun,Jul,Ago,Sep,Oct,Nov,Dic
```

`Month_names` variable is only verified if `Date_format` is configured to `MMMDDYYYY` or `DDMMMYYYY`. If `Month_names` 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 is displayed as result of activating NCP. Menu options are organized following latest OS menu distribution.



- Date & Time
- `NW Control Panel` as product name
- Primary Communication Technology
- Link Layer Connection Status
- Signal level on wireless devices
- Battery status on portable devices
- Menu to bring up UI options
- Exit to return the CONSOLE to the application that activated NCP

Once any menu option is selected, all sub-menus will be text-only; browsing menu options is available via keys e & f. Alternatively keys 2 and 8 may be used (similar to cell phone).

NOTE

For devices without function keys (i.e. Vx 700) alternate keys are used and factory default will redefine each on a case to case basis.

Menu: Tools

This menu groups all menu options requiring operations, either locally to the network interface(s) available or externally over network interface(s) services.

Diagnostics

For all following menu options, IP addresses for those operations are previously configured.

1 All

When selected, it runs all tests listed below automatically.

- Ping Servers will use the Default NWIF
- 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.

2 Ping Servers

When selected, it prompts the user to select the NWIF. If selected `All` it will run the PING Servers for all servers for all NWIF.

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.

Ping Primary DNS Server	
Input	Primary DNS Server's IP address from NWIF selected
Action	PING Primary 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.

Ping Secondary DNS Server	
Input	Secondary DNS Server's IP address from NWIF selected
Action	PING Secondary 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.

3 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

4 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.

5 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.

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:

1 Network Restart

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.

2 Network Start

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.

3 Network Stop

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.

Download

Currently VxDL is the default application for Software Downloads for most of the latest products. Moving forward to Verix eVo, 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, Partial and Restart, or Partial` download
- How to reach VeriCentre Server
  - IP address
  - PORT number
- Terminal ID
- Application ID

- Selecting `TCP` vs `SSL`

Summary screen will summarize all information for confirmation or editing before starting the download.

Once user confirms data entered, similar to the process described in IP downloads from System Mode download starts.

NOTE NCP mimics VxDL features for secured (SSL) downloads. No SSL Server or Client certificates are supported.

Menu: Terminal Info (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 in the Printer ownership section, if PRINTER device is available, on any page, the user will also have the option to print all pages.

IP Status (IP addresses status)

This menu option will be listed separately from the menu option Comm Technology (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

Comm Technology (Communication Technology)

CE allows every technology to be automatically or manually started on power up. NCP will retrieve the current setting via CEIF and will allow the user to change its value.

Based on the Communication Technologies available on the device, the following screens will be generated

NOTE The lists below represent the minimum data to be provided by the DDI driver for display. The DDI driver may provide additional information not listed on this document which may be displayed by NCP.

i. Ethernet Status

- Device Name
- Device Driver name
- Connection Status
- Link Speed
- MAC address
- Driver Build Date
- Driver Version

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
- Driver Version
- Driver Build Date
- Link Status
- Min Battery

v. PPP over GSM Status

- Device Name
- Device Handler
- Phone Number
- Username

- Password
- Connection Status

vi. PPP over Dial Status

- Device Name
- Device Handler
- Phone Number
- Username
- Password
- Connection Status
- Device Driver Name
- Driver Version
- Driver Build Date
- Fast Connect
- SDLC

Versions

- VxNCP version
- OS version and HW information
- SDK version (static library)
- CommEngine and CEIF versions
- Device Driver(s) Software (DDI) version
- TCP/IP Driver size and timestamp
- SSL Library size and timestamp
- eVo LOG version (static library)
- eVo package version

Menu: Setup

This menu option is password protected. User must type the System Password for GID1.

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) .

Communication Technology on Idle

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

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` will not be openly displayed. The masked string represents its current value. If the user selects the option to modify it, the newly entered values will be visible until the user presses `Enter`, masking the new current value on the display.

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

- 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
    - PPP AUTH TYPE

v. PPP over GSM

- 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. PPP over Dial

- 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
    - PPP AUTH TYPE

NOTE

No Network Interface for “Land-Line” or “GSM” (without PPP) are supported by VxNCP.

Default settings may be defined via configuration files. Run-time edition of those files will be available via NCP.

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.

Device Driver (s)

NCP as default UI for all Verix eVo 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 `.INI` and `.MTD` configuration files.

The 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 `.INI` file

Menu: Exit

This menu option is available on IDLE screen. It will automatically return the CONSOLE device to the application that activated NCP.

Menu option will not be present when NCP is started from System Mode after user selects TCP/IP downloads.

VeriFone, Inc.
2099 Gateway Place, Suite 600

San Jose, CA, 95110 USA.

1-800-VeriFone

www.verifone.com

Verix eVo Volume II: Operating System and Communication

Programmers Manual

VeriFone Part Number DOC00302, Revision A