[redacted]

V/OS

Programmers Manual

[redacted] Part Number DOC00501, Revision G

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

CONTENTS

PREFACE

This programmer's manual describes the V/OS programming environment and functions unique to V/OS-based terminals. This manual:

- Provides descriptions of the CONFIG.SYS variables
- Provides API descriptions, function calls, and code examples
- Discusses system and communication devices
- Describes security features
- Describes working with the IPP (internal PINpad)
- Presents terminal-specific content in Appendix B, V/OS on the MX Terminal, Appendix C–Appendix E, V/OS on the UX Terminals, and Appendix G, V/OS on the VX Terminal.

The V/OS-based terminals are designed to support many types of applications, especially in the point-of-sale (POS) environment. This manual is designed to help programmers develop applications. This manual also contains explicit information regarding the Application Programming Interface (API) with the V/OS, and with optional peripheral or internal devices. Distinction is drawn between functions built into the terminal that are available when no application is present, and functions only available when an application performs certain operations.

This manual also contains explicit information regarding the API with the V/OS operating system, and with optional peripheral or internal devices.

Organization

This manual is organized as follows:

Chapter 1	Provides a quick reference to function calls and error codes for V/OS.
Chapter 2	Presents a basic overview of system architecture and the boot up process.
Chapter 3	Provides information on Secure Installer, including application download instructions and function calls.
Chapter 4	Provides a basic overview of terminal system mode. Refer to the reference manual for your terminal for complete menu descriptions.
Chapter 5	Is an overview of the multi-application environment, including setting up device managers.
Chapter 6	Discusses power management features for V/OS terminals.
Chapter 7	Discusses using service calls to share access to specific device resources.
Chapter 8	Presents information on how the Input Event module manages touch panel and USB HID events.

Chapter 9	Provides information on using device drivers for supported devices.
Chapter 10	Presents the function calls used to manage the V/OS terminal display.
Chapter 11	Presents the function calls used to manage the V/OS terminal LEDs.
Chapter 12	Describes the V/OS CardSlot Library used to build smartcard applications for V/OS terminals.
Chapter 13	Presents the function calls used to manage the V/OS terminal touch panel.
Chapter 14	Presents the beeper APIs for V/OS terminals.
Chapter 15	Discusses the serial port protocols used to manage ports on V/OS terminals.
Chapter 16	Presents the function calls used to manage the V/OS terminal internal PINpad.
Chapter 17	Discusses the V/OS Account Data Encryption (ADE) module and presents ADE function calls.
Chapter 18	Presents the function calls used to manage the V/OS terminal magnetic stripe reader.
Chapter 19	Presents the function calls used to manage the V/OS terminal contactless card reader.
Chapter 20	Presents the function calls used to manage the V/OS terminal USB device.
Chapter 21	Discusses using NET service calls to manage Ethernet and Wi-Fi devices.
Chapter 22	Provides information on client/server network programming.
Chapter 23	Presents the function calls used to manage the V/OS terminal Bluetooth interface.
Chapter 24	Provides information on the V/OS terminal security module.
Chapter 25	Discusses RTC functionality.
Chapter 26	Provides information on secure password and user management.
Appendix A	Discusses the VCL software component.
Appendix B	Provides MX terminal-specific information.
Appendix C	Provides general UX terminal information.
Appendix D	Provides UX100 terminal-specific information.
Appendix E	Provides UX300 terminal-specific information.
Appendix F	Provides information on Sysmode for UX300 and VX530 terminals.
Appendix G	Provides VX terminal-specific information.
Appendix H	Provides information on PAYware Vision compatibility.
Appendix I	Provides information on compatibility with legacy IPP libraries.

Audience

This document is of interest to application developers creating applications for use on V/OS-based terminals.

Assumptions About the Reader

It is assumed that the reader:

- Understands Linux programming concepts and terminology.
- Has access to a PC running Windows XP or Windows 7.
- Has installed the Mentor Sourcery CodeBench DTK and Verifone SDK on this machine.
- Has access to a V/OS terminal.

Conventions and Acronyms

Abbreviation	Definition
BFI	Buffer Flush Interval
bps	bits per second
CRC	Cyclic Redundancy Check
DHCP	Dynamic Host Configuration Protocol
FA	File Authentication
Firmware	Software in FLASH/ROM
FTP	File Transfer Protocol
GISKE	Global Interoperable Secure Key Exchange
GPRS	General Packet Radio Service
HID	Human Interface Device
IPP	Internal PIN Pad
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISDN	Integrated Services Digital Network
ISR	Interrupt Service Routine
KLK	VSS Key Loading Key
KSN	Key Serial Number
KVC	Key Verification Code
LED	Light Emitting Diode
MS	Master Session
MSR	Magnetic Stripe Reader
NFS	Network File System
OSS	Open Sound System
PED	PIN Entry Device
PEK	PIN Encryption Key
PPP	Point-to-Point Protocol
PSTN	Public Switched Telephone Network
RFCR	RF Card Reader
RRT	Receive Record Threshold
RTC	Real-time Clock

Abbreviation	Definition
SAM	Security Access Module
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol (TCP) and Internet Protocol (IP)
V/OS	Verifone Operating System
VRK	VeriShield Remote Key
VSS	VeriShield Security Scripts
Wi-Fi	Wireless Fidelity
XML	Extensible Markup Language

Major Hardware Devices

- ARM11
- ARM SoC
- Graphic Display

Recommended Documentation

Refer to the following set of documents for more information:

- *Beginning Linux Programming* (4th Edition) by Neil Matthew and Richard Stones
- *V/OS CardSlot Library Programmers Guide,* VPN - DOC00510

Wireless Interface Applications - Required Actions and Best Practices

When writing applications for wireless interfaces, beware the risk of attack. Use MasterCard Worldwide specifications to ensure secure connections. Review this document at
http://www.mastercard.com/ us/sdp/assets/pdf/wl_entire_manual.pdf

The following table lists applicable wireless interfaces with pages referenced in the MasterCard Worldwide manual that details security concerns and accepted operational modes necessary to mitigate them.

Wireless Interface	Basic Information	Security Risks	Security Guidelines
Wi-Fi	2-2	2-2 through 2-3	2-4 through 2-7
GPRS and GSM	2-12	2-12	2-13
Bluetooth	2-14	2-14 through 2-15	2-16

NOTE Visa, Payment Card Industry (PCI), and/or other important entities may offer documents detailing requirements as well. Application writers should inquire with any such pertinent entities for documentation before beginning application development.

Guidelines in Using Wireless Interfaces

The following are guidelines for application developers in using wireless interfaces:

WiFi

Do not use WEP if payment application uses WiFi.

Bluetooth

Use S3 and SSP if payment application uses Bluetooth. Payment applications must never use “Just Works” mode under any circumstance. Similarly, devices must never use “Just Works” association model, and therefore, must immediately discard all unauthenticated Just Works link keys after pairing to terminate such connections.

Guidance on Digital Certificates with 1024-bit Keys (Including SSL Certificates)

In accordance with guidance from the National Institute of Standards and Technology (NIST), Certificate Authorities (CAs) are advised to follow the recommendations published initially in advisory 800-57 and later 800-131A. CA's are advised to deprecate signing Digital Certificates that contained RSA Public Keys of 1024 bits after 31st December 2010 and cease signing completely by 31st December 2013 (Table 2, Section 3 of 800-131A). At the time, the general consensus was all previously issued, long-lived certificates expiring after the 31st December 2013 should be dealt with nearer to the deadline.

As a forward thinking Certificate Authority with an SSL mission to improve how CAs deploy SSL and end users rely on SSL, GlobalSign helped its customers stay protected and benefit from the highest levels of security available, by mandating a stronger security level than the NIST guidance and therefore ahead of the industry norm. From 1st January 2011, GlobalSign introduced RSA key size requirements to no longer accept 1024 bit Certificate Signing Requests (CSRs). This thinking was aligned with the decision, back in 1998, to create a 2048 bit Root Certificate and therefore a full 2048 bit hierarchy of services including issuing CAs, CRLs and OCSP responders.

NOTE

Refer to Security Architect Document Annex Communication Guide for more information.

CHAPTER 1

Quick Reference

This section provides programmers quick access to system function calls, CONFIG.SYS variables, device variables, error codes, download procedures, instructions on erasing flash, keypad key return values, and printer control codes for the V/OS.

Function Calls

The functions listed in Table 1 are arranged by device or purpose. Refer to the function description for associated parameters, valid error conditions, and details on how and when to use the function. In the online version of this manual, click the page number to jump to the function description.

Table 1 Function Calls

Function Call	Description	Page
Secure Installer		
Secins_install_pkgs()	Installs packages from the download file.	87
Secins_reboot()	Reboot the device.	88
Secins_start_user_apps()	Used by sysmode to start user applications.	89
Secins_start_app()	Used by user applications to start other user applications.	90
Secins_read_pkglist_entry()	Retrieves a list of installed packages.	91
Secins_close_pkglist()	Closes the list of installed packages.	93
Secins_strerror()	Returns a pointer to the error message associated with the err non-zero error code.	94
Secins_share_gid()	Returns the group ID (GID) for the share group.	95
Secins_system_gid()	Returns the GID for the system group.	96
Secins_users_gid()	Returns the GID for the users group.	97
Secins_user_app()	Returns the application user usr1–16.	98
Secins_sys_app()	Returns the system user sys1–sys16.	99
Secins_sysmode_share_gid()	Returns the sysmode share GID for the calling user.	100
Secins_config_file_share_gid()	Returns the sysmode share GID for the specified config file.	101
User/Groups		
Secins_get_uid_gid_range()	Retrieves the range of users and groups assigned to the calling primary user.	103
Secins_add_group()	Adds a group.	104
Secins_add_user()	Adds a user.	105

Table 1 Function Calls (continued)

Function Call	Description	Page
Secins_add_group_member()	Adds the specified UID to the specified GID.	106
Secins_remove_user()	Removes the specified UID.	107
Secins_remove_group()	Removes the specified GID.	108
Secins_start_app_uid()	Starts an application with specified UID and GID.	109
Power Management		
powermngt_getVersion()	Returns the POWER_MGMT service version.	122
powermngt_get_config()	Returns the power management (PM) configuration.	123
powermngt_set_config()	Sets the PM configuration.	125
powermngt_Shutdown()	Forces shut down.	127
powermngt_SetWakeupTime()	Sets RTC alarm to wake up the unit.	128
powermngt_CritcalSectionEnter()	Enters the PM critical section.	129
powermngt_CritcalSectionExit()	Exits the PM critical section.	130
powermngt_GetInfo()	Retrieves PM information.	132
powermngt_GetEvent()	Registers to PM events.	133
powermngt_Susspend()	Suspends PM events.	134
powermngt_Standby()	Places PM events in standby mode.	135
powermngt_get_wakeupdevices()	Determines which device wakes the system.	136
powermngt_set_wakeupdevices()	Sets which device wakes the system.	137
Input Event		
inputOpen()	Opens the USB device.	210
inputRead()	Captures read events.	211
inputClose()	Closes the device.	212
Display		
dspSetBrightness()	Adjusts display brightness.	216
LED		
Led_ledOn()	Enables the LEDs.	218
led_ledOff()	Disables one or more LEDs.	219
led_init()	Initializes the LED file descriptor effect.	220
led_release()	Closes the LED file descriptor.	221
led_start_blinking()	Starts LED blinking.	222
led_stop_blinking2()	Stops LED blinking.	223
led_read_led()	Reads the current LED status.	224
led_read_led_ctls()	Reads the current CTLS LED status.	225
led_switch_color()	Specifies the color of a specific LED.	226
Touch Panel		
SigCap2Tiff()	Creates a TIFF format file of the signature capture.	231

Table 1 Function Calls (continued)

Function Call	Description	Page
Internal PINPad		
ippOpen()	Takes ownership of the IPP and clears the internal IPP FIFO.	242
ippClose()	Releases ownership of the IPP.	243
ippRead()	Transfers data from the IPP FIFO to the application data buffer.	244
ippWrite()	Transfers a single IPP packet into the IPP.	245
SetSecurePINDisplayParameters()	Sets the hot spot table and registers the callback function.	246
ippPinEntryStatus()	Returns the PIN entry status.	248
ippTerminatePinEntry()	Ends the PIN session.	249
Account Data Encryption (ADE)		
ade_encrypt()	Encrypts the input data.	325
ade_status()	Returns the status of the ADE module.	328
ade_active()	Indicates if the ADE system is fully functional.	329
Magnetic Stripe Reader		
msrOpen()	Prepares the firmware to accept and store card reads.	332
msrRead()	Reads the decoded and formatted MSR data.	333
msrWrite()	Transfers data from an application buffer into the device driver’s buffer.	335
msrMagneticCardPresent()	Returns the current status of the MSR data.	336
msrRaw()	Allows an application to retrieve the raw magnetic stripe data.	337
msrStructured()	Allows an application to retrieve the decoded magnetic stripe data.	338
msrEnableLicenseDecode()	Enables the decoding of Drivers License magnetic stripe data.	339
msrDisableLicenseDecode()	Disables Drivers License data decoding.	340
msrSetMinBytes()	Sets the minimum number of track data bytes each track must have before decoding.	341
msrSetMaxNoDataCount()	Sets the number of consecutive times no data in all three tracks are seen before reporting as EMF induced.	342
msrReportExtendedErrors()	Enables negative error returns on errors.	343
msrVersion()	Reads the MSR library version.	344
Contactless RF Card Reader (RFCR)		
CTLSClientGetVersion ()	Retrieves the RFCR library version.	350
CTLSInitInterface()	Performs device initialization and initializes the serial port.	350
CTLSOpen()	Opens the serial port.	350

Table 1 Function Calls (continued)

Function Call	Description	Page
CTLSGetUI()	Retrieves the current UI handler function pointers.	351
CTLSSetUI()	Assigns the handler functions.	352
CTLSSend()	Sends a message to the CTLS application.	353
CTLSReceive()	Receive a message from the CTLS application.	354
CTLSClose()	Closes the serial port.	355
NET Service		
net_getVersion()	Retrieves the version of the NET service.	397
net_interfaceSet()	Writes the IP interface configuration to netconf.xml.	398
net_interfaceGet()	Reads the IP interface configuration from netconf.xml.	399
net_interfaceUp()	Brings the interface up.	410
net_interfaceDown()	Closes the interface.	411
net_interfaceStatus()	Reads the interface link status.	415
net_networkUp()	Brings the network up.	413
net_networkDown()	Closes the network.	414
net_interfaceStatusList()	Reads the active network interface configuration list.	415
net_getTable()	Reads the routing table.	416
net_getRoutes()	Reads all routes to the specified destination.	417
net_addHostRoute()	Adds a route to the specified host.	418
net_delHostRoute()	Deletes a route to the specified host.	419
net_addNetRoute()	Adds a route to the specified network.	420
net_delNetRoute()	Deletes a route to the specified network.	421
net_autoSetDefaultRoute()	Selects the default route.	422
net_setDefaultRoute()	Sets the default route.	423
net_getDefaultRouteInfo()	Retrieves the default route interface name.	424
net_startDHCP()	Starts a DHCP client.	425
net_renewDHCP()	Causes `udhcpc` to renew the current lease.	426
net_releaseDHCP()	Causes `udhcpc` to release the current lease.	427
net_stopDHCP()	Stops the DHCP client.	428
net_setDNS()	Sets the DNS.	429
net_addRouteFromXML()	Adds static routes from the specified XML file.	430
net_nsLookup()	Performs DNS resolution.	431
net_ping()	Removes all stored DNS resolution results.	432
net_wifiStart()	Starts the Wi-Fi module and WPA supplicant.	433
net_wifiStop()	Stops the Wi-Fi module and WPA supplicant.	434

NOTE

- MX terminal-specific information is in Appendix B.
- UX terminal-specific information is in Appendix C through Appendix F.
- VX terminal-specific information is in Appendix G.

Table 1 Function Calls (continued)

Function Call	Description	Page
net_wifiSiteSurvey()	Obtains a Wi-Fi site survey list.	435
net_wifiInfo()	Retrieves Wi-Fi information.	436
net_setNTP()	Sets the NTP host name.	437
TCPIP Ethernet		
net_interfaceUp()	Activates the network interface.	473
net_interfaceDown()	Deactivates the network interface.	474
net_interfaceStatus()	Reads the interface link status.	475
net_networkUp()	Brings up all network interfaces.	476
net_networkDown()	Closes all network IP interfaces.	477
net_interfaceStatusList()	Reads the list active network interface configurations.	478
net_getTable()	Reads the routing table.	479
net_getRoutes()	Reads all routes to a specific destination.	480
net_addHostRoute()	Adds a route to the specified host.	481
net_delHostRoute()	Deletes a route to the specified host.	482
net_addNetRoute()	Adds a route to the specified network.	483
net_delNetRoute()	Deletes a route to the specified network.	484
net_autoSetDefaultRoute()	Adds a route to the specified host.	485
net_setDefaultRoute()	Selects the default route.	486
net_getDefaultRouteInfo()	Gets the default route network.	487
net_startDHCP()	Starts a DHCP client.	488
net_renewDHCP()	Causes `udhcpc` to renew the current lease.	489
net_releaseDHCP()	Causes `udhcpc` to release the current lease.	490
net_stopDHCP()	Stops the DHCP client.	491
net_setDNS()	Sets the DNS.	492
net_addRouteFromXML()	Adds static routes from the specified XML file.	493
net_nsLookup()	Performs DNS resolution.	494
net_setNTP ()	Removes all stored DNS resolution results.	495
net_wifiStart()	Starts the Wi-Fi module and WPA supplicant.	498
net_wifiStop()	Stops the Wi-Fi module and WPA supplicant.	499
net_wifiSiteSurvey()	Obtains a Wi-Fi site survey list.	500
net_wifiInfo()	Retrieves Wi-Fi information.	501
net_ping()	Sets the NTP host name.	503
ftpGet()	Transfers the file from the FTP server to the terminal.	504
ftpPut()	Transfers a file from the terminal to the FTP server.	505
Bluetooth		
bluetooth_getVersion()	Retrieves the version of the Bluetooth device firmware.	508
btConfig::address[]	Configures the Bluetooth device address.	515

Table 1 Function Calls (continued)

Function Call	Description	Page
btConf::discoverable	Retrieves the Bluetooth discovery call.	516
btConf::name[]	Sets the Bluetooth device name.	517
btDevice::address[]	Retrieves the Bluetooth device address.	519
btDevice::class[]	Retrieves the Bluetooth device class.	520
btDevice::name[]	Retrieves the Bluetooth device name.	521
btDevice* btDeviceList::device	Return a list of scanned Bluetooth devices.	522
btDeviceList::deviceCount	Retrieves the number of devices found.	523
bluetooth_getVersion()	Retrieves the Bluetooth service version.	524
btConf bluetooth getConfig()	Retrieves the current Bluetooth configuration.	529
btDeviceList bluetooth getConnectedHids()	Retrieves list of connected HIDs.	528
version bluetooth getVersion()	Retrieves the Bluetooth service version.	525
bluetooth hidConnect()	Connects to an HID device.	526
bluetooth hidDisconnect()	Disconnects from an HID device.	527
bluetooth_getConnectedHids()	Retrieves a list of connected HIDs.	513
bluetooth isUp()	Retrieves Bluetooth power status.	530
bluetooth power()	Retrieves current Bluetooth configuration.	531
bluetooth scan()	Scans for Bluetooth devices.	532
btDeviceList bluetooth scanResults()	Retrieves the last scan results.	533
Security		
iPS_DeleteKeys()	Deletes keys.	537
iPS_LoadSysClearKey()	Loads the system keys.	538
iPS_LoadSysEncKey()	Loads the system encryption keys.	539
iPS_LoadMasterClearKey()	Loads the master key of the security script.	540
iPS_LoadMasterEncKey()	Loads the security script's master keys.	541
iPS_CheckMasterKey()	Returns the key present in the specified location.	542
SetSecurePINDisplayParameters()	Registers the PIN entry callback function.	543
iPS_SetPINParameter()	Configures VSS PIN session parameters.	544
iPS_SelectPINAlgo()	Selects the PIN algorithm.	546
iPS_RequestPINEntry()	Initiates PIN collection.	547
iPS_GetPINResponse()	Checks PIN session status.	548
iPS_CancelPIN()	Cancels PIN processing.	551
iPS_InstallScript()	Installs a VSS file.	552
iPS_GetScriptStatus()	Checks if a VSS file is installed.	553
iPS_UninstallScript()	Uninstalls a VSS file.	554
iPS_ExecuteScript()	Spawns the execution of a given macro from the VSS.	555
pcPS_GetVSSVersion()	Returns the version of the VSS interpreter.	556
cryptoWrite()	Encrypts and writes data from the buffer to a file.	558
cryptoRead()	Reads encrypted data from a file.	559
rsa_calc()	Performs a public key RSA computation.	560

Table 1 Function Calls (continued)

Function Call	Description	Page
vfi_SHA1()	Performs a SHA-1 computation.	561
DES()	Performs DES, DESX and Triple-DES computations.	562
AES()	Performs AES computations.	563
generateRandom()	Returns a random 8-byte value.	564
isAttacked()	Indicates if an attack occurred.	565
secVersion()	Returns the version number.	566
authFile()	Checks file authentication.	567
copyPublickey_RKL()	Copies the VRK public key certificate to the /tmp directory.	586
getKeyStatus_RKL()	Returns the current VRK key/certificate status.	587
Real-Time Clock (RTC)		
setRTC()	Sets the RTC hardware to the Linux RTC time/date.	572
setDateTime()	Sets the Linux RTC.	573
MX Terminal-Specific Functions		
ledRunway()	Enables or disables the 3-LED runway effect.	596
startPktMode()	Initializes Packet mode.	600
endPktMode()	Frees the Packet mode buffers at session completion.	601
packetRX()	Receives packet messages.	602
comMonitorHandshake()	Enables a counter increment on transitions on the CTS and/or DCD lines.	605
touchCmd()	Sets and retrieves touch panel functionality.	610
SigCapCount()	Returns the number of available signature points.	612
SigCapGet()	Copies data from the kernel buffer to the user buffer.	613
SigCapBoxApply()	Applies a signature box to the data.	614
vf_sig_cature_options()	Sets the rendering options.	615
RFCRlibVersion()	Stores the RFCR library version.	617
RFCRInit()	Performs device initialization, and opens and configures the serial port.	618
RFCRClose()	Closes the serial port that communicates with the Contactless RFCR module.	619
RFCRGetVersion()	Returns the firmware version of the RFCR.	620
RFCRPing()	Tests for Contactless RFCR module response.	621
RFCRReset()	Configures the RESET line of the Contactless RFCR module.	622
RFCRSetAntenna()	Configures the antenna control of the Contactless RFCR module.	623
RFCRSetIndicator()	Configures the optional indicator controls of the Contactless RFCR module.	624

Table 1 Function Calls (continued)

Function Call	Description	Page
RFCRGetCardPayload()	Stores the Card Payload packet.	625
RFCRParseCardPayload()	Parses the Card Payload data to the CardPayload structure.	626
RFCRPurge()	Purges any pending input from the reader.	627
RFCRInputPending()	Returns the number of bytes available for reading.	628
RFCRRawWrite()	Sends raw data to the Contactless RFCR module.	629
RFCRRawRead()	Reads raw data from the Contactless RFCR module.	630
RFCRAddCRC()	Calculates the CRC of the data contained in buff and inserts it at the offset position of the buffer.	631
RFCRCheckCRC()	Checks the CRC data in buff.	632
RFCRReceiveACKFrame()	Receives an ACK frame from the Contactless RFCR module.	633
RFCRReceiveDataFrame()	Receives a data frame from the Contactless RFCR module.	634
RFCRSetPollMode()	Sets the RFCR polling mode.	635
RFCRGetTransactionResult()	Returns the card data to buff.	636
RFCRActivateTransaction()	Polls reader for valid card and continues transaction.	637
RFCRActivateMxItransaction()	Polls reader for valid MxI card and continues transaction.	638
RFCRDebitWrite()	Continues to the second leg of an MxI payment transaction to complete the transaction.	639
RFCRCancelTransaction()	Cancels the polling of any approaching card after the Activate Transaction command or Update Balance command is sent to the reader.	640
RFCRGetFullTrackData()	Retrieves full track data from the Contactless RFCR module.	641
RFCRUpdateBalance()	Updates the balance on the card.	642
RFCRSetEMVParameters()	Sets or changes the values of the specified data objects in the reader.	643
RFCRGetEMVParameters()	Retrieves the values of the specified data objects in the reader from nonvolatile memory.	644
RFCRSetBurstMode()	Enables RFCR Burst mode to send a data packet from the RFCR to the terminal on each successful card read.	645
RFCRSetPassThroughMode()	Enables RFCR Pass Through mode to stop the RFCR polling for cards it recognizes.	646
RFCRPollForToken()	Starts polling for Type A or Type B PICC until a PICC is detected or timeout occurs.	647
RFCRIsoApduExchange()	Sends, via the RFCR, application-level APDUs to a PICC that supports the 14443-4 protocol.	648

Table 1 Function Calls (continued)

Function Call	Description	Page
RFCRPcdSingleExchange()	Sends, via the RFCR, raw data to a ISO 14443 PICC that does not support the ISO 14443-4 protocol (such as Mifare).	649
RFCRGetPcdPiccParam()	Retrieves the PCD and PICC related parameters from the RFCR.	650
RFCRMifareAuthenticateBlock()	Authenticates the Mifare card sector containing the specified block of data.	651
RFCRMifareReadBlocks()	Reads data from one or more blocks on the Mifare card and returns the data to the terminal.	652
RFCRMifareWriteBlocks()	Writes data to one or more blocks on the Mifare card.	653
RFCRMifarePurseCommand()	Enacts debit, credit, and backup operations on value blocks on a Mifare card.	654
RFCRHighLevelHaltCommand()	Sends a HALT command to the card, used for any Type A or Type B card.	655
RFCRSetCAPublicKey()	Sends the data related to a CA Public Key to the RFCR.	656
RFCRDeleteCAPublicKey()	Deletes a previously set CA Public Key.	657
RFCRDeleteAllCAPublicKeys()	Deletes all previously set CA Public Keys.	658
RFCRSetRTCTime()	Sets a specific time in the real-time clock.	659
RFCRGetRTCTime()	Returns the current time from the real-time clock.	660
RFCRSetRTCDate()	Sets the real-time clock date.	661
RFCRGetRTCDate()	Returns the current date set in the real-time clock.	662
RFCRSetBaudRate()	Sets the baud rate.	663
RFCRSetRTCSource()	Sets the source for RTC/LCD/Buzzer/LED on the RFCR.	664
RFCRGetRTCSource()	Retrieves the source for the RTC/LCD/Buzzer/LED on the RFCR.	665
RFCRGetType()	Retrieves the RFCR type.	666
RFCRLedControl()	Controls the LEDs when the RFCR is in Pass Through mode.	667
RFCRBuzzerControl()	Enables the RFCR buzzer only when the RFCR is in Pass Through mode.	668
RFCRGetConfigurableAID()	Reads the configurable AID parameters.	669
RFCRGetAllAIDs()	Read all AIDs in the RFCR to verify the configured AIDs in the reader.	670
RFCRGetConfigurableGroup()	Reads the configurable Group EMV parameters.	671
RFCRGetAllGroups()	Reads all groups in the RFCR to verify all configured groups in the reader.	672
RFCRWriteDataCommand()	Stores a ticket on the card.	673
RFCRSetConfigurableAID()	Creates or selects an AID for configuration or deletion.	674
RFCRSetConfigurableGroup()	Creates or selects a group for configuration.	675

Table 1 Function Calls (continued)

Function Call	Description	Page
RFCRSDeleteConfigurableAID()	Deletes or disables a configurable AID.	676
RFCRDeleteConfigurableGroup()	Deletes a configurable group.	677
RFCRGetBuild()	Reads the firmware build ID in the RFCR.	678
RFCRGetAllVariables()	Reads all RFCR variables.	679
RFCRGetProductType()	Reads the RFCR product type.	680
RFCRGetProcessorType()	Reads the RFCR processor type.	681
RFCRGetMainFwVersion()	Reads the RFCR main firmware version.	682
RFCRGetFwSubsystemSuite()	Reads the RFCR firmware subsystem suite.	683
RFCRGetSerialProtocolSuite()	Reads the RFCR serial protocol suite.	684
RFCRGetPayPassVersion()	Reads the RFCR Layer 1 Pay Pass version.	685
RFCRGetAntiCollisionResolution()	Reads the RFCR Layer 1 Anti-Collision Resolution version.	686
RFCRGetCardApplicationSuite()	Reads the RFCR Layer 2 Card Application suite.	687
RFCRGetUserExperienceSuite()	Reads the RFCR User Experience suite.	688
RFCRGetSystemInformationSuite()	Reads the RFCR System Information suite.	689
RFCRGetCAPublicKey()	Reads the CA Public Key from the RFCR.	690
RFCRGetSerialNumber()	Reads the serial number from the RFCR.	691
RFCRSetSerialNumber()	Stores the serial number in the RFCR.	692
RFCRFlushTrackData()	Flushes track data.	693
RFCRSetRfErrorReporting()	Enables or disables RF error reporting for the Get Full Track Data command.	694
RFCRGetUSBBootLoaderVersion()	Retrieves the ViVOpay USB Boot Loader version.	695
RFCRStoreLCDMessage()	Stores the LCD message.	696
RFCRGetLCDMessage()	Sends messages to the terminal.	697
RFCRGetAdditionalAIDParams()	Retrieves additional AID parameters.	698
RFCRGetRevocationParam()	Retrieves all Revocation structure parameters.	699
RFCRGetAllAdditionalAIDParams()	Retrieves all additional AID parameter structures.	700
RFCRGetAllRevocationParams()	Retrieves all Revocation parameter structures.	701
RFCRGetAllExceptionParams()	Retrieves all exception PANs.	702
RFCRGetAdditionalReaderParams()	Retrieves all additional reader parameters.	703
RFCRGetVFIVersion()	Retrieves the software release version.	704
RFCRGetTypeApprovalVersions()	Returns a string containing version information.	705
RFCRSetExistAdditionalAIDParams()	Updates existing additional AID parameters.	706
RFCRSetNewAdditionalAIDParams()	Adds AID parameters.	707
RFCRDeleteAdditionalAIDParams()	Deletes an additional AID parameter structure.	708
RFCRSetExistRevocationParam()	Updates existing Revocation structure parameters.	709
RFCRSetNewRevocationParam()	Adds a Revocation structure parameter.	710
RFCRDeleteRevocationParam()	Deletes a Revocation structure parameter.	711
RFCRSetNewExceptionParam()	Adds a PAN to the exception PANs list.	712
RFCRDeleteExceptionParam()	Deletes an Exception PAN from the exception PANs list.	713

Table 1 Function Calls (continued)

Function Call	Description	Page
RFCRSetAdditionalReaderParam()	Updates existing additional RFCR parameters.	714
RFCRRestoreDefaultConfig()	Restores the default reader configuration.	715
RFCRGetFwFullVersion()	Returns a string containing version information for each software module for individual card types.	716
RFCRGetBaudRate()	Retrieves the current baud rate set in the RFCR.	717
RFCRChangeBaudRate()	Sets or changes the baud rate in the RFCR.	718
UX100/UX110 Terminal-Specific Functions		
ux100_dispGetInfo()		734
ux100_dispSetContrast()		735
ux100_dispSetBacklightMode()		736
ux100_buzzerSetBeep()		738
ux100_buzzerGetInfo()		739
ux100_buzzerSetKeyboardBeep()		740
ux100_buzzerGetKeyboardBeepInfo()		741
ux100_pinStartPinEntry()		743
ux100_pinGetNbCurrDigitEntered()		744
ux100_pinDelete()		745
ux100_pinAbort()		746
ux100_pinSendEncPinBlockToVault()		747
ux100_pinEntrymode()		748
ux100_getVersion()		750
ux100_infoGetDeviceList()		751
ux100_infoGetSecureVersion()		752
ux100_infoGetPartNumber()		753
ux100_infoGetSerialNumber()		754
ux100_infoGetHardwareVersion()		755
ux100_infoGetPCIHardwareVersion()		756
ux100_infoGetModelNumber()		757
ux100_infoGetCountryName()		758
ux100_infoGetPairingStatus()		759
ux100_infoGetRemSwStatus()		760
ux100_infoGetTamperStatus()		761
ux100_infoGetCoinChargingStatus()		762
ux100_infoGetSensorTemperature()		763
ux100_infoGetLogEvents()		764
ux100_infoSetIdCard()		765
ux100_infoGetConnectionStatus()		766
ux100_infoGetDeviceMode()		767
ux100_infoGetOperationalMode()		768
ux100_infoGetPermanentTerminalID()		769

Table 1 **Function Calls** (continued)

Function Call	Description	Page
ux100_infoGetUnitType()		770
ux100_infoResetLogHeader()		771
ux100_infoSetLogEvent()		772
ux100_infoGetAtmelSerialNumber()		773
ux100_infoSetRTC()		774
ux100_infoGetRTC()		775
ux100_infoGetHeaterMode()		776
ux100_infoGetHeaterStatus()		777
ux100_infoGetBootVersion()		778
ux100_infoGetDiagCounter()		779
ux100_infoSetDiagCounter()		780
ux100_secGenerateRandom()		782
ux100_secSetSecureDate()		783
ux100_secLoadCertificate()		784
ux100_secLoadPublicKey()		785
ux100_secReadPublicKey()		786
ux100_secReadCertificate()		787
ux100_secGetKeyList()		788
ux100_secGetCurrSECIRQ()		789
ux100_secGetSavedSECIRQ()		790
ux100_secGetCurrentUKSR()		791
ux100_secGetSavedUKSR()		792
ux100_secTamperNumEntHandl()		793
ux100_mfgRemovalSwitchReset()		795
ux100_mfgClearTamper()		796
mdb_setupDevices()		798
mdb_closeDevices()		799
mdb_receive()		800
mdb_peek()		801
mdb_advance_buffer_tail()		802
mdb_getVersion()		803
mdb_getConfig()		804
mdb_setConfig()		805
mdb_setBusTimings()		806
mdb_setMdbLinkState()		807
mdb_getTimeSinceLastPoll()		808
mdb_checkPolling()		809
mdb_setCardReaderAddress()		810
mdb_getCardReaderAddress()		811
mdb_sendJustReset()		812

Table 1 Function Calls (continued)

Function Call	Description	Page
mdb_send()		813
mdb_flush()		814
mdb_cmdType()		815
mdb_displayText()		816
mdb_clearDisplay()		817
mdb_sendReaderConfigData()		818
mdb_sendPeripheralId()		820
mdb_setOptionalFeatures()		822
mdb_sendDateTime()		823
mdb_sendDateTimeRequest()		824
mdb_sendDravs()		825
mdb_sendMalfunctionError()		826
mdb_sendDiagnosticResponse()		827
mdb_sendSendDataEntryRequestResp()		828
mdb_sendDataEntryCancel()		829
mdb_sendFTLRetryDeny()		830
mdb_sendFTLSendBlock()		831
mdb_sendFTLOkToSend()		833
mdb_sendFTLRequestToSend()		834
mdb_FTLReceiveFile()		835
mdb_FTLReceiveFileExt()		837
mdb_FTLRespondToReceiveRequest()		839
mdb_FTLSendFile()		841
mdb_sendBeginSession()		843
mdb_sendSessionCancelRequest()		845
mdb_sendVendApproved()		846
mdb_sendVendDenied()		847
mdb_sendEndSession()		848
mdb_sendOutOfSequence()		849
mdb_sendCancelled()		850
mdb_sendRevalueApproved()		851
mdb_sendRevalueDenied()		852
mdb_sendRevalueLimitAmount()		853
mdb_setDisplayData()		854
remoteprinter_getVersion()		861
remoteprinter_Open()		862
remoteprinter_TurnOn()		863
remoteprinter_TurnOff()		864
remoteprinter_Flush()		865
remoteprinter_GetStatus()		866

Table 1 Function Calls (continued)

Function Call Error Codes

Error codes for V/OS are listed in Table 2.

Table 2 Error Codes

Error Code	Value	Description
E_KM_ACCESS_DENIED		Too many PIN sessions requested.
E_KM_BAD_SEQUENCE		Bad key management session sequence.
E_KM_KEY_INTEGRITY_ERROR		VSS_KLK is corrupt.
E_KM_NO_KEY_LOADED		VSS_KLK is absent; no encrypted loading possible.
E_KM_OUT_OF_RANGE		ucKeySetID or ucKeyID is out of range or script is not loaded.
E_KM_SYSTEM_ERROR		
E_VS_BAD_CHAINING		Bad sequence of macro.
E_VS_MACRO_NOT_EXIST		Macro does not exist in this script.
E_VS_SCRIPT_NOT_LOADED		Script not installed or not accessible.
EBADMSG		Number of packets received lower than packets transmitted.
ECOMM		Communication error.
EDESTADDRREQ		No route found in the routing table.
EHOSTDOWN		Host device is not available.
EHOSTUNREACH		Host name resolution error.
EINPROGRESS		Scan in progress.
ENETDOWN		Network interface is not online.
ENOBUF		Not enough room in the buffer for the message.
ENOEXEC		Routing table column missing.
ENOLINK		Error while creating the link file.
ENOMSG		No packet transmitted.
ENONET		No network interface available.
ENOTCONN		Device not on.
ETIMEDOUT		Search stopped; no network found.
ENOENT	2	No such file or directory.
ESRCH	3	No such process.
EINTR	4	Interrupted system call.
EIO	5	Failure to write first portion of command.
ENXIO	6	No such device or address (or beyond limit).
EBADF	9	All functions other than open(); Console owned by another task; invalid file handle.
EAGAIN	11	Resource temporarily unavailable.
ENOMEM	12	No memory available.
EACCES	13	Caller’s parameter is invalid because it is not part of the caller’s memory (permission denied).

Table 2 Error Codes (continued)

Error Code	Value	Description
EFAULT	14	Bad address (hardware fault using argument).
EBUSY	16	A write or control function was issued before the previous function is completed (device or directory in use).
ENODEV	19	Console currently owned by another task.
EINVAL	22	Invalid function parameter (argument).
ERROR_SECINS_NO_MEMORY	100	Out of memory.
ERROR_SECINS_MKDIRe	101	Failed to make directory.
ERROR_SECINS_FILE_AUTH	102	Failed to authenticate package.
ERROR_SECINS_MOUNT_RFS	103	Failed to mount file system.
ERROR_SECINS_INVALID_INIT_FILE	104	System init file not found.
ERROR_SECINS_PATHNAME_TOO_LONG	105	Pathname too long
ERROR_SECINS_CHROOT	106	Switch root operation failed
ERROR_SECINS_EXEC_INIT	107	Failed to run the system init file.
ERROR_SECINS_OPENDIR	108	Failed to open directory.
ERROR_SECINS_RENAME_DIR	109	Failed to rename directory.
ERROR_SECINS_CHMOD	110	Failed to set file permissions.
ERROR_SECINS_CHOWN	111	Failed to set file ownership.
ERROR_SECINS_FILE_CREATE	112	Failed to create file.
ERROR_SECINS_FILE_OPEN	113	Failed to open file.
ERROR_SECINS_FILE_WRITE	114	Failed to write file.
ERROR_SECINS_NO_SIG	115	No package signature file.
ERROR_SECINS_NO_PKG	116	No matching package file with signature.
ERROR_SECINS_INVALID_FILENAME	117	Invalid p7s signature filename.
ERROR_SECINS_INVALID_INSTALL	118	Installed package is invalid.
ERROR_SECINS_INVALID_INI_SECTION	119	Invalid ini file section.
ERROR_SECINS_MKNOD	120	Failed to create device node.
ERROR_SECINS_INVALID_INI_PROPERTY	121	Invalid ini file property.
ERROR_SECINS_NO_CONTROL_FILE	122	No control file in package.
ERROR_SECINS_INVALID_CONTROL_FILE	123	Invalid control file in package.
ERROR_SECINS_CREATE_CONTROL_FILE	124	Failed to install control file.
ERROR_SECINS_OLD_PKG_VERSION	125	Newer package already installed.
ERROR_SECINS_INVALID_PKG_TYPE	126	Invalid package type.
ERROR_SECINS_EXTRACT_PKG_CONTROL	127	Failed to extract package CONTROL directory.
ERROR_SECINS_EXTRACT_PKG	128	Failed to extract package archive.
ERROR_SECINS_INVALID_PKG	129	Invalid package.
ERROR_SECINS_PKG_LIST	130	Failed to read package contents.
ERROR_SECINS_PKG_FILENAME_TOO_LONG	131	Package filename too long.
ERROR_SECINS_INSTALL_CONTROL_FILE	132	Failed to install package control file.
ERROR_SECINS_INSTALL_PKG	133	Failed to install package.

Table 2 Error Codes (continued)

Error Code	Value	Description
ERROR_SECINS_INSTALL_SIG	134	Failed to install package signature.
ERROR_SECINS_PKG_NOT_FOUND	135	Incomplete package install.
ERROR_SECINS_ADD_GROUP	136	Failed to add group.
ERROR_SECINS_ADD_USER	137	Failed to add user.
ERROR_SECINS_ADD_SHADOW	138	Failed to add shadow password entry.
ERROR_SECINS_GET_USER	139	User not found.
ERROR_SECINS_GET_GROUP	140	Group not found.
ERROR_SECINS_PROCESS_FORK	141	Unused
ERROR_SECINS_CHECK_GRSEC	142	grsecuity policy check failed.
ERROR_SECINS_START_GRSEC	143	Failed to enable grsecurity.
ERROR_SECINS_FORK	144	Failed to create child process.
ERROR_SECINS_GET_RESOURCE	145	Failed to get system resource limits.
ERROR_SECINS_SETSID	146	Failed to create new session.
ERROR_SECINS_START_LINE_TOO_LONG	147	Start file line too long.
ERROR_SECINS_START_LINE_INVALID	148	Start file line invalid.
ERROR_SECINS_START_ENTRY_UNKNOWN	149	Start file entry unknown.
ERROR_SECINS_EXEC_APP	150	Failed to start application.
ERROR_SECINS_MODE_APP	151	Start file entry not executable.
ERROR_SECINS_SYMLINK	152	Failed to create symlink.
ERROR_SECINS_INVALID_INSTALL_FILE	153	Invalid install archive type.
ERROR_SECINS_EXTRACT_INSTALL_FILE	154	Failed to extract download file.
ERROR_SECINS_INVALID_PKG_FORMAT	155	Invalid package file type.
ERROR_SECINS_AUTH_GID	156	Invalid filetype for authenticating.
ERROR_SECINS_ADD_CERT	157	Failed to add certificate.
ERROR_SECINS_PATCH_VERSION	158	Patch version does not match the installed package.
ERROR_SECINS_PATCH	159	Failed to patch package.
ERROR_SECINS_START_SVCSEC	160	Failed to start svcsec application.
ERROR_SECINS_INVALID_CERT	161	Invalid certificate file.
ERROR_SECINS_PUT_ENV	162	Failed to process package config file.
ERROR_SECINS_SET_USER	163	Failed to set user ID.
ERROR_SECINS_INVALID_PKG_CATEGORY	164	Invalid package category specified.
ERROR_SECINS_INVALID_CMD	165	Invalid command in remove file.
ERROR_SECINS_INVALID_BUNDLE_FILE	166	Invalid bundle file.
ERROR_SECINS_EXTRACT_BUNDLE_CONTROL	167	Failed to extract bundle file.
ERROR_SECINS_INVALID_BUNDLE_USER	168	Invalid bundle user specified.
ERROR_SECINS_INVALID_PKG_USER	169	Package user does not match bundle user.
ERROR_SECINS_NO_BUNDLE	170	No bundle file matching bundle p7s signature.
ERROR_SECINS_OPEN_SOCKET	171	Failed to open socket.
ERROR_SECINS_BIND_FAILURE	172	Failed to bind UNIX domain address.

Table 2 **Error Codes** (continued)

Error Code	Value	Description
ERROR_SECINS_UNKNOWN_MSG	173	Unknown Secure Installer message.
ERROR_SECINS_MSG_SND	174	Failed to send API response message.
ERROR_SECINS_MSG_RCV	175	Failed to read API message.
ERROR_SECINS_MSG_SIZE	176	API message too big.
ERROR_SECINS_RESP_SIZE	177	API response message too big.
ERROR_SECINS_INVALID_RESP	178	Invalid API response.
ERROR_SECINS_FILE_READ	179	Failed to read file.
ERROR_SECINS_FILE_RENAME	180	Failed to rename file.
ERROR_SECINS_USER_APP_START	181	Failed to start application.
ERROR_SECINS_GET_CWD	182	Failed to get current working directory.
ERROR_SECINS_INVALID_PARAM	183	Invalid parameter specified.
ERROR_SECINS_NOT_ALLOWED	184	Permission denied.
ERROR_SECINS_SYSMODE_START	185	No sysmode loaded.
ERROR_SECINS_INVALID_PKGLIST	186	Package list file invalid.
ERROR_SECINS_SOCK_LISTEN	187	Failed to setup API socket.
ERROR_SECINS_CONNECT_FAIL	188	Failed to connect to secins API socket.
ERROR_SECINS_PATCH_APPLIED	189	Low layer patch already applied.
ERROR_SECINS_RELOAD_GRSEC	190	Failed to reload the grsecurity policy.
ERROR_SECINS_CREATE_GRSEC_FILE	191	Failed to create grsec policy file.
ERROR_SECINS_GRSEC_SUBJECT	192	Subject in grsec file does not exist.
ERROR_SECINS_INVALID_POLICY_FILE	193	Invalid grsec policy file.
ERROR_SECINS_INVALID_OBJECT	194	Object in grsec file invalid or not allowed.
ERROR_SECINS_INVALID_OBJECT_MODE	195	Object mode in grsec file not allowed.
ERROR_SECINS_NO_POLICY_FILE	196	System grsec policy file not found.
ERROR_SECINS_INVALID_CAPABILITY	197	Capability in grsec file not allowed.
ERROR_SECINS_OBJECT_NOT_HIDDEN	198	Object in grsec file must be hidden.
ERROR_SECINS_OBJECT_NOT_FOUND	199	Required object in grsec file not found.
ERROR_SECINS_FILE_CAPS	200	Failed to add file capabilities.
ERROR_SECINS_SUBJECT_MODE	201	Invalid subject mode in grsec file.
ERROR_SECINS_REMOVE_USER	202	Failed to remove user.
ERROR_SECINS_REMOVE_GROUP	203	Failed to remove group.
ERROR_SECINS_PID_NOT_FOUND	204	Process does not exist.
ERROR_SECINS_SIGNAL_FAIL	205	Failed to send signal.
ERROR_SECINS_LOAD_VSS	206	Failed to load VSS script to vault.

Table 2 Error Codes (continued)

Error Code	Value	Description
ERROR_SECINS_UNSIGNED_PKG_USER	207	Unsigned package being installed by invalid user.
ERROR_SECINS_PATCH_AUTH	208	Failed to authenticate patched package.
ERROR_SECINS_EXTRACT_PATCH_CONTROL	209	Failed to extract control file from patched package.
ADE Module Error Codes		
Error Code	**Value**	**Description**
ADE_SUCCESS	0	Success.
ADE_ERR_PM_PTR	-1	A pointer parameter references unusable memory (for example, a null pointer, the buffer is too small, and so on).
ADE_ERR_PM_LEN	-2	`ade_encrypt_in ptextLen` is an invalid value.
ADE_ERR_PM_KEY	-3	`ade_encrypt_in keyIndex` is an invalid value.
ADE_ERR_PM_ALG	-4	`ade_encrypt_in encAlg` is an invalid value.
ADE_ERR_PM_MODE	-5	`ade_encrypt_in encMode` is an invalid value.
ADE_ERR_PM_IV	-6	`ade_encrypt_in iv` is an invalid value.
ADE_ERR_PM_PAD	-7	`ade_encrypt_in pad` is an invalid value.
ADE_ERR_NO_KEY	-11	A key cannot be retrieved from the specified slot.
ADE_ERR_OFF	-12	ADE is turned off.
ADE_ERR_GENERIC	-99	An undefined or unexpected error occurred.

CHAPTER 2

Overview

V/OS is the next-generation Linux OS. Initial V/OS platforms are based on the Linux kernel 2.6.31.

Terminal Features

Table 3 lists Verifone V/OS-based terminals and the features.

NOTE Some terminals have configurable modules and may not be configured with the supported feature.

Table 3 Terminal Feature Sets

	Mx915	Mx925	Ux200	Ux300	Ux301	VX 520 Color	VX 675 GPRS	VX 675 BT/Wi-Fi	VX 675 3G	VX 820
LED	✓	✓		✓	✓	✓	✓			✓
Display	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Bluetooth	✓	✓						✓		
Printer				✓	✓	✓	✓	✓	✓	
CTLS	✓	✓		✓	✓	✓			✓	
USB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
COM Ports 1–4	✓	✓	✓	✓	✓	✓	VX 675 COM ports are via dongle or when docked in the docking station.			✓
COM Port 6			✓							
RT Clock	✓	✓		✓	✓	✓	✓	✓	✓	✓
Mag Stripe	✓	✓		✓	✓	✓	✓			✓
Audio	✓	✓								
Touch	✓	✓								✓
Tailgate	✓	✓				✓	✓	✓	✓	✓
IPP	✓	✓								
ISDN				✓	✓					
Modem				✓	✓	✓	✓	✓	✓	

Table 3 Terminal Feature Sets (continued)

	Mx915	Mx925	Ux200	Ux300	Ux301	VX 520 Color	VX 675 GPRS	VX 675 BT/Wi-Fi	VX 675 3G	VX 820
GPRS							✓			
WDE	✓	✓								
Multi-App	✓	✓				✓	✓	✓	✓	✓
Function Calls	✓	✓		✓	✓					
Input Events						✓	✓	✓	✓	✓
Crypto Lib	✓	✓		✓	✓	✓	✓	✓	✓	✓
IPP Legacy	✓	✓				✓	✓	✓	✓	✓
Wi-Fi	✓	✓					✓	✓		

C/C++ Compiler and Tools

The C/C++ compiler and development tools can be purchased through Verifone and comprise a customized version of the Mentor Sourcery CodeBench DTK. The CodeBench DTK is a Windows-based IDE with an integrated debugger. Verifone distributes a separate SDK that includes the libraries, headers, and tools specific for developing V/OS-based devices.

Symbolic Debugger

A source code symbolic debugger for debugging application programs will be included in the Eclipse Integrated Development Environment. The debugger utilized is the open source GNU Debugger (GDB).

System Overview

Figure 1 illustrates the V/OS system structure.

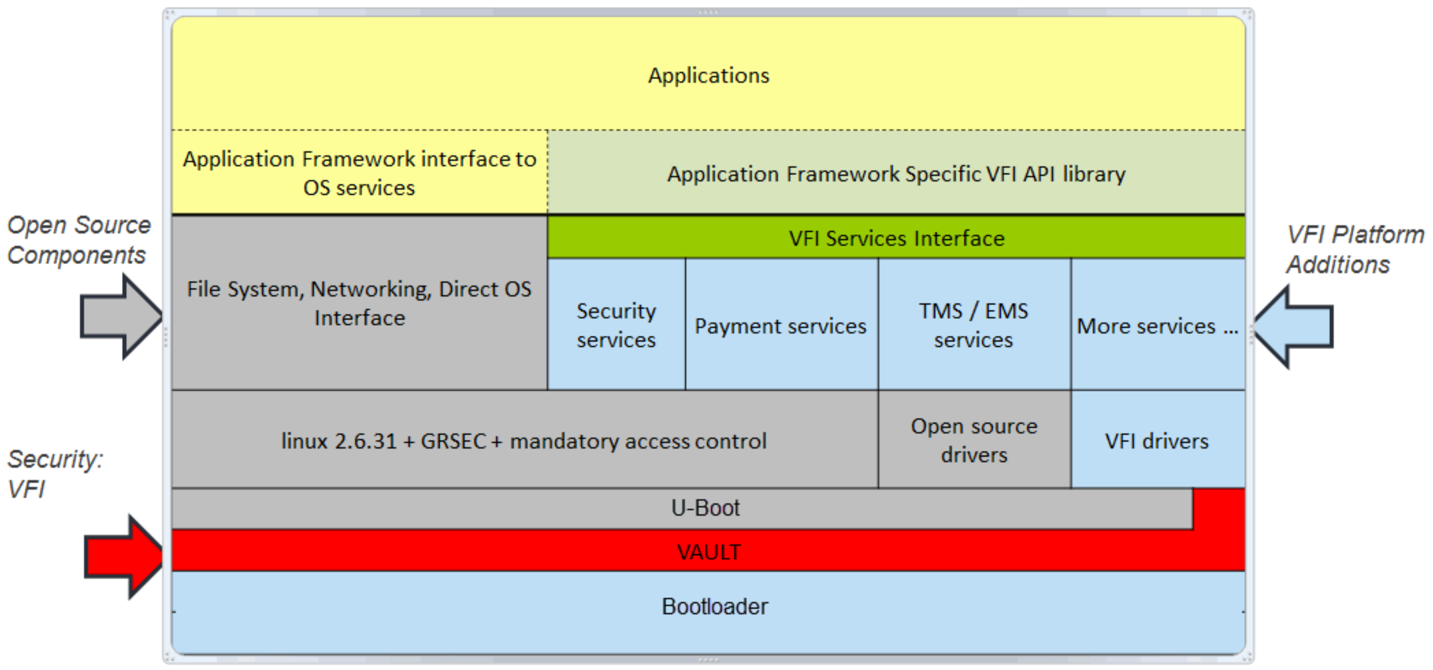


Figure 1 System Architecture

Boot-up Sequence

When the V/OS terminals boot up and the Linux U-Boot loader completes loading, the following occurs:

1. The kernel initially boots using initramfs.
2. The symlink `/init` in initramfs is loaded. This is the symlink to the first-stage Secure Installer (SI), which authenticates and builds the final root file system (RFS).
3. A switch root operation occurs that makes the mounted tmpfs the root file system (RFS).
4. The second-stage Secure Installer starts in this new RFS.

   The system boots, calling rc.d scripts to initialize the OS.
5. After all rc.d scripts execute, SI authenticates and extracts all app packages to DRAM using the provided User home accounts.
6. Secure Installer starts the applications as defined in the “start” file of the associated app package.

Environment and Configuration Variables

Linux has long had the concept of environment variables (also called configuration variables). Unfortunately, environment variables are not permanent and Linux normally requires a shell script that sets the environment variables prior to their use.

getenv() and putenv() are native functions to Linux. Use putenv() to set/change/delete an environment variable. These values do not persist after a power failure. V/OS terminals support the enhanced APIs getEnvFile() and putEnvFile() to get or set environment variables in the flash file system.

File Format

V/OS terminals use a standard file format to store configuration variables. The file stores ASCII information in `.ini` format For example:

```
[perm]
i4683=R232
o4683=01234567
[reg]
store=sportworld
appid=1234
```

`ini` files consist of section headers (defined within []) followed by keys (or labels) and their associated values. For example, `appid` is a key with value 1234 and it resides in the `reg` section. By default, all configuration files have two sections, `perm` and `reg`.

To aid compatibility with previous terminals, all keys that begin with ‘ * ’ a automatically placed in the perm section, provided the section parameter is a pointer to an empty, null-terminated string (that is, empty quotes " ") The leading ‘ * ’ character is not actually stored in this case. It is only there to indicate to store or retrieve the label field from the perm section.

If a section is specified, then the value field is stored as it is given to the function, including any ‘ * ’ character. This also applies even if the section is specified as perm. It is only if an empty, null-terminated string is specified for the section that leading ‘ * ’ characters are ignored.

All created sections and label entries are converted to lowercase before being stored to eliminate case sensitivity. The getEnvFile() and putEnvFile() functions always convert to lowercase before searching or storing.

The value field is case-sensitive and is always stored as passed to the functions. There are certain reserved characters with the INI parser library that cannot be used in section, label, or value fields, and certain characters are not accepted. All other characters can be used, including non-printable ASCII characters. Refer to the INI parser HTML document included in the SDK for how the INI parser works and what is supported. In addition to that documentation, the following characters are reserved and cannot be used within one or more of the fields of the environment variable or have special rules concerning them:

Character	Hex	Decimal	Usage
Null	0x00	0	Used to terminate string and cannot be stored as part of section, label, or value field.
Space	0x20	32	Cannot be used in leading and trailing position of section, label, or value field. Can only be used in non-leading and trailing positions.
Equal (=)	0x3D	61	This is a reserved character and cannot be used in any field. If used, unexpected and unsupported results may occur.
Horizontal Tab	0x09	9	Cannot be used in leading and trailing position of section, label, or value field. Can only be used in non-leading and trailing positions.
Line Feed	0x0A	10	This is a reserved character and cannot be used in any field. If used, unexpected and unsupported results may occur.
Vertical Tab	0x0B	11	Cannot be used in leading and trailing position of section, label, or value field. Can only be used in non-leading and trailing positions.
Form Feed	0x0C	12	This is a reserved character and cannot be used in any field. If used, unexpected and unsupported results may occur.

Character	Hex	Decimal	Usage
Carriage Return	0x0D	13	This is a reserved character and cannot be used in any field. If used, unexpected and unsupported results may occur.
Hash Sign (#)	0x23	35	This character is used to insert comments into configuration file. Any section specified for a label starting with the # character is inserted into the file and the line is considered a comment.
Semi-colon (;)	0x3B	59	This character is to insert comments into configuration file. Anything after a ";" character is treated as and inserted into the file as a comment.
Forward Slash (/)	0x2F	47	This is a reserved character and cannot be used within the section field.
Left Bracket ([)	0x5B	91	This is a reserved character and cannot be used within any field. If used, unexpected and unsupported results may occur.
Right Bracket (])	0x5D	93	This is a reserved character and cannot be used within the section field.

NOTE

No configuration file environment variables actually reside in the current shell environment.

Most shell environment variables in Linux/UNIX systems are stored in uppercase. If an application expects an environment variable that resides in a configuration file to use the current case, the application must retrieve the environment variable from the configuration file and set it to use the current shell environment using putEnvFile().

For example, if the environment variable "`myfile=some_path`" is stored in a configuration file, the application must perform a getEnvFile() for "myfile" to retrieve its value, and then set it in the current shell environment with putenv("`MYFILE=some_path`") as the application expects the case to be set. Until a putEnvFile() is performed with case sensitivity in mind, the variable does not reside in the current shell environment.

There are also command line versions of the two functions getEnvFile() and putEnvFile() that are mainly used during development. When these commands are used, each part, label, and value fields must be in separate double-quotes (`"  "`) to prevent the Linux shell from interpreting the characters. For example: `putenvfile "*dhcp" "1"`.

These functions do not support custom section creation. To create a custom section use the API putEnvFile() with the label parameter set to an empty, null-terminated string, and the section parameter set to the desired custom section name. The custom section is not created if the label parameter is not an empty string.

Each user has its own `config.usrx` configuration file, where the value of x is 1-16. Root has its own configuration file. Configuration files are stored in the flash file system at `/mnt/flash/config`.

Environment Variable Functions

This section describes the Environment Variable calls.

getEnvFile()

Retrieves the configuration file environment variables.

Prototype

```
int result = getEnvFile(char *section, char *label, char *value,
int vlen);
```

Parameters

`section`	Points to a null-terminated string containing the `.ini` file section name. Maximum length of the section name is 32 bytes. **Note:** If section is a pointer to null, the system automatically reads from with `perm` or `reg` sections. `perm` is selected if the first character of label is an “ * ”, otherwise the `reg` section is used.
`label`	Points to a null-terminated string containing the label (reference name) of the entry. Maximum length of the label is 32 bytes and is automatically converted to lowercase before searching for its value.
`value`	Points to an area where the value of the environment variable is copied. Maximum length is 512 bytes. Value is null-terminated.
`vlen`	Maximum length of the `value` field. If the length of `value` is larger than `vlen-1`, then only `vlen-1` bytes are copied to `value`.

Return Values

> 0	Length of `value`
= 0	Entry not found
< 0	An error occurred
-ENOBUFS	Internal message error

putEnvFile()

Set environment variables in the flash file system.

Prototype

```
int result = putEnvFile(char *section, char *label, char *value, unsigned
short vlen, unsigned short options);
```

Parameters

`section`	Points to a null-terminated string containing the `.ini` file section name. Maximum length of the section name is 31 bytes. **Note:** If section is a pointer to null, the system automatically reads from the `perm` or `reg` section. `perm` is selected if the first character of label is an “ * ”, otherwise the `reg` section is used.
`label`	Pointer to a null-terminated string containing the label (reference name) of the entry. Maximum length of the label is 32 bytes and is automatically converted to lowercase before being stored. To create a new section, pass a pointer to null for the `label` parameter.
`value`	Pointer to a null-terminated string containing the value associated with the label. Maximum length is 512 bytes. To delete an entry pass a pointer to null.

Return Values

= 0	No error
< 0	An error occurred
-ENOBUFS	No environment variable space

System Configuration Variables

The following table of configuration variables are read by the system on power up and reboot. These configuration variables must be set in the usr1 account.

Variable Name	Values	Definition
`*GO`	Executable name	The path `/home/usrx` (where *x* is the user that the package was signed and installed as) is pre-pended to the value set in `*GO`. Example: `*GO=myapp` The system will attempt to execute the file: `/home/usrx/myapp` Note that Linux is case-sensitive.
`*DHCP`	Non-zero	If `*DHCP` is present and the system supports Ethernet, then it attempts to initialize its connection using DHCP.
`*DHCPID`	Unique ASCII string	Use the Client ID as the identifier in DHCP. By default, the MAC address is used as the identifier by the protocol.
`*DNCPHOST`	Variable ASCII string	Use this Hostname option to name of the DHCP client.
`*ETHERSPEED`	“100F”, “100H”, “10F”, or “10H”	Use to override full auto negotiation of Ethernet device eth0 by advertising the value as the maximum. 10 or 100 Mbps are the speed and F or H are full or half duplex mode, respectively. The default is to auto-negotiate with all capabilities up to 100H.
`*IFCONFIG`	Per Linux – No need to set MAC address as the system will do this for you. See the TCP/IP Ethernet chapter.	Use either `*DHCP` or `*IFCONFIG`.
`*GATEWAY`	IP Address in the form xxx.xxx.xxx.xxx	Use to define the address of the gateway (router).
`*TELNET`	1	If `*TELNET` is present and the system supports Ethernet, then it starts the Telnet server daemon.

Variable Name	Values	Definition
*DBG	1	Starts the gdbserver when an application (specified by: *GO) starts. Use gdbserver in conjunction with the Eclipse development environment.
*RLOG	1	Enables rlogcntrld (remote syslog control daemon).
*BACKLIGHT	1–63	Sets the display backlight brightness. Default is 15.
*VOLUME	0–100	Sets the speaker volume. Default is 50.
*FTPHOST	xxx.xxx.xxx.xxx or ASCII server DNS name Max length = 96 characters	Either IP address or DNS name of FTP server.
*FTPPORT	Default = 21	Port number for FTP connection.
*FTPFILE	Default=app.tgz Max length = 96 characters	Remote file name (and path) of file to be retrieved from the FTP server.
*FTPPWD	Default = Terminal S/N Max length = 32 characters	FTP password.
*FTPUSER	Default = anonymous Max length = 32 characters	FTP user ID.

*SYSLOG_LOGSIZE	Default = 64
	Min = 16
	Max = 128
		Size of each log file in Kbytes.

*SYSLOG_MARK	Default = 30
	Min = 10
	Max = 1440
	Off = 0
		Specifies when a “MARK” timestamp marker is placed in the syslog messages file.

*SYSLOG_RHOST	Default = null	Specify a remote IP address to send syslog messages using a UDP network connection.

*SYSLOG_ROTATE	Default = 2
	Min = 0
	Max = 3
		Number of rotating log files.

Variable Name	Values	Definition
`*SYSLOG_RPORT`	Default = 5140	Specifies the port to use when sending syslog messages using a remote UDP network connection.
`*SYSLOG_RLOG`	Default = FALSE	Logs syslog messages remotely.
`*SYSLOG_USRLOCAL`	Default = TRUE	Logs user log messages locally (this option does not affect OS or root generated messages).
`*VPAY`	1	Used if no application loaded. The system attempts to contact the PAYware Vision host for an application download.
`*VPSERVERADDRESS`	xxx.xxx.xxx.xxx	IP address of PAYware Vision server.
`*VPSERVERPORT`	xxxxx	Connection port address on PAYware Vision server.
`*VPDOWNPORT`	xxxxx	Connection port address on the V/OS terminal.
`*VPSSL`	1	If set to 1, then System Mode uses SSL to retrieve the initial download.
`*BOOTDL`	“vp”, “ftp”, “usb”, or “ecr”	Used when no application loaded. The system attempts to perform an application download only by the specified means. only use this variable when the terminal uses two possible mechanisms, such as VP and ECR.

CHAPTER 3

Secure Installer

The Secure Installer is a root process responsible for installing all files on the system, starting user applications, and providing a basic message interface for running applications to access some root services for example, reboot.

All files to install on the system must be contained in signed packages within signed bundles. The root file system (RFS) is in DRAM and all signed software packages are stored in flash. Software packages are authenticated and extracted to the RFS at every boot up.

Software Installation

An important distinction is made between the install packages that reside in the system itself, and the files used to download those packages to the terminal:

- A signed package (that is, the package file and its p7s signature file) can only be downloaded in a signed bundle.

  A bundle is an archive that contains one or more signed packages, optional signer certificates, and the control and remove files. Bundles must also be signed and have an associated p7s signature.
- To download all files as a single file, combine the install bundles and their associated signatures as an archive.

Download files can contain multiple bundles, and bundles can contain multiple application packages each with a signed install package, signer certificates, and the remove file.

Install Packages

Install packages are designated OS, System, or User, as identified in the control file. Each install package type is authenticated using a filetype value, and only authorized signers can install a package.

For example, OS packages are authenticated using file type `'0'`. If the signer certificate does not contain filetype `'0'`, authentication fails and the package is rejected.

OS Packages

OS packages are signed by the OS signer and authenticated using the filetype `'0'` in the control file. Standard OS package types create the RFS in DRAM. OS package types extract to the RFS directory (/). All files and directories in the package must be relative to this directory (/). File ownership on all files in the package after extraction to the RFS is `root:root` (the owner is root and `Group: root`). Use rc scripts to start system processes, as usual.

Signed osflash package types are extracted to flash at `/mnt/flash`. Use this package type for large media files, fonts, and so on. These file types can be authenticated and extracted to flash one time only. Executable files and libraries cannot be loaded in this way.

The lowlayers package type upgrades the low layers such as sbi, uboot, vault, kernel, and initramfs.

app Packages

app packages are signed by an App signer and authenticated using the filetype corresponding to the user indicated in the package control file. For example, use filetype `'1'` for user usr1, filetype `'2'` for user usr2, and so on through filetype `'16'` for user usr16. This implies that user information is contained within the package.

The user app package type is extracted to the relevant user home account created in DRAM at boot up. The files extracted to the home account are owned by the relevant user. User packages contain a start file that indicates which applications to automatically start. Once all app packages extract to the RFS, Secure Installer starts the user applications.

The package types `user`, `userflash`, `config`, `flashconfig`, and `service`, are supported. The flash directory for system applications is:
`/mnt/flash/userdata/<username>`
where, <username> is the user usr*x* value. For example, for usr1, the flash directory is: `/mnt/flash/userdata/usr1`

System app Packages

System app packages are app packages where the user is the system (sys1 through sys16), and is designated as `filetype 'S1'` for user sys1, `filetype 'S2'` for sys2, and so on through `filetype 'S16'` for sys16. System apps are signed by the System signer (under the Verifone OS sponsor), and authenticated using `filetype` set to the corresponding user as indicated in the package control file.

The package types `user`, `userflash`, `config`, `flashconfig`, and `service`, are supported for System signers. The flash directory for system applications is:
`/mnt/flash/sysdata/<username>`
where, <username> is the user sys*x* value. For example, for sys1, the flash directory is: `/mnt/flash/sysdata/sys1`

Downloads

Secure Installer does not download packages to the unit; it installs download files downloaded by the sysmode application or other user applications. Signed packages must download in a download file. Secure Installer processes download files placed in the `/mnt/flash/install/dl` directory, which has open write permissions.

Secure Installer processes a download file and installs all included packages in flash. If the system is running during this process (for example, if an application downloads a file and calls the Secins_install_pkgs()), running applications are not affected. Newly installed packages are not extracted to DRAM or flash (for '`userflash`' package types) until a reboot or the application calls Secins_start_user_apps().

Since the DRAM tmpfs executes in place, extracting newly installed DRAM packages while applications are running could cause serious problems. '`unsigned`' package types are the only exception, as they only extract in the User flash directory designated in the control file.

Flash Directories

Directories in flash include the user flash directories previously described, plus the log and share directories.

The log directory contains directories for each user on the system to store log files. Secure Installer creates a '`logs`' symlink in the User home account, pointing to the User log directory. Ensure that log file symlink references are relative to the home account. There is also the common `/mnt/flash/logs/share` directory that all users can access.

The share directory `/mnt/flash/userdata/share` is accessible by all app users (usr1–usr16).

Users and Groups

Applications can run as different users so that the files belonging to different applications can remain private. Several applications can also run as the same user.

To be able to easily move your application to a different user, do not make assumptions about its user name and home account. Follow these guidelines:

- Only use relative pathnames when referencing files in the home account. Do not use absolute pathnames.

  All references must be relative to the home account. For example, `mydata.bin` or `./mydata.bin`, not `/home/usr1/mydata.bin`
- All references to the User flash directory must use the flash symlink in the home account. For example, `./flash/video.mpg`
- Use the `HOME` environment variable to retrieve the home account pathname. It is the working directory when the application first starts, after which use `getcwd()` to retrieve the home account pathname.
- Do not reference files in another User home account. Applications share files using a common folder in the RFS or by using other forms of IPC (Inter-Process Communication).

Following these guidelines also makes it easy to change the application user. Just change the user specified in the package control file and sign the package.

Assign Users

All install packages must be signed. With Pedguard there can be only one application sponsor, but there can be multiple application signers in a terminal. A single application signer can have multiple signer cards.

The install package indicates the user that the package is installed as (for example, usr1, usr2, and so on). usr1 is assumed if no user is specified. The filetype directly relates to the user the package is installed to. Filetype `'1'` is usr1, filetype `'2'` is usr2, and so on.

The system determines the User account using the App signer certificate and the application install package. The User account can only be reassigned by changing the specified User value in the control file and downloading a new signed install package. The package filetype is passed to authenticate() during authentication, so the signer certificate must allow the filetype specified in the install package. Authentication fails otherwise.

Ensure that app signers, in particular those whose applications may run side-by-side on the same terminal, have a different range of filetypes specified in their certificates. This makes it impossible applications to run as the same user. If every app signer uses filetype `'1'` and two are required to run on the same terminal, the install package must be changed to indicate a different user and re-signed.

This scheme guarantees that applications signed by different app signers will run as different users, as long as the app signer certificates have a completely different range of filetypes specified. For example, if two application developers have applications that are required to run side-by-side on the same terminal, one should have the signer assigned filetypes `'1'` … `'4'`, and the other use filetypes `'5'` … `'8'`, to ensure that their applications can never run as the same user.

NOTE Both signers must share a common app sponsor, so that the sponsor can control the filetypes assigned to each signer.

Example 1–Default User

- The install package specifies usr1 or does not specify the user, so the default, usr1, is used.
- The app signer certificate contains filetype `'1'` in the list of allowed filetypes.

For this package then the following applies.

- User name: usr1.
- The home directory is `/home/usr1`
- The uid is assigned by the system (uid 500 is for usr1, 501 for usr2, and so on).

Example 2–Not the Default User

- The install package specifies usr2.
- The app signer certificate contains filetype `'2'` in the list of allowed filetypes.

Then:

- The user name is usr2
- The home directory is `/home/usr2`

Assign Groups

Secure Installer uses a user-private-group scheme when assigning users to groups. When a new user is added to the system, a user private group is created. This group has the same name as the user and that user is the only member of that private group.

There is also an App signer group created for each App signer. The system contains two application signer groups: `users` for customer/user applications, and `system` for system applications. All users belonging to a particular App signer are automatically a member of that App signer group. For example, users usr1, usr2, …, usr16 are members of the `users` group, and users sys1, sys2, …sys8, are members of the `system` group.

There is also the common group, `share`. All users, usr1–usr16 and sys1–sys16, are members.

Install packages indicate whether to install its contents using the user private group (default), the App signer group (system or users), or the share group. To allow a file be readable by other applications in your App signer group (but not readable by all), requires that the file is installed using that App signer group. So, from the install package point of view the possibilities are:

- Package group not specified: use the user private group.
- Package group specified can only be one of the following:
  - Matches the user pseudonym specified in the package, which uses the user private group.
  - Specify the users group for customer applications or the system group for system applications using the App signer group.
  - Specify the share group to share files with all installed applications.

For example, if the install package control file specifies `user user1` and no group is specified, then User is usr1, `/home/usr1` is the home directory, and the group defaults to usr1. If the App signer group is specified and install package control file specifies `user usr1`, then `/home/usr1` is the home directory and the group is `users`.

Shared Libraries

A normal application install package can include shared libraries, which are extracted to the user home directory (or a sub-directory, depending on package layout). To enable these libraries to be automatically picked up by the loader, secure installer uses `ldconfig` to ensure that the lib directory in the user home directory is automatically searched for shared libraries.

If the home directory is `/home/usr2`, `/home/usr2/lib` is searched, and the libraries in there are automatically picked up by the loader.

Note that while the lib directory in the home accounts of all users on the system are searched for shared libraries. Normally, a user does not have permissions to read libraries in other users' home accounts. If a particular app signer wants to allow different users access to shared libraries, they must be installed as the group "users."

`LD_LIBRARY_PATH` cannot be reliably used on V/OS (see File Capabilities, page 69). To ensure that the lib directory in the user home directory will be searched before any system directory or any other user home directory, use the ld linker option `rpath`, to specify the runtime library search path of the particular program being compiled. To invoke the `rpath` option, from gcc:

```
-Wl,-rpath,lib
```

Packages

Install packages contain the file system content as defined for that package type. Install packages must be signed. Install bundles contain install packages and their associated p7s file (they may also contain a remove file). Install bundles must also be signed. Install bundles and their associated p7s files are combined into a single download file.

Install Packages

The install package is a tar ball and must not be compressed. This allows packages to extract to DRAM quickly, and also means that you can use bsdiff for efficient patching. All packages must be signed, and must be accompanied by a p7s signature file. The package filename is not important but the signature must have the same filename as the package file with the .P7S or .p7s extension.

The directory structure and files in the tar-ball are as they will appear in RFS after installation; relative to / for OS-signed packages, and relative to the user home account for App- or System-signed packages.

The CONTROL directory must exist at the top-level of the tar-ball directory structure. It must contain the control file. Other files that can be in the CONTROL directory are the start, grsec, and env files.

Package Control File

The package control file is a text file that contains information relating to the package. The format is individual lines in the form *Field:value*. Place the package control file in the package CONTROL directory. The fields listed in Table 4 are supported.

NOTE

The field and value strings are not case sensitive.

Table 4 Package Control File Fields

Field	Max Length	Description
Package	24	Name of the package or bundle (required).
Version	16	Package version (required).
User	8	Specified user: usr1–usr16 or sys1–sys16 (optional). • The default value is root for OS install package types, and usr1 for all other package types. • The default value is OS for bundles.

Group	8	Group designation (optional). Only applies to system or user packages. OS packages can specify `root`.
		This value can match `User` or contain the string:
		`system/users`
		For system or user signed packages, respectively or
		`share`
		The default value matches `User`, in that way the user private group is used. Only specify this field if the package must be installed using the App signer group.

Category	16	Package category (optional): • fs – file system content, including the standard OS, system, and user packages. (default) • patch – update packages that patch an installed package.
Date	10	Package creation date in the format: YYYY-MM-DD (optional).
Option	64	Additional information.

Table 4 Package Control File Fields (continued)

Field	Max Length	Description
Type	16	Package type (optional). For fs category packages, the following types are supported: • os – standard OS packages, which extract to root (/) to form the basic RFS. (OS package type default) • osflash – signed os flash packages. • lowlayers – updates to sbi, uboot, and so on. Must be OS signed. • user – standard user packages, which extract to the user home account in DRAM on boot up. (default) • userflash – packages that extract to the User flash area. The package is deleted after extraction. • service – services, which must be readable by the svcmgr process. • config – install global config files. • flashconfig – install config files to flash. This package is deleted after extraction. • vss – install VSS scripts. • userfont – installs a font to a central location so it is available system wide. This is also a flash package type. • unsigned – unsigned user data packages. **Note:** For patch category types, `Type` must match the installed package type being patched.
Umask		

Control File Examples

- OS packages: For normal OS package downloads, the minimum fields required in the control file are `Package` and `Version`.

  ```
  Package: busybox
  Version: 1.01
  ```

  `Category` defaults to `fs`, `Type` defaults to `os`, and `user` defaults to `root`.

- user packages: For normal user package downloads, either `User` or `Type` must be specified.

  ```
  Package: testapp
  Version: 1.00
  User: usr1
  ```

  `Category` defaults to `fs`, `Type` defaults to `user` (because `User` is specified as usr1). This can also be specified as:

  ```
  Package: testapp
  Version: 1.00
  ```

```
Type: user
```

In this case, `Category` defaults to `fs`, and `User` defaults to `usr1`.

Package Start File

The start file in the CONTROL directory lists applications to automatically start for that package. Each line is a command line used to start an application. Note that command lines are parsed by Secure Installer and not in a bash shell. Special characters normally handled by the bash shell (such as, &, |, ;, and so on) have no significance. Command lines can include environment definitions and arguments, as well as the executable name.

Example

```
myapp -port /dev/ttyS0
testapp
```

This start file starts myapp and testapp.

The executable is referenced relative to the home account and not as an absolute pathname.

Environment variables can be specified prior to the executable name, as shown in this example:

```
VER=v1.1 myapp --port /dev/ttyS0
```

Package env File

Use the env file in the CONTROL directory to set up the user application environment. This file is in ini file format. Environment definitions in the global section are passed to every app started for the package. Environment definitions in sections that match the executable name are passed to the relevant application. For example:

```
[global]
DEBUG=off
PLATFORM=Fusion

[myapp]
DEBUG_LEVEL=1
```

Package grsec File

The grsec file in the CONTROL directory sets up the grsecurity policy for an application. This file is in ini file format. This file should contain a section for each executable that requires access to system resources other than those contained by default in the User account.

An application will, by default, have full grsecurity access to its home account and flash directory.

Applications can access system libraries, some standard devices (such as, `/dev/null`, `/dev/urandom`, and so on). The grsec file must request access to resources not allowed by default. These are checked against the user policy. Capabilities can also be requested in the grsec file. No capabilities are allowed by default. For example:

```
[myapp]
```

```
/dev/ttyS0    rw
+CAP_SYS_TIME

[testapp]
/dev/mag      r
```

grsecurity

grsecurity is a set of patches for the Linux kernel with an emphasis on enhanced security. One component of grsecurity is the full role-based access control (RBAC) system. RBAC is an approach to restricting system access to authorized users. To restrict access to files, capabilities, resources, or sockets to all users, including root, you must have an RBAC system. grsecurity creates a “least-privileged” system, where users and processes have the absolute minimum privileges to work and nothing more. This way, if the system is compromised, the ability by an attacker to damage or gain sensitive information on the system is reduced. See http://grsecurity.net/ for information.

grsecurity Policy File

In grsecurity, the RBAC system is managed through a policy file (`/etc/grsec/policy`), which contains a system-wide set of rules. Secure Installer is responsible for enabling grsecurity on the system, and since it is Secure Installer that adds all users to the system (based on the information in the install packages), it must also set up the grsecurity policy file based on the information in the packages.

For each user added to the system, Secure Installer adds a user role to the system policy file. This user role consists of:

- The default subject (`subject /`) that lists the objects and object modes available to the user by default.
- Other subjects can be added by the user through the install package grsec file. For example, if the user application “myapp” required a capability such as CAP_NET_RAW, which is not included in the default subject, it can include the following in the package grsec file:

```
[myapp]
    +CAP_NET_RAW
```

Secure Installer adds a subject to the user role for myapp. For example, for usr1 it would be in `/etc/grsec/policy` as:

```
subject /home/usr1/myapp
    +CAP_NET_RAW
```

The install package grsec files are subject to strict checking. User subjects are only granted access to objects already deemed appropriate for users. For example, if a user has `+CAP_SYS_ADMIN` or `/ rw`' in its package grsec file, it will not be allowed.

NOTE The default role should cover the vast majority of what a standard user requires access to on the system. So, in most cases a grsec file is not required in an install package.

File Capabilities

Capabilities allow applications to define user processes in lieu of using root processes. For example, to set the system time, the application can pass the capability CAP_SYS_TIME. File capabilities are implemented using the file system extended attributes, and are assigned based values in the grsec file.

If the grsec file requests a capability that is allowed by the user policy file, then that capability is added to the system grsecurity policy file and Secure Installer adds the corresponding file capability to the executable (in the dramfs) before it is started. Only applications that Secure Install starts directly can ever be assigned a capability. That is, if a user process starts another process using `system()` or `fork/exec`, then it can never be assigned a capability.

LD_LIBRARY_PATH is bypassed for processes that run with escalated privileges, whether `setuid` or capabilities, because someone could overlay some of the functions which the executable calls with malicious code. Because of this LD_LIBRARY_PATH is not supported, and Secure Installer does not set this variable in the environment for any user process. See Shared Libraries.

File System Content Packages

The following describes package types in the fs category, which is the default category. If specified, the Category field is set to fs for these package types.

NOTE The Package and Version fields must be present in the control file for all fs type packages.

OS Packages

OS packages extract relative to root (/) and form the DRAM file system.
OS packages must be signed by the OS signer, and have the `Type` field set to `os` in the control file. The system considers this is the default package type when `User` is not specified. If `User` is root ,`Type` defaults to `os`.

Secure Installer does not start processes automatically for OS packages as it does for user packages. You must start processes using Linux start-up scripts (for example, `rc.d`).

In the following example OS package, the tar-ball installs files to `/usr/local/lib` and `/usr/local/sbin`. `Type` defaults to `os`, therefore `User` defaults to root.

control file contents:
```
Package: vfirkl
Version: 1.0.2
```

Example OS package

```
|-- CONTROL
|   `-- control (see below)
`-- usr
`-- local
|-- lib
|   |-- libvfirkl.so -> libvfirkl.so.1
|   |-- libvfirkl.so.1 -> libvfirkl.so.1.1.3
|   `-- libvfirkl.so.1.1.3
`-- sbin
`-- rklload
```

osflash Packages

osflash packages extract relative to `/mnt/flash`. osflash packages be signed by the OS signer, and have the `Type` field in the package control file set to `osflash`. Once extracted, the package tarball is removed from the system (since flash package types are only required to be extracted once). However, the package control information remains, so flash packages can be updated and/or removed like other packages.

In the following example, the package tar-ball installs files to `/mnt/flash/system/images/ctlsl2`.
`Type` must be `osflash`, and `User` defaults to root.

control file contents:

```
Package: ctls
Version: 1.0.3
Type: osflash
```

Example osflash package

```
|-- CONTROL
|   `-- control (see below)
`-- system
`-- images
`-- ctlsl2
|-- 320x240
|   |-- blueled.png
|   |-- greenled.png
|   |-- redled.png
|   `-- yellowled.png
|-- 480x272
|   |-- blueled.png
|   |-- greenled.png
|   |-- redled.png
|   `-- yellowled.png
|-- 800x480
|   |-- blueled.png
|   |-- CTLSV.png
|   |-- greenled.png
|   |-- redled.png
|   `-- yellowled.png
`-- Decker.ttf
```

lowlayers Packages

lowlayers packages extract to a temporary location for processing. lowlayers packages must be signed by the OS signer, have the `Type` field in the control file set to `lowlayers`. Once processing completes, the package is removed. Table 5 lists allowable contents for lowlayers packages.

Table 5 Acceptable lowlayers Package Files

File name	Description
uuImage uuImage.p7s	Used to update the Linux kernel and signature.
uuImage.patch uuImage.p7s	uuImage.patch runs first to patch the existing uuImage file, creating a new uuImage file. uuImage.p7s is the signature file for this new uuImage file. The signature is verified and uuImage installs to verify that the patch applied successfully. Full uuImage updates or patches can be contained in lowlayers packages; not both.
ext-initramfs.img ext-initramfs.img.p7s	Used to update the initramfs and signature.
ext-initramfs.img.patch ext-initramfs.img.p7s	ext-initramfs.img.patch is run to patch the existing ext-initramfs.img file, creating a new ext-initramfs.img file. ext-initramfs.img.p7s is the signature file for the new ext-initramfs.img file. The signature is verified and ext-initramfs.img installs to verify that the patch applied successfully. Full ext-initramfs.img updates or a patches can be contained in lowlayers packages; not both.
*.cib	Used to update the CIB file.
cib.*	Used to update the alias CIB file.
sbi.*	Used to update the SBI file.
ssbi.*	Used to update SSBI file.
vault.*	Used to update the Vault file.
u-boot.*	Used to update u-boot file.

user Packages

user is the default package type when the `User` field is specified and is not root. If present, the `Type` field in the package control file is set to `user`. user packages extract to the User home account in the DRAM file system. These packages contain the executable and data files required to run user programs. Directory structure is not important, as all paths are relative to the User home account.

Set the `User` field in the control file to determine the user for the installation. This also determines the location and ownership of the files when extracted to DRAM; App signed files use usr1 through usr16. System signed files use sys1 through sys16. usr1 is the default user.

The `Group` field determines the group ownership. This defaults to the user private group associated with the package (for example, `Group: usr1` for User usr1). Use `Group: users` or `Group: system` to share files with other users of the same application group (that is, application users or system application users, respectively). Use `Group: share` to share the files in the package with all users. Multiple packages are required to install files requiring different group ownership.

User library files can also be in User packages, but to be picked up automatically must be placed in a lib directory in the user home account (that is, a lib directory in the install package).

Secure Installer automatically starts any processes listed in the start file in the CONTROL directory.

In the following example user package, the package tar-ball installs the testapp file in the user home account `/home/usr1`, and creates the directory `lib` that contains a library and symlinks.

control file contents:	`Package: testapp`
	`Version: 1.0.1`
	`Type: user`

start file contents:	`testapp arg1 "This is arg2"`

env file contents:	`[global] PLATFORM=<terminal>`
	`[testapp] NAME=testapp VERSION=1.0.1`
	For example:
	`[global] PLATFORM=Mx9`
	`[testapp] NAME=testapp VERSION=1.0.1`

Example user package

```
|-- CONTROL
|   |-- control
|   |-- env
|   `-- start
|-- lib
|   |-- libtest.so -> libtest.so.1
|   |-- libtest.so.1 -> libtest.so.1.0.1
|   `-- libtest.so.1.0.1
`-- testapp
```

userflash Packages

userflash packages are App signer packages to download large data files to flash. userflash packages cannot contain executable files. This package is only extracted to flash once, and then it is deleted.

Set the `Type` field in the package control file to `userflash`. Set the `User` and `Group` fields the same as user packages. File ownership and location are determined on extraction by using a symlink reference to the user home account in flash.

In the following userflash package example, the package tar-ball installs a theme in the flash directory for the user home account, which is the symlink `/home/usr1/flash` (a pointer to `/home/usr1/userdata/usr1`). `User` defaults to `usr1`. `Group` defaults to the user private group, also `usr1`.

control file contents:	`Package: testappflash`
	`Version: 1.0.1`
	`Type: userflash`

Example userflash package

```
|-- CONTROL
|   `-- control

`-- themes
`-- themes
|-- Contemporary
|   |-- background.png
|   |-- banner_bg.png
|   |-- button_100d_normal.png
|   |-- button_100d_pressed.png
|   |-- button1.png
|   |-- button1_pressed.png
|   |-- button_ok_normal.png
|   |-- button_ok_pressed.png
|   |-- customer_logo.png
|   |-- fonts
|   |   |-- Vera_Bold.ttf
|   |   |-- Vera_Italic.ttf
|   |   `-- Vera.ttf
|   |-- keypad_numeric_normal.png
|   |-- keypad_numeric_pressed.png
|   |-- keypad_textbox_bg.png
|   |-- metrics.fpt
|   |-- Pen.png
|   |-- preview.png
|   |-- Thumbs.db
|   |-- verifone_logo.png
|   `-- wip04-131633cd-2.wav
`-- theme.txt
```

service Packages

Service libraries must be accessible by both user and svcmgr processes, which typically run as a system user. svcmgr looks in a single location for all services: `/usr/local/lib/svcmgr`. The service package installs to this directory.

Set the `Type` field in the package control file to `service`. The `User` and `Group` fields are the same as the user packages, and determine file ownership once extracted. Note however that the default `Group` value for service packages is `share`. service packages are authenticated and extracted to DRAM on every boot-up.

In the following service package example, the tar-ball installs libraries in `/usr/local/lib/svcmgr`. `User` is set to `sys2`. `Group` defaults to `share`.

control file contents:

```
Package: vfiservices
Version: 1.0.5
User: sys2
Type: service
```

Example service package

```
|-- CONTROL
|   `-- control
|-- libsvc_ecrcom.so
|-- libsvc_ethernet.so
|-- libsvc_event.so
|-- libsvc_kbd.so
|-- libsvc_led.so
|-- libsvc_msr.so
|-- libsvc_netloader.so
|-- libsvc_networkapps.so
|-- libsvc_powermngt.so
|-- libsvc_rfcr.so
|-- libsvc_security.so
|-- libsvc_serialcom.so
|-- libsvc_sigcap.so
|-- libsvc_sysinfo.so
|-- libsvc_utility.so
`-- libsvc_vp2.so
```

config Packages

This App signer package downloads global configuration files (that, is config files that must be accessible to many users). config packages extract to the directory `/etc/config`, and are authenticated and extracted to DRAM on every boot up. Set the `Type` field in the control file to `config`. The `User` and `Group` fields are the same as the user packages, and determine file ownership once extracted.

In the following config package example, the tar-ball installs config files in `/etc/config`. `User` is set to `sys2`. `Group` is set to `share`.

control file contents:

```
Package: wdesvcconfs
Version: 1.0.0
User: sys2
Type: config
Group: share
```

Example config package

```
|-- CONTROL
|   `-- control
|-- lighttpd
|   `-- lighttpd.conf
`-- php
`-- php.ini
```

flashconfig Packages

flashconfig packages are similar to config package types, but because it is a flash package, it is authenticated and the contents extract to flash only once. The contents extract to `/mnt/flash/etc/config`, and the package is removed. Use flashconfig packages for config files that you want to download and change afterwards (for example, write to).

Set the `Type` field in the package control file to `flashconfig`. The `User` and `Group` fields are the same as the user packages, and determine file ownership once extracted.

vss Packages

Use vss packages to download and install VSS scripts.

NOTE VSS scripts are authenticated using `filetypes A, B, C,` through `P` for users usr1–usr16, respectively. The signer certificate must have the same filetype enabled.

Because VSS scripts cannot be modified by the User but must be readable by that User, they are owned by root with the group ownership set to the User private group of that User. vss packages extract to `/etc/vss/<username>` (where, `<username>` is determined by the package user: usr1, usr2, and so on), and are authenticated and extracted to DRAM on every boot-up.

Set the `Type` field in the control file to `vss`.

In the following example vss package, the tar-ball installs vss script files in `/etc/vss/usr1`. `User` defaults to `usr1`.

control file contents:

```
Package: ldckkeys
Version: 1.0.0
Type: vss
```

Example vss package

```
|-- CONTROL
```

```
|   `-- control
`-- ldckkeys.vso
```

unsigned Packages

unsigned packages always extract to the User's flash directory. They are not signed or authenticated. They contain user data extracted by Secure Installer to allow these packages to download in the download file with signed installer packages.

NOTE unsigned packages are not signed and cannot be inside a bundle, which is always signed. They must be at the top level of the download file, which can also contain signed bundles.

The compressed tar-ball must contain a CONTROL directory and control file. The `Name`, `Version`, and `Type` fields are required. Set the `Type` field to `unsigned`. The `User` and `Group` fields are the same as the user packages, and determine file ownership once extracted.

NOTE unsigned packages can only be installed by the User they are targeted for. They must be downloaded by the user application. Do not use sysmode.

Secure Installer does not keep a log of unsigned packages in the installed package list.

In the following example unsigned package, the files install in`/usr1`.

control file contents:	`Package: verafont`
	`Version: 1.0.1`
	`Type: userfont`
	`Group: share`

Example unsigned package

```
|-- CONTROL
|   `-- control
`-- themes
|-- Contemporary
|   |-- spectrum_bottom_off.png
|   `-- spectrum_bottom.png
`-- theme.txt
```

userfont Packages

This app signer package installs a system-wide font to flash. The `Type` field in the package control file must be set to `userfont`. The `User` and `Group` fields are the same as the `user` packages in that they determine ownership of the files on extraction. The default Group for the `userfont` packages is the `share` group. userfont packages extract to `/mnt/flash/system/fonts` only once, and then it is deleted.

In the following example userfont package, the files install in `/mnt/flash/install/fonts`. `User` is set to `usr1`.

control file contents:	`Package: unsigned`
	`Version: 1.0.0`
	`User: usr1`
	`Type: unsigned`

Example unsigned package

```
-- CONTROL
| `-- control
|-- Vera_Bold.ttf
|-- Vera_Italic.ttf
`-- Vera.ttf
```

patch Packages

Use patch packages to patch or update installed packages. The result is a new version of an existing package.

Set the `Category` field in the control file to `patch`. The `Type`, `Name`, and `Version` fields of the patch package must match those fields for the target installed package. The `Name` and `Version` of the updated package are available to Secure Installer after the patch is applied.

Along with the control file in the CONTROL directory, patch packages must contain the following two files at the top level:

- A patch file generated using bsdiff
- A signature file update for the new package

In the following example patch package, the myapp-1.0.0.tar and signature file myapp-1.0.0.tar.p7s. files update with myapp-1.0.1.tar and signature myapp-1.0.1.tar.p7s. `User` is set to `user`.

control file contents:	`Package: myapp`
	`Version: 1.0.0`
	`Type: user`
	`Category: patch`

First, the patch was generated using:

```
bsdiff myapp-1.00.tar myapp-1.01.tar myapp-1.01.patch
```

When generating a patch file with bsdiff, use the exact package file as installed on the target rather than new one, as it may not be exactly what was previously released (for example, if the UID/GID information is different).

Example patch package

```
|-- CONTROL
|   `-- control
|-- myapp-1.0.1.patch
|-- myapp-1.0.1.tar.p7s
```

Install Bundles

All install packages (and their associated p7s signature files) must be contained in a signed bundle. A bundle can contain multiple packages. Bundles allow sets of packages to download at the same time, and ties these packages to the optional remove file.

The install bundle is a compressed tar-ball. Use gzip or bzip2 compression. Bundles must be signed and be accompanied by a p7s signature file. The signature file must have the same file name as the bundle file with a .P7S or .p7s extension.

To authenticate bundles, the `filetype` and `user` must be specified. The bundle must contain the control file in `CONTROL/control`, which must specify the `Name`, `Version`, and `User`.

Since a bundle is authenticated with a specified `filetype`, a bundle can only contain install packages for a single user. Other user packages are not allowed within that bundle. Download files can contain multiple bundles for different users.

An install bundle can contain one or more of the following.

- Install packages and associated signature files
- Signer certificates contained in the crt (or CRT) sub-directory.

  Since install packages are signed and require a certificate for authentication, signer certificates cannot be contained within a signed package. Certificates found in the install bundle are processed before the install packages.
- The package remove file

NOTE Remove files are only required in special circumstances, such as when completely changing the installed application set. If the download is simply updating existing packages or adding packages, the remove file is not required.

The package remove file removes packages and is processed before any install packages. Once the install bundle file is in flash (assuming it is signed properly), it installs (immediately or on the next boot up). It is safe to process the remove file before the packages are processed.

The remove file is tied to a particular set of install packages by being in the same signed bundle. For example, if a remove file contains a command to remove the entire OS, then the bundle must also contain a new set of OS type packages to install. The user and system signers cannot remove packages installed by other signers. The OS signer can remove all installed packages.

Bundle Control File

The bundle control file is a text file that contains information relating to the bundle. The format is individual lines in the form *Field:value*. Place the bundle control file in the bundle CONTROL directory. The fields listed in Table 6 are supported.

Table 6 Bundle Control File Fields

Field	Max Length	Description
Name	24	Name of the bundle (required). The Package field can alternatively be used. The Name and Package labels are inter-changeable.
Version	16	Bundle version (required).
User	8	Bundle user. This field is required for all bundle types, except OS bundles. Defaults to OS if not specified.
Category	16	Bundle category (optional). • fs – file system content, including the standard OS, system, and user packages. (default). Defaults to fs if not specified. • patch – update to installed bundle By definition the fs bundle is complete. patch bundles are an update to an existing bundle but do not contain the complete content for that bundle, and therefore require that a previous version of the bundle is already installed on the system.
SrcVersion	16	Contains the version of the bundle being patched. This field is required in patch bundles.
Date	10	Bundle creation date in the format: YYYY-MM-DD (optional).
Option	64	Additional information (optional).
Type	16	Set this type field value to signed if used (optional). Defaults to signed if not specified.

Example Bundle Control File

control file contents:

```
Name: sec-rfsbundle
Version: 1.0.5
User: os
Date: 2012-05-08
```

patch Bundles

While fs (file system) bundles are considered complete and entirely replace previously installed versions, patch bundles are considered an update to existing bundles, and must only contain the differences between the new bundle and bundle being updated. patch bundles always require that a previous version of the bundle is already installed on the system.

There are two options for updating existing installed bundles:

- Download the entire fs bundle.
- Download only changed packages in the bundle.

  This option has a number of possibilities:

- When using a patch bundle to download changed packages in a bundle, either complete packages are downloaded or patch packages (where patch packages just download a binary diff file between the old and new package versions).

The following example v1.0.0 bundle contains three packages.

control file contents:	`Package: testbundle`
	`Version: 1.0.0`
	`User: usr1`

Example patch bundle

```
|-- CONTROL
| `-- control
|-- crt
| `-- appsigner.crt
|-- testapp1-1.1.0.tar
|-- testapp1-1.1.0.tar.p7s
|-- testapp2-1.0.2.tar
|-- testapp2-1.0.2.tar.p7s
|-- testconfig-1.0.0.tar
`-- testconfig-1.0.0.tar.p7s
```

If one of the testapp packages is updated, the three options for version 1.0.1 of this bundle are as follows.

Example fs option

control file contents:	`Package: testbundle`
	`Version: 1.0.1`
	`User: usr1`

```
|-- CONTROL
| `-- control
|-- crt
| `-- appsigner.crt
|-- testapp1-1.1.0.tar
|-- testapp1-1.1.0.tar.p7s
|-- testapp2-1.0.3.tar
|-- testapp2-1.0.3.tar.p7s
|-- testconfig-1.0.0.tar
`-- testconfig-1.0.0.tar.p7s
```

Example patch bundle option

control file contents:

```
Package: testbundle
Category: patch
Version: 1.0.1
SrcVersion: 1.0.0
User: usr1
```

```
|-- CONTROL
| `-- control
|-- crt
| `-- appsigner.crt
|-- testapp2-1.0.3.tar
`-- testapp2-1.0.3.tar.p7s
```

Example binary patch bundle option

control file contents:

```
Package: testbundle
Category: patch
Version: 1.0.1
SrcVersion: 1.0.0
User: usr1
```

```
|-- CONTROL
| `-- control
|-- crt
| `-- appsigner.crt
|-- testapp2-1.0.3-patch.tar
`-- testapp2-1.0.3-patch.tar.p7s
```

Remove File

The optional package remove file in an install bundle is a simple text file that contains commands, one per line, interpreted by Secure Installer. If using the remove file, place it at the top-level of the compressed bundle file–not in a sub-directory and not in the package tar-ball.

If a remove file is required it is part of the bundle along with the signed packages. Because the bundle is signed, the remove file and set of packages with it cannot be separated.

Only the following commands are supported. Certain commands are only available to the OS signer.

- removeall

  ```
  removeall [os | sys | user ] [<user>]
  ```

  For the OS signer:

  - If all optional arguments are omitted, *all* installed packages are removed.
  - If the value for the first argument is `os`, all OS packages are removed.

- If the value for the first argument is `sys` or `user` and the second argument is omitted (for example, `removeall sys` or `removeall user`), all system or user signed installed packages, respectively are removed.
- If the value of the first argument is `sys` or `user` and the second argument is specified (for example, `removeall sys sys1` or `removeall user usr1`), all packages for the specified user are removed.

Other users can only remove packages they own. The `removeall` arguments are ignored.

- removepkg

  ```
  removepkg <Package> [<Version>]
  ```

  Only removes the specified package from a bundle containing a remove file. Only use `removepkg` to remove packages between different versions of the same bundle. If `Version` is specified, only that version of the specified package is removed (if found). Any signer (OS, system, or user) can only remove packages installed by that signer.

- removeconfig

  ```
  removeconfig <user>
  ```

  Removes legacy config files (for example, config.usr1) for the specified user.

- removelogs

  ```
  removelog <user>
  ```

  Deletes the contents of the log directory for the specified user.

- removeshare

  ```
  removeshare
  ```

  Deletes the contents of the common flash share directory.

- removebundle

  ```
  removebundle <Bundle> [<Version>]
  ```

  Deletes the specified bundle. All packages in that bundle are removed. If `Version` is specified, then only that version of the specified bundle is removed (if installed). Any signer (OS, system, or user) can only remove a bundle installed by that signer.

- removepkg

  ```
  removeshare
  ```

  Deletes the contents of the common flash share directory.

Download Files

Install bundles and their associated signatures must be presented to Secure Installer (that is, they must download to the user directory) in a single compressed tar-ball. The download file can be a plain tar archive or a compressed tar-ball. Either gzip or bzip2 compression can be used. Download files only contain compressed install bundles and their associated signature files. Do not place single App package or install package files at the top level of the download file.

Installed Packages

Users must use the libsecins call Secins_read_pkglist_entry() to get the installed package list info.

Create Install Packages and Bundles

The process to create and install packages and bundles is:

1. Generate the install package.
2. Generate the install bundle.
3. Generate the download file.

Generate the install package

Place the files in a directory structure as they should appear on the device: relative to the user home account for sys or user packages, or root (/) for OS packages. Add the CONTROL directory with the control file. Add any other required files (for example, start, grsec, or env files).

In the following example, the package contains the testapp executable and a shared library for the lib directory.

control file contents:

```
Package: testapp
Version: 1.02
Type: user
```

start file contents:

```
testapp arg1 "arg 2" argument3
```

grsec file contents:

```
[testapp]
   /dev/console     rw
   /dev/ttyAMA0     rw
   /dev/urandom     rw
   +CAP_SYS_TIME
```

env file contents:

```
[global]
   DEBUG=on
[testapp]
   COMPORT=/dev/ttyAMA0
```

Example install package

```
package
|-- CONTROL
|   |-- control
|   |-- env
|   |-- grsec
|   `-- start
|-- lib
|   |-- libdes.so -> libdes.so.1
|   |-- libdes.so.1 -> libdes.so.1.0.1
|   `-- libdes.so.1.0.1
`-- testapp
```

Generate the package tar ball

```
$ cd package
$ tar cf ../testapp-1.02.tar * $ cd ..
```

Sign the package tar ball

In a production environment signer cards must be used for signing. In a development environment, with test keys the vfisigner tool can be used as follows:

```
$ vfisigner --cert appsignerA1.crt --key appsignerA1.key testapp-1.02.tar
Signing testapp-1.02.tar ...
Writing testapp-1.02.tar.p7s ...
testapp-1.02.tar signed ok
$
```

`appsignerA1.key` and `appsignerA1.crt` are the signer key and signer certificate, respectively. The following two files then comprise the signed install package: `testapp-1.02.tar` and `testapp-1.02.tar.p7s`

Generate the install bundle

The bundle is a compressed tar ball of signed packages, but can also include the signer certificate in a crt directory and the remove file. In this example there is no remove file:

```
bundle
|-- crt
|   `-- appsignerA1.crt
|-- testapp-1.02.tar
`-- testapp-1.02.tar.p7s
```

This example includes a remove file:

```
bundle
|-- crt
|   `-- appsignerA1.crt
|-- testapp-1.02.tar
|-- testapp-1.02.tar.p7s
`-- remove
```

Note that the remove file is always part of the bundle, and not part of the package. The CONTROL directory or control file are not necessary if the User name is included in the bundle filename. Since the package user is not specified in the control file, the default User is usr1. In this example, the bundle contains a single package. In the bundle directory, use the following command to create the bundle:

```
$ cd bundle
$ tar czf ../USR1.testapp-1.02.tgz *
$ cd ..
```

Sign the bundle

```
$ vfisigner --cert appsignerA1.crt --key appsignerA1.key
USR1.testapp-1.02.tgz
```

```
Signing USR1.testapp-1.02.tgz ...
Writing USR1.testapp-1.02.tgz.p7s ...
USR1.testapp-1.02.tgz signed ok
```

For this example, the USR1.testapp-1.02.tgz and USR1.testapp-1.02.tgz.p7s files comprise the signed bundle.

Generate the download file

The download file contains the signed bundles.

```
$ tar cvzf testapp-1.02-dl.tgz USR1.testapp-1.02.tgz
USR1.testapp-1.02.tgz.p7s
```

Logs

Secure Installer maintains a log file at `/mnt/flash/install/secins.log`. All log entries have a timestamp. The following events are logged:

- Processing download files. File name is logged.
- Successful package installation. Package name and version are logged.
- Failed package installation. Package name, version, and failure cause are logged.
- Signer certificate added successfully. Certificate name and serial number are logged.
- Signer certificate fails to install. Certificate name, serial number, and failure cause are logged.
- Package removal successful. Package name and version are logged.
- Package extraction to DRAM file system failed. Package name, version, and failure cause are logged.
- Secure Installer fails to start application. Application name and fail cause logged.

Sysmode Application

Secure Installer deals with install packages, but does not make any distinction between different packages within a particular signer group, for example, all OS packages are treated alike, as are all sys and user packages. Secure Installer does not differentiate between different user or system processes. It starts applications as directed in the control files.

Secure Installer must first start the sysmode application, if there are no user applications to start. Similarly, if sysmode is manually started when user applications are running (on VX 520 terminals: press the 1, 5, and 9 keys; on Ux300 terminals: press the recessed sysmode button on the back of the device), those applications must be stopped and sysmode started. To facilitate this, the sys4 system user is reserved for sysmode. Secure Installer treats this user differently. On start up when starting installed system applications, sys4 applications are not started automatically. If there are no user applications to start, Secure Installer starts all sys4 applications (if present).

When the sysmode is manually started, Secure Installer stops all running user applications from users usr1–usr16, and then starts the sys4 applications.

Main sysmode menu access is restricted. There are four user accounts defined:

- Supervisor
- Level 1
- Level 2
- Maintenance

On productions units the first time an operator logs in, they are prompted to enter the default password and then enter a new password. Passwords must be a minimum of five digits, and a maximum of 8 digits. During development, each user account can be accessed using the default password 166831. If the user cannot log in, cancel is the only available option.

See

Secure Installer Functions

Once the system is up, Secure Installer remains running and provides a message interface to allow user applications to access certain root services (for example, reboot). The libsecins.so shared library allows access to the functionality provided. Function prototypes, error codes, and so on are in libsecins.h.

This section describes the current function calls. Unless otherwise noted, return values are 0 on success or an appropriate error code (see Error Codes).

Secins_install_pkgs()

Installs packages from the download file to `/mnt/flash/install/dl`.

Prototype

```
int Secins_install_pkgs(int *reboot_reqd);
```

Parameters

reboot_reqd	If `reboot_reqd` is not null, Secure Installer sets it to 1 if a reboot is required after installation completes (for example, if an OS package was installed); otherwise it is set to 0.

Secins_reboot()

Reboot the device. This call first send SIGTERM to all user processes, followed by SIGKILL.

Prototype

```
int Secins_reboot(void);
```

Secins_start_user_apps()

Used by sysmode to start user applications. This call stops sysmode sys4 processes before starting user processes.

Prototype

```
int Secins_start_user_apps(void);
```

Secins_start_app()

Used by user applications to start other user applications.

Prototype

```
int Secins_start_app(const char *file);
```

Parameters

file	Executable to start.

Secins_read_pkglist_entry()

Called recursively to retrieve the list of installed packages, which returns in pkginfo. The SecinsPkgInfo is defined in libsecins.h.

Prototype

```
SecinsPkgInfo *Secins_read_pkglist_entry(SecinsPkgInfo *pkginfo,
int pkginfosize);
```

Parameters

pkginfosize	Set to the size of the pkginfo buffer.
SecinsPkgInfo	Defined in libsecins.h as follows:

```
#define MAX_PKG_NAME_LEN      24
#define MAX_PKG_VERSION_LEN   16
#define MAX_PKG_USER_LEN      8
#define MAX_PKG_TYPE_LEN      16
#define MAX_SIGNER_LEN        4
// os, sys, user typedef struct
{
char name[MAX_PKG_NAME_LEN + 1];
char version[MAX_PKG_VERSION_LEN + 1];
char user[MAX_PKG_USER_LEN + 1]; char
signer[MAX_SIGNER_LEN + 1]; char
type[MAX_PKG_TYPE_LEN + 1];
} SecinsPkgInfo;
```

where,

- name is the package name
- `version` is the package version string
- `user` is the package user usr1–usr16 or sys1–sys16 (note that sys4 is reserved)
- `signer` is the package signer OS, sys, or user
- type is package type OS, user, userflash, and so on

Return Values

On success, pgkinfo returns. NULL returns when no more entries are available.

Example

This example prints the installed package versions.

```
static void print_pkg_versions(void)
{
    SecinsPkgInfo pkginfo;
    fprintf(stderr, "Package Versions\n");
    while (Secins_read_pkglist_entry(&pkginfo, sizeof pkginfo) != NULL)
    {
       fprintf(stderr,
       "%s %s %s %s %s\n", pkginfo.name, pkginfo.version, pkginfo.user,
       pkginfo.signer, pkginfo.type);
    }
    fprintf(stderr, "\n");
}
```

Secins_close_pkglist()

Closes the list of installed packages. Only call Secins_close_pkglist() when Secins_read_pkglist_entry() is not being called repeatedly until NULL returns. For example, if the `while` loop in the previous example terminated prematurely.

Prototype

```
void Secins_close_pkglist(void);
```

Secins_strerror()

Returns a pointer to the error message associated with the err non-zero error code, which must have returned from a Secure Installer call. Use Secins_strerror() when a libsecins call returns non-zero for a verbose message associated with the failure.

Prototype

```
const char *Secins_strerror(int err);
```

Secins_share_gid()

Returns the group ID (GID) for the share group. All users on the system are a member of this share group.

Prototype

```
gid_t Secins_share_gid(void);
```

Secins_system_gid()

Returns the GID for the system group. All system application users are a member of this system group.

Prototype

```
gid_t Secins_system_gid(void);
```

Secins_users_gid()

Returns the GID for the users group. All application users usr1–usr16 are a member of this users group.

Prototype

```
gid_t Secins_users_gid(void);
```

Secins_user_app()

Returns the application user usr1–16.

Prototype

```
unsigned int Secins_user_app(void);
```

Return Values

Returns 1 if the caller is an application user usr1–usr16; otherwise returns 0.

Secins_sys_app()

Returns the system user sys1–sys16.

Prototype

```
unsigned int Secins_sys_app(void);
```

Return Values

Returns 1 if the caller is a system application user; otherwise returns 0.

Secins_sysmode_share_gid()

Returns the sysmode share GID for the calling user. For each application user usr1–usr16 a special group is created that contains the user and the sysmode sys4 user as members. For other users, the current GID returns.

Prototype

```
gid_t Secins_sysmode_share_gid(void);
```

Secins_config_file_share_gid()

Returns the sysmode share GID for the specified config file.

Prototype

```
gid_t Secins_config_file_share_gid(const char *configfile);
```

Parameters

configfile	Specified config file: config.usr1, config.usr2, config.sys1, config.network, and so on.

Return Values

For config.usr1–config.usr16, the sysmode share GUD returns. For other config files, the current GID returns.

Users/Groups Functions

The application manager must be able to dynamically add users and groups to the system. Secure Installer assigns a range of UIDs and GIDs to each user, allowing them to control users and groups. Any user can only use the UIDs and GIDs assigned to it. Use Secins_get_uid_gid_range() to obtain the range to assign.

In the following, users usr1–usr16 are referred to as the primary user. The users and groups assigned to each primary user are referred to as secondary users.

Secins_get_uid_gid_range()

Retrieves the range of users and groups assigned to the calling primary user.

NOTE

All specified UIDs and GIDs for the calls in this section must be in this range.

Prototype

```
int Secins_get_uid_gid_range(UID_GID_RANGE *range, int rangesize);
```

Parameters

rangesize	The size of the assigned range returned in the range struct.

Example

The UID_GID_RANGE struct is defined in libsecins.h:

```
typedef struct
{
   int start_uid;
   int num_uids;
   int start_gid;
   int num_gids;
} UID_GID_RANGE;
```

where,

- `start_uid` is the start UID in the assigned range.
- `num_uids` is the number of UIDs assigned. So the allowed UID is `start_uid`, `start_uid` + 1, `start_uid` + 2, …, `start_uid` + `num_uids` - 1.
- `start_gid` is the start GID in the assigned range.
- `num_gids` is the number of GIDs assigned. So the allowed GIDs is `start_gid`, `start_gid` + 1, `start_gid` + 2, …, `start_gid` + `num_gids` - 1.

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_add_group()

Adds a group with the specified `gid` and `groupname`. Only the primary user can use this call.

Prototype

```
int Secins_add_group(int gid, const char *groupname);
```

Parameters

gid	Must be in range assigned to the primary user returned by Secins_get_uid_gid_range().
groupname	Group name: 16 chars maximum.

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_add_user()

Adds a user with the specified `uid`, `username`, and `gid`. The specified UID and GID must be in range assigned to the primary user with Secins_get_uid_gid_range(). The GID must already exist (that is, must have been added using Secins_add_group()). Only the primary user can use this call.

Prototype

```
int Secins_add_user(int uid, const char *username, int gid);
```

Parameters

username	Primary user name; 16 chars maximum.

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_add_group_member()

Adds the specified UID to the specified GID. The specified UID and GID must be in range assigned to the primary user using Secins_get_uid_gid_range(). The referenced user and group must already exist. Only the primary user can use this call.

Prototype

```
int Secins_add_group_member(int gid, int uid);
```

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_remove_user()

Removes the specified UID. The specified UID must exist and be in range assigned to the primary user using Secins_get_uid_gid_range(). Only the primary user can use this call.

Prototype

```
int Secins_remove_user(int uid);
```

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_remove_group()

Removes the specified GID. The specified GID must exist and be in range assigned to the primary user using Secins_get_uid_gid_range(). Only the primary user can use this call.

Prototype

```
int Secins_remove_group(int gid);
```

Return Values

Returns 0 on success; otherwise an error code returns.

Secins_start_app_uid()

Starts an application (cmdline) with specified UID and GID. The specified UID and GID must exist and be in range assigned to the primary user using Secins_get_uid_gid_range(). Only the primary user can use this call. The executable file specified in cmdline must be owned by the primary user.

Prototype

```
int Secins_start_app_uid(pid_t *app_pid, const char *cmdline, int uid,
int gid);
```

Parameters

app_pid	The process ID of the application.
cmdline	The command line to use to start the application.
uid	The app runs with this user ID.
gid	The app runs with this group ID.

Return Values

Returns 0 on success; otherwise one of the ERROR_SECINS error codes.

Error Codes

Error Code	Value	Description
ERROR_SECINS_NO_MEMORY	100	Out of memory.
ERROR_SECINS_MKDIR	101	Failed to make directory.
ERROR_SECINS_FILE_AUTH	102	Failed to authenticate package.
ERROR_SECINS_MOUNT_RFS	103	Failed to mount file system.
ERROR_SECINS_INVALID_INIT_FILE	104	System init file not found.
ERROR_SECINS_PATHNAME_TOO_LONG	105	Pathname too long
ERROR_SECINS_CHROOT	106	Switch root operation failed
ERROR_SECINS_EXEC_INIT	107	Failed to run the system init file.
ERROR_SECINS_OPENDIR	108	Failed to open directory.
ERROR_SECINS_RENAME_DIR	109	Failed to rename directory.
ERROR_SECINS_CHMOD	110	Failed to set file permissions.
ERROR_SECINS_CHOWN	111	Failed to set file ownership.
ERROR_SECINS_FILE_CREATE	112	Failed to create file.
ERROR_SECINS_FILE_OPEN	113	Failed to open file.
ERROR_SECINS_FILE_WRITE	114	Failed to write file.
ERROR_SECINS_NO_SIG	115	No package signature file.
ERROR_SECINS_NO_PKG	116	No matching package file with signature.
ERROR_SECINS_INVALID_FILENAME	117	Invalid p7s signature filename.
ERROR_SECINS_INVALID_INSTALL	118	Installed package is invalid.
ERROR_SECINS_INVALID_INI_SECTION	119	Invalid ini file section.
ERROR_SECINS_MKNOD	120	Failed to create device node.
ERROR_SECINS_INVALID_INI_PROPERTY	121	Invalid ini file property.
ERROR_SECINS_NO_CONTROL_FILE	122	No control file in package.
ERROR_SECINS_INVALID_CONTROL_FILE	123	Invalid control file in package.
ERROR_SECINS_CREATE_CONTROL_FILE	124	Failed to install control file.
ERROR_SECINS_OLD_PKG_VERSION	125	Newer package already installed.
ERROR_SECINS_INVALID_PKG_TYPE	126	Invalid package type.
ERROR_SECINS_EXTRACT_PKG_CONTROL	127	Failed to extract package CONTROL directory.
ERROR_SECINS_EXTRACT_PKG	128	Failed to extract package archive.
ERROR_SECINS_INVALID_PKG	129	Invalid package.
ERROR_SECINS_PKG_LIST	130	Failed to read package contents.
ERROR_SECINS_PKG_FILENAME_TOO_LONG	131	Package filename too long.
ERROR_SECINS_INSTALL_CONTROL_FILE	132	Failed to install package control file.
ERROR_SECINS_INSTALL_PKG	133	Failed to install package.
ERROR_SECINS_INSTALL_SIG	134	Failed to install package signature.
ERROR_SECINS_PKG_NOT_FOUND	135	Incomplete package install.
ERROR_SECINS_ADD_GROUP	136	Failed to add group.
ERROR_SECINS_ADD_USER	137	Failed to add user.
ERROR_SECINS_ADD_SHADOW	138	Failed to add shadow password entry.
ERROR_SECINS_GET_USER	139	User not found.

Error Code	Value	Description
ERROR_SECINS_GET_GROUP	140	Group not found.
ERROR_SECINS_PROCESS_FORK	141	Unused
ERROR_SECINS_CHECK_GRSEC	142	grsecuity policy check failed.
ERROR_SECINS_START_GRSEC	143	Failed to enable grsecurity.
ERROR_SECINS_FORK	144	Failed to create child process.
ERROR_SECINS_GET_RESOURCE	145	Failed to get system resource limits.
ERROR_SECINS_SETSID	146	Failed to create new session.
ERROR_SECINS_START_LINE_TOO_LONG	147	Start file line too long.
ERROR_SECINS_START_LINE_INVALID	148	Start file line invalid.
ERROR_SECINS_START_ENTRY_UNKNOWN	149	Start file entry unknown.
ERROR_SECINS_EXEC_APP	150	Failed to start application.
ERROR_SECINS_MODE_APP	151	Start file entry not executable.
ERROR_SECINS_SYMLINK	152	Failed to create symlink.
ERROR_SECINS_INVALID_INSTALL_FILE	153	Invalid install archive type.
ERROR_SECINS_EXTRACT_INSTALL_FILE	154	Failed to extract download file.
ERROR_SECINS_INVALID_PKG_FORMAT	155	Invalid package file type.
ERROR_SECINS_AUTH_GID	156	Invalid filetype for authenticating.
ERROR_SECINS_ADD_CERT	157	Failed to add certificate.
ERROR_SECINS_PATCH_VERSION	158	Patch version does not match the installed package.
ERROR_SECINS_PATCH	159	Failed to patch package.
ERROR_SECINS_START_SVCSEC	160	Failed to start svcsec application.
ERROR_SECINS_INVALID_CERT	161	Invalid certificate file.
ERROR_SECINS_PUT_ENV	162	Failed to process package config file.
ERROR_SECINS_SET_USER	163	Failed to set user ID.
ERROR_SECINS_INVALID_PKG_CATEGORY	164	Invalid package category specified.
ERROR_SECINS_INVALID_CMD	165	Invalid command in remove file.
ERROR_SECINS_INVALID_BUNDLE_FILE	166	Invalid bundle file.
ERROR_SECINS_EXTRACT_BUNDLE_CONTROL	167	Failed to extract bundle file.
ERROR_SECINS_INVALID_BUNDLE_USER	168	Invalid bundle user specified.
ERROR_SECINS_INVALID_PKG_USER	169	Package user does not match bundle user.
ERROR_SECINS_NO_BUNDLE	170	No bundle file matching bundle p7s signature.
ERROR_SECINS_OPEN_SOCKET	171	Failed to open socket.
ERROR_SECINS_BIND_FAILURE	172	Failed to bind UNIX domain address.
ERROR_SECINS_UNKNOWN_MSG	173	Unknown Secure Installer message.
ERROR_SECINS_MSG_SND	174	Failed to send API response message.
ERROR_SECINS_MSG_RCV	175	Failed to read API message.
ERROR_SECINS_MSG_SIZE	176	API message too big.
ERROR_SECINS_RESP_SIZE	177	API response message too big.
ERROR_SECINS_INVALID_RESP	178	Invalid API response.
ERROR_SECINS_FILE_READ	179	Failed to read file.

Error Code	Value	Description
ERROR_SECINS_FILE_RENAME	180	Failed to rename file.
ERROR_SECINS_USER_APP_START	181	Failed to start application.
ERROR_SECINS_GET_CWD	182	Failed to get current working directory.
ERROR_SECINS_INVALID_PARAM	183	Invalid parameter specified.
ERROR_SECINS_NOT_ALLOWED	184	Permission denied.
ERROR_SECINS_SYSMODE_START	185	No Sysmode loaded.
ERROR_SECINS_INVALID_PKGLIST	186	Package list file invalid.
ERROR_SECINS_SOCK_LISTEN	187	Failed to setup API socket.
ERROR_SECINS_CONNECT_FAIL	188	Failed to connect to secins API socket.
ERROR_SECINS_PATCH_APPLIED	189	Low layer patch already applied.
ERROR_SECINS_RELOAD_GRSEC	190	Failed to reload the grsecurity policy.
ERROR_SECINS_CREATE_GRSEC_FILE	191	Failed to create grsec policy file.
ERROR_SECINS_GRSEC_SUBJECT	192	Subject in grsec file does not exist.
ERROR_SECINS_INVALID_POLICY_FILE	193	Invalid grsec policy file.
ERROR_SECINS_INVALID_OBJECT	194	Object in grsec file invalid or not allowed.
ERROR_SECINS_INVALID_OBJECT_MODE	195	Object mode in grsec file not allowed.
ERROR_SECINS_NO_POLICY_FILE	196	System grsec policy file not found.
ERROR_SECINS_INVALID_CAPABILITY	197	Capability in grsec file not allowed.
ERROR_SECINS_OBJECT_NOT_HIDDEN	198	Object in grsec file must be hidden.
ERROR_SECINS_OBJECT_NOT_FOUND	199	Required object in grsec file not found.
ERROR_SECINS_FILE_CAPS	200	Failed to add file capabilities.
ERROR_SECINS_SUBJECT_MODE	201	Invalid subject mode in grsec file.
ERROR_SECINS_REMOVE_USER	202	Failed to remove user.
ERROR_SECINS_REMOVE_GROUP	203	Failed to remove group.
ERROR_SECINS_PID_NOT_FOUND	204	Process does not exist.
ERROR_SECINS_SIGNAL_FAIL	205	Failed to send signal.
ERROR_SECINS_LOAD_VSS	206	Failed to load VSS script to vault.
ERROR_SECINS_UNSIGNED_PKG_USER	207	Unsigned package being installed by invalid user.
ERROR_SECINS_PATCH_AUTH	208	Failed to authenticate patched package.
ERROR_SECINS_EXTRACT_PATCH_CONTROL	209	Failed to extract control file from patched package.

CHAPTER 4

System Mode for V/OS Terminal

Use the built-in GUI on V/OS terminals to perform maintenance tasks. Refer to the reference manual for your terminal for complete menu descriptions.

Entering System Mode

With an application loaded, use the procedure in the hardware guide for your terminal to enter System Mode.

NOTE Before entering System Mode and selecting the function(s) to perform, verify that the V/OS terminal is installed as described in the installation guide for your terminal. Ensure that the terminal is connected to a power source, connected to the network, and is turned on.

Exiting System Mode

After successful completion, some operations automatically exist System Mode and reboot the terminal. To manually exit System Mode and restart the terminal, press the Reboot button on the System Mode idle screen.

CHAPTER 5

Multi-Application Environment

To create a multi-application solution, the V/OS terminal operating system provides primitives, but does not attempt to solve higher level multi-application issues such as device sharing (that is, the MSR, smartcard, and communication ports). Write the application architecture to work either in cooperation with the OS or to use a manager to control resources.

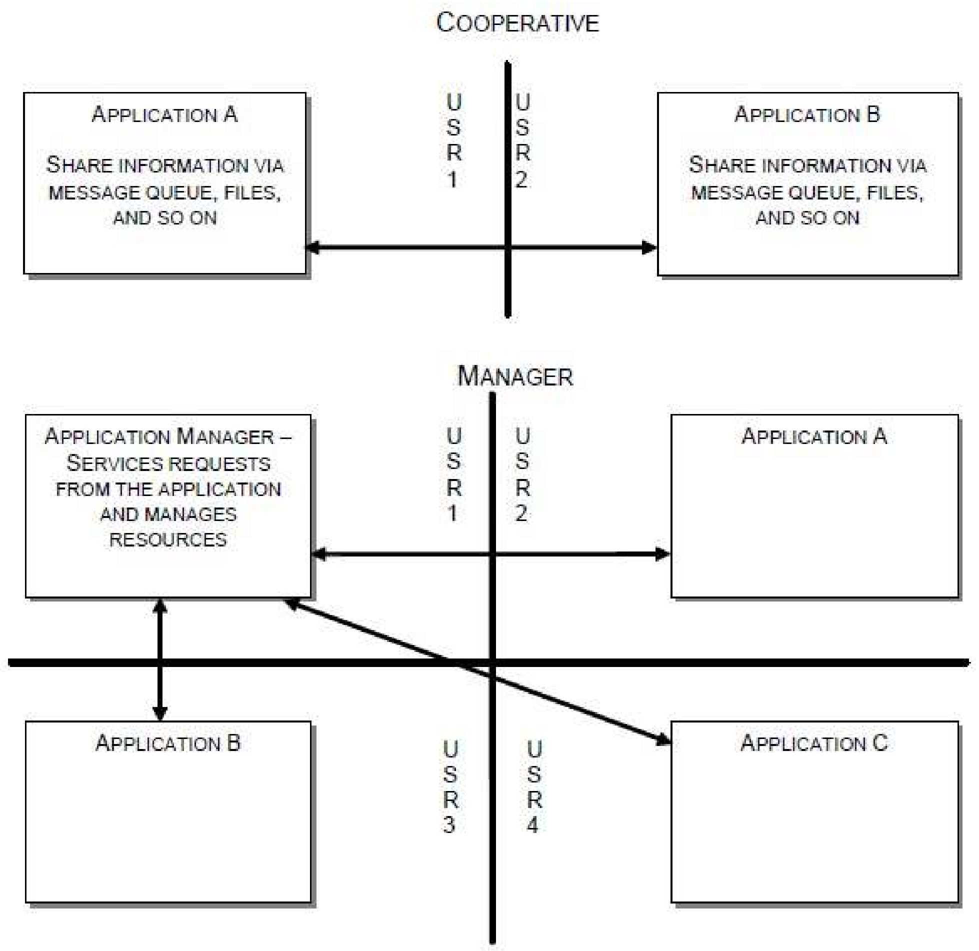

COOPERATIVE
USR1
USR2
APPLICATION A
SHARE INFORMATION VIA MESSAGE QUEUE, FILES, AND SO ON
APPLICATION B
SHARE INFORMATION VIA MESSAGE QUEUE, FILES, AND SO ON
MANAGER
APPLICATION MANAGER – SERVICES REQUESTS FROM THE APPLICATION AND MANAGES RESOURCES
USR1
USR2
APPLICATION A
APPLICATION B
USR3
USR4
APPLICATION C

Figure 2 Application User Accounts in Cooperative and Managed Environments

User Accounts

Linux supports a multi-user environment. V/OS has 16 user accounts, usr1…usr16. These accounts each have a base directory in `/home`.

All user accounts allow both reads and executes by the other accounts. usr1–usr16 are set so that only the account owner can perform write operations to their home directory.

File Permissions

Linux naturally supports a comprehensive file permission mechanism that is very helpful with regard to multi-application support. File access may be controlled via read, write and execute permissions. Each user account can control the permission settings of its files. All user accounts are members of the same group called “users”. If a file allows group permissions, then all user accounts can access that file. Properly using Linux file permissions makes it very easy to share some files while hiding others. It is recommended that before creating an application download .tar file, that file permissions be checked and adjusted as desired.

Launching Applications

On power-up and restarts, the system looks at the *GO parameter in the usr1 configuration file (`/mnt/sram/config.usr1`) to determine which application to start. This application must start all other application(s). Use `svcRunUsr()` to execute a different user account’s application. For example,

```
svcRunUsr("usr2","app.exe");
```

launches the application `app.exe` in `/home/usr2`. Note that `app.exe` must have group execute permissions.

NOTE `svcRunUsr()` starts the specified application without access to a virtual terminal. This means that the application cannot access the keypad and touch panel. This is useful when the multi-application design includes a director or controller process (those run in the usr1 account and are started using `*GO` variable). Director applications can manage the display, keypad, and touch panel.

GUI Applications

Two (or more) GUI applications can run simultaneously as long as only one application controls the display, keypad, and touch panel. This cooperative implementation. Other applications cannot force a display refresh (such as playing a video). The Linux Virtual Terminal mechanism controls which application owns the display, keypad, and touch panel.

When an application starts, it is assigned a virtual terminal descriptor. This descriptor enables the virtual terminal of the application. An application can determine its virtual terminal descriptor in several ways. One way is using the following code snippet:

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/vt.h>

// Return Virtual Terminal number or -1 = ERROR
int getVT (void) {
   int result = -1;
   struct vt_stat st;
```

```
    int fd = open("/dev/tty0", O_RDONLY);
    if (fd >= 0)
    {
        if (ioctl(fd, VT_GETSTATE, &st) >= 0)
            result=st.v_active;
        close(fd);
    }
    return (result);
}
```

Applications must share their virtual terminal descriptor to control the active application.

To switch control of the display, keypad, and touch panel, either perform a system call using the `chvt x` command (where, *x* is the virtual terminal to switch to) or use the following code snippet:

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/vt.h>

// Activate a Virtual Terminal, num = Virtual terminal #
// Return: 0=OK, -1=ERROR
int activateVT (int num) {
    int result = -1;
    struct vt_stat st;

    int fd = open("/dev/tty0", O_RDONLY);
    if (fd >= 0)
    {
        ioctl(fd, VT_ACTIVATE, num);
        result=ioctl(fd, VT_WAITACTIVE, num);
        close(fd);
    }
    return (result);
}
```

CHAPTER 6

Power Management

Power Management is the effort to minimize power usage in portable and non-portable systems.The primary benefit is increased battery life, but there are other benefits such as reduced heat dissipation. The V/OS reduces power consumption two ways:

- During sustained periods of system/application inactivity.

  Inactivity is when all active tasks are placed in a wait state for a specified period of time, during which there are no activities, such as a keypad or touch screen press, card swipe, send/receive of data, and so on.

- A predefined period sleep time.

  You can define when the system enters sleep mode, and when is wakes up in normal mode.

The actions taken by the V/OS to reduce power consumption include:

- Peripherals are powered off, including the magnetic card reader, LCDs, and so on.
- Putting the microprocessor in sleep mode
- Putting memory in self-refresh mode
- Disabling of the phased-locked loop (PLL) circuitry for clock generation
- Shutting down the core processor
- Reducing power to approximately 30 mA

Standby Mode

In Standby mode, the LCD and LEDs are dimmed, and the processor state is maintained. The DRAM remains awake. Send a wake-up devices event to power on the unit.

Sleep Mode

Once in Sleep mode, the unit only wakes on service events. All devices are powered down. The DRAM is placed in self-refresh mode, periodically waking up. Send an RTC alarm signal to wake the unit.

Wakeup Events

The V/OS polls for various wake-up events, including expired timers in the application area, screen touch, keypad press, RTC alarm time expiration, data received over the serial port, or USB device detection.

Use the Power Management service to control which devices wake the system.

Power Management System

The V/OS has a configurable script-based Power Management policy. On the terminal, use the Power Setting screen to configure Power Management options.

Key Power Management points are:

- The terminal always powers up to a running state.
- The unit enters Standby and Sleep modes when all applications are idle for a specified period.
- When the unit enters Standby mode, the backlight and LEDs are turned off.
- When the unit enters Sleep mode, all devices are shut down except those used to wake the system, and only the RTC clock continues to run. The DRAM is placed in Self-refresh mode to preserve the machine state (system and application text/data loaded in memory).
- The unit wakes from Standby or Sleep mode on a wake-up event such as a key press, or based on the expiration of an application timer.
- An attack on the security system (such as opening the case) causes the unit to perform an orderly power-off sequence.

Application Interface

Application participation in Power Management is indirect but absolutely essential for the success of the system. Each application must be designed as an event-driven program. Application can configure power management functionality by PM Service.

Applications can register to the Power Management callback function for tasks to do during system mode changes (standby or sleep modes). This callback function is also used to allow and disallow the system to enter power consumption mode.

Power Management Structure

The following is the Power Management XML structure:

```
struct apm_config
{
    unsigned int Signature;
    unsigned int Size;             /**<  Size of struct */
    unsigned int apm_on;          /**<   1=On
                                         0=Off */
    unsigned int InactivityTimeToSusspend; /**< Minutes */
    unsigned int auto_mode;       /**<   1= On
                                         0=Off */
    unsigned int AutoTimeToShutDown; /**< Time to shut down */
    unsigned int AutoTimeToWakeUp; /**< Time to wake up */
    unsigned int Standby_timeout;
    unsigned int communication_on;
    unsigned int Shutdown_timeout;   /**< Works only if supported on the
                                          device, Value is in Minutes */
    unsigned int permit_state;
};
```

Power Management Functions

Applications must use the following system calls and XML methods for V/OS-based terminals.

powermngt_getVersion()

Returns the POWER_MGMT service version.

Prototype

```
struct version powermngt_getVersion(void);
```

Return Values

The version structure defined in `svcmgrSvcDef.h`.

XML Command

```
<svc_cmd>
   <powermngt>
      <getVersion/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
<powermngt>
      <getVersion>
         <return type="container">
            <major>int</major>
            <minor>int</minor>
            <maint>int</maint>
            <build type="string">string [max len 16]</build>
         </return>
      </getVersion>
   </powermngt>
</svc_rtn>
```

powermngt_get_config()

Returns the Power Management configuration. View the struct apm_config for details.

Prototype

```
int powermngt_get_config(struct apm_config  *apm);
```

Parameters

```
apm_config

apm
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <get_config>
         <apm type="list">
            <item type="container">
               <Signature>unsigned int [default:0]</Signature>
               <Size>unsigned int [default:0]</Size>
               <apm_on>unsigned int [default:0]</apm_on>
               <InactivityTimeToSusspend>
               unsigned int [default:0]
               </InactivityTimeToSusspend>
               <auto_mode>unsigned int [default:0]</auto_mode>
               <AutoTimeToShutDown>
               unsigned int [default:0]
               </AutoTimeToShutDown>
               <AutoTimeToWakeUp>
               unsigned int [default:0]
               </AutoTimeToWakeUp>
               <Standby_timeout>
               unsigned int [default:0]
               </Standby_timeout>
               <communication_on>
               unsigned int [default:0]
               </communication_on>
               <Shutdown_timeout>
               unsigned int [default:0]
               </Shutdown_timeout>
               <permit_state>unsigned int [default:0]</permit_state>
            </item>
         </apm>
      </get_config>
```

```
    </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
    <powermngt>
        <get_config>
            <return>int</return>
        </get_config>
    </powermngt>
</svc_rtn>
```

powermngt_set_config()

Sets the Power Management configuration. View the struct apm_config for details.

Prototype

```
int powermngt_set_config(struct apm_config *apm);
```

Parameters

`apm_config`

`apm`

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
    <powermngt>
        <set_config>
            <apm type="list">
                <item type="container">
                    <Signature>unsigned int [default:0]</Signature>
                    <Size>unsigned int [default:0]</Size>
                    <apm_on>unsigned int [default:0]</apm_on>
                    <InactivityTimeToSusspend>
                    unsigned int [default:0]
                    </InactivityTimeToSusspend>
                    <auto_mode>unsigned int [default:0]</auto_mode>
                    <AutoTimeToShutDown>
                    unsigned int [default:0]
                    </AutoTimeToShutDown>
                    <AutoTimeToWakeUp>
                    unsigned int [default:0]
                    </AutoTimeToWakeUp>
                    <Standby_timeout>
                    unsigned int [default:0]
                    </Standby_timeout>
                    <communication_on>
                    unsigned int [default:0]
                    </communication_on>
                    <Shutdown_timeout>
                    unsigned int [default:0]
                    </Shutdown_timeout>
                    <permit_state>unsigned int [default:0]</permit_state>
                </item>
            </apm>
        </set_config>
    </powermngt>
```

```
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <set_config>
         <return>int</return>
      </set_config>
   </powermngt>
</svc_rtn>
```

powermngt_Shutdown()

Immediately forces shut down. The unit wakes up on an RTC alarm (if previously set using powermngt_SetWakeupTime()).

Prototype

```
int powermngt_Shutdown(void);
```

Return Values

On success, the function does not return.	
-1	Fail

XML Command

```
<svc_cmd>
    <powermngt>
        <Shutdown/>
    </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
    <powermngt>
        <Shutdown>
            <return>int</return>
        </Shutdown>
    </powermngt>
</svc_rtn>
```

powermngt_SetWakeupTime()

Sets RTC alarm to wake up the unit as set in the `utildt` parameter.

Prototype

```
int powermngt_SetWakeupTime(struct svc_power_DateTime *utildt);
```

Parameters

`svc_power_DateTime`	
`utildt`	

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <SetWakeupTime>
         <utildt type="list">
            <item type="container">
               <tm_sec>int [default:0]</tm_sec>
               <tm_min>int [default:0]</tm_min>
               <tm_hour>int [default:0]</tm_hour>
               <tm_mday>int [default:0]</tm_mday>
               <tm_mon>int [default:0]</tm_mon>
               <tm_year>int [default:0]</tm_year>
               <tm_wday>int [default:0]</tm_wday>
               <tm_yday>int [default:0]</tm_yday>
               <tm_isdst>int [default:0]</tm_isdst>
            </item>
         </utildt>
      </SetWakeupTime>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <SetWakeupTime>
         <return>int</return>
      </SetWakeupTime>
   </powermngt>
</svc_rtn>
```

powermngt_CritcalSectionEnter()

Enters the PM critical section. PM events are suspended until a call to powermngt_CritcalSectionExit(). During this section, the system cannot enter Standby or Sleep mode.

Prototype

```
int powermngt_CritcalSectionEnter(void);
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <CritcalSectionEnter/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <CritcalSectionEnter>
         <return>int</return>
      </CritcalSectionEnter>
   </powermngt>
</svc_rtn>
```

powermngt_CritcalSectionExit()

Exits the Power Management critical section. Power Management events resume at their last state. The system is allowed by the application to enter Standby or Suspend mode.

Prototype

```
int powermngt_CritcalSectionExit(void);
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <CritcalSectionExit/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <CritcalSectionExit>
         <return>int</return>
      </CritcalSectionExit>
   </powermngt>
</svc_rtn>
```

powermngt_declare_state()

Prototype

```
int powermngt_declare_state(int proc_id, int state);
```

Parameters

Return Values

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <DeclareState/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
```

powermngt_GetInfo()

Retrieves Power Management information.

Prototype

```
int powermngt_GetInfo(struct svc_apm_info *svc_apm_info_ptr);
```

Parameters

`svc_apm_info`	
`svc_apm_info_ptr`	Pointer to the buffer containing the Power Management data.

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <GetInfo>
         <svc_apm_info_ptr type="list">
            <item type="container">
               <driver_version type="string">
               string [max len 10] [default:NULL]
               </driver_version>
               <apm_version_major>int [default:0]</apm_version_major>
               <apm_version_minor>int [default:0]</apm_version_minor>
               <apm_flags>int [default:0]</apm_flags>
               <ac_line_status>int [default:0]</ac_line_status>
               <battery_status>int [default:0]</battery_status>
               <battery_flags>int [default:0]</battery_flags>
               <battery_percentage>int [default:0]</battery_percentage>
               <battery_time>int [default:0]</battery_time>
               <using_minutes>int [default:0]</using_minutes>
            </item>
         </svc_apm_info_ptr>
      </GetInfo>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <GetInfo>
         <return>int</return>
      </GetInfo>
   </powermngt>
</svc_rtn>
```

powermngt_GetEvent()

Registers Power Management events.

Prototype

```
int powermngt_GetEvent(unsigned int callback_event);
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <GetEvent>
         <callback_event>unsigned int [default:0]</callback_event>
      </GetEvent>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <GetEvent>
         <return>int</return>
      </GetEvent>
   </powermngt>
</svc_rtn>
```

powermngt_Susspend()

Places the system in Suspend (Sleep) mode.

Prototype

```
int powermngt_Susspend(void);
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <Susspend/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <Susspend>
         <return>int</return>
      </Susspend>
   </powermngt>
</svc_rtn>
```

powermngt_Standby()

Places the system in Standby mode.

Prototype

```
int powermngt_Standby(void);
```

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <Standby/>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <Standby>
         <return>int</return>
      </Standby>
   </powermngt>
</svc_rtn>
```

powermngt_get_wakeupdevices()

Determines (bitwise) which device wakes the system.

Prototype

```
int powermngt_get_wakeupdevices(unsigned int * WakeUpDevices);
```

Parameters

`WakeUpDevices`	The device wake-up bitmap is: • PM_WAKE_DEVICE_KEYPAD • PM_WAKE_DEVICE_RTC_ALARM • PM_WAKE_DEVICE_TOUCH_SCREEN • PM_WAKE_DEVICE_UART0 • PM_WAKE_DEVICE_UART1 • PM_WAKE_DEVICE_UART2 • PM_WAKE_DEVICE_UART3 • PM_WAKE_DEVICE_USB_OTG • PM_WAKE_DEVICE_USB_HOST

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <get_wakeupdevices>
         <WakeUpDevices type="list">
            <item>unsigned int</item>
         </WakeUpDevices>
      </get_wakeupdevices>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <get_wakeupdevices>
         <return>int</return>
      </get_wakeupdevices>
   </powermngt>
</svc_rtn>
```

powermngt_set_wakeupdevices()

Sets (bitwise) which device wakes the system.

Prototype

```
int powermngt_set_wakeupdevices(unsigned int WakeUpDevices);
```

Parameters

`WakeUpDevices`	Handle of the activated device that wakes the unit out of PM mode. The device wake-up bitmap is: • PM_WAKE_DEVICE_KEYPAD • PM_WAKE_DEVICE_RTC_ALARM • PM_WAKE_DEVICE_TOUCH_SCREEN • PM_WAKE_DEVICE_UART0 • PM_WAKE_DEVICE_UART1 • PM_WAKE_DEVICE_UART2 • PM_WAKE_DEVICE_UART3 • PM_WAKE_DEVICE_USB_OTG • PM_WAKE_DEVICE_USB_HOST

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <set_wakeupdevices>
         <WakeUpDevices>unsigned int [default:0]</WakeUpDevices>
      </set_wakeupdevices>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <set_wakeupdevices>
         <return>int</return>
      </set_wakeupdevices>
   </powermngt>
</svc_rtn>
```

powermngt_get_sufficient_battery()

Checks that the battery has sufficient power to perform at least one transaction.

Prototype

```
int powermngt_get_sufficient_battery(unsigned int *sufficient);
```

Parameters

`sufficient`	Outputs true/false according to results of the check.

Return Values

=0	Success
-1	Fail

XML Command

```
<svc_cmd>
   <powermngt>
      <get_sufficient_battery>
         <sufficient type="list">
            <item>unsigned int</item>
         </sufficient>
      </get_sufficient_battery>
   </powermngt>
</svc_cmd>
```

XML Return

```
<svc_rtn>
   <powermngt>
      <get_sufficient_battery>
         <return>int</return>
      </get_sufficient_battery>
   </powermngt>
</svc_rtn>
```

#defines

```
#define APM_CONFIG_TIME_TOO_SHORT_ERROR -2
#define APM_CONFIG_FAUILR_ERROR -3
#define PM_WAKE_DEVICE_KEYPAD 0x1
#define PM_WAKE_DEVICE_RTC_ALARM 0x2
#define PM_WAKE_DEVICE_TOUCH_SCREEN 0x4
#define PM_WAKE_DEVICE_UART0 0x8
#define PM_WAKE_DEVICE_UART1 0x10
#define PM_WAKE_DEVICE_UART2 0x20
#define PM_WAKE_DEVICE_UART3 0x40
#define PM_WAKE_DEVICE_USB_OTG 0x80
#define PM_WAKE_DEVICE_USB_HOST 0x100
#define APM_SYS_STANDBY 0x0001
#define APM_SYS_SUSPEND 0x0002
#define APM_NORMAL_RESUME 0x0003
#define APM_CRITICAL_RESUME 0x0004
#define APM_LOW_BATTERY 0x0005
#define APM_POWER_STATUS_CHANGE 0x0006
#define APM_UPDATE_TIME 0x0007
#define APM_CRITICAL_SUSPEND 0x0008
#define APM_USER_STANDBY 0x0009
#define APM_USER_SUSPEND 0x000a
#define APM_STANDBY_RESUME 0x000b
#define APM_CAPABILITY_CHANGE 0x000c
#define SVC_AC_LINE_STATUS_OFF (0)
#define SVC_AC_LINE_STATUS_ON (1)
#define SVC_AC_LINE_STATUS_BACKUP (2)
#define SVC_AC_LINE_STATUS_UNKNOWN (0xff)
#define SVC_BATTERY_STATUS_HIGH (0)
#define SVC_BATTERY_STATUS_LOW (1)
#define SVC_BATTERY_STATUS_CRITICAL (2)
#define SVC_BATTERY_STATUS_CHARGING (3)
#define SVC_BATTERY_STATUS_ABSENT (4)
#define SVC_BATTERY_STATUS_UNKNOWN (0xff)
#define SVC_BATTERY_FLAGS_HIGH (0x1)
#define SVC_BATTERY_FLAGS_LOW (0x2)
#define SVC_BATTERY_FLAGS_CRITICAL (0x4)
#define SVC_BATTERY_FLAGS_CHARGING (0x8)
#define SVC_BATTERY_FLAGS_ABSENT (0x80)
#define SVC_BATTERY_PERCENTAGE_UNKNOWN (-1)
#define SVC_BATTERY_TIME_UNKNOWN (-1)
```

Typedefs

```
typedef unsigned short apm_event_t
typedef int( powermngt_event_callback )(apm_event_t event)
```

CHAPTER 7

Services

Services allow the traditional C application world and the WDE (Web Development Environment) world share access to specific device resources and functionality. Service calls expose core V/OS functionality, or add value to the OS, for application use. A Service must have at minimum a service interface comprised of a C header file that defines the Service API and a shared library that implements it.

A service can provide service server-side functionality. This implementation is divided between the functions directly called using the shared service library and the server-side functions (also in the shared library) called by the service server executable (provided by the service manager stack).

Service developers must decide if the shared library is adequate or if a service server is also required and if so, how to determine work division using the following factors:

- how state data is maintained and the scope (visibility) of this data
- whether system resources are acquired
- how competing resource requests are handled

A shared library loads into application data space. Each application must have its own copy of Service data. This may be fine for services providing status or local functionality, but if the service manages a resource (for example, a COM port), then a central entity (that is, server-side service executables) must manage requests from multiple sources (applications).

The service interface and shared library is required even if all implementation is performed by a service server. The shared library always implements the API functions called even if they are nothing more than proxies passing the call on to the executable.

Applications cannot call executable functions directly; only the interface APIs can through the shared library. This means that the executable does not implement the APIs as defined in the interface. The shared library is the only caller in to the executable so it can preprocess or transform the data, handle anything that can be kept in the application scope, and streamline data transfer to/from the executable. The shared library/executable interface must minimal and lightweight.

The shared library and executable communicate using a Service Manager message infrastructure. Both must link with the message library to access this message infrastructure.

Services are designed to be used by both C applications, which call service functions directly, and the Web Development Environment (WDE), which makes calls using XML.

To accommodate WDE, XML commands call a front-end XML parser that is context unique to each Service, and compiles with the Service implementation C code, as shown by the blue box in Figure 3.

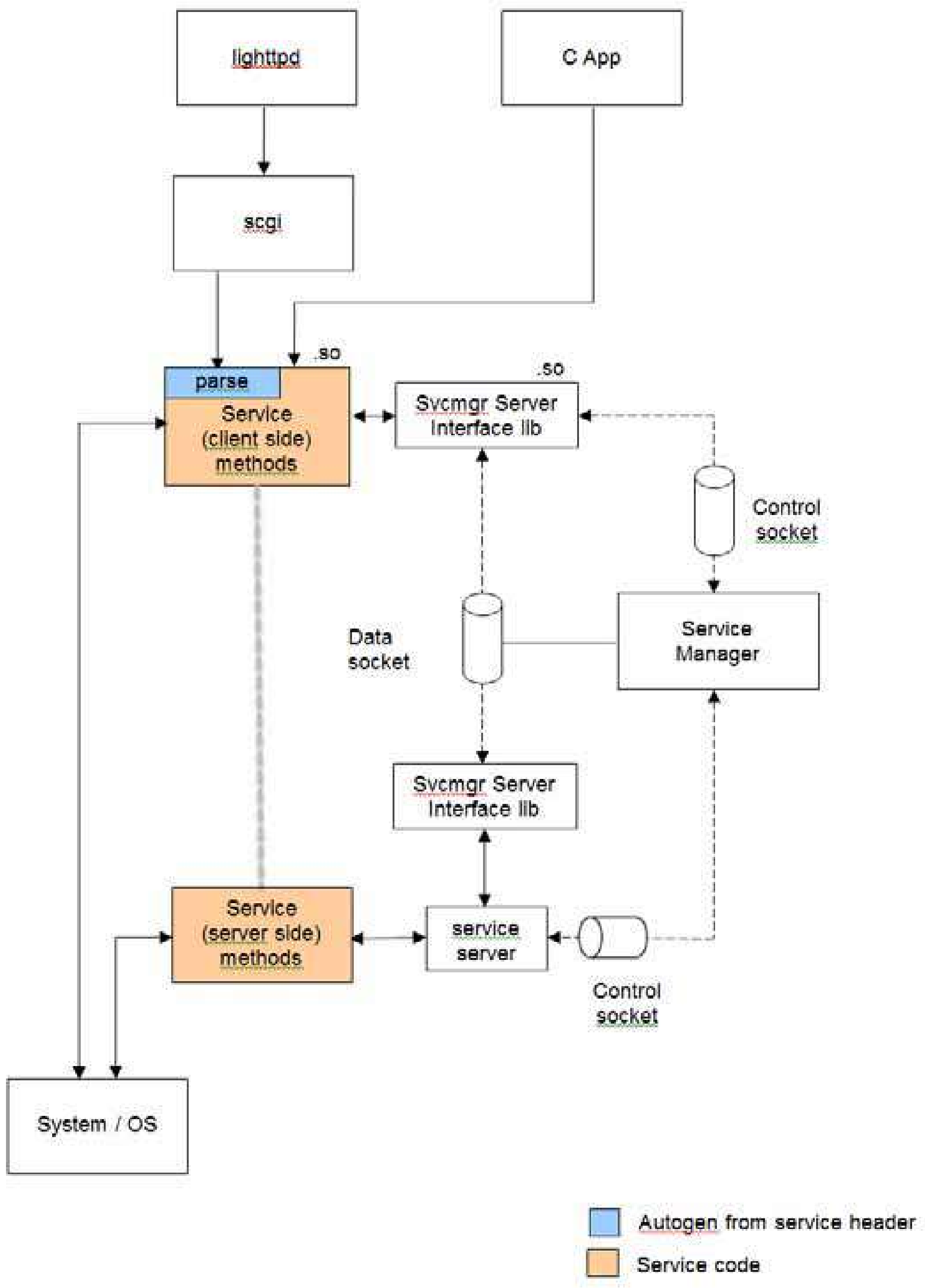

lighttpd
C App
scgi
.so
parse
Service (client side) methods
.so
Svcmgr Server Interface lib
Control socket
Data socket
Service Manager
Svcmgr Server Interface lib
Service (server side) methods
service server
Control socket
System / OS
Autogen from service header
Service code

Figure 3 Service Interface

The XML parsing code is auto-generated using a Java utility that uses the service header file as input. There are limitations on the service interface (header file). These limitations and correct service interface header file construction are presented in this chapter.

V/OS Services

Table 7 lists V/OS services.

Table 7 V/OS Services

svc.h Entry	Service Description
svcMgrService_bluetooth	Bluetooth service
svcMgrService_ecrcom	ECR communications service
svcMgrService_ethernet	Ethernet module service
svcMgrService_event	Events service
svcMgrService_kbd	Keyboard service
svcMgrService_led	LED service
svcMgrService_mdb	
svcMgrService_mplayer	
svcMgrService_msr	Magnetic stripe reader service
svcMgrService_net	
svcMgrService_netloader	Netloader service
svcMgrService_networkapps	
svcMgrService_powermngt	Power management service
svcMgrService_proximity	
svcMgrService_remoteprinter	Remote printer service
svcMgrService_rfcr	Contactless RF card reader service
svcMgrService_security	Security service
svcMgrService_serialcom	Serial port service
svcMgrService_sigcap	Signature capture service
svcMgrService_socbridge	
svcMgrService_sound	Sound service
svcMgrService_sysinfo	System information service
svcMgrService_tempsensor	Temperature sensor service
svcMgrService_utility	
svcMgrService_ux100	Ux100 terminal-specific service
svcMgrService_VHQ	
svcMgrService_videoswitch	Video switch service
svcMgrService_wde	west console application service
svcmgrstack	Stack service

Flow for Service Server Functions

As illustrated in Figure 4, when a client makes a server-side call:

1 The Service Manager listens for connections.

2 The client creates a socket and connects to the Service Manager.

3 The Service Manager accepts and waits to process client requests.

4 The Client requests a socket pair for client/server communication with the named service.

5 The Service Manager checks if access is allowed, then starts a server-side child process, `smi_svcserver_main()`, if not already running.

At start up (that is, on the first request to service server-side), the Service Manager looks for and executes the service server initialization method, `{service}_serverInit()`, where, `{service}` is the service name.

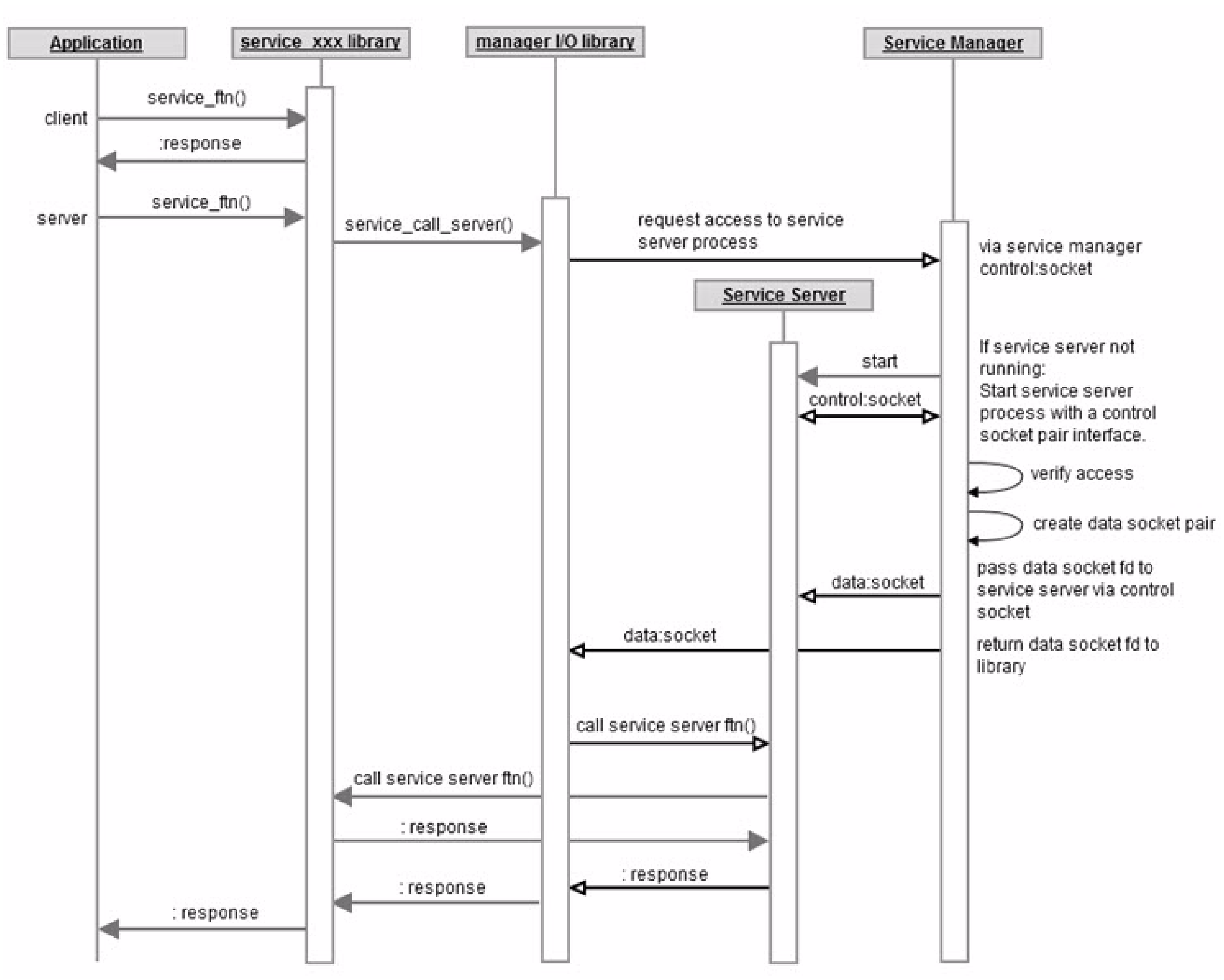

Application
service_xxx library
manager I/O library
Service Manager
client
service_ftn()
:response
server
service_ftn()
service_call_server()
request access to service server process
via service manager control:socket
Service Server
start
control:socket
If service server not running: Start service server process with a control socket pair interface.
verify access
create data socket pair
data:socket
pass data socket fd to service server via control socket
data:socket
return data socket fd to library
call service server ftn()
call service server ftn()
: response
: response
: response
: response

Figure 4 Service Interface Sequence

6 The Service Manager creates a socket pair for client/server.

7 The client receives a descriptor.

8 The server-side opens the shared service library, and requests socket descriptor for communication.

9 The server-side process receives the descriptor and looks to receive a command on the socket.

10 The client-side sends the request.

11 The server-side processes the request and calls the shared service library.

12 The server-side sends a response to the client.

13 The server-side child process remains active waiting for next request to this service server-side process.

14 The socket descriptor pair is closed.

XML Service

This section presents service XML elements, commands, and error codes.

XML Elements

The XML that issues WDE commands or return data is directly based on the C prototype of the command function. Each supported C data type construct can be matched directly to one of the root elements shown in the following code.

```
// Simple (native C type)
{Element<tagname>} := <tagname>value</tagname>

// String (char*)
{Element<tagname>} := <tagname type = "string">value</tagname>

// Binary (void*)
{Element<tagname>} := <tagname type = "binary">base64binary value</
tagname>

// Container (struct)
{Element<tagname>} := <tagname type = "container">
                         {Element1}
                         {Element2}
                         ...
                         {ElementN}
                      </tagname>

// List (array)
{Element<tagname>} := <tagname type = "list">
                         // Note: All "list" {Element}'s are same type.
                         {Element<item>}
                         {Element<item>}
                 ...
                         {Element<item>}
                      </tagname>
```

```
// Pairlist (struct pair*)
{Element<tagname>} := <tagname type = "pairlist">
                         <name1>value1</name1>
                         <name2>value2</name2>
                         ...
                         <nameN>valueN</nameN>
                      </tagname>
```

Notice that the list and container elements contain other elements, so it is possible to create XML of any depth. Multi-element assemblies are defined as composite elements. All elements (root or composite) have a single root tag. This is important as elements are associated one-for-one with C function parameters and return data.

Although the XML can be used without any thought as to the prototypes of service functions, it is easier to understand how the XML is constructed and used when associated with C. This also shows how it is possible to auto-generate XML format documentation from Service header files.

Each element is associated with a C data type and its data, either native or complex. In the following descriptions and examples of the root elements note that the examples depict the element data as function parameters for simplicity; any root or composite element can also be used as a function return type.

- Simple (native):

  ```
  {Element<tagname>} := <tagname>value</tagname>
  ```

  Represents a single datum of any type. `<tagname>` corresponds to the C parameter name and the value is the data. This is the only element where `<tagname>` does not have a type attribute.

  Example:

```
void fn(int age, float height, char initial);

   <age>30</age>
   <height>30</height>
   <initial>X</initial>
```

- String (`char*`):

  ```
  {Element<tagname>} := <tagname type = "string">value</tagname>
  ```

  Represents a string (null-terminated array of `char`), specifically, a `char*`. No other type is a string. For example, an `unsigned char*` is a list of `unsigned char`, not a string. `<tagname>` corresponds to the C parameter name and value is the string data.

  Example:

```
void fn(char* title, int page_cnt);

   <title type = "string">An Inconvenient Truth</title>
```

```
<page_cnt>432</page_cnt>
```

- Binary (`void*`):

  ```
  {Element<tagname>} := <tagname type = "binary">base64binary
  value</tagname>
  ```

  Represents a binary data ‘blob.’ This converts to a variable of type `void*` and an associated count. See Array Handling, specifically array handling construct #3, for information on runtime array management.

  `<tagname>` corresponds to the C parameter name and value is the binary data in base64 format.

  Example:

```
void fn(struct bindata my_bdata);

   struct bindata {
      void* data;
      int data_count;
   }
   ...
      <my_bdata type = "container">
         <data type="binary">QQ==</data>
      </my_rect>
```

- Container (structure):

```
{Element<tagname>} := <tagname type = "container">
                          {Element1}
                          {Element2}
                       ...
                          {ElementN}
                          </tagname>
```

  Represents a C structure. Structure members can be of any element type. The example below only uses simple member elements.

  Example:

```
void fn(struct rectangle my_rect);

   struct rectangle {
      int left;
      int top;
      int right;
      int bottom;
   }

       <my_rect type = "container">
         <left>0</left>
         <top>0</top>
         <right>100</right>
```

```
        <bottom>200</bottom>
    </my_rect>
```

- List (array):

```
{Element<tagname>} := <tagname type = "list">
                            // Note: All "list" {Element}'s are same type.
                                {Element<item>}
                                {Element<item>}
                                ...
                                {Element<item>}
                                </tagname>
```

Represents an array of a simple or container elements. Note that unlike container, each element in a list must be the same element type and all have `<tagname>`.

List Elements can represent pointers, and both variable length and constant (compile time defined) arrays. It is important to understand these subtleties when using arrays. Refer to Array Handling.

Example:

```
void fn(struct data_array myData);

    struct data_array {
        int* data;
        int data_count;
    }

        <myData type = "container">
            <data_count>4</data_count>
            <data type = "list">
            <item>0</item>
            <item>1</item>
            <item>10</item>
            <item>100</item>
        </data >
    </myData>
```

- Pairlist (`struct pair*`):

```
{Element<tagname>} := <tagname type = "pairlist">
                                <name1>value1</name1>
                                <name2>value2</name2>
                                ...
                                <nameN>valueN</nameN>
                            </tagname>
```

This is a special type for passing any number of `name/value` pairs as a single element. You can create this type from the other root elements (it is a list of containers of two simple elements), but because this is likely to be a common composite type, this element provides a shorthand to significantly reduce the XML. Structure Definition, it has two `char*` members: `name` and `value`.

Refer to Service Development for detailed information on `pairlist` usage.

Example:

```
void fn(struct pairlist receipt);

        <receipt type = "container">
            <list_count>4</list_count>
            <list type = "pairlist">
                <pline>first line</pline>
                <plinen>another line</plinen>
                <pcmd>set bold</pcmd>
                <plinen>last line</plinen>
            </list>
    </receipt>
```

XML Command and Return Format

- SVC_CMD XML format:

```
    <serviceName>
        <methodName>
{Element<>}*        // zero or more {Element}'s
        </methodName>
    </serviceName>
</svc_cmd>
```

- SVC_RTN XML format:

```
<svc_rtn>
    <serviceName>
        <methodName>
            <errcode>IntErrcodeVal</errcode>
                -- or --
            <callcode>IntCallcodeVal</callcode>
                -- or --
            {Element<return>}?// zero or one {Element}
        </methodName>
    </serviceName>
</svc_rtn>
```

Again, both the command XML and return XML is handled by the auto-generated code. Service developers only write C functions and do not interact with XML.

`svc_rtn` must return either:

- a return value as `{Element}` XML (or no return value if the function returns void),
- an error code, or

- a call code.

The called function returns a value and errno set to 0.

Error Codes

- Always set errno to 0 on the line above a `return()` statement when the return value is known to be valid.

  An error code returns when a function returns with errno ≠ 0. The errcode value (`IntErrcodeVal` in XML) is the value of errno. It is important to set errno = 0 when a function return value is valid as errno is always checked and an `<errcode>` returned instead of the function return value when errno ≠ 0.

- Always free any malloc'd return data before exiting a client Service function with errno ≠ 0.

  It is important that any `malloc` on behalf of return variables are `free`'d before returning when errno ≠ 0. Normally, auto-generated code frees all return data pointer variables. This is not the case when errno ≠ 0; <errcode> returns and no operations are performed on return data. Memory leaks can occur if the heap is not clean when errno ≠ 0.

Call Codes

A call code returns on internal error and when successful command completion cannot not be accomplished. The call code values (`IntCallcodeVal` in XML) are:

Call code	Description
0	An error occurred before the command `xml (<svc_cmd>)` parsed. Returns: • `serviceName` = '`svcmgr`' • `methodName` = '`error`'
1	Internal error prevented `<svc_rtn> xml` to be built. This is severe and suggests memory or domain errors.
2	`svc_cmd serviceName` does not exist or did not load. A log entry in `/var/log/messages` indicates cause of failure.
3	`svc_cmd methodName` does not exist.
4	`svc_cmd` parse error. Returns: • `serviceName` = '`svcmgr`'
5	`svc_cmd` missing a required parameter.
6	WDE interface unavailable. Service not compiled with auto-generated code.
7	`svc_rtn` did not build due to internal error.
8	`svc_cmd` not processed due to Service Manager security policy settings.
9	Internal error. Configuration file location not defined. Returns: • `serviceName` = '`svcmgr`' • `methodName` = '`error`'
10	Conversion of a binary type (`void*`) variable failed.

NOTE Call code 2 typically occurs because the service was built without specifying all dependent libraries. A simple test is to create a small executable that includes the service header (the main() can actually be empty), and links only against the service in question (-lsvc_myservice). This reveals missing dependencies.

XML Command and Return Examples

The following command and return examples are fictitious. There is no attempt to make them cohesive nor are they designed to demonstrate best practices, only to illustrate XML. The examples are followed by minimal contents of a valid Service Interface header file that encompasses all XML command and return examples.

- Command XML

```
<svc_cmd>
   <myService>
      <set_image_rect>
         <imgName type="string">mona_lisa.png</imgName>
         <loc_x>10</loc_x>
         <loc_y>20</loc_y>
         <size_x>200</size_x>
         <size_y>300</size_y>
      </set_image_rect>
   </myService>
</svc_cmd>



<svc_cmd>
   <myService>
      <print>
         <title type = "string">things to do<title>
         <lines type = "container">
            <list_count>4<list_count>
            <list type="pairlist">
               <pline>first line</pline>
               <plinen>another line</plinen>
               <pcmd>set bold</pcmd>
               <plinen>last line</plinen>
            </list>
         </lines>
      </print>
   </myService>
</svc_cmd>
```

- Return XML:

```
<svc_rtn>
   <myService>
      <packages>
         <return type="container">
```

```
                <packages_count>2</packages_count>
            <packages type="list">
            <package type="container">
                <user type="string">usr1</user>
                <pkg_name type="string">curl</pkg_name>
                <version>1.6</version>
                <date type="string">24Aug2010</date>
                </package>
                <package type="container">
                    <user type="string">usr1</user>
                    <pkg_name type="string">openssl</pkg_name>
                    <version>2.3</version>
                    <date type="string">12Oct2010</date>
                </package>
            </packages>
        </return>
        </packages>
    </myService>
</svc_rtn>


<svc_rtn>
    <myService>
        <cables>
            <return type="container">
                <type type="string">mx_series</type>
                <usb type="string">host</usb>
                <Ethernet type="string">yes</ethernet>
                <comports_count>3</comports_count>
                <comports type="list">
                <comport type="container">
                    <port type="string">com1</port>
                    <type>232</type>
                    <connected type="string">no</connected>
                    <firmware type="string">N/A</firmware>
                </comport>
                <comport type="container">
                    <port type="string">com2</port>
                    <type>232</type>
                    <connected type="string">no</connected>
                    <firmware type="string">N/A</firmware>
                </comport>
                <comport type="container">
                    <port type="string">com3</port>
                    <type>232</type>
                    <connected type="string">yes</connected>
                    <firmware type="string">1.1</firmware>
                </comport>
            </comports>
            </return>
        </cables>
    </myService>
</svc_rtn>
```

- Service Interface header file for these XML examples:

```
#ifndef __MY_SERVICE_H__
#define __MY_SERVICE_H__

// This is needed since 'pairlist' is used by this service
#include "svcmgrSvcDef.h"

/*SVC_SERVICE:myService*/

/*SVC_STRUCT*/
struct package {
   char * user;
   char * pkg_name;
   float  version;
   char * date;
}

/*SVC_STRUCT*/
struct package_array {
   struct package* packages;
   int packages_count;
}

/*SVC_STRUCT*/
struct comport {
   char * port;
   int    type;
   char * connected;
   char * firmware;
}

/*SVC_STRUCT*/
struct cabledata {
   char * type;
   char * usb;
   char * ehternet;
   struct comport* comports;
   int comports_count;
}

/*SVC_PROTOTYPE*/
int myService_set_image_rect(char* imgName,
                            int loc_x, int loc_y, int size_x, int size_y);

/*SVC_PROTOTYPE*/
void myService_print(char* title, struct pairlist lines);

/*SVC_PROTOTYPE*/
struct package_array myService_packages(void);

/*SVC_PROTOTYPE*/
struct cabledata myService_cables(void);

#endif //__MY_SERVICE_H__
```

Service Development

A Service must, at minimum, provide a C interface API defined by a header file supported by a shared library, `.so`. All references to Service developer source files must the syntax `{service}` where, `{service}` is the Service name.

The Service Interface header file must be named `svc_{service}.h`. For example, a printer Service may contain the files `svc_printer.h` and `printer.c`.

C applications that link to a Service or are invoked through WDE can call the Service interface API directly. Except client-side service methods, which are called directly by a linked application or through the WDE stack, a service can contain server-side services. Server-side services are methods (contained in the `.so` library) managed by a separate single-threading process. Any interface to server-side methods are contained and managed by the client-side methods. No application or WDE directly calls or invokes server-side methods.

WDE uses XML calls and return results. The code that transforms command XML to/from native C is auto-generated using the Service Interface header file. Both allowing Service calls through WDE and utilizing auto-generated code places restrictions on the format and content of the C interface.

C Content Restrictions discusses C code limitations. Comment Tag Directives discusses special C comments used to tag interface content. XML Command and Return Format defines the command and return XML. The XML format definition guarantees that all XML can be represented in C. Each XML element conforms to one of the following:

- a native C type (such as, `int`, `float`, `char`, and so on)
- a string (`char*`)
- a binary base64 formatted string (`void*`)

- a user-defined C structure that contains any or all native types, strings, arrays, pairlists, or user-defined structs
- an array of a C type (except char) or user-defined structure
- a pairlist–a special designation for an array of structures of type pair (`struct pair*`)

C Content Restrictions

Because the Service Interface API (that is, C prototypes) can be called by WDE (XML files), it is not possible, for instance, to return data using parameter pointers. Similarly, since the XML parsing code is auto-generated, the length of an array must be well defined. Due to these and other limitations, the C prototypes that define the Service interface must conform to the following restrictions:

- Prototype parameters cannot return data; only [IN] params.
- The prototype return value is for all return data.
- errno returns error conditions. errno must return 0 on success.
- structs cannot contain structs with the same name or pointers to structs of the same name. No recursive structures. No linked lists.
- structs can define only one variable for each type definition.

  For example, two integer variables 'i' and 'j' must be defined as
  `int i; int j;` (on separate lines)
  not
  `int i, j;`
- No multi-dimensional variables. Only linear arrays (single pointers) allowed.

  The exception to this is `char*`, which is treated as a string. Arrays of strings are allowed, which in C is defined as `char**`.
- All pointers to data returned by interface functions must be `malloc`'d by the function and `free`'d by the caller.
- No `enum` or `typedef`.
- Arrays must be either a single element (such as a pointer), or have the length defined at compile time (for example, `int i[20];`), or contained within a structure accompanied by an `{array_name}_count` member of type `int`. This provides for run-time variable length arrays.
- No function name, parameter name, structure name, or structure member name can begin with an underscore.

NOTE

There is a non-obvious limitation due to Service precompiler grammar:
If a parser error occurs stating:
`Encountered "<EOF>" was expecting <COMMENT>`…
the problem is a C style comment (/**/) at the end of the header file. Fix this error by either changing it to C++ style comments (//) or move the comment above the last non-comment line.

Comment Tag Directives

The preprocessor requires knowledge of the Service name (`{service}`), which declarations are part of the exposed API, the required parameters, default values, and so on. To provide this information and maintain proper C syntax, in the Service Interface header file define set of preprocessor directives in C style comments (/**/).

Service Name Definition

```
/*SVC_SERVICE:myservice*/
```

Each Service must specify its service name in the header file. This is only done once, preferably at or near the top of the file. This single C style comment contains the keyword `SVC_SERVICE` (all uppercase) followed by a colon and the name of the service. No spaces allowed.

Example:

```
/*SVC_SERVICE:led*/
```

The header file contains the interface for the LED Service.

Function Prototype Definition

```
/*SVC_PROTOTYPE*/
return_type myservice_fnname(type var_name [/*default_val | REQ*/], ...);
```

Each exported Service function must have a preceding comment that tags it as an API function (for example, `/*SVC_PROTOTYPE*/`). Locate this comment anywhere before the function prototype, but only white space or comments can be between this comment and the function prototype.

Prototype function names are prefixed with the service name and underscore. `methodName` is the function name without the service name prefix. The Service Manager Stack prefixes the service name when making the call. C applications are required to call `serviceName_methodName` since functions are called directly.

The prototype is defined normally, except that each variable can include a default value. The default value is used when a Service call is made with the `value` variable absent. This is only possible when calling from XML (WDE), as C calls are compiled and based on the header file. The default value must be in comments (/**/) with no spaces between the asterisks and the value. If a default value comment is absent, value= 0. Use `/*REQ*/` tag comments to indicate that the variable is required. This means that if the value is missing in XML, an error (call code) returns. `type` and `return_type` can be a standard C type, a string (`char*`), or a defined `/*SVC_STRUCT*/` structure.

Example:

```
/*SVC_PROTOTYPE*/
int myservice_fn(int i, double d /*3.14*/, char c /*REQ*/);
```

- The default value for `i` is 0 because it is undefined.
- The d parameter has a default value of 3.14.

- The c parameter is required and has no default value.

Structure Definition

```
/*SVC_STRUCT*/
struct struct_name {
type member_name [/*default_val*/];
...
};
```

Each exported Service structure must have a preceding comment to tag it as a `struct` (`/*SVC_STRUCT*/`). This comment can be anywhere before the structure definition, but only white space or comments can be between this comment and the struct definition.

Structure member types can be standard C types, strings (`char*`), pointers of `std` types and strings, and structs. Also, members can provide default values, like function prototypes (see Function Prototype Definition) with one notable exception: structs cannot use the `/*REQ*/` comment tag.

NOTE

Only one member per defined type is allowed. Multiple members of the same type must be listed separately.

For example, `i` and `j` integer variables must be defined as:

```
int i;
int j;
```

on separate lines, not:

```
int i, j;
```

Example:

```
/*SVC_STRUCT*/
struct person {
   char*  name;
   int    age;
   char*  parent[2];
   char** children;
   int    children_count;
};
```

None of the members have default values, and default to zero (NULL for pointers). Note that the variable length string array `children` must be accompanied by an array count member of the same name, suffixed with `_count`. See Array Handling for more detail on passing/returning arrays.

Service Interface and the C Preprocessor

The Service Precompiler utility does not run the standard C preprocessor therefore, standard preprocessor directives such as `#define` and `#include` are not expanded.

It is recommended that the Service Interface header file contain standard C preprocessor directives only when necessary. With very few exceptions this means eliminating all directives except `#include`. In addition, define all interface definitions in the Service Interface header file to eliminate the need for `#include`, and eliminating the need for the C preprocessor.

It is not always possible to eliminate all `#include`s. Using the `pairlist` structure requires a `#include` of `svcmgrSvcDef.h`. When C preprocessor execution is needed, the output passes as the input to the Service preprocessor. It is important that C comments are not stripped by the C preprocessor as they are used as Service Precompiler directives. Use the provided CodeSourcery C compiler for C preprocessing with the following command:

```
arm-none-linux-gnueabi-cpp -E -C -o outfile_name infile_name
```

where,

- E specifies run the preprocessor only
- C specifies keep C comments
- The `outfile_name` is used as input to the Service Precompiler.

Use pipes to avoid intermediate `outfile_name`:

```
arm-none-linux-gnueabi-cpp -E -C infile_name | java -jar
ServicePreprocessor.jar
```

Note that the C++ compiler must be provided directory paths to header files not in the `infile_name` directory; use the `-Ifull_pathname` command line parameter as usual.

Array Handling

Service Interface functions cannot pass or return data by reference because function calls can be made using WDE (XML) or C. All pointers are considered arrays, and the number of array elements must be known either implicitly or explicitly. Three array constructs are possible in Service definitions:

- A pointer variable (either function parameter or structure member) with no associated `_count` variable. This is treated as an array with an element count of one (1).
- The array count defined at compile time using standard bracket [] format.
- A pointer variable contained within a structure with an associated `_count` variable.

Construct (1)

Use construct (1) pointers to data to pass a function parameter, function return value, or structure member. Remember that pointers passed as function parameters can only be used as `[IN]` parameters. The pointer cannot be “filled” within the function with the expectation that the caller accesses it after the call completes. All data to return by a function must be in the return value.

Examples:

```
int* fn(void);
char* fn(int* pi, float f);
void fn(int i, int* pi);
```

Construct (2)

Specify arrays using compile-time syntax. Note that arrays can only be one dimensional, and therefore, except for the string type (`char*`), the pointer asterisk cannot be used when the length is compile-time defined.

Example:

```
char* fn(int i[10], float f);
void fn(int i, char* sz[5]);
struct my_struct{
    char  c;
    int   i[100];
    char* name;
}
```

Construct (3)

A standard idiom in C functions is to pass an array as one parameter and the count as another, for example:

```
int fn(float* my_array, int array_count);
```

This is inadequate for Service Interface functions because the auto-generated code has no association between the array and count function parameters.

To form an association between an array and its count, a variable length array must be contained within a structure, and the structure must also contain an int member that provides the array count. The count member name must be the array member name suffixed with `_count`. The structure can contain any number of other members, including other arrays.

The simplest array construct is a two member structure such as:

```
/*SVC_STRUCT*/
struct my_array {
    int* data;
    int data_count;
};
```

The array is wrapped in a structure that includes an int member, named `{array_name}_count` (in this case `data_count`). Note that there is no limitation on the structure itself. It can contain any number of members, including any number of arrays. The only requirement is that variable length arrays are contained within a struct and there be an associated `_count member`. For example:

```
/*SVC_STRUCT*/
struct points {
    char*  title;
float* x;
       float* y;
       float* z;
       int    x_count;
       int    y_count;
       int    z_count;
   };
```

In the above structure; `title` is not an array, it is a string, and therefore requires no `_count` member. The three arrays (and their counts) are on separate lines, as defined in Comment Tag Directives.

An array of strings can be passed typed as `char**`, and requires the associated `_count` member. Note that "arrays of arrays" is not allowed; `char**` is the only valid "pointer of pointer" construct since this is interpreted as an array of strings.

Binary Data

Pass binary data using XML variables of type `void*`. Auto-generation converts `void*` variables to base64binary (ASCII), and visa-versa.

C programs never see this conversion since it does not call through the auto-generator code, it therefore continues to pass and retrieve binary data as normal.

The binary `void*` variable follows Service Manager pointers of data types (arrays) rules: The variable must be contained within a structure with an accompanying integer member that defines the data length (see Construct (3)). The following is a simple binary data construct:

```
struct myStruct {
   void* data;
   int data_count;
}
```

A function can pass or return this struct:

```
struct myStruct fn(struct myStruct bindata);  // fn passes binary data
                                              // both in & out
```

Consider the following defined header example for Service `mysvc`:

```
/*SVC_SERVICE:mysvc*/

/*SVC_STRUCT*/
struct mysvcBinData{
   void* b;
   int b_count;
}

/*SVC_PROTOTYPE*/
void mysvc_fn(struct mysvcBinData bdata);
```

Use the following XML to execute `mysvc_fn()` using WDE:

```
<svc_cmd>
```

```
   <mysvc>
      <fn>
         <bdata type="container">
            <b type="binary">a base64binary string</b>
         </bdata>
      </fn>
   </mysvc>
</svc_cmd>
```

The `binary` element is the only element that removes a C variable in its XML format, in this case, the `_count` member associated with the `void*`. The `base64binary` data is a transformation of binary to ASCII, and is not the same length as the original binary data. The auto-generation tool performs this transformation and populates the count along with the binary (`void*`) data.

Pairlist

The element type `pairlist` allows name and value pairs pass in a compact XML format through WDE. The following two structures are Verifone-defined and are in the `svcmgrSvcDef.h`, as follows:

```
/*SVC_STRUCT*/
struct pair {
       char* name;
       char* value;
};

/*SVC_STRUCT*/
struct pairlist {
       struct pair * list;
       int list_count;
};
```

To use a `pairlist` in the Service Interface, `#include svcmgrSvcDef.h` in the header file, then the `struct pairlist` type can be passed as a function parameter and/or return value. The following example depicts a Service Interface that includes a function prototype using a `pairlist` and the command XML to pass in three name/value pairs:

```
   #include "svcmgrSvcDef.h"
   /*SVC_SERVICE:myService*/
   /*SVC_PROTOTYPE*/ void myService_myFn(struct pairlist myList);

<svc_cmd>
   <myService>
      <myFn>
         <myList type = "container">
            <list_count>3</list_count>
            <list type = "pairlist">
               <style>craftsman</style>
```

```
                <sqft>1100</sqft>
                <heating>baseboard electric</heating>
            </list>
        </myList>
    </myFn>
  </myService>
</svc_cmd>
```

Memory Management

The auto-generated code that interfaces between WDE XML commands and Service functions must perform the following:

1. Parse an incoming XML command string, building up variables as needed to pass as Service function arguments.
2. Call the Service function.
3. Convert the Service function return value into an XML response string.

To do this in a systematic way without preference as to the complexity of the Service function prototype requires a well-defined protocol for managing ownership of allocated memory. For Service Interface functions the rules are:

Input	All input parameter data is owned by the caller and treated as if it were dynamically allocated, and be `free`'d immediately on function return. This means a Service function should never hold parameter data beyond the scope of the function. The Service should instead copy any parameter data it needs to retain. Also, a Service function should never free parameter data. This is the responsibly of the caller.
Output	For function return data, all pointers including strings must be `malloc`'d, and all non-pointers must not. It is the responsibly of the caller to free all pointer data returned from a Service function. Note that this prohibits the return of string constants or pointers returned from most functions. This is important. The auto-generated code always (tries to) `free`'s pointer data after converting the return value into an XML response string. When a function returns with errno $\neq$ 0, the return data is assumed invalid and the caller must not try to free it.

In this (contrived) example, a function that returns the last name of a person given the full first and last name as a string separated by a space (for example, “john doe” returns “doe”). Error checking is excluded for clarity.

```
// *WRONG*
// This disobeys both the Input and Output rules!
char* myService_getLastName(char* full_name)
{
   errno = 0;
   return strchr(full_name, ' ') + 1;
}
```

```
// *CORRECT*
char* myService_getLastName(char* full_name)
{
   char *c = strchr(full_name, ' ') + 1;
   errno = 0;
   return strdup(c);
}
```

See the test service code file (test.c) for examples on working with various data types including structs, arrays, and pairlists.

Events

Many services require the ability to send asynchronous events, status messages, or input needed requests. In the WDE, the concept of server-side events is not yet well supported. With this in mind, the Event Service was created.

The Event Service is no different than any other Service, but allows asynchronous communication between a client and server (in particular, from server to client). The Event Service is a utility for other Services and not direct application-level invocation. Applications do not call Event Service methods directly, but instead call appropriate wrapper methods provided by the Services using events.

The Event Service operates using Linux FIFOs. The FIFO provides a simple mechanism for the server to queue events and the client to read events without the need for sending constant messages from service client to server.

The Event Service provides the following interface:

```
struct event {
int            event;
unsigned int   size;
void*          data;
};

void         event_open(char* key);
void         event_close(char* key);
int          event_cancel(char* key);
struct event event_getEvent(char* key, int flags);
void         event_setEvent(char* key, struct event e);
```

An event in the above structure is defined as an event number, and may optionally include event data. For events without data, set `size` to zero and `data` to null.

Mechanics

An Event channel is identified by a key. All event methods require this key. The key is a standard C string (`char*`) of a maximum of 99 characters. Keys must be unique to all Services using events across the system. It is recommended that the Service name be part of the key. Since the client and server code are in one source file, the key is commonly `#define`'d at the top of the Service source file. For example:

```
#define EVENT1_KEY ("myServiceE1")
```

A Service can use multiple event channels, each requires a unique key. There is a system-wide limit of 256 event channels. An error (errno=`EFAULT`) returns from `event_open()` if this limit is reached.

An event channel is opened when `event_open()` is called and remains open and active until closed. When closed, the Event channel is cleaned from the system and any unread events in the queue are lost. A channel can be closed and reopened using the same key to flush the FIFO.

Any process with a valid key can make event method calls. The main intent is for one process to `setEvent()` calls, and another to `getEvent()` calls, but various protocols of event life cycle management are also possible.

The typical use scenario is the server-side is to open an event channel on initialization, and keeps the event open for the life of the server. The server sets events as they occur. Clients then receive events.

As seen by `struct event` an event is comprised of an event number (int) and optional event data. Set `event.size` to zero and `event.data` to null when no data is attached to the event. Otherwise, set the `event.data` pointer and `event.size`. `size` is the number of data bytes to extract on `getEvent()`. Ensure that `size` includes null-termination bytes of strings, and so on. Event data can be a maximum of 4000 bytes. All Service rules apply. Pointer data in both input and output parameters are owned by, and must be `free`'d by, the caller.

The `flags` parameter of method `event_getEvent()`, is a bit mask of any, all, or none of:

```
GETEVENT_NONBLOCK   (0x01)
GETEVENT_LAST       (0x02)
GETEVENT_CANCEL     (0x04)
```

Set `flags` to zero to perform a blocking `getEvent()`. Then waits for an event until either:

- an event is available in the queue.

  If there are events in the queue `getEvent()` immediately returns with the next event.
- the `getEvent()` is canceled, and `getEvent()` returns with errno=EINTR.

An active `getEvent()` can be canceled two ways:

- calling `event_cancel()`
- calling `getEvent()` with the `GETEVENT_CANCEL` flag bit set.

Only one `getEvent()` per key can be active at any time so a second call to `getEvent()` either:

- Returns errno=EBUSY and the `GETEVENT_CANCEL` flag bit is not set.

- Becomes the new in-progress `getEvent()` and the `GETEVENT_CANCEL` flag bit set. In this case, the previous `getEvent()` returns errno=`EINTR`.

The `GETEVENT_NONBLOCK` flag bit returns the next event in the queue if one exists or returns errno= `EFAULT` if null. In either case, the call is non-blocking and returns immediately.

The `GETEVENT_LAST` flag bit retrieves the event from the previous `getEvent()` call (most recently read message). This supports WDE where it is difficult to store state information. If an event is missed (for example, due to a web page change), the newly loaded web page can request the last read event and get back in sync with a Service. If `getEvent(last)` is called without a previous `getEvent()` (such as when the last event buffer is empty), errno=`EAGAIN` returns.

All methods, except `event_open()`, return errno=`EFAULT` when called with an unopened key. `event_open()` still opens the channel when presented with a unopened key, but does return errno=`EFAULT` when called with an already opened key.

In the diagram in Figure 5 `setEvent()` is in blue, and `getEvent()` is in red. Note that on `getEvent()`, the event data read from the FIFO is saved in Last Event, so the last event can be retrieved as many times as is necessary if control of the Service moves from one WDE page to another.

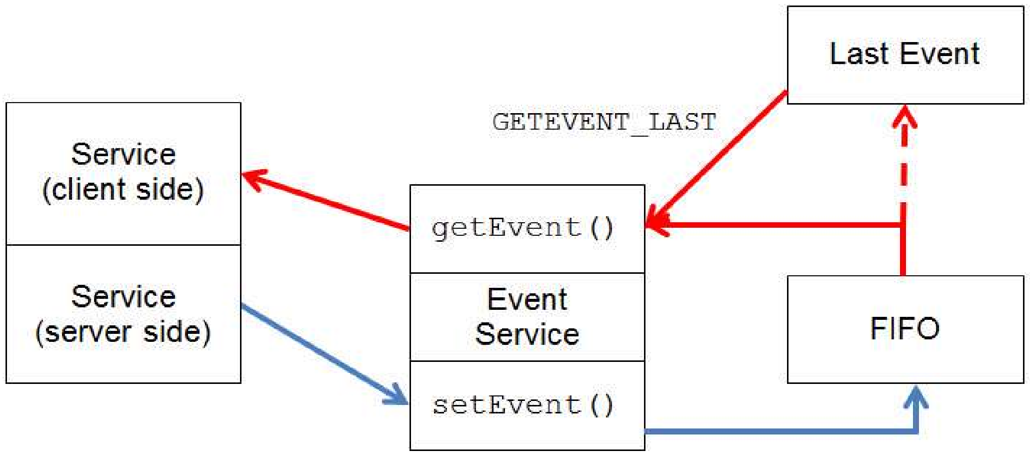


Figure 5 Event Flow

On the server-side, events are put on the event FIFO as they occur. If there is a pending `getEvent()`, then the requester retrieves the data and continues running. The data put on the event FIFO must be no longer than 4000 bytes. A FIFO can store approximately 64K of event data. This means that a number of events can be placed in the FIFO if requests for event data are infrequent.

svcmgrSvcDef.h

This header file is defined and maintained by Verifone to provide necessary infrastructure components. The file must be `#include`'d in the Service Interface header file as it contains the definition for `struct version`, which is returned by the `{service}_getVersion()` function required by every Service. `svcmgrSvcDef.h` also contains structures to support Pairlists.

Properties

An application that uses a Service (including WDE) can supply properties to the Service. Properties are `name/value` pairs read only by the Service, and designed to allow a caller to provide context. Properties are defined by the application, not the Service.

When writing a Service only the following properties are valid.

Property name	Property value
language	Web browser language identification code (for example, en-us).

The `svcmgrSvcInfConsumer.h` header file includes the API for retrieving property names and values. Since property names are predefined, a Service only needs to call:

```
char* smi_GetPropertyValue(const char* name);
```

The complete property API is:

```
// Callback prototype for use with smi_GetPropertyNames()
// 'name' is owned by the library, do not free.
typedef void (*cbPropName)(char* name, void* data);

// Returns each property name sequentially to callback.
// 'data' allows user defined data to be passed into callback, set to NULL
if unused.
void smi_GetPropertyNames(cbPropName callback, void* data);

// Returns NULL if property 'name' does not exist.
// The returned pointer (string) is owned by the library, do not free.
char* smi_GetPropertyValue(const char* name);
```

Example code using the property calls can be found in `test.c`, the C file in the test Service.

Required Interface Functions

Every Service is required to support a set of core interface functions to provide uniformity across Services. The Service precompiler checks for these functions and outputs a compliance error when a function is missing. Required functions are:

- `struct version {service}_getVersion(void);`

  where, `{service}` is the Service name.

  `struct version` is defined in `svcmgrSvcDef.h`.

NOTE

Currently, `{service}_getVersion()` is the only required function.

- `struct version {service}_getVersion(void);`

This function allows applications to query a Service for its version number. The returned structure is:

```
struct version {
       int major;
       int minor;
       int maint;
       char build[16];
    }
```

This structure describes a three dot version number (x.y.z) and optional build string.

NOTE

When returning an empty build string, set `build` to a null-terminated empty string (that is, `build[0] = 0;`).

Service Server Methods

To address requirements such as controlling and accessing hardware through a single process or single-thread processing, the Service Manager stack allows a single-server process to be associated with each Service that needs this type of control. As noted, use of a Server component is not restricted to hardware interfaces, and the developer can use this Server service for other purposes (for example, when single-thread and/or single processing is desired). To simplify Service development, a Server daemon is supplied and managed by the Service Manager.

All server-side Service methods are just methods within the Service (`.so`) itself that have a specified prototype. Service methods are invoked using other non server-side methods (considered client-side):

```
int svcclient_call_server_function(const char  * service_name,
const char *fn_name, const char *commandArgs, const int argsSize,
char **rtnData, int *rtnSize);
```

where:

`service_name`	Name of the service this function belongs to.
`fn_name`	Name of server-side method to call or invoke.
`commandArgs`	Pointer to any data to be passed to this server-side method (`string`, `int`, `array`, `struct`, and so on).
`argsSize`	Number of bytes of interest in `commandArgs`.
`rtnData`	Pointer to a pointer that contains a `malloc`'d chunk of memory containing the returned data.
`rtnSize`	Number of bytes of interest in the returned `malloc`'d data.

NOTE

It is the responsibility of the caller method to `free()` memory allocated for `rtnData`.

This method interfaces to the Service Manager through the service manager interface library. This function/method is prototyped in the auto-generated file, `autogen_service.h`, where `service` is the name of this Service. Auto-generated files are created running the autogen process for your Service at build time.

All server-side service methods must have the following prototype:

```
int service_myfn(const char *commandArgs, const int argsSize,
char **rtnData, int *rtnSize);
```

where

`service`	Name of this Service (for example, `printer`).
`myfn`	Sub-name of server-side method.
`commandArgs`	Pointer to any data to pass to this server-side method (`string`, `int`, `array`, `struct`, and so on).
`argsSize`	Number of bytes of interest in `commandArgs`.
`rtnData`	Pointer to a pointer that contains a `malloc`'d chunk of memory containing the returned data.
`rtnSize`	Number of bytes of interest in the returned `malloc`'d data.

NOTE

It is the responsibility of the server-side method to `malloc()` memory allocated for `rtnData`.

Special Server-side Methods

When a service must maintain state, you create the interface to a single user device using a server-side process. The server-side process is automatically created when the first call is made to a server-side method from the client-side method (see `svcclient_call_server_function()`).

For more control of service server process startup, use the special server-side method calls in this section within the server-side process.

int {service}_serverOnBoot(void);

At system boot, the service manager scans all installed services and looks for any service with a `{service}_serverOnBoot()`, where `{service}` is the name of the service method.

If the service finds this service, the service server process starts and `{service}_serverOnBoot()` executes within that process. This allows the service to have its server daemon process start at bootup.

NOTE

The return value is currently ignored.

Notes

1 This method should ONLY be used when a service server process must be started at bootup (controls fans, and so on). `{service}_serverInit()` is the preferred method.

2 If this method exists, a `{service}_serverInit()` is not executed at server process startup. However, the service can execute this method (for example, from `{service}_serverOnBoot()`).

3 Do not call another service from this method. Other services may have not started yet.

int {service}_serverInit(void);

On the first server-side request (when a client-side method makes a call to a server-side method), if the server process is not already active or running (that is, no `{service}_serverOnBoot()` method exists), a daemon process starts for the service. On start, the server-side process looks for and executes `init method {service}_serverInit()`, where `{service}` is name of the service, if it exists.

This method is preferred over `{service}_serverOnBoot()` for the following reasons:

- The service server process only starts or executes as required by the application running in the system.
- Boot time is not affected, since the server-side process is started on demand only.
- Development debug and testing is easier (that is, the unit does not have to be rebooted for testing).
- Do not call other services during this call, since it is running in a protected state.

int {service}_serverExit(void);

If a service server process receives a termination signal (SIGTERM) and if it exists, the `{service}_serverExit()` method executes prior to terminating the daemon process. Typically, a service server method is never terminated (an exception may be during development/testing).

Service Compilation Flow

Figure 6 shows the flow of file development/processing to create a Service (shared `.so` library). `{service}.c` and `svc_{service}.h` are Service developer source files. All others are either provided or generated by Verifone.

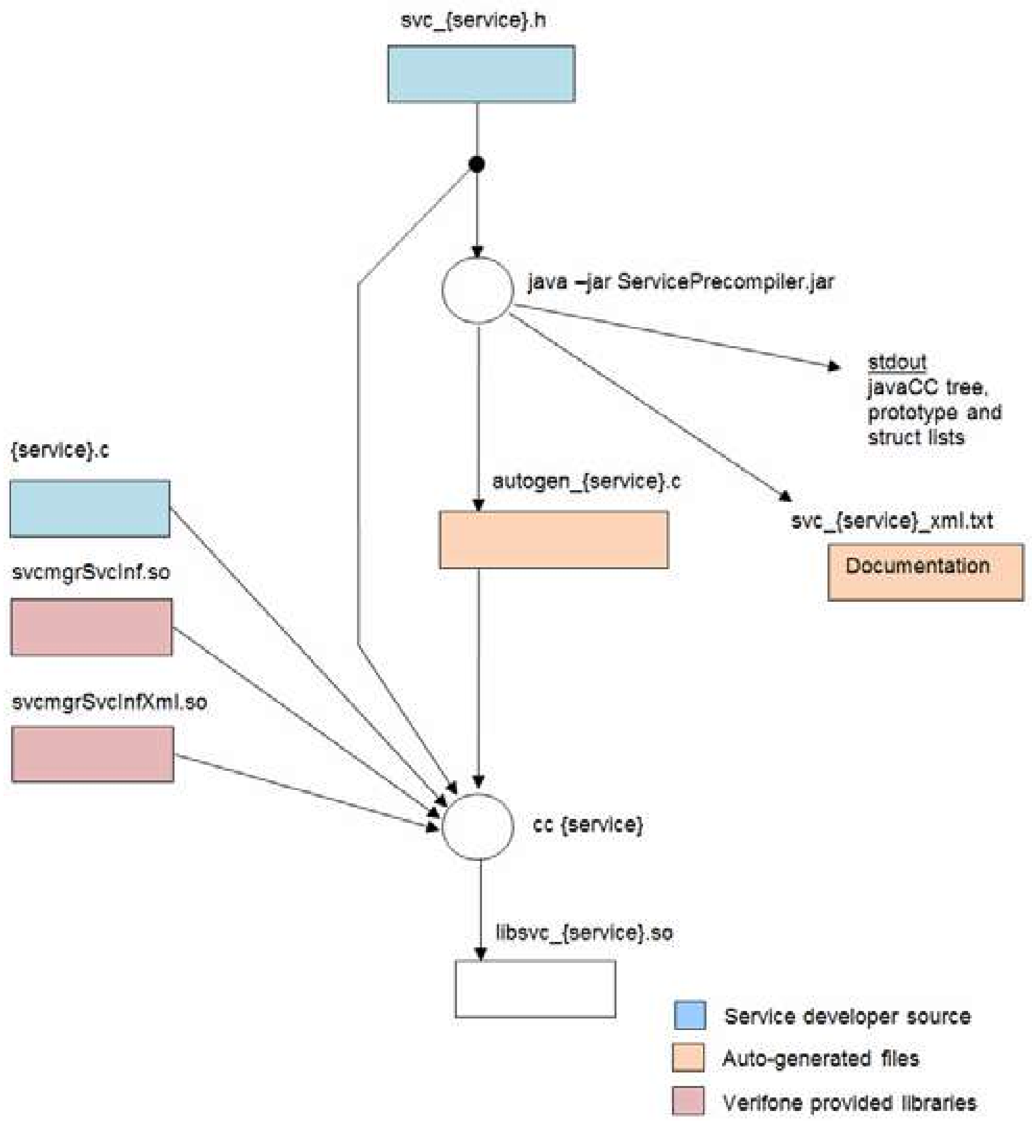


Figure 6 Service Development Flow

Building a Service

The diagram in Figure 6 depicts the process of building a simple client-side only Service. At the minimum Services require an interface definition (header file) and its implementation (C file).

The first step is running the Service precompiler. This Java utility uses the interface definition as input, and outputs a C file providing the links between the Service methods and WDE XML commands. Service Interface and the C Preprocessor provides command line syntax. Execute the Service precompiler as:

```
(CC) -E -C (Includes) svc_{service}.h | java
                                        -jar /usr/local/bin/ServicePrecompiler.jar
```

where:

`(CC)`	Path name to the C compiler, usually `arm-none-linux-gnueabi-cpp`.

`(Includes)`	List of '`-I`' paths to `#include`'d files not in the Service source directory. Include `-I/usr/local/include` for Verifone-provided header files.

The Service precompiler generates two files and outputs the Service Interface to `stdout`. After running the Service precompiler, examine the console output. The Service precompiler performs a number of compliance checks. When an interface fails a compliance check the auto-generated files are not created and a Compliance error is output. For example:

```
--------------------------
  Compliance Error:
  method return type is void* !

  method : led_on
--------------------------
```

When the interface file passes all compliance checks, the output files are generated and the interface is output to the console. It is important to verify that the console output interface is correct; this is the interface detected by the Service Precompiler and may be different than expected. For instance, if a comment tag directive is missing, the definition is not included in the interface, or a function prototype is not proceeded by `/*SVC_PROTOTYPE*/` is not seen in the console interface output and is not handled by the WDE autogen interface code.

The two auto-generated output files are:

`autogen_{service}.c`	Source file that must be included in the Service build. Do not edit this file. If the interface changes, rerun the Service precompiler to regenerate. This file provides the links between WDE XML commands and the Service C function calls.
`svc_{service}_xml.txt`	Simple ASCII file documenting all Service WDE command and response XML formats. This provides developers further documentation.

After you are satisfied the interface is correctly processed, build the Service library. The following required files are included in a Service compile:

`svc_{service}.h`	Your Service interface header file.
`{service}.c`	Your Service implementation file. There may several.
`autogen_{service}.c`	Output file of Service precompiler.
`/usr/local/include/svcmgrSvcInfConsumer.h`	Constants and APIs used by `autogen_{service}.c`, needed by `{service}.c` if accessing properties.

`/usr/local/lib/libsvcmgrSvcInf.so`	Shared library to implement Service Interface functionality.
`/usr/local/lib/libsvcmgrSvcInfXml.so`	Shared library to implement XML functionality used by the autogen code.
`/usr/lib/libxml2.so`	Shared library used by `libsvcmgrSvcInfXml.so`.
`/usr/local/include/svcmgrSvcDef.h`	Provides definitions for `struct version` and `pairlist` elements

Compile the sources files to build the `libsvc_{service}.so` shared library:

```
(CC) -c -I/usr/include/libxml2 -I/usr/local/include -fPIC (SrcFiles)
(CC) -shared -lxml2 -lsvcmgrSvcInf -lsvcmgrSvcInfXml
     -o libsvc_{service}.so (ObjFiles)
```

Service Testing

Use the west tool to perform services testing.

WDE Environment Service Tester (west)

west allows direct testing of Nexgen Services, bypassing the browser and server (`lighttpd`). It works with command and test XML files to interact with a service. To use west, develop a set of command and/or test files that exercise the specified service functionality.

Starting west and Command Line Options

west is a console application with several optional command line arguments. Command line usage is output using the `-h` help option.

```
$ west -h

   Web Environment Service Tester
   Provides testing of WDE Services without browser & server.

   Usage :

   west [-r cmd:fname] [-e] [-f] [-s] [-l logfname] [-t] [-i
count[:delay]] [-h]

      -r  - optional - run an execute command on program startup.
      -e  - optional - exit program without displaying command prompt.
      -f  - optional - format xml output, default is no whitespace.
      -s  - optional - suppress messages to stdout.
      -l  - optional - write messages to log file.
      -t  - optional - create a test file from a cmd file.
      -i  - optional - run each cmd 'count' times, opt delay each run
      (millisec)
      -h  - optional - Print this usage.
   valid -r 'cmd' values : cmd, cmdlist, test, testlist
```

Command Prompt

All command line arguments are optional. When west starts normally the command prompt displays:

```
Enter Command [ m for menu ] :
```

Enter `m` to display the menu (output shown below). To keep the screen uncluttered, the menu is not output at the command prompt.

```
Commands:
   [1] cmd      ; Run svc_cmd file
   [2] cmdlist  ; Run svc_cmd list file
   [3] test     ; Run svc_test file
   [4] testlist ; Run svc_test list file

   [5] Toggle xml formatting
   [6] Change iterate count:delay
   [9] Run last command (cmd:myCmdFile)

   [0] Exit
```

Enter a command number at the prompt.

- Entering 1, 2, 3, or 4 prompts for an executable filename, including the full pathname to the file if not in the same directory.
- Option 5 toggles XML formatting between `no whitespace` and `indented xml`. This is equivalent to the `-f cmdline` option.
- Option 9 only exists after a command is entered.
- Any entry other than m, 1, 2, 3, 4, 5, 6, or 9 exits.

File Types and Formats

The four west run commands, `cmd`, `cmdlist`, `test`, and `testlist` use one of two file formats. `cmd` and `test` commands use a sectioned XML file. `cmdlist` and `testlist` commands use a file that lists a set of cmd/test files.

cmd and test Command File Formats

`cmd` and `test` commands execute an XML file that can contain up to three separate sections:

- an optional `properties xml` section,
- a `cmd xml` section, and
- a `rtn xml` section.

The `rtn xml` section is not used when executing `cmd` commands, but can exist in the file. This allows a single file to be used for both `cmd` and `test` commands.

Each `xml` section must be separated by at least one carriage return <CR>.

Order the sections as `properties xml`, then `cmd xml`, and lastly `rtn xml`.

`properties xml` has property name/value pairs contained within a `<svc_properties>` parent:

```
<svc_properties>
     <propName1>propValue1</propName1>
     <propName2>propValue2</propName2>
     ...
     <propNameN>propValueN</propNameN>
   </svc_properties>
```

Refer to XML Command and Return Examples for detailed information on the `<svc_cmd>` and `<svc_rtn>` XML formats. In general, they are of the form:

```
<svc_xxx>
     <myServiceName>
       <myMethodName>
         ... Method Data ...
       </myMethodName>
     </myServiceName>
   </svc_xxx>
```

Note that in the WDE system, the properties provided by the browser furnish information on settings a Service may need to react to, such as locality (language). Refer to WDE Environment Service Tester (west) for the list of valid WDE properties.

cmdlist and testlist Command File Format

This file list any number of cmd or test file pathnames, one filename per line. Each cmd/test file pathname must be either the full pathanme or relative to the west executable. Use this to sequentially execute many cmd or test commands.

Iterating (-i cmdline option)

The `-i cmdline` option (option 6 menu) provides looping run commands. The iterate settings format is `count[:delay]`.

`count` is an integer that specifies the number of times to rerun each command. Set `count` to zero to specify it run infinitely. The only way to stop running infinitely is to kill the process (such as using Ctrl-C).

`delay` (optional int) specifies the amount of time (in milliseconds) to pause between each loop. When not specified, the default value is zero; no delay between runs. The default iterate setting is 1:0 – run once and no delay.

Automated Testing

The majority west command line parameters enable hands-off testing. For example, a full regression test could be accomplished with a set of `test` command files and a single `testlist` command file listing them. Then issue the following west command:

```
$ west -r testlist:myTestlistFile -e -s -l testLogFile
```

to execute west and issue the `testlist` command (option 4), using `myTestlistFile`.

`-e`	Exit west on completion of this command (that is, do not display the command prompt).
`-s`	Suppress output to stdout; as the terminal is not interactive, there is no reason to have it output to the screen.
`-l`	Write all output to the specified log file; in this case, `testLogFile`.

Auto-create Test Files from a cmd File

The `-t` command line option tells west to create a test file when executing `cmd`. This greatly simplifies the creation of test files.

First create a `cmd` command file with a `<svc_cmd>` section and optionally a `<svc_properties>` section. Then run west with the `-t` option and execute `cmd`. A new file named `test_myCmdFilename` (`test_` prefixed to the original `cmd` filename) is created that contains the original `cmd` followed by the XML output returned by the Service.

Use this file as a test command file. It is best to have a Service function fully functional so that the `rtn` output is correct, but the `test` command file can always be edited or recreated if a function return changes.

Example 1

This is a simple Service function and combined `cmd`/`test` command file (with no properties section).

function:

```
int math_square(int x)
   {
      errno = 0;
      return (x*x);
   }
```

cmd/test file:

```
<svc_cmd>
     <math>
       <square>
         <x>5</x>
```

```
    </square>
  </math>
</svc_cmd>

<svc_rtn>
  <math>
    <square>
      <return>25</return>
    </square>
  </math>
</svc_rtn>
```

SCGI Interface to Service Manager

This section details the requirements, architecture, and design of the Simple Common Gateway Interface (SCGI) interface to the Services of the Service Manager stack.

The SCGI interface consists of an SCGI control daemon and supporting interface library. This library is used by the http server (`lighttpd`), a PHP extension, and the Stack Test Harness.

Requirements

Following is a list of requirements, starting with the SCGI daemon manager, and then the supporting library.

SCGI daemon manager

- Interfaces directly with `lighttpd` CGI module.
- Must manage multiple forked CGI processes to handle simultaneous calls to services (multi-process), including:
  - Track how many processes are running.
  - Track which processes are actively processing commands/requests.
  - Restrict forked processes to a maximum specified number.
- Processes remain active between Service calls to minimize overhead (performance issues) of forking and service class lookup (dynamic loading of service libraries).
- When maximum simultaneous processes are active, block new requests for a specified time, then return an error if a open slot is not available.
- Monitor forked process failures (crashes) and when detected, return a error response to the caller (XML error response).
- Gather incoming command data (XML format), and obtain the language type from the SCGI interface.
- Place language (from the web interface) in a name/value pairlist along with the command data interfaces to the support library.
- Return response data (XML) from the support library back to caller (`lighttpd` SCGI module). No processing required.

SCGI Support Library

- Called by SCGI forked daemon processes, PHP extension, and test harness.
- Does NOT link with the service libraries. Services can be removed and added without affecting the support library and/or the SCGI daemon processes.
- Parses passed XML command to determine which Service the request is for.
- Dynamically loads the service library.
- Call the specified service interface method, passes the XML command data and the name/value pairlist.
- Returns the return response from the Service Interface (XML) unaltered.

Architecture

The Service Stack allows C applications, http server, or PHP to interface the Service methods. Figure 7 illustrates the Service Stack architecture.

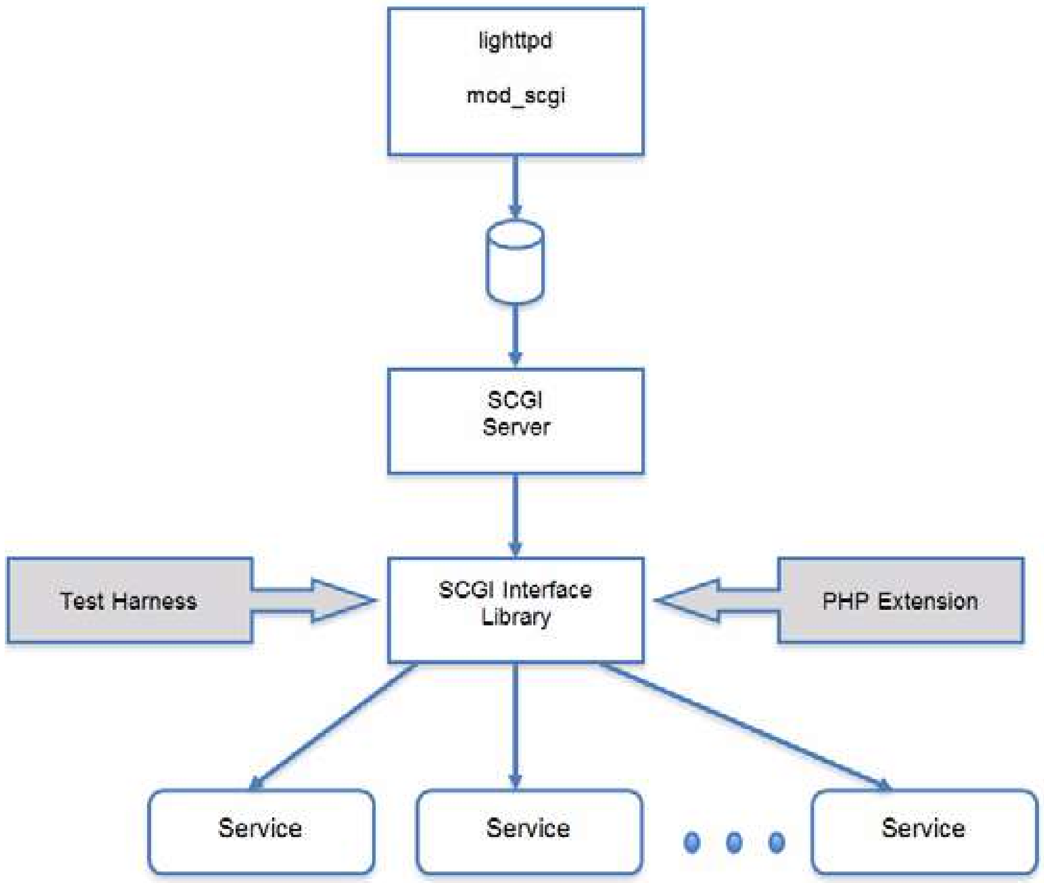


Figure 7 Service Stack Architecture

Service Utility

This section provides information on the Service utility.

XML Interface

This section contains the C and XML command and return format for each exposed method in the Service utility.

utility getVersion

C Prototype

```
struct version utility_getVersion(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <getVersion/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <getVersion>
         <return type="container">
         <major>int</major>
         <minor>int</minor>
         <maint>int</maint>
         <build type="string">string [max len 16]</build>
         </return>
      </getVersion>
   </utility>
</svc_rtn>
```

utility gettime

C Prototype

```
char * utility_gettime(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <gettime/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <gettime>
         <return type="string">string</return>
      </gettime>
   </utility>
</svc_rtn>
```

utility settime

C Prototype

```
int utility_settime(struct utilityDateTime * utildt /*NULL*/);
```

Command XML

```
<svc_cmd>
   <utility>
      <settime>
         <utildt type="list">
            <item  type="container">
               <tm_sec>int  [default:0]</tm_sec>
               <tm_min>int  [default:0]</tm_min>
               <tm_hour>int [default:0]</tm_hour>
               <tm_mday>int  [default:0]</tm_mday>
               <tm_mon>int  [default:0]</tm_mon>
               <tm_year>int [default:0]</tm_year>
               <tm_wday>int  [default:0]</tm_wday>
               <tm_yday>int  [default:0]</tm_yday>
               <tm_isdst>int [default:0]</tm_isdst>
            </item>
         </utildt>
      </settime>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <settime>
         <return>int</return>
      </settime>
   </utility>
</svc_rtn>
```

utility copyTime linuxRTC

C Prototype

```
int  utility_copyTime_linuxRTC(void);
```

Return Values

0 = Success

-1 = Failure

Command XML

```
<svc_cmd>
   <utility>
      <copyTime_linuxRTC/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <copyTime_linuxRTC>
         <return>int</return>
      </copyTime_linuxRTC>
   </utility>
</svc_rtn>
```

utility reboot

C Prototype

```
int utility_reboot(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <reboot/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <reboot>
         <return>int</return>
      </reboot>
   </utility>
</svc_rtn>
```

utility getenvfile

C Prototype

```
struct utilityGetEnvFile utility_getenvfile(char * section /*NULL*/,
                                            char * name /*NULL*/);
```

Command XML

```
<svc_cmd>
   <utility>
      <getenvfile>
         <section  type="string">string  [default:NULL]</section>
         <name type="string">string [default:NULL]</name>
      </getenvfile>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <getenvfile>
         <return type="container">
            <count>int</count>
            <value type="string">string [max len 512]</value>
         </return>
      </getenvfile>
   </utility>
</svc_rtn>
```

utility getenvfilename

C Prototype

```
struct utilityGetEnvFile utility_getenvfilename(char * pathname /*NULL*/,
                                                char * section /*NULL*/, c
```

Command XML

```
<svc_cmd>
   <utility>
      <getenvfilename>
         <pathname type="string">string [default:NULL]</pathname>
         <section type="string">string  [default:NULL]</section>
         <name type="string">string   [default:NULL]</name>
      </getenvfilename>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <getenvfilename>
         <return type="container">
            <count>int</count>
            <value type="string">string [max len 512]</value>
         </return>
      </getenvfilename>
   </utility>
</svc_rtn>
```

utility putenvfile

C Prototype

```
int utility_putenvfile(char * section /*NULL*/, char * name /*NULL*/,
                       char * value /*NULL*/);
```

Command XML

```
<svc_cmd>
   <utility>
      <putenvfile>
         <section  type="string">string  [default:NULL]</section>
         <name type="string">string   [default:NULL]</name>
         <value type="string">string [default:NULL]</value>
      </putenvfile>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <putenvfile>
         <return>int</return>
      </putenvfile>
   </utility>
</svc_rtn>
```

utility putenvfilename

C Prototype

```
int utility_putenvfilename(char * pathname /*NULL*/, char * section /
                          *NULL*/, char * name /*NULL*/,
```

Command XML

```
<svc_cmd>
   <utility>
      <putenvfilename>
         <pathname type="string">string [default:NULL]</pathname>
         <section type="string">string  [default:NULL]</section>
         <name type="string">string   [default:NULL]</name>
         <value type="string">string [default:NULL]</value>
      </putenvfilename>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <putenvfilename>
         <return>int</return>
      </putenvfilename>
   </utility>
</svc_rtn>
```

utility iniContents

C Prototype

```
struct utilityIniContents utility_iniContents(char * pathname /*NULL*/);
```

Command XML

```
<svc_cmd>
   <utility>
      <iniContents>
         <pathname type="string">string [default:NULL]</pathname>
      </iniContents>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <iniContents>
         <return type="container">
            <sections type="list">
               <item  type="container">
               <section type="string">string</section>
                  <entries type="container">
                     <list type="pairlist">
                        <nameN>valueN</nameN>
                        <!-- ... -->
                     </list>
                     <list_count>int</list_count>
                  </entries>
               </item>
               <!-- ... list <item> count defined by sections_count -->
            </sections>
            <sections_count>int</sections_count>
         </return>
      </iniContents>
   </utility>
</svc_rtn>
```

utility installfile

C Prototype

```
struct installStatus utility_installfile(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <installfile/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <installfile>
         <return type="container">
            <result>int</result>
            <msg  type="string">string  [max  len  256]</msg>
         </return>
      </installfile>
   </utility>
</svc_rtn>
```

utility runApplication

C Prototype

```
struct installStatus utility_runApplication(char * app /*NULL*/);
```

Command XML

```
<svc_cmd>
   <utility>
      <runApplication>
         <app  type="string">string  [default:NULL]</app>
      </runApplication>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <runApplication>
         <return type="container">
            <result>int</result>
            <msg  type="string">string  [max  len  256]</msg>
         </return>
      </runApplication>
      </utility>
</svc_rtn>
```

utility externalStorage

C Prototype

```
struct utilityExternalStorage utility_externalStorage(void);
```

Return Values

A list of available external devices.

Command XML

```
<svc_cmd>
   <utility>
      <externalStorage/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <externalStorage>
         <return type="container">
            <storage_count>int</storage_count>
            <storage type="list">
               <item  type="container">
                  <type>int</type>
                  <mountPoint type="string">string [max len 64]
                  </mountPoint>
               </item>
               <!-- ... list <item> count defined by storage_count -->
            </storage>
         </return>
      </externalStorage>
   </utility>
</svc_rtn>
```

utility displayBrightness

C Prototype

```
int utility_displayBrightness(int level /* 50 */);
```

Command XML

```
<svc_cmd>
   <utility>
      <displayBrightness>
         <level>int  [default:  50  ]</level>
      </displayBrightness>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <displayBrightness>
         <return>int</return>
      </displayBrightness>
   </utility>
</svc_rtn>
```

utility fbgrab

C Prototype

```
int utility_fbgrab(char * pngfile /*""*/, int interlace /* 0  */,
                  int x  /*0*/, int y  /*0*/, int w  /*0);
```

Parameters

`pngfile`	Full pathname of PNG image file to produce from grab of framebuffer content.
`interlace`	Enables interlacing in PNG output file
`x`	X-origin to start screen capture from.
`y`	Y-origin to start screen capture from.
`w`	Width in pixels of screen capture (0 causes full width starting from x).
`h`	Height in pixels of screen capture (0 causes full height starting from y).

Return Values

0 = Successful request

-1 = Unable to complete request.

Notes:

- If pngfile is NULL or empty, default "/tmp/screenshot.png" will be used.
- All output capture files are of type png, regardless of name extension specified.
- To get display pixel (w,h), see svc_sysinfo.h - sysinfo_platform(); On error return, errno:
  - EINVAL if the input params are invalid (negative or out of range).
  - EACCES unable to create/open for write, pngfile as specified.
  - EBADF unable to write to pngfile (no space?).

Command XML

```
<svc_cmd>
   <utility>
      <fbgrab>
         <pngfile type="string">string [default:""]</pngfile>
         <interlace>int  [default:  0  ]</interlace>
         <x>int [default:0]</x>
         <y>int [default:0]</y>
         <w>int [default:0]</w>
```

```
        <h>int [default:0]</h>
      </fbgrab>
    </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <utility>
      <fbgrab>
        <return>int</return>
      </fbgrab>
    </utility>
</svc_rtn>
```

utility isCgiWDE

C Prototype

```
int utility_isCgiWDE(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <isCgiWDE/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <isCgiWDE>
         <return>int</return>
      </isCgiWDE>
   </utility>
</svc_rtn>
```

utility isSourceTrustedWDE

C Prototype

```
int utility_isSourceTrustedWDE(void);
```

Command XML

```
<svc_cmd>
   <utility>
      <isSourceTrustedWDE/>
   </utility>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <utility>
      <isSourceTrustedWDE>
         <return>int</return>
      </isSourceTrustedWDE>
   </utility>
</svc_rtn>
```

Data Structures

This section presents the Service utility data structures.

installStatus Struct Reference

Returns install status.

```
#include <svc_utility.h>
```

Data Fields

```
int result
```

- 0 = Install complete; no reboot
- 1 = Install complete; reboot required
- < 0 = Error

```
char msg [256]
```

- Install error message (only valid when result < 0)

int installStatus::result

- 0 = Install complete; no reboot
- 1 = Install complete; reboot required
- < 0 Error

The documentation for this struct was generated from the following file:

- svc_utility.h

utilityAnExternalStorage

Return information on a single external storage device.

```
#include <svc_utility.h>
```

Data Fields

- int type – Storage media type:
  - 1=USB
  - 2 = microSD

```
int  utilityAnExternalStorage::type
```

- char mountPoint [64] – Mount point.

Mount point.

```
char utilityAnExternalStorage::mountPoint[64]
```

utilityDateTime

Replica of the Linux structure `tm` in time.h.

```
#include <svc_utility.h>
```

Data Fields

- int tm_sec – seconds

```
int utilityDateTime::tm_sec
```

- int tm_min – minutes

```
int utilityDateTime::tm_min
```

- int tm_hour – hours

```
int utilityDateTime::tm_hour
```

- int tm_mday – day of the month

```
int  utilityDateTime::tm_mday
```

- int tm_mon – month

```
int utilityDateTime::tm_mon
```

- int tm_year – year

```
int utilityDateTime::tm_yday
```

- int tm_wday – day of the week

```
int  utilityDateTime::tm_wday
```

- int tm_yday – day in the year

```
int utilityDateTime::tm_year
```

- int tm_isdst – daylight saving time

```
int utilityDateTime::tm_isdst
```

utilityExternalStorage

Returns information on all available external storage devices.

```
#include <svc_utility.h>
```

Data Fields

- int storage_count

```
int utilityExternalStorage::storage_count
```

- struct utilityAnExternalStorage?

```
struct utilityAnExternalStorage? utilityExternalStorage::storage
```

utilityGetEnvFile

Obtain value for name in section within user config ini.

```
#include <svc_utility.h>
```

C Prototype

```
utility_getenvfile(char * section, char * name);
```

Parameters

`section`	Section of interest.
`name`	Name to get value for.

Return Values

```
struct getEnvFile
```

Notes:

- On getEnvFile.count < 0 return, errno:
  - ENOENT when name entry not found in section specified in config ini file.

For XML interface, see xml:utility_getenvfile

Data Fields

- int count
  - >=0 – section/name exists in envFile,
  - < 0 Error

```
int utilityGetEnvFile::count
```

- char value [UTILITY_MAX_VALUE_LEN] – Environment variable for section and name (only valid when `int count` > 0)

```
char utilityGetEnvFile::value[UTILITY_MAX_VALUE_LEN]
```

utilityIniContents

Retrieves the complete contents of a Config (ini) file.

```
#include <svc_utility.h>
```

Data Fields

- struct utilityIniSection * sections – The array of ini section data.

```
struct utilityIniSection * utilityIniContents::sections
```

- int sections_count – The number of ini sections (array len)

```
int utilityIniContents::sections_count
```

utilityIniSection

Retrieves the contents of a single section of a Config (ini) file.

```
#include <svc_utility.h>
```

Data Fields

- char * section – The section name.

```
char * utilityIniSection::section
```

- struct pairlist entries – The name/value pairs under section.

```
struct pairlist utilityIniSection::entries
```

File Structures

This section presents the Service utility file structures.

svc utility.h

The utility service interface methods file. The utility service contains general purpose (non-category) methods for doing things such as get system time, reboot, and so on.

```
#include  "svcmgrSvcDef.h"
```

Data Structures

- `struct installStatus` – Returns the install status.
- `struct utilityGetEnvFile` – Returns getEnvFile.
- `struct utilityIniSection` – Returns the contents of a single section of a Config (ini) file.
- `struct utilityIniContents` – Returns the complete contents of a Config (ini) file.
- `struct utilityAnExternalStorage` – Returns information on a single external storage device.
- `struct utilityExternalStorage` – Returns information on all available external storage devices.
- `struct utilityDateTime` – Replica of the Linux structure tm in time.h.

Defines

- #define UTILITY_MAX_LABEL_LEN (32)
- #define UTILITY_MAX_VALUE_LEN (512)

External Storage Types

- #define EXTERNAL_STORAGE_USB 1 – USB memory device.
- #define EXTERNAL_STORAGE_MICRO_SD 2 – MicroSD memory device.

Functions

- `struct version utility_getVersion (void)` – Retrieves the service version.

Return Values

Structure version containing version.

For XML interface, see xml:utility_getVersion

- `char * utility_gettime (void)` – Obtains the system time.

Return Values

malloced string containing the system time or NULL on error.

- On NULL return, errno:
  - ENOMEM – Unable to malloc space for return string.

For XML interface, see xml:utility_gettime.

- `int utility_settime (struct utilityDateTime *utildt)` – Sets the system time and RTC.

Parameters

`time`	The structure utilityDateTime

Return Values

- = 0 – Success
- = -1 – Failure

For XML interface, see xml:utility_settime

- `int utility_copyTime_linuxRTC (void)` – Copies the Linux time to RTC.
- `int utility_reboot (void)` – Request a system reboot.

Return Values

- = 0 – successful request
- = -1 – unable to complete request.
- On error return, errno:
  - ECHILD when unable to fork child process to handle reboot request.

For XML interface, see xml:utility_reboot

- `struct utilityGetEnvFile utility_getenvfile (char *section, char *name)` – Obtains the value for name in section within user config ini.
- `struct utilityGetEnvFile utility_getenvfilename (char *pathname, char *section, char *name)` – Obtains the value for name in section within specified ini.

Parameters

`pathname`	The full pathname of ini file, reverts to _getenvfile() when NULL.
`section`	Section of interest.
`name`	Name to get value for.

Return Valutes

`getEnvFile`

- On getEnvFile.count < 0 return, errno:
  - ENOENT when name entry not found in section specified in config ini file.

For XML interface, see xml:utility_getenvfile

- `int utility_putenvfile (char *section, char *name, char *value)` – Sets the value for name in section within user config ini.

Parameters

`*section`	Section of interest.
`*name`	Name to set value for.
`*value`	Value to set.

Return Values

- = 0 – Successful request
- = -1 – unable to complete request.

NOTE

`section` must be specified. If `value` is specified, `name` must be specified. If name is specified, and value == empty, the entry is removed.

- On error return, errno:
  - EINVAL – The above combinations are not met.
  - EACCES – Unable to set name in section in env file.

  For XML interface, see xml:utility_putenvfile

- `int utility_putenvfilename (char *pathname, char *section, char *name, char *value)` – Sets the value for name in section within specified ini.

Parameters

`pathname`	The full pathname of ini file, reverts to _putenvfile() when NULL.
`section`	Section of interest.
`value`	Value to set.

Return Values

= 0 – Successful request

= -1 – unable to complete request.

NOTE

`section` must be specified. If `value` is specified, `name` must be specified. If name is specified and `value` == empty, the entry is removed.

- On error return, errno:
  - EINVAL if the noted above combinations are not met.
  - EACCES unable to set name in section in env file.

For XML interface, see xml:utility_putenvfile

- `struct utilityIniContents utility_iniContents (char *pathname)` – Obtains the value for name in section within user config ini.

Parameters

`pathname`	The full pathname of ini file to retrieve contents.

Return Values

`struct utilityIniContents`

- On error return, errno:
  - ENODATA – The filename is NULL or empty.
  - ENOENT – The filename not found or could not be opened.
  - ENOMEM – Out of memory.
  - EFAULT – Internal error is preventing successful completion.

- `struct installStatus utility_installfile (void)` – Installs the file/package in the system. For XML interface, see xml:utility_installfile.

Return Values

- = 0 – not from WDE cgi process
- = 1 – from WDE cgi process.

- `struct installStatus utility_runApplication (char *app)` – Runs an application.

Parameters

app	Executable to start, set to NULL to exit system mode and start default applications.

Return Values

- `installStatus` – The install status code and error string (if appropriate)
- = 0 – success
- <0 error (see installStatus).

For XML interface, see xml:utility_installfile

- `struct utilityExternalStorage utility_externalStorage (void)` – External storage.
- `int utility_displayBrightness (int level)` – Set or gets the display brightness (backlight) level (0 to 100).
  - 0 = off
  - 100 = max
  - 50 = typical
  - < 0 = current level returned

- If >= 0 current level returned (post level set); If = -1 then error, check errno:

Return Values

- EINVAL – Input level is > 100.
- ENOENT – Unable to open display brightness control device for read.
- EPERM – Unable to open display brightness control device for write.
- EBADF – Write of new brightness level failed.

NOTE Due to translation of user levels to hardware levels, setting brightness to a level may result in another level value returned when successful. If the brightness is decreased, a translated value is used that will decrease the brightness (down to minimum allowed) that may result in a returned value different from what was requested. The same is true when increasing brightness.

- `int utility_fbgrab (char *pngfile, int interlace, int x, int y, int w, int h)` – Takes a screenshot using the framebuffer
- `int utility_isCgiWDE (void)` – Requests if the source is from a WDE cgi process.
- `int utility_isSourceTrustedWDE (void)` – Requests if the WDE source is trusted.

Return Values

- = 0 – from WDE and source is not trusted (unsigned)
- = 1 – Success. Trusted WDE source or any other source.
- errno == 0 always when returning from this method.

int utility_copyTime_linuxRTC ()

CHAPTER 8

Input Events

V/OS terminals support input events as captured through the Linux kernel Input Event module. The Input Events module supports the USB Human Interface Device (HID), including the keyboard and scanner. The mouse device is not fully implemented at this time. There is full support for the keyboard and scanner devices.

Input Event Functions

These calls control the USB HID and capture event data.

inputOpen()

Opens the USB device. Only one device can be open at a time.

Prototype

```
int inputOpen(int vfi_device);
```

Parameters

`vfi_device`	The device desired to open as defined in `vfiInputAPI.h` header file as follows: • `VFI_USB_KBD` • `VFI_USB_SCANNER`

Return Values

> 0	Successful execution corresponding to the handle of the opened device.
-1	And errno set to: • -EINVAL = More than one or no devices plugged in to the USB host port. • -EBUSY = One device plugged in to the USB port opened successfully; opening subsequent devices was attempted but failed.

inputRead()

Capture read events. Currently, touchpad and mouse events are not captured. Only keyboard and scanner data is read. If `inputRead()` is called in a loop, data is continually read from the keyboard or scanner device, allowing data to be captured as it is entered/scanned.

Prototype

```
int inputRead(int inHdl);
```

Parameters

`inHdl`	The device handle obtained from opening the device with `inputOpen()`.

Return Values

Returns the ASCII value or the raw scancode of the key pressed or data scanned.

0	No data read from the device.
Negative value	The device corresponding to the handle cannot be found.

inputClose()

Close the device. When the device is closed, lit LEDs are shut off.

Prototype

```
int inputClose(int inHdl);
```

Parameters

`inHdl`	The device handle obtained from opening the device with `inputOpen()`.

Return Values

0	Success
Negative value	An error occurred while trying to close the device.

CHAPTER 9

Device Drivers

The following device drivers are supported by V/OS terminal:

Ethernet Port	Standard Linux support as `/eth0`
Serial Ports	Standard Linux support as `/dev/ttyAMA0`, `ttyAMA1`, `ttyAMA2`
USB	Standard Linux support as `/dev/usb1 /dev/usb2` USB device serial port is defined as: COM5 = `/dev/ttyGS0`
Display	Standard Linux support as `/dev/fb0`
Magnetic Stripe Reader	Verifone unique device `/dev/msr`
Smart Card	Verifone unique device `/dev/scdrv`
Touch panel	Standard Linux `/dev/input/mice`
Printer	Verifone unique device `/dev/ltp`

Details on the standard Linux drivers are available from the open source community. The succeeding sections will detail the Verifone specific drivers.

CHAPTER 10

Display

The V/OS supports the display of all devices on all platforms. The V/OS display backlight supports 64 levels of intensity. Setting the brightness to 0 turns off the backlight. The legacy API `dspSetBrightness()` changes the backlight intensity to two counts with each call for compatibility.

Note that if legacy applications directly set the backlight intensity using the *BACKLIGHT configuration variable, that value must be scaled (from 1–63) before being set.

Refer to the hardware guide for your terminal for display resolution values.

Display Functions

Use the following function calls to manage the V/OS display.

dspSetBrightness()

Adjusts display brightness by using repeated calls to `dspSetBrightness()`.

NOTE Each call to `dspSetBrightness()` adjusts the level by two steps. If the `*BACKLIGHT` configuration variable is set directly using the application or config file, then the value must be between 1 and 63.

On power up, the brightness is automatically set. Set the `*BACKLIGHT` environment variable (1–63) to permanently change the backlight from its default value.

NOTE When `direction` = 0, the backlight turns off.

Prototype

```
short dspSetBrightness(short direction);
```

Parameters

`direction`	0	Decrease brightness.
	1	Increase brightness.

Return Values

= 0	OK.
< 0	Brightness at max/min setting.

CHAPTER 11

LEDs

V/OS-based terminals have LEDs to inform the user that they can swipe their card. V/OS supports all types of LEDs on devices of all platforms. LEDs are terminal-specific. V/OS supports the following LED configurations:

- Keypad
- MSR
- Smart card
- CTLS
- Device logo
- System indication

LEDs are controlled by the application using libled library. More complex use of these LEDs requires a thread or timer for synchronization. Use the ecore timers available in the GUI library.

NOTE

For legacy applications, if your application directly addresses the LEDs, update the calls to use the definitions found in svc.h.

LED Functions

This section presents LED function call descriptions.

Led_ledOn()

Enables the LEDs. A bit mask selects which LED(s) to turn on.

Prototype

```
Enum result led_ledOn(int leds);
```

Parameters

`leds`

`LED_MSR`	MSR top LED
`LED_MSR1`	MSR middle LED
`LED_MSR2`	MSR bottom LED
`LED_KP`	Keypad LED
`LED_LOGO`	Logo LED
`LED_BOOT_ERROR`	Boot LED
`LED_SC`	Smartcard LED
`CTLS_FIRST`	First CTLS LED
`CTLS_SECOND`	Second CTLS LED
`CTLS_THIRD`	Third CTLS LED
`CTLS_FOURTH`	Fourth CTLS LED
`LED_SYSTEM_GREEN`	System LED
`LED_SYSTEM_RED`	System LED

To set multiple parameters, use a pipe between parameters, for exampe:

```
leds = LED_MSR1|LED_MSR2|LED_MSR3
```

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };.
```

led_ledOff()

Disables one or more LEDs. A bit mask selects which LED(s) to turn off.

Prototype

```
void led_ledOff(int leds);
```

Parameters

`leds`

`LED_MSR`	MSR top LED
`LED_MSR1`	MSR middle LED
`LED_MSR2`	MSR bottom LED
`LED_KP`	Keypad LED
`LED_LOGO`	Logo LED
`LED_BOOT_ERROR`	Boot LED
`LED_SC`	Smartcard LED
`CTLS_FIRST`	First CTLS LED
`CTLS_SECOND`	Second CTLS LED
`CTLS_THIRD`	Third CTLS LED
`CTLS_FOURTH`	Fourth CTLS LED
`LED_SYSTEM_GREEN`	System LED
`LED_SYSTEM_RED`	System LED

Return Values

None.

led_init()

Initialize the LED file descriptor.

Prototype

```
int led_ledinit();
```

Parameters

None

Return Values

= 0	Success
= -1	Failure

led_release()

Closes the LED file descriptor.

Prototype

```
int led_release();
```

Parameters

None

Return Values

= 0	Success
= -1	Failure

led_start_blinking()

Start LED blinking.

Prototype

```
enum result * led_start_blinking(int leds, unsigned int duration_on,
                                  unsigned int duration_off,unsigned int
timeout);
```

Parameters

`leds`	The ioctl to one of the drivers (can be the LED driver or the smart card driver).
`duration_on`	The duration (in milliseconds) that the LED is on.
`duration_off`	The duration (in milliseconds) that the LED is off.
`timeout`	if > 0, the LED stops blinking after `timeout` seconds.

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };.
```

led_stop_blinking2()

Stop LED blinking.

Prototype

```
enum result led_stop_blinking2(int leds);
```

Parameters

`leds`

`LED_MSR`	MSR top LED
`LED_MSR1`	MSR middle LED
`LED_MSR2`	MSR bottom LED
`LED_KP`	Keypad LED
`LED_LOGO`	Logo LED
`LED_BOOT_ERROR`	Boot LED
`LED_SC`	Smartcard LED
`CTLS_FIRST`	First CTLS LED
`CTLS_SECOND`	Second CTLS LED
`CTLS_THIRD`	Third CTLS LED
`CTLS_FOURTH`	Fourth CTLS LED
`LED_SYSTEM_GREEN`	System LED
`LED_SYSTEM_RED`	System LED

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };.
```

led_read_led()

Read current LED status (on/off).

Prototype

```
enum result led_read_led(int leds, unsigned char *cmd);
```

Parameters

`leds`		
	`LED_MSR`	MSR top LED
	`LED_MSR1`	MSR middle LED
	`LED_MSR2`	MSR bottom LED
	`LED_KP`	Keypad LED
	`LED_LOGO`	Logo LED
	`LED_BOOT_ERROR`	Boot LED
	`LED_SC`	Smartcard LED
	`CTLS_FIRST`	First CTLS LED
	`CTLS_SECOND`	Second CTLS LED
	`CTLS_THIRD`	Third CTLS LED
	`CTLS_FOURTH`	Fourth CTLS LED
	`LED_SYSTEM_GREEN`	System LED
	`LED_SYSTEM_RED`	System LED
	`Cmd`	LED status (on/off): • 0 = LED off • 1 = LED on

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };.
```

led_read_led_ctls()

Read current CTLS LED status (on/off).

Prototype

```
enum result led_read_led_ctls(int leds, int *read_res);
```

Parameters

`leds`

`LED_MSR`	MSR top LED
`LED_MSR1`	MSR middle LED
`LED_MSR2`	MSR bottom LED
`LED_KP`	Keypad LED
`LED_LOGO`	Logo LED
`LED_BOOT_ERROR`	Boot LED
`LED_SC`	Smartcard LED
`CTLS_FIRST`	First CTLS LED
`CTLS_SECOND`	Second CTLS LED
`CTLS_THIRD`	Third CTLS LED
`CTLS_FOURTH`	Fourth CTLS LED
`LED_SYSTEM_GREEN`	System LED
`LED_SYSTEM_RED`	System LED
`read_res`	LED status (on/off): • 0 = LED off • 1 = LED on

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };.
```

led_switch_color()

Specify the color of a specific LED; terminal-specific.

Prototype

```
enum result led_switch_color(int leds,char color);
```

Parameters

leds

`LED_MSR`	MSR top LED
`LED_MSR1`	MSR middle LED
`LED_MSR2`	MSR bottom LED
`LED_KP`	Keypad LED
`LED_LOGO`	Logo LED
`LED_BOOT_ERROR`	Boot LED
`LED_SC`	Smartcard LED
`CTLS_FIRST`	First CTLS LED
`CTLS_SECOND`	Second CTLS LED
`CTLS_THIRD`	Third CTLS LED
`CTLS_FOURTH`	Fourth CTLS LED
`LED_SYSTEM_GREEN`	System LED
`LED_SYSTEM_RED`	System LED
`color`	Toggle LED color: • 0 = color 1 • 1 = color 2

Return Values

```
enum result  {
  ERROR_INIT=1<<0,
  ERROR_KP=1<<1,
  ERROR_LOGO=1<<2,
  ERROR_BOOT_ERROR=1<<3,
  ERROR_MSR=1<<4,
  ERROR_MSR2=1<<5,
  ERROR_MSR3=1<<6,
  ERROR_SC=1<<7,
  LEDS_SUCCESS=1<<8,
  ERROR_CTLS=1<<9,
  ERROR_SYSTEM_GREEN=1<<10,
  ERROR_SYSTEM_RED=1<<11,
  };
```

CHAPTER 12

CardSlot Library

The V/OS CardSlot Library is a layer above the OS, which communicates with ICCs (Integrated Chip Cards). This library provides a high-level interface independent of the protocol negotiated between the chip card slot and the ICC.

Use the V/OS CardSlot Library to build smartcard applications on V/OS-based terminals. This document details the supported APIs. Figure 8 illustrates the CardSlot architecture.

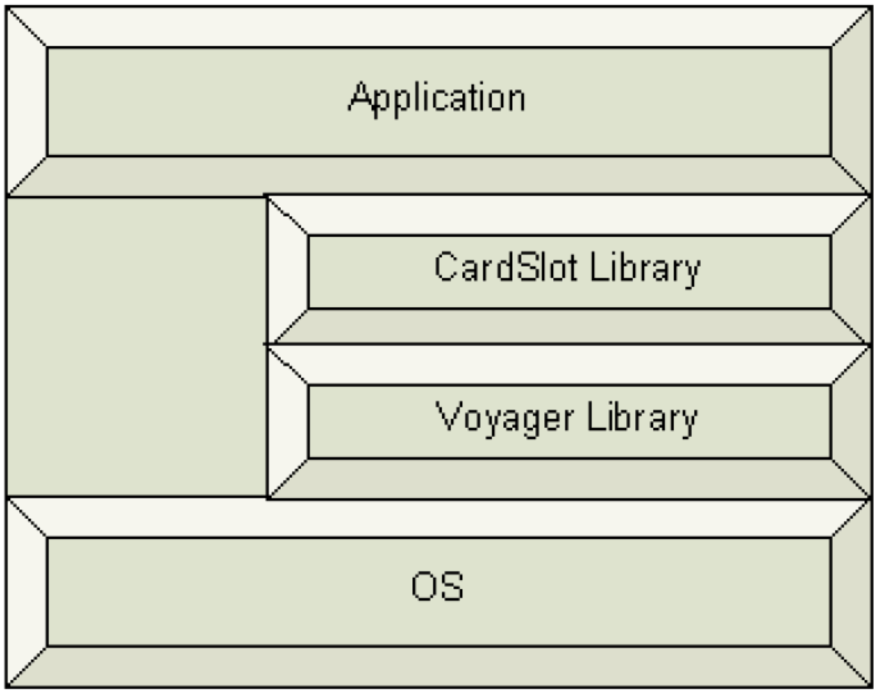


Figure 8 CardSlot Architecture

Refer to the *V/OS CardSlot Library Programmers Guide* (document number DOC00510) for information on the CardSlot API.

CHAPTER 13

Touch Panel, Signature Capture, and TIFF

Some V/OS-based terminals have a touch panel, which can be used for signature captures. The touch panel is calibrated (also called compensation) at the factory. The touch panel requires calibration once placed in service. Once calibrated after installation, no periodic calibration is required. Touch panel calibration is performed using the System Mode menu under Administration > Touch Panel.

Touch Panel

Touch capture technology supports Stylus Priority, which means that if a finger and the stylus are placed on the panel, the position input comes from the stylus and the finger is ignored. Stylus Priority mode is enabled by default.

TIFF Functions

The TIFF function calls allow the application to generate a TIFF file from captured signature data. It is normally called after `SigCapGet()`.

These calls require `libtiff.so` available from Verifone in a publicly available library distribution. The library distribution files are not changed in any way by Verifone; we simply run a special configuration and build to reflect the functionality to extract from the library, facilitating new library releases.

The library has four application header files: `tiffvers.h`, `tiffconf.h`, `tiffio.h`, and `tiff.h`, built when the library gets configured and built. Do not alter these files as any rebuild of the library might invalidate or overwrite them. Applications using `libtiff` use `#include tiffio.h`, which automatically includes these three files.

The main features (such as, CCITTFAX4 compression currently used in Verifone products) are enabled. However, enabling every feature would result in a significantly larger `libtiff` file. That is why JPEG is not supported.

Verifone also supplies library `libcaptouch.so`, which provides a wrapper allowing easy use of the library for typical Verifone applications. The signature capture functionality provided in the library is as follows:

`#include sig.h` that includes `sigtiff.h`,which prototypes the following:

```
typedef struct {
short x;
short y;
} __attribute__((packed)) xy_t;
```

It continues with:

```
typedef struct
{
short left,upper,right,lower;
} SigCapBox_t;
typedef struct
{
long joinPoints: 1;
long trimWidth: 1;
long trimHeight: 1;
long xminus1yminus1 : 1;
long xyminus1 : 1;
long xplus1yminus1 : 1;
long xminus1y : 1;
long xy : 1;
long xplus1y : 1;
long xminus1yplus1 : 1;
long xyplus1 : 1;
long xplus1yplus1 : 1;

long reserved : 20;
} SigCapOptions_t;
```

SigCap2Tiff()

Creates the `fname` file in TIFF format. Do not set `fname` to 0; returns an error.

Prototype

```
int SigCap2Tiff
(
char *fname,xyz_t *sig, int count,short compression_scheme,
xy_t *dpi,SigCapBox_t *box, SigCapOptions_t *options,
void (*setTiffUserTags)(TIFF *)
);
```

Parameters

`fname`	TIFF filename to create from the signature data.
`sig`	Pointer to the user's signature data buffer consisting of points of type `xyz_t`. The z data is currently ignored, but may process (yet to be defined) in future releases. Set `sig` to 0 returns an error.
`count`	Number of signature points in the caller's buffer. A negative value returns on error.
`compression_scheme`	Defaults to `COMPRESSION_CCITTFAX4` if the caller sets the parameter to 0. Otherwise the scheme specified is used. The compression schemes are `#defined` in `tiff.h`. **Note:** The current TIFF engine implementation only supports CCITT Group 4 FAX Compression. Setting `compression_scheme` to any value other than 0 results in erroneous behavior.
`dpi`	Pointer to an `xy_t` structure (as prototyped in `sigtiff.h`) that specifies the desired *x* and *y* TIFF image resolution in dpi. Set to NULL to force maximum resolution.
`box`	Pointer to a signature box specified in QVGA display coordinates Data outside the box is interpreted as `PENUP`. Set to NULL to set box coordinates to the entire screen.

`options`	Pointer to a structure specifying points to join (using Bresenham's algorithm), whether to apply trimming to the width and height of the image, and whether and how the image is thickened. Trimming removes any empty space at the left and right or top and bottom of the image. This results in a smaller image and image file. Thickening plots extra points for each x,y value, and works by imagining a 3 x 3 pixel matrix with the original x,y point in the very middle. 9 bits are provided in the `options` structure to specify which the points actually plotted. See the `SigCapOptions_t` structure definition above for these individual bit names, starting at `xminus1yminus1`, and ending at `xplus1yplus1`. This allows the signature to be 1, 2, or 3 pixels thick along the x and y axis. For example, to make the signature 2 pixels wide along both axes, set the following `options` bits: `xy`, `xyplus1`, `xplus1y`, and `xplus1yplus1`. Equivalent to this is `xminus1yminus1`, `xyminus1`, `xminus1y`, and `xy`. Set the pointer to NULL for the default options, which joins the points, trims along both axis, and does not thicken.
`setTiffUserTags`	Optional pointer to a user function that sets various user tags. Set to 0 if not used.

NOTE

`setTiffUserTags` can override the default date/time tag is automatically inserted into the file.

This optional user-supplied parameter can override standard tags (be careful or the TIFF image may be affected or unusable), and define and specify new user tags in the range `MIN_TIFFTAG_USER` to `MAX_TIFFTAG_USER` (both defined in `sigtiff.h`).

The following code snippet is an example of `setTiffUserTags` calling the structure containing the user-specified tags. in the example, `TIFFTAG_GEO`… user-defined tags have values in the range `MIN_TIFFTAG_USER` to `MAX_TIFFTAG_USER`. Consult the `tiffio.h` header file for a summary of fields in the `TIFFFieldInfo` structure.

CAUTION

The open source TIFF library does not implement complete error checking. Use only TIFF tag values within the defined range. Unpredictable results occur with other tag values.

```
static const TIFFFieldInfo xtiffFieldInfo[] =
{
/* XXX Insert Your tags here */
{ TIFFTAG_GEOPIXELSCALE,-1,-1,TIFF_DOUBLE,FIELD_CUSTOM,TRUE,TRUE,
"GeoPixelScale" },
{ TIFFTAG_GEOTRANSMATRIX,-1,-1,TIFF_DOUBLE,FIELD_CUSTOM,TRUE,TRUE,
"GeoTransformationMatrix" },
{ TIFFTAG_GEOTIEPOINTS,-1,-1,TIFF_DOUBLE,FIELD_CUSTOM,TRUE,TRUE,
"GeoTiePoints" },
{ TIFFTAG_GEOKEYDIRECTORY,-1,-1,TIFF_SHORT,FIELD_CUSTOM,TRUE,TRUE,
"GeoKeyDirectory" },
{ TIFFTAG_GEODOUBLEPARAMS,-1,1,TIFF_DOUBLE,FIELD_CUSTOM,TRUE,TRUE,
"GeoDoubleParams" },
{ TIFFTAG_GEOASCIIPARAMS,-1,-1,TIFF_ASCII,FIELD_CUSTOM,TRUE,FALSE,
"GeoASCIIParams" },
};


static void setTiffUserTags(TIFF *tif)
{
TIFFMergeFieldInfo(tif,xtiffFieldInfo,N(xtiffFieldInfo));
TIFFSetField(tif,TIFFTAG_GEOASCIIPARAMS,"Geo ASCII Params
(Custom)Field");
TIFFSetField(tif,TIFFTAG_DOCUMENTNAME,"Document Name Field");
}
```

This function makes the user-defined tags available to the TIFF library, but only actually uses one tag. The other tag is a standard TIFF tag set to a user-specified value.

It is not necessary to call `TIFFMergeField` or a `TIFFFieldInfo` structure to use standard tags. Consult `tiff.h` for a full list. Avoid using tags assigned by other manufacturers. These tags are safest to use:

```
TIFFTAG_DOCUMENTNAME, TIFFTAG_IMAGEDESCRIPTION, TIFFTAG_MAKE,
TIFFTAG_MODEL, TIFFTAG_PAGENAME, TIFFTAG_SOFTWARE, TIFFTAG_ARTIST,
TIFFTAG_HOSTCOMPUTER, TIFFTAG_TARGETPRINTER, TIFFTAG_COPYRIGHT
```

Return Values

= 0	Success
< 0	Error

CHAPTER 14

Audio and Beeper

V/OS terminals use the Advanced Linux Sound Architecture (ALSA) solution. V/OS terminals support both uncompressed audio (wav) and MP3 format audio.

Currently, V/OS terminals do not support a beeper function. All V/OS terminals units have an independent audio codec that supports stereo output.

Audio Functions

Use the following calls to control audio in V/OS terminals.

soundCtrl()

NOTE This legacy audio function call is supported on V/OS, but there is no support for setting bass and treble.

The Verifone supplied library (`libvfisvc.so`) contains support for controlling the audio volume.

Prototype

```
int soundCtrl(int volume, int bass, int treble);
```

Parameters

`volume`	Valid values: 0–100.	
	0	Sound off.
	100	Maximum volume.
`bass`	Accepted values 50–100.	
`treble`	Accepted values 50–100	

Return Values

= 0	No error.
< 0	Error.

CHAPTER 15

Serial Ports and Protocols

The Linux operating system is divided into two spaces: the kernel space and the user space. The kernel space is for device drivers that must interact closely with the hardware. This interaction includes interrupt handling, precise timing, and hardware interfaces that require bit flipping.

The user space code is easier to debug, maintain, and provides superior protection from bugs. V/OS terminals perform most of its protocol processing in libraries that reside in the user space. Kernel drivers implement the standard open/close/read/write functions needed by the protocol libraries.

The mapping of COMx to ttyAMAx is different on each terminal.

NOTE On V/OS-based terminals the I/O module determines available ports. The Wrenchman co-processor is in select I/O modules. COM1/COM2/COM3/COM4 may not be available on all configurations.

Port assignments are dependent on terminal configuration. All ports are available as general RS-232 ports configured using different baud rates, character sizes, parity, and stop bits through either standard Linux calls. Supported baud rates for all ports are: 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200.

Serial port devices can be opened for control by more than one process at a time. The following `ioctl()` calls prevent processes from using the port, allowing exclusive access to the port.

TIOCEXCL	Put the ttyAMAx into exclusive mode, where no further `open()` operations are permitted. They will fail with EBUSY, except for root.
TIOCNXCL	Disable exclusive mode.

V/OS-based terminal COM ports do not support parity errors and BREAK condition detection. The application must know which control and status lines are supported by each COM port. Control lines not supported by the hardware report the last status set by the application, and unsupported status lines report as asserted.

When a port is open through the `O_NONBLOCK` option, the `read()` command return value for that port can have different results if configured with flow control enabled. When flow control is enabled and no data is available on the port, `read()` returns 0. If flow control is not enabled and no data is available, `read()` returns -1 with errno set to EAGAIN (11 – try again). Both values indicate that no data is available.

COM1

COM1, or device `/dev/ttyAMA0`, supports the RTS, CTS, and DCD hardware lines.

COM2

COM2, or device `/dev/ttyAMA1`, supports the RTS, CTS, DTR, and DCD hardware lines. RTS/CTS flow control is under device driver control, and the RTS line must be controlled by the application using Linux `ioctl()` calls:

```
#include <unistd.h>
#include <termios.h>
int fd;
int status;
ioctl(fd, TIOCMGET, &status);
status &= ~TIOCM_RTS;
ioctl(fd, TIOCMSET, &status);
```

The USB (version 1.1) serial device gadget is only supported over a berg cable. The USB OTG port does not support USB serial devices.

V/OS terminals support the following ioctl() to detect cable status:

```
int handle, connected;
#define port "/dev/ttyGS0"

// Open USB gadget port
handle = open(port,O_RDWR | O_NONBLOCK);

// Check for error
if (handle > -1)
{
   // Read connected state
   connected = ioctl(handle,VFI_GADGET_CONNECTED);

   // If the connected state = -1 then cable was disconnected and
reconnected
       if (connected == -1)
   {
      // Cable was removed and re-connected, close port
      close(handle);

      // Re-open port
```

```
        handle = open(port,O_RDWR | O_NONBLOCK);
        if (handle < 0)
            // Port open failed!
    }
    else if (connected==1)// Good connection, OK to write date to port
    {
        // Write data to port
    }
    else          // No connection
        // Cable is disconnected
}
else
    // Port open failed!
```

See Appendix B for MX terminal function calls.

CHAPTER 16

IPP Module

In V/OS terminals the internal PINpad (IPP) chip is emulated in software. Applications access the IPP through a virtual communication port. V/OS terminal IPP emulation contains most IPP7/IPP8 features, including 3DES master/session, multiple DUKPT, and MAC processing. It does not support Spain, Interac, or Secure Messaging (SM) modes.

IPP Functions

All IPP functions are defined in the header file `svcsec.h`. Applications must link with the `libvfisec.so` library using `-lvfisec`.

ippOpen()

Takes ownership of the IPP and clears the internal IPP FIFO. This function always returns 0.

Prototype

```
int ippOpen(void);
```

Return Values

= 0	Success

ippClose()

Releases ownership of the IPP. All unread data is lost. This function always returns 0.

Return Values

= 0	Success

ippRead()

Transfers data from the IPP FIFO to the application data buffer.

Prototype

```
int ippRead(char *buffer, int size);
```

Parameters

`buffer`	Pointer to the data area.
`size`	Maximum number of bytes to read.

Return Values

> 0	Number of bytes returned in buffer
= 0	No data to read
-EBADF	The task does not own the IPP.

PIN Session Timeout

One requirement for application independence in the Payment Card Industry PED (PIN Entry Device) specification is a default timeout for PIN entry function calls from the application. The OS implements a PIN session timeout of 5 minutes. This value cannot be modified. Applications requiring a shorter timeout can issue the following calls to end the PIN session:

`ippTerminatePinEntry()`	For IPP PIN entry.
`iPS_CancelPIN()`	For VSS PIN entry.

After the OS timeout expires:

For IPP PIN entry the IPP sends an <EOT> (0x04) character

For VSS PIN entry the `iStatus` value returned by the `iPS_GetPINResponse()` function is set to 0x0C. Subsequent calls to `iPS_GetPINResponse()` return a `iStatus` value of 0x01 indicating that the PINpad is idle (that is, not in a PIN session).

ippWrite()

Transfers a single complete IPP packet or a single character from the buffer into the IPP. Incomplete, incorrectly framed packets, overly large, or multiple packets in a single write are rejected. The valid start-of-packet characters are STX and SI. The valid end-of-packet characters are ETX and SO. The only single characters accepted are ACK, NAK, and EOT.

Prototype

```
int ippWrite(char *buffer, int size);
```

Parameters

`buffer`	Pointer to the data area.
`size`	Maximum number of bytes to write.

Return Values

`=size`	The packet was transferred to the IPP.
`-EBADF`	The task does not own the IPP.
`-EACCES`	Too may PIN sessions requested during a short period of time. Try again in a few seconds. See note below.
`-EINVAL`	Buffer is too large to be a valid IPP packet, the buffer pointer is not valid, the single character was not one of [ACK, NACK, EOT], the packet has a bad LRC, or the packet is not framed correctly.

NOTE

PIN encryption is limited to one per 30 seconds on average to deter an exhaustive PIN search.

The algorithm is best explained in terms of tokens in a bucket. An encryption request is only accepted if there is a token in a bucket. A token is placed in the bucket every 33 seconds, with a maximum of 10 tokens allowed in the bucket.

Every time a PIN is entered, a token is removed from the bucket. If there is no token in the bucket, the PIN entry request returns an error:

- IPP PIN entry: `ippWrite` returns `-EACCES`.
- VSS PIN entry: `iPS_RequestPINEntry` returns `E_KM_ACCESS_DENIED`.

Allows an average of one PIN encryption per 30 seconds, but over a long period of time. The intention is that under normal use, PIN entry is not denied.

SetSecurePINDisplayParameters()

On touch screen displays, this parameter registers the PIN entry callback function for the upcoming PIN session. This function must be called before requesting a PIN session either through an IPP packet (Z60, Z63 or Z69) or through the VeriShield Security Script calls (`iPS_RequestPINEntry()`).

The hot spot table is relevant only for terminal models that perform PIN collection through the touch panel. The first parameter is not used and is reserved for future use. To ensure future compatibility, applications must pass a NULL pointer for the first parameter.

Prototype

```
void setSecurePINDisplayParameters(struct touch_hs_s, void *callback);
```

With the callback function, the PIN entry process can control the audio and visual aspects as each key press is detected, and its prototype is:

```
void callback(char value);
```

Value	Action
Lower nibble:	
`0x?1`	A numeric key was pressed. A PIN digit has been added to the internal PIN buffer. Application should display an <echo> character.
`0x?2`	BACKSPACE key was pressed. A PIN digit has been removed from the internal PIN buffer. Application should display a <default> character (for example, space, '-', or '_') in place of the last <echo> character.
`0x?3`	CLEAR key was pressed. All PIN digits have been removed from the internal PIN buffer. Application should replace all <echo> characters with <default> characters.
`0x?4`	PIN entry is not allowed.
`0x?5`	Other key #1 was pressed. PIN entry is canceled as if the CANCEL key was pressed. The application can use this code to define an option key such as a CREDIT button.
`0x?6`	Other key #2 was pressed. PIN entry is canceled as if the CANCEL key was pressed. The application can use this code to define an option key such as a CREDIT button.
`0x?7`	PIN entry is canceled as if the CANCEL key was pressed. The application can use this code to define an option key such as a CREDIT button.
`0x?8`	Pin entry had started. The keypad is in Secure mode, and applications do not receive the actual key, but only receive a value from this callback.

Value	Action
Upper nibble:	
`0x7?`	Play “normal” sound.
`0xF?`	Play “error” sound. This is sent when: • BACKSPACE or CLEAR is pressed when there is no PIN digit in the internal buffer. • A numeric key is pressed when there is already the maximum number of PIN digits in the internal buffer. • ENTER is pressed when there is not the minimum number of PIN digits in the internal buffer.

Example 1: A valid numeric entry detected; the function is called once:

```
callback(0x71);
/* Tell the application to play normal sound and to display <echo>
character */
```

Example 2: Backspace key detected:

```
callback(0x72);
/* Tell the application to play normal sound and that BACKSPACE was
pressed*/
```

Example 3: A key is pressed to clear the line when three inputs are entered:

```
callback(0x73);
/* Tell the application to play normal sound and that CLEAR was pressed*/
```

ippPinEntryStatus()

Returns the PIN entry status, the number of PIN digits currently in the internal PIN buffer and the code of the last non-numeric key pressed.

Prototype

```
int ippPinEntryStatus(int *count, int *lastNonNumericKey);
```

Parameters

`count`	Pointer to an integer that receives the current count of PIN digits in the internal buffer.
`lastNonNumericKey`	Pointer to an integer that receives the code of the last non-numeric key pressed. It receives 0 if no new non-numeric key is pressed since last call.

Return Values

= 1	PIN entry in progress.
= 0	No PIN entry in progress.
< 0	Error.
-EBADF	The task does not own the IPP.

ippTerminatePinEntry()

Ends the PIN session, for example, for a time out.

Prototype

```
int ippTerminatePinEntry(void);
```

Return Values

= 0	Success
< 0	Error
`-EBADF`	The task does not own the IPP

IPP MS and DUKPT Communications Packets

The required packet commands of the IPP for MS (Master Session) or DUKPT operations supported by the V/OS are described in this section.

Advanced Programming in IPP

The differences between the V/OS IPP MS and DUKPT are as follows:

IPP	IPP6	IPP7	VVIPP	IPP8	VVIPP8	V/OS IPP8
Secure Message Mode	No	Yes	No	Yes	No	No
Spain SEMP/4B	Yes	Yes	No	Yes	No	No
Key tagging	Yes	No	No	No	No	No
DUKPT Engines	1	1	1	3	3	3

VVIPP supports IPP7 GISKE 3DES key features with one enhancement: All 10 master keys can be triple-length. IPP7 is limited to at most three triple-length keys.

Minor Differences by Packet

This section describes the differences between the IPP7 used in legacy products and the V/OS IPP modules.

<SI>0103<SO> PROM Checksum

The value of the checksum does not match IPP7 because the V/OS IPP does not use the same code.

<SI>0108<SO> IPP ROM Version Number

The return packet is

```
<SI>14IPP8 EMULvvv mm/yy<SO>{LRC}
```

where, vvv is the version number, `mm` is the release month, and yy is the release year.

<SI>13n<SO> Select Baud Rate

Since there is no IPP UART, setting the baud rate does not affect anything. However, the baud rate is stored in non-volatile memory so that it can return in diagnostics packets.

In platforms with an IPP chip, the application must determine the baud rate of the IPP by sending a test packet at all possible baud rates until the IPP responds with an ACK. In V/OS terminals, there is no UART so baud rate mismatch is not possible. Applications that try all possible baud rates receive an ACK on the first test packet thus, speeding up applications slightly.

<SI>15SPAIN<SO> Set IPP6 Key Management Mode

Spain mode is not supported and switching to Spain mode erases keys. This is done because some programs depend on this feature to erase keys.

<SI>17xyz<SO> Set IPP7 Key Management Mode

SM mode is not supported but switching to SM mode erases keys. This is done because some programs depend on this feature to erase keys.

<SI>02…<SO> Set Master Key

IPP7 can hold a maximum of three (3) triple-length keys. In VVIPP, all ten key locations can hold a single-, double-, or triple-length key.

<STX>75..<ETX> DUKPT Accept and Encrypt PIN/Data Authentication Response

ANSI DUKPT MAC is only defined for 3DES DUKPT. VVIPP returns error code 8 if ANSI DUKPT MAC is requested when using 1DES DUKPT. IPP7 returns undefined results in this case.

Packets

The packet set is similar to that used for external PIN pads, such as the PINpad1000, however, unlike previous IPPs, the V/OS terminals IPP is a software module running on the main CPU. Previous IPPs used dedicated microcontrollers connected to the main CPU through a serial port. In V/OS terminals IPP the serial port is emulated in software along with all IPP functionality.

The IPP command and response packets can be divided into the following categories:

- Common packets: Packets used in both MS and DUKPT.
- MS-specific packets: Packets used while doing MS.
- DUKPT-specific packets: Packets used while doing DUKPT.
- MAC-specific packets: MAC generation of received message packets.

NOTE

The V/OS terminals IPP do not support Spain SEMP/4B mode or Secure Messaging (SM) mode.

The IPP supports both MS and DUKPT key management modes concurrently. Also, the IPP supports MAC processing while doing MS or DUKPT.

The following table lists packets used in both MS and DUKPT sessions.

Packet	Description
01	Interactive diagnostic routine
05	Transfer serial number
06	Request PIN pad serial number
09	Response to Packet 01
11	PIN pad connection test
12	Dummy packet
13	Select baud rate
14	Response to Packet 01

Packet	Description
15	Set IPP key management mode
17	Set IPP7 key management mode
18	Check IPP7 key management mode
M04	Read Permanent Unit Serial Number (IPP8 Emulation)

The following table lists packets supported by IPP for MS.

Packet	Description
02	Load/set master key
04	Check master key
07	'Dummy' DES reliability test
08	Select master key
Z60	Accept and encrypt PIN (VISA mode)
Z63	Accept and encrypt PIN, custom PIN entry requirements (VISA mode)
71	Response PIN block
Z66	MAC processing
Z67	Return MAC
72	Cancel MAC session

The following table lists packets supported by IPP for DUKPT.

Packet	Description
90	Load initial key
91	Confirm initial key
75	Encrypt PIN/authentication data response
78	Encrypt PIN/authentication data test request
76	PIN entry test request
71	Response PIN entry test request of “76”
Z60	Accept and encrypt PIN request (VISA mode)
Z63	Accept and encrypt PIN, custom PIN entry requirements (VISA mode)
Z69	Accept and encrypt PIN/data authentication request (VISA mode)
73	Response PIN block
19	Select a DUKPT Engine (IPP8 Emulation)
25	Check the DUKPT Engine (IPP8 Emulation)

Packet Acknowledgment and Timing

The IPP only responds to commands that have the proper packet format. The packet can be in the form of `<STX>msg<ETX>[LRC]`or `<SI>msg<SO>[LRC]` according to the specific command.

The IPP returns <`ACK`> within 20ms to the terminal when it receives a properly framed packet with a valid LRC. When other framing is received for a command that requires <`STX`><`ETX`> framing (for example, <`SI`><`SO`>, <`SI`><`ETX`>, or <`STX`><`SO`>), <`ACK`> is returned if the LRC is valid; only the specified framing is processed.

This rule also applies to <`SI`><`SO`> packet commands. The IPP does not act on an incorrectly formatted packet. This includes a packet with a wrong header, wrong trailer, wrong field separator, an out of range indexing, or incorrect packet length. An example of a packet that has an out of range indexing would be `packet 02, master key address = 15`.

The response message from the IPP follows the <`ACK`> if the packet command has a response. However, the timing varies from different commands.

Encryption

There are two methods of PIN encryption in IPP:

- MS
- DUKPT

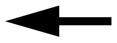
MS Method

IPP encrypts the customer's PIN according to the ANSI X9.8 standard and the ANSI X9.24 master key management method, based on the ANSI X3.92 DES algorithm implemented in the IPP firmware. The encryption during a transaction is as follows:

1. The master device sends a private communication key (or *working* key) to the IPP, where it is decrypted using the currently selected master key. An account number and PIN are also entered to IPP through the master device.
2. The IPP generates the clear text PIN block using the account number and PIN.
3. Using the decrypted working key, the IPP encrypts the PIN block using the DES algorithm and working key, then sends the encrypted PIN block to the master device.
4. The master device appends the encrypted PIN block to a request packet and forwards the completed request packet to the host.

The following illustrates an MS encryption session.

Master Device		IPP
1 Forwards the encrypted working key, account number, and PIN to the IPP.	→	



1 Decrypt the working key using the master key.
2 Encrypts the PIN block with the decrypted working key.
3 Sends the PIN block to the master device.

4 Appends the PIN block to the request packet.
5 Forwards the packet to the host.

DUKPT Method

The IPP encrypts the customer’s PIN according to the ANSI X9.8 standard and Visa's ANSI X9.24 DUKPT key management method, based on the ANSI X3.92 DES algorithm implemented in the IPP firmware.

Before actual operation, each IPP must be loaded with a unique initial KSN (Key Serial Number) and a unique initial PEK (PIN Encryption Key). And the encryption counter of the IPP is set to zero. The initial PEK is generated by encrypting the initial KSN using appropriate derivation key.

The encryption per transaction of IPP during actual operation is as follows:

1 The master device sends an account number and a PIN to the IPP.

2 The IPP generates the clear-text PIN block using the account number and PIN.

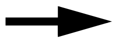
3 Using the generated PEK based on the encryption counter which is updated after each transaction, the IPP do a special encrypt to the PIN block using the DES algorithm and PEK, then sends the encrypted PIN block with current KSN (the concatenation of the initial KSN and the encryption counter) to the master device.

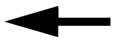
4 The master device then appends the encrypted PIN block and current KSN to a request packet and forwards the completed request packet to the host.

The following illustrates the DUKPT method of encryption.

Master Device IPP

1 Forwards the account number and PIN to the IPP.





2 Creates the PIN block.
3 Encrypts PIN block with the generated PEK.
4 Sends the PIN block and current KSN (key serial number) to the master device.

5. Appends the PIN block and KSN to the request packet.
6. Forwards the packet to the host.

Constraints

The known software constraints for IPP are:

- All communication must be asynchronous, half-duplex, 1200/2400/4800/9600/19200 baud, 7 data bits, even parity, and 1 stop bit (7E1).
- Packet length is limited to 255 characters.

NAKs

When the IPP receives NAK, it retransmits the last message and increments a NAK counter for that communication session. If more than three NAKs are received during any attempt to transmit the same item, the transmitting party send an EOT, terminating the session.

Time Outs

During a communication session, the IPP or the terminal times out if it does not receive the expected communication within 15 seconds. The unit sends an EOT to terminate the communication session.

Key Insertion

This section describes MK insertion and DUKPT initial PIN encryption key insertion.

Master Key Insertion

For each master key injection session, the IPP checks to see if it is the first time that user tried to load the master key. If it is the first time, the IPP clears all master keys to zero before loading a new master key.

NOTE All master keys must be loaded in the same key injection session, otherwise the previous master key is erased in the next master key injection session. A master key injection session is the duration of the power level is maintained in the IPP. The master key insertion rule does not apply to the GISKE key loading key (KLK).

The terminal or master device uses Packet 02: Transfer Master Key to transfer the master keys into the IPP for MS.

DUKPT Initial PIN Encryption Key Insertion

The terminal or master device uses DUKPT Packet 90: Load Initial Key Request to load the initial PIN encryption key into the IPP for DUKPT.

Entering a PIN

Packets Z60, Z63, and Z69 are used to get and encrypt a PIN from the user. Z63 is similar to Z60, but allows more options for PIN entry, such as minimum and maximum PIN length and echo character. Z69 is similar to Z60, but does DUKPT MAC processing as well as PIN encryption using the same DUKPT key.

Restrict the Speed of the PIN Encryption Operation

PIN encryption is limited to one per 30 seconds on average to deter an exhaustive PIN search. The algorithm is best explained in terms of tokens in a bucket.

A PIN encryption request is only accepted if there is a token in a bucket. A token is placed in the bucket every 33 seconds, with a maximum of 10 tokens allowed in the bucket. (The number of tokens in the bucket is limited to 2 on power up.) Every time a PIN is entered, a token is removed from the bucket. If there is no token in the bucket, the PIN entry request returns an error.

This allows an average of one PIN encryption per 30 seconds, but over a long period of time. The intention is that under normal use PIN entry is not denied.

IPP7

This section discusses IPP7-specific features for the V/OS terminal IPP. IPP7 is backward compatible with IPP6 and IPP5. Exceptions to this rule are noted.

GISKE

GISKE (Global Interoperable Secure Key Exchange) is an industry standard key block format for secure transfer of secret keys between two devices that share a secret key. Both master and session keys can be in GISKE format. The GISKE KLK (Key Loading Key) is used to encrypt and authenticate master keys. Master keys can be remotely updated using this key. GISKE is designed for secure transfer of double- and triple-length 3DES keys. For details on GISKE, refer GISKE Key Block Spec, P/N 22986.

Key Management Switching

The rules for key management switching see Packet 17: Set IPP7 Key Management Mode.

Key

- NC = no change
- E = all keys erased
- 1K = valid 1DES keys (single-length keys) retained, other keys erased
- 2/3K = valid 3DES keys (double- and triple-length keys) retained, other keys erased

The following table lists the Key Management Switching Rules.

Rules	To 1DES (VISA)	To 1DES (SPAIN)[a]	To Mixed Mode	To 3DES	To SM[a]
From 1DES[b] (VISA)	NC	E	NC	2/3K	E
From 1DES[a] (SPAIN)	E	NC	E	E	E
From Mixed mode[c]	1K	E	NC	2/3K	E
From 3DES[d]	E	E	E	NC	E
From SM[a]	E	E	E	E	NC

Key Mode	1DES and 3DES Key Usage Rules[e]
1DES only[b]	• Load and use of 1DES MS keys allowed [f] • Load KLK allowed • Load 3DES master keys allowed • Use of 3DES master keys not allowed • Load 3DES session keys not allowed • Use of 3DES session keys not allowed • Key attributes verified[g], except key usage = ‘AN’ – ANY is allowed • GISKE key block verified[h]
Mixed mode[c]	• Load and use 1DES or 3DES MS keys allowed • Load KLK allowed • 1DES master keys used for 1DES session keys • 3DES master keys used for 1DES and 3DES keys • Key attributes verified, except: key usage = ‘AN’ – ANY is allowed • GISKE key block verified
3DES only[d]	• Load and use 3DES MS keys allowed • Load KLK allowed • Load 1DES master keys not allowed • Use of 1DES master keys not allowed • Load 1DES session keys not allowed • Use of 1DES session keys not allowed • Key attributes verified; no exceptions allowed • GISKE key block verified

a. Spain and SM modes not supported in the V/OS IPP. Keys are erased as specified.
b. Least secure mode.
c. For transition period.
d. Most secure mode.
e. The key management register is set using Packet 17: Set IPP7 Key Management Mode.
f. All DUKPT related keys, counters, and registers are erased when the IPP KM switches between 1DES DUKPT and 3DES DUKPT. Other MS related information remains untouched.
g. Key attributes verified means that when a key stored in the IPP is used, the IPP must validate the content of all key attributes. The attributes of the key are validated against the GISKE specification acceptable for that command.
h. GISKE key block verified means that when receiving a key block, the IPP must validate both the key block binding method of the key block and the content of the header. The header of the key is validated against a list of headers acceptable for that command.

Using a Session Key

This sections describes session key loading, master keys for PIN encryption, rules for downloading master keys, the GISKE KLK keys, 3DES keys, and 1DES keys.

Loading the Session Key

3DES session keys are only loaded in GISKE cipher text under the protection of the indexed master key, as long as that key has its attribute set to ‘KEK’ (key usage attributes = “K0”). The master key must be 3DES. The version of the incoming key is not checked or saved. The usage attribute of the incoming working key is checked, but is not saved.

The GISKE key length decryption rule is applied. The length of the master key must be greater or equal to the length of the working key.

1DES session keys in key-only format are loaded in cipher text under the protection of the indexed master key, if that key has its attribute set to ‘ANY’ or ‘KEK’ (key usage attributes = “K0”). The master key can be a single-, double-, or triple-length key.

1DES session keys in GISKE format are loaded in cipher text under the protection of the indexed master key, if that key has its attribute set to ‘KEK’ (key usage attributes = “K0”). The version of the incoming key is not checked or saved. The usage attribute of the incoming working key is checked, but not saved. The master key can be a single-, double-, or triple-length key.

Master Key for PIN Encryption

Where the PIN Entry zero session key method for 1DES is used, the current master key must be tagged ANY or PIN ENCRYPTION.

Where the tagged zero GISKE session key method for 3DES is used, the current master key must be tagged for the specified purpose – key usage =

- ‘P0’ - ‘PIN ENCRYPTION’
- Key Algorithm = 'T' -TDES for double or triple-length keys
- ‘D’ - DES for single-length key
- ‘AN’ – ANY

NOTE

Zero GISKE session key for 3DES means all fields are zero in the GISKE key block.

If zero GISKE support is disabled, the zero GISKE session key causes an error response from the IPP. The zero session key support is enabled or disabled through the KM flag. Zero GISKE session key support (PIN entry) is enabled or disabled through the KM flag.

Rules for Loading the Master Key (MS only)

This section provides details on IPP7 key attributes, key version, and key length.

On erasure, the master key usage attribute is set to 0, the version is set to 0, and the length is set to 1DES.

NOTE

Each key has its own key attribute register, key version register, and key length register.

The register listed in the following table applies to 1DES master key, 3DES master key (GISKE), and KLK (GISKE). The original GISKE (ASCII-hex) key usage attribute value is saved in RAM (2 bytes).

Key Attribute Register	Value	Definition
[XX]	AN	ANY: Key is available in IPP, but the Key was not loaded using GISKE format.
	D0	Data encryption
	I0	IV
	T0	Control vector
	K0	Key encryption or wrapping
	G0	MAC generation
	M0	MAC verification
	P0	PIN encryption
	V0	PIN verification
	C0	CVK: card verification key
	B0	BDK: base derivation key [A]
	00	ISO 9797-1, MAC algorithm 1– 56 bits
	10	ISO 9797-1, MAC algorithm 1–112 bits
	20	ISO 9797-1, MAC algorithm 2–112 bits
	30	ISO 9797-1, MAC algorithm 3–112 bits
	40	ISO 9797-1, MAC algorithm 4–112 bits
	50	ISO 9797-1, MAC algorithm 5–56 bits
	60	ISO 9797-1, MAC algorithm 5–112 bits

The key version of an incoming GISKE format key must be greater than or equal to the version set in the key attribute table for all keys (that is1DES master key, 3DES master key GISKE, and KLK GISKE). The rules for the GISKE key version are:

- when the version is greater than or equal to the current key, OK is returned and the IPP updates the new key

- when the version is less than the current key version, an error returns and the IPP rejects the new key

NOTE

The key version comparison is only compared to the key it is replacing, not to any other keys.

The following table lists the key length register values for 1DES, 3DES, and three-key 3DES.

Length	Comments
1DES	Single-length key: Key length register = 00
3DES	Double-length key: Key length register = 01
3-Key 3DES	Triple-length key: Key length register = 10
Reserved	Key length register = 11

KLK

The GISKE KLK is loaded as clear text if the KLK is not present in IPP. The version of the incoming key is not checked. The version of the stored key is the version carried in the message. The stored key attribute is set to the value in the GISKE message, which should be 'K0'.

The GISKE KLK is loaded in cipher text if the stored KLK attribute location is 'K0' and the KLK present flag in the IPP is set. The new GISKE KLK load is protected by the previous GISKE KLK. The current and new KLK key must be a double- or triple-length key. The version of the key is checked against the stored version. The version of the stored key is the version carried in the message. The stored key usage attribute is set to that carried in the GISKE message, which should be 'K0'.

The rules for the KLK are:

- **KLK is present** and clear text is being loaded, the IPP returns an error.
- **KLK is not present** and clear text is being loaded, OK is returned and the IPP stores the first KLK.
- **KLK is present** and cipher text is being loaded that is not encrypted with the previous KLK, the IPP returns an error.
- **KLK is not present** and cipher text is being loaded that is not encrypted with the previous KLK, the IPP returns an error.
- **KLK is present** and cipher text is being loaded that is encrypted with the previous KLK but has an incorrect key version, the IPP returns an error.
- **KLK is not present** and cipher text is being loaded that is encrypted with the previous KLK but has an incorrect key version, the IPP returns an error.
- **KLK is present** and cipher text is being loaded that is encrypted with the previous KLK, has the correct key version and the key attribute is not equal to “KEK”, the IPP returns an error.

- **KLK is present** and cipher text is being loaded that is encrypted with the previous KLK, has the correct key version and the key attribute is equal to “KEK”, the IPP stores the KLK and its attributes.
- **KLK is not present** and cipher text is being loaded that is encrypted with the previous KLK, has the correct key version, the key attribute KEK value has no effect, the IPP returns an error.

3DES

All 3DES key loads are in GISKE format. 3DES master keys are loaded in clear text without cryptographic protection if the KLK present flag is clear in the IPP. The MAC value is all zero bytes. The version of the incoming key is checked against the stored version. The version of the stored key is the version carried in the GISKE message. The stored key attribute is set to that in the GISKE message.

3DES master keys load in cipher text under the protection of the KLK if the KLK present flag is set. The KLK must be 3DES. The version of the key is checked against the stored version. The version of the stored key is the version carried in the GISKE message. The stored key usage attribute is set to that in the GISKE message.

The rules for 3DES are:

- **KLK is present** (the current key attribute register in the IPP is GISKE format) and clear text 3DES master key is being loaded, the IPP returns error
- **KLK is not present** (the IPP KLK present flag is clear) and clear text 3DES master key is being loaded, the IPP stores the 3DES key
- **KLK is present** (the current key attribute register in the IPP is GISKE format) and cipher text 3DES master key is being loaded with an incorrect key version, the IPP returns an error
- **KLK is present** (the current key attribute register in the IPP is GISKE format) and cipher text 3DES master key is being loaded with the correct key version, the IPP decrypts and stores the 3DES key master key attribute equal to the GISKE format length and equal to 3DES
- **KLK is not present** (the IPP KLK present flag is clear) and cipher text 3DES master key is being loaded, the IPP returns an error

1DES

The 1DES master keys loaded in the short-form method (that is, IPP6 key-only format) have the 'ANY' and 1DES attributes set. The 1DES master keys in GISKE format are be loaded in GISKE clear text without cryptographic protection, if the KLK present flag is clear in the IPP. The MAC value is all zero bytes. The version of the incoming key is checked. The version of the stored key is the version carried in the GISKE message. The stored key attribute is set to that carried in the GISKE message.

The 1DES master keys in GISKE format are loaded in cipher text under the protection of the KLK, if the KLK present flag is set. The KLK master key must be 3DES. The version of the key is checked against the stored version. The version of the stored key is the version carried in the GISKE message. The stored key attribute is set to that carried in the GISKE message.

Master Key Addressing

In V/OS terminals all master key locations 0–9 can hold single-, double-, or triple-length DES keys. The V/OS IPP can hold a maximum of three triple-length keys.

Clear Text GISKE Key Block Loading Rule

The following are Verifone-proprietary rules for GISKE key block loading, and are not part of the ANSI GISKE specification.

- If the KLK is not loaded, the GISKE key block is loaded in clear text.
- The clear-text GISKE key bock must be padded to a length of 120 bytes, as shown in the following examples.

Key

HB	indicates the header block
KB	indicates the key block
eHB	indicates the encrypted header block
eKB	indicates the encrypted key block.

GISKE key block:

8 HB + 24 HB + 24 KB + 8 MAC

Cipher text GISKE key block for transmit (encrypted with KLK or KEK):

8 HB + 48 eHB + 48 eKB + 16 MAC

Cleartext GISKE key block (MAC is all zeros):

8 HB + 24 HB + 48 KB + 16 MAC

To pad the clear text GISKE key block to a total length of 120 bytes and be consistent with its counterpart (that is, the cipher text GISKE key block), 24 HB is expanded to 48 HB. The high and low nibbles of ASCII are converted to an individual hex value. For example:

	D	**0**	**A**	**...**
	0x44	0x30	0x41	(ASCII)
expanded HB =	0x34 0x34	0x33 0x30	0x34 0x31	(hex)

Padded clear text GISKE key block (MAC is all zeros):

8 HB + 48 HB + 48 KB + 16 MAC

Common Packets

This section presents the packets common to all protocols.

Packet 01: Interactive Diagnostic Routine

Packet 01 has the IPP run a specified self-diagnostic test. Information on the test in progress is provided using response packets 9 and 14, depending on the selected test. The following table lists the Packet 01 Format.

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 01
Diagnostic # [dd]	2N	2-byte ASCII code of the diagnostic test to run.
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error check

Packet 01 Length

- MAX: 7 characters
- MIN: 7 characters

Packet 01 Example

Send the IPP the request to run diagnostic test 1, RAM test/one time:

```
<SI>0101<SO>{LRC}
```

Packet 05: Transfer Serial Number

The master device uses this packet to transfer a serial number to the IPP. The following table lists the Packet 05 format.

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 05
[vvv]	3AN	PINpad version number
[dddddd]	6N	Release date -- format: YYMMDD
[p]	1AN	Production facility code
[bb]	2AN	Production batch code
[nnnn]	4N	Serial # for group ID 0001–9999
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 05 Length

- MAX: 21 characters

- MIN: 21 characters

Packet 05 Example

Set the IPP serial number to 246880401A001234:

```
<SI>05246880401A001234<SO>{LRC}
```

Packet 05 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>05[vvv][dddddd][p][bb][nnnn]<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect
	←	• EOT after 3 NAKs `<SI>05[vvv][dddddd][p][bb][nnnn]<SO>{LRC}`
• ACK if LRC • NAK if LRC incorrect; the IPP saves serial number • EOT after 3 NAKs	→	
	←	EOT

Packet 06: Request PIN Pad Serial Number

The master device uses this packet to request the serial number from the IPP. If no serial number stored in the IPP, 16 bytes of ASCII zeros will be returned to the master device. The following table lists the Packet 06 format.

Data Element	Characteristic	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	2AN	Value: 06
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 06 Length:

- MAX: 5 characters
- MIN: 5 characters

Request Sample Packet

```
<SI>06<SO>{LRC}
```

The following table lists the Packet 06 format.

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 06
[vvv]	3AN	PIN Pad version number
[dddddd]	6N	Release date, format: YYMMDD
[p]	1AN	Production facility code
[bb]	2AN	Production batch code
[nnnn]	4N	Serial # for Group ID 0001 - 9999
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 06 Length

- MAX: 21 characters
- MIN: 21 characters

Response Sample Packet

```
<SI>06246880401A001234SO>{LRC}
```

Packet 06 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>06<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs `<SI>06[vvv][dddddd][p][bb][nnnn]<SO>{LRC}`
• ACK if LRC • NAK if LRC incorrect, the IPP saves serial number • EOT after 3 NAKs	→	
	←	EOT

Packets 09 and 14: Response Packet to Packet 01

In response to packet 01, the IPP returns packets 09 and 14 to the master device:

- Packet 09 is the response packet to packet 01 with diagnostic # 07 (UART Loopback Test).
- Packet 14 is the response packet to the packet 01 with diagnostics #00, 01, 02, 03, 06, 08, 09, and 10.

Packets 09 and 14 are in the format shown in the following table.

Master Device	Transmit Direction	IPP
00 Current Baud Rate		
`<SI>0100<SO>{LRC}`	→	
	←	ACK/NAK/EOT
	←	`<SI>14yyyyy<SO>{LRC}` where, *yyyyy* indicates the current baud rate: • 1200 • 2400 • 4800 • 9600 • 19200
ACK/NAK/EOT	→	
	←	EOT to terminate process.
01 RAM Test/One-Time		
`<SI>0101<SO>{LRC}`	→	
	←	ACK/NAK/EOT
		`<SI>14RAM TST BEGIN<SO>{LRC}`
ACK/NAK/EOT	→	

	←
	←
		`<SI>14RAM TST OK<SO>{LRC} or`
		`<SI>14BAD RAM<SO>{LRC}`

ACK/NAK/EOT	→	
	←	EOT to terminate process.

Master Device	Transmit Direction	IPP
02 RAM Test/Continuous		
`<SI>0102<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>14RAM TST BEGIN<SO>{LRC}`
ACK/NAK/EOT	→	
ACK	→	

	←	`<SI>14RAM TST OK<SO>{LRC} or`
		`<SI>14BAD RAM<SO>{LRC}`

ACK/NAK/EOT	→	
	←	EOT to terminate process.
03 PROM Checksum Test		
`<SI>0103<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>14xx<SO>{LRC}` where, *xx* is the one-byte PROM internal checksum. There are two checksums inside the IPP: • The PROM checksum, which is 2-bytes long and is located at 7FFE/7FFF. This checksum is for manufacturing purposes. • The PROM internal checksum.
ACK/NAK/EOT	→	
	←	EOT to terminate process.

Master Device	Transmit Direction	IPP
06 Serial Number Check		
`<SI>0106<SO>{LRC}`	→	IPP6 and earlier
	←	ACK/NAK/EOT
		`<SI>14xxxxxxxxxxxxxxxx<SO>{LRC}` where, *xxxxxxxxxxxxxxxx* indicates the serial number of the IPP. Length is 16 digits, for example, 1234567890123456.
ACK/NAK/EOT	→	
	←	EOT to terminate process.
07 UART Loopback Test		
`<SI>0107<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>09<SO>{LRC}`
ACK/NAK/EOT	→	
`<SI>09<SUB>PROCESS-ING<SO>{LRC}`	→	
	←	ACK/NAK/EOT
	←	<SI>09<SUB>PROCESSING<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process.

Master Device	Transmit Direction	IPP
08 IPP PROM Version Number		
`<SI>0108<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>14IPPx vvvvxxx MM/YY<SO>{LRC}` where: • *vvvv*: 4-digit software ID number. For IPP5, 0PGP. • *xxx*: 3-digit software version number. For example, xxx = 011 indicates the software version number is 1.1; if 11A (11B, 12D, 21A, and so on), the software is not ECO released and is for test and qualification purposes only. For formal ECO released versions, xxx is all numbers. • *MM/YY*: date of software. For example, MM/YY = 05/95 means the software was created May 1995.
ACK/NAK/EOT	→	
	←	EOT to terminate process.
09 Reset IPP		
`<SI>0109<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>14RESET COMPLETE<SO>{LRC}`
ACK/NAK/EOT	→	
	←	EOT to terminate process. (The IPP restarts. Insert a delay before sending data to the IPP.)

Master Device	Transmit Direction	IPP
10 Clear IPP		
`<SI>0110<SO>{LRC}`	→	IPP7
	←	ACK/NAK/EOT
		`<SI>14CLR COMPLETE<SO>{LRC}`
ACK/NAK/EOT	→	
	←	EOT to terminate process.

Packet 11: PIN Pad Connection Test

The master device uses this packet to check the connection with the IPP. If the connection is good, the master device receives an ‘ACK’ from the IPP within one second. Else, it assumes that the IPP is not connected. The following table lists the Packet 11 Format.

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 11
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 11 Length

- MAX: 5 characters
- MIN: 5 characters

Sample Packet

```
<SI>11<SO>{LRC}
```

Packet 11 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>11<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs

Packets 7 and 12: Dummy Packets

Packets 7 and 12 are dummy packets. When the IPP receives these packets it sends out only <ACK> within one second.

Packet 13: Select Baud Rate

V/OS terminals support this packet but it has no effect. V/OS terminals do not use an RS-232 interface so do not need this setting. However, it is supported for compatibility with other IPPs.

This packet command selects the baud rate for the RS-232 communication interface. Through packet command 01 diagnostic 00, the current baud rate can be determined. The factory default is 1200 bps.

The baud rate setting is stored in backup RAM. After a power cycling memory test or loss of backup battery power, the baud rate setting is reset to the default.

Packet 13 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 13
Packet Data	1N	Baud rate codes: 1–5 • 1 = 1200 bps (default) • 2 = 2400 bps • 3 = 4800 bps • 4 = 9600 bps • 5 = 19200 bps
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 13 Length

- MAX: 6 characters
- MIN: 6 characters

Packet 13 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>13x<SO>{LRC}`	→	
	←	ACK if LRC okay; NAK if LRC incorrect.
		`<SI>14yyyyy<SO>{LRC}` where,

x = baud rate code	*yyyyy* = string for selected baud rate
• 1	• 1200 (default)
• 2	• 2400
• 3	• 4800
• 4	• 9600
• 5	• 19200

Master Device	Transmit Direction	IPP
		The baud rate code must be in the range 1–5; all other codes are ignored and directly echo [EOT] with the baud rate unchanged.
ACK/NAK	→	
	←	EOT to terminate process (the PIN pad uses the new baud rate accordingly).

Packet 15: Set IPP Key Management Mode

This packet command changes the secret key management mode that the IPP uses for the transaction. The IPP supports two modes of secret key management:

- IPP5 or VISA MASTER SESSION+DUKPT mode

VISA MASTER SESSION+DUKPT mode covers MS and DUKPT and MAC process of standard ANSI MAC. The V/OS terminals IPP do not include SEMP/4B mode, and erases keys when this mode is selected.

NOTE

In V/OS terminals IPP, switching to SEMP/4B mode clears all IPP memory but leaves the IPP in VISA M/S+DUKPT mode.

Request Packet Format

```
<SI>15[Key Code][<SO>{LRC}
```

IPP Request Packet 15 Format

Data Element	Field	Length	Comments
<SI>	Start of packet	1	Shift in. Control character- 0Fh
15	Packet type	2	Set IPP key management mode
[Key Code]	Packet parameters	4 or 5	The key management operation mode for the IPP • “SPAIN” – Spain SEMP/4B mode • “VISA” – IPP mode • Other characters – no change
[SO]	End of packet	1	Shift out. Control character- 0Eh
{LRC}	Block code check	1	Error check character

Response Packet Format

```
<SI>15[Key Code]<SO>{LRC}
```

IPP Response Packet 15 Format

Data Element	Field	Length	Comments
<SI>	Start of packet	1	Shift in. Control character- 0Fh
15	Packet type	2	Set IPP key management mode
[Key Code]	Packet parameters	4 or 5	The key management operation mode for the IPP: • “SPAIN” – Spain SEMP/4B mode • “VISA” – IPP mode • Other characters – no change
[SO]	End of packet	1	Shift out. Control character- 0Eh
{LRC}	Block code check	1	Error check character

If the terminal receives the response without any errors, then it sends ACK to the IPP. The IPP then sends <EOT> `{ ASCII CODE is 04 }` to terminate the session.

Packet 15 Example

```
<SI>15SPAIN<SO>{LRC}( set Spain SEMP/4B mode)
<SI>15VISA<SO>{LRC}( set MS / DUKPT mode)
```

NOTE

In IPP6, the following packet 15 variation is included for compatibility purposes only, and does not result in the key information being erased.

Packet 15 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 15
Packet Data	4AN	Value: 'IPP2', fixed as password
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

Packet 15 Length

- MAX: 9 characters
- MIN: 9 characters

Packet 15 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>15IPP2<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect
		`<SI>15IPP2<SO>{LRC}`
ACK/NAK	→	
	←	EOT to terminate process.

Packet 17: Set IPP7 Key Management Mode

This packet sets or clears a number of control switches in the key management options register for IPP7 key management configuration. IPP7 supports the following additional functions (as compared to IPP6):

- Triple DES (3DES) DUKPT support
- GISKE MS Key support
- Zero (0) key support

Note that the new MAC alternatives apply only when GISKE is active, and are selected by key attributes and not the key management switch.

For compatibility, the default key management mode for IPP7 is set to IPP5 mode (MS- DUKPT or single DES mode). Once a new key management scheme is selected, it is retained during power cycles.

Setting a new mode causes the IPP7 to erase all existing keys or non-volatile security values stored for secure messaging.

Incoming Packet Format

```
<SI>17[KMM][PINER]<SO>{LRC}
```

Packet 17 Format

Data Elements	Characteristics	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	2AN	Value: 17

Data Elements	Characteristics	Comments
Key Management Mode [KMM]	2AH	The two ASCII hex digits are concatenated big-endian, to produce a single control byte. The key management mode register (8 bits) in IPP7 is as follows:

Bit			Description
2	1	0	
0	0	0	1DES MS (default)
0	0	1	Mixed mode (1DES and 3DES GISKE)
0	1	0	3DES GISKE MS
0	1	1	Secure messaging (not supported)

Bit 3	Description
0	1DES DUKPT (default)
1	3DES DUKPT

Bit 4	Description
0	Zero key support off (default)
1	Zero key support on

Bit 5	Description
0	Zero GISKE session key support off (default)
1	Zero GISKE session key support on

Bit 6	Description
0	N/A
1	Clear all MS master keys and KLK

Bit 7	Description
0	MAC empty working key support off (default)
1	MAC empty working key support on

Data Elements	Characteristics	Comments
DUKPT Engine 1/2 Mode Flag [DEMF] **Note:** This field was added for IPP8 emulation.	1AH	The one ASCII-Hex digit is used produce half of a control byte. Bit 0 (DUKPT Engine "1") Description 0 1DES DUKPT - Default 1 3DES DUPKT Bit 1 (DUKPT Engine "2") Description 0 1DES DUKPT - Default 1 3DES DUPKT Bit 2 ~ 3 ---------- Reserved Example Engine= 1 2 DEMF = 0x30 0 1DES 1DES 0x31 1 3DES 1DES 0x32 2 1DES 3DES 0x33 3 3DES 3DES
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

Packet 17 Length

- MAX: 8 characters
- MIN: 8 characters

Packet 17 Set IPP Key Management Mode

Master Device	Transmit Direction	IPP
`<SI>17[KMM][PINER]<SO>{LRC}`	→	
	← ←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
		`<SI>17[KMM][PINER]<SO>{LRC}`
• ACK if LRC and key management echo is OK • NAK if LRC incorrect • EOT after 3 NAKs • EOT if LRC is correct, but key management echo is not OK.	→	
	←	EOT to terminate process. The IPP saves the new key management accordingly.

Notes:

- The default setting of the IPP KM mode is “old single DES mode” (IPP5/6 = all zeros in the KMM register). When defaulting to IPP5/6 mode, the IPP is also set to default to VISA mode (not SPAIN).
- When the IPP receives packet 17 to change KM modes (for example, to 3DES or SM mode). the master device must know the new specification and functions associated with the IPP. If the IPP is not in the “old single DES” mode (IPP5/6), the IPP ignores packet 15 and will not allow itself to be switched to SPAIN mode unless the KMM register is set to IPP5/6 mode.
- SPAIN mode is a submode of the single DES (IPP5/6) KMM register setting. A change from 1DES to 3DES or mixed mode will disable SPAIN mode.
- When zero GISKE session key support is enabled (that is, on), the current master key is used for PIN encryption only if packet Z60 has a zero GISKE (3DES) session key and the current master key has its key attribute set to “PIN Encryption” or “ANY.” A zero GISKE (3DES) session key means that all fields are zero in the GISKE key block.
- The master device must delay for at least 500 msec before send a packet to the IPP when the KMM is switched from IPP7 to SM or from SM to IPP7.
- Switching from SM to IPP7 mode causes a factory reset. The IPP clears the contents of RAM and communication to the IPP is reset to the default, 1200 baud, 7 data bits, even parity, and 1 stop bit (7E1).

- Changing the MAC empty working key support flag erases all keys (that is, the KLK, MS key, and DUKPT key).

Packet 17 Examples

The following examples only illustrate the command packet sent from the master device. Some valid IPP KMM are shown below.

1. 1DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

   ```
   <SI>17000<SO>{LRC}
   ```

2. Mixed MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

   ```
   <SI>17010<SO>{LRC}
   ```

3. 3DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

   ```
   <SI>17020<SO>{LRC}
   ```

4. 1DES MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

   ```
   <SI>17080<SO>{LRC}
   ```

5. Mixed MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

   ```
   <SI>17090<SO>{LRC}
   ```

6. 3DES MS mode, zero key support off, zero GISKE session support off, and 3DES DUKPT mode:

   ```
   <SI>170A0<SO>{LRC}
   ```

7. 1DES MS mode, zero key support on, zero GISKE session support off, and 1DES DUKPT mode:

   ```
   <SI>17100<SO>{LRC}
   ```

8. Mixed MS mode, zero key support on, zero GISKE session support on, and 1DES DUKPT mode:

   ```
   <SI>17310<SO>{LRC}
   ```

9. 3DES MS mode, zero key support off, zero GISKE session key support on, and 1DES DUKPT mode:

   ```
   <SI>17220<SO>{LRC}
   ```

10. 1DES MS mode, zero key support on, zero GISKE session key support off, and 3DES DUKPT mode:

    ```
    <SI>17180<SO>{LRC}
    ```

11. Mixed MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

```
<SI>17390<SO>{LRC}
```

12 3DES MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

```
<SI>172A0<SO>{LRC}
```

The combinations of KMM setting are limited, which means that the mixtures of MS mode, zero key support, zero GISKE session key support, DUKPT mode, and SM mode are not applicable in some cases. If there is a conflict in the KMM setting, the following priority rules apply:

Priority	KMM setting	Notes
1	MS/DUKPT mode vs. SM mode	If bit 1 and bit 0 of the KMM register is set to “ONE,” the IPP switches to SM mode, regardless how the other bits are set.
2	MS mode vs. zero key support	Zero key support is not applicable in 3DES MS mode, due to the key usage rule (that is, single-length key use is not allowed in 1DES MS mode). The IPP stores the setting, but it has no affect on the MS function.
3	MS mode vs. zero GISKE session key support	Zero GISKE session key support is not applicable in 1DES MS mode, due to the key usage rule (triple-length key use is not allowed in 3DES MS mode). The IPP stores the setting, but it has no affect on the MS function.

Packet 18: Check IPP7 Key Management Mode

Checks the setting in the IPP7 key management options register.

Request Packet Format

```
<SI>18<SO>{LRC}
```

Packet 18 Format

Data Elements	Characteristics	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	2AN	Value: 18

Data Elements	Characteristics	Comments
Key Management Mode [KMM]	2AH	The two digits are concatenated big-endian, to produce a single control byte. The key management mode register (8 bits) in IPP7 is as follows:

Bit 2	Bit 1	Bit 0	Description
0	0	0	Old single DE
0	0	1	Mixed mode (1DES and 3DES GISKE).
0	1	0	3DES GISKE MS
0	1	1	Secure messaging (not supported)

Bit 3	Description
0	1DES DUKPT
1	3DES DUKPT

Bit 4	Description
0	Zero key support off
1	Zero key support on.

Bit 5	Description
0	Zero GISKE session key support off
1	Zero GISKE session key support on

Bit 6	Description
0	At least one MS key or KLK key has been loaded.
1	All MS master keys and the KLK are clear (no keys loaded).

Bit 7	Description
0	MAC empty working key support off.
1	MAC empty working key support on.

Data Elements	Characteristics	Comments
DUKPT Engine 1/2 Mode Flag [DEMF] **Note:** This field was added for IPP8 emulation.	1AH	The one ASCII-Hex digit is used produce half of a control byte. Bit 0 (DUKPT Engine “1”) Description 0 1DES DUKPT - Default 1 3DES DUPKT Bit 1 (DUKPT Engine “2”) Description 0 1DES DUKPT - Default 1 3DES DUPKT Bit 2 ~ 3 ---------- Reserved Example: Engine= 1 2 DEMF = 0x30 0 1DES 1DES 0x31 1 3DES 1DES 0x32 2 1DES 3DES 0x33 3 3DES 3DES
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

Packet 18 Length

- MAX: 8 characters
- MIN: 8 characters

Packet 18 Check IPP7 Key Management Mode

Master Device	Transmit Direction	IPP
`<SI>18<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
		`<SI>18[KMM][PINER]<SO>{LRC}`
• ACK/NAK	→	
	←	EOT to terminate process.

Packet 18 Examples

The following examples show the response packet, `<SI>18[KMM][PINER]<SO>{LRC}` from the IPP.

1 1DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

```
<SI>18000<SO>{LRC}
```

2 Mixed MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

```
<SI>18010<SO>{LRC}
```

3 3DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

```
<SI>18020<SO>{LRC}
```

4 1DES MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

```
<SI>18080<SO>{LRC}
```

5 Mixed MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

```
<SI>18090<SO>{LRC}
```

6 3DES MS mode, zero key support off, zero GISKE session support off, and 3DES DUKPT mode:

```
<SI>180A0<SO>{LRC}
```

7 1DES MS mode, zero key support on, zero GISKE session support off, and 1DES DUKPT mode:

```
<SI>18100<SO>{LRC}
```

8. Mixed MS mode, zero key support on, zero GISKE session support on, and 1DES DUKPT mode:

   ```
   <SI>18310<SO>{LRC}
   ```

9. 3DES MS mode, zero key support off, zero GISKE session key support on, and 1DES DUKPT mode:

   ```
   <SI>18220<SO>{LRC}
   ```

10. 1DES MS mode, zero key support on, zero GISKE session key support off, and 3DES DUKPT mode:

    ```
    <SI>18180<SO>{LRC}
    ```

11. Mixed MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

    ```
    <SI>18390<SO>{LRC}
    ```

12. 3DES MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

    ```
    <SI>182A0<SO>{LRC}
    ```

13. 1DES MS mode, zero key support on, zero GISKE session key support off, and 3DES DUKPT mode:

    ```
    <SI>18580<SO>{LRC}
    ```

14. Mixed MS mode, zero key support on, zero GISKE session key support on, and 3DES DUKPT mode

    ```
    <SI>18790<SO>{LRC}
    ```

15. 3DES MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

    ```
    <SI>186A0<SO>{LRC}
    ```

Packet Z60: Accept and Encrypt PIN (VISA Mode)

On receipt of the Z60 packet, V/OS terminals read the user's PIN from the keyboard, echoing to the display an asterisk for each digit accepted. The PIN length can be between 4 and 12 digits. There are two variations of the request packet: Master/Session and DUKPT.

Sample Packet Z60 for MS

```
Request   <STX>Z60.[acct num]<FS>[working key]<ETX>{LRC}
Response  <STX>71.0[PIN len][PIN block format]
[encrypted PIN block]<ETX>{LRC}
```

Sample Packet Z60 for DUKPT

```
Request   <STX>Z60.[acct num]<FS>DUKPT ENCRYPTION<ETX>{LRC}
Response  <STX>73.00000[key serial number]
[encrypted PIN block]<ETX>{LRC}
```

On receipt of a packet Z60 that contains the account number and working key (if MS) or DUKPT ENCRYPTION (if DUKPT), the IPP gets the PIN from the user then checks if MS or DUKPT is selected.

- If MS is selected, the IPP encrypts the formatted PIN block using the working key that was decrypted using the selected master key. The IPP returns the cipher-text PIN block using packet 71 (see MS Packet 71: Transfer PIN Block.
- If DUKPT is selected, the IPP encrypts the formatted block using the DUKPT algorithm. The IPP returns the key serial number and cipher-text PIN block using packet 73 (see DUKPT Packet 73: Transfer PIN Block (for Packets Z60 or Z63)).

Packet Z60 Format MS

```
<STX>Z60.[aaa…aaa]<FS>[www…www]<ETX>{LRC}
```

Packet Z60 Format DUKPT

```
<STX>Z60.[aaa…aaa]<FS>[DUKPT ENCRYPTION]<ETX>{LRC}
```

Packet Z60 Format

Data Elements	Characteristics	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	3AN	Value: Z60
Packet Delimiter	1A	Value: (.), 2Eh
[aaa...aa]	8-19N	Card account number
<FS>	1H	Field Separator, Value: 1Ch

Data Elements	Characteristics	Comments
[www...www] or DUKPT ENCRYPTION	16AH or 120AH	[www….www] – encrypted working key (encrypted session key) DUKPT ENCRYPTION means DUKPT is selected. Otherwise, it is the working key of MS encrypted under the master key. GISKE is used here for 3DES session key support. Size of [www…www] indicates which packet format is used: • 16AH – 1DES, key-only format • 120AH – GISKE key block format. For more details on GISKE refer GISKE Key Block Spec, P/N 22986. • (1DES only) If zero key support is enabled and the encrypted working key is zero-filled, the currently selected master key is used as the working key. • (1DES only) If zero key support mode is disabled, the passed key is used regardless of the encrypted key value. • Zero GISKE session key support for GISKE key block format communication protocol. (see Using a Session Key). • Zero key support and zero GISKE session key support are controlled by a switch in the key management option register set using packet 17 and checked using packet 18.
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error Check

Packet Z60 Length

- Maximum: 147 characters
- Minimum: 32 characters

Sample Packet Z60 for MS 1DES

```
<STX>Z60.0123456789012345678<FS>0123456789012345<ETX>{LRC}
```

Sample Packet Z60 for DUKPT:

```
<STX>Z60.0123456789012345678<FS>DUKPT ENCRYPTION<ETX>{LRC}
```

Sample Packet Z60 for MS GISKE:

```
<STX>Z60.0123456789012345678<FS>0123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123
456789<ETX>{LRC}
```

Errors returned by write()

Some packet format errors are caught when the packet is written to the IPP. In this case, `write()` returns –1 and errno set. The packet is not ACKed or NAKed, and no response packet returns.

Z60 Format Error	errno
No <FS>	EINVAL
PIN entry too fast.	EACESS
See Restrict the Speed of the PIN Encryption Operation.	

Packet Z63: Accept and Encrypt PIN–Custom PIN Entry Requirements (VISA Mode)

On receipt of the Z63 packet, the terminal reads the user's PIN from the keyboard, echoing to the display [echo char] for each digit accepted. The PIN length can be between `[min len]` and `[max len]` digits, inclusive, or 0 if the `[NULL allowed flag]` is set. There are two variations of these request packets: MS and DUKPT.

Sample Packet Z63 for MS

```
Request   <STX>Z63.[acct num]<FS>[working key][min len][max len]
[NULL allowed flag][echo char]<ETX>{LRC}
Response     <STX>71.0[PIN len][PIN block format]
                        [encrypted PIN block]<ETX>{LRC}
```

Sample Packet Z63 for DUKPT

```
Request   <STX>Z63.[acct num]<FS>DUKPT ENCRYPTION[min len][max len]
                    [NULL allowed flag][echo char]<ETX>{LRC}
Response  <STX>73.00000[key serial number][encrypted PIN block]<ETX>{LRC}
```

Note that `[min len]` and `[max len]` are two-character ASCII digits that represent values between 04 and 12, inclusive. `[max len]` should not be less than `[min len]` that is:

```
04 ≤ [min len] ≤ [max len] ≤ 12
```

Furthermore, `[NULL allowed flag]` and `[echo char]` each are 1-byte values with the following requirements:

- `[NULL allowed flag]` = Y allows a zero-length PIN entry
- `[NULL allowed flag]` = N does not allow zero-length PIN entries
- `[echo char]` should be displayable and cannot be <STX>, <ETX>, <SI>, <SO>, or <FS>, even if the currently selected font can display characters 02h, 03h, 0Fh, 0Eh, or 1Ch.

If any of these four fields do not conform to the restrictions, then the packet is rejected by the driver (return code of -1 with errno set to `EINVAL`).

Packet Z63 Format

Data Elements	Characteristics	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	3AN	Value: Z63
Packet Delimiter	1A	Value: (.), 2Eh [aaa...aa] 8-19N Card account number
<FS>	1H	Field Separator; Value: 1Ch
[www...www]	16AH or 120AH	Encrypted working key or (encrypted session key) DUKPT. DUKPT ENCRYPTION means DUKPT is ENCRYPTION selected. Otherwise, it is the working key of MS encrypted under the master key. GISKE is used here for 3DES session key support. Size of [www…www] indicates which packet format is used: • 16AH: 1DES, key-only format • 120AH: GISKE key block format. For more details on GISKE refer GISKE Key Block Spec, P/N 22986. • (1DES only) If zero key support is enabled and the encrypted working key is zero-filled, the currently selected master key is used as the working key. • (1DES only) If zero key support mode is disabled, the passed key is used regardless of the encrypted key value. • Zero GISKE session key support for GISKE key block format communication protocol. (see Using a Session Key). • Zero key support and zero GISKE session key support are controlled by a switch in the key management option register set using packet 17 and checked using packet 18.
[min len]	2N	Minimum PIN length. 04–12
[max len]	2N	Maximum PIN length. 04–12
[Null PIN allowed]	1A	Null (zero length) PIN allowed. Y or N.
[echo char]	1AN	Echo character.
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error Check

Errors returned by write()

Some packet format errors are caught when the packet is written to the IPP. In this case, `write()` returns –1 and errno set. The packet is not ACKed or NAKed, and no response packet returns.

Z60 Format Error	errno
No <FS> invalid MIN, MAX, echo character, or null PIN flag	EINVAL
PIN entry too fast.	EACESS
See Restrict the Speed of the PIN Encryption Operation.	

Packet M04: Read Permanent Unit Serial Number

Check the permanent unit serial number.

NOTE

This packet is added for IPP8 emulation.

Request Packet Format

```
<SI>M04<SO>{LRC}
```

Packet M04 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	3AN	Value: M04
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

Packet M04 Length

- Maximum: 6 characters
- Minimum: 6 characters

Response Packet Format

```
<SI>M04[PUSN]<SO>{LRC}
```

Packet M04 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	3AN	Value: M04
Permanent Unit Serial Number [PUSN]	11AN	Unit Serial Number format: xxx-xxx-xxx
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

Packet M04 Length

- Maximum: 17 characters

- Minimum: 17 characters

Packet M04 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>M04<SO>{LRC}`	→	
	←	• ACK if LRC okay. • NAK if LRC incorrect (EOT after 3 NAKs).
	←	<SI>M04[PUSN]<SO>{LRC}
• ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs).	→	
	←	EOT terminates session.

MS-Specific Packets

The following packets are specific to MS 1DES and 3DES operations. The default mode for the IPP at power up is MS 1DES.

Packet 02: Transfer Master Key

The master device uses this packet to send a master key to the IPP. The response from the IPP to the master device depends on the value of the key management option register.

MS Packet 02 Format

Data Element	Characteristic	Comments
<STX>	1H	Shift In, Value: 0Fh
Packet Type	3AN	Value: 02
[n]	1N	Address or key usage identifier. 1DES: • Master key address is 0-9 • 3DES: • Master key address for double- or triple-length keys is 0–9, 'F[a]
[hhh...hh]	16H	Master key in ASCII. • 16Ah: 1DES mode for single-length key • 120Ah: GISKE mode for double- and triple-length key, including key block header, master key, and MAC. For more details on GISKE refer GISKE Key Block Spec, P/N 22986.

Data Element	Characteristic	Comments
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

a. When the GISKE KEK is passed to the IPP in this message, the KEK usage identifier is checked in the GISKE key header block before the key is accepted.

MS Packet 02 Length

- MAX: 126 characters
- MIN: 22 characters

Communication Protocols

Each key stored in the IPP contains its own key attributes.

Key-only Format

The key attribute information is not available when the key is loaded using the key-only format (as compared to the GISKE communication protocol). The IPP sets the default attributes to the key, as shown in the following table.

Key Attributes	Value	Hex	Definition
Key usage	AN	0x41, 0x4E	Any; no special restrictions
Key Algorithm	D	0x44	DES
Key mode of use	N	0x4E	No special restrictions
Key version	00	0x30, 0x30	version = zero
Key length	1	0x31	single-length key

The single-DES communication protocol between the master device and the IPP is shown below.

Sample Packet 02 in Key-only Format

This sample packet requests the IPP to load master key 0123456789ABCDEF into location '0'.

```
<SI>0200123456789ABCDEF<SO>{LRC}
```

Packet 02 Key-Only Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>02[n][hhhhhhhhhhhhh-hhh]<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
	←	`<SI>02[n][hhhhhhhhhhhhhhhh]<SO>{LRC}`

Master Device	Transmit Direction	IPP
• ACK if LRC and key echo are okay • NAK if LRC incorrect • EOT after 3 NAKs • EOT if LRC correct, but key echo incorrect	→	
	←	• IPP saves the new master key only on receipt of ACK • EOT terminates session

GISKE Key Block Format

3DES communication protocol between the master device and the IPP is as follows:

Sample Packet 02 in GISKE Key Block Format:

This sample packet requests that the IPP load the 120-byte GISKE key block into address 0
“`0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF`
`0123456789ABCDEF0123456789ABCDEF012345678901234567890123`:”

```
<SI>
02000123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234
56789ABCDEF0123456789ABCDEF012345678901234567890123<SO>{LRC}
```

Packet 02 Response Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 02
[n]	1N	Response code (0–7): • 0 = No error • 1 = Error: IPP in incorrect KM mode • 2 = Error: incorrect key usage, mode of use, algorithm, or key length • 3 = Version error • 4 = Error: KLK already exists or new KLK was not encrypted from the previous KLK • 5 = GISKE decryption or MAC error • 6 = Error: master key address does not match the address rang e described in Packet 02: Transfer Master Key • 7 = Error: inappropriate master key addressing
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

Packet 02 GISKE Key Block Format Length

- MAX: 102 characters
- MIN: 6 characters

Packet 02 GISKE Key Block Format Example

```
<SI>020<SO>{LRC}
```

Packet 02 GISKE Key Block Format Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>02[r][hhh.hhh]<SO>{LRC}`	→	
	← ←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs `<SI>02[n]<SO>{LRC}`
• ACK if LRC, no errors, and key echo okay • NAK if LRC incorrect • EOT after 3 NAKs • EOT if LRC correct, but key echo incorrect	→	
	←	• IPP saves new key only on receipt of ACK • EOT terminates session

Packet 04: Check Master Key

The master device sends this packet to check if the IPP has a master key stored at a designated master key address. To avoid an overwrite, this packet *must* be sent before sending packet 02 to check that a valid master key is already stored in the designated address.

MS Packet 04 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 04
[a]	1N	Master Key address: 0–9
		KLK: F
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

MS Packet 04 Length

- MAX: 6 characters
- MIN: 6 characters

Packet 04 Communication Protocol

Packet 04 has two types of communication format: key-only and GISKE key block format. The communication format depends on the IPP key management setting and the length of the key at address [a]. The use of the communication protocol is as follows:

IPP Key Management Setting	Key Length at Address [a]	Communication Protocol Used
1DES mode	1DES (single-length key)	Key-only format (IPP5/ IPP6)
	3DES (single-, double-, or triple-length key)	GISKE key block format[a]
Mixed or 3DES mode	1DES (single-length key)	GISKE key block format
	3DES (single-, double-, or triple-length key)	GISKE key block format

a. If a single-, double-, or triple-length key stored in the IPP contains the key attribute information described in the GISKE specification, this indicates that master device must be compatible with the IPP7 (3DES) specification. Therefore, the master device can understand the GISKE key block format communication protocol.

Packet 04 Key-only Format

To check if the master key is loaded at address 5, the request sample packet 04 for key-only format is

```
<SI>045<SO>{LRC}
```

Response Packet 04 Key-only Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 04
[r]	1AN	Response code: • 0 = No master key at address [a] • F = Master key present at address [a]
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

Response Packet 04 Key-only Format Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>040<SO>{LRC}`	→	
	←	• ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs • PINpad checks requested address [a]. `<SI>04[r]<SO>{LRC}`
• ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs	→	
	←	EOT

Response Packet 04 Key-only Format Length

- MAX: 6 characters
- MIN: 6 characters

Response Packet 04 Key-only Format Example

```
<SI>040<SO>{LRC}
```

Packet 04 GISKE Key Block Format

The GISKE key block format communication protocol is used when the IPP is in mixed or 3DES mode. The original GISKE (ASCII-hex) key usage attribute value is saved in RAM (2 bytes).

```
<SI>042<SO>{LRC}
```

Response Packet 04 GISKE Key Block Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 04
[r]	1AN	Response code: • 0 = No master key at address [a] • F = Master key present at address [a]
Key Usage Attribute (KUA)	2AH	Only when master key is present at address [a]: • AN: ANY: The key is available in the IPP, but was not loaded using GISKE format. • D0: Data encryption • I0: IV • T0: control vector • K0: key encryption or wrapping • G0: MAC generation • M0: MAC verification • P0: PIN encryption • V0: PIN verification • C0: CVK (card verification key) • B0: BDK (base derivation key [A]) • 00: ISO 9797-1 MAC algorithm 1 (1–56 bits) • 10: ISO 9797-1 MAC algorithm 1 (1–112 bits) • 20: ISO 9797-1 MAC algorithm 2 (2–112 bits) • 30: ISO 9797-1 MAC algorithm 3 (3–112 bits) • 40: ISO 9797-1 MAC algorithm 4 (4–112 bits) • 50: ISO 9797-1 MAC algorithm 5 (5–56 bits) • 60: ISO 9797-1 MAC algorithm 5 (5–112 bits)

Data Element	Characteristic	Comments
Algorithm	1AH	(optional) Only if the master key is present at address [a]. The value is stored in the Key Attributes register. • D: DES [0] • R: RSA [1] • A: AES [2] • S: DSA [3] • T: TDES [4] • U: Unknown [5] • E: Elliptic Curve [6] • [7]–[F] = Reserved **Note:** To save storage space in RAM, the algorithm attribute is converted to [x], a hex number ranging form 0–F (4 bits). In the response packet (to packet 04), the IPP converts the number back to characters used in GISKE specification.
Mode of Use Attribute (MOUA)	1AH	(optional) Only if the master key present at address [a]. The value is stored in the key attributes register. • N: No special restrictions [0] • E: Encryption only [1] • D: Decryption only [2] • S: Signature only [3] • 0: IV [4] • G: MAC generate [5] • V: MAC verify [6] • C: Calculate = generate or verify) [7] • [6]–[F]: Reserved **Note:** To save storage space in RAM, the mode of use attribute is converted to [x], a hex number ranging form 0–F (4 bits). In the response packet (to packet 04), the IPP converts the hex number back to characters used in the GISKE specification.
Key Version (KV)	2AH	(optional) Only if the master key present at address [a]. The 2-digit ASCII character version number is optionally used to re-inject old keys. If not used, this field is filled with ASCII 0 (0x30). **Note:** The IPP allocates 1 byte per key for each key version register.

Data Element	Characteristic	Comments
Key Length (KL)	1AH	(optional) Only if the master key present at address [a]. • 1: single-length key • 2: double-length key • 3: triple-length key **Note:** The IPP allocates 1 byte per key for each key version register.
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

Response Packet GISKE Block Format 04 Length

- MAX: 12 characters
- MIN: 12 characters

Response Packet 04 GISKE Block Format Example

```
<SI>040[KUA][MOUA][MACMA][KV][KL]<SO>{LRC}
```

Response Packet 04 GISKE Block Format Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>04[a]<SO>{LRC}`	→	
	←	• ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs • PINpad checks requested address [a]. `<SI>04[r][KUA][MOUA][MACMA][KV][KL]<SO>{LRC}`
• ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs	→	
	←	EOT

MS Packet 08: Select a Master Key

The master device sends this packet to the IPP to select one of the ten possible master keys (0–9). It is recommended that the master device should always send this packet first before sending a packet (for example, Packet Z60: Accept and Encrypt PIN (VISA Mode) to request for PIN entry.

MS Packet 08 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 08
[a]	1N	Master Key address: 0–9
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

MS Packet 08 Length

- MAX: 6 characters
- MIN: 6 characters

MS Packet 08 Example

To select Master Key 7:

```
<SI>087<SO>{LRC}
```

MS Packet 08 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>08[a]<SO>{LRC}`	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs • PINpad makes master key [a] active.
	←	EOT

Notes:

1 The 1DES and 3DES key usage rules (see Rules for Loading the Master Key (MS Only)) applies when selecting a master key. If the selecting master key is not available or is not applicable due to the 1DES or 3DES key usage rule, no response is returned to the master device.

2 If the master key address does not contain any key, the IPP still sets the currently selected key as the active master key due to a backward compatibility requirement.

MS Packet 71: Transfer PIN Block

Response packet to Packet Z60: Accept and Encrypt PIN (VISA Mode) and Packet Z63: Accept and Encrypt PIN-Custom PIN Entry Requirements (VISA Mode). The IPP encrypts the formatted clear-text PIN block and sends the cipher-text PIN block to the master device.

MS Packet 71 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	2AN	Value: 71
Packet Delimiter	1A	Value: (.), 2Eh
Function Key Indicator	1N	Value is 0; Function key feature not implemented.
PIN Length	2N	Range 00, 04 to 12
PIN Block Format	2N	Value: 01; Format of PIN block prior to encryption
Encrypted PIN Block	16H	The 64-bit encrypted PIN block represented as 16 hexadecimal digits. Present only if PIN entered.
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error check

MS Packet 71 Length

- MAX: 27 characters
- MIN: 27 characters

MS Packet 71 Example

```
<STX>71.000010123456789123456<ETX>{LRC}
```

This packet 71 is the response packet to PIN request Z60 and Z63 when no errors are detected in the Z60 or Z63 packet. If errors are detected in the Z60 or Z63 packet, the response packet is in the following format:

MS Response Packet 71 Format: Errors in Z60 or Z63 Packets

Data Element	Characteristic	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	2AN	Value: 71

Data Element	Characteristic	Comments
Error Code	1N	• 1 = no master key • 2 = account or working key error • 3 = PIN too long. • 4 = PIN too short / non-decimal digit(s) in PIN. • 5 = PIN pad used as DUKPT[a] • 6 = Master key attributes error • 7 = KOF/GISKE working key attributes error, key attributes: key usage, algorithm, mode of use, key version, or key length
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error Check

a. Error code 5 does not occur in the IPP, since it supports simultaneous DUKPT and MS.

MS Packet 71 Length

- MAX: 6 characters
- MIN: 6 characters

MS Packet 71 Example

```
<STX>711<ETX>{LRC}
```

Packet 07: Dummy Packet

To have the IPP pass the DES reliability test on the MKI program, a dummy packet 07 is added. When this packet is received, the IPP only returns an <ACK>, followed by an <EOT> after a 1 second delay (this delay is necessary for compatible with the MKI program).

DUKPT-Specific Packets

The following packets are specific to DUKPT operation. Two DUKPT modes are implemented in IPP7: 1DES or 3DES. All keys associated with DUKPT are erased when switching between 1DES and 3DES DUKPT modes.

Packet 19: Select a DUKPT Engine

The application sends this packet to the IPP to select one of the DUKPT engines (0, 1, or 2). It is recommended that the application always send this packet first before sending a DUKPT packet (for example, packet Z60, Z63, 76, Z69, and 90).

NOTE

This packet was added for IPP8 emulation.

DUKPT Packet 19 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 19
[a]	1N	DUKPT Engine: “0”, “1”,or “2”
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

DUKPT Packet 19 Length

- Maximum: 6 characters
- Minimum: 6 characters

Sample Packet

To select second DUKPT Engine:

```
<SI>192<SO>{LRC}
```

DUKPT Packet 19 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>19[a]<SO>{LRC}`	→	
	←	• ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs).
	←	Echo packet 19 setting `<SI>19[a]<SO>{LRC}`
• ACK if LRC okay • NAK if LRC incorrect • (EOT after 3 NAKs)	→	
		… IPP changes DUKPT engine …
	←	EOT to terminate process.

Notes

- If there is any packet format error, IPP does not echo the response packet back to the master device. The incorrect packet format includes out of range DUKPT engine, incorrect packet type, incorrect packet length, etc.
- The default DUKPT engine is set to “0”.

Packet 25: Check the DUKPT Engine

The application sends this packet to the IPP to check the current active DUKPT engines (“0”, “1”, or “2”).

NOTE

This packet is added for IPP8 emulation.

Request Packet Format

```
<SI>25<SO>{LRC}
```

Packet 25 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 25
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

Packet 25 Length

- Maximum: 5 characters
- Minimum: 5 characters

Response Packet Format

```
<SI>25[PUSN]<SO>{LRC}
```

Packet 25 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 25
[a]	1N	Active DUKPT Engine: “0”, “1”,or “2”
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

Packet 25 Length

- Maximum: 6 characters
- Minimum: 6 characters

Sample Packet

To Check DUKPT Engine:

```
<SI>25<SO>{LRC}
```

Response packet, DUKPT Engine “1” = active DUKPT Engine:

```
<SI>251<SO>{LRC}
```

Packet 25 Communication Protocol

Master Device	Transmit Direction	IPP
`<SI>25<SO>{LRC}`	→	
	←	• ACK if LRC okay. • NAK if LRC incorrect (EOT after 3 NAKs).
	←	Response current active DUKPT engine. `<SI>25[a]<SO>{LRC}`
• ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs).	→	
	←	EOT terminates session.

DUKPT Packet 73: Transfer PIN Block (for Packets Z60 or Z63)

Response packet to Packet Z60: Accept and Encrypt PIN (VISA Mode) and Packet Z63: Accept and Encrypt PIN-Custom PIN Entry Requirements (VISA Mode). The IPP encrypts the formatted clear-text PIN block and sends the cipher-text PIN block to the master device.

DUKPT Packet 73 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, Value: 02h
Packet Type	2AN	Value: 73
Packet Delimiter	1A	Value: (.), 2Eh
00000	5N	Value: 00000
[KSN]	10–20AH	Key serial number; hex. (leading Fs suppressed). Presented only if a PIN is entered; length is 0 if no PIN is entered.
Encrypted PIN Block	16AH	The 64-bit encrypted PIN block represented as 16 hexadecimal digits.
<ETX>	1H	End of text, Value: 03h
{LRC}	1H	Error check

DUKPT Packet 73 Length

- MAX: 47 characters
- MIN: 27 characters

DUKPT Packet 73 Example

```
<STX>73.0000001234567890123456789123456<ETX>{LRC}
```

Packet 73 is the response packet to Packet Z60: Accept and Encrypt PIN (VISA Mode), the PIN request packet with no errors detected. If errors are detected in the Z60 or Z63 packet, the response packet is in the following format:

MS Response Packet 73 Format: Errors in Z60 or Z63 Packet

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 73
Error Code	1N	• 1 = no key • 2 = account error • 3 = PIN too long • 4 = PIN too short / non-decimal PIN digit in PIN • 5 = PIN pad used as MS[a] • 6 = PIN pad has over 1 million transactions
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

a. Error code 5 does not occur in the IPP, since the IPP supports simultaneous DUKPT and MS.

DUKPT Packet 73 Length

- MAX: 6 characters
- MIN: 6 characters

DUKPT Packet 73 Example

```
<STX>731<ETX>{LRC}
```

DUKPT Packet 90: Load Initial Key Request

Loads initial key to the IPP. After the initialization of packet 21, future keys, the IPP responds with packet 91 with confirmation status.

DUKPT Packet 90 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, Value: 02h
Packet Type	2AN	Value: 90
[IPEK]	16H or 32H	• 16AH: 1DES Mode • 32AH: 3DES Mode The IPEK is a 8- (16AH) or 16- (32AH) byte binary key value. To put it in packet format, it must be converted from binary to ASCII hex value.

Data Element	Characteristic	Comments
[KSN]	20H	Key Serial Number; hex (leading Fs included)
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check character

DUKPT Packet 90 Length

One character equals one byte.

- MAX: 57 characters with IPEK 32AH
- MIN: 41 characters with IPEK 16AH

NOTE The difference between DUKPT 1DES mode and DUKPT 3DES mode is in the size of the initial PIN encryption key and the sizes of the future keys.

DUKPT Packet 90 Communication Protocol

Master Device	Transmit Direction	IPP
90 Packet	→	
	←	• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs • Initialization of 21 Future Keys
		Packet 91 with confirmed status
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs	→	

DUKPT Packet 91: Load Initial Key Response

Response packet to packet 90. Response to controller with confirmation status. If 21 Future Keys are successfully initialized, Packet 91 responds with confirmation. Else, negative response packet 91 returns.

DUKPT Packet 91 Format

Data Element	Characteristic[a]	Comments
<STX>	1H	Start of text; 02h value
Packet Type	2AN	Value: 91

Data Element	Characteristic[a]	Comments
[CS]	1N	Confirmation status: • 0 = Confirmed • 1 = Not confirmed • 2 = (IPP7 only) Error; incorrect key length. Confirmation status 2 only applies to IPP7. It indicates that the length of the initial PIN encryption key does not comply with 1DES or 3DES DUKPT mode, as follows:

Initial PIN encryption key length (through packet 90) sent by the master device	IPP7 Current DUKPT Mode	[CS] response from the IPP
16AH	3DES	2
32AH	1DES	2

Data Element	Characteristic[a]	Comments
<ETX>	1H	End of text; 03h value
{LRC}	1H	Error check character

a. The following explains the alpha-numeric symbols represented in this table:
A - ASCII
AH - ASCII Hex
AN - ASCII Numeric
H - Hex
N - Numeric
1N and 1AN are the same, for example, 1N or 1AN is a 1-byte field ‘1’.

DUKPT Packet 91 Length

- MAX: 6 characters
- MIN: 6 characters

DUKPT Packet 91 Communication Protocol

Master Device	Transmit Direction	IPP
		Packet 91
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs	→	

DUKPT Packet 76: PIN Entry Test Request

Directly presets the PIN code '1234' to do encryption and send response packet 71.

DUKPT Packet 76 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 76
[account#]	8-19N	Card account number
<FS>	1H	Field separator, value: 1Ch
[C/D]	1H	Credit/Debit indicator, value: 43h/44h
[amount]	3-7N	Transaction amount must include the decimal point.
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

DUKPT Packet 76 Length

- MAX: 33 characters
- MIN: 18 characters

NOTE

The amount filed must be present in the packet command, but the format is not confirmed.

DUKPT Packet 76 Communication Protocol

Master Device	Transmit Direction	IPP
76 Packet	→	
	←	• ACK if LRC • NAK if LRC incorrect
		Packet 71 with PIN = 1234
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs	→	

DUKPT Packet 71: Transfer PIN Block - (for Packet 76)

Response packet to packet 76, request for PIN. The IPP encrypts the formatted clear-text PIN block and sends the cipher-text PIN block to the master device. (refer to the *VISA PIN Processing and Data Authentication* specification, International version 1.0)

Packet 71 has a different packet format and meaning than the response PIN block 71 in MS. This is for compatibility with existing third parties (for example, Racal) to initialize the DUKPT key.

DUKPT Packet 71 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 71
Function Key	1N	Value: 0, function key indicator feature not implemented
Key Serial Number	10-20H	Hexadecimal (leading Fs suppressed.); Included only if PIN entered; length is 0 if no PIN entered
Encrypted PIN Block	16H	The 64-bit encrypted PIN block represented as 16 hexadecimal digits; length is 0, if no PIN entered.
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

DUKPT Packet 71 Length

- MAX: 42 characters
- MIN: 6 characters (if NULL entered)

DUKPT Packet 71 Example:

```
<STX>710[KSN]0123456789123456<ETX>{LRC}
```

When no errors are detected in packet 76, the IPP returns response packet 71. If errors are detected in packet 76, response packet 71 is in the following format:

DUKPT Packet 71 Format: Errors Detected in Packet 76

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 71
Error Code	1N	• 1 = no key • 2 = account error • 5 = C|D field error • 6 = PIN pad has over 1 million transactions
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

DUKPT Packet 71 Length

- MAX: 6 characters
- MIN: 6 characters (if NULL entered)

DUKPT Packet 71 Example

```
<STX>711<ETX>{LRC}
```

DUKPT Packets 92 and 93

The DUKPT reinitialization request and reinitialization response packets are not supported.

DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request

On receipt of the Z69 packet, the terminal reads the user’s PIN from the keyboard, echoing to the display an asterisk for each digit accepted. The PIN length can be between 4 and 12 digits.

DUKPT Packet Z69 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	3AN	Value: Z69
[account#]	8–19N	Card account number.
<FS> or <US>	1H	<FS> is the field separator that indicates VISA MACing is used. <US> is the field separator that indicates ANSI 9.19 MACing is used.
[C/D]	1H	Credit/debit indicator, value 43h/44h
[amount]	3–13N	Transaction amount including the decimal point.
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check character

DUKPT Packet Z69 Length

- MAX: 24 characters
- MIN: 45 characters

DUKPT Packet Z69 Communication Protocol

Master Device	Transmit Direction	IPP
Z69 Packet	→	
	←	• ACK of LRC okay • NAK if LRC incorrect • EOT after 3 NAKs
	←	Packet 75 with confirmed status
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs		

DUKPT Packet Z69 Example:

VISA:

```
<STX>Z6901234567890<FS>C19.99<ETX>{LRC}
```

ANSI:

```
<STX>Z6901234567890<US>C19.99<ETX>{LRC}
```

Errors returned by write()

Some packet format errors are caught when the packet is written to the IPP. In this case, `write()` returns –1 and errno set. The packet is not ACKed or NAKed, and no response packet returns.

Z60 Format Error	errno
No <FS>	EINVAL

DUKPT Packet 75: DUKPT Accept and Encrypt PIN/Data Authentication Response

Response packet to packet DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request or Packet 78: DUKPT Accept and Encrypt PIN/Data Authentication Test Request to the controller with confirmation status. Authentication code #1 is the MAC on this message. If the request is approved, the MAC received with the approval response message exactly matches authentication code #2. If the request is declined, the MAC received with the decline response message must exactly match authentication code #3.

DUKPT Packet 75 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 75
[Auth. Code #1]	8H	Authentication code #1, message MAC
		In ANSI mode, Auth Code is padded with all 0s (0x30h).
[Auth. Code #2]	8H	Authentication code #2, transaction approved check value
		In ANSI mode, Authentication Code #2 is the left 4 bytes of the MAC value.
[Auth. Code #3]	8H	Authentication code #3, transaction declined check value
		In ANSI mode, Authentication Code #3 is the right 4 bytes of the MAC value.
Function Key	1N	Value is 0, Function Key Indicator feature not implemented

Data Element	Characteristic	Comments
Key Serial Number	10–20H	Hexadecimal (leading Fs suppressed.); Included only if PIN entered; Length is 0 if no PIN entered
Encrypted PIN Block	16H	The 64-bit encrypted PIN block represented as 16 hexadecimal digits. Length is 0, if no PIN entered.
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check character

DUKPT Packet 75 Length

- MAX: 57 characters
- MIN: 67 characters

DUKPT Packet 75 Example

VISA:

```
PC ---> PINPAD : <STX>7801234567890<FS>C19.99<ETX>{LRC}
PC <--- PINPAD : <ACK>
PC <--- PINPAD :
<STX>75FCD3CA45D04396624CF6892B04A000002468000048D5D7AF0333800FD<ETX>{LRC
}
PC ---> PINPAD : <ACK>
```

ANSI:

```
PC ---> PINPAD : <STX>7801234567890<US>C19.99<ETX>{LRC}
PC <--- PINPAD : <ACK>
PC <--- PINPAD :
<STX>7500000000D04396624CF6892B04A000002468000048D5D7AF0333800FD<ETX>{LRC
}
PC ---> PINPAD : <ACK>
```

Packet 75 is the response packet to packet Z69 or packet 78, PIN request, when no errors are detected in the request packet. If errors are detected in packet Z69 or packet 78, the response packet is in the following format:

DUKPT Packet 75 Format: Errors Detected in Packet Z69 or Packet 78

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 75

Data Element	Characteristic	Comments
Error Code	1N	• 1 = no key • 2 = account error • 3 = PIN too long • 4 = PIN too short/non-decimal digit in PIN • 5 = C|D field error/not DUKPT mode • 6 = PIN pad has over 1 million transactions • 7 = amount error • 8 = ANSI MAC not allowed when using 1DES DUKPT
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

DUKPT Packet 75 Length

- MAX: 6 characters
- MIN: 6 characters

DUKPT Packet 75 Example

```
<STX>751<ETX>{LRC}
```

Packet 78: DUKPT Accept and Encrypt PIN/Data Authentication Test Request

Packet 78 requests PIN encryption and MAC processing using a fixed PIN of '1234'. The response packet is packet 75.

NOTE

Packet 78 is similar to packet Z69, but the PIN code is preset to “1234.” The user is *not* prompted to enter a PIN.

This packet is used for testing and should not be used by applications.

DUKPT Packet 78 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	2AN	Value: 78
[account#]	8–19N	Card account number
<FS> or <US>	1H	<FS> is the field separator that indicates VISA MACing is used. <US> is the field separator that indicates that ANSI 9.19 MACing is used.
[C/D]	1H	Credit/Debit indicator, value: 43h/44h
[amount]	3-13N	Transaction amount, decimal point included.

Data Element	Characteristic	Comments
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

DUKPT Packet 78 Length

- MAX: 33 characters
- MIN: 18 characters

NOTE Per the VISA specification: The amount field should be 3–12 numeric characters, excluding the decimal point. Due to compatibility concerns, this packet is designed to be the same as the Z60 or 76 packet commands. However, the amount length is extended to be able to accept 12 numeric characters. The lack of a decimal point or multiple decimal points does not cause an error. The PIN pad does not confirm the decimal point location. The MAC value is calculated across the entire account number and all amount numbers, and the decimal point is filtered out.

DUKPT Packet 78 Example

VISA:

```
<STX>7801234567890<FS>C19.99<ETX>{LRC}
```

ANSI:

```
<STX>7801234567890<US>C19.99<ETX>{LRC}
```

DUKPT Packet 78 Communication Protocol

Master Device	Transmit Direction	IPP
78 Packet	→	
	←	• ACK if LRC • NAK if LRC incorrect
		Packet 75 with PIN = 1234
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs	→	

MAC-Specific Packets

This section describes the master-session MAC generation of received message packets for the IPP. Two packet formats are specified: Z66 and Z67. The detailed module design and interface design are discussed. ANSI (Standard) MAC algorithms are used.

MAC Packet Z66: Request MAC

Used by the master device to direct the IPP to generate the MAC of the current packet. If it is the first Z66 packet, the IPP begins MAC generation. If it is the last Z66 packet, the IPP completes the MAC calculation for current packet, and returns the MAC to the master device through the Z67 packet. Otherwise, the IPP calculates the MAC from current packet and stores it in memory.

MAC Packet Z66 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	3AN	value: Z66
[flag]	1N	ANSI (Standard) MAC: ASCII Data: Range: 4–5 • 4 = the last packet • 5 = the first/middle packet Binary Data; Range: 6–7 • 6 = the last packet • 7 = the first/middle packet
[sequence]	2N	Range: 00–99
Master Key Pointer	1N	Optional; Range: 0–9
<FS>	1H	Field separator, value: 1Ch
Working Key	16H	Encrypted working key for DES
<FS>	1H	Field separator, value: 1Ch
Second Key	1N	Optional; Second master key pointer; Range: 0–9
<FS>	1H	Field separator, value: 1Ch

Message for MAC[a]	8*XAN0	ASCII message or ASCII-coded binary data:
		X= 0–28 for ASCII data
		X= 0,2,4,6,...,27,28 for binary data

<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

a. ASCII messages for MAC should not include ETX(0x03) or SO(0x0E).

MAC Packet Z66 Length

- MAX: 255 characters
- MIN: 12 characters

MAC Packet Z66 Example

```
<STX>Z663002<FS>0123456789123456<FS><FS>0123456789ABCDEF<ETX>{LRC}
```

Notes

1 A maximum of 100 Z66 packets can be sent during one MAC session.

2 8*XAN in “Message for MAC” represents the number of 8-byte (or character) blocks. For example,

- X = 0: no message data
- X = 1: 8 bytes of message data
- X = 2: 16 bytes of message data
- X = 3: 24 bytes of message data
  :
  :
- X = 27: 216 bytes of message data
- X = 28: 224 bytes of message data

For ASCII data, all values of X from 0–28 are allowed.

For Binary data, only 0, 2, 4, 6,...., 26, 28 are permitted. (X = 2 * N, where N = 0 to 14.)

3 If the length of “Message for MAC” is not a multiple of 8 in the final Z66 packet, the PIN pad automatically pads it with zeros (ASCII 30h) internally.

4 An example of a 8-byte data block for the ASCII message “AMT$1.99” is “414D5424312E3939”

5 ASCII-coded binary message is two hex digits that represent a byte value, see the conversion result above.

6 If the working key is loaded in 1DES key-only format, either ANSI (standard) or MAC is used (depending on the status of the flag in the packet Z66).

7 If the working key is loaded in the GISKE format, the IPP uses the MAC algorithm specified in the Key Usage Attributes of the GISKE key block.

8 When the key length and the MAC algorithm do not match, an error code (key attribute/usage error) returns. For example, a single-length key is used with a 3DES MAC algorithm.

9 MAC algorithms used: ISO 9797-1 MAC Algorithm 1–56 bits, MAC Algorithm 1–112 bits, MAC Algorithm 2–112 bits, MAC Algorithm 3–112 bits, MAC Algorithm 4–112 bits, MAC Algorithm 5–56 bits, MAC Algorithm 5–112 bits.

10 The GISKE working key can only be a single- or double-length key. Master key used to encrypt the working key can be a single-, double-, or triple-length key (the GISKE length encryption rule still applies). If a triple-length GISKE working key is used in Z66, a working key error is returned.

Rules of Request MAC

The following rules are imposed to the size of the “Message for MAC” field.

Packet Type	Size of X	Maximum Size of Message (bytes)	Apply to Message Sequence	Comments
Key-only format	ASCII: X = 0, 1, 2 – 27, 28	224	00–99	
	Binary: X = 0, 2, 4 – 26, 28	224		
GISKE Key Block Format mode	ASCII: X = 0, 1, 2 – 14, 15	120	00 – 99	Due to size of GISKE key block, the size of message is reduce to 120 bytes.
	Binary: X =0, 2, 4 – 12, 14	112		

MAC Packet Z67: Return MAC

This multi-purpose packet:

- Sends a signal to the master device that the IPP is ready for the next Z66 packet.
- Sends an error code to the master device if there any error is detected during the MAC session.
- Sends the MAC value to the master device.

MAC Packet Z67 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, value: 02h
Packet Type	3AN	Value: Z67
Process Code	1N	Range: 0–9: • 0 = no error and MAC follows • 1 = ready for next Z66 packet and no MAC follows • 2 = out-of-order error and no MAC follows • 3 = [pointer] error and no MAC follows • 4 = [second key] error and no MAC follows • 5 = packet frame error and no MAC follows • 6 = [flag] error • 7 = [message] error • 8 = [working key] error/GISKE key usage, algorithm, mode of use, or key length error • 9 = incorrect key attributes of the master key (first or second)
MAC field	16H	Optional; only sent when no errors are detected

Data Element	Characteristic	Comments
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

MAC Packet Z67 Length

- MAX: 23 characters
- MIN: 7 characters

MAC Packet Z67 Example

```
<STX>Z671<ETX>{LRC}
```

Packet 72: Cancel MAC Session

Cancels the MAC session if an error is detected in a multi-MAC session. After the IPP receives packet 72, ACK is returned to terminate the session.

MAC Packet Z66 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of text, Value: 02h
Packet Type	3AN	Value: 72
<ETX>	1H	End of text, value: 03h
{LRC}	1H	Error check

Packet 72 Length

- MAX: 5 characters
- MIN: 5 characters

Packet Z72 Example

```
<STX>712 <ETX>{LRC}
```

Packet 72 Communication Protocol

Master Device	Transmit Direction	IPP
`<STX>72<ETX>{LRC}`	→	
	→	• ACK if LRC • NAK if LRC incorrect

MAC Module Design

ANSI (Standard) MAC Algorithms

The algorithm to calculate the MAC is fully compatible with the ANSI X9.19 1986, *Financial Institution Retail Message Authentication* specification. Within this standard, there are two modes of operation: CBC (Cipher Block Chaining) and CFB-64 (64-bit Cipher Feedback). In IPP5 implementation (that is, IPP5 and higher, including IPP6, IPP7, and V/OS IPP), CBC is used for MAC calculation.

The master key and the working key for MAC calculation can be downloaded with Z66 packet. Selection of these keys depends on the first Z66 packet configurations within each MAC session, as summarized in the following table.

MAC for Master and Working Keys

[pointer]	[working key]	Selection
present	present	Master key selected by [pointer]; working key decrypted by master key.
absent	present	Working key decrypted by current active master key.

After the MAC calculation, there is an optional procedure used to increase protection against exhaustive key determination. This procedure can be turned on/off by the `[second key]` field of the first Z66 packet. If this second key was provided with the first Z66 packet, this procedure generates the final MAC and uses `[second key]` as the master key pointer. If no `[second key]` is provided, no procedure is performed on the current MAC.

One thing to note is that `[second key]` is used on a session-by-session basis. Each `[second key]` field of the first Z66 packet defines its own optional procedure on/off status during that MAC session. For more detailed information about MAC optional procedure, please refer section 2.4.4.5 of the ANSI X9.19 specification.

After the process completes, a 64-bit MAC is generated. This MAC value returns to the master device with packet Z67. If there any errors are detected during the MAC process, packet Z67 returns with `[code]` set to an error code.

References

IPP Specifications

Software Technical Specification IPP7 P/N 22985

Software Technical Specification PP1000se & IPP8 P/N 23143

Appendix 15 of the *Verix V Programmer's Guide*, P/N 23230

GISKE Specification

GISKE Key Block Specification P/N 22986.

CHAPTER 17

Account Data Encryption Module

This section describes V/OS Account Data Encryption (ADE) module, and presents ADE function calls.

ADE State Controls

The ADE module is controlled by three states:

- ADE on or off state; default is off
- ADE feature state; default is disabled
- ADE keys are loaded; default is no keys loaded

  ADE must be on and a minimum of one usable ADE key must be loaded to enable ADE.

Turning ADE On

When the following is true, call ade_status() or ade_active() to turn on ADE:

NOTE

ADE can only be turned on if these requirements are met.

- At least one usable ADE key is loaded.
- A valid FE license for ADE is installed.

ADE Key Loading

V/OS manages ten 2TDEA DUKPT engines for account data encryption. Key generation applies the “Data Encryption, request or both ways” variant, as defined in ANSI X9.24-1:2009.

Unique Key Enforcement

V/OS ensures that all keys are unique by rejecting duplicate key loads. When a DUKPT key loads, the KSN is compared to the KSNs of all previously loaded DUKPT keys to ensure that it is unique. Key uniqueness is enforced across IPP DUKPT, VSS DUKPT, and ADE DUKPT. A SHA256 of the DUKPT IK is also stored. If the KSN matches then the SHA256 of the DUKPT, IK is also checked.

During a key load, the target key slot is not checked for uniqueness, which allows a key to be replaced by an identical key. Any key clearing caused by a key load is always performed before the uniqueness check.

Cleartext Key Loading using Virtual Terminal Management

In System Mode, use the **ADE Key Loading** menu to load ADE keys. During ADE key loading:

- Keys are sent in cleartext.

NOTE

Only perform cleartext key loading in a secure environment.

At the start of a cleartext key load session, all existing ADE keys are cleared. IPP, VSS, VCL, and VRK keys are not affected. An ADE cleartext key load session begins when the first valid cleartext ADE key load packet is received after a restart.

- Keys are loaded using a packet-based protocol (see IPP MS and DUKPT Communications Packets).
- The default port settings are 1200 7E1.
- Two timeouts are available:
  - 15-minute session timeout – Time since session started.
  - 1-minute communications timeout – Time since terminal last received data from key loader.
- Two methods end the key loading session:
  - When the user presses an exit key.
  - When the Timeout timer expires.

ADE Packets

This section presents the packets common to ADE protocols.

Packet A90

This packet loads a 2TDEA ADE DUKPT key. The packet length is 60 characters (bytes).

Master Device	Transmit Direction	IPP
`<STX>A90[Slot][IK][KSN]<ETX><LRC>`	→	
	←	ACK
	←	`<STX>A90[Result]<ETX><LRC>`
ACK	→	
	←	EOT

where,

- Slot is a 2-byte ADE key slot. Valid values: 00–09
- IK is a 16-byte DUKPT initial key (convert to 32 ASCII Hex characters to put in packet)

- KSN is a 10-byte key serial number (convert to 20 ASCII hex characters to put in packet)
- Result is a 2-byte result code:
  - 00 = Success
  - 01 = Internal error
  - 02 = Invalid format
  - 03 = Unsupported key slot
  - 04 = Duplicate key

Encrypted Key Loading using VRK

You can load ADE keys using VRK. The following values must be in your TR-31 header:

- Algorithm: “T” (TDEA)
- Mode of use: “E” (Encryption)
- Optional Block 04 (Key Engine Type): “a” (ADE)
- Optional Block 05 (Key Slot of Engine): A single digit between 0 and 9, inclusive

ADE System Mode Menus

On your V/OS terminal in System Mode the following two ADE menus are available:

- ADE Status
- ADE Key Load

ADE Status Menu

Displays the current status for the following ADE features:

- Key status for ADE DUKPT engines 0 through 9
- ADE on/off status (default: off)
- ADE feature status (default: disabled)

This menu displays the same information available by calling ade_status().

ADE Key Load

You can start the ADE cleartext key loading service from this menu.

ADE Functions

This section presents the ADE function calls.

#define Values

ADE Encryption Algorithms

```
#define ADE_ALG_TDEA       0
```

ADE Encryption Modes of Operation

```
#define ADE_MODE_ECB       0
#define ADE_MODE_CBC       1
```

ADE IV Settings

```
#define ADE_IV_NONE       0
#define ADE_IV_ZERO       1
#define ADE_IV_RAND       2
```

ADE Padding Schemes

```
#define ADE_PAD_NONE      0
#define ADE_PAD_PKCS7     1
#define ADE_PAD_X923      2
#define ADE_PAD_ISO7816   3
```

ADE Error Codes

Error	Code	Description
ADE_SUCCESS	0	Success.
ADE_ERR_PM_PTR	-1	A pointer parameter references unusable memory (for example, a null pointer, the buffer is too small, and so on).
ADE_ERR_PM_LEN	-2	`ade_encrypt_in ptextLen` is an invalid value.
ADE_ERR_PM_KEY	-3	`ade_encrypt_in keyIndex` is an invalid value.
ADE_ERR_PM_ALG	-4	`ade_encrypt_in encAlg` is an invalid value.
ADE_ERR_PM_MODE	-5	`ade_encrypt_in encMode` is an invalid value.
ADE_ERR_PM_IV	-6	`ade_encrypt_in iv` is an invalid value.
ADE_ERR_PM_PAD	-7	`ade_encrypt_in pad` is an invalid value.
ADE_ERR_NO_KEY	-11	A key cannot be retrieved from the specified slot.
ADE_ERR_OFF	-12	ADE is turned off.
ADE_ERR_GENERIC	-99	An undefined or unexpected error occurred.

ADE Status Codes (bitmap)

```
#define ADE_STS_ENG0      1<<0
#define ADE_STS_ENG1      1<<1
#define ADE_STS_ENG2      1<<2
#define ADE_STS_ENG3      1<<3
#define ADE_STS_ENG4      1<<4
#define ADE_STS_ENG5      1<<5
#define ADE_STS_ENG6      1<<6
#define ADE_STS_ENG7      1<<7
#define ADE_STS_ENG8      1<<8
#define ADE_STS_ENG9      1<<9
#define ADE_STS_ON        1<<10
#define ADE_STS_ENABLED       1<<11
```

ade_encrypt()

Encrypts the input data using the specified settings and ADE key.

Prototype

```
int ade_encrypt(ade_encrypt_in* in, ade_encrypt_out* out);
```

Parameters

`ade_encrypt_in`	Where `in` is:

- `ptext` – Plain text data to encrypt.
- `ptextLen` – The number of bytes to encrypt:
  - If `ade_encrypt_in pad` does not equal `ADE_PAD_NONE`, valid values are 1–383, inclusive.
  - If `ade_encrypt_in pad` equals `ADE_PAD_NONE`, valid values are between 1–384, inclusive. This value must be divisible by the block size of the encryption algorithm (`ade_encrypt_in encAlg`)
- `keyIndex` – ADE key index, valid values are 0–9, inclusive, for ADE DUKPT engines 0 through 9.
- `encAlg` – Encryption algorithm, valid values are:
  - `ADE_ALG_TDEA` = TDEA (block size is 8 bytes).
- `encMode` – Encryption mode of operation, valid values are:
  - `ADE_MODE_ECB` = ECB mode
  - `ADE_MODE_CBC` = CBC mode
- `iv` - IV setting, valid values are:
  - `ADE_IV_NONE` = no IV (not allowed for CBC mode)
  - `ADE_IV_ZERO` = zero IV
  - `ADE_IV_RAND` = random IV
- `pad` – The padding scheme, valid values are:
  - ADE_PAD_NONE = No padding added
  - ADE_PAD_PKCS7 = PKCS#7 padding. Each padding byte equals the padding length. For example:

    ```
    XX XX XX XX 04 04 04 04
    ```
  - ADE_PAD_X923 = ANSI X.923 padding. The final padding byte equals the padding length. All other padding bytes equal 0x00. For example:

    ```
    XX XX XX XX 00 00 00 04
    ```
  - ADE_PAD_ISO7816 = ISO/IEC 7816-4 padding. The first padding byte equals 0x80. All other padding bytes equal 0x00. for example:

    ```
    XX XX XX XX 80 00 00 00
    ```

`ade_encrypt_out`	Where `out` is:

- `ctext` – Cipher text. This output buffer must be able to hold the entire cipher text; not to exceed 384 bytes.
- `ctextLen` – The number of bytes of cipher text.
- `ivData` - IV data. This output buffer must be able to hold the entire IV data; not to exceed the block size of the encryption algorithm.

Note: If `ade_encrypt_in iv` equals `ADE_IV_NONE`, `ivData` is not used.

- `supData` – Supplementary data. This output buffer hold all supplementary output data. For example, if using a DUKPT engine, `ade_encrypt_out supData` contains the 10-byte KSN.

Return Values

ADE_SUCCESS	Success.
ADE_ERR_PM_PTR	A pointer parameter references unusable memory (for example, a null pointer, the buffer is too small, and so on).
ADE_ERR_PM_LEN	`ade_encrypt_in ptextLen` is an invalid value.
ADE_ERR_PM_KEY	`ade_encrypt_in keyIndex` is an invalid value.
ADE_ERR_PM_ALG	`ade_encrypt_in encAlg` is an invalid value.
ADE_ERR_PM_MODE	Parameter `ade_encrypt_in encMode` is an invalid value.
ADE_ERR_PM_IV	Parameter `ade_encrypt_in iv` is an invalid value.
ADE_ERR_PM_PAD	Parameter `ade_encrypt_in pad` is an invalid value.
ADE_ERR_NO_KEY	A key cannot be retrieved from the specified slot.
ADE_ERR_OFF	ADE is disabled.
ADE_ERR_GENERIC	An undefined or unexpected error occurred.

Example

```
int ade_encrypt(ade_encrypt_in* in, ade_encrypt_out* out)

typedef struct
{
   unsigned char* ptext;      // Plaintext
   int ptextLen;              // Plaintext Length
   int keyIndex;              // Key Index
   int encAlg;                // Encryption Algorithm
   int encMode;               // Encryption Mode of Operation
   int iv;                    // IV Setting
   int pad;                   // Padding Scheme
} ade_encrypt_in;

typedef struct
{
   unsigned char* ctext;          // Ciphertext
   int ctextLen;                  // Ciphertext Length
   unsigned char* ivData;         // IV Data
   unsigned char* supData;        // Supplementary Data
} ade_encrypt_out;
```

ade_status()

Returns the status of the ADE module. ADE is on when the “turning ADE on requirements” are met.

Prototype

```
int ade_status();
```

Return Values

Returns a 32-bit map of the ADE status:

ADE_STS_ENG0–ADE_STS_ENG9	Status of ADE DUKPT engines 0 through 9: • 0 = Engine is not usable (for example, no key loaded or engine exhausted) • 1 = Engine is usable (for example, key is loaded)
ADE_STS_ON	ADE status • 0 = ADE is turned off • 1 = ADE is turned on
ADE_STS_ENABLED	ADE feature status • 0 = ADE is disabled • 1 = ADE is enabled

ade_active()

Indicates if the ADE system is fully functional. Calls ade_status() to turn on ADE if the Turning ADE On requirements are met.

Prototype

```
int ade_active();
```

Return Values

= 0	The ADE system is not currently usable.
≠ 0	ADE is turned on and has a minimum of one usable key loaded.

CHAPTER 18

Magnetic Stripe Reader

All V/OS terminals include a triple track magnetic stripe reader.

Accessing the magnetic stripe reader requires linking with the shared library: `libvfimsr.so`. The header file `msrDevice.h` is used by the application to access the library.

Magnetic Stripe Reader Functions

Use the following calls to access the magnetic stripe reader.

msrOpen()

Prepares the firmware to accept and store card reads. If the programmer does not make this call, the terminal ignores all card reads. The MSR device allows only one open at a time.

Prototype

```
int msrOpen(int flags, void *callback);
```

Parameters

`flags`	Specify device access permissions for the MSR device.	
	O_RDONLY	Read only.
	O_RDWR	Read and write.
	O_NONBLOCK	Read is non-blocking (default is blocking).
`callback`	Pointer to callback function. If available, this callback function is called when data is available.	

Return Values

>= 0	Magnetic stripe reader is open and ready.
< 0	Error.

msrRead()

Reads the decoded and formatted MSR data. If the device is not opened with the O_NONBLOCK flag set, this call is blocked until data is available.

Prototype

```
int msrRead(char *buffer, int size);
```

Parameters

`buffer`	Pointer to data area
`size`	Maximum number of bytes to read. Each invocation of `msrRead()` transfers data from a card reader scan into the buffer.

The format of the buffer returned is:

```
c1  s1  d1  c2  s2  d2  c3  s3  d3
```

where:

`c1`	a one-byte size of c1+s1+d1
`s1`	a one-byte status of reading track 1
`d1`	any data read in (might not exist)
`c2`	a one-byte size of c2+s2+d2
`s2`	a one-byte status of reading track 2
`d2`	any data read in (might not exist)
`c3`	a one-byte size of c3+s3+d3
`s3`	a one-byte status of reading track 3
`d3`	any data read in (might not exist)

The data includes the `Start Sentinel`, `End Sentinel`, and the `LRC` characters.

The status byte (s1,s2,s3) can have one of the following values:

0	valid data
1	no data
2	missing start sentinel or insufficient data
3	missing end sentinel or excessive data
4	missing BCC or BCC error
5	parity error
6	special error

NOTE The returned error status may not reflect the exact cause because the algorithm tries to decode data in both direction streams. An error in one direction stream may not produce the same error as in the other.

The decode algorithm searches the entire data stream for the start sentinel.

Return Values

> = 0	Total number of bytes read
< 0	Error
	-4 = EMF (if enabled using `msrReportExtendedErrors()`)

msrWrite()

Transfers data from an application buffer into the device driver’s buffer. The data is used for the next read operation.

Prototype

```
int msrWrite(char *buffer, int size);
```

Parameters

`buffer`	Pointer to data area.
`size`	Maximum number of bytes to read. Each invocation of `msrRead()` transfer data from a card reader scan into the buffer.

Return Values

= 0	No data written.
> 0	Number of bytes written.

NOTE

This call is placed to support test program development and debugging modes.

msrMagneticCardPresent()

Returns the current status of the MSR data. If all three tracks report no data, then a strong magnetic field may be present to set off an invalid MSR read.

Prototype

```
int result = msrMagneticCardPresent(void);
```

Example

```
void msrCallback(void)
{
    if (msrMagneticCardPresent() == 1)
    {
        /* valid data */
        msrRead(…);
        …
    }
    else if (msrMagneticCardPresent() == 2)
    {
        /* magnetic field present */
        …
    }
}
```

"Magnetic Field Present" is reported to let the application know that a stray electromagnetic field is interfering with the MSR data. It is up to the application to notify the users so they can take appropriate action.

Prior to svcpak-016-02, EMF is returned when all three tracks report no data for their status, a very common result of EMF-induced interrupts. For an application that calls the blocking `msrRead()` instead of callbacks, it can use the same criteria to determine if the terminal is in the presence of magnetic field induced interrupts. However, that there may be instances where EMF-induced data from the chip exceed the decodable data threshold, which then results in a different error such as missing start or end sentinel.

Svcpak-016-02 and later improves the detection of EMF and will report it on the msrRead() call, if enabled via the msrReportExtendedErrors(), making this function obsolete. In svcpak-016-02 and later, also allows the application to configure the number of times no data is reported on all tracks before deeming it as EMF.

Return Values

= 0	No data available.
= 1	Data available.
= 2	Magnetic field present.

msrRaw()

Allows an application to retrieve the raw magnetic stripe data and perform a custom decode.

NOTE This API is disabled if the HTdes module is installed.

Prototype

```
int msrRaw(MSR_DATA * msr);
```

Parameters

The `MSR_DATA` structure is as follows:

```
typedef struct {
                 unsigned char ucStatus; // status of track
                 unsigned char ucCount; // size in bytes of track data
                 char *cpData; // pointer to track data
                 } MSR_TRACK_DATA;

typedef struct {
                 MSR_TRACK_DATA stTrack1;
                 MSR_TRACK_DATA stTrack2;
                 MSR_TRACK_DATA stTrack3;
             } MSR_DATA;
```

Return Values

= 0	Data available.
-1	No data available.

msrStructured()

Allows an application to retrieve the decoded magnetic stripe data in a structure.

Prototype

```
int msrStructured(MSR_DATA * msr);
```

Parameters

The `MSR_DATA` structure is as follows:

```
typedef struct {
                unsigned char ucStatus; // status of track
                unsigned char ucCount; // size in bytes of track data
                char *cpData; // pointer to track data
             } MSR_TRACK_DATA;

typedef struct {
                MSR_TRACK_DATA stTrack1;
                MSR_TRACK_DATA stTrack2;
                MSR_TRACK_DATA stTrack3;
             } MSR_DATA;
```

Return Values

= 0	Data available.
-1	No data available.
-4	EMF detected (if enabled using `msrReportExtendedErrors()`).

msrEnableLicenseDecode()

Enables the decoding of California Drivers License and American Association of Motor Vehicle Administrators (AAMVA) Drivers License.

NOTE

By default, California Drivers Licenses are not decoded. This is for compatibility with existing terminals and tests.

Prototype

```
int msrEnableLicenseDecode(void);
```

Return Values

= 0	Always returns zero.

msrDisableLicenseDecode()

Disables the decoding of California Drivers License and American Association of Motor Vehicle Administrators (AAMVA) Drivers License.

NOTE By default, California Drivers Licenses are not decoded. This is for compatibility with existing terminals and tests.

Prototype

```
int msrDisableLicenseDecode(void);
```

Return Values

= 0	Always returns zero.

msrSetMinBytes()

Sets the minimum number of track data bytes each track must have before decoding. This value can also be changed by setting the `*MSRMINBYTES` environment variable. If all three tracks have less than the minimum number of bytes specified, it is deemed to be due to the presence of Electromagnetic Field. This minimum can be increased for noisy environments, but ensure that actual card swipes are not dismissed as being EMF induced.

Prototype

```
void msrSetMinBytes(int min_bytes);
```

Parameters

`min_bytes`	The minimum number of track data bytes each track must have before decoding. The default value is 3. The track is deemed to have no data if it has less than minimum bytes.

msrSetMaxNoDataCount()

Sets the number of consecutive times no data in all three tracks are seen before reporting as EMF induced. This value can also be changed by setting the `*MSRMAXNODATA` environment variable.

Prototype

```
void msrSetMaxNoDataCount(int max_count);
```

Parameters

`max_count`	The number of consecutive times no data in all three tracks are seen before reporting as EMF induced. The default value is 3.

msrReportExtendedErrors()

Enables the `msrRead()` and `msrStructured()` functions to return negative error returns for error such as when interference from an electromagnetic field is detected. This function is so that `msrRead()` is backwards compatible with older applications. Applications should provision for possible future function call error returns.

Prototype

```
void msrReportExtendedErrors(void);
```

msrVersion()

Reads the MSR library version.

Prototype

```
int msrVersion(char*version);
```

Parameters

version	Pointer to read in the MSR library version, in the form: `xx.yy.zz`

Return Values

= 0	Success
< 0	Error

Example Card Reader Code

This program reads the card reader device and prints the result to STDOUT.

```
         main()
         {     int   result;
               char  buffer[200];

               result = msrOpen( O_RDONLY, NULL );
               msrReportExtendedErrors();

               while(1)  /* infinite loop */
               {  memset(buffer, 0, sizeof(buffer));
                  result = msrRead(buffer,sizeof(buffer));
                        /* wait for input from card reader */
                  if (result > 0)
                        printf("msrRead returned %d bytes of
data\n", result);
                  else
                        printf("msrRead returned %d error\n",
result);
            }
         }
```

CHAPTER 19

Contactless RF Card Reader

The V/OS terminal supports the Contactless RF Card Reader (RFCR or CTLS). The module is embedded in V/OS terminal and uses sockets for inter-process communication using a simple command and response protocol.

The user space shared library `libvfirfcr` and the `RFCR_NGapi.h` file helps to interface common features of the module and are detailed in this chapter.

NOTE Applications can also use the low-level commands to communicate with the module. Refer to the hardware reference guide your terminal for command protocol, status, and error returns.

Contactless RFCR Functions

These sections discuss the calls that support the RFCR, the UI framework helper structures, CTLS application framework structures, and code examples.

NOTE See the programmers manual for your device for functions specific to your terminal.

CTLSClientGetVersion ()

Returns the CTLS Interface version.

Prototype

```
CTLSClientGetVersion(char* cVersion);
```

Parameters

`cVersion`	The version of the RFCR interface.

Return Values

CTLSInitInterface()

Starts a user interface child thread during initialization. This child thread creates and monitors IPC for contactless user interface requests.

Prototype

```
CTLSInitInterface(int   timeout);
```

Parameters

`timeout`	The timeout period while the child thread starts.

Return Values

CTLSOpen()

Establish ownership of the RFCR.

Prototype

```
CTLSOpen(void);
```

Return Values

CTLSGetUI()

(Optional) Retrieve the current UI handler function pointers used by the framework.

Prototype

```
CTLSGetUI(CTLSUIFuncs* uiFuncs);
```

Parameters

`uiFuncs`	Pointer to the UI handler functions.

Return Values

CTLSSetUI()

(Optional) Assign the handler functions called by the UI framework.

Prototype

```
CTLSSetUI(CTLSUIFuncs* uiFuncs);
```

Parameters

`uiFuncs`	Pointer to the UI handler functions.

Return Values

If CTLSSetUI () passed a NULL argument, the CTLS UI framework callback functions are set to their default values. If any function pointer in the callback table is NULL, the CTLS UI framework uses the default handler for the corresponding user interface function.

CTLSSend()

Send a message to the CTLS application. The message payload must be formatted according to the protocol supported by the CTLS application. Currently, the CTLS Application supports the VivoPay protocol.

Prototype

```
CTLSSend(char * buff, int length);
```

Parameters

`buff`	Pointer to the buffer where the message is stored.
`length`	Message payload length.

Return Values

CTLSReceive()

Receive a message from the CTLS application.

Prototype

```
CTLSReceive(char * buff, int length);
```

Parameters

`buff`	Pointer to the buffer where the message is stored.
`length`	Message payload length.

Return Values

CTLSClose()

Close the RFCR connection.

Prototype

```
CTLSClose();
```

Return Values

Optional UI Framework Helper Functions

The following are the optional UI framework helper functions:

- `CTLSClientStartUI (CTLSUIParms* pParms)` – This is the default CTLS UI Framework handler for starting the CTLS UI.
- `CTLSClientStopUI (void* tbd)` – This is the default CTLS UI framework handler for stopping the CTLS UI.
- `CTLSClientUIParamHandler(int paramId, void* paramVal, int paramLen)` – This is the default CTLS UI framework handler for parameter assignments.
- `CTLSClientBuzzerHandler(CTLSBuzzerRequst* parms)` – This is the default CTLS UI framework handler for buzzer requests.
- `CTLSClientLEDHandler(CTLSLEDStatus* leds)` – This is the default CTLS UI framework handler for LED requests.
- `CTLSClientLCDHandler(CTLSLCDText* pText)` – This is the default CTLS UI framework handler for displaying text on the LCD.
- `CTLSClientClearHandler(void* tbd)` – This is the default CTLS UI Framework handler for clearing the display.

CTLS Application Framework Structures

This section presents the CTLS framework structures.

CTLSUIFN : CTLS UI Framework Callback Table

```
typedef struct
{
   CTLSStartUIFuncstartUIHandler;
   CTLSStopUIFunc stopUIHandler;
   CTLSUIParamFunc uiParamHandler;
   CTLSLEDFunc ledHandler;
CTLSLCDTextFunc textHandler;
   CTLSLCDClearFunc clearDisplayHandler;
   CTLSBuzzerFunc buzzerHandler;
   void*       reserved1;
} CTLSUIFuncs;
```

CTLSUIParms Structure

```
typedef struct
 {
   int uiMode;// EMV or PassThrough style.
} CTLSUIParms;
```

`uiMode` valid values are:

- UI_EMV_STYLE
- UI_PASSTHRU_STYLE

CTLSLCDText Structure

```
typedef struct
{
   char stringToDisplay[64];
   int    backgroundColor;
   int    foregroundColor;
   int    fontId;
   int    format;// alignment
   int    column;
   int    row;
} CTLSLCDText
```

where:

- `stringToDisplay` – A NULL-terminated string to display.
- `backgroundColor` – RFU.
- `foregroundColor` – RFU.
- `fontId` – RFU (these IDs map to font files)
- `format` – A bit map defining the location to display the text. Valid values are
  - CTLS_SET_LEFT
  - CTLS_SET_RIGHT
  - CTLSI_SET_CENTER
  - CTLS_SET_HPIXEL
  - CTLS_SET_TP
  - CTLS_SET_BTTM
  - CTLS_SET_VCENTER
  - CTLS_SET_SPREAD
  - CTLS_SET_VPIXEL
- `column` – The column (x coordinate) in pixels.
- `row` –
  - 1/2 – display in message area above the tap icon.
  - 3/4 – display below the icon.
  - The constants LCD_LINE1_Y and LCD_LINE2_Y are defined with corresponding pixel locations.

CTLSBuzzerRequest Structure

```
typedef struct
{
int frequency;
int pause;
int duration;
int nTimes;
} CTLSBuzzerRequest;
```

where:

- `frequency` – The tone frequency (in Hz).
  - UI_BUZZER_FREQUENCY_OFF – Turns the buzzer off.
- `pause` – The silent time (in msec) between beeps.
- `duration` – The length of the beep (in msec).
  - INT_MAX – Turns the buzzer on.
- `nTimes` – The number of beeps.

CTLSLEDStatus Structure

```
typedef struct
 {
   int       nLED;
   unsigned char led[MAX_LED];
 } CTLSLEDStatus;
```

Where each byte in the LED array contains the setting for each LED (left to right). The LED settings are:

- CTLS_LED_OFF
- CTLS_LED_ON
- CTLS_LED_FLASH

Code Examples

This section presents CTLS routine examples.

Example 1

Implements communication with CTLS application:

```
// get  CTLS Interface version

 CTLSClientGetVersion (cVersion);
   printf ("%s\n",cVersion);

// Open communications with NexGen CTLS

CTLSInitInterface (0);
```

```
   // establish ownership of CTLS device

   If ((ctlsfd = CTLSOpen ())! = CTLSINTERFACE_OK)
   {
       dbprintf ("error \n");
   }

   // initialization of  UI functions

   CTLSSetUI (&CTLtest); //set UI functions or set default UI functions
(CTLSSetUI (NULL)).


   //example for frame
length =17;// frame's length

memmove (data,"ViVOtech2\x00\x29\x00\x00\x00\xDE\xA0", length);

   //Send the frame

CTLSSend (data, length); //sendVivo packet to CTLS application

   //receive packet from CTLS application
   Datalength=CTLSReceive (response_packet, PACKET_LENGTH);
// PACKET_LENGTH = max packet length
```

Example 2

Using SetUI():

```
{

      CTLSUIFuncs CTLtest;

   CTLtest. UiParamHandler = & new_UIParam; // new_UIParam =user
application

   CTLtest. StartUIHandler = &new_setUI; // new_setUI =user application

CTLtest. StopUIHandler = &new_stopUI; new_stopUI =user application

CTLtest.buzzerHandler = &new_buzzer; // new_buzzer =user application
function.

CTLtest. LedHandler = &new_leds; // new_leds =user application function.

   CTLtest.textHandler = NULL; //using  default
CTLSClientLCDTextHandler  function.
```

```
   CTLSSetUI (&CTLtest);

   }

// new_buzzer implementation at application

int  new_buzzer (CTLSBuzzerRequst * parms}
{
.
.
.
   }
```

CHAPTER 20

USB Device/Host

The V/OS supports up to three USB ports, 0–2. USB Port 0 is a USB OTG port (on the I/O board on MX terminals) that supports either host or device functionality on-the-go (OTG), depending on the connected device. Port 1 can either be a USB host or USB device, depending on the multi-port cable type (USB host or USB device port).

NOTE

This is not a dynamic switch. A system reboot is required. You can switch multi-port cables without powering off the unit.

On some V/OS-based terminals and depending on cable configuration, USB port 2 is the host.

USB Device

When acting as a USB device, V/OS terminals conform to the Linux USB On-the-Go (OTG) 2.0 framework (see http://www.usb.org/developers/docs/) with full API functionality to operate the USB OTG stack controlling USB port 0. The USB OTG SW is built into a kernel module (usb_otg.ko) loaded by default on system reboot.

The device supports high-speed (480 Mbps), full-speed (12 Mbps) and low-speed (1.5 Mbps) data transfer. The two devices that V/OS terminals emulate are:

Serial	This exposes a tty style serial line interface, usable with Minicom and similar tools. (There's no serial console support at this time.) Most Linux hosts can talk to this using the generic **usb-serial** driver. The latest versions of this driver implement the CDC ACM class, as might be implemented by cell phones or other modems. This driver works with the MS Windows `usbser.sys` driver, the Linux **cdc-acm** driver, and many other USB Host systems. The V/OS terminal SDK includes an `.inf` file and `usbser.sys` required by Windows 2000/XP to interface with the USB serial device. On the V/OS terminal, the application can open COM5 or /dev/ttygser device. DDL supports application download over serial USB. To enable the USB Serial driver on the V/OS terminal, use the System Mode Configure USB display or set the environment variable `*USBDEVICE=1`.

RNDIS When talking to MS Windows hosts, RNDIS is used. RNDIS is Microsoft's analogue of CDC Ethernet, with complex frame capsulation and its own internal RPC protocol. (Clicking on this link, http://www.microsoft.com/whdc/device/network/NDIS/usbrndis.mspx, may help you and a partial protocol specification is available. For example, some requests from Windows 2000 and XP are undocumented.) Use the `Documentation/usb/linux.inf` file (convert to DOS CRLF format) to install the driver. The driver is bundled in XP and the URL in `linux.inf` says where to get Microsoft's drivers for older Windows releases. For some `step-by-step instructions with WinXP screenshots`, see http://www.gumstix.org/tikiwiki/tiki-index.php?page=Windows_XP_usbn showing one way to use that "linux.inf" file. Do not forget to read the comments there, explaining how to shortcut past some needless complications in those instructions.

USB Host

V/OS terminals can support USB host functionality, and run specific drivers for different devices. V/OS terminals have tested the following devices:

- USB keyboard and mouse (HID Class)
- USB memory sticks and hard drives through the MSDOS-compatible FAT32/VFAT format

  Specific support for most USB devices requires that drivers be manually loaded into the system and manually configured. Support for USB memory and mass storage devices requires little or no manual intervention.
- USB hubs (expansion ports), either externally or bus powered.
- PC connections using a USB cable for file transfers.
- IBM AT keyboards and scanners using AT keyboard scan codes.

If a USB keyboard or a scanner is plugged into the V/OS USB host port, the necessary HID drivers to support the device automatically load. To open and read data from these devices, use `inputOpen()`, `inputRead()`, and `inputClose()`. USB HID device support is described below.

NOTE The USB host is only supported using specific V/OS cables. These cables are:

- Red cable P/N 23739-02
- Green cable P/N 23740-02

USB Mass Storage and Memory Devices

V/OS terminals have built-in automatic support for USB mass storage and memory devices. The unit supports a single memory device plugged into the USB host port on a V/OS terminal cable, or up to four memory devices at any one time plugged in a USB hub.

V/OS terminals do not support more than four USB memory/mass storage devices, but they can be plugged into the unit in any configuration of single or multiple hubs. V/OS terminals automatically detect any memory/mass storage device plugged in to the unit, and automatically mount the device on one of the four directories located in the `/mnt` directory. Directory names for the memory devices are:

- `/mnt/usbstor1`
- `/mnt/usbstor2`
- `/mnt/usbstor3`
- `/mnt/usbstor4`

NOTE The p*x* suffix is added for partitioned disks. For example, for a DOK system with two partitions:

- `/mnt/usbstor1p1`
- `/mnt/usbstor1p2`

These directories are used sequentially as devices are plugged in to the unit. When a device is removed, it is automatically unmounted from the directory mount point, and its data is no longer accessible. While the device is plugged in, any application can access the data. The application can determine if a memory/mass storage device was detected, mounted properly, and ready for access using `utilityExternalStorage`.

`utilityExternalStorage` expects a pointer to a `char` buffer that can hold at least 4 bytes. When called, it determines which mount points currently have a device plugged in, mounted on its directory, and available for access. It returns the pointer with a value of 1 for each mounted device, and 0 for each device that is not. All files located on the device can be accessed at that mount point. For example, if `utilityExternalStorage` returns the values: 0, 1, 0, 1. then:

`/mnt/usbstor1`	No USB memory/storage device plugged in and mounted.
`/mnt/usbstor2`	Device plugged in and files located on the memory device can be accessed.
`/mnt/usbstor3`	No USB memory/storage device plugged in and mounted.
`/mnt/usbstor4`	Device plugged in and files located on the memory device can be accessed.

These directories exist on the terminal regardless if a device is detected and mounted. The application cannot to write to a directory without a device present. If this is done, the files remain on the terminal instead of going to the memory device as intended. Only when the device is present as indicated in `utilityExternalStorage` that a file can be written to a memory device correctly (that is, the file remains on the memory device and in not terminal memory).

NOTE USB device detection is performed by plug-and-play hardware and software. Typically it takes ~10 seconds for all devices to be detected, enumerated, and initialized before they are available for use.
Applications and users should be aware of this delay before devices can be accessed after plug in or removed from the V/OS terminal.

The USB host is reset by removing all plugged in devices, and waiting for ~10–15 seconds. All entries, environment variables, and mounted devices are removed and cleared. The USB host port can then accept newly inserted devices.

USB Human Interface Device (HID) Support

V/OS terminals also have built-in USB HID support for some devices through the Linux kernel Input Event module. When a HID is plugged in to the USB Host port, it is automatically detected and the appropriate USB HID and event drivers load. The device can be opened, read/written, and closed from an application. Currently, the V/OS terminal either partially or fully supports HIDs:

USB Host Keyboard	Enables an event interface to capture keys pressed on a USB host keyboard. V/OS terminals only support IBM AT keyboards, with `scancode` set 1. Other `scancode` sets (2 and 3) are not supported. There is full support for this device. It can be opened, read, and closed using V/OS terminal Input Library APIs. This library consists `inputOpen()`, `inputRead()`, and `inputClose()`.
USB Host Scanner	Enables an event interface to capture scanned data from a USB handheld scanner. The scanner must utilize the IBM AT keyboard `scancode` set 1. There is full support for this device. It can be opened, read, and closed using V/OS Input Library APIs. This library consists `inputOpen()`, `inputRead()`, and `inputClose()`.
USB Host Mouse	Enables an event interface the application uses to capture events from a USB mouse. There is support to open and close the device, but is not fully supported at this time. Further implementation of a complete API to read events from the mouse may be done some time in the future.

CHAPTER 21

NET Service Ethernet and Wi-Fi Configuration

Some V/OS-based terminals use NET Service to configure IPv4 Ethernet and Wi-Fi communications device (terminal specific) and support various protocols (IP, PPP, SSL, and so on). NET Service uses an XML file to store the Ethernet and the Wi-Fi parameters.

NOTE

NET Service supports both IPv4 and IPv6, however, in V/OS only v4 is currently supported. Parameters related to IPv6 will be ignored.

This chapter describes creating and modifying the XML configuration file, enabling the Ethernet and Wi-Fi interfaces, and the NET application function calls.

NOTE

For UX terminal-specific PPP services, see Appendix C. For VX terminal-specific PPP services, see Appendix F.

Management Communication Sessions for Applications

NET Service establishes communication sessions for applications. The following are supported session types:

- TCP/IP (or SSL) over Ethernet
- TCP/IP (or SSL) over Wi-Fi
- TCP/IP (or SSL) over PPP over Modem (PSTN, ISDN)
- TCP/IP (or SSL) over PPP over Serial Port
- Modem direct calls

The application uses a configuration file (XML format) to perform communication sessions through the NET Service. The configuration file uses communication *tags* to describe the session type such as SSL over PPP over modem. It also stores the corresponding parameters such as SSL certificates, PPP user name and password, the phone number, and so on. Figure 9 shows how NET Services facilitate application communications.

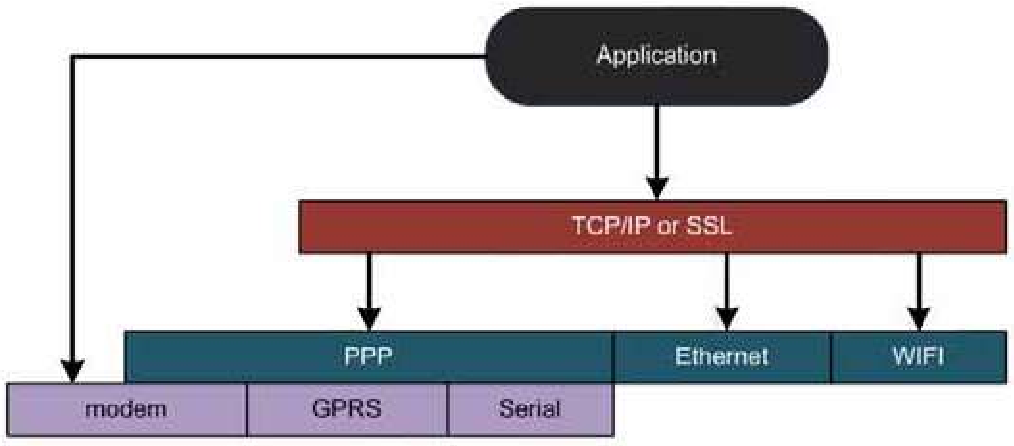


Figure 9 NET Service Communication Sessions

The application uses a NET Service API to load its session file to perform a connection. It then calls another NET Service API to start the session and retrieves an output value session ID (handle) that identifies the current session. The application uses this handle for all subsequent calls (read/write/disconnect).

Establishing Communication Sessions

An application must provide a session file in XML format to perform a communication session using the NET Service. Each supported communication type is described in this session and how to build them from the application context.

Figure 10 summarizes the main steps for an application to establish a communication session using the NET Service.

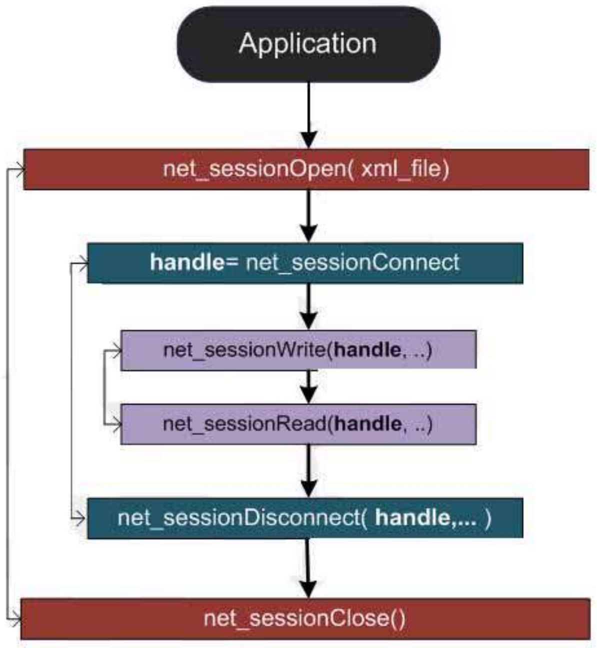

Application
net_sessionOpen(xml_file)
handle= net_sessionConnect
net_sessionWrite(handle, ..)
net_sessionRead(handle, ..)
net_sessionDisconnect(handle,...)
net_sessionClose()

Figure 10 NET Service Communication Session Flow

1 Use net_sessionOpen() to load the session file.

```
int net_sessionOpen( char *p_xmlFile );
```

This function takes as parameter the configuration file in XML format, and returns 0 on success or -1 on error.

2 Use net_sessionConnect() to start the communication session.

```
int handle = net_sessionConnect( int sessionType );
```

A session handle (>0) returns on success.

You can define more than one communication session (for example, SSL over Wi-Fi and SSL over Ethernet) in the same XML configuration file. The `sessionType` parameter selects which session to use first.

NOTE `handle`–the value returned by `net_sessoionConnect (>0))`–uniquely identifies the current session. The application uses `handle` to exchange data with the remote server.

3 Use the net_sessionWrite() to send data to the remote server.

```
int net_sessionWrite(int handle, struct netDataTransfer data );
```

where,

- `handle` = the value returned by net_sessionConnect()

- `data` = the data to send to the peer. The structure definition is:

```
struct netDataTransfer
{
   void *data;/**< array of bytes to transfer */
   int data_count; /**< number of bytes to transfer */
};
```

4 Use the net_sessionRead() to read data from the remote server.

```
struct netDataTransfer net_sessionRead(int handle, int count, int
timeout);
```

where,

- `handle` = the value returned by net_sessionConnect()
- `count` = the number of bytes to read
- `timeout` = the number of seconds to wait for data

A `netDataTransfer` structure returns. The structure definition is:

```
struct netDataTransfer
{
   void *data;/**< array of bytes to transfer */
   int data_count; /**< number of bytes to transfer */
};
```

NOTE On success (errno=0), the application must free the data in the `netDataTransfer` structure.

5 Use net_sessionDisconnect() to terminate the current communication session.

```
int net_sessionDisconnect(int handle );
```

where,

- `handle` = the value returned by net_sessionConnect()

The handle value is passed as a parameter to indicate which session to close.

NOTE Once net_sessionDisconnect() is called, the handle becomes invalid. To obtain a new handle, the application can start a new session by calling net_sessionConnect().

6 Use net_sessionClose() to unload the XML configuration file.

```
int net_sessionClose( void );
```

This call releases the context. All handles are no longer valid.

NOTE Once net_sessionDisconnect() is called, the handle becomes invalid. To obtain a new handle, the application can start a new session by calling net_sessionConnect().

XML Communication Files

The following is the general structure of a NET Service communication files:

```
<?XML VERSION="1.0" ENCODING="UTF-8" STANDALONE="YES" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="session name" next_media="tag1 name" />
</SESSION_LIST>

<TAG1_LIST>
<TAG1 name_id="tag1 name" param1a="value" …param1n="value"
next_media="tag2 name"/>
</TAG1_LIST>

<TAG2_LIST>
<TAG2 name_id="tag2 name" param2a="value" …param2n="value" />
</TAG2_LIST>

</SETTINGS>
```

Every NET Service XML communication file begins with a SESSION tag. This is mandatory. Each SESSION tag entry is uniquely identified by the `name_id` parameter within the entire XML file (here session name). The `next_media` parameters (`tag1 name`) links the SESSION tag to another tag in the XML file (TAG1).

TAG1 indicates what the purpose of this session (for example, TCP/IP, SSL, modem, and so on). TAG1 parameters (`param1a` … `param1n`) Convoy data for TAG1 (for example, the IP address for a TCP/IP session or SSL certificates for SSL).

TAG1 is uniquely identified by its `name_id` (`tag1 name`) within the entire XML file. `next_media` refers to another tag (TAG2) in the XML file may be required to fully work. For example, if TAG1 is TCP/IP, TAG2 can store parameters for Ethernet, Wi-Fi, or PPP. Some tags such as SESSION require a `next_media` parameter, and some don't such as Ethernet, Wi-Fi.

Every NET Service communication file follows this structure.

NET Service Communication Tags

This section lists all supported communication tags and corresponding attributes. NET Service tags are grouped into five categories:

- Session tags define an entry point for an XML file.

TAG Name	Description
SESSION	Entry point for a communication file.

- Protocols tags define SSL and TCP/IP layers in the configuration file.

TAG Name	Description
SOCKPROTO	TCP/IP socket parameters.
SSL	SSL parameters.

- IP Interfaces tags define the IP interfaces used by the protocol tags.

TAG Name	Description
ETHLINK	Ethernet parameters.
WIFI	Wi-Fi parameters.
PPPPROTO	PPP parameters.

- Modem tags define tags for modems and the GPRS layer. Modem tags can link to a PPP layer or be used directly by a SESSION entry.

TAG Name	Description
MODLINK	PSTN modem parameters.
MODISDNLINK	ISDN modem parameters.
GPRS	GPRS parameters.

- Option tags define tags for Option.

TAG Name	Description
OPTION	Option parameters.

NOTE

A tag is defined by a list of attributes and their values. Not all attributes are required. Mandatory attributes are written in bold in the following tables.

Tag name: SESSION

Use this tag to configure a communication session.

Attribute Name	Type	Accepted Values	Description
name_id	***char***[32]	media name	Identification name of the media
Priority	int	1 …	Priority of the session
Timeout	double	1 …	Connection timeout (RFU)
Retry	int	0 …	Number of connection tries (RFU)
next_media	***char***[32]	Next media name	Identifies the protocol used by the session.

Tag name: SOCKPROTO (TCP/IP)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identification name of the media.
protocol	char[8]	tcp	Socket type (for example, one possible value is TCP).
remote_ip	char[128]	IP:PORT or DNS:PORT	Server address and port number.
next_media	char[32]	Next media name	Identifies the physical layer used by this protocol such as Wi-Fi, ETHERNET, or PPP.

Tag name: SSL

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the SSL layer.
remote_ip	char[128]	IP:PORT or DNS:PORT	Socket protocol.
server_profile	char[128]	Full path + file name	Identifies the place where the server CA is stored.
client_profile	char[128]	Full path + file name	Identifies the place where the client certificate and the client private key are stored.
session_ttl	int	0…	Indicates in seconds how long to keep the Resume Session before performing a complete handshake.
next_media	char[32]	Next media name	Identifies the linked media.

Tag name: ETHLINK (Ethernet)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
usedhcp	int	0, 1	1=DHCP is used 0 = DHCP not used
local_ip	char[16]	ip range	Identifies the network adapter IP address.
netmask	char[16]	ip range	Network mask.
broadcast	char[16]	ip range	Broadcast.
gateway	uint8[16]	ip range	Gateway IP address.
Activate	int	0,1	Indicates if this interface must be activated.
Dhcpid	char[16]		The client id string for DHCP.
Clienthostname	char[128]		The client hostname string for DHCP.
Speed	char[16]	(0, Auto), (1,100F), (2, 10F), (3,100H), (4,10H)	Defines the link speed as 10F, 100F, 10T, 100T, or AUTO (default is AUTO).
dns1	char[16]	ip range	Domain Name Server number 1.
dns2	char[16]	ip range	Domain Name Server number 2.

Tag name: WIFI

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
usedhcp	int	0, 1	1=DHCP is used 0 = DHCP not used
local_ip	char[16]	ip range	Identifies the network adapter IP address
netmask	char[16]	ip range	Network mask.
broadcast	char[16]	ip range	Broadcast.
gateway	uint8[16]	ip range	Gateway IP address.
activate	int	0,1	Indicates if this interface must be activated.
suppliconf	char[16]		Wi-Fi configuration security file location (standard format).
dns1	char[16]	ip range	Domain Name Server number 1.
dns2	char[16]	ip range	Domain Name Server number 2.

Tag name: PPPPROTO (PPP)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies name of the media.
user	char[32]		User name for PPP.
password	char[32]		Password for PPP.
auth	char[32]	auto, chap, pap	Authentication mode for PPP.
pppOption	char[16]		RFU.
next_media	char[16]	ip range	Indicates the media used to connect the PPP layer

Tag name: MODLINK (PSTN Modems)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
number	char[32]		The phone number to dial.
dial_type	char	1=Tone, 2=Pulse	Identifies the dial tone type.
mode	char	1=Sync, 2=Async	Identifies the mode as asynchronous or synchronous.
max_retries	char	0…	Identifies the number of dial retries.

Attribute Name	Type	Accepted Values	Description
max_line_rate	char	(1=**300**), (2=**300_B**), (3=**1200**), (4=**1200_FAST**), (6=**1200_B**), (7=**2400**), (8=**4800**), (12=**14400**), (14=**33600**), (15=**56000**),	Identifies the modem connection rate.
connect_timeout	char		Identifies the whether the modem is waiting for an answer. If expired, there are no more retries.
error_correction	char	1 = Activate correction error 0 = otherwise	Indicates whether to activate the modem connection error.
compression	char	1	error_correction must be enabled to use compression.

Tag name: MODISDNLINK (Modem ISDN)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
number	char[32]		The phone number to dial
isdn_prot	char	(3, HDLC_SYNC_TO_ASYNC), (4, HDLC_TRANSPARENT), (10, X75), (20, X31_B_CHL)	ISDN transmission protocols.
x25_packet_size	char[5]	64 - 2048	Identifies the mode as asynchronous or synchronous.
facilities	char[20]	starting with 'G'	Access to X.25 Closed User Group.
nui	char[20]	a-z, A-Z, 0-9	NUI (Network User Identification) and password with call setup.
x25_number	char[20]		X.25 call number, (X.25 B channel only).

Attribute Name	Type	Accepted Values	Description
user_data	char[20]		User data starting with D (without protocol ID, data length max. 16 char). P with protocol ID, data length max. 12 characters.
max_retries	char	0…	The maximum number of dial retries.
connect_timeout	char		Identifies the whether the modem is waiting for an answer. If expired, there are no more retries.
country_code	char		Identifies the country code.

Tag name: GPRS

NOTE

GPRS is not supported in UX terminals.

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
APN	char[32]		GPRS access point name.
operator_id	char		Operator LAI number.
reg_mode	char[5]	1 = AUTO 2 = MANUAL 3 = AUTO_MANUAL	Network registration mode.
number	char[32]		Phone number for GSM data calls (RFU).
bearer	char[32]		GSM Data modulation (RFU).

Tag name: OPTION

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	Media name of another tag in the XML file for which this option applies	Identifies the name of the media.
option	char[256]		List of options.

NET Service Communication File Examples

This section provides XML example files for NET Service communication structures.

TCP/IP Over Ethernet

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="eth-session" priority="1" timeout="10000" retry = "3"
next_media="tcp-layer" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id="tcp-layer" protocol="tcp"
remote_ip="www.verifone.com:80" next_media="eth-layer" />
</SOCKPROTO_LIST>
<ETHLINK_LIST>
<ETHLINK name_id="eth-layer" interface="eth0" usedhcp="1"/>
</ETHLINK_LIST>
 </SETTINGS>
```

TCP/IP Over Wi-Fi

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="wifi-session" priority="1" next_media="tcp-layer"
timeout="10000" retry = "3"/>
</SESSION_LIST>
 <SOCKPROTO_LIST>
SOCKPROTO name_id="tcp-layer" protocol="tcp"
remote_ip="www.google.com:80" next_media="wifi-iface" />
</SOCKPROTO_LIST>
 <WIFI_LIST>
<WIFI name_id="wifi-iface" interface="eth0" usedhcp="yes" suppliconf="/
mnt/flash/etc/config/svcnet/wificfg.conf"/>
</WIFI_LIST>
 </SETTINGS>
```

SSL Over Ethernet

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
```

```
<SESSION_LIST>
<SESSION name_id="ssl-session" priority="1" next_media="ssl-eth"
timeout="10000" retry = "3"/>
</SESSION_LIST>
 <SSL_LIST>
<SSL name_id="ssl-eth" remote_ip="mmsgto.fr:443"  server_profile="/tmp/"
client_profile="/tmp/" next_media="eth-iface" />
</SSL_LIST>
 <ETHLINK_LIST>
<ETHLINK name_id="eth-iface" interface="eth0" usedhcp="1"/>
</ETHLINK_LIST>
 </SETTINGS>
```

SSL is the tag for an SSL entry used with the following attributes:

- remote_ip: defines the server IP address to connect to.
- server_profile: specifies the path that contains the server CA root.
- client_profile: specifies the path that contains the client CA root and the Private key.

NOTE

- For server_profile, the server CA root name must be CA_Key.pem. For example:

  ```
  /tmp/CA_Key.pem
  ```
- For client_profile, the client CA root name must be CC_Key.pem and the Private Key name is CP_Key.pem. For example:

  ```
  /tmp/CA_Key.pem
  /tmp/CP_Key.pem
  ```
- client_profile must be provided only when SSL mutual authentication is required.

Private Key Protection

When the private key is password protected, the application must use net_sessionStorePassword() to provide its plain text value to NET Service. This API can be called at any time.

```
int net_sessionStorePassword(char *profile, char *password, int gshared);
```

where,

- profile = client_profile
- password = the passphrase that protects the private key in plain text
- gshared is 1 if the passphrase is shared within the group, or 0 if it is private to the application

Overriding the Default PCI - Cipher List

In some cases the application may need to use its own cipher list to connect to a specific server. The application uses the net_sessionCipherList() API to indicate to the NET Service layer which cipher suite applies for a specific profile. This API can be called at any time.

```
o  int net_sessionCipherList(char *profile, char *cipher, int gshared);
```

where,

- profile = server_profile
- cipher = the cipher suite used for a given server profile.
- gshared = 1 if the cipher suite is shared within the group, or 0 if it is private to the application.

Default PCI - Cipher List

An SSL/PCI configuration file restricts the SSL ciphers to only those that are PCI compatible. It allows defining the cipher suite used for SSL sessions if the application is not providing one. The file is located at `/mnt/flash/etc/config/svcnet/cipher_pci.xml`

This file is loaded only when the application is not providing its own cipher suite.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<CIPHER_SUITE_LIST>
<CIPHER_SUITE cipher_id="1" cipher="pciciphersuite" />
</CIPHER_SUITE_LIST>
</SETTINGS>
```

where,

- pciciphersuite = the PCI cipher suite string.
- cipher_id="1" This is reserved for future use, and is always 1.

NOTE

- If the cipher_pci.xml file is not present and the application does not provide one, the default OpenSSL cipher list applies.
- Applications cannot change the PCI cipher suite.

TCP/IP over PPP over PSTN Modem

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-modem" protocol="TCP"
remote_ip="www.verifone.com:80" />
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
```

```
<PPPPROTO name_id=" ppp-modem" user="magic" password="isdn" auth="pap"
pppOption="" next_media="modem_uart"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<MODLINK name_id="modem_uart" number="0820903002" dial_type="1" mode="2"
max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1"/>
</MODLINK_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-modem" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
<OPTION name_id="modem_uart" option="dial_no_wait no prefix 8"/>
</OPTION_LIST>
</SETTINGS>
```

TCP/IP over PPP over ISDN Modem

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-modem" protocol="TCP"
remote_ip="www.verifone.com:80" />
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
<PPPPROTO name_id=" ppp-modem" user="magic" password="isdn" auth="pap"
pppOption="" next_media="modem_uxsidn"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<MODLINK name_id="modem_uxisdn" number="0820903002" dial_type="1"
mode="2" max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1" isdn_prot="3"/>
</MODLINK_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-modem" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
<OPTION name_id="modem_uxisdn" option="file /tmp/usrcmdfile_pstn1.txt"/>
</OPTION_LIST>
</SETTINGS>
```

TCP/IP over PPP over GPRS

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-gprs" protocol="TCP"
remote_ip="www.verifone.com:80" />
```

```
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
<PPPPROTO name_id=" ppp-gprs" user="magic" password="isdn" auth="pap"
pppOption="" next_media="gprs_layer"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<GPRS_LIST>
<GPRS name_id="gprs_layer" APN="m2minternet"  />
</GPRS_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-gprs" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
</OPTION_LIST>
</SETTINGS>
```

Creating a Communication File Dynamically

Applications may need to create an XML communication file. This section describes how to do that using the NET Service API.

Each communication type (TCP/IP, SSL, Wi-Fi) has a corresponding descriptor structure defined in svc_net.h. This structure writes data in the XML communication file or reads from it. NET Service associates each communication family (Socket, SSL, Wi-Fi) with a fixed communication type value, COM Type, which are also defined in svc_net.h. Descriptors and COM types create and modify the XML files.

Table 8 lists all NET Service communication descriptors and corresponding COM types.

Table 8 NET Service COM Types

Name	Tag	COM Type	Description
sessionDescriptor	SESSION	NETCOM_SESSION	Parameters for SESSION tags.
socketDescriptor	SOCKPROTO	NETCOM_SOCKPROTO	Parameters for socket tags.
sslDescriptor	SSL	NETCOM_SSL	Parameters for SSL tags.
ethernetDescriptor	ETHLINK	NETCOM_ETHLINK	Parameters for Ethernet tags.
wifiDescriptor	WIFI	NETCOM_WIFI	Parameters for Wi-Fi tags.
pppDescriptor	PPPPROTO	NETCOM_PPPPROTO	Parameters for PPP tags.
modemDescriptor	MODLINK	NETCOM_MODLINK	Parameters for PSTN modem tags.
modemIsdnDescriptor	MODISDNLINK	NETCOM_MODISDNLINK	Parameters for ISDN modem tags.
gprsDescriptor	GPRS	NETCOM_GPRS	Parameters for GPRS tags.
gprsDescriptor	OPTION	NETCOM_OPTION	Parameters for Option tags.

- net_sessionSetTag() writes a descriptor to the XML file.

```
int net_sessionSetTag(int comtype, char *nameid, struct netDataTransfer
data);
```

- net_sessionGetTag() reads a descriptor from the XML file.

```
struct netDataTransfer net_sessionGetTag( int comtype, char *nameid );
```

Writing a Communication Descriptor to an XML File

Use the steps in this section to add communication tags to a NET Service communication file. The following example adds a TCP/IP layer. The socketDescriptor structure and NETCOM_SOCKPROTO are required to add the TCP/IP layer. Adding any other tags follows the same path, but replaces socketDescriptor and NETCOM_SOCKPROTO with the desired quantities.

1 Create the data transfer structures:

```
struct socketDescriptor sckDesc;
/** Create a socket descriptor structure */
struct netDataTransfer sckData;
/** Create a general data transfer structure */
memset( &sckDesc, 0, sizeof(struct socketDescriptor));
memset( &sckData, 0, sizeof(struct netDataTransfer));
```

2 Open the file to store the data:

```
int ret = net_sessionOpen("/tmp/com_file.xml");
```

If the path /tmp/com_file.xml does not exist, it is created at the end of the process.

3 Fill the socket descriptor parameters:

```
strncpy( sckDesc.name, "socket", strlen("socket") );
/** value for name_id*/
strncpy( sckDesc.protocol, "tcp", strlen("tcp") );
strncpy( sckDesc.remote_ip, "www.verifone.com:80",
strlen("www.verifone.com:80") );
strncpy( sckDesc. nextMedia, "eth-layer", strlen("eth-layer") );
/* value for next_media */
```

4 Copy the socket descriptor structure in the `netDataTransfer` structure:

```
sckData.data = &sckDesc;
sckDesc.data_count = sizeof(struct socketDescriptor );
```

5 Use net_sessionSetTag() to add `netDataTransfer struct (sckData)` in the context:

```
int net_sessionSetTag(NETCOM_SOCKPROTO, "socket", sckData );
```

The data is stored in the context, but is not saved in the destination file until the next step.

6 Save the file.

```
int ret = net_sessionSave( "/tmp/com_file.xml" );
```

You can use a different filename. The existing file is not modified.

The result /tmp/com_file.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
 <SOCKPROTO_LIST>
SOCKPROTO name_id="socket" protocol="tcp" remote_ip="www.verifone.com:80"
next_media="eth-layer" />
```

```
</SOCKPROTO_LIST>
 </SETTINGS>
```

Reading a Communication Descriptor from an XML File

Use the steps in this section to read communication tags from a NET Service communication file. We will read a TCP/IP layer as an example. The socketDescriptor structure and NETCOM_SOCKPROTO COM type are required to read a TCP/IP layer. All other tags follow the same path, but replace socketDescriptor and NETCOM_SOCKPROTO with the desired quantities.

Use the following steps to read a communication descriptor from an XML file:

1 Create the data transfer structures:

```
struct socketDescriptor sckDesc;
/** Create a socket descriptor structure */
struct netDataTransfer sckData;
/** Create a general data transfer structure */
memset( &sckDesc, 0, sizeof(struct socketDescriptor));
memset( &sckData, 0, sizeof(struct netDataTransfer));
```

2 Open the file from which to read the data:

```
int ret = net_sessionOpen("/tmp/com_file.xml");
```

NOTE

/tmp/com_file.xml must be an existing file.

3 Fill the socket descriptor parameters:

```
sckData = net_sessionGetTag( NETCOM_SOCKPROTO, "socket" ); /* indicates
the name_id you want to read */
if ( errno == 0)
{/* In case of success errno is 0 */

   If (sckData.data_count == sizeof(struct socketDescriptor) )
   {
      memcpy( &sckDesc, sckData.data, sckData.data_count);
   }
   free(sckData.data ); /* The caller must free the data*/
}
```

NOTE

If errno is 0 (success), the caller must free the `netDataTransfer` structure.

Ethernet and Wi-Fi Configuration

Ethernet or Wi-Fi configuration is performed in two steps:

1 Create the XML configuration file.

The Net Service uses an XML to store Ethernet and Wi-Fi parameters, and provide calls to create or to modify this file.

2 Enable the IP interface.

Net Service provides calls to enable the Ethernet and Wi-Fi interfaces as set by the configuration file parameters.

Create and Modify the Default XML Configuration File

The Ethernet and Wi-Fi parameters are stored in `netconf.xml`, the default XML configuration file located at: `/mnt/flash/etc/config/svcnet/netconf.xml`.

Use the `netIfConfig` structure to Convoy parameters for both Ethernet and Wi-Fi. Its definition is shown in NET Service Structures. The next section shows how to use this structure to read or to write data to or from the `netconf.xml`.

Write Ethernet Parameters to netconf.xml

First, create an instance of the `netIfconfig` structure that contains the Ethernet parameters, and then use the `net_interfaceSet` function to store Ethernet configuration in `netconf.xml`.

To create the `netIfConfig` structure for the Ethernet parameters:

```
struct netIfconfig config;
memset( &config, 0, sizeof(struct netIfconfig) );
```

Set the `config.interface` parameter to `"eth0"`, to indicate that the data is for Ethernet:

```
strcpy( config.interface, "eth0");
strcpy( config.local_ip, "192.168.18.2");
strcpy( config.broadcast, "192.168.45.15");
strcpy( config.netmask, "255.255.255.0");
strcpy( config.gateway, "192.168.18.1");
strcpy( config.dns1, "10.35.07.14");
strcpy( config.dns2, "10.35.13.16");
config.usedhcp = 0;
config.activate = 1;
```

Parameters

`config.interface`	IP interface name.
`config.local_ip`	Local IP address.
`config.broadcast`	Broadcast IP address.
`config.gateway`	Address of the gateway.
`config.dns1`	Domain name server 1 IP address.
`config.dns2`	Domain name server 2 IP address.
`config.usedhcp`	0 = Use static IP address. 1 = Start DHCP client.
`config.activate`	0= The Ethernet interface is never brought up. 1= Enable the Ethernet interface.

Write netIfConfig Structure to netconf.xml file using net_interfaceSet()

Once the `netIfconfig` structure Ethernet parameters are set, use net_interfaceSet() to write data to the `netconf.xml` file:

```
int ret;
ret = net_interfaceSet( config );//
```

The resultant `netconf.xml` file contains:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<ETHLINK_LIST>
<ETHLINK name_id="eth0" interface="eth0" local_ip ="192.168.18.2"
         broadcast = "192.168.45.15" netmask ="255.255.255.0"
         gateway="192.168.18.1" dns1 ="10.35.07.14" dns2="10.35.07.16"
         usedhcp="0" activate="1"/>
</ETHLINK_LIST>
</SETTINGS>
```

Read Ethernet Parameters from netconf.xml

The `netIfconfig` structure returns from the function net_interfaceGet() to read the Ethernet parameters from `netconf.xml`. Use the `eth0` parameter to indicate that the Ethernet parameters are being read:

```
config = net_interfaceGet( "eth0" );
```

On success (errno=0), `config` contains the Ethernet parameters stored in `netconf.xml`.

Write Wi-Fi Parameters to netconf.xml

First, create an instance of the `netIfconfig` structure to contain the Wi-Fi parameters, and then use the `net_interfaceSet()` call to store the Wi-Fi configuration in `netconf.xml`.

To create the `netIfConfig` structure for the Wi-Fi parameters:

```
struct netIfconfig config;
memset( &config, 0, sizeof(struct netIfconfig) );
```

Set the interface parameter to "`wlan0`" to indicate that the data is for Wi-Fi:

```
strcpy( config.interface, "wlan0");
strcpy( config.suppliconf, "/mnt/flash/etc/config/svcnet/wificfg.conf" );
config.usedhcp = 1;
config.activate = 1;
```

Parameters

`config.interface`	Wi-Fi interface name (`wlan0`).
`config.suppliconf`	Pathname for the `wificfg.conf` file.
`config.usedhcp`	0 = Use static IP address. 1 = Use DHCP configuration.
`config.activate`	0= Wi-Fi interface never brought up. 1= Enable the Wi-Fi interface.

NOTE `/mnt/flash/etc/config/svcnet/wificfg.conf` is the WPA Supplicant configuration file. It is named and loaded by the user. The user package must have the following package control file:

```
Package: wpa-config-file
Version: 1.0.0
Type: flashconfig
User: usr1
Group: system
Unmask:
```

Write netIfConfig structure to netconf.xml file using net_interfaceSet()

Once the `netIfconfig` structure Wi-Fi parameters are set, use net_interfaceSet() to write data to `netconf.xml`:

```
int ret;
ret = net_interfaceSet( config );
```

The resultant `netconf.xml` file contains:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<ETHLINK_LIST>
<ETHLINK name_id=" eth0" interface="eth0"
         local_ip = "192.168.18.2" broadcast = "192.168.45.15"
         netmask = "255.255.255.0" gateway = "192.168.18.1"
         dns1 = "10.35.07.14" dns2= "10.35.07.16" usedhcp="0"
         activate="1"/>
</ETHLINK_LIST>
<WIFI _LIST>
```

```
<WIFI name_id="wlan0" interface="wlan0" usedhcp="1" activate="1"
suppliconf="/mnt/flash/etc/config/svcnet/wificfg.conf"/>
</WIFI_LIST>
</SETTINGS>
```

Read Wi-Fi Parameters from netconf.xml

The `netIfconfig` structure returns from the function net_interfaceGet() to read the Wi-Fi parameters from `netconf.xml`. Use the `wlan0` parameter to indicate that the Wi-Fi parameters are being read:

```
config = net_interfaceGet( "wlan0" );
```

On success (errno=0), `config` contains the Wi-Fi parameters stored in `netconf.xml`.

Bring Up a Specific Network Interface

Use `net_interfaceUp()` to bring up the Ethernet or the Wi-Fi interface. `net_interfaceUp()` reads the configuration parameters from `netconf.xml`.

Configure the Ethernet Interface

There are two methods for configuring the Ethernet interface:

- Automatic

  When an Ethernet cable is plugged in or during the boot process, the V/OS terminal reads the Ethernet parameters from `netconf.xml` and brings up the network (`activate` = 1 to enable the network).

  - If `activate` = 0, Ethernet is not configured.
  - If `usedhcp` = 1, the DHCP client starts; otherwise use static IP.
  - If the Ethernet cable is unplugged, the Ethernet interface is disabled.

- Manual

  Net Service provides calls to manually enable/disable the Ethernet interface.

  - Use net_interfaceUp() to bring up the Ethernet interface:

    ```
    net_interfaceUp( "eth0", NET_IP_V4 );
    ```

  When DHCP is enabled, net_interfaceUp() exits even if a new IP address is not obtained. The system will continue to try to bring up the Ethernet interface.

  - Use net_interfaceDown() to bring up the Wi-Fi interface:

    ```
    net_interfaceDown( "eth0",NET_IP_V4 );
    ```

NOTE

The `eth0` parameter indicates only that the Ethernet interface is down (disabled).

Configure the Wi-Fi Interface

To configure the Wi-Fi interface:

- Configure `netconf.xml`

  The `netconf.xml suppliconf` attribute must contain a valid Wi-Fi configuration file.

- Load the Wi-Fi packages

  The `wlan` and `dbus` packages are required for Wi-Fi. These are provided as extra packages in V/OS delivery, and must be loaded into the terminal separately. They must also be properly signed.

- Start the WPA supplicant

  Use net_wifiStart() to start the WPA supplicant daemon.

  - If the `net_wifiStart() startwpa` parameter is 1, the Wi-Fi module powers up and the WPA supplicant daemon starts.
  - If `startwpa` = 0, only the Wi-Fi module powers up. For example, so the user can scan for Wi-Fi networks.

  It is not possible to get an IP address if the WPA supplicant has not started.

- Get an IP address on the Wi-Fi interface (wlan0)

  - Automatic configuration

  V/OS terminals automatically configure the Wi-Fi IP interface as set in `netconf.xml` each time the terminal registers to a Wi-Fi network.

    - If `usedhcp` is 1, the DHCP client starts
    - If `usedhcp` = 0, static IP is used for the Wi-Fi interface.
    - If the Wi-Fi connectivity is lost, the Wi-Fi interface is disabled.

  - Manual configuration

    - Use net_interfaceUp() to bring up the Wi-Fi interface. "`wlan0`" indicates that only the Wi-Fi interface is configured.

      ```
      net_interfaceUp( "wlan0", NET_IP_V4 );
      ```

    - Use net_interfaceDown() to bring down the Wi-Fi interface. "`wlan0`" indicates that only the Wi-Fi interface down.

      ```
      net_interfaceDown( "wlan0",NET_IP_V4 );
      ```

NOTE

`net_interfaceDown()` releases the IP address. The user can call `net_interfaceUp()` to get a new IP address.

When DHCP is enabled, net_interfaceUp() exits even if a new IP address is not obtained. The system will continue to try to bring up the Ethernet interface.

  - Use net_wifiStop() to disconnect the terminal from the Wi-Fi network.

    - If the `stopmodule` parameter is 1, both the WPA supplicant daemon and the Wi-Fi module are stopped.
    - If `stopmodule` = 0, the WPA supplicant stops, but the Wi-Fi module remains on.

Configure all IP Interfaces

Use net_networkUp() and net_networkDown() to bring the network up or down in one call (that is, without specifying the interface name).

Bring the Network Up During the Boot Process

When the V/OS terminal boots up and the corresponding `activate` parameters = 1, the Ethernet and the Wi-Fi parameters are read from `netconf.xml`. Only the specified network interfaces are enabled.

Configuring PPP Over V/OS PSTN Modems

1. Use the net_interfaceSet() command and the `netIfconfig` structure to write the PPP parameters in netconf.xml.
2. Use the `net_interfaceModemExtSet(NET_MODEM_PSTN, netModemInfo)` command to write the V/OS PSTN modem to netconf.xml.

   The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="modem_uart"/>
</PPPPROTO_LIST>

<MODLINK_LIST>
<MODLINK name_id=" modem_uart" … other parameters … />
</MODLINK_LIST>

</SETTINGS>
```

3. Use the `net_interfaceUp( "ppp0", NET_IP_V4 )` command to connect the modem and bring the PPP interface up.
   - Use the `net_interfaceDown( "ppp0" )` command to disconnect the PPP interface and hang up.

NET Service API

This section presents the NET Service constants, structures, and function calls.

NET Service Constants

```
/** @name Define XML default file */
/** \{ */
#define   NET_XML_DEFAULT "/mnt/flash/etc/config/svcnet/netconf.xml"
                          /**< XML default file location */
/** \} */
/** @name Parameter values for ipVersion passed to net_startDHCP,
net_renewDHCP, net_releaseDHCP, ... */
/** \{ */
#define   NET_IP_V4   1 /**< IPv4 is requested */
#define   NET_IP_V6   2 /**< IPv6 is requested */
/** \} */
/** @name svc net max sizes*/
/** \{ */
#define   NET_IFACE_MAX          32 /**< Interface name max size */
#define   NET_NAMEID_MAX         32 /**< XML entry name max size */
#define   NET_IPV4_MAX           16 /**< IPv4 max length */
#define   NET_IPV6_MAX           48 /**< IPv6 max length */
#define   NET_MAC_MAX            17 /**< MAC address max length */
#define   NET_DHID_MAX           16 /**< DHCP id max length */
#define   NET_HOST_MAX          128 /**< DHCP id max length */
#define   NET_SPEED_MAX           8 /**< Ethernet speed parameter */
                                 /**< max length */
#define   NET_PROTO_MAX           8 /**< Protocol name max size */
#define   NET_PPP_USER_MAX       32 /**< PPP username max length */
#define   NET_PPP_PASSWD_MAX     32 /**< PPP password max length */
#define   NET_PPP_AUTH_MAX        8 /**< PPP authentication mode */
                                   /**<  maxlength */
#define   NET_PPP_OPTION_MAX    128   /**< PPP options max length */
#define   NET_WIFI_WPACONF_MAX  128   /**< WPA supplicant configuration */
                                    /**< file max length */
#define   NET_WIFI_SSID_MAX      64  /**< ssid string max length */
#define   NET_WIFI_ENCTYPE_MAX   32 /**< WiFi encryption type max size */
#define   NET_WIFI_AUTHMODE_MAX  32 /**< WiFi authentication mode */
                                   /**<  max size */
/** \} */
?
/** @name values for networkType in netWifiInfo & netSiteSurvey */
/** \{ */
#define   NET_WIFI_NETWORK_MANAGED 1
#define   NET_WIFI_NETWORK_ADHOC   2
/** \} */

/** @name values for speed in netIfconfig structure */
/** \{ */
#define   NET_ETH_AUTO 0 /**<Auto speed negotiation*/
#define   NET_ETH_100F 1 /**<100Mb full duplex */
#define   NET_ETH_10F  2  /**<10Mb full duplex */
#define   NET_ETH_100H 3 /**<100Mb half duplex */
#define   NET_ETH_10H  4 /**<10Mb half duplex */
/** \} */
```

NET Service Structures

- Define an IP interface information

```
struct netIfconfig
{
   char interface[NET_IFACE_MAX];   /**< interface name (eth0, ppp0, */
                                    /**< wlan0...)*/
   char mac[NET_MAC_MAX+1];         /**<MAC address of the interface */
                                    /**< (Read Only)*/
   char local_ip[NET_IPV4_MAX];     /**< current interface IP address */
                                    /** (IPv4) */
   char broadcast[NET_IPV4_MAX];          /**< broadcast */
   char netmask[NET_IPV4_MAX];            /**< netmask */
   char gateway[NET_IPV4_MAX];            /**< gateway */
   char dns1[NET_IPV4_MAX];               /**< DNS primary server */
   char dns2[NET_IPV4_MAX];               /**< DNS secondary server */
   char suppliconf[NET_WIFI_WPACONF_MAX]; /**< WPA_Supplicant */
                                          /**< configuration file */
                                          /**< for WiFi interfaces */
   int usedhcp;/**< 1=DHCP is used, 0 if not */
   char dhcpid[NET_DHID_MAX]; /**< client id string for dhcp */
                              /**< (typically ?? ? not specified) */
   int activate;/**< 1=Bring up the interface at boot, 0 do nothing */
   char user[NET_PPP_USER_MAX];/**< username for PPP */
   char password[NET_PPP_PASSWD_MAX];/**< password for PPP */
   char auth[NET_PPP_AUTH_MAX];/**< authentication mode for */
                               /**< PPP (CHAP or PAP) */
   char pppOption[NET_PPP_OPTION_MAX];/**< PPP options */
   char pppPort[NET_IFACE_MAX];/**< link used for */
                               /**< PPP (Modem, Serial, GPRS) */
};
```

- Define an IPv6 interface:

```
struct netIfconfigIpv6
{
   char interface[NET_IFACE_MAX];/**< interface name (eth0, ppp0, */
                                 /**< wlan0...) */
   char mac[NET_MAC_MAX+1]; /**<MAC address of interface (Read Only) */
   char linkLocal[NET_IPV6_MAX]; /**< address on the link of the */
                                 /**< interface */
   unsigned char linkLocalMask;/**< Network Mask of the */
                               /**< linkLocal address */
   char siteLocal[NET_IPV6_MAX];/**< IPv6 unicast site-local address */
   unsigned char siteLocalMask;  /**< Network Mask of the */
                                 /**< siteLocal address */
   char globalAddress[NET_IPV6_MAX];/**< Routable IPv6 address */
                                    /**< (public address) */
   unsigned char globalAddressMask; /**< Network Mask of the */
                                    /**< globalAddress */
   char dns1Ipv6[NET_IPV6_MAX];  /**< dns primary server for */
                                 /**< IPv6 domain name resolution */
   char dns2Ipv6[NET_IPV6_MAX];  /**< dns secondary server for */
```

```
                                    /**< IPv6 domain name resolution */
    char gatewayIpv6[NET_IPV6_MAX];  /**< gateway for IPv6 */
                                        /**< configuration */
    char suppliconf[NET_WIFI_WPACONF_MAX];/**< WPA_Supplicant */
                            /**< configuration file for WiFi interfaces */
    int usedhcp;/**< 1=DHCP is used, 0 if not */
    int activate;/**< 1=Bring up the interface at boot, 0 do nothing */
};
```

```
/**
```

- Define a list of interface configuration:

```
*/
struct netIfconfigList
{
    struct netIfconfig *ifconfig; /**< list of interfaces IPv4 */
                                    /**< configured */
    int ifconfig_count; /**< number of interfaces in the */
                        /**< list IPv4 configured */
    struct netIfconfigIpv6 *ifconfigIpv6;/**< list of interfaces */
                                            /**< IPv6 configured */
    int ifconfigIpv6_count; /**< number of interfaces in the */
                            /**< list IPv6 configured */
};
```

- Define a route to an IPv4 address:

```
struct netRoute
{
    char interface[NET_IFACE_MAX];/**< interface name (eth0, */
                                    /**< ppp0, wlan0...)*/
    char destination[NET_IPV4_MAX];   /**< route destination */
    char gateway[NET_IPV4_MAX];       /**< route gateway */
    char netmask[NET_IPV4_MAX];       /**< network route netmask */
};
```

- Define a route to an IPv6 host:

```
struct netRouteIpv6
{
    char interface[NET_IFACE_MAX];/**< interface name (eth0, ppp0, */
                                    /**< wlan0...)*/
    char globalHostAddress[NET_IPV6_MAX];/**< global address */
    unsigned char globalHostAddressMask; /**< Network mask */
    char IPv6Gateway[NET_IPV6_MAX];/**< Gateway to globalHostAddress */
};
```

- Define a list of routes:

```
struct netRouteTable
{
   struct netRoute *route;/**< list of routes to IPv4 addresses */
   int route_count; /**< number of IPv4 routes in the list */
   struct netRouteIpv6 *routeIpv6;   /**< list of routes to */
                                     /**< IPv6 addresses */
   int routeIpv6_count; /**< number of IPv6 routes in the list */
};
```

- Define ping statistics:

```
struct netPingInfo
{
   unsigned char ntransmitted;/**< number of transmitted packets */
   unsigned char nreceived;/**< number of received packets */
   double tsum;/**< total round trip */
   double tmin;/**< max round trip */
   double tmax;/**< min round trip */
};
```

- Wi-Fi site survey structure:

```
struct netSiteSurvey
{
   char ssid[NET_WIFI_SSID_MAX]; /**< ssid string */
   int  channel; /**< channel number 1 - 14 */
   int  signalStrength; /**< signal strength 0 - 100 (max) */
   char bssid[NET_MAC_MAX+1]; /**< bssid string */
   char encrypType[NET_WIFI_ENCTYPE_MAX];  /**< encryption type: */
                                            /**< none,wep,tkip,aes */
char authMode[NET_WIFI_AUTHMODE_MAX]; /**< authentication mode: */
             /**< open,shared,wep,wpa-personal,wpa2-personal, */
             /**> wpa-none,wpa-enterprise,wpa2-enterprise */
int  networkType; /**< adhoc, managed (infrastructure) */
};
```

- Define a list of site survey elements:

```
struct netSiteSurveyList
{
   struct netSiteSurvey *siteSurvey;/**< list of site survey elements */
   int siteSurvey_count; /**< number of elements in list */
};
```

- Wi-Fi connection info structure:

```
struct netWifiInfo
{
   char ssid[NET_WIFI_SSID_MAX]; /**< ssid string */
```

```
    char bssid[NET_MAC_MAX+1]; /**< bssid string */
    int  networkType; /**< adhoc, infrastructure */
    int  signalStrength; /**< signal strength 0 - 100 (max) */
};
```

- Modem connection info structure:

```
*/
struct netModemInfo
{
    char number[NET_MODEM_NUMBER_MAX];/**<phone number to dial */
    char dial_type;/**<tone / pulse */
    char mode;/**<async / sync */
    char max_retries;/**<dial retries */
    char max_line_rate;/**<Modem Rate */
    char connect_timeout;/**<waiting for answer, if expired no more
retries */
    char error_correction;/**<error_correction must be enable*/
    char compression; /**Compression activated, yes or no */
};
```

- Modem ISDN info structure:

```
*/
struct netModemIsdnInfo
{
    char number[NET_MDM_ISDN_NUMBER_MAX_LEN];
    int isdn_prot; /**<See above for isdn_prot values */
    char x25_packet_size[NET_MDM_X25_PACKET_SIZE_LEN];/**< packet size for
X.25 connection (Value from 64 - 2048) */
    char facilities[NET_MDM_FACILITIES_MAX_LEN];/**< access to X.25 Closed
User Group , starting with 'G' */
    char nui[NET_MDM_NUI_MAX_LEN]; /**< NUI (Network User Identification)
and password with call setup, allowed chars : a-z, A-Z, 0-9  */
    char x25_number[NET_MDM_X25_NUMBER_MAX_LEN];/**< dialed X.25 call
number, (X.25 B channel only) */
    char user_data[NET_MDM_USER_DATA_MAX_LEN];/**< user data starting with
D(without protocol ID, data length max. 16 char). P:  with protocol ID,
data length max. 12 char */
    char max_retries; /**<dial retries */
    char connect_timeout; /**<waiting for answer, if expired no more
retries */
    int country_code; /**<currently without meaning */
};
```

- Define a descriptor for a SESSION entry in a session XML file:

```
*/
struct sessionDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current session in the XML
file */
   int priority;/**< Define the current session priority */
   char nextMedia[NET_NAMEID_MAX];/**< indentify the protocol used by the
session */
   double timeout;/**< timeout for the connection */
   int retry;/**< number of retries for the connection */
};
```

- Define a descriptor for a SOCKET entry in a session XML file:

```
struct socketDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current socket in the XML
file */
   char protocol[NET_PROTO_MAX];/**< socket type (possible values: TCP or
UDP */
   char remote_ip[NET_HOST_MAX];/**< host dnsname/IPv4 address format:
IP:PORT */
   char nextMedia[NET_NAMEID_MAX];/**< indentify the physical layer (WIFI,
ETHERNET) used by this protocol */
};
```

- Define a descriptor for an Ethernet entry in a session XML file:

```
struct ethernetDescriptor
{
    char name[NET_IFACE_MAX];/**< Indentify the current Ethernet entry in
the XML file */
    struct netIfconfig ifconfig;/**< IP interface configuration */
};
```

- Define a descriptor for a Wi-Fi entry in a session XML file:

```
*/
struct wifiDescriptor
{
    char name[NET_IFACE_MAX];/**< Indentify the current ethernet entry in
the XML file */
    struct netIfconfig ifconfig;/**< IP interface configuration */
};
```

- Define a descriptor for a serial entry in a session XML file:

```
*/
struct serialDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current Serial entry in
                                   the XML file */
};
```

- Define a descriptor for a modem entry in a session XML file:

```
*/
struct modemDescriptor
{
  char name[NET_NAMEID_MAX];/**< Indentify the current Modem entry in the
                                 XML file */
   struct netModemInfo mdmInfo;/**< Contains the modem parameters */
};
```

- Define a descriptor for an ISDN modem entry in a session XML file:

```
struct modemIsdnDescriptor
{
  char name[NET_NAMEID_MAX];/**< Indentify the current Modem entry in the
XML file */
   struct netModemIsdnInfo mdmInfo;/**< Contains the modem parameters */
};
```

- Define a descriptor for a PPP entry in a session XML file:

```
struct pppDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current PPP entry in the
XML file */
   struct netIfconfig ifconfig;/**< IP interface configuration */
};
```

- Define a descriptor for an Option entry in a session XML file:

```
struct optionDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current Option entry in
the XML file */
   char option[NET_OPTION_MAX];/**<Used to store the current option
list>*/
};
```

- Define a descriptor for an SSL entry in a session XML file:

```
struct sslDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current Option entry in
the XML file */
   char remote_ip[NET_HOST_MAX];/**< host dnsname/IPv4 address format:
IP:PORT */
   char server_profile[NET_HOST_MAX];/**< identify the place where the
Server CA is stored */
   char client_profile[NET_HOST_MAX];/**< identify the place where the
Client Certificate and the Client Private Key ara stored */
```

```
   int  session_ttl;/**< Indicate is seconds how log to keep the Resume
Session before to perform a complete Handshake */
   char nextMedia[NET_NAMEID_MAX];/**< indentify the physical layer (WIFI,
ETHERNET) or the protocol used by this SSL layer */
};
```

- Define a descriptor for a GPRS entry in a session XML file:

```
struct netGprsInfo
{
   char APN[NET_HOST_MAX];/**< GPRS Access Point Name */
   int operator_id;/**< Operator LAI number */
   int reg_mode;/**< Network Registration Mode (NET_GPRS_REG_AUTO,
NET_GPRS_REG_MANUAL, NET_GPRS_REG_AUTO_MANUAL)  */
   char number[NET_GPRS_NUMBER_MAX];/**< Phone number used for GPRS
activation or for GSM data calls */
   char bearer[NET_GPRS_BEARER_MAX];/**< GSM Data modulation (NOT
IMPLEMENTED!) */
};
```

- GPRS connection info structure:

```
struct gprsDescriptor
{
   char name[NET_NAMEID_MAX];/**< Indentify the current Modem entry in the
XML file */
   struct netGprsInfo gprsInfo;/**< Contains the modem parameters */
};
```

- Hold the current and the next media information:

```
struct netSessionInfo
{
   int comtype;/**< This is the current media type */
   char name[NET_NAMEID_MAX];/**< This is the current interface name*/
};
```

- Convey the NET session data:

```
struct netDataTransfer
{
   void *data;/**< array of bytes to transfer */
   int data_count; /**< number of bytes to transfer */
};
```

- Convey information about GSM environment (module+sin+user):

```
struct netGsmInfo
{
   char library_version[NET_NAMEID_MAX];/**< GSM library Version */
   char manufacturer[NET_NAMEID_MAX];/**< GSM Module manufacturer */
   char model[NET_NAMEID_MAX];/**< GSM Module model */
   char revision[NET_NAMEID_MAX];/**< GSM Module version */
   char imei [NET_NAMEID_MAX];/**< Module IMEI */
   char sim_card_id[NET_NAMEID_MAX];/**< SIM card */
   char imsi [NET_NAMEID_MAX];/**< User IMSI (PIN Needed)*/
};
```

- Get the GSM RSSI level:

```
struct netGsmRssi
{
   int rssi; /**< signal strength */
};
```

NET Service Functions

This section contains V/OS-based terminal NET service function call descriptions.

net_getVersion()

Retrieves the version of the NET service.

Prototype

```
struct version net_getVersion(void);
```

Return Values

`version` struct as defined in `svcmgrSvcDef.h`.

On success, errno = 0.

net_interfaceSet()

Writes the IP interface configuration to `netconf.xml`.

Prototype

```
int net_interfaceSet( struct netIfconfig config );
```

Parameters

`config`	Convoy interface data to store in the XML file.

Return Values

errno = 0	Success
errno ! = 0	On error with errno set to:

- EINVAL – Invalid input parameter.
- ESRCH – `netconf.xml` file read error.

net_interfaceGet()

Reads the IP interface configuration from `netconf.xml`.

Prototype

```
struct netIfconfig net_interfaceGet( char *p_iface );
```

Parameters

`p_iface`	Network interface name (eth0 or wlan0) where the data to read is present.

Return Values

errno = 0	Success A `netIfconfig struct` containing fields corresponding to `p_iface`.
errno ! = 0	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceModemSet()

Write the modem interface configuration to the XML file.

Prototype

```
int net_interfaceModemSet(struct netModemInfo mdnInfo);
```

Parameters

`[input] mdnInfo`	Convoy modem parameters to store in the XML file.

Return Values

errno = 0	Success A `netModemInfo struct` containing modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceModemGet ()

Read modem configuration from the XML file.

Prototype

```
struct netModemInfo mdnInfo net_interfaceModemGet();
```

Return Values

errno = 0	Success A `netModemInfo struct` containing modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceModemExtSet()

Write modem interface configuration in the XML file.

Prototype

```
int net_interfaceModemExtSet(char *iface, struct netModemInfo mdnInfo);
```

Parameters

`[input] iface`	The modem name.
`[input] mdnInfo`	Convoy modem parameters to store in the XML file.

Return Values

errno = 0	Success A `netModemInfo struct` containing modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceModemExtGet()

Read modem configuration from the XML file.

Prototype

```
struct netModemInfo mdnInfo net_interfaceModemExtSet(char *iface);
```

Parameters

`[input] iface`	The modem name.

Return Values

errno = 0	Success A `netModemInfo struct` containing modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceModemIsdnSet()

Write the modem ISDN interface configuration in the XML file.

Prototype

```
int net_interfaceModemIsdnSet( struct netModemIsdnInfo mdnInfo );
```

Parameters

`[input] mdnInfo`	Convey the ISDN modem parameters to store in the XML file.

Return Values

errno = 0	Success A `netModemInfo struct` containing the ISDN modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – XML file error

net_interfaceModemIsdnGet()

Read the modem ISDN configuration from the XML file.

Prototype

```
struct netModemIsdnInfo net_interfaceModemIsdnGet();
```

Return Values

errno = 0	Success A `netModemInfo struct` containing the ISDN modem parameters read from the XML file.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – Interface not found in `netconf.xml`.

net_interfaceGprsSet()

Write the GPRS interface configuration in the XML file.

Prototype

```
int net_interfaceGprsSet( struct netGprsInfo gprsInfo );
```

Parameters

None.

Return Values

errno = 0	Success.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – XML file error.

net_interfaceGprsGet()

Read the GPRS configuration from the XML file.

Prototype

```
struct netGprsInfo gprsInfo net_interfaceGprsSet();
```

Parameters

`[input] gprsInfo`	Convoy GPRS parameters to store in the XML file.

Return Values

errno = 0	Success
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – XML file error.

net_optionSet()

Write options for an Interface into the XML file.

Prototype

```
int net_optionSet( char *p_iface, char *p_option );
```

Parameters

`[input] p_iface`	Interface for Wi-Fi to write options.
`[input] p_option`	Options to write to the XML file.

Return Values

errno = 0	Success
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – XML file error.

net_optionGet()

Read options for an Interface from the XML file.

Prototype

```
char *net_optionGet(char *p_iface);
```

Parameters

`[input] p_iface`	Interface name on which to read the configuration.
`[input] p_option`	Options to write to the XML file.

Return Values

errno = 0	On success the option string read from the XML file. **Note:** The user must free the string.
errno = -1	On error with errno set to: • EINVAL – Invalid input parameter. • ESRCH – The option was not found in the XML file.

net_interfaceUp()

Brings the network up for the specified interface set by `netconf.xml` parameters. If the `netconf.xml activate` attribute is set to 1, bring up the network; otherwise do nothing.

Prototype

```
int net_interfaceUp( char *p_iface, int ipVersion );
```

Parameters

`p_iface`	Network interface to configure (eth0 or wlan0).
`ipVersion`	NET_IP_V4

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – `netconf.xml` file read error.
- ESRCH – Interface not found in `netconf.xml`.

net_interfaceDown()

Closes the network for the specified IP interface. The `activate` parameter in `netconf.xml` is not impacted.

Prototype

```
int net_interfaceDown( char *p_iface, int ipVersion );
```

Parameters

`p_iface`	Network interface to bring down (`eth0` or `wlan0`).
`ipVersion`	NET_IP_V4

Return Values

=0	Success
-1	On error with errno set to:

- ENONET – No network interface up.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – `netconf.xml` file read error.

net_interfaceStatus()

Reads the interface link status and return the current interface IP information. The MAC Address is filled. The `activate` parameter updates according to its value in `netconf.xml`.

Prototype

```
struct netIfconfig net_interfaceStatus( char *p_iface );
```

Parameters

`p_iface`	Network interface to read (eth0 or wlan0).

Return Values

errno = 0	Success
errno ! = 0	On error with errno set to:

- ENONET – No network interface up.
- ENETDOWN – Network Interface is not online (for example, the Ethernet cable is not plugged in).
- ENOTCONN – Device not on (Wi-Fi module down);

net_networkUp()

Brings the network up for all interfaces as set in the `netconf.xml` parameters. If the `netconf.xml activate` parameter is 1, bring up the network; otherwise do nothing.

Prototype

```
int net_networkUp( int ipVersion );
```

Parameters

`ipVersion`	Must be NET_IP_V4.

Return Values

= 0	Success
= -1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – `netconf.xml` file read error.
- ESRCH – Interface not found in `netconf.xml`.
- ENOTCONN – `eth0` and `wlan0` not configured.

net_networkDown()

Closes the network for all interfaces as set in the `netconf.xml` parameters. The `netconf.xml activate` parameter is not impacted.

Prototype

```
int net_ networkDown ( int ipVersion );
```

Parameters

`ipVersion`	Must be NET_IP_V4.

Return Values

=0	Success
-1	On error with errno set to:

- ENONET – No network interface is up.
- ENETDOWN – Network Interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – `netconf.xml` file read error.

net_interfaceStatusList()

Read the list of all active network interface configurations.

Prototype

```
struct netIfconfigList net_interfaceStatusList( void );
```

Return Values

errno = 0	Success `struct netIfconfigList` contains the list of network interface configurations.
errno ! = 0	On error with errno set to: • ENONET – No active network interface found.

net_getTable()

Read the routing table.

Prototype

```
struct netRouteTable net_getTable( void );
```

Return Values

errno = 0	Success
errno ! = 0	On error with errno set to: • EBADF – Routing table read error. • ENOEXEC – Routing table column missing. • EDESTADDRREQ – No route found in the routing table.

net_getRoutes()

Read all routes to the specified destination.

Prototype

```
struct netRouteTable net_getRoutes( char *p_dest );
```

Parameters

`p_dest`	IP destination for the routes.

Return Values

errno = 0	Success `struct netRouteTable` the table route information structure.
errno ! = 0	On error with errno set to: • EBADF – Routing table read error. • ENOEXEC – Routing table column missing. • EDESTADDRREQ – No route found in the routing table.

net_addHostRoute()

Add a route to the specified host.

Prototype

```
int net_addHostRoute( char *p_dest, char *p_gateway, char *p_iface );
```

Parameters

`p_dest`	Host IP address.
`p_gateway`	Gateway address for host.
`p_iface`	IP interface (`eth0` and `wlan0`).

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT– Invalid IP address passed.
- EBADF – Routing table read error.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.
- EHOSTUNREACH – Host name resolution error.

net_delHostRoute()

Delete a route to the specified host.

Prototype

```
int net_delHostRoute( char *p_dest );
```

Parameters

`p_dest`	Host IP address.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT– Invalid IP address passed.
- EBADF – Routing table read error.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.

net_addNetRoute()

Add a route to the specified network.

Prototype

```
int net_addNetRoute( char *p_net, char *p_netmask, char *p_gateway,
char *p_iface );
```

Parameters

`p_dest`	Host IP address.
`p_netmask`	Network mask for the host.
`p_gateway`	Gateway address for host.
`p_iface`	IP interface (`eth0` and `wlan0`).

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT– Invalid IP address passed.
- EBADF – Routing table read error.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.
- ENETDOWN – Interface not found in the active interfaces list.

net_delNetRoute()

Deletes a route to the specified network.

Prototype

```
int net_delNetRoute( char *p_net, char *p_netmask, char *p_gateway,
char *p_iface );
```

Parameters

`p_dest`	Host IP address.
`p_netmask`	Network mask for the host.
`p_gateway`	Gateway address for host.
`p_iface`	IP interface (`eth0` and `wlan0`).

Return Values

=0 Success

-1 On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT– Invalid IP address passed.
- EBADF – Routing table read error.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.
- ENETDOWN – Interface not found in the active interfaces list.

net_autoSetDefaultRoute()

Selects the default route per priority policy (`eth0`, `wlan0`) settings.

Prototype

```
int net_addNetRoute( char *p_net, char *p_netmask, char *p_gateway,
char *p_iface );
```

Return Values

=0	Success
-1	On error with errno set to:

- ENONET – No active interface found.

net_setDefaultRoute()

Sets the default route on the specified interface.

Prototype

```
int net_setDefaultRoute( char *p_iface );
```

Parameters

`p_iface`	Default IP interface (`eth0` and `wlan0`).

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EBADF – Routing table read error.
- ENONET – No active interface found.

net_getDefaultRouteInfo()

Retrieves the default route interface name.

Prototype

```
char * net_getDefaultRouteInfo( void );
```

Return Values

Interface name (`eth0`, `wlan0`) of the default route.

Null on error.

On error with errno set to:

- = 0 – Success
- EINVAL – Invalid input parameter.
- EBADF – Routing table read error.
- ENONET – No active interface found.

net_startDHCP()

Starts a DHCP client for the specified interface. Does nothing if DHCP is up. Switches from static IP to dynamic configuration.

Prototype

```
int net_startDHCP( char *p_iface, int ipVersion );
```

Parameters

`p_iface`	IP interface (`eth0` and `wlan0`).
`ipVersion`	Must be NET_IP_V4.

Return Values

=0 Success

-1 On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).

net_renewDHCP()

Causes `udhcpc` to renew the current lease or, if it does not have one, obtain a new lease.

Prototype

```
int net_renewDHCP( char *p_iface, int ipVersion );
```

Parameters

`p_iface`	IP interface (`eth0` and `wlan0`) for the renewal request.
`ipVersion`	Must be NET_IP_V4.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- ENONET – No network interface found.

net_releaseDHCP()

Causes `udhcpc` to release the current lease. Call `net_renewDHCP()` to get a new lease.

Prototype

```
int net_releaseDHCP( char *p_iface, int ipVersion );
```

Parameters

`p_iface`	IP interface (`eth0` and `wlan0`) to release.
`ipVersion`	Must be NET_IP_V4.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- ENONET – No network interface found.

net_stopDHCP()

Stops the DHCP client for the specified interface.

Prototype

```
int net_stopDHCP( char *p_iface,  int ipVersion );
```

Parameters

`p_iface`	IP interface (`eth0` and `wlan0`) of the DHCP client.
`ipVersion`	Must be NET_IP_V4.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENETDOWN – Network interface is not online (for example, the Ethernet cable is not plugged in).
- ENONET – No network interface found.

net_setDNS()

Sets the DNS for the specified interface.

Prototype

```
int net_setDNS( char *p_iface, char *p_dns1, char *p_dns2 );
```

Parameters

`p_iface`	IP interface (`eth0` and `wlan0`) of the DHCP client.
`p_dns1`	Domain name server 1.
`p_dns2`	Domain name server 2.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT – Invalid IP address passed.
- EDESTADDRREQ – No route found.
- ENETDOWN – No gateway found.

net_addRouteFromXML()

Adds static routes from the specified XML file.

Prototype

```
int net_addRouteFromXML( char *p_fileName );
```

Parameters

`p_fileName`	XML file containing the static routes.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ESRCH – XML file read error.

net_nsLookup()

Performs DNS resolution on the specified interface.

Prototype

```
struct netIfconfig net_nsLookup( char *p_dnsName, char *p_iface );
```

Parameters

`p_dnsName`	DNS name to resolve.
`p_iface`	IP interface (`eth0` and `wlan0`) of the DHCP client.

Return Values

errno = 0 Success

errno ! = 0 On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT – Parameter passed invalid IP address.
- EBADF – DNS file read error.
- ENONET – IP interface is up.
- ENOLINK – DNS link file error.

net_ping()

Removes all stored DNS resolution results for the specified server.

Prototype

```
struct netPingInfo net_ping( char *p_host, unsigned char pingCount );
```

Parameters

`p_host`	Ping destination (IPv4 or IPv6).
`pingCount`	

Return Values

errno = 0	Success
errno ! = 0	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENOMSG – No packet transmitted.
- EHOSTUNREACH – No packet received.
- EBADMSG – Number of packets received lower than packets transmitted.
- EFAULT – `p_host` invalid.

net_wifiStart()

Starts the Wi-Fi module and WPA supplicant as set in the specified parameter.

Prototype

```
int net_wifiStart( int startwpa );
```

Parameters

`startwpa`	= 1 – power up the Wi-Fi module and start the WPA supplicant
	= 0 – only power up the Wi-Fi module.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EBADF – Wi-Fi configuration file not found.

net_wifiStop()

Stops the Wi-Fi module and WPA supplicant as set in the specified parameter.

Prototype

```
int net_wifiStop( int stopmodule);
```

Parameters

`stopmodule`	= 1 – Stop the WPA supplicant and Wi-Fi module.
	= 0 – Stop the WPA supplicant only.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EBADF – Wi-Fi configuration file not found.

net_wifiSiteSurvey()

Obtains a Wi-Fi site survey list.

Prototype

```
struct netSiteSurveyList net_wifiSiteSurvey( void );
```

Return Values

errno = 0	Success `struct netIfconfigList` is a list of discovered Wi-Fi sites.
errno ! = 0	On error with errno set to: • ENOEXEC – Scan command not ready. • ETIMEDOUT – No Wi-Fi network found. • EBADF – Scan result parsing error. • ENOTCONN – Device not on (Wi-Fi module down).

net_wifiInfo()

Retrieves Wi-Fi information for the specified Wi-Fi interface (`wlan0`).

Prototype

```
struct netWifiInfo net_wifiInfo( char *p_iface );
```

Parameters

`p_iface`	IP interface (`wlan0`).

Return Values

errno = 0	Success `struct netWifiInfo` – structure containing connected Wi-Fi network information.
errno ! = 0	On error with errno set to: • ENETDOWN – No Wi-Fi network found. • EBADF – Scan result parsing error. • ENOTCONN – Device not on (Wi-Fi module down).

net_setNTP()

Sets the NTP (Network Time Protocol) host name.

Prototype

```
int net_setNTP( char *p_hostname );
```

Parameters

`p_hostname`	Network Time Protocol host name. If NULL, pool.ntp.org is used as the default NTP server.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT – Invalid host name.

net_sessionOpen()

Opens a communication session file.

Prototype

```
int net_sessionOpen( char *p_xmlFile );
```

Parameters

`*p_xmlFile`	Name of the XML file that stores the current session information.

Return Values

=0	Success
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EBADF – Error when reading the XML file.

net_sessionConnect()

Starts the current communication session as specified in the session file.

Prototype

```
int net_sessionConnect(int sessionType);
```

Parameters

`sessionType`	Valid values are: • NET_FIRST_SESSION – Connect the session with lowest priority. • NET_NEXT_SESSION – Connect the following session in the XML file. • NET_CURRENT_SESSION – Reconnect to the current session.

Return Values

=0	Success; connection handle returns
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT – Invalid IP address passed.
- EHOSTUNREACH – Unable to reach the host.
- ETIMEDOUT – Connection timeout.

net_sessionConnectExt()

Connect to a session by its priority.

Prototype

```
int net_sessionConnectExt(int priority);
```

Parameters

`priority`	Session priority number to connect to (must be greater than 0)

Return Values

=0	Success; connection handle returns.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- EFAULT – Invalid IP address passed.
- EHOSTUNREACH – Unable to reach the host.
- ETIMEDOUT – Connection timeout.

net_sessionRead()

Reads data from the host.

Prototype

```
struct netDataTransfer net_sessionRead(int handle, int count,
                                       int timeout);
```

Parameters

`handle`	Communication returned by net_sessionConnect().
`count`	The number of bytes to read.
`timeout`	The first timeout in seconds to wait for data reception.

Return Values

=0 Success.

The `struct netDataTransfer` contains the general data structure transfer.

-1 On error with errno set to:

- EINVAL – Invalid input parameter.
- ENONET – No active interface found.
- ETIMEDOUT – Connection timeout.

net_sessionWrite()

Writes data to the host.

Prototype

```
int net_sessionWrite(int handle, struct netDataTransfer data);
```

Parameters

`handle`	Communication returned by net_sessionConnect().
`data`	Convoy data to send to the host

Return Values

=0	Success. The `struct netDataTransfer` contains the general data structure transfer.
-1	On error with errno set to: • EINVAL – Invalid input parameter. • ENONET – No active interface found.

net_sessionSetTag()

Changes a tag entry in the currently open XML file.

Prototype

```
int net_sessionSetTag(int comtype, char *nameid,
                      struct netDataTransfer data);
```

Parameters

`comtype`	Communication family type (NETCOM_SESSION, COM_SSL, and so on).
`nameid`	The entry in the entire XML file
`data`	Contains a pointer to the tag descriptor (sessionDescriptor, socketDescriptor, and so on).

Return Values

=0	Success.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ESRCH – Errors while writing to the XML file.

net_sessionGetTag()

Retrieves a tag entry in the currently open XML file.

Prototype

```
int net_sessionSetTag(int comtype, char *nameid,
                      struct netDataTransfer data);
```

Parameters

`comtype`	Communication family type (NETCOM_SESSION, COM_SSL, and so on).
`nameid`	The entry in the entire XML file

Return Values

=0	Success.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ESRCH – Errors while writing to the XML file.

net_sessionDisconnect()

Terminates the current communication session.

Prototype

```
int net_sessionDisconnect(int handle);
```

Parameters

`handle`	Current session ID.

Return Values

=0	Success.
-1	On error with errno set to: • EINVAL – Invalid input parameter.

net_sessionClose()

Terminates the current communication session and releases all resources.

Prototype

```
int net_sessionClose();
```

Parameters

None.

Return Values

=0	Success.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.

net_gprsVerifyPin()

Unlocks the SIM by providing the PIN.

Prototype

```
int net_gprsVerifyPin(int pin);
```

Parameters

`pin`	PIN value to send to the SIM card to unlock it.

Return Values

=0	Success.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.

net_gprsGetStatus()

Retrieves the GPRS module status (SIM present, PIN Ready, GPRS Connected, and so on).

Prototype

```
int net_gprsGetStatus();
```

Parameters

None.

Return Values

=0	Success returns NET_GSM_STT_[XXX].
-1	On error with errno set to:

- EINVAL – Invalid input parameter.
- ENOEXEC – Unable to communicate with the GPRS module.

net_gprsGetStatus()

Returns information about the GSM module.

Prototype

```
int net_gprsGetStatus();
```

Parameters

None.

Return Values

=0	Success returns the GSM module information to struct netGsminfo.
-1	On error with errno set to: • ENOEXEC – Unable to communicate with the GPRS module.

net_gprsGetRssi()

Returns the signal RSSI level.

Prototype

```
int net_gprsGetRssi();
```

Parameters

None.

Return Values

=0	Success returns the RSSI level information to struct netGsmRssi.
-1	On error with errno set to: • ENOEXEC – Unable to communicate with the GPRS module.

net_gprsRssi()

Returns the signal RSSI level.

Prototype

```
int net_gprsRssi();
```

Parameters

`pin`	PIN value to send to the SIM card to unlock it.

Return Values

=0	Success.
-1	On error with errno set to:

- EINVAL – Invalid input parameter.

WPA Supplicant for Wi-Fi Security

The Wi-Fi feature is subject to hardware availability.

When the WPA Supplicant package is installed and selected for Wi-Fi networking (`*NET` set to ‘w’, and `*WPASUPP` set to 1 in config.usr1), then the `wlan.conf` file contents configure the Wi-Fi hardware (channel, power mode, and so on), and specifies the DHCP or static IP (with gateway, and DNS1 and 2) and the `/home/usr1/secure/wpa_supplicant.conf` file (or `/home/usr16/secure/wpa_supplicant.conf` if `conf.usr1` is missing) is used by the WPA supplicant daemon establish an encrypted connection. The only portion of wlan.conf not used are the SSID, encryption type, and password parameters.

Example 1 wpa_supplicant.conf

```
ctrl_interface=/var/run/wpa_supplicant

network={
    ssid="my_ssid_here"
    key_mgmt=WPA-EAP
    proto=WPA2
    pairwise=CCMP
    group=CCMP
    eap=TLS
    identity="client"
    ca_cert="/etc/wpa_supplicant/cert/ca.crt"
    client_cert="/etc/wpa_supplicant/cert/client1.crt"
    private_key="/etc/wpa_supplicant/cert/client1.key"
}
```

Example 2 wpa_supplicant.conf

```
# WPA-PSK/TKIP

ctrl_interface=/var/run/wpa_supplicant

eapol_version=1
# ap_scan=2 was the one for me you may try 0 or 1 indstead of 2
ap_scan=2
fast_reauth=1

#
# WPA Personal, TKIP algorithm, shared key: testing123-vfi
network={
        ssid="ViperNest"
        proto=WPA
        key_mgmt=WPA-PSK
        pairwise=TKIP
        group=TKIP
        psk="testing123-vfi"
}
```

Help description for wpa_supplicant.conf

```
##### Example wpa_supplicant configuration file #########################
#
# This file describes configuration file format and lists all available
# options.
# Please also take a look at simpler configuration examples in 'examples'
# subdirectory.
#
# Empty lines and lines starting with # are ignored

# NOTE! This file may contain password information and should probably be
# made readable only by root user on multiuser systems.

# Note: All file paths in this configuration file should use full
# (absolute, not relative to working directory) path in order to allow
# working directory to be changed. This can happen if wpa_supplicant is
# run in the background.

# Whether to allow wpa_supplicant to update (overwrite) configuration
#
# This option can be used to allow wpa_supplicant to overwrite
# configuration file whenever configuration is changed (e.g., new network
# block is added with wpa_cli or wpa_gui, or a password is changed). This
# is required for wpa_cli/wpa_gui to be able to store the configuration
# changes permanently.
# Please note that overwriting configuration file will remove the comments
# from it.
#update_config=1

# global configuration (shared by all network blocks)
#
# Parameters for the control interface. If this is specified,
# wpa_supplicant will open a control interface that is available for
# external programs to manage wpa_supplicant. The meaning of this string
# depends on which control interface mechanism is used. For all cases, the
# existence of this parameter in configuration is used to determine
# whether the control interface is enabled.
#
# For UNIX domain sockets (default on Linux and BSD): This is a directory
# that will be created for UNIX domain sockets for listening to requests
# from external programs (CLI/GUI, etc.) for status information and
# configuration. The socket file will be named based on the interface
# name, so multiple wpa_supplicant processes can be run at the same time
if more than one interface is used.
# /var/run/wpa_supplicant is the recommended directory for sockets and by
# default, wpa_cli will use it when trying to connect with wpa_supplicant.
#
# Access control for the control interface can be configured by setting
# the directory to allow only members of a group to use sockets. This way,
# it is possible to run wpa_supplicant as root (since it needs to change
# network configuration and open raw sockets) and still allow GUI/CLI
# components to be run as non-root users. However, since the control
# interface can be used to change the network configuration, this access
# needs to be protected in many cases. By default, wpa_supplicant is
# configured to use gid 0 (root). If you want to allow non-root users to
# use the control interface, add a new group and change this value to
# match with that group. Add users that should have control interface
# access to this group. If this variable is commented out or not included
# in the configuration file, group will not be changed from the value it
# got by default when the directory or socket was created.
#
# When configuring both the directory and group, use following format:
# DIR=/var/run/wpa_supplicant GROUP=wheel
# DIR=/var/run/wpa_supplicant GROUP=0
# (group can be either group name or gid)
#
# For UDP connections (default on Windows): The value will be ignored.
# This variable is just used to select that the control interface is to be
# created.
# The value can be set to, e.g., udp (ctrl_interface=udp)
#
# For Windows Named Pipe: This value can be used to set the security
# descriptor for controlling access to the control interface. Security
# descriptor can be set using Security Descriptor String Format (see
# http://msdn.microsoft.com/library/default.asp?url=/library/en-us/
# secauthz/security/security_descriptor_string_format.asp).
# The descriptor string needs to be prefixed with SDDL=. For example,
# ctrl_interface=SDDL=D: would set an empty DACL (which will reject all
# connections). See README-Windows.txt for more information about SDDL
# string format.
#

ctrl_interface=/var/run/wpa_supplicant

# IEEE 802.1X/EAPOL version
# wpa_supplicant is implemented based on IEEE Std 802.1X-2004 which
# defines EAPOL version 2. However, there are many APs that do not handle
# the new version number correctly (they seem to drop the frames
# completely). In order to make wpa_supplicant interoperate with these
# APs, the version number is set to 1 by default. This configuration
# value can be used to set it to the new version (2).
eapol_version=1

# AP (Access Point) scanning/selection
# By default, wpa_supplicant requests driver to perform AP scanning and
# then uses the scan results to select a suitable AP. Another alternative
# is to allow the driver to take care of AP scanning and selection and use
# wpa_supplicant just to process EAPOL frames based on IEEE 802.11
# association
# information from the driver.
# 1: wpa_supplicant initiates scanning and AP selection
# 0: driver takes care of scanning, AP selection, and IEEE 802.11
# association parameters (e.g., WPA IE generation); this mode can also be
# used with non-WPA drivers when using IEEE 802.1X mode; do not try to
# associate with APs (i.e., external program needs to control
# association). This mode must also be used when using wired Ethernet
# drivers.
# 2: like 0, but associate with APs using security policy and SSID (but
# not BSSID); this can be used, e.g., with ndiswrapper and NDIS drivers to
# enable operation with hidden SSIDs and optimized roaming; in this mode,
#    the network blocks in the configuration file are tried one by one
#    until the driver reports successful association; each network block
#    should have explicit security policy (i.e., only one option in the
#    lists) for key_mgmt, pairwise, group, proto variables
# ap_scan=1

# EAP fast re-authentication
# By default, fast re-authentication is enabled for all EAP methods that
# support it. This variable can be used to disable fast re-authentication.
# Normally, there is no need to disable this.
fast_reauth=1

# Maximum lifetime for PMKSA in seconds; default 43200
#dot11RSNAConfigPMKLifetime=43200
# Threshold for reauthentication (percentage of PMK lifetime); default 70
#dot11RSNAConfigPMKReauthThreshold=70
# Timeout for security association negotiation in seconds; default 60
#dot11RSNAConfigSATimeout=60

# network block
#
# Each network (usually AP's sharing the same SSID) is configured as a
# separate block in this configuration file. The network blocks are in
# preference order (the first match is used).
```

```
#
# network block fields:
#
# disabled:
#   0 = this network can be used (default)
#   1 = this network block is disabled (can be enabled through ctrl_iface,
#       e.g., with wpa_cli or wpa_gui)
#
# id_str: Network identifier string for external scripts. This value is
# passed to external action script through wpa_cli as WPA_ID_STR
# environment variable to make it easier to do network specific
configuration.
#
# ssid: SSID (mandatory); either as an ASCII string with double quotation
# or as hex string; network name
#
# scan_ssid:
#   0 = do not scan this SSID with specific Probe Request frames (default)
#   1 = scan with SSID-specific Probe Request frames (this can be used to
#       find APs that do not accept broadcast SSID or use multiple SSIDs;
#       this will add latency to scanning, so enable this only when needed)
#
# bssid: BSSID (optional); if set, this network block is used only when
#   associating with the AP using the configured BSSID
#
# priority: priority group (integer)
# By default, all networks will get same priority group (0). If some of
# the networks are more desirable, this field can be used to change the
# order in which wpa_supplicant goes through the networks when selecting a
# BSS. The priority groups will be iterated in decreasing priority (i.e.,
# the larger the priority value, the sooner the network is matched against
# the scan results).
# Within each priority group, networks will be selected based on security
# policy, signal strength, etc.
# Please note that AP scanning with scan_ssid=1 and ap_scan=2 mode are not
# using this priority to select the order for scanning. Instead, they try
# the networks in the order that used in the configuration file.
#
# mode: IEEE 802.11 operation mode
# 0 = infrastructure (Managed) mode, i.e., associate with an AP (default)
# 1 = IBSS (ad-hoc, peer-to-peer)
# Note: IBSS can only be used with key_mgmt NONE (plaintext and static
# WEP) and key_mgmt=WPA-NONE (fixed group key TKIP/CCMP). In addition,
# ap_scan has to be set to 2 for IBSS. WPA-None requires following network
# block options: proto=WPA, key_mgmt=WPA-NONE, pairwise=NONE, group=TKIP
# (or CCMP, but not both), and psk must also be set.
```

```
#
# frequency: Channel frequency in megahertz (MHz) for IBSS, e.g.,
# 2412 = IEEE 802.11b/g channel 1. This value is used to configure the
# initial channel for IBSS (adhoc) networks. It is ignored in the
# infrastructure mode.
# In addition, this value is only used by the station that creates the
# IBSS. If an IBSS network with the configured SSID is already present,
the frequency of the network will be used instead of this configured
value.
#
# proto: list of accepted protocols
# WPA = WPA/IEEE 802.11i/D3.0
# RSN = WPA2/IEEE 802.11i (also WPA2 can be used as an alias for RSN)
# If not set, this defaults to: WPA RSN
#
# key_mgmt: list of accepted authenticated key management protocols
# WPA-PSK = WPA pre-shared key (this requires 'psk' field)
# WPA-EAP = WPA using EAP authentication (this can use an external
#  program, e.g., Xsupplicant, for IEEE 802.1X EAP Authentication
# IEEE8021X = IEEE 802.1X using EAP authentication and (optionally)
# dynamically generated WEP keys
# NONE = WPA is not used; plaintext or static WEP could be used
# If not set, this defaults to: WPA-PSK WPA-EAP
#
# auth_alg: list of allowed IEEE 802.11 authentication algorithms
# OPEN = Open System authentication (required for WPA/WPA2)
# SHARED = Shared Key authentication (requires static WEP keys)
# LEAP = LEAP/Network EAP (only used with LEAP)
# If not set, automatic selection is used (Open System with LEAP enabled
# if LEAP is allowed as one of the EAP methods).
#
# pairwise: list of accepted pairwise (unicast) ciphers for WPA
# CCMP = AES in Counter mode with CBC-MAC [RFC 3610, IEEE 802.11i/D7.0]
# TKIP = Temporal Key Integrity Protocol [IEEE 802.11i/D7.0]
# NONE = Use only Group Keys (deprecated, should not be included if APs
# support pairwise keys)
# If not set, this defaults to: CCMP TKIP
#
# group: list of accepted group (broadcast/multicast) ciphers for WPA
# CCMP = AES in Counter mode with CBC-MAC [RFC 3610, IEEE 802.11i/D7.0]
# TKIP = Temporal Key Integrity Protocol [IEEE 802.11i/D7.0]
# WEP104 = WEP (Wired Equivalent Privacy) with 104-bit key
# WEP40 = WEP (Wired Equivalent Privacy) with 40-bit key [IEEE 802.11]
# If not set, this defaults to: CCMP TKIP WEP104 WEP40
#
# psk: WPA preshared key; 256-bit pre-shared key
# The key used in WPA-PSK mode can be entered either as 64 hex-digits,
# i.e., 32 bytes or as an ASCII passphrase (in which case, the real PSK
# will be generated using the passphrase and SSID). ASCII passphrase must
# be between 8 and 63 characters (inclusive).
# This field is not needed, if WPA-EAP is used.
# Note: Separate tool, wpa_passphrase, can be used to generate 256-bit
# keys from ASCII passphrase. This process uses lot of CPU and
# wpa_supplicant startup and reconfiguration time can be optimized by
# generating the PSK only when the passphrase or SSID has actually
# changed.
#
# eapol_flags: IEEE 802.1X/EAPOL options (bit field)
# Dynamic WEP key required for non-WPA mode
# bit0 (1): require dynamically generated unicast WEP key
# bit1 (2): require dynamically generated broadcast WEP key
#   (3 = require both keys; default)
# Note: When using wired authentication, eapol_flags must be set to 0 for
# the authentication to be completed successfully.
#
# mixed_cell: This option can be used to configure whether so called mixed
# cells, i.e., networks that use both plaintext and encryption in the same
# SSID, are allowed when selecting a BSS form scan results.
# 0 = disabled (default)
# 1 = enabled
#
# proactive_key_caching:
# Enable/disable opportunistic PMKSA caching for WPA2.
# 0 = disabled (default)
# 1 = enabled
#
# wep_key0..3: Static WEP key (ASCII in double quotation, e.g. "abcde" or
# hex without quotation, e.g., 0102030405)
# wep_tx_keyidx: Default WEP key index (TX) (0..3)
#
# peerkey: Whether PeerKey negotiation for direct links (IEEE 802.11e DLS)
# is allowed. This is only used with RSN/WPA2.
# 0 = disabled (default)
# 1 = enabled
#peerkey=1
#
# Following fields are only used with internal EAP implementation.
# eap: space-separated list of accepted EAP methods
#   MD5 = EAP-MD5 (unsecure and does not generate keying material ->
#         cannot be used with WPA; to be used as a Phase 2 method
#         with EAP-PEAP or EAP-TTLS)
#     MSCHAPV2 = EAP-MSCHAPv2 (cannot be used separately with WPA; to be
#      used as a Phase 2 method with EAP-PEAP or EAP-TTLS)
#      OTP = EAP-OTP (cannot be used separately with WPA; to be used
#      as a Phase 2 method with EAP-PEAP or EAP-TTLS)
#       GTC = EAP-GTC (cannot be used separately with WPA; to be used
#      as a Phase 2 method with EAP-PEAP or EAP-TTLS)
#   TLS = EAP-TLS (client and server certificate)
#   PEAP = EAP-PEAP (with tunnelled EAP authentication)
#   TTLS = EAP-TTLS (with tunnelled EAP or PAP/CHAP/MSCHAP/MSCHAPV2
#          authentication)
#   If not set, all compiled in methods are allowed.
#
# identity: Identity string for EAP
# anonymous_identity: Anonymous identity string for EAP (to be used as the
#   unencrypted identity with EAP types that support different tunnelled
#   identity, e.g., EAP-TTLS)
# password: Password string for EAP
# ca_cert: File path to CA certificate file (PEM/DER). This file can have
# one or more trusted CA certificates. If ca_cert and ca_path are not
# included, server certificate will not be verified. This is insecure and
# a trusted CA certificate should always be configured when using
# EAP-TLS/TTLS/PEAP. Full path should be used since working directory may
# change when wpa_supplicant is run in the background.
# On Windows, trusted CA certificates can be loaded from the system
# certificate store by setting this to cert_store://<name>, e.g.,
# ca_cert="cert_store://CA" or ca_cert="cert_store://ROOT".
# Note that when running wpa_supplicant as an application, the user
# certificate store (My user account) is used, whereas computer store
# (Computer account) is used when running wpasvc as a service.
# ca_path: Directory path for CA certificate files (PEM). This path may
# contain multiple CA certificates in OpenSSL format. Common use for this
# is to point to system trusted CA list which is often installed into
# directory like /etc/ssl/certs. If configured, these certificates are
# added to the list of trusted CAs. ca_cert may also be included in that
# case, but it is not required.
# client_cert: File path to client certificate file (PEM/DER)
# Full path should be used since working directory may change when
# wpa_supplicant is run in the background.
# Alternatively, a named configuration blob can be used by setting this
# to blob://<blob name>.
# private_key: File path to client private key file (PEM/DER/PFX)
#   When PKCS#12/PFX file (.p12/.pfx) is used, client_cert should be
#   commented out. Both the private key and certificate will be read from
#   the PKCS#12 file in this case. Full path should be used since working
#   directory may change when wpa_supplicant is run in the background.
#   Windows certificate store can be used by leaving client_cert out and
#   configuring private_key in one of the following formats:
```

```
#  cert://substring_to_match
#  hash://certificate_thumbprint_in_hex
#  for example: private_key="hash://63093aa9c47f56ae88334c7b65a4"
#  Note that when running wpa_supplicant as an application, the user
#  certificate store (My user account) is used, whereas computer store
#  (Computer account) is used when running wpasvc as a service.
#  Alternatively, a named configuration blob can be used by setting this
#  to blob://<blob name>.
# private_key_passwd: Password for private key file (if left out, this
# will be asked through control interface)
# dh_file: File path to DH/DSA parameters file (in PEM format)
#  This is an optional configuration file for setting parameters for an
#  ephemeral DH key exchange. In most cases, the default RSA
#  authentication does not use this configuration. However, it is
# possible setup RSA to use ephemeral DH key exchange. In addition,
# ciphers with DSA keys always use ephemeral DH keys. This can be used to
# achieve forward secrecy. If the file is in DSA parameters format, it
# will be automatically converted into DH params.
# subject_match: Substring to be matched against the subject of the
#  authentication server certificate. If this string is set, the server
#  sertificate is only accepted if it contains this string in the subject.
#  The subject string is in following format:
#  /C=US/ST=CA/L=San Francisco/CN=Test AS/emailAddress=as@example.com
# altsubject_match: Semicolon separated string of entries to be matched
# against the alternative subject name of the authentication server
# certificate.
#  If this string is set, the server sertificate is only accepted if it
#  contains one of the entries in an alternative subject name extension.
#  altSubjectName string is in following format: TYPE:VALUE
#  Example: EMAIL:server@example.com
#  Example: DNS:server.example.com;DNS:server2.example.com
#  Following types are supported: EMAIL, DNS, URI
# phase1: Phase1 (outer authentication, i.e., TLS tunnel) parameters
#  (string with field-value pairs, e.g., "peapver=0" or
#  "peapver=1 peaplabel=1")
#  'peapver' can be used to force which PEAP version (0 or 1) is used.
#  'peaplabel=1' can be used to force new label, "client PEAP encryption",
#  to be used during key derivation when PEAPv1 or newer. Most existing
#  PEAPv1 implementation seem to be using the old label, "client EAP
#  encryption", and wpa_supplicant is now using that as the default value.
#  Some servers, e.g., Radiator, may require peaplabel=1 configuration to
#  interoperate with PEAPv1; see eap_testing.txt for more details.
#  'peap_outer_success=0' can be used to terminate PEAP authentication on
#  tunneled EAP-Success. This is required with some RADIUS servers that
#  implement draft-josefsson-pppext-eap-tls-eap-05.txt (e.g.,
#  Lucent NavisRadius v4.4.0 with PEAP in "IETF Draft 5" mode)
#  include_tls_length=1 can be used to force wpa_supplicant to include
#  TLS Message Length field in all TLS messages even if they are not
#  fragmented.
#  sim_min_num_chal=3 can be used to configure EAP-SIM to require three
#  challenges (by default, it accepts 2 or 3)
# phase2: Phase2 (inner authentication with TLS tunnel) parameters
#  (string with field-value pairs, e.g., "auth=MSCHAPV2" for EAP-PEAP or
#  "autheap=MSCHAPV2 autheap=MD5" for EAP-TTLS)
# Following certificate/private key fields are used in inner Phase2
# authentication when using EAP-TTLS or EAP-PEAP.
# ca_cert2: File path to CA certificate file. This file can have one or
# more trusted CA certificates. If ca_cert2 and ca_path2 are not included,
#  server certificate will not be verified. This is insecure and a trusted
#  CA certificate should always be configured.
# ca_path2: Directory path for CA certificate files (PEM)
# client_cert2: File path to client certificate file
# private_key2: File path to client private key file
# private_key2_passwd: Password for private key file
# dh_file2: File path to DH/DSA parameters file (in PEM format)
# subject_match2: Substring to be matched against the subject of the
#  authentication server certificate.
# altsubject_match2: Substring to be matched against the alternative
# subject name of the authentication server certificate.
#
# fragment_size: Maximum EAP fragment size in bytes (default 1398).
#  This value limits the fragment size for EAP methods that support
#  fragmentation (e.g., EAP-TLS and EAP-PEAP). This value should be set
#  small enough to make the EAP messages fit in MTU of the network
#  interface used for EAPOL. The default value is suitable for most
#  cases.
#
# EAP-PSK variables:
# eappsk: 16-byte (128-bit, 32 hex digits) pre-shared key in hex format
# nai: user NAI
#
# EAP-PAX variables:
# eappsk: 16-byte (128-bit, 32 hex digits) pre-shared key in hex format
#
# EAP-SAKE variables:
# eappsk: 32-byte (256-bit, 64 hex digits) pre-shared key in hex format
#  (this is concatenation of Root-Secret-A and Root-Secret-B)
# nai: user NAI (PEERID)
#
# EAP-GPSK variables:
# eappsk: Pre-shared key in hex format (at least 128 bits, i.e., 32 hex
digits)
```

```
# nai: user NAI (ID_Client)
#
# EAP-FAST variables:
# pac_file: File path for the PAC entries. wpa_supplicant will need to be
# able to create this file and write updates to it when PAC is being
#  provisioned or refreshed. Full path to the file should be used since
#  working directory may change when wpa_supplicant is run in the
#  background. Alternatively, a named configuration blob can be used by
#  setting this to blob://<blob name>
# phase1: fast_provisioning=1 option enables in-line provisioning of EAP-
# FAST credentials (PAC)
#
# wpa_supplicant supports number of "EAP workarounds" to work around
# interoperability issues with incorrectly behaving authentication
# servers.
# These are enabled by default because some of the issues are present in
# large number of authentication servers. Strict EAP conformance mode can
# be configured by disabling workarounds with eap_workaround=0.

# Example blocks:

# Simple case: WPA-PSK, PSK as an ASCII passphrase, allow all valid
# ciphers
network={
   ssid="simple"
   psk="very secret passphrase"
   priority=5
}

# Same as previous, but request SSID-specific scanning (for APs that r
# eject broadcast SSID)
network={
   ssid="second ssid"
   scan_ssid=1
   psk="very secret passphrase"
   priority=2
}

# Only WPA-PSK is used. Any valid cipher combination is accepted.
network={
   ssid="example"
   proto=WPA
   key_mgmt=WPA-PSK
   pairwise=CCMP TKIP
   group=CCMP TKIP WEP104 WEP40
   psk=06b4be19da289f475aa46a33cb793029d4ab3db7a23ee92382eb0106c72ac7bb
```

```
    priority=2
}

# Only WPA-EAP is used. Both CCMP and TKIP is accepted. An AP that used
# WEP104 or WEP40 as the group cipher will not be accepted.
network={
    ssid="example"
    proto=RSN
    key_mgmt=WPA-EAP
    pairwise=CCMP TKIP
    group=CCMP TKIP
    eap=TLS
    identity="user@example.com"
    ca_cert="/etc/cert/ca.pem"
    client_cert="/etc/cert/user.pem"
    private_key="/etc/cert/user.prv"
    private_key_passwd="password"
    priority=1
}

# EAP-PEAP/MSCHAPv2 configuration for RADIUS servers that use the new
# peaplabel (e.g., Radiator)
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=PEAP
    identity="user@example.com"
    password="foobar"
    ca_cert="/etc/cert/ca.pem"
    phase1="peaplabel=1"
    phase2="auth=MSCHAPV2"
    priority=10
}

# EAP-TTLS/EAP-MD5-Challenge configuration with anonymous identity for the
# unencrypted use. Real identity is sent only within an encrypted TLS
# tunnel.
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=TTLS
    identity="user@example.com"
    anonymous_identity="anonymous@example.com"
    password="foobar"
    ca_cert="/etc/cert/ca.pem"
    priority=2
```

```
}

# EAP-TTLS/MSCHAPv2 configuration with anonymous identity for the
# unencrypted use. Real identity is sent only within an encrypted TLS
# tunnel.
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=TTLS
    identity="user@example.com"
    anonymous_identity="anonymous@example.com"
    password="foobar"
    ca_cert="/etc/cert/ca.pem"
    phase2="auth=MSCHAPV2"
}

# WPA-EAP, EAP-TTLS with different CA certificate used for outer and inner
# authentication.
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=TTLS
    # Phase1 / outer authentication
    anonymous_identity="anonymous@example.com"
    ca_cert="/etc/cert/ca.pem"
    # Phase 2 / inner authentication
    phase2="autheap=TLS"
    ca_cert2="/etc/cert/ca2.pem"
    client_cert2="/etc/cer/user.pem"
    private_key2="/etc/cer/user.prv"
    private_key2_passwd="password"
    priority=2
}

# Both WPA-PSK and WPA-EAP is accepted. Only CCMP is accepted as pairwise
# and group cipher.
network={
    ssid="example"
    bssid=00:11:22:33:44:55
    proto=WPA RSN
    key_mgmt=WPA-PSK WPA-EAP
    pairwise=CCMP
    group=CCMP
    psk=06b4be19da289f475aa46a33cb793029d4ab3db7a23ee92382eb0106c72ac7bb
}
```

```
# Special characters in SSID, so use hex string. Default to WPA-PSK, WPA-
# EAP and all valid ciphers.
network={
    ssid=00010203
    psk=000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
}


# EAP-SIM with a GSM SIM or USIM
network={
    ssid="eap-sim-test"
    key_mgmt=WPA-EAP
    eap=SIM
    pin="1234"
    pcsc=""
}


# EAP-PSK
network={
    ssid="eap-psk-test"
    key_mgmt=WPA-EAP
    eap=PSK
    identity="eap_psk_user"
    eappsk=06b4be19da289f475aa46a33cb793029
    nai="eap_psk_user@example.com"
}


# IEEE 802.1X/EAPOL with dynamically generated WEP keys (i.e., no WPA)
# using EAP-TLS for authentication and key generation; require both
# unicast and broadcast WEP keys.
network={
    ssid="1x-test"
    key_mgmt=IEEE8021X
    eap=TLS
    identity="user@example.com"
    ca_cert="/etc/cert/ca.pem"
    client_cert="/etc/cert/user.pem"
    private_key="/etc/cert/user.prv"
    private_key_passwd="password"
    eapol_flags=3
}


# LEAP with dynamic WEP keys
```

```
network={
    ssid="leap-example"
    key_mgmt=IEEE8021X
    eap=LEAP
    identity="user"
    password="foobar"
}

# EAP-FAST with WPA (WPA or WPA2)
network={
    ssid="eap-fast-test"
    key_mgmt=WPA-EAP
    eap=FAST
    anonymous_identity="FAST-000102030405"
    identity="username"
    password="password"
    phase1="fast_provisioning=1"
    pac_file="/etc/wpa_supplicant.eap-fast-pac"
}

network={
    ssid="eap-fast-test"
    key_mgmt=WPA-EAP
    eap=FAST
    anonymous_identity="FAST-000102030405"
    identity="username"
    password="password"
    phase1="fast_provisioning=1"
    pac_file="blob://eap-fast-pac"
}

# Plaintext connection (no WPA, no IEEE 802.1X)
network={
    ssid="plaintext-test"
    key_mgmt=NONE
}


# Shared WEP key connection (no WPA, no IEEE 802.1X)
network={
    ssid="static-wep-test"
    key_mgmt=NONE
    wep_key0="abcde"
    wep_key1=0102030405
    wep_key2="1234567890123"
    wep_tx_keyidx=0
    priority=5
}


# Shared WEP key connection (no WPA, no IEEE 802.1X) using Shared Key
# IEEE 802.11 authentication
network={
    ssid="static-wep-test2"
    key_mgmt=NONE
    wep_key0="abcde"
    wep_key1=0102030405
    wep_key2="1234567890123"
    wep_tx_keyidx=0
    priority=5
    auth_alg=SHARED
}


# IBSS/ad-hoc network with WPA-None/TKIP.
network={
    ssid="test adhoc"
    mode=1
    frequency=2412
    proto=WPA
    key_mgmt=WPA-NONE
    pairwise=NONE
    group=TKIP
    psk="secret passphrase"
}


# Catch all example that allows more or less all configuration modes
network={
    ssid="example"
    scan_ssid=1
    key_mgmt=WPA-EAP WPA-PSK IEEE8021X NONE
    pairwise=CCMP TKIP
    group=CCMP TKIP WEP104 WEP40
    psk="very secret passphrase"
    eap=TTLS PEAP TLS
    identity="user@example.com"
    password="foobar"
    ca_cert="/etc/cert/ca.pem"
    client_cert="/etc/cert/user.pem"
    private_key="/etc/cert/user.prv"
    private_key_passwd="password"
    phase1="peaplabel=0"
}

# Example of EAP-TLS with smartcard (openssl engine)
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=TLS
    proto=RSN
    pairwise=CCMP TKIP
    group=CCMP TKIP
    identity="user@example.com"
    ca_cert="/etc/cert/ca.pem"
    client_cert="/etc/cert/user.pem"

    engine=1

    # The engine configured here must be available. Look at
    # OpenSSL engine support in the global section.
    # The key available through the engine must be the private key
    # matching the client certificate configured above.

    # use the opensc engine
    #engine_id="opensc"
    #key_id="45"

    # use the pkcs11 engine
    engine_id="pkcs11"
    key_id="id_45"

    # Optional PIN configuration; this can be left out and PIN will be
    # asked through the control interface
    pin="1234"
}

# Example configuration showing how to use an inlined blob as a CA
# certificate data instead of using external file
network={
    ssid="example"
    key_mgmt=WPA-EAP
    eap=TTLS
    identity="user@example.com"
    anonymous_identity="anonymous@example.com"
    password="foobar"
    ca_cert="blob://exampleblob"
    priority=20
```

```
}

blob-base64-exampleblob={
SGVsbG8gV29ybGQhCg==
}


# Wildcard match for SSID (plaintext APs only). This example select any
# open AP regardless of its SSID.
network={
    key_mgmt=NONE
}
```

TCP/IP Networking

For legacy support the V/OS terminal supports the network configuration and function calls in this chapter.

The V/OS terminals fully support the Linux sockets interface for client and server network programming. The networking API is contained in the svc.net.h header file and the libvfisvc shared library.

"localhost" (loopback) is supported only on the Ethernet port, the eth0 device and localhost, with a standard IP address of 127.0.0.1. Wireless support is subject to hardware availability.

Network Configuration

Configuration variables are read for network configuration in bringing the interface up at boot time and user control. Tune network parameters using `putSysctl()` to optimize performance and for security purposes.

Environment Variables

The following table of configuration variables is read by the system on power up/ reboot and in bringing the interface up. Either the `*DHCP` or `*IFCONFIG` environment variable must be defined to bring up the `eth0` network interface.

The configuration variables in Table 9 must be set in the usr1 account.

Table 9 Network Configuration Environment Variables

Variable Name	Values	Definition
`*NET`	"e","E" (default) or ""n","N"	If "e", "E" or not defined, use Ethernet (wired) for network interface. This applies to `netUp( )`. "n", "N" specifies no networking (disabled).
`*DHCP`	1	If `*DHCP` is present and the system supports Ethernet, then it attempts to initialize its connection using DHCP.
`*DNS1`	IP Address in the form xxx.xxx.xxx.xxx	If not DHCP, `*DNS1` is used as the name server IP address.
`*DNS2`	IP Address in the form xxx.xxx.xxx.xxx	If not DHCP, `*DNS2` is used as the name server IP address.

Table 9 Network Configuration Environment Variables (continued)

Variable Name	Values	Definition
`*IFCONFIG`	Per Linux – No need to set MAC address as the system will do this for you. Typically used: “eth0 xxx.xxx.xxx.xxx netmask xxx.xxx.xxx.xxx”	If not DHCP, use `*IFCONFIG` to set the following parameters: • host static IP address • netmask • broadcast (optional)
`*GATEWAY`	IP Address in the form xxx.xxx.xxx.xxx	Defines the address of the gateway (router).

NOTE

Possible future enhancement:

For optimum performance, modify network parameters using `putSysctl()`. For example, the `net.ipv4.tcp_syn_retries` value is set to 5 by default, or about 3 minutes to connect to the remote host. Lower the retries value to a reasonable timeout. (svcpak-016-01 and forward, this value is set to 1 by default, or 9 seconds.)

Retries	Timeout
0	03 seconds
1	09 seconds
2	21 seconds
3	45 seconds
4	93 seconds

Network Functions

This section lists supported network function calls.

net_interfaceUp()

Activates the network interface specified by `*NET` in `config.usr1`. If the activate attribute in the XML file is set to 1, bring up the network, otherwise do nothing.

NOTE This function uses the Network Configuration environment variables and either the `*DHCP` or `*IFCONFIG` variable must exist. `*DHCP` takes precedence if both variables exist.

Prototype

```
int net_interfaceUp(char *p_iface, int ipVersion );
```

Parameters

`p_iface`	The interface to configure. Valid values = eth0.
`ipVersion`	Set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Bad input parameters.
- ENETDOWN – Network interface not online (for example, the Ethernet cable is not plugged in).
- EBADF – Error set when reading the default file.
- ESRCH – Error set while changing the Ethernet speed.

net_interfaceDown()

Deactivates the specified network interface. The activate parameter in the XML file is not impacted.

Prototype

```
int net_interfaceDown(char *p_iface, int ipVersion);
```

Parameters

`p_iface`	The interface to deactivate. Valid values = eth0.
`ipVersion`	Set to `NET_IP_V4`.

Return Values

= 0 — Success

-1 — On error with errno set.

- ENONET – No network interface is active.
- ENETDOWN – The network interface is not online (for example, the Ethernet cable not plugged in).
- EBADF – Error when reading the default configuration file.

net_interfaceStatus()

Reads the interface link status and returns the current Interface IP information. The MAC address is the interface status. The activate parameter updates according to its XML file value.

Prototype

```
struct netIfconfig net_interfaceStatus(char *p_iface );
```

Parameters

`p_iface`	The network interface to read.

Return Values

`struct netIfconfig` IP information of the network interface.

-1 on error with errno set.

- ENONET – The network is down.
- ENETDOWN – The network interface is not online (for example, the Ethernet cable not plugged in or the WIFI network is disconnected).
- ENOTCONN – The device is not on (WIFI module is down).

net_networkUp()

Brings up all network interfaces set by parameters in the default file. If the activate attribute in the XML file is set to 1 for a specific interface, bring up the network, otherwise do nothing for that interface.

Prototype

```
int net_networkUp(int ipVersion);
```

Parameters

`ipVersion`	Set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Bad input parameters.
- ENETDOWN – The interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – Error when reading the default file.
- ESRCH – Error while changing the Ethernet speed.
- ENOTCONN – Failed to configure both Ethernet and WIFI.

net_networkDown()

Closes all network IP interfaces. The `activate` parameter in the XML file is not impacted.

Prototype

```
int net_networkDown(int ipVersion);
```

Parameters

`ipVersion`	Set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- ENONET – No network interface up.
- ENETDOWN – The network interface is not online (for example, the Ethernet cable is not plugged in).
- EBADF – Error when reading the default configuration file.

net_interfaceStatusList()

Reads the list of all active network interface configurations.

Prototype

```
struct netIfconfigList net_interfaceStatusList(void);
```

Return Values

`struct netIfconfigList` is the list of network interface configurations.

= 0	Success

Error with errno set.

- ENONET – No network interface up.

net_getTable()

Reads the routing table.

Prototype

```
struct netRouteTable net_getTable(void);
```

Parameters

`ipVersion`	Set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EBADF – Error while reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.

net_getRoutes()

Reads all routes to a specific destination.

Prototype

```
struct netRouteTable net_getRoutes(char *p_dest);
```

Parameters

`p_dest`	Destination to load the routes.

Return Values

= 0	Success
-1	Error with errno set.

- EBADF – Error while reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found in the routing table.

net_addHostRoute()

Adds a route to the specified host.

Prototype

```
int net_addHostRoute(char *p_dest, char *p_gateway, char *p_iface);
```

Parameters

`p_dest`	Destination host IP address.
`p_gateway`	Gateway to this host.
`p_iface`	IP interface (for example, eth0, wlan0, and so on).

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EFAULT – Invalid IP address passed.
- EBADF – Error when reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found.
- EHOSTUNREACH – Host name resolution error (host unreachable).

net_delHostRoute()

Deletes a route to the specified host.

Prototype

```
int net_delHostRoute(char *p_dest);
```

Parameters

`p_dest`	Destination host IP address.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EFAULT – Invalid IP address passed.
- EBADF – Error when reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found.

net_addNetRoute()

Adds a route to a network.

Prototype

```
int net_addNetRoute( char *p_net, char *p_netmask, char *p_gateway,
char *p_iface );
```

Parameters

`p_net`	Network IP address.
`p_netmask`	Network mask.
`p_gateway`	Gateway to use.
`p_iface`	IP interface (for example, eth0, wlan0, and so on).

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EFAULT – Invalid IP address passed.
- EBADF – Error when reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found.
- ENETDOWN – Network interface not found in the active interfaces list.

net_delNetRoute()

Deletes a route to a network.

Prototype

```
int net_delNetRoute(char *p_net, char *p_netmask, char *p_gateway,
char *p_iface);
```

Parameters

`p_net`	Network IP address.
`p_netmask`	Network mask.
`p_gateway`	Gateway to use.
`p_iface`	IP interface (for example, eth0, wlan0, and so on).

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EFAULT – Invalid IP address passed.
- EBADF – Error when reading the routing table.
- ENOEXEC – Routing table column missing.
- EDESTADDRREQ – No route found.
- ENETDOWN – Network interface not found in the active interfaces list.

net_autoSetDefaultRoute()

Selects the default route according to the priority policy (eth0, wlan0).

Prototype

```
int net_autoSetDefaultRoute(void);
```

Return Values

= 0	Success
	Error with errno set.

- ENONET – No active network interface found.

net_setDefaultRoute()

Sets the default route on a specified network interface.

Prototype

```
int net_setDefaultRoute(char *p_iface);
```

Parameters

`p_iface`	IP interface for the default route (for example, eth0, wlan0, and so on).

Return Values

= 0	Success

Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Error when reading the routing table.
- ENONET – No active network interface found.

net_getDefaultRouteInfo()

Gets the default route network interface name.

Prototype

```
char * net_getDefaultRouteInfo(void);
```

Return Values

Success = Interface name (eth0, wlan0) of the default route.

Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Error when reading the routing table.
- ENONET – No active network interface found.

net_startDHCP()

Starts a DHCP client for a specified network interface. If DHCP already up, do nothing. If static IP up, start DHCP and switch to dynamic configuration.

Prototype

```
int net_startDHCP(char *p_iface, int ipVersion);
```

Parameters

`p_iface`	IP interface for the default route (for example, eth0, wlan0, and so on).
`ipVersion`	Must be set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENETDOWN – Network interface not found in the active interfaces list.

net_renewDHCP()

Causes UDHCPC to renew the current lease; if it does not have one, obtains a new lease.

Prototype

```
int net_renewDHCP(char *p_iface, int ipVersion);
```

Parameters

`p_iface`	IP interface to request the renewal (for example, eth0, wlan0, and so on).
`ipVersion`	Must be set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENETDOWN – Network interface not found in the active interfaces list.
- ENONET – No active network interface found.

net_releaseDHCP()

Causes UDHCPC to release the current lease. To get a new lease, call net_renewDHCP().

Prototype

```
int net_releaseDHCP(char *p_iface, int ipVersion);
```

Parameters

`p_iface`	IP interface to release (for example, eth0, wlan0, and so on).
`ipVersion`	Must be set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENETDOWN – Network interface not found in the active interfaces list.
- ENONET – No active network interface found.

net_stopDHCP()

Stops the DHCP client for the specified interface.

Prototype

```
int net_stopDHCP(char *p_iface,  int ipVersion);
```

Parameters

`p_iface`	IP interface to stop DHCP on (for example, eth0, wlan0, and so on).
`ipVersion`	Must be set to `NET_IP_V4`.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENETDOWN – Network interface not found in the active interfaces list.
- ENONET – No active network interface found.

net_setDNS()

Specifies the DNS servers for the specified interface.

Prototype

```
int net_setDNS(char *p_iface, char *p_dns1, char *p_dns2);
```

Parameters

`p_iface`	IP interface to set DNS on (for example, eth0, wlan0, and so on).
`p_dns1`	Domain name server 1.
`p_dns2`	Domain name server 2.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EFAULT – Invalid IP address provided.
- EDESTADDRREQ – No route found.
- ENETDOWN – Network interface not found in the active interfaces list.

net_addRouteFromXML()

Adds static routes from an XML file.

Prototype

```
int net_addRouteFromXML(char *p_fileName);
```

Parameters

`p_fileName`	XML file that contains static routes.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Error when reading the XML file.

net_nsLookup()

Performs DNS resolution on the specified interface.

Prototype

```
struct netIfconfig net_nsLookup(char *p_dnsName, char *p_iface);
```

Parameters

`p_dnsName`	Domain name server name to resolve.
`p_iface`	IP interface for domain name resolution.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Error when reading the DNS file.
- EFAULT – The parameter provided is not an IP address.
- ENONET – IP interface up.
- ENOLINK – Error while creating the DNS link file.

net_setNTP ()

Sets the NTP (Network Time Protocol) host name.

Prototype

```
int net_setNTP(char *p_hostname);
```

Parameters

`p_hostname`	Network Time Protocol host name. If `p_hostname` is NULL, pool.ntp.org is used.

Return Values

= 0	Success
-1	Error with errno set. • EINVAL – Invalid input parameters. • EFAULT – Invalid host name.

Wireless Network Interface Functions

These functions tune and configure the wireless device used for network communication. These functions are for use in an application in special architecture/configuration. Otherwise, the application should use the `wlan.conf` file (located in `/home/usr1/`).

NOTE

The wireless network interface is subject to hardware availability.

`ssid`	{0~z,1~32 ASCII characters} Place in double quotes
`networktype`	{0=MANAGED,1=ADHOC} infra == managed
`countryregion`	{0~7} default=0
`countryregionaband`	{0~10} default=0
`channel`	{1~14} range depends on CountryRegion default=1
`wirelessmode`	{0~4} default=0
`txrate`	{0:Auto, 1:1Mbps, 2:2Mbps, 3:5.5Mbps, 4:11Mbps, 5:6Mbps, 6:9Mbps, 7:12Mbps,8:18Mbps, 9:24Mbps, 10:36Mbps, 11:48Mbps, 12:54Mbps} default=0
`bgprotection`	{0:Auto, 1:Always on, 2:Always off} default=0
`txpreamble`	{0:Preamble Long, 1:Preamble Short, 2:Auto} default=2
`rts`	{0=off,1=auto,2=fixed} default=0 (rts threshold)
`frag`	{0=off,1=auto,2=fixed} default=0 (fragment threshold)
`rtsthreshold`	{1~2347} default=2347
`fragthreshold`	{256~2346} default=2346
`txburst`	{0:Disable, 1:Enable} default=0
`psmode`	{0=CAM, 1=MAX_PSP, 2=FAST_PSP} default=0 (Power Savings Mode)
`adhocmode`	{0: WIFI mode (11b rates only), 1: b/g mixed, 2: 11g only, 3: 11a only} default=0
`authmode`	{0=OPEN,1=SHARED,2=WEPAUTO,3=WPAPSK, 4=WPA2PSK, 5=WPANONE}
`encryptype`	{0=NONE,1=WEP,2=TKIP,3=AES}
`defaultwepkey`	{1~4} default=1

Enclose the following in double quotes.

`wepkey1`	64-bit - {5 ASCII characters(or 10 hex #'s) or 128-bit - (13 ASCII characters (or 26 hex #'s)
`wepkey2`	64bit - {5 ascii characters(or 10 hex #'s) or 128bit - (13 ascii characters(or 26 hex #'s)
`wepkey3`	64bit - {5 ascii characters(or 10 hex #'s) or 128bit - (13 ascii characters(or 26 hex #'s)
`wepkey4`	64bit - {5 ascii characters(or 10 hex #'s) or 128bit - (13 ascii characters(or 26 hex #'s)
`wpapasswd`	{8~63 ascii or 64 hex #'s}
`dhcp`	{0=static, 1=DHCP} default=1 addressing mode
`ipaddr`	IP address in the form "xxx.xxx.xxx.xxx"
`netmask`	network mask in the form "xxx.xxx.xxx.xxx"
`gateway`	IP address of gateway in the form "xxx.xxx.xxx.xxx"
`dns1`	IP address if DNS server in the form "xxx.xxx.xxx.xxx"
`dns2`	IP address if DNS server in the form "xxx.xxx.xxx.xxx"

net_wifiStart()

Starts the module and Wi-Fi Supplicant specified in `startwpa`.

Prototype

```
int net_wifiStart(int startwpa);
```

Parameters

`startwpa`	1 = Power up the module and start the WPA Supplicant.
	0 = Only power up the module.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Wi-Fi configuration file not found.

net_wifiStop()

Stops the module and Wi-Fi Supplicant and module specified in `stopmodule`.

Prototype

```
int net_wifiStop( int stopmodule);
```

Parameters

`stopmodule`	1 = Stops WPA Supplicant and Wi-Fi module.
	0 = Stops the WPA Supplicant only.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- EBADF – Wi-Fi configuration file not found.

net_wifiSiteSurvey()

Gets the Wi-Fi site survey list.

Prototype

```
struct netSiteSurveyList net_wifiSiteSurvey(void);
```

Return Values

= 0	Success; `struct netIfconfigList` returns a list of discovered Wi-Fi sites.
-1	Error with errno set.

- ENOEXEC – Scan command not ready.
- ETIMEDOUT – No Wi-Fi network found.
- EBADF – Error while parsing scan results.
- ENOTCONN – The device not on (Wi-Fi module down).

net_wifiInfo()

Gets Wi-Fi information for the specified Wi-Fi interface (wlan0).

Prototype

```
struct netWifiInfo net_wifiInfo(char *p_iface);
```

Parameters

`p_iface`	Wi-Fi interface; valid value is wlan0.

Return Values

= 0	Success; `struct netWifiInfo` structure containing connected Wi-Fi information.
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENETDOWN – No Wi-Fi network found.
- ENOEXEC – Command execution error.
- ENOTCONN – The device not on (Wi-Fi module down).

Supported Network Programs

The V/OS terminal API includes support for common networking programs such as ping and FTP file transfers.

For FTP file transfers, the FTP server must support Passive mode for the data socket connection and the file transfers are always in Binary mode. The FTP server must support the following FTP commands:

- USER
- PASS
- TYPE I
- PASV
- RETR
- CWD
- STOR
- QUIT

net_ping()

Removes all stored DNS resolution results for the specified server.

Prototype

```
struct netPingInfo net_ping(char *p_host, unsigned char pingCount);
```

Parameters

`p_host`	Server ping destination (IPv4 or IPv6).
`pingCount`	Number of times to retry ping.

Return Values

= 0	Success
-1	Error with errno set.

- EINVAL – Invalid input parameters.
- ENOMSG – No packet transmitted.
- EHOSTUNREACH – No packet received.
- EBADMSG – Number of received packets lower than transmitted.
- EFAULT – Invalid `p_host`.

ftpGet()

Transfers the file from the FTP server to the terminal.

NOTE This function logs in to the FTP server, retrieves the file, and logs off.

Prototype

```
int result = netGetConfig(net_conf_t *net);
```

Parameters

`net`	Pointer to the FTP parameter structure	

`typedef struct{`
`char ftpHost[96];`
`char port[8];`
`char userID[32];`
`char password[32];`
`char localFile[96];`
`char`
`remoteFile[96];`
`char errorMsg[128];`
`} ftp_parm_t;`
	`ftpHost`	In the “xxx.xxx.xxx.xxx” IP address format or the fully qualified domain name format ftp.site.com

	`port`	FTP port. Defaults to 21 if not specified.
	`userID`	FTP user name
	`password`	FTP password
	`localFile`	Local file name, with directory path (relative to `“/home/usr1/”`)
	`remoteFile`	Remote file name: • For `ftpGet()`, this is the directory pathname and filename. • For `ftpPut()`, this is the directory pathname only.
	`errorMsg`	Returns the verbose error message

Return Values

= 0	Success; network interface is up.
< 0	Error

ftpPut()

Transfers a file from the terminal to the FTP server.

NOTE

This function logs in to the FTP server, sends the file to the server, and logs off.

Prototype

```
int result = ftpPut(FTP_parm_t *ftp)
```

Parameters

`ftp`	Pointer to FTP parameter structure (`see ftpGet()`). The `remoteFile` field is the directory path name only. The resultant filename is the same as the source filename.

Return Values

= 0	Success
< 0	Error

CHAPTER 23

Bluetooth Service

Some V/OS-based terminals support Bluetooth devices.

Bluetooth Functions

This section lists the calls for the Bluetooth service.

bluetooth_getVersion()

Retrieves the version of the Bluetooth device firmware.

Prototype

```
struct version bluetooth_getVersion(void);
```

XML Prototype

```
<svc_cmd> <bluetooth> <getVersion/> </bluetooth> </svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth> <getVersion>
<return type="container">
<major>int</major>
<minor>int</minor>
<maint>int</maint>
<build type="string">string [max len 16]</build>

</return> </getVersion> </bluetooth> </svc_rtn>
```

bluetooth_scan()

Scans for nearby Bluetooth devices.

Prototype

```
int bluetooth_scan(void);
```

Return Values

```
<svc_rtn> <bluetooth> <scan> <return>int</return> </scan> </bluetooth>
</svc_rtn>
```

bluetooth_scanResults()

Returns the results of the Bluetooth device scan.

Prototype

```
struct btDeviceList bluetooth_scanResults(void);
```

XML Prototype

```
<svc_cmd> <bluetooth> <scanResults/> </bluetooth> </svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth> <scanResults>
<return type="container">
<deviceCount>int</deviceCount>
<device type="list">

<item type="container"> <name type="string">string [max len 128]</name>
<address type="string">string [max len 18]</address>
<class type="string">string [max len 3]</class>
</item> </device> </return> </scanResults> </bluetooth> </svc_rtn>
```

bluetooth_hidConnect()

Connects to an HID.

Prototype

```
int bluetooth_hidConnect(char * remoteDeviceAddr /*NULL*/);
```

XML Prototype

```
<svc_cmd> <bluetooth> <hidConnect> <remoteDeviceAddr type="string">
string [default:NULL]</remoteDeviceAddr> </hidConnect> </bluetooth>
</svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth>
```

bluetooth_hidDisconnect()

Disconnect from an HID.

Prototype

```
int bluetooth_hidDisconnect(char * remoteDeviceAddr /*NULL*/);
```

XML Prototype

```
<svc_cmd> <bluetooth> <hidDisconnect> <remoteDeviceAddr
type="string">string [default:NULL]</remoteDeviceAddr> </hidDisconnect>
</bluetooth> </svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth> <hidDisconnect> <return>int</return>
</hidDisconnect> </bluetooth> </svc_rtn>
```

bluetooth_getConnectedHids()

Retrieves a list of connected HIDs.

Prototype

```
struct btDeviceList bluetooth_getConnectedHids(void);
```

XML Prototype

```
<svc_cmd> <bluetooth> <getConnectedHids/> </bluetooth> </svc_cmd>
```

Return Values

Data Structure Index

This is the Bluetooth data structure:

```
<svc_rtn> <bluetooth> <getConnectedHids> <return type="container">
<deviceCount>int</deviceCount> <device type="list">
<item type="container"> <name type="string">string [max len 128]</name>
<address type="string">string [max len 18]</address>
<class type="string">string [max len 3]</class> </item> </device>
</return> </getConnectedHids> </bluetooth> </svc_rtn>
```

Data Structures

Data structures are:

- `btConf` – Local Bluetooth configuration structure
- `btDevice` – Bluetooth information structure
- `btDeviceList` – List Bluetooth devices found during scan

File List

This section lists all files in the local Bluetooth configuration structure.

```
#include <svc_bluetooth.h>
```

Data Fields

- `char name [128]` – Bluetooth device name.
- `char address [18]` – Bluetooth device address.
- `char discoverable` – Bluetooth discovery mechanism.

btConfig::address[]

Configures the Bluetooth device address in the format xx:xx:xx:xx:xx:xx

Prototype

```
char btConfig::address[18]
```

btConf::discoverable

Retrieves the Bluetooth discovery call.

Prototype

```
char btConf::discoverable
```

btConf::name[]

Sets the Bluetooth device name.

Prototype

```
char btConf::name[128]
```

The documentation for this struct was generated from svc_bluetooth.h.

btDevice Struct Reference

Bluetooth info structure is

```
#include <svc_bluetooth.h>
```

Data Fields

char name [128] – Bluetooth device name.

char address [18] – Bluetooth device address.

char class [3] – Bluetooth device class.

btDevice::address[]

Retrieves the Bluetooth device address in the format xx:xx:xx:xx:xx:xx

Prototype

```
char btDevice::address[18]
```

btDevice::class[]

Retrieves the Bluetooth device class.

Prototype

```
char btDevice::class[3]
```

btDevice::name[]

Retrieves the Bluetooth device name.

```
char btDevice::name[128]
```

Documentation for this structure is generated from `svc_bluetooth.h`.

btDeviceList Struct Reference

Return a list of Bluetooth devices found during a scan.

```
#include <svc_bluetooth.h>
```

Data Fields

```
int deviceCount();
struct btDevice * device
```

Returns the number of Bluetooth devices found.

btDevice* btDeviceList::device

Return a list of Bluetooth devices found during a scan.

Prototype

```
struct btDevice* btDeviceList::device
```

btDeviceList::deviceCount

Retrieves the number of devices found.

Prototype

```
int btDeviceList::deviceCount
```

The documentation for this structure is generated from `svc_bluetooth.h`.

```
#include "svcmgrSvcDef.h"
```

Data Structures

- `btDevice` – Bluetooth info structure.
- `btDeviceList` – List Bluetooth devices found during a scan.
- `btConf` – Local Bluetooth configuration structure.

Defines

- `#define BT_PWR_DOWN 0` –Bluetooth power down.
- `#define BT_PWR_UP 1` –Bluetooth power up.

bluetooth_getVersion()

Retrieves the version of the Bluetooth service.

Prototype

```
struct version bluetooth_getVersion(void);
```

version bluetooth getVersion()

Retrieves the Bluetooth service version.

Prototype

```
struct version bluetooth getVersion(void); [read]
```

Return Values

The version structure defined in `svcmgrSvcDef.h`.

bluetooth hidConnect()

Connects to an HID device.

Prototype

```
int bluetooth hidConnect ( char * remoteDeviceAddr );
```

Return Values

Success	`struct btDeviceList` for the nearby devices.
Failure	An empty structure with errno set: • ENOENT – Bluetooth SW not found. • ENODEV – Bluetooth not available. • EINVAL – Invalid Bluetooth address. • ECOMM – Communication error with the Bluetooth daemon. • EIO – Communication error with a Bluetooth device.

bluetooth hidDisconnect()

Disconnect from an HID device.

Prototype

```
int bluetooth hidDisconnect ( char * remoteDeviceAddr );
```

Return Values

Success	`struct btDeviceList` for nearby devices.
Failure	An empty structure with errno set:

- ENOENT – Bluetooth SW not found.
- ENODEV – Bluetooth not available.
- EINVAL – Invalid Bluetooth address.
- ECOMM – Communication error with Bluetooth daemon.
- EIO – Communication error with Bluetooth device.

btDeviceList bluetooth getConnectedHids()

Retrieves list of connected HIDs.

Prototype

```
struct btDeviceList bluetooth getConnectedHids(void); [read]
```

Return Values

Success	`struct btDeviceList` for nearby devices.
Failure	An empty structure with errno set: • ENOENT –Bluetooth SW not found. • ENODEV – Bluetooth not available. • EIO – Communication error with Bluetooth device. • ENOMEM – Out of memory.

NOTE

There is no class information in `struct btDevice`.

btConf bluetooth getConfig()

Retrieves the current Bluetooth configuration.

Prototype

```
struct btConf bluetooth getConfig( void );
```

Return Values

Success	`struct btConf` - bluetooth configuration structure.
Failure	An empty structure with errno set

- ENOENT – Bluetooth SW not found
- ENODEV – Bluetooth not available
- EHOSTDOWN – Bluetooth is down

XML Prototype

```
<svc_cmd> <bluetooth> <getConfig/> </bluetooth> </svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth> <getConfig>
<return type="container"> <name type="string">string [max len 128]
</name> <address type="string">string [max len 18]</address>
<discoverable> char</discoverable>
</return> </getConfig> </bluetooth> </svc_rtn>
```

bluetooth isUp()

Bluetooth power status.

Prototype

```
int bluetooth isUp(void);
```

Return Values

= 0	Bluetooth down
1	Power is up
-1	On error with errno set:

- ENOENT – Bluetooth SW not found.
- ENODEV – Bluetooth not available.

XML Prototype

```
<svc_cmd> <bluetooth> <power> <operation>int [default:0]</operation>
</power> </bluetooth> </svc_cmd>
```

Return Values

```
<svc_rtn> <bluetooth> <power> <return>int</return> </power> </bluetooth>
</svc_rtn>
```

bluetooth power()

Switches Bluetooth power.

Prototype

```
int bluetooth power(int operation);
```

Parameters

`operation`

Return Values

= 0	Success
-1	On error with errno set:

- ENOENT – Bluetooth SW not found.
- ENODEV – Bluetooth not available.
- EINVAL – Invalid argument.
- EACCES – Permission denied.

NOTE

`BT_PWR_ON` does nothing when Bluetooth is already up (return value is still 0).

bluetooth scan()

Scan for nearby discoverable Bluetooth devices.

Prototype

```
int bluetooth scan(void);
```

Return Values

=0	Scan stared successfully
-1	On error with errno set:

- ENOENT – Bluetooth SW not found.
- ENODEV – Bluetooth not available.
- EHOSTDOWN – Bluetooth down.
- EAGAIN – System lacked necessary resources.

btDeviceList bluetooth scanResults()

Retrieve last scan results.

Prototype

```
struct btDeviceList bluetooth scanResults(void);
```

Return Values

Success	`struct btDeviceList` for nearby devices.
Failure	An empty structure with errno set:

- ENOENT – Bluetooth SW not found.
- ENODEV – Bluetooth not available.
- EINPROGRESS – Scan in progress.
- EIO – Communication error with Bluetooth device.

CHAPTER 24

Security Module

This section describes V/OS function calls related to security and cryptography. The functions are divided into two groups:

- VeriShield Security Scripts (VSS) functions related to the use of scripts to support custom key managements beyond the usual DUKPT and M/S schemes.
- Generic functions that provided services related to security and cryptography such as `DES`, `AES`, `SHA-1`, `RSA` computation support, file encryption support, random generation, tamper detection status, file authentication and OS file upgrades.
- Service switch functions related to tamper detection.

VeriShield Security Scripts Functions

The V/OS terminal supports the VeriShield Security Scripts concept as implemented in the SC 5000 PINpad and V/OS terminals. Existing scripts run on the V/OS terminal platform without requiring any modification. All VSS-related functions listed below are defined in the header file `svcsec.h`. Applications must link with the `libvfisec.so` library by using `-lvfisec`.

NOTE

Refer to document P/N 21883, *VeriShield Security Scripts*, for information implementing security scripts.

In its default configuration, the unit supports two key management schemes through the IPP emulation: DUKPT and Master/Session. Those two schemes should meet the needs of most of the customers and since they are hard coded, no customizing of the security module is required.

For customers who need more flexibility, the VeriShield Security Script feature provides support for:

- different key management schemes,
- different PIN block formats such as PVV, CVV, IBM3624,
- different encryption algorithms such as triple-DES, AES, RSA.

All the information is written in a script file (ASCII) using a `.VSS` extension. This script is processed by a PC tool and converted into a downloadable file (`*.VSO`). The download is protected by the VeriShield File Authentication (FA) module. Therefore, the VeriShield Security Script file will have to be downloaded along with its signature file generated with the VeriShield File Signature tool.

Up to eight VeriShield Security Scripts can coexist in the unit at a same time. Each script defines its independent key space and defines whether or not those keys can be loaded using the generic key loading functions (`iPS_LoadMasterClearKey()` and `iPS_LoadMasterEncKey()`).

The VSS Key Loading Key (VSS_KLK) is a double-length key. It is loaded in the clear, but can also be loaded encrypted under its previous value. Since there is no default value in the firmware for the VSS_KLK, the first time it must be loaded in the clear. In that case, other keys in the unit will be erased, so it must be loaded before all other keys. This must be done in a secure environment before deployment.

The security script's master keys can be loaded in the clear or encrypted under VSS_KLK. Loading additional keys without erasing the keys previously loaded must be done in encrypted form and therefore requires the knowledge of VSS_KLK.

Each script defines its own set of keys and also defines if the keys may be loaded with those generic key-loading functions. Some scripts may disallow their use and implement custom macro commands for key loading.

iPS_DeleteKeys()

Deletes the specified set of keys.

Prototype

```
int iPS_DeleteKeys(unsigned long ulKeyType);
```

Parameters

`ulKeyType` Indicates which set of keys to erase. Each bit corresponds to a set of keys; that is, several sets can be erased in one function call.

DEL_SYSTEM	System key (VSS_KLK)
DEL_VSS0	Keys associated to VeriShield Security Script loaded in slot #0
DEL_VSS1	Keys associated to VeriShield Security Script loaded in slot #1
DEL_VSS2	Keys associated to VeriShield Security Script loaded in slot #2
DEL_VSS3	Keys associated to VeriShield Security Script loaded in slot #3
DEL_VSS4	Keys associated to VeriShield Security Script loaded in slot #4
DEL_VSS5	Keys associated to VeriShield Security Script loaded in slot #5
DEL_VSS6	Keys associated to VeriShield Security Script loaded in slot #6
DEL_VSS7	Keys associated to VeriShield Security Script loaded in slot #7
DEL_ALL	Delete all keys in the unit.

For example, `iPS_DeleteKeys(DEL_VSS2 | DEL_VSS3)` deletes only those keys belonging to the Security Scripts loaded in slot #2 and #3.

Return Values

- = 0 Success
- E_KM_SYSTEM_ERROR

iPS_LoadSysClearKey()

Loads the VSS_KLK (that is, system keys). The values are presented in cleartext. Before writing the new value of the key, all other keys in the terminal are erased.

Use this function exclusively in a secure environment.

Prototype

```
int iPS_LoadSysClearKey(unsigned char ucKeyID,
unsigned char * pucINKeyValue);
```

Parameters

`ucKeyID`	The key identifier.	
	0x00	VSS_KLK (16 bytes)
`pucINKeyValue`	16-byte buffer containing the encrypted key.	

Return Values

• = 0	Success
• E_KM_SYSTEM_ERROR	

iPS_LoadSysEncKey()

Loads the system encryption keys. The new values must be presented encrypted under the current value of VSS_KLK. Contrary to cleartext loading, this encrypted loading does not erase all other secrets in the unit. An error code returns if the VSS_KLK is not present.

Prototype

```
int  iPS_LoadSysEncKey(unsigned char ucKeyID,
unsigned char * pucINKeyValue);
```

Parameters

`ucKeyID`	The key identifier.	
	0x00	VSS_KLK (16 bytes)
`pucINKeyValue`	16-byte buffer containing the encrypted key.	

Return Values

• = 0	Success
• E_KM_NO_KEY_LOADED	VSS_KLK is absent. No encrypted loading possible
• E_KM_KEY_INTEGRITY_ERROR	VSS_KLK is corrupted
• E_KM_SYSTEM_ERROR	

iPS_LoadMasterClearKey()

Loads the master key of the security script. The values are sent in cleartext, but must be all loaded in the same session. Before loading the first key after a power cycle, all previously loaded keys (including the system keys) are erased. This means that loading additional keys in a different session must be done encrypted.

Use this function to load the keys defined by the Security Module. Use this function exclusively in a secure environment.

Prototype

```
int iPS_LoadMasterClearKey(unsigned char ucKeySetID,
unsigned char ucKeyID, unsigned char * pucINKeyValue);
```

Parameters

`ucKeyID`	The key set identifier.	
	00	Key set defined in VeriShield Security Script #0.
	01	Key set defined in VeriShield Security Script #1.
	...	
	07	Key set defined in VeriShield Security Script #7.
`ucKeyID`	The key identifier.	
	Master key number/key index in the selected set	
`pucINKeyValue`	Pointer to the 8-byte buffer containing the encrypted value Master Key.	

Return Values

• = 0	Success
• E_KM_OUT_OF_RANGE	`ucKeySetID` or `ucKeyID` is out of range or script is not loaded.
• E_KM_SYSTEM_ERROR	

iPS_LoadMasterEncKey()

Loads the security script's master keys without deleting the keys already loaded. The new values must be presented encrypted under the current value of VSS_KLK. Use this function to load the keys defined by the Security Module. An error code returns if the VSS_KLK is not present.

Prototype

```
int  iPS_LoadMasterEncKey(unsigned char ucKeySetID,
unsigned char ucKeyID, unsigned char * pucINKeyValue);
```

Parameters

`ucKeyID`	The key set identifier. 00 Key set defined in VeriShield Security Script #0. 01 Key set defined in VeriShield Security Script #1. ... 07 Key set defined in VeriShield Security Script #7.
`ucKeyID`	The key identifier. Master key number/key index in the selected set.
`pucINKeyValue`	Pointer to the 8-byte buffer containing the cleartext value Master Key.

Return Values

• = 0	Success
• E_KM_NO_KEY_LOADED	VSS_KLK is absent. No encrypted loading possible
• E_KM_KEY_INTEGRITY_ERROR	VSS_KLK is corrupted
• E_KM_OUT_OF_RANGE	`ucKeySetID` or `ucKeyID` is out of range or script is not loaded.
• E_KM_SYSTEM_ERROR	

iPS_CheckMasterKey()

Indicates if a key is present in the specified location. The Key Verification Code (KVC) argument is irrelevant in the V/OS terminal because this function is used only for security script keys. The key can be part of a double- or triple-length DES key. The KVC of a portion of the key cannot be returned.

Prototype

```
int iPS_CheckMasterKey(unsigned char ucKeySetID, unsigned char ucKeyID,
unsigned char * pucINKVC);
```

Parameters

`ucKeyID`	The key set identifier. 00 Key set defined in VeriShield Security Script #0. 01 Key set defined in VeriShield Security Script #1. ... 07 Key set defined in VeriShield Security Script #7.
`ucKeyID`	The key identifier. Key index in the selected set.
`pucINKVC`	Unused

Return Values

• = 0	Success
• E_KM_NO_KEY_LOADED	The key is not loaded
• E_KM_KEY_INTEGRITY_ERROR	The key is corrupted
• E_KM_OUT_OF_RANGE	`ucKeySetID` or `ucKeyID` is out of range
• E_KM_SYSTEM_ERROR	

SetSecurePINDisplayParameters()

Registers the callback function for the upcoming PIN session. Call this function each time before requesting a PIN session either through an IPP packet (Z60, Z63 or Z69) or through a VeriShield Security Script API (`iPS_RequestPINEntry()`). See SetSecurePINDisplayParameters(), page 246.

Prototype

```
void setSecurePINDisplayParameters(struct touch_hs_s *hotspot_table,
void *callback);
```

iPS_SetPINParameter()

Configures several parameters for the upcoming VSS PIN session.

Prototype

```
int iPS_SetPINParameter( PINPARAMETER * psKeypadSetup);
```

Parameters

```
psKeypadSetup
typedef struct {
unsigned char ucMin,
unsigned char ucMax,
unsigned char ucEchoChar,
unsigned char ucDefChar,
unsigned char ucOption,
} PINPARAMETER;
```

`ucMin`	Minimum number of PIN digits. It must be in the range [4..12].
`ucMax`	Maximum number of PIN digits. It must be at least equal to Min but not greater than 12.
`ucEchoChar`	Not used on V/OS. The characters echoing PIN digits are displayed by the `SetSecurePINDisplayParameters()`'s callback function.
`ucDefChar`	Not used on V/OS of terminals. The default field fill characters are displayed using the `SetSecurePINDisplayParameters()` callback function.
`ucOption.bit0`	= 1 turns Auto Enter feature ON
`ucOption.bit1`	= 1 accepts NO PIN entry (pressing ENTER before any digit).
`ucOption.bit2`	RFU - Must be 0
`ucOption.bit3`	= 1 makes the <CLEAR> key behave like a backspace key. Only one digit is deleted instead of all the digits entered so far.
`ucOption.bit4`	= 1 cancels the PIN session when the <CLEAR> key is pressed with no PIN in the buffer (The user has not entered any PIN digit, or has already cleared out all the PIN digits). When the PIN session is canceled this way, the `*piStatus` value returned by `iPS_GetPINResponse()` is 0x0A.
`ucOption.bit5..7`	RFU – Must be 0

Return Values

• = 0	Success

• E_KM_OUT_OF_RANGE	At least one of the parameters is out of range
• E_KM_SYSTEM_ERROR	

iPS_SelectPINAlgo()

Selects the PIN algorithm or mode for the next VSS PIN session. The PIN algorithm cannot be changed during a PIN session.

In the V/OS terminal, the only supported modes are `0Ah` for EMV offline PIN or `0Bh` for use with VeriShield Security Scripts. In these modes, the PIN is saved internally and retrieved by the security code for presentation to the smartcard (in plaintext or encrypted form) or by a Security Script command for post-processing.

Prototype

```
int iPS_SelectPINAlgo(unsigned char ucPinFormat);
```

Parameters

`ucPinFormat`	0Ah	EMV mode: Store the PIN internally to present to the smartcard later.
	0Bh	Store the PIN internally for post-processing by a VeriShield Security Script command.
	All other modes are not supported in V/OS terminals.	

Return Values

• = 0	Success
• E_KM_OUT_OF_RANGE	`ucPinFormat` is out of range or unsupported.
• E_KM_BAD_SEQUENCE	PIN algorithm cannot be changed during a PIN session.
• E_KM_SYSTEM_ERROR	

iPS_RequestPINEntry()

Initiates PIN collection. Once the PIN entry is complete, the PIN is formatted and encrypted according to the algorithm specified by `iPS_SelectPINAlgo()`. The encrypted PIN block is placed in a buffer and made available for `iPS_GetPINResponse()`. This function is non-blocking, which allows the unit to perform other tasks while the customer is entering their PIN.

Prototype

```
int iPS_RequestPINEntry( unsigned char cPANDataSize,
unsigned char * pucINPANData);
```

Parameters

`ucPANDataSize`	RFU – ignored in V/OS terminals.
`pucINPANData`	RFU – ignored in V/OS terminals.

Return Values

• = 0	Success
• E_KM_ACCESS_DENIED	Too many PIN sessions requested in a short period of time. Try again in few seconds. See the note on the PIN session timeout in ippRead(), page 244.
• E_KM_BAD_SEQUENCE	PIN session already started.
• E_KM_SYSTEM_ERROR	

iPS_GetPINResponse()

Checks the status of the VSS PIN session. It is typically used by the application in a loop to poll the system until the PIN session ends. The information returned by this function during the PIN session can be used in conjunction with a timer to implement an inter-character timeout as required in certain countries. This functions returns the number of PIN digits entered and the last non-numeric pressed.

Prototype

```
int ippGetPINResponse (int * piStatus, PINRESULT * pOUTData);
```

Parameters

`piStatus`

OK (0x00)	Done. Data contains the result of the comparison/ encryption.
0x01	Unit is idle
0x02	Collecting PIN
0x05	Aborted by user: The <CANCEL> key has been pressed.
0x06	NO PIN entered (Only if this option is turned on)
0x0A	Aborted by user: The <CLEAR> key has been pressed with no PIN digit in the buffer (The user had not entered any PIN digit, or had already cleared out all the PIN digits). This value can be obtained only if ucOption.bit4 has been set using function `ippSetPINParameters( )`.
0x0C	The PIN session timed out. See the Note on the PIN session timeout in ippRead(), page 244.

`pOUTData` Pointer to an object of the following type:

```
typedef struct {
unsigned char nbPinDigits;
unsigned char encPinBlock[8];
} PINRESULT;
```

This structure returns different information depending on the PIN session status. If `*piStatus` is equal to:

OK(0x00) Done.

`pOUTData ->nbPinDigits`	Number of PIN digits entered (PIN length).
`pOUTData ->encPinBlock`	Contains no relevant information.

0x01 Unit is idle

`pOUTData ->encPinBlock`	Contains no relevant information.

0x02: Collecting PIN.

`pOUTData ->nbPinDigits`	Number of PIN digits entered.
`pOUTData ->encPinBlock`	The first byte of the buffer contains the value of the last non-numeric key pressed. Relevant values are: • 0x00: No non-numeric key has been pressed since the last call to `iPS_GetPINResponse()`. • 0x0D: <ENTER> key • 0x08: <CLEAR> key • 0xF9: The user attempted to enter one more PIN digit than the maximum number allowed (`ucMax`) • Any other non-numeric key value defined in the hotspot table.

0x05: or `0x0A:`	Aborted by user		
	`pOUTData ->nbPinDigits`	0	
	`pOUTData ->encPinBlock`	The first byte of the buffer contains the value of the last non-numeric key pressed. Values can be: • 0x00: If the last key pressed was a numeric key (PIN digit). • 0x1B: <CANCEL> key • 0x08: <CLEAR> key	
0x06	No PIN entered (only if this option is enabled). - `pOUTData` contains no relevant information.		
0x0C	Timed out. - `pOUTData` contains no relevant information.		

Return Values

• = 0	Success
• E_KM_SYSTEM_ERROR	

iPS_CancelPIN()

Cancels PIN processing.

Prototype

```
int iPS_CancelPIN(void);
```

Return Values

• = 0	Success
• E_KM_SYSTEM_ERROR	

iPS_InstallScript()

NOTE Included for backward compatibility only. V/OS script management is done using Secure Installer.

Installs a VeriShield Security Script file.

Prototype

```
int iPS_InstallScript(void);
```

Return Values

• = 0	Always

iPS_GetScriptStatus()

Checks if a VeriShield Security Script file is installed in the specified script location and if so, returns the name of the script.

Prototype

```
int ippGetScriptStatus(unsigned char ucScriptNumber,
unsigned char *pucINName);
```

Parameters.

`ucScriptNumber`	Script number. Range [0..7].
`pucINName`	The pointer to the application buffer to which the 8-character name of the VeriShield Security Script value transfers.

Return Values

• = 0	Success
• E_VS_SCRIPT_NOT_LOADED	This script is not installed or not accessible from the current application group.

iPS_UninstallScript()

NOTE

Included for backward compatibility only. V/OS script management is done using Secure Installer.

Script uninstall is done on download using the secure installer.

Prototype

```
int ippGetScriptStatus(unsigned char ucScriptNumber);
```

Parameters.

`ucScriptNumber`	Script number. Range [0..7].

Return Values

- = 0   Always

iPS_ExecuteScript()

Spawns the execution of a given macro from a loaded VeriShield Security Script.

Prototype

```
int iPS_ExecuteScript(unsigned char ucScriptNumber,
unsigned char ucMacroID, unsigned short usINDataSize,
unsigned char * pucINData unsigned short usMaximumOUTDataSize,
unsigned short *pusOUTDataSize, unsigned char * pucOUTData);
```

Parameters

`ucScriptNumber`	Script number. Range [0..7].
`ucMacroID`	Number of the macro function to be executed.
`usINDataSize`	The size of the input data (in bytes).
`pucINData`	The pointer to the buffer containing the input data.
`usMaximumOUTDataSize`	The maximum size of the output data. This is typically the size of the output buffer.
`pusOUTDataSize`	Pointer to number of bytes returned by the macro in the output buffer.
`pucOUTData`	The pointer to the output buffer. The number of bytes returned in the output buffer is specified by `pusOUTDataSize`. If the macro is returns more data than the output buffer can contain, an error `E_VS_BAD_LENGTH` is returned and nothing is copied into the output buffer.

Return Values

• = 0	Success
• Value in the range [0...255]	Macro execution error - The returned value is the value of the opcode that caused the execution error. Refer to P/N 21883 for the list of opcodes.
• E_VS_SCRIPT_NOT_LOADED	This script is not loaded, not authenticated, or not accessible from the current application group.
• E_VS_MACRO_NOT_EXIST	This macro does not exist in this script.
• E_VS_BAD_LENGTH	`usINDataSize` is less than the value expected by the macro or `usMaximumOUTDataSize` is less than the number of bytes that the macro is attempting to return.
• E_VS_BAD_CHAINING	Bad sequence of macro (see chaining mechanism in P/N 21883).
• E_VS_SYSTEM_ERROR	

pcPS_GetVSSVersion()

Returns the version of the VeriShield Security Script interpreter.

Prototype

```
char* pcPS_GetVSSVersion(void);
```

Return Values

It returns a char pointer to the following null-terminated string:

```
"PSVSSvX.Y"
```

- X Major version
- Y Minor version

Security Services Functions

All listed security services calls are defined in the header file `svcsec.h`. Applications must link with the `libvfisec.so` library by using `-lvfisec`.

cryptoWrite()

Use the File Encryption feature to guarantee that the file content is lost if the unit is tampered with. The file is encrypted with a key derived from the top-level key erased from the terminal in case of attack, making it impossible to recover the content of the encrypted file. The key is unique for each terminal and is not known outside the cryptographic unit of the terminal.

This feature can be used, for instance, when tamper detection must cause the deletion of the transaction batch file.

`cryptoWrite()` encrypts and writes `count` bytes of data from `buffer` to the open file associated with `handle`. It returns the number of bytes actually written, which is `count` bytes unless an error occurs. The file must be open for both reads and writes (that is, if the file was opened with flag `O_WRONLY` set, the function returns –1 and errno set to `EBADF`).

Prototype

```
int cryptoWrite(int handle, const char *buffer, int count);
```

Parameters

`handle`	File handle
`buffer`	Pointer to the buffer holding the input data.
`count`	Number of byes to write.

Return Values

• >= 0	Success (number of bytes written).
• -1	File error; errno is set accordingly.
• < -1	System error.

cryptoRead()

Reads a maximum of `count` bytes of encrypted data from the open file associated with `handle`, decrypts the data, and stores the result in `buffer`. It returns the number of bytes actually read, which may be less than `count` if fewer bytes are available.

Prototype

```
int cryptoRead(int handle, char *buffer, int count);
```

Parameters

`handle`	File handle.
`buffer`	Pointer to the buffer holding the input data.
`count`	Maximum number of byes to write.

Return Values

>= 0	Success (number of bytes read).
-1	File error; errno is set accordingly.
< -1	System error.

rsa_calc()

Performs a public key RSA computation. It supports key lengths from 9 bits up to 2048 bits, and exponent values that can be written as (2^{exp}+1), for instance 2, 3, 65537.

Prototype

```
int rsa_calc(unsigned char * msg, unsigned char * mod, int count, int exp,
unsigned char * result);
```

Parameters

`msg`	Pointer to the buffer holding the input data.
`mod`	Pointer to the buffer holding the modulus. **Note:** If `count` is odd, the first byte (leftmost) cannot be null, if `count` is even the first two bytes (leftmost) cannot be null or the function returns an error.
`count`	Length of the modulus and input data in bytes. Min = 3; Max = 256.
`exp`	Code for exponent: actual exponent is 2^{exp}+1. For example, set `exp` to 16 for exponent 65537.
`result`	Pointer to the buffer receiving the result on exit.

Return Values

• = 0	Success
• < 0	Error - Invalid parameter.

vfi_SHA1()

Performs a SHA-1 computation as described in FIPS PUB 180-2. It returns a 20-byte message digest.

Prototype

```
int vfi_SHA1(unsigned char * option, unsigned char * input_buffer,
unsigned long nb, unsigned char * sha20);
```

Parameters

`option`		
	`SHA1INIT`	First call. The SHA-1 engine is initialized before processing the data. No digest is returned.
	`SHA1BUFF`	Intermediate call. It feeds the SHA-1 engine more data. No digest is returned.
	`SHA1TERM`	Final call. The 20-byte digest is returned after the data is processed.
	`SHA1ALL`	One-step operation; combines all options above.

`input_buffer`	Pointer to the input buffer holding the message to be processed.
`nb`	Number of bytes in the buffer.
`sha20`	Pointer to the 20-byte buffer where the message digest transfers.

Return Values

- = 0 Success
- < 0 Error

DES()

Performs DES, DESX and Triple-DES computations. The operation type and key length are specified using the parameter `ucDeaOption`.

Prototype

```
int DES(unsigned char ucDeaOption, unsigned char *pucDeaKey8N,
unsigned char *pucInputData, unsigned char *pucOutputData);
```

Parameters

`ucDeaOption` Algorithm:

`DESX1KE(02h)`	DEAX encryptions with single-length key
`DESX1KD(03h)`	
`DESX2KE(04h)`	DEAX encryptions with double-length key
`DESX2KD(05h)`	
`DESX3KE(06h)`	DEAX encryptions with triple-length key
`DESX3KD(07h)`	
`DESE(08h)`	DEA encryptions with single-length key
`DESD(09h)`	
`TDES2KE(0Ch)`	TDEA encryptions with double-length key
`TDES2KD(0Dh)`	
`TDES3KE(0Eh)`	TDEA encryptions with triple-length key
`TDES3KD(0Fh)`	

`pucDeaKey8N`	Pointer to 8N-byte key block (N=1, 2 or 3).
`pucInputData`	Pointer to 8-byte input block.
`pucOutputData`	Pointer to 8-byte output block

Return Values

• = 0	Success
• < 0	Error

AES()

Performs AES computations on a 128-bit data block. The operation type and key length are specified using the parameter `ucAesOption`.

Prototype

```
int AES(unsigned char ucAesOption, unsigned char *pucAesKey8N,
unsigned char *pucInputData, unsigned char *pucOutputData);
```

Parameters

`ucAesOption` Algorithm:

`AES128E (04h)`	AES encryptions using a 128-bit key.
`AES128D (05h)`	
`AES192E (06h)`	AES encryptions using a 192-bit key.
`AES192D (07h)`	
`AES256E (08h)`	AES encryptions using a 256-bit key.
`AES256D (09h)`	

`pucAesKey8N`	Pointer to 8N-byte key block (N=1, 2 or 3).
`pucInputData`	Pointer to 16-byte input block.
`pucOutputData`	Pointer to 16-byte output block.

Return Values

- = 0 Success
- < 0 Error

generateRandom()

Returns a random 8-byte value.

Prototype

```
int generateRandom(char *random8);
```

Parameters

`random8`	Pointer to the 8-byte buffer where the random value is transferred.

Return Values

- = 0 Success
- < 0 Error

isAttacked()

NOTE

Included for backward compatibility only. On V/OS terminals, always returns 0.

Indicates if an attack occurred causing the loss of the transactions key and/or encrypted files. However, the V/OS system does not allow applications to run until the unit is de-tampered, so the return value is always 0.

Prototype

```
int isAttacked(void);
```

Return Values

• = 0	Always

secVersion()

Returns the version number strings for the security module and the security library in the form `xx.yy.zz`.

Prototype

```
int secVersion(char *pchModVersion, char *pchLibVersion);
```

Parameters

`pchModVersion`	Pointer to the 9-byte buffer receiving the security module version number.
`pchLibVersion`	Pointer to the 9-byte buffer receiving the security library version number.

Return Values

• = 0	Success
• < 0	Error

authFile()

Checks if the specified file is authenticated.

Prototype

```
int authFile(const char *filename);
```

NOTE

When specifying a filename, ensure that the full path is included. For example:

`/home/usr1/example.exe.`

Parameters

`filename`	Pointer to the name of the file to authenticate.

Return Values

• = 1	File is authenticated
• <= 0	File failed authentication

loadOSFiles()

NOTE Deprecated – Do not use – See Secure Installer for information on downloads to V/OS terminals.

Prototype

```
int loadOSFiles(void)
```

Return Values

• = 0	Always

osHash()

Not currently supported. Always returns 0.

CHAPTER 25

Real-Time Clock (RTC)

Linux has a rich API to support RTC functionality, so we will not be supplementing or modifying that API. The V/OS of terminals include Verifone-specific RTC hardware required to keep the time accurate when the unit is off.

On power up, the operating system automatically sets the Linux (soft) RTC from the V/OS terminal RTC hardware. Call `setRTC()` immediately after any call that sets the time and date of the Linux RTC or the updated time/date is lost when the unit is off.

setRTC()

Sets the RTC hardware to the Linux RTC time/date. This function must be called immediately after any function that sets the Linux RTC time/date. This API has no parameters or return codes.

Prototype

```
void setRTC(void);
```

setDateTime()

As user processes are not allowed to set the Linux RTC, applications must call `setDateTime()`, passing the date and time in the same format used in the shell command date: MMDDhhmmYYYY.ss, where:

- MM = Month
- DD = Day
- hh = Hour (24-hour format)
- mm = Minute
- YYYY = Year
- ss = Seconds

NOTE

`setDateTime()` only sets the Linux RTC. To set the hardware RTC, the application must follow `setDateTime()` with `setRTC()`.

Prototype

```
int setDateTime(char *dateTime);
```

Return Values

= 0	Success
< 0	Error

CHAPTER 26

System Mode Security

System security is enhanced beyond PCI requirements.

Key Security Features

- Pre-expired passwords
- Logical System mode logins
- Login/password management
- System mode access control lists
- Secure remote password download
- HMAC of OS components

Pre-expired Passwords

It is a PCI requirement for passwords allowing access to sensitive areas of the system to expire when the unit is shipped from the factory. The first login after the unit leaves the factory forces the operator to change the password. The new password cannot be the same as the default password.

Unique Password	Used by	Notes
System mode (sensitive areas) / Tamper clearing	Verifone, Acquirer for cleartext key loading	This is the usr1 account password. The password cannot be set by the user to the default value.
Cleartext key loading	Deployment, Acquirer	Two passwords names keyloading1 and keyloading2. Accessed through the System mode menu after usr1 login. The password cannot be set by the user to the default value.
System mode Limited access	Deployment, Customer	usr2 through usr16 The password cannot be set by the user to the default value.

Unique Password	Used by	Notes
System mode Limited access	Repair depots	Maintenance mode Does not require expired passwords as there is no ability to access sensitive areas or clear tamper codes. Allows the password to be reset to the default.
System mode	Customer	Level1 and Level2 System mode access. There is no default password for these logical logins so they expire. An authenticated security policy file is required to enable Level1 and Level2 logins.

System Mode Logins

The following accounts are enabled:

- **Supervisor:** Full access to all System mode functionality.
- **Level1:** This optional login has System mode access limits defined by the security policy file. Note that the Level1 login acts as a subset of the usr1 account.
- **Level2**: This optional login has System mode access limits defined by the security policy file. Note that the Level2 login acts as a subset of the usr1 account.
- **Maintenance:** View settings and perform diagnostics either at the customer site or a repair depot. Does not allow changes to the device.

The following accounts are used to enter the system in a secure state.

- **Keyload1:** First user authenticated to enter a secure state (such as direct key injection).
- **Keyload2:** Second user authenticated to enter a secure state (such as direct key injection).

Login / Password Management

Prior to entering System mode, the PINpad allows you to select a login account, and then prompts for a password. If the password is expired or pending change, you must enter the current password and then a new password (pre-defined for pending password changes). New passwords must be entered twice for validation.

System Mode Access Control

System Mode access can be limited when logging in as Level1 and Level2. Use configuration variables in the security policy file to define specific tabs accessible to the user. For example, if the INFORMATION tab is disabled there is no access to the functionality (sub-menus) defined under it. This includes the OS Details, Packages, Flash, Cable, COM3, and Ethernet buttons.

Another example is the CONFIGURATION tab. You can disable all capabilities in the sub-menus or the ability to select one or more sub-menu items. For example, it is possible to only disable the ECR PORT and EDITOR functions, but leave all remaining features enabled.

See Security Policy File Details for how to define System mode access control variables for Level1 and Level2 users.

Secure Remote Password Download

Simple and consistent secure password download are allowed to:

- Perform a SHA256 hash on all password values, ensuring they are never passed in the clear.
- Enhance the strength of the SHA256 by using both the existing and new password in the hash generation.
- Use the authenticated security policy file to download new passwords as well as password state changes.

A simple PC tool, the Security Policy File Generation tool, was developed to simplify creation of the SHA256 from old and new passwords. The output of this tool is a Security Policy file that can then be signed and downloaded.

Password States

As demonstrating in Figure 11 the V/OS PINpad is secure in each stage of its product life cycle. There is no need for different controlling entities to share common passwords.

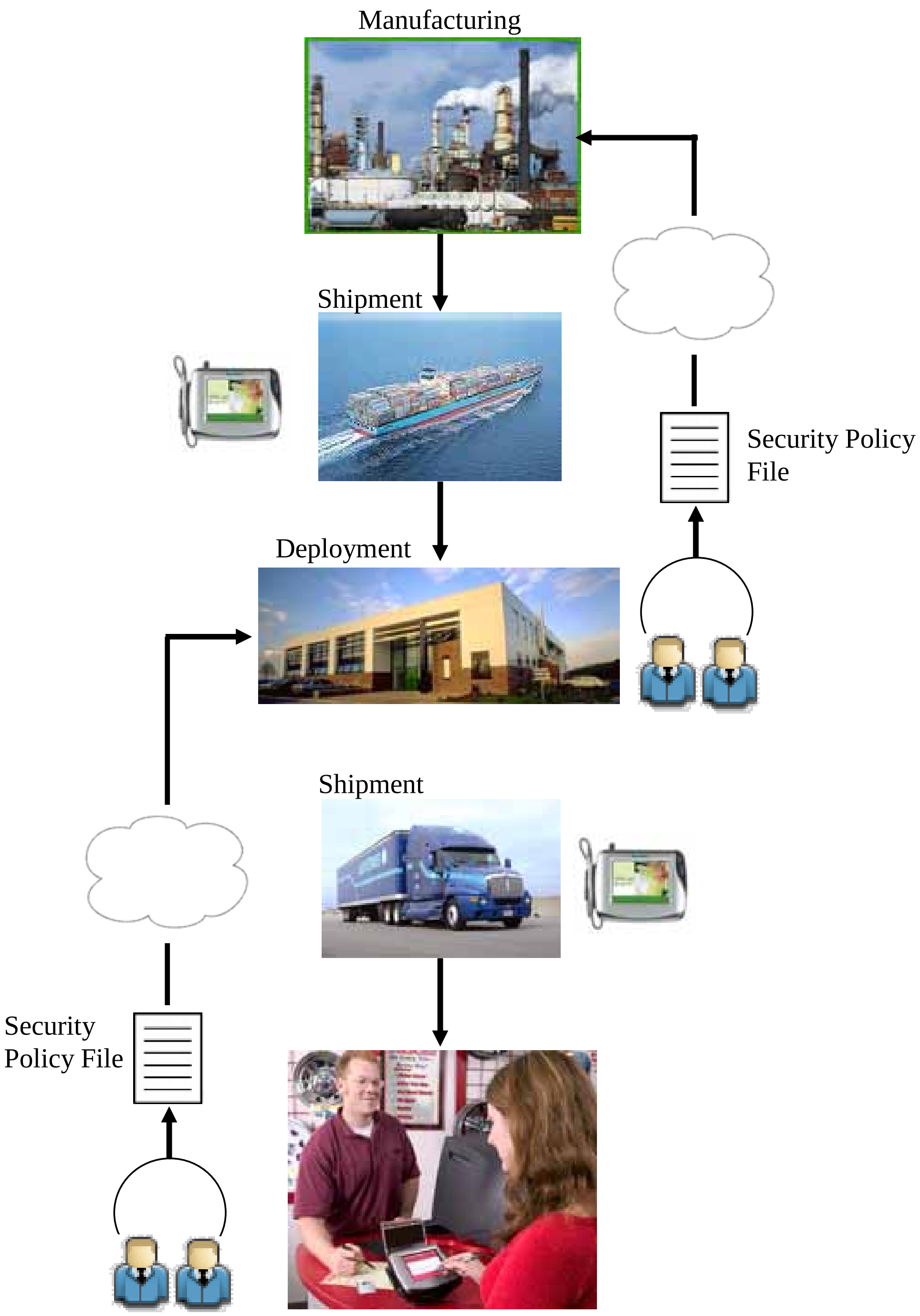


Figure 11 Password Lifecycle Flow

Entity	Role
Manufacturing	Creates device, performs initial firmware install, loads a security policy file generated by the deployment entity. Security policy includes pending password changes for all system passwords. The Security Policy file can be viewed in the clear without concern of discovering dual control passwords. Existing (default) passwords expire prior to shipment, placing the unit into the deploy state.
Shipment	Being in the deploy state keeps the device secure from tampering while being transported between controlling entities.
Deployment	Accepts the device from manufacturing. Dual operators must present passwords to gain access to the device. Once the device is under deployment control it is prepared for the end-user by loading customer-specific application software. This application software includes a signed (using Verifone VeriShield) Security Policy file that includes pending passwords for all system passwords. Existing (deployment) passwords expire prior to shipment.
End-User	Accepts the device from Deployment, and installs it at a retail location. The application automatically activates. If the store manager requires access to a System mode function, a dual control password known only to them is required. The unit is secure and the Manufacturing and Deployment entities have no knowledge of the system access password.

State	Definition
`Password Change is Pending`	The Security Policy file contains a pending password change variable for a given login. The pending password is composed of a SHA256 of the old and new password. Hashing the old and new password enhances security and allows the value to be publicly invisible. At the next login, the user must present the old and new passwords (dual control) that comprise the pending password.
`Password is Expired`	The Security Policy file contains a flag for a given account that determines if on the next login the current password must be changed. Note that when the Expired flag is set it is required that the existing password be entered before the user can enter a new password.
`BOTH Password Expired AND Change Pending`	**This is a SPECIAL case.** When both the password expired flag and the password change pending value are set, on the next login the user must present two passwords that make up the pending password change SHA256 value. The existing password is ignored because it is expired. Implementing this state allows the unit to remain secure without having a default password or sharing a password between controlling entities.

State	Definition
`Secure`	No password change is pending AND the password is not expired. This is the most common in field state where no password changes are required.
`Pending`	A password change is pending AND the password is not expired. A download has included an authenticated Security Policy file that defines a new password. On login, the operator is asked to enter the existing and new passwords.
`Deploy`	A password change is pending AND the password is expired. The Deploy state is used when a unit is being transferred from one device controlling entity to another. For example, from the deployment facility to the customer site.
`Expired`	No password change is pending AND the password is expired. When set without a pending password change, security is at its weakest in that on login the operator must present the current password and then must enter a new (arbitrary) password. While this state meets PCI 2.0 regulations, it is recommended that a password pending change be implemented at the same time.

NOTE

System mode supports a feature called “Prepare unit for shipment” that performs one of the following:

- Reset all passwords to default settings and expire all passwords (except Maintenance)

OR

- Expire all passwords (except Maintenance)

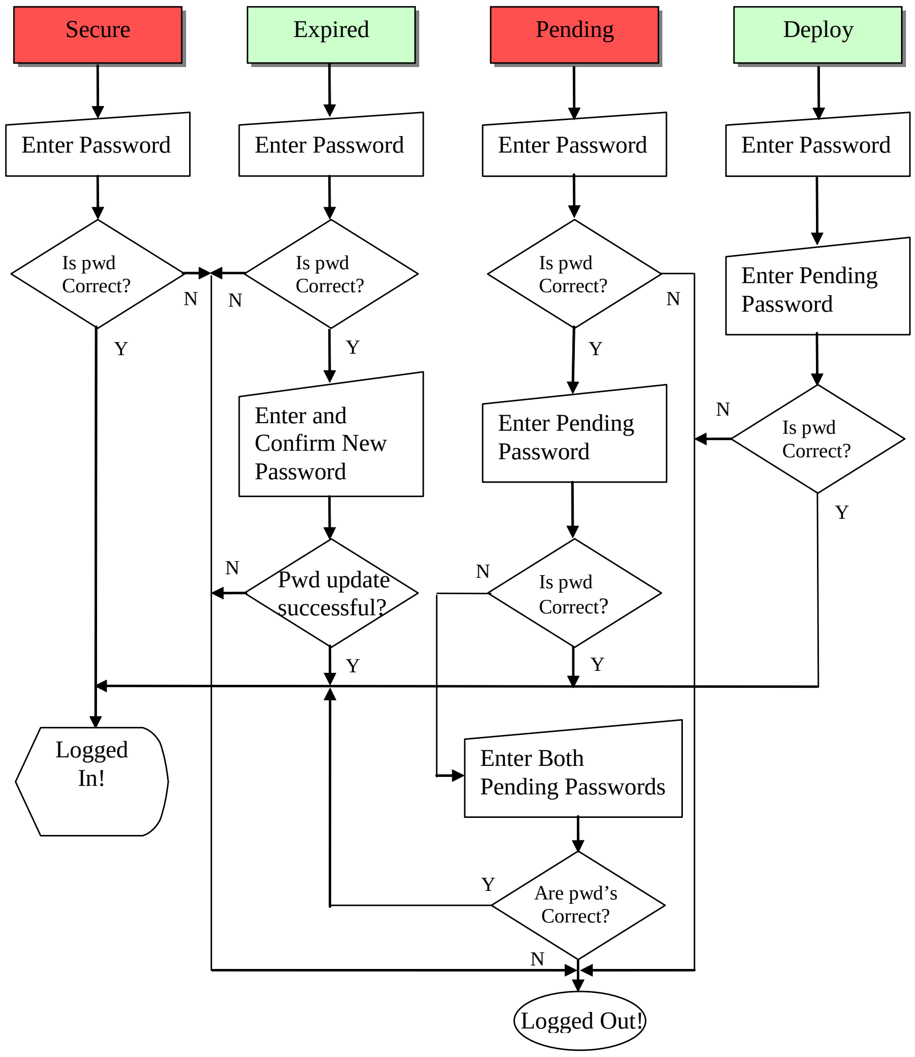


Figure 12 Login Logic Flow by State

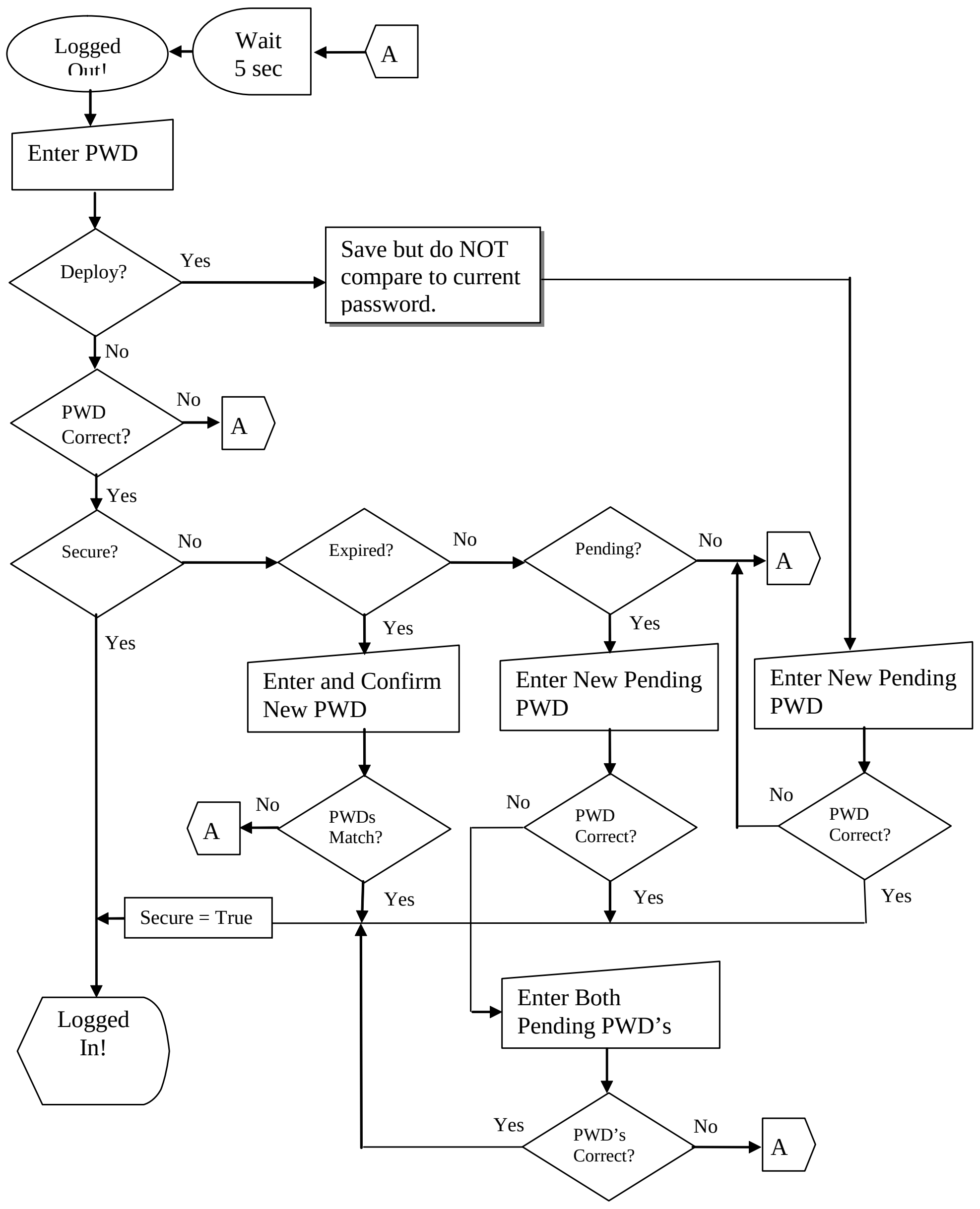


Figure 13 Login Logic Flow

Use Case

To better explain the flow chart in Figure 13, review the life cycle of the V/OS PINpad being built at the factory, shipped to a deployment center, and finally arriving at a customer's location.

At the Factory

The factory assembles and tests the V/OS PINpad. To accomplish this task, access is needed to the following System mode functionality:

- Information screens
- Limited TTY–needed to run mfginit process

- Diagnostics

The factory MAY download test applications, therefore access to the usr1 account is required. All passwords are set to their default values with no pending or expired state changes.

Optional (but strongly recommended)

During final inspection of the unit, the factory downloads a small Security Policy file that defines a pending password change for all critical accounts. The critical passwords are only known to the deployment facility and can be changed as needed. This option is strongly recommended to maximize the security of the unit as it travels from the factory to the deployment center.

The factory must select the System mode option that prepares the unit for shipment. Selecting this option resets and expires all passwords.

The V/OS PINpad unit is now in either the Deploy or Expired state, that is if a pending password was loaded. If the unit is only in the Expired state, the default password is in effect but it must be changed on first use.

At the Deployment Site

The V/OS PINpad arrives at the deployment site in one of two states depending on the process followed at the factory. The deployment site most likely performs key and application loading. This requires access to the key loading passwords and the usr1 password. These passwords are either in the default state or a state (expired) only known by the deployment group (pending). If the passwords are expired, the deployment group can set them to any value, however consistency is important.

TIP If the Security Policy file was downloaded at the factory prior to shipping the unit, there is no need for the deployment site to share ANY passwords with the factory. Security is greatly enhanced for the unit as it travels from site to site and there is no sharing of secrets!

NOTE The customer application includes an updated Security Policy file with password pending changes set for all critical passwords.

Once the deployment site has loaded the customer application, it selects the System mode option to prepare the unit for shipment. This expires all passwords.

In the Field at the Customer Site

On arrival at the customer site, the V/OS PINpad is most likely “plug and play.” In the unlikely event that System mode must be accessed, the unit prompts for the old and new password as a pending password change was enabled using the Security Policy file included with the application download. The old password is in no way related or known by the deployment site. By entering two unique passwords, the customer is forced to conform to the split knowledge, dual control methodology standard set by security people.

Security Policy File Details

The Security Policy file is similar to the `config.usr` file in that it uses the INI format with the concept of variable/label = value. The Security Policy file does not support sections, as supported on `config.usr` files. Also, unlike the `config.usr` files, the Security Policy file only supports a fixed set of entries. It is not intended for general-purpose application parameter storage. The Security Policy file MUST be signed using VeriShield and the application certificates. Once processed, the Security Policy file is deleted and the values set in it cannot be viewed or edited from System mode.

Variable/Label	Values	Function

`*supervisorpending`
`*level1pending`
`*level2pending`
`*keyload1pending`
`*keyload2pending`
	64-byte Hex ASCII String
	Computed SHA256 of current and new password.
	Use the provided Security Policy file generation tool to calculate the SHA256 values.
		Forces password change on the next login/use.

`*supervisorexpired`
`*level1expired`
`*level2expired`
`*keyload1expired`
`*keyload2expired`
	“1” or nothing to delete the variable.
	Example:
	`level1expired=`
		Set the variable to 1 to signal the system that a password has expired and must be updated on next use.

`level1access` `level2access`	Use the Security Policy File generation tool to create access strings. The Security Policy File generation tool is included with the SDK.	String that defines either the System Mode menus that are Enabled or Disabled for login.

VeriShield Remote Key (VRK) Functions

VeriShield Remote Key (VRK) occurs as part of a download without interaction with the application. The following function calls are available for use by applications.

copyPublickey_RKL()

Copies the VRK public key certificate to the `/tmp` directory. The filename is `KRD_sss-sss-sss.crt` where, `sss-sss-sss` is the serial number of the unit.

Prototype

```
void copyPublicKey_RKL(void);
```

Return Values

• = 0	Successful
• -1	Error

getKeyStatus_RKL()

Returns the current VRK key/certificate status.

Prototype

```
rkl_key_status_t getKeyStatus_RKL(void);
```

From `svcsec.h`:

```
typedef enum  // enumeration for RKL file status
{
    NO_KEYS_RKL,      // Neither file exists
    PUBLIC_KEY_RKL,   // Only /root/rkl_keys/rkl_cert.crt exists
    PRIVATE_KEY_RKL,// Only /root/rkl_keys/rkl_priv_key.der exists
    BOTH_KEYS_RKL    // Both files exist
}rkl_key_status_t;
```

Return Values

• >=0	Enumeration for RKL file status. Normal response from a unit with VRK is BOTH_KEYS_RKL.
• -1	Error

CHAPTER 27

Diagnostic Counters

NOTE

Reference the diag_counters_API.h header file to supplement the information in this chapter.

The diagnostic counters are a set of 256 4-byte variables in Non-Volatile Memory (NVM). Most are straight counters, but some have other purposes such as to record time stamps. Counters are referenced as numbers 0–255, symbolically defined in `enum diag_counter_index_number` in diag_counters_API.h.

Counter 0 is the ID (signature). On terminal boot up, the ID is checked against the opsys counters table in RAM (see Accessing Group Descriptions). If the ID is incorrect, the block is cleared to 0 and the ID reset to the value in the counters table. This allows new terminals to self-initialize the counter block.

Counters 1–127 are reserved for definition by applications. Counters 128–254 record various operating system (opsys) events. Counter 255 is reserved for system use.

Diagnostic Counters Functions

The Diagnostic Counter APIs provide calls to read and write the counters.

NOTE

The opsys counters, 0 and 128–255, are read only.

Table 10 Diagnostic Counter APIs

API	Purpose
`int diag_counter_clr(__s16 index);`	Clears counter to 0
`int diag_counter_set(__s16 index,__s32 value);`	Sets counter to <`value`>
`int diag_counter_get(__s16 index,__s32 *value);`	Reads to <`*value`>
`int diag_counter_add(__s16 index,__s32 value);`	Increments by <`value`>
`int diag_counter_sub(__s16 index,__s32 value);`	Decrements by <`value`>
`int diag_counter_inc(__s16 index);`	Adds 1
`int diag_counter_dec(__s16 index);`	Subtracts 1
`int diag_counter_set_bits(__s16 index,__s32 bitMask);`	Sets bits specified as 1s in <`bitMask`>
`int diag_counter_clr_bits(__s16 index,__s32 bitMask);`	Clears bits specified as 1s in <`bitMask`>

Table 10 Diagnostic Counter APIs (continued)

API	Purpose
`int diag_counter_set_time(__s16 index);`	Loads with number of seconds since the epoch 1/1/1970.
`int diag_counters_get_all(__s32 *buff);`	Reads all counters into <`buff[256]`>

diag_counter_info Structure

```
struct diag_counter_info
{
    char text[17];
   // Max of 16 chars + 1 null - Concise ascii description of the counter
    __u32 groups; // Bit mask of group(s) the counter belongs to
    __u32 flags; // Bit mask of display behavior modifiers
    __s32 value; // Convenient place to retrieve counter value
    __s16 index; // Counter number
};
```

Counter Attributes

In addition to the counters in NVM, the opsys maintains in RAM an array of 256 copies of the diag_counter_info structure to define the attributes of each counter.

- The <`text`>, <`groups`>, and <`flags`> elements are for formatting display routines (for example, as used by the Home > Info > Counters tab).

NOTE The <`groups`> and <`flags`> elements are defined in `enum diag_counter_group_mask` and `enum diag_counter_flags_mask` in diag_counters_API.h.

- When retrieving counter attributes, the <`value`> element retrieves the current counter value.
- The <`index`> element specifies the counter number.

Setting Counter Attributes

Applications can use counters 1–127, and assign attributes to these counters. For example, application counters can define groups and flags, or re-use those defined in the `diag_counter` enums mentioned above.

Use this call to set counter attributes:

```
int diag_counter_set_info(__s16 index, struct diag_counter_info *info);
```

Formatting Counter Displays

Use these calls to format counter displays. They are also used by the Home > Info > Counters tab.

- To get counter attributes and value:

  ```
  int diag_counter_get_info(__s16 index,struct diag_counter_info *info);
  ```

- To read the attributes and values of all counters into the `struct diag_counter_info[256]` array:

  ```
  int diag_counters_get_info_all(struct diag_counter_info *info);
  ```

Accessing Group Descriptions

The opsys maintains a small table of group descriptions. This table is for opsys use only. However, applications can use this call to access group descriptions for counter display formatting. The first entry in the group table that matches any bit set in `ginfo` returns.

```
int diag_counter_get_info_group(struct diag_counter_info_group *ginfo);
```

diag_counter_info_group Structure

```
struct diag_counter_info_group
{
    char text[13]; // Max of 12 chars + null
    __u32 group;
};
```

The text field contains an ASCII description of the group.

APPENDIX A

VeriShield Crypto Library (VCL)

The VCL software component provides Format Preserving encryption services for payment applications. It is transparent to the payment running application and to the payment infrastructure, except that data must be decrypted before presentation to the payment authorizing gate. See the *VeriShield Crypto Library Technical Reference Manual* (P/N D45-20021) for complete information.

APPENDIX B

V/OS on the MX Terminal

This section contains V/OS information specific to MX terminals. For MX terminals, this information augments that in the main body of this document.

LEDs

MX terminals have three blue LEDs located in magnetic stripe reader, an LED in the smartcard reader, and an LED array for the keypad. These inform the user that they can swipe their card. More complex use of these LEDs requires a thread or timer for synchronization. Use the ecore timers available in the GUI library.

NOTE For legacy applications, if your application directly addresses the LEDs, update the calls to use the definitions found in svc.h.

ledRunway()

Enables or disables the 3-LED runway effect.

NOTE Disable the runway effect before calling led_ledOff() or Led_ledOn() as undesirable side affects may occur.

Prototype `void ledRunway(int enable);`

Parameters

`enable`	0 = Disable the runway effect
	1 (non-zero) = Enable the runway effect

Serial Ports and Protocols

Figure 14 presents an overview of the three UARTs available on the multiport cable on MX terminals.

NOTE On V/OS-based terminals the I/O module determines available ports. The Wrenchman co-processor is in select I/O modules. COM1/COM2/COM3 may not be available on all configurations.

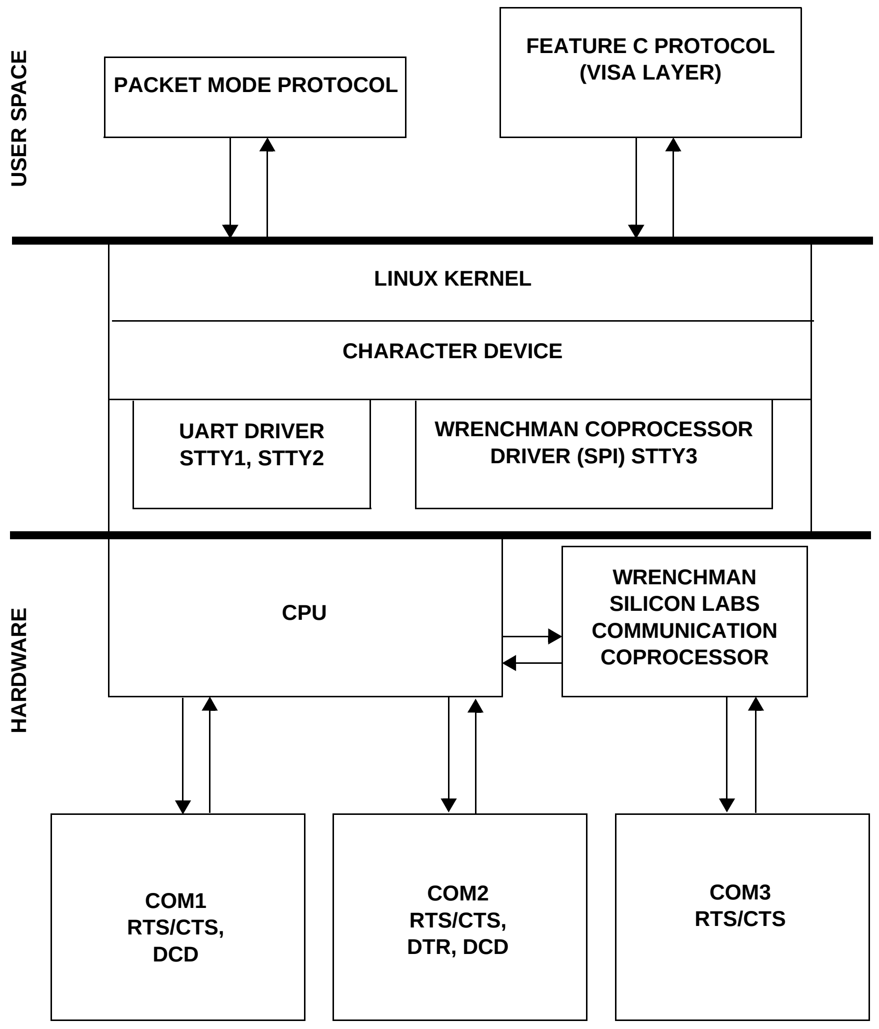


Figure 14 COM Module Architecture–MX Terminals

Serial Communication Control Structure

The following structures are defined for configuring serial ports on V/OS terminals. They contain all elements for defining the ports to the different supported modes. These structures comprise the necessary settings to configure a serial port for RS-232/485, Tailgate, Visa, Packet mode, or other protocols.

```
typedef struct{
short status;
short comm_handle;
short visa_sel; /* 0 - visa not selected, 1 - visa selected */
int port;
void (*pEcrReceive) (void); /* ecr rx callback for application */
struct Opn_Blk openBlock;
}ECR_INFO;


/* --------------------- Packet Defines --------------------------*/
typedef struct {
unsigned char stx_char;
unsigned char etx_char;
unsigned char count;
int max_pkt;
void (*pPktHandler) (char , int); /* rx callback for packet mode */
} packet_parm_t;


/* --------------------- ECR tailgate Defines ---------------------*/
typedef struct {
unsigned char poll_addr;
} ECRtailgate_parm_t;


/* -------------------- Open Block Structure ----------------------*/
 struct Opn_Blk {
unsigned int rate;
unsigned int format;
unsigned int protocol;
unsigned int parameter;
packet_parm_t packet_parms;
             union {
                ECRtailgate_parm_t ECRtailgate_parms;
             } trailer;
};
```

Parameters

`protocol`	The main structure is the Open Block structure (`Opn_Blk`), which contains the `protocol` element. `protocol` indicates port mode configuration and defines applicable fields for that mode.
`trailer`	The `trailer` field value depends on `protocol`. Because `trailer` consists of different types, only one type is valid at any time. If Tailgate mode is the protocol, `ECRtailgate_parms` is the valid field. All future mutually exclusive protocols are defined here.

`packet_parms`	The `packet_parms` field indicates the packet mode. It is defined outside the `trailer` field because different types of protocols can work in conjunction with packet mode. A serial port can be configured for RS-232 packet mode or as Tailgate with Visa protocol enabled that uses packet mode. This structure must be configured before calling `startPktMode()`.

Packet Mode Functions

Packet mode protocol manages packet-based protocols over any serial port. It supports the definition of a packet start and end character, as well as error detection characters (such as LRC/CRC).

The packet mode library buffers incoming data until a complete packet is received. Once a complete packet is in the receive buffer, the library calls a callback function to deliver the message to the application. Outgoing messages must be fully defined including start, end, and error detection characters before writing to the packet library.

Packet mode protocol has been implemented as the `libvfiprot.a` static library.

startPktMode()

Initializes Packet mode. Call `startPktMode()` before reading or writing messages in packet mode to configure the packet mode parameters. The `packet_parms` structure (in `Opn_Blk`) must be completed before calling `StartPktMode()`. Opening the ECR using `ecrOpen()` is automatically set. If packet mode is required on a port not opened with `ecrOpen()`, the application must complete this structure.

```
/* --------------------- Packet Defines --------------------------*/
typedef struct
{
unsigned char stx_char; // Define start character
unsigned char etx_char; // Define end character
unsigned char count; // Define number of characters for error check
int max_pkt; // Maximum message length in bytes
void (*pPktHandler) (char *, int); // RX message callback function
} packet_parm_t;
```

Prototype

```
int startPktMode(int hCom, struct Opn_Blk *openBlock);
```

Parameters

`hCom`	Handle of an open serial port.
`openBlock`	Serial communication control structure.

Return Values

= 0	Success
< 0	Error

endPktMode()

Frees the Packet mode buffers at session completion.

Prototype

```
void endPktMode(void);
```

packetRX()

Receives packet messages. `packetRX()` configures a callback function that executes message receipt.

Prototype

```
void packetRX(char * rxBuffer, int rxLength);
```

Parameters

`rxBuffer`	Pointer to a character buffer with the received message.
`rxLength`	Length of received message.

COM Ports

The following COM ports are available on MX terminals, and are referenced in software as the following devices:

- COM1 = `/dev/ttyAMA0`
- COM2 = `/dev/ttyAMA1`
- COM3 = `/dev/ttyWR0`
- COM4 = `/dev/ttyAMA2`
- COM5 = `/dev/ttyGS0`

Some ports are optional depending on terminal configuration. All ports are available as general RS-232 ports configured using different baud rates, character sizes, parity, and stop bits through either standard Linux calls or the legacy call `svcSetOpenBlock()`. Configure ports to work in character or packet mode such as the Visa protocol, and with or without flow control if available on the port. Supported baud rates for all ports are: 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200.

Serial port devices can be opened for control by more than one process at a time. The following `ioctl()` calls prevent processes from using the port, allowing exclusive access to the port.

TIOCEXCL	Put the ttyAMAx into exclusive mode, where no further `open()` operations are permitted. They will fail with EBUSY, except for root.
TIOCNXCL	Disable exclusive mode.

V/OS terminals COM ports do not support parity errors and BREAK condition detection.

The application must know which control and status lines are supported by each COM port. Control lines not supported by the hardware report the last status set by the application, and unsupported status lines report as asserted.

When a port is open through the `O_NONBLOCK` option, the `read()` command return value for that port can have different results if configured with flow control enabled. When flow control is enabled and no data is available on the port, `read()` returns 0. If flow control is not enabled and no data is available, `read()` returns -1 with errno set to EAGAIN (11 – try again). Both values indicate that no data is available.

COM1

COM1, or device `/dev/ttyAMA0`, supports the RTS, CTS, and DCD hardware lines.

COM2

COM2, or device `/dev/ttyAMA1`, supports the RTS, CTS, DTR, and DCD hardware lines. RTS/CTS flow control is under device driver control, and the RTS line must be controlled by the application using Linux `ioctl()` calls:

```
#include <unistd.h>
#include <termios.h>
```

```
int fd;
int status;
ioctl(fd, TIOCMGET, &status);
status &= ~TIOCM_RTS;
ioctl(fd, TIOCMSET, &status);
```

COM Port Functions

Use the following functions to control the communications ports of V/OS terminals.

comMonitorHandshake()

NOTE

Due to hardware constraints, this function ONLY works for COM 2 on V/OS terminals.

Enables a counter increment on transitions on the CTS and/or DCD lines. A call back function notifies the application of the handshake line state change. Before calling `comMonitorHandshake()`, open the COM2 port as the file descriptor is required for the second parameter.

The `mask` parameter monitors the DCD, CTS, or both handshake lines.

CAUTION

It is VERY IMPORTANT to call `comMonitorHandshake()` with a NULL for the callback function parameter immediately before closing the COM port.

Failure to do so may crash the application.

`comMonitorHandshake()` allows detection of rapid changes in the serial handshake lines. If transitions occur slowly there is no reason not to use standard Linux `ioctl()` calls to read port status. Use `comMonitorHandshake()` to start a background thread to monitor port status and callback. `comMonitorHandshake()` requires the `libvfiprot` library and the `vficom.h` and `protPktAPI.h` header files.

Prototype

```
int result = int comMonitorHandshake(void(*cmdsCallback)
(int, int, int),int fd, int mask);
```

Parameters

`void *cmdsCallback (int cts, int dcd, int status)`	Pointer to a callback function that must accept three parameters: • `cts` = Number of CTS transitions since last callback • `dcd` = Number of DCD transitions since last callback • `status` = State of DCD and CTS lines. Bit values are defined by `MON_DCD` and `MON_CTS` A transition is in either an asserted-to-deasserted state change, or a deasserted-to-asserted state change. Note: **Before closing the COM port pass a NULL to disable callback.**
`fd`	File descriptor for COM2.
`mask`	Handshake lines to monitor, bits. Specify `MON_DCD` and/or `MON_CTS`. Set to 0 to temporarily disable the callback.

Return values

= 0	Success

< 0	Error

COM3 COM3, or device `/dev/ttyWR0` can communicate with an Electronic Cash Register (ECR) using the Tailgate protocol. This is the only port that can communicate using this protocol.

As shown in Figure 14, page 597, COM3 is directly connected to the Wrenchman F331 8051 device controller, which handles all Tailgate protocol. A special driver allows this port to function either as a generic RS-232, RS-485, or SIO Tailgate device. It also allows special `ioctl()` calls to configure and obtain information on port status. `ioctl()` calls for this port are also available with the SVC COM3 service functions.

NOTE Due to firmware limitations, this port does not support true full duplex mode. Do not use full duplex protocols with this port.

If this port is connected to a full duplex device and a full duplex protocol is used in communication, this port may not be able to transmit and receive data without loss, depending on the amount of data and configured baud rate.

If this port is connected to any device capable of full duplex mode but uses a half duplex protocol in its communication (that is, the packet is sent, waits for response such as ACK/NAK, packet is sent), there is no problem of data loss.

Protocols and devices that do not use full duplex and that can safely connect to this port are: ECR, Zontalk, DL, DL5, DDL, most COM devices, printers, and check readers.

NOTE All baud rates configurations listed in this section, character sizes, parity and stop bits are supported, except 7N1 (7 bits, No Parity, and 1 stop bit). This setting is not supported on COM3. Use 2 stop bits (7N2) if you require the port be configured for 7-bit character sizes with no parity bit.

COM5 COM5, or device `/dev/ttyGS0`, is a serial device that transmits and receives data over a USB cable using USB Gadget technology. This allows V/OS terminals to act like any other serial port when connected to a USB host port. Configure the V/OS terminals to be a USB device that emulates a serial port connected to a USB host. Connect the terminal's USB device cable to a Windows or Linux OS based machine as another COM port. This allows serial USB transfers between the terminal and a PC.

The USB (version 1.1) serial device gadget is only supported over a berg cable. The USB OTG port does not support USB serial devices. Use the COM5 definition in vficom.h for the V/OS device name. V/OS terminals support the following ioctl() to detect cable status:

```
int handle, connected;

// Open USB gadget port
handle = open(COM5,O_RDWR | O_NONBLOCK);

// Check for error
if (handle > -1)
{
   // Read connected state
   connected = ioctl(handle,VFI_GADGET_CONNECTED);

   // If the connected state = -1 then cable was disconnected and
reconnected
       if (connected == -1)
   {
      // Cable was removed and re-connected, close port
      close(handle);

      // Re-open port
      handle = open(COM5,O_RDWR | O_NONBLOCK);
      if (handle < 0)
         // Port open failed!
   }
   else if (connected==1)// Good connection, OK to write date to port
   {
      // Write data to port
   }
   else         // No connection
      // Cable is disconnected
}
else
   // Port open failed!
```

Service Switch

When license enabled, the service switch triggers a stand tamper event on V/OS-based terminals to signal when the unit is removed from its stand. A “true” security tamper event is not triggered; the unit does not erase sensitive information on service switch tampers. There are two stand tamper modes:

- Default mode – the operating system manages the stand tamper event. On a stand tamper event, the unit reboots and displays the System Mode login screen. If the stand tamper signal is not active and a valid password is entered, the stand tamper event clears. While the unit is in a stand tamper condition, System Mode runs normally except that applications cannot execute.

  Set the following configuration variable to enable OS-managed stand tamper events:

  ```
  *standtamperactive= 0 | 1 | 2
  ```

  where, 0, 1, 2 identify the system mode login account that can clear the tamper.

  - 0 = Supervisor (can always clear a stand tamper event)
  - 1 = Level 1
  - 2 = Level 2

NOTE

The maintenance account can never be used to clear a stand tamper.

- Mode 1 – the application manages the service switch using the security service function calls.

Touch Panel Functions

This section presents MX terminal touch screen specifications and function calls.

Touch Panel Specifications

- X-axis values are 0 – 1880
- Y-axis values are 0 – 1360
- Z-axis values are 0 – 63
- Sample rate:
  - 200 per second for finger
  - 300 per second for stylus
  - The maximum active area is:

Parameter	MX925	MX915
Raw Capture Resolution (x,y)	1879 x 1359	1879 x 1359
Display Dimensions (x,y) inches	6.06 x 3.38	3.93 x 2.35
Display Resolution (x,y)	800 x 480	480 x 272

Parameter	MX925	MX915
Capture resolution (x,y) dpi	309 x 401	478 x 577
Signature Capture Rate (pps)	300	300
Non-signature Capture Rate (pps)	200	200

The Linux interface is through `/dev/input/mice` - the touch panel emulates a PS2 mouse. Use the FancyPants GUI APIs to access touch activity and position.

touchCmd()

Sets and retrieves touch panel functionality.

Prototype

```
int touchCmd(int cmd, int value);
```

Parameters

cmd	value	return	Notes
2	N.A	Bit values	Returns the status of the stylus. Non-zero means a stylus is attached.
4	0 or 1	N.A	Set input method: • 0 = Finger or Stylus (Automatic) • 1 = Stylus only
X	N.A	N.A	All `cmd` values not listed are reserved.

Signature Capture Functions

Signature Capture is implemented as a widget at the FancyPants GUI level. The widget supports the definition of a signing region as well as the ability to track stylus movement. Note that the strokes returned by the Signature Capture widget are scaled to 320x234 (72dpi). During signature capture, the system automatically enables palm rejection (finger is not recognized). Applications can use Stylus Priority mode during signature capture, but must call `touchCmd(11,0)` to disable palm rejection before initiating a signature capture. `touchCmd(11,1)` to restore palm rejection during signature capture.

Use the following calls to read and process the raw data for higher resolution images.

NOTE

During signature capture with a stylus connected, the unit switches to stylus-only mode.

```
#include "svc.h"
#include "sig.h"
typedef struct {
long x : 12; // X co-ordinate 0…1880 of touched point
long y : 12; // Y co-ordinate 0…1360 of touched point
long z :   8;   //  Z co-ordinate/pressure 0…63 of touched point
} __attribute__((packed)) xyz_t;
```

This is a representation of the x,y,z value of a single touched point packed into 32-bits – x,y, and z are signed quantities.

Also defines `PENUP`, which has the value:

```
{.x = -1, .y = -1, .z = -1}
```

These functions return -1 on error with the errno set.

SigCapCount()

Returns the number of available signature points.

Prototype

```
int SigCapCount(void);
```

SigCapGet()

Copies up to `maxPoints` of data from the kernel buffer to the user buffer. Returns the number of points actually copied, which is less than `maxPoints` if fewer points are available.

Use `SigCapCount()` to retrieve the number of points captured. Ensure that the buffer is large enough to hold the number of points captured. For example, if `SigCapCount` returns 1000, then the buffer must be (`xyz_t`)*1000 bytes.

Prototype

```
int SigCapGet(void *data, int maxPoints);
```

SigCapBoxApply()

Applies a signature box to the data in the results of `SigCapGet()`. Data outside the box is replaced by `PENUP`. The data is also compressed to remove adjacent duplicate points and adjacent `PENUP`s. The function returns the new number of unique points.

The application supplies the signature box to the function call.

The box coordinates are in screen format, that is:

- x = 0...479 and y = 0...271 for MX915 terminals
- x = 0...799, y = 0...479 for MX925 terminals

Signature data is in touch panel coordinates 0…1880 and 0…1360 for best resolution. It is not necessary to call this function before calling `SigCap2Tiff()`.

`SigCapGet` and `SigCapCount`

By design, a single `PENUP` (-1,-1,-1) is inserted at the beginning of the buffer as soon as the signature capture starts. This is why a count of 1 is returned and the point is returned as (-1,-1,-1). As long as the pen stays up, no points are inserted. Once the pen rests, new data (x,y,z) are added. A `PENUP` is also always inserted at the end of the buffer for quick the pen movements of just one down point to register. The count is 3 (initial pen up, the down point, and another pen up).

`SigCapGet` and `SigCapCount` always return the total number of raw points collected. The longer the pen is in contact with the touch panel, the higher the count – it does not matter if the points are in the box. Part of this count is residue from the initial pressing of the STROKES button in the test program. Those points are replaced with -1,-1,-1 when `SigCapBoxApply` is called. Positive results of the above function should be passed as the `count` parameter to `SigCapBoxApply`.

`SigCapBoxApply` does two things on the fly:

- Replaces all points outside the box by -1,-1,-1
- Compresses (shuffles toward the beginning of the buffer) the data so that all adjacent duplicates (for example several -1,-1,-1 in a row after applying the box) are replaced by a single copy of the data (in this case a single -1,-1,-1).

This function returns the new total number of points after compression. The original and compressed counts will differ. When displaying compressed data, use the return of `SigCapBoxApply` as the new size unless it is desired to display the raw data.

Prototype

```
int SigCapBoxApply(xyz_t *Sig, int count, SigCapBox_t *box);
```

vf_sig_cature_options()

Sets the rendering options (signature thickness bits) according to the union pointed to by the argument. It controls only how the signature looks on the screen. Note that a separate method in the TIFF function calls control thickening of signature lines in the TIFF file.

To retrieve the call and the `union` definitions:

```
#include vf_sig_capture.h
```

Prototype

```
void vf_sig_cature_options(union renderOptions *ro);
union renderOptions
{
   struct

      int xminus1yminus1 : 1;
      int xyminus1 : 1;
      int xplus1yminus1 : 1;
      int xminus1y : 1;
      int xy : 1;
      int xplus1y : 1;
      int xminus1yplus1 : 1;
      int xyplus1 : 1;
      int xplus1yplus1 : 1;
   } __attribute__((packed));
   int options;
};
```

`union` structure represents a bit map of pixels to plot in addition to the original pixel when a line is rendered using Bresenham's algorithm. The options are specified as either a bit field or an absolute value from 0 ... 511. This is how the bits are mapped to pixels:

0(x-1,y-1)	1(x,y-1)	2(x+1,y-1)
3(x-1,y)	4(x,y)	5(x+1,y)
6(x-1,y+1)	7(x,y+1)	8(x+1,y+1)

A null pointer causes a local static copy of `union` to be cleared; otherwise the `union` pointed to is copied into the local static copy. A cleared `union` is the power-up default and means that only the original point x,y is plotted with no thickening. This also provides runtime-savings as there is no scanning of individual bits when the `union` is clear.

To plot point (x,y) as well as (x+1,y) and (x,y+1) to get a thicker line use:

```
vf_sig_capture_render_options(&(union renderOptions){.options = 0x0b0});
```

which is equivalent to (note the double braces):

```
vf_sig_capture_render_options (&(union renderOptions)
{{.xy=1,.xyplus1=1,.xplus1y=1}});
```

RFCR Functions

RFCRlibVersion()

Stores the RFCR library version.

Prototype

```
int RFCRlibVersion(char *libVersion);
```

Parameters

`libVersion`	Pointer to read in the RFCR library version, in the form `xx.yy.zz`

Return Values

> = 0	Success
< 0	Error

RFCRInit()

Performs device initialization, and opens and configures the serial port. The device handle returned on success can be used by applications.

Prototype

```
int RFCRInit(void);
```

Return Values

>= 0	Success
< 0	Error
-ENXIO	No RFCR Module detected

RFCRClose()

Closes the serial port used to communicate with the Contactless RFCR module. This function does not power down the Contactless RFCR module.

Prototype

```
int RFCRClose(int handle);
```

Parameters

`handle`	The handle returned from `RFCRInit()`.

Return Values

= 0	OK
< 0	Error

RFCRGetVersion()

Returns the firmware version of the RFCR.

Prototype

```
int RFCRGetVersion(char *fwVersion);
```

Parameters

`fwVersion`	Pointer to read in the RFCR firmware version.

Return Values

= 0	Success
< 0	Error
-EBADF	Not initialized
> 0	NAK error

RFCRPing()

Tests for Contactless RFCR module response.

Prototype

```
int RFCRPing(void);
```

Return Values

= 0	Success
< 0	Error
-EBADF	If not initialized

RFCRReset()

Configures the RESET line of the Contactless RFCR module.

Prototype

```
int RFCRReset(int onOffPulse);
```

Parameters

`onOffPulse`	0 = OFF
	1 = ON
	2 = PULSE (OFF then ON)

Return Values

= 0	Success, for OFF, ON or PULSE
< 0	Error
-EBADF	If not initialized

RFCRSetAntenna()

Configures the antenna control of the Contactless RFCR module.

Prototype

```
int RFCRSetAntenna(short onOff);
```

Parameters

`onOff`	0 = Disable the RF Antenna
	1 = Enable the RF Antenna

Return Values

= 0	Success, for OFF
< 0	Error
-EBADF	If not initialized
> 0	NAK status

RFCRSetIndicator()

Configures the optional indicator controls of the Contactless RFCR module. The configurable LED is located on the right side of the reader face. The LED on the top-left side of the reader face is not configurable.

NOTE

The current RFCR hardware does not support the optional buzzer control.

Prototype

```
int RFCRSetIndicator(int led, int buzz);
```

Parameters

`led`	0 = disable LED
	1 = enable LED for 100 ms
	2 = enable LED for 200 ms
	...
	15 = enable LED for 1500 ms
`buzz`	Not used at this time

Return Values

= 0	Success
< 0	Error
-EBADF	If not initialized
> 0	NAK status

RFCRGetCardPayload()

Stores the Card Payload packet.

Prototype

```
int RFCRGetCardPayload(char* buff, int maxlen);
```

Parameters

`buff`	Pointer to store the Card Payload packet.
`maxlen`	Maximum size of `buff`.

Return Values

> 0	Success with the length of the Card Payload packet data read.
< 0	Error

RFCRParseCardPayload()

Parses the Card Payload data to the `CardPayload` structure.

Prototype

```
int RFCRParseCardPayload(CardPayload* payLoad, char* buff, int len);
```

Parameters

`payLoad`	Pointer to `CardPayload` structure to store the data.
`buff`	Pointer to Card Payload packet data.
`len`	Size of the Card Payload packet data in `buff`.

Return Values

= 1	Success
not 1	Any return value that is not 1 (either < = 0 or >1) are considered errors.

The CardPayload structure is:

```
typedef struct {
    char status;// see below
    char cardType;// see below
    short trk1Start;// index to beginning of Track 1 data
    short trk1Length;// length of Track 1 data
    short trk2Start;// index to beginning of Track 2 data
    short trk2Length;// length of Track 2 data
    unsigned char crc1;// LSB of CRC
    unsigned char crc2;// MSB of CRC
} CardPayload;
```

Refer to the *Omni 7xxx and MX Series Contactless Modules Programmers Manual*, P/N 23309 for the latest status code and the card type values.

Track 1 and Track 2 data contain the `Start` and `End Sentinels`.

RFCRPurge()

Purges any pending input from the reader.

Prototype

```
void RFCRPurge(void);
```

RFCRInputPending()

Returns the number of bytes available for reading.

Prototype

```
int RFCRInputPending(void);
```

Return Values

= 0	No data available
> 0	Number of bytes available for reading
< 0	Error

RFCRRawWrite()

Sends raw data to the Contactless RFCR module.

Prototype

```
int RFCRRawWrite(unsigned char *buff, int len);
```

Parameters

`buff`	Pointer containing the data to send to the RFCR module
`len`	Number of bytes to send

Return Values

> 0	Number of bytes written
< 0	Error

RFCRRawRead()

Reads raw data from the Contactless RFCR module.

Prototype

```
int RFCRRawRead(unsigned char *buff, int maxlen, int msecs);
```

Parameters

`buff`	Pointer to store the data read from the RFCR module.
`maxlen`	Maximum size of the buffer.
`msecs`	Maximum wait time in milliseconds for data to arrive.

Return Values

> = 0	Number of bytes read
< 0	Error

RFCRAddCRC()

Calculates the CRC of the data contained in `buff` and inserts it at the offset position of the buffer.

Prototype

```
void RFCRAddCRC(char* buff, int offset, int order);
```

Parameters

`buff`	Buffer containing the Command Frame to calculate the CRC.
`offset`	The position in the buffer to insert the CRC (and the size of the data to CRC), usually 14 for Command Frame.
`order 0`	Insert CRCs LSB first and MSB second.
`order 1`	Insert CRCs MSB first and LSB second.

RFCRCheckCRC()

Checks the CRC data in `buff`.

Prototype

```
int RFCRCheckCRC(char* buff, short len, unsigned short* calcCRC);
```

Parameters

`buff`	Buffer containing the data and CRC, with the CRC being the last 2 bytes of the buffer data.
`len`	Length of the data including the CRC.
`calcCRC`	Pointer to store the calculated CRC.

Return Values

= 1	CRC is valid
= 0	CRC did not match

RFCRReceiveACKFrame()

Receives an ACK frame from the Contactless RFCR module. The contents of the ACK frame specify whether the reader accepted the command. Depending upon the command, the ACK frame may provide additional information.

Prototype

```
int RFCRReceiveACKFrame(char* buff);
```

Parameters

`buff`	Pointer to store the ACK frame. Must have space for 16 bytes.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRReceiveDataFrame()

Receives a data frame from the Contactless RFCR module. The `Size` of the data frame may vary.

Prototype

```
int RFCRReceiveDataFrame(char* buff, int maxlen);
```

Parameters

`buff`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`maxlen`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read.
< 0	Error

RFCRSetPollMode()

Sets the RFCR polling mode.

Prototype

```
int RFCRSetPollMode(int mode);
```

Parameters

`mode`	• 00h Auto Poll • 01h Poll on Demand

Return Values

= 0	Success
< 0	Error

RFCRGetTransactionResult()

Returns the card data to `buff`.

The command is meant to be used in “Auto Poll” mode. When it detects a card, it carries out a transaction with the card, immediately returns the card data to the terminal, and resets its data buffer.

Prototype

```
int RFCRGetTransactionResult(char* buff, int max);
```

Parameters

`buff`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`max`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRActivateTransaction()

Polls reader for valid card and continues transaction.

The command is meant to be used in “Poll on Demand” mode. When this command is sent to the reader, it starts polling for cards. If it does not find a supported card for the specified time duration, it times out and ends the transaction. If it finds a card within the specified time interval, it carries out the transaction with the card and returns card data to the terminal.

Prototype

```
int RFCRActivateTransaction(char* data, int length);
```

Parameters

`data`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`length`	Length of the data field in command packet.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRActivateMxItransaction()

NOTE

This command is not supported in Gen3 architecture.

Polls reader for valid MxI card and continues transaction.

The command is meant to be used in “Poll on Demand” mode and an ePurse or Ticketing PayPass MxI card. When this command is sent to reader, it starts polling for cards. If it does not find a supported card for the specified time duration, it times out and ends the transaction. If it finds a card within a specified time interval, it carries out the transaction with the card and returns card data to the terminal.

Prototype

```
int RFCRActivateMxItransaction(char* cmdData,int cmdLen,char* rspData,int
rspMax);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRDebitWrite()

NOTE

This command is not supported in Gen3 architecture.

Continues to the second leg of an MxI payment transaction to complete the transaction. When the Debit Write command is sent to the RFCR, it attempts to carry out the remaining transaction

Prototype

```
int RFCRDebitWrite(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRCancelTransaction()

Cancels the polling of any approaching card after the Activate Transaction command or Update Balance command is sent to the reader.

Prototype

```
int RFCRCancelTransaction(void);
```

Return Values

= 0	Success
!= 0	Error status code

RFCRGetFullTrackData()

Retrieves full track data from the RFCR. If a card is read, this call carries out the transaction and returns the card data to the terminal.

Prototype

```
int RFCRGetFullTrackData(char* data, int length);
```

Parameters

`data`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`length`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRUpdateBalance()

Updates the balance on the card.

The command is meant to be used in “Poll on Demand” mode, and after the reader sends an online request to the issuer.

Prototype

```
int RFCRUpdateBalance (unsigned char status);
```

Parameters

`status`	Status of the Update Balance command.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRSetEMVParameters()

Sets or changes the values of the specified data objects in the reader. Used to set parameters for Auto Poll as well as Poll on Demand mode.

Prototype

```
int RFCRSetEMVParameters(char* data, int dataLen);
```

Parameters

`data`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`dataLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRGetEMVParameters()

Retrieves the values of the specified data objects in the reader from nonvolatile memory.

Prototype

```
int RFCRGetEMVParameters(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRSetBurstMode()

Enables RFCR Burst mode to send a data packet from the RFCR to the terminal on each successful card read. The terminal does not have to send any command or data to RFCR.

Prototype

```
int RFCRSetBurstMode(int burstMode);
```

Parameters

`burstMode`	• 00h – Disable Burst mode. • 01h – Enable Burst mode.

Return Values

= 0	Success
< 0	Error

RFCRSetPassThroughMode()

Enables RFCR Pass Through mode to stop the RFCR polling for cards it recognizes. Until Pass Through mode is disabled, only Pass Through commands will allow the terminal to communicate directly with PICC.

Prototype

```
int RFCRSetPassThroughMode(int mode);
```

Parameters

`mode`	• 00h – Disable Pass Through mode • 01h – Enable Pass Through mode

Return Values

= 0	Success
< 0	Error

RFCRPollForToken()

Starts polling for Type A or Type B PICC until a PICC is detected or timeout occurs. Once Pass Through mode starts, the RFCR stops polling for supported cards and remains dormant until this command is sent. If a PICC is detected within the specified time, the RFCR activates it and responds with card data.

Prototype

```
int RFCRPollForToken(int sec, int msec, char* rspData, int rspMax);
```

Parameters

`sec`	Timeout in seconds.
`msec`	Timeout in milliseconds.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRIsoApduExchange()

Sends, via the RFCR, application-level APDUs to a PICC that supports the 14443-4 protocol. The PICC response is returned by the RFCR to the terminal.

Prototype

```
int RFCRIsoApduExchange(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRPcdSingleExchange()

Sends, via the RFCR, raw data to a ISO 14443 PICC that does not support the ISO 14443-4 protocol (such as Mifare). The PICC response is returned by RFCR to the terminal.

Prototype

```
int RFCRPcdSingleExchange(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetPcdPiccParam()

Retrieves the PCD and PICC related parameters from the RFCR.

Prototype

```
int RFCRGetPcdPiccParam(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRMifareAuthenticateBlock()

Authenticates the Mifare card sector containing the specified block of data.

Prototype

```
int RFCRMifareAuthenticateBlock(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRMifareReadBlocks()

Reads data from one or more blocks on the Mifare card and returns the data to the terminal. The terminal can instruct the RFCR to read up to 15 blocks.

Prototype

```
int RFCRMifareReadBlocks(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRMifareWriteBlocks()

Writes data to one or more blocks on the Mifare card. The terminal can instruct the RFCR to write up to 15 blocks of data.

Prototype

```
int RFCRMifareWriteBlocks(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRMifarePurseCommand()

Enacts debit, credit, and backup operations on value blocks on a Mifare card.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRMifarePurseCommand(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRHighLevelHaltCommand()

Sends a HALT command to the card, used for any Type A or Type B card. This command can only be used in pass through mode.

Prototype

```
int FCRHighLevelHaltCommand(char halt);
```

Parameters

`halt`	• 01h Send HALT for Type A card • 02h Send HALT for Type B card

Return Values

= 0	Success
< 0	Error

RFCRSetCAPublicKey()

Sends the data related to a CA Public Key to the RFCR for storing it in a secure environment (Crypto Chip Memory).

Prototype

```
int RFCRSetCAPublicKey(char* data, int dataLen);
```

Parameters

`data`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`dataLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteCAPublicKey()

Deletes a previously set CA Public Key from secure storage in Crypto Chip Memory.

Prototype

```
int RFCRDeleteCAPublicKey(char* data, int dataLen);
```

Parameters

`data`	Pointer to the data field in command packet. The format and contents of the data field are given in *Contactless Software Programmers Guide* (P/N 26807).
`dataLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteAllCAPublicKeys()

Deletes all previously set CA Public Keys from secure storage.

Prototype

```
int RFCRDeleteAllCAPublicKeys(void);
```

Return Values

= 0	Success
< 0	Error

RFCRSetRTCTime()

Sets a specific time in the real-time clock.

Prototype

```
int RFCRSetRTCTime(int hour, int min);
```

Parameters

`hour`	Hour value.
`min`	Minute value.

Return Values

= 0	Success
< 0	Error

RFCRGetRTCTime()

Returns the current time from the real-time clock.

Prototype

```
int RFCRGetRTCTime(void);
```

Return Values

hhmm	On success returns the current time in `hhmm` format
< 0	Error

RFCRSetRTCDate()

Sets the real-time clock date.

Prototype

```
int RFCRSetRTCDate(int year, int month, int day);
```

Parameters

`year`	Year value.
`month`	Month value.
`day`	Day value.

Return Values

= 0	Success
< 0	Error

RFCRGetRTCDate()

Returns the current date set in the real-time clock.

Prototype

```
int RFCRGetRTCDate(void);
```

Return Values

yyyymmdd	On success, returns the current date in yyyymmdd format
< 0	Error

RFCRSetBaudRate()

Sets the baud rate. If the RFCR supports the specified baud rate, it returns an ACK response and then switches to the specified baud rate.

Prototype

```
int RFCRSetBaudRate(int baud);
```

Parameters

`baud`	Baud rate. • 01h 9600 baud • 02h 19200 baud • 03h 38400 baud • 04h 57600 baud • 05h 115200 baud

Return Values

= 0	Success
< 0	Error

RFCRSetRTCSource()

Sets the source for RTC/LCD/Buzzer/LED on the RFCR. Configure the reader to use an internal or external source or both an internal and external source, except for RTC.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRSetRTCSource(int mode);
```

Parameters

`mode`	The definition of the mode is given in *Contactless Software Programmers Guide* (P/N 26807).

Return Values

= 0	Success
< 0	Error

RFCRGetRTCSource()

Retrieves the source for the RTC/LCD/Buzzer/LED on the RFCR.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetRTCSource(void);
```

Return Values

= 0	Success, definition of return is given in *Contactless Software Programmers Guide* (P/N 26807).
< 0	Error

RFCRGetType()

Retrieves the RFCR type.

Prototype

```
int RFCRGetType(void);
```

Return Values

1	Gen1
2	Gen2
3	Gen3
99	Unknown type

RFCRLedControl()

Controls the LEDs when the RFCR is in Pass Through mode.

Prototype

```
int RFCRLedControl(int led, int mode);
```

Parameters

`led`	• 0 – LED 0 • 1 – LED 1 • 2 – LED 2 • FF – All three LEDs
`mode`	• 0 – LED Off • 1 – LED On

Return Values

= 0	Success
< 0	Error

RFCRBuzzerControl()

Enables the RFCR buzzer only when the RFCR is in Pass Through mode.

Prototype

```
int RFCRBuzzerControl(int mode, int beeps);
```

Parameters

`mode`	• 1 – N short beeps • 2 – single long beep
`beeps`	• Number of beeps (for 1) • Duration (for 2)

Return Values

= 0	Success
< 0	Error

RFCRGetConfigurableAID()

Reads the configurable AID parameters. The application must send an AID TLV as the first TLV in the command. The reader then returns all tags associated with that AID.

Prototype

```
int RFCRGetConfigurableAID(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllAIDs()

Read all AIDs in the RFCR to verify the configured AIDs in the reader.

Prototype

```
int RFCRGetAllAIDs(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetConfigurableGroup()

Reads the configurable Group EMV parameters. The application must send a Group tag as the only TLV in the command. The reader then returns all tags associated with this configurable Group.

Prototype

```
int RFCRGetConfigurableGroup(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllGroups()

Reads all groups in the RFCR to verify all configured groups in the reader.

Prototype

```
int RFCRGetAllGroups(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRWriteDataCommand()

Stores a ticket on the card. When a valid Write data command is sent to the RFCR, it attempts to carry out payment at the exit transaction.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRWriteDataCommand(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are given in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRSetConfigurableAID()

Creates or selects an AID for configuration or deletion. There are seven TLVs that can be included in this command; some are mandatory.

Prototype

```
int RFCRSetConfigurableAID(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetConfigurableGroup()

Creates or selects a group for configuration. The application can configure a specific group by communicating tags that provide the desired functionality associated with that group to the RFCR.

Prototype

```
int RFCRSetConfigurableGroup(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSDeleteConfigurableAID()

Deletes or disables a configurable AID. The AID TLV of the AID to remove must be included. Include no other TLVs.

Prototype

```
int RFCRSDeleteConfigurableAID(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteConfigurableGroup()

Deletes a configurable group. The group can then no longer be used to load the parameters for a transaction.

Prototype

```
int RFCRDeleteConfigurableGroup(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRGetBuild()

Reads the firmware build ID in the RFCR.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetBuild(char* verString, int verMax);
```

Parameters

`verString`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`verMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllVariables()

Reads all RFCR variables.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetAllVariables(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetProductType()

Reads the RFCR product type.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetProductType(char* proId, int proMax);
```

Parameters

`proId`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` should be able to accept the maximum possible size of the data frame.
`proMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetProcessorType()

Reads the RFCR processor type.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetProcessorType(char* proId, int proMax);
```

Parameters

`proId`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`proMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetMainFwVersion()

Reads the RFCR main firmware version.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetMainFwVersion(char* fwId, int fwMax);
```

Parameters

`fwId`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`fwMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetFwSubsystemSuite()

Reads the RFCR firmware subsystem suite.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetFwSubsystemSuite(char* fwId, int fwMax);
```

Parameters

`fwId`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`fwMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetSerialProtocolSuite()

Reads the RFCR serial protocol suite.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetSerialProtocolSuite(char* serPro, int serMax);
```

Parameters

`serPro`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`serMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetPayPassVersion()

Reads the RFCR Layer 1 Pay Pass version.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetPayPassVersion(char* payPas, int payMax);
```

Parameters

`payPas`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`payMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAntiCollisionResolution()

Reads the RFCR Layer 1 Anti-Collision Resolution version.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetAntiCollisionResolution(char* acrVer, int arcMax);
```

Parameters

`acrVer`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`arcMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetCardApplicationSuite()

Reads the RFCR Layer 2 Card Application suite.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetCardApplicationSuite(char* appSuite, int appMax);
```

Parameters

`appSuite`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`appMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetUserExperienceSuite()

Reads the RFCR User Experience suite.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetCardApplicationSuite(char* usrExp, int usrMax);
```

Parameters

`usrExp`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`usrMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetSystemInformationSuite()

Reads the RFCR System Information suite.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetSystemInformationSuite(char* sysInfo, int sysMax);
```

Parameters

`sysInfo`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`sysMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetCAPublicKey()

Reads the CA Public Key from the RFCR.

Prototype

```
int RFCRGetCAPublicKey(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetSerialNumber()

Reads the 14-digit serial number from the RFCR stored in its non-volatile memory.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetSerialNumber(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but the `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRSetSerialNumber()

Stores the 14-digit serial number in the RFCR non-volatile memory. If a serial number is already set in the reader, this command fails with command-not-allowed error status.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRSetSerialNumber(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRFlushTrackData()

Flushes any track data previously read from a card but not yet transferred to the terminal. The RFCR clears all pending card data.

Prototype

```
int RFCRFlushTrackData(void);
```

Return Values

= 0	Success
< 0	Error

RFCRSetRfErrorReporting()

Enables or disables RF error reporting for the Get Full Track Data command. If there is any RF error code, it is reported to the terminal through the ACK frame for the Get Full Track Data command.

NOTE This command is not supported in Gen3 architecture.

Prototype

```
int RFCRSetRfErrorReporting(int mode, char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.
`mode`	• 0 – Disable RF Error Reporting • 1 – Enable RF Error Reporting

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetUSBBootLoaderVersion()

Retrieves the ViVOpay USB Boot Loader version from the RFCR.

NOTE

This command is not supported in Gen3 architecture.

Prototype

```
int RFCRGetUSBBootLoaderVersion(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRStoreLCDMessage()

Stores the LCD message in non-volatile memory of the reader (only English messages are supported).

Prototype

```
int RFCRStoreLCDMessage(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRGetLCDMessage()

Sends messages stored in the RFCR EEPROM to the terminal (English only).

Prototype

```
int RFCRGetLCDMessage(int msgId, char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.
`msgId`	Message ID or FF (request all saved messages).

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAdditionalAIDParams()

Retrieves additional AID parameters that do not exist in VIVOpay specifications per specific AID.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetAdditionalAIDParams(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetRevocationParam()

Retrieves all Revocation structure parameters per required RID.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetRevocationParam(char* cmdData, int cmdLen, char* rspData,
int rspMax);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.
`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllAdditionalAIDParams()

Retrieves all additional AID parameter structures in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetAllAdditionalAIDParams(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllRevocationParams()

Retrieves all Revocation parameter structures in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetAllRevocationParams(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAllExceptionParams()

Retrieves all exception PANs in the RFCR.

NOTE This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetAllExceptionParams(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetAdditionalReaderParams()

Retrieves all additional reader parameters in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetAdditionalReaderParams(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but the `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetVFIVersion()

Retrieves the software release version or the list of software components and version numbers.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetVFIVersion(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetTypeApprovalVersions()

Returns a string containing version information grouped by card brand as they appear on each brand's reader approval.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetTypeApprovalVersions(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRSetExistAdditionalAIDParams()

Updates existing additional AID parameters.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetExistAdditionalAIDParams(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetNewAdditionalAIDParams()

Adds AID parameters.

NOTE This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetNewAdditionalAIDParams(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteAdditionalAIDParams()

Deletes an additional AID parameter structure stored in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRDeleteAdditionalAIDParams(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetExistRevocationParam()

Updates existing Revocation structure parameters stored in the RFCR.

NOTE This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetExistRevocationParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetNewRevocationParam()

Adds a Revocation structure parameter to the RFCR.

NOTE This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetNewRevocationParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteRevocationParam()

Deletes a Revocation structure parameter stored in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRDeleteRevocationParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetNewExceptionParam()

Adds a PAN to the exception PANs list in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetNewExceptionParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRDeleteExceptionParam()

Deletes an Exception PAN from the Exception PANs list stored in the RFCR.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRDeleteExceptionParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRSetAdditionalReaderParam()

Updates existing additional RFCR parameters.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRSetAdditionalReaderParam(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRRestoreDefaultConfig()

Restores the default reader configuration.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRRestoreDefaultConfig(char* cmdData, int cmdLen);
```

Parameters

`cmdData`	Pointer to the data field in the command packet. The format and contents of the data field are in the *Contactless Software Programmers Guide* (P/N 26807).
`cmdLen`	Length of the data field in the command packet.

Return Values

= 0	Success
< 0	Error

RFCRGetFwFullVersion()

Returns a string containing version information for each software module for individual card types.

NOTE

This command is not supported in ViVOpay Gen1 and Gen2 architecture.

Prototype

```
int RFCRGetFwFullVersion(char* rspData, int rspMax);
```

Parameters

`rspData`	Pointer to store the ACK frame. The number of bytes returned may vary, but `buff` must be able to accept the maximum possible size of the data frame.
`rspMax`	Maximum size of `buff`.

Return Values

> 0	Success with the number of bytes read
< 0	Error

RFCRGetBaudRate()

Retrieves the current baud rate set in the RFCR.

Prototype

```
int RFCRGetBaudRate(void);
```

Return Values

> 0	Success, readers baud rate: 9600, 19200, 38400, 57600, 115200
< 0	Error

RFCRChangeBaudRate()

Sets or changes the baud rate in the RFCR.

Prototype

```
int RFCRChangeBaudRate(int iBaud);
```

Parameters

`iBaud`	Baud rate: 9600, 19200, 38400, 57600, 115200

Return Values

> 0	Success and the device handle is returned.
< 0	Error

RFCR Return Values

Along with the generic error returns in `errno.h`, specific RFCR error returns are listed below (most are from the `RFCRUpdateFW()` call).

Error	Value
No File	-401
Bad File	-402
Enter ISP Error	-403
Autobaud Error	-404
Frequency Error	-405
Erase Error	-406
Receive Timeout	-407
File Read Error	-408
Big Record Error	-409
Burn Error	-410
Cleanup Error	-411
Invalid Data Frame	-420
Invalid Card Payload	-422
Buffer Too Small	-430

RFCR Protocol Compatibility Table

The following table describes compatibility with contactless protocol APIs implemented in different contactless modules based on ViVOpay (GEN1, GEN2) and Verifone (GEN3) architecture.

Table 11 Protocol Compatibility

RFCR Function Call	ALL	GEN1	GEN2	GEN3
`RFCRlibVersion()`	X			
`RFCRInit()`	X			
`RFCRClose()`	X			
`RFCRGetVersion()`	X			
`RFCRPing()`	X			
`RFCRReset()`	X			
`RFCRSetAntenna()`	X			
`RFCRSetIndicator()`		X		
`RFCRGetCardPayload()`	X			
`RFCRParseCardPayload()`	X			
`RFCRUpdateFW()`	X			
`RFCRPurge()`	X			
`RFCRInputPending()`	X			
`RFCRRawWrite()`	X			
`RFCRRawRead()`	X			
`RFCRAddCRC()`	X			
`RFCRCheckCRC()`	X			
`RFCRReceiveACKFrame()`	X			
`RFCRReceiveDataFrame()`	X			

Table 11 Protocol Compatibility (continued)

RFCR Function Call	ALL	GEN1	GEN2	GEN3
`RFCRSetPollMode()`			X	X
`RFCRGetTransactionResult()`			X	X
`RFCRActivateTransaction()`			X	X
`RFCRActivateMxTransaction()`			X	
`RFCRDebitWrite()`			X	
`RFCRCancelTransaction()`			X	X
`RFCRGetFullTrackData()`			X	X
`RFCRUpdateBalance()`			X	X
`RFCRSetEMVParameters()`			X	X
`RFCRGetEMVParameters()`			X	X
`RFCRSetBurstMode()`			X	X
`RFCRSetPassThroughMode()`			X	X
`RFCRPollForToken()`			X	X
`RFCRIsoApduExchange()`			X	X
`RFCRPcdSingleExchange()`	X			
`RFCRGetPcdPiccParam()`			X	X
`RFCRMifareAuthenicateBlock()`			X	X
`RFCRMifareReadBlocks()`			X	X
`RFCRMifareWriteBlocks()`			X	X
`RFCRMifarePurseCommand()`			X	
`RFCRHighLevelHaltCommand()`			X	X
`RFCRSetCAPublicKey()`	X			
`RFCRDeleteCAPublicKey()`	X			
`RFCRDeleteAllCAPublicKey()`			X	X
`RFCRSetRTCTime()`	X			
`RFCRGetRTCTime()`	X			
`RFCRSetRTCDate()`	X			
`RFCRGetRTCDate()`	X			
`RFCRSetBaudRate()`			X	X
`RFCRSetRTCSource()`			X	
`RFCRGetRTCSource()`			X	
`RFCRGetType()`	X			
`RFCRLedControl()`			X	X
`RFCRBuzzerControl()`			X	X
`RFCRGetConfigurableAID()`			X	X
`RFCRGetAllAIDs()`			X	X
`RFCRGetConfigurableGroup()`			X	X
`RFCRGetAllGroups()`			X	X
`RFCRWriteDataCommand()`			X	
`RFCRSetConfigurableAID()`			X	X

Table 11 Protocol Compatibility (continued)

RFCR Function Call	ALL	GEN1	GEN2	GEN3
RFCRSetConfigurableGroup()			X	X
RFCRSDeleteConfigurableAID()			X	X
RFCRDeleteConfigurableGroup()			X	X
RFCRGetBuild()			X	
RFCRGetAllVariables()			X	
RFCRGetProductType()			X	
RFCRGetProcessorType()			X	
RFCRGetMainFwVersion()			X	
RFCRGetFwSubsystemSuite()			X	
RFCRGetSerialProtocolSuite()			X	
RFCRGetPayPassVersion()			X	
RFCRGetAntiCollisionResolution()			X	
RFCRGetCardApplicationSuite()			X	
RFCRGetUserExperienceSuite()			X	
RFCRGetSystemInformationSuite()			X	
RFCRGetCAPublicKey()			X	X
RFCRGetSerialNumber()			X	
RFCRSetSerialNumber()			X	
RFCRFlushTrackData()			X	X
RFCRSetRfErrorReporting()		X	X	
RFCRGetUSBBootLoaderVersion()		X	X	
RFCRStoreLCDMessage()			X	X
RFCRGetLCDMessage()			X	X
RFCRGetAdditionalAIDParams()				X
RFCRGetRevocationParam()				X
RFCRGetAllAdditionalAIDParams()				X
RFCRGetAllRevocationParams()				X
RFCRGetAllExceptionParams()				X
RFCRAdditionalReaderParms()				X
RFCRGetVFIVersion()				X
RFCRGetTypeApprovalVersions()				X
RFCRSetExistAdditionalAIDParams()				X
RFCRSetNewAdditionalAIDParams()				X
RFCRDeleteAdditionalAIDParams()				X
RFCRSetExistRevocationParam()				X
RFCRSetNewRevocationParam()				X
RFCRDeleteRevocationParam()				X
RFCRSetNewExceptionParam()				X
RFCRDeleteExceptionParam()				X
RFCRSetAdditionalReaderParam()				X

Table 11 Protocol Compatibility (continued)

RFCR Function Call	ALL	GEN1	GEN2	GEN3
`RFCRRestoreDefaultConfig()`	X			
`RFCRGetFwFullVersion()`			X	X
`RFCRGetBaudRate()`	X			
`RFCRChangeBaud Rate()`			X	X

PCI 4 Support

APPENDIX C

V/OS on the UX Terminals

The following appendices contain V/OS information specific to the UX terminal. For these terminals, this information supersedes that in the main body of this document.

The UX terminal family includes:

- UX100 PINpad with display and UX110 PINpad without display, details in Appendix D
- UX300 card reader, details in Appendix E

NOTE

See Appendix F for details about the VX 520 and UX300 Sysmode Application.

This appendix contains topics general to UX terminals, including:

- Device Configuration for Development
- Application Development Package (ADK)
- Error Codes and errno Table
- Power Management

Device Configuration for Development

For application development, connect the UX100 or UX110 PINpad to the UX300 through the main USB port, located on the terminal base (Figure 15).

For UX300 console access, connect through the serial port to a PC.

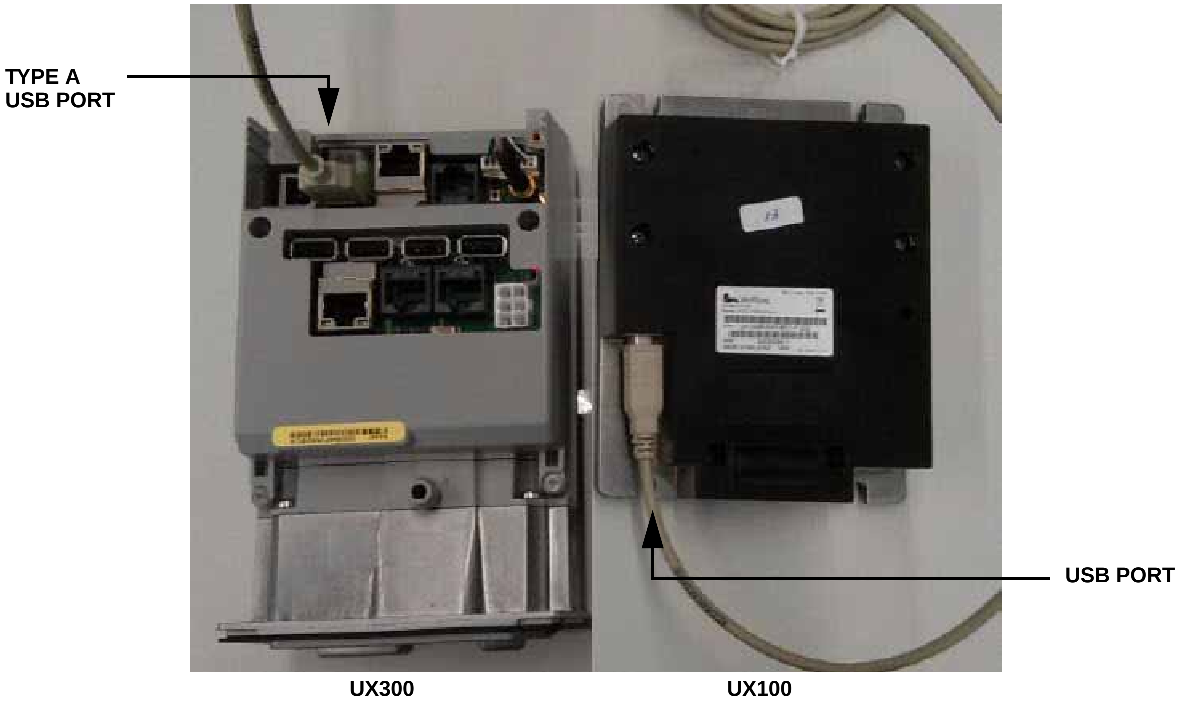


Figure 15 UX Development Configuration

Anti-Removal Switches (ASRs)

The UX300 card reader and UX100 and UX110 PINpads have two anti-removal switches. When the ASRs are triggered and the device is not installed, PIN entry is rejected and applications will not start automatically at device boot-up. When all four ASRs are pressed (UX300 and connected UX1XX PINPad), Sysmode provides an opportunity to reset the ASR state. When ASR trigger status displays before the Sysmode login prompt, press <OK> to reset the ASRs.

You will have to enter your ASR password. If you don't have your ASR password, reset the password to the default:

1. Press the Sysmode button on the terminal.
2. Log in to Sysmode as supervisor.
3. Select **Security > Password Mgr > Expire > Switch Passwords**.
4. Restart Sysmode and reset the ARS using the following predefined default passwords:
   - 4275386
   - 2615948
5. Enter a new password at the prompt.

Application Development Package (ADK)

The ADK package:

- Enables serial console access on COM1
- Installs the developer tools (gdbserver, strace, and so on)
- Runs the Telnet server (for access to more virtual consoles)
- Allows mounting of remote NFS shares or direct execution or installation of applications through NFS share on Windows or Linux

Error Codes and errno Table

No.	Function Status	errno code	Explanation
1	OK	0	
2	Bad parameter	EINVAL	
3	Code Error	EPERM	
4	Keyboard Key disabled	EACCES	
5	Key not present	ENODATA	Key missing for authentication, PIN, and so on.
6	Key verification NOK	EFAULT	Key type incorrect.
7	Other key issue	ENOTBLK	
8	Function parameter missing	EINVAL	
9	Key bad MAC check	ETXTBSY	
10	Bad data format, or PIN format	ENOEXEC	
11	Challenging error	EXDEV	
12	Required key missing	EBADF	
13	Key press timeout	ETIME	
14	PIN timeout	ETIME	
15	No PIN mode for PIN session error; Invalid request code for PIN delete	EBADRQC	
16	Error code for the COR key in PIN mode	EINVAL	
17	PIN length definition error	EINVAL	
18	Invalid key press for PIN	EINVAL	
19	Unit is busy	EBUSY	
20	Keyboard initialization failure	EIO	
21	Keyboard parameter error	EIO	
22	General error	EPERM	
23	Unknown error	EIO	

Power Management

As detailed in Chapter 6, Standby and Suspend power management modes are available on all V/OS devices. Shutdown mode is also available on the UX300. The Power Management Service provides an API to control power management. Power management is disabled by default; applications must control it. Refer to Power Management for functions to force the system into standby or suspend mode.

Use the APM service to allow the application to stop the system from entering Suspend mode during critical operations such as PIN entry or host transactions.

Use the powermngt_declare_state() function to declare the current state of application. Application must change the state to IDLE once it is no longer busy. For example:

```
ret = powermngt_declare_state(getpid(), BUSY_FULL);
ret = powermngt_declare_state(getpid(), BUSY_BACKGROUND);
ret = powermngt_declare_state(getpid(), IDLE);
```

Do not use powermngt_CritcalSectionEnter() or powermngt_CritcalSectionExit() when writing applications for UX terminals.

APPENDIX D

UX100 and UX110 Terminals

This section contains information specific to the UX100 PINpad with display, and UX110 PINpad without display.

For these terminals, this information supersedes that in the main body of this document. Topics include:

- Firmware Overview
- Sysmode
- Password Management
- Terminal Updates
- Display Functions
- Buzzer Functions
- PIN Entry Functions
- Info Functions
- Security Functions
- Manufacturer Functions
- Performing a Card Reader Reset
- Multi Drop Bus
- Remote Printer

Firmware Overview

The UX 100 firmware is made of device drivers. Each device driver calls a device task to process time-consuming commands and allow simultaneous device command processing. Quick commands are directly processed by the device driver without using tasks. For example, the secure device uses secure tasks to process RSA computations, but it answers directly to a version information request. The following are the device drivers:

- The Main task waits for incoming command requests and dispatches them to the destination device or returns an error if the destination device is cannot be accessed. The Main task also handles the buzzer, cell coin, and temperature sensor.
- The Keyboard task handles the UX PINpad.
- The Display task handles the LCD display.
- The Security tasks run as separate tasks in parallel time-consuming processes such as RSA computations, PIN entry, and so on.

- The Anti-removal switch (ARS) task handles the anti-removal switch.

Each time a is task called though the UX service, there is a wait time for the device answer before it accepts calls again. Two threads cannot call the same device at the same time.

Sysmode

This section describes entering Sysmode.

UX 100

Use the arrow keys to access the Sysmode menus.

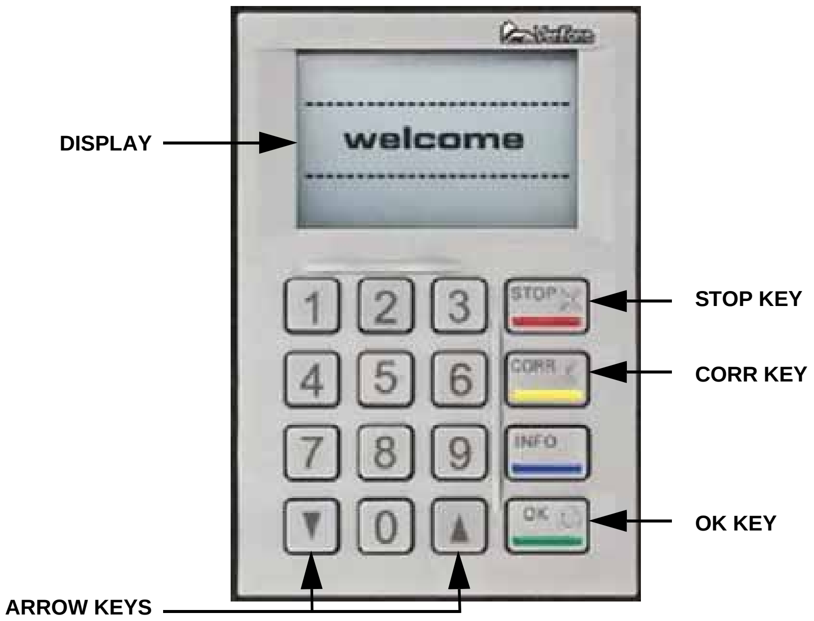


Figure 16 UX100 Display and Keypad

- Use the arrow keys to navigate menus
- Use the OK key to select the menu option or answer yes to questions
- Use the STOP and CORR to cancel an action, leave a menu, or answer no to questions

UX 110

Connect the UX110 to a UX300 to access Sysmode using the *ncurses* library. Connections are either USB, standard serial or Ethernet. See also, VX 520 and UX300 Sysmode Application.

To configure the UX300, you must first create a sys4 user configuration file. File content depends on the desired connection, as described in the following sections. Create the file using a user config secins download file.

NOTE

The file must be created for the sys4 user, and signed with the OS signer. See Secure Installer.

USB Connection Configuration

To enable the USB connection:

1 Create the sys4 user configuration file with the following content:

```
?
[serial]
port=/dev/ttyGS0
```

2 Install the configuration file using a configuration package (see Secure Installer):

```
/mnt/flash/config/config.sys4
```

3 Connect a USB cable to the UX300 and the PC host port.

NOTE Ensure that you are using the latest USB Serial gadget driver for Windows, included in the WinSDK package.

4 Connect the UX110 to the UX300.

5 Connect to the virtual serial port using PuTty or another TTY client.

6 Ensure that the assigned COM port in Windows Device Manager is assigned dynamically.

7 Boot the UX300.

NOTE This configuration is ignored when using a USB-to-Serial converter.

Serial Connection Configuration

To enable the serial connection:

1 Create the sys4 user configuration file with the following content:

```
?
[serial]
port=/dev/ttyAMA0
```

2 Install the configuration file using a configuration package (see Secure Installer):

```
/mnt/flash/config/config.sys4
```

3 Connect a serial cable to the UX300.

4 Connect the UX110 to the UX300.

5 Connect to over the Serial port using PuTty or TTY other client.

6 Boot the device.

NOTE

This configuration is ignored when using a USB-to-Serial converter.

USB Connection Configuration

If you configure the USB port, output is available on all interfaces (eth0 and eth1), because the server listens for a connection on all interfaces.

To enable the serial connection:

1. Create the sys4 user configuration file with the following content:

```
?
[net]
port=7777
```

2. Install the configuration file using a configuration package (see Secure Installer):

```
/mnt/flash/config/config.sys4
```

3. Connect a serial cable to the UX300.
4. Connect the UX110 to the UX300.
5. Connect the Ethernet cable
6. Ensure that the Ethernet connection is available on UX300.
7. Boot the device.
8. Establish the connection using the Linux “nc” command:

```
?
#nc 10.184.8.12:7777
```

NOTE

This configuration is ignored when using a USB-to-Serial converter.

Password Management

The UX terminal supports three password groups:

- USERS – Log in password for the supervisor, level1, and level2 users
  - Supervisor – Access to all Sysmode menus
  - level1, level2, maintenance – Can only access the Sysmode menus allowed by the Security Policy File
- KEYLOAD – keyload1 and keyload2 passwords for key injection
- SWITCH – switch1 and switch2 passwords to activate devices when an Anti-Removal Switch is pressed

You are prompted to set passwords when the device first boots. Use the keypad and press **OK** to set the new password. Each password has an expiration set at manufacture. Save the password in a secure location.

NOTE

level2 passwords cannot be recovered. UX terminals must be returned to Verifone or an authorized repair center if the level2 password (customer's supervisor) is lost.

Security Policy File

This file contains the user access definitions:

- level1 has access to the ARS menu and can change ASR passwords.
- level2 (usually the customer's supervisor user) has level1 access, plus access to the Expire Switch Password menu to reset ASR passwords to their default value.
- maintenance only has access the Information menu, Diagnostic menu, and Configuration/Status menu.

Reset Passwords

Use the following procedure to reset the USER password.

Use the arrow keys to navigate through the Sysmode menus. Press **OK** after making each selection.

1. Select **Login >** ***user***.
2. Enter the current password for the selected user.
3. Select **Security > Password Mgr**.
4. Select **Expire > Switch Passwords**.
5. Press **OK** at the prompt to select Yes.

   The switch password is reset. Use the arrow key to select No and cancel the password reset.

Terminal Updates

To update the UX100 and UX110 PINpads:

1. Select the desired tar package from the **Update** menu (for example, DL.FWUX.01.00.XX.00XX.WW01_(prod).tgz).

   Installation takes approximately 5 minutes.

2. Choose **Yes** at the restart prompt.

   The terminal will flash and beep during installation.

NOTE

The terminal may reboot during installation. If the upload fails, reinstall the update package on another terminal and connect the UX100 terminal that failed.

The menu displays on update success.

3 Log in as supervisor, and select **Pinpad > Software** and scroll to verify the current version.

The last entry is the current build, which should correspond to the package chosen in Step 1.

Display Functions

This section presents the APIs to drive the display using `svc_ux100`.

errno Values

- EINVAL – A general display error.
- EPERM – Display initialization failed.
- BUSY – The device is busy.

ux100_dispGetInfo()

Returns the display settings.

Prototype

```
struct ux100DisplayInfoStruct ux100_dispGetInfo(void);
```

Return Values

0 =	ux100KeybdInfoStruct is filled.
-1 =	Error with errno set.

Example

```
struct ux100DisplayInfoStruct {
Int currentBacklightMode, // curent backlight setting of the display
Int currentBklTime, // current backlight timer in ms
Int currentContrast, // current contrast of the display
Int MaxBacklight, // maximum backlight setting for the display
Int MaxContrast, // maximum contast setting for the display
Int MinBacklight, // minimum backlight setting for the display
Int MinContrast, // minumim contrast setting for the display
Int ScreenHeight, // Heigth of the screen in pixels
Int ScreenWidth, // Width of the screen in pixels
Int ScreenColorDepth //Depth of the screen color if any
}
```

ux100_dispSetContrast()

Sets the contrast of the display.

Prototype

```
int ux100_dispSetContrast(int level);
```

Parameters

`level`	The level of contrast.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_dispSetBacklightMode()

Selects the LCD backlight mode:

- OFF mode – Deactivates the backlight. The `time` parameter is not used.
- ON mode – Activates the backlight during the specified time.
- AUTOMATIC mode – Activates the backlight during the specified time if a key is pressed or the display updates.

Prototype

```
int ux100_dispSetBacklightMode(int mode, unsigned int time);
```

Parameters

`mode`	Sets the backlight mode: • 0 = OFF mode • 1 = ON mode • 2 = AUTOMATIC mode
`time`	Time in milliseconds that the backlight is on. 0 = the backlight remains on infinitely.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

Buzzer Functions

This section presents the APIs for the buzzer interface using the PINpad service `svc_ux100`.

ux100_buzzerSetBeep()

Activates the buzzer for the specified time, at a specific frequency and volume.

NOTE Do not call this function during the PIN entry.

Prototype

```
int ux100_buzzerSetBeep(unsigned int frequency, unsigned int time,
unsigned int volume);
```

Parameters

`frequency`	Sets the buzzer frequency in Hz.
`time`	Time in milliseconds that the buzzer is on.
`volume`	Sets the buzzer volume. Valid values are 0–10. 0 = no buzzer. **Note:** This parameter is not used on the UX100.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_buzzerGetInfo()

Returns buzzer settings.

Prototype

```
struct ux100BuzzerInfoStruct ux100_buzzerGetInfo(void);
```

Return Values

0 =	The ux100BuzzerInfoStruct is filled.
≠ 0	errno indicates the error type.

Example

```
struct ux100BuzzerInfoStruct {
unsigned int currentFrequency, // current frequency of the buzzer
unsigned int currentTime, // current time (duration) of the buzzer
unsigned int currentVolume, // current volume of the buzzer
}
```

ux100_buzzerSetKeyboardBeep()

Defines the frequency, volume, and duration of a beep on a key press.

NOTE

Do not call this function during the PIN entry.

Prototype

```
int ux100_buzzerSetKeyboardBeep(unsigned int frequency, unsigned int time,
unsigned int volume);
```

Parameters

`frequency`	Sets the buzzer frequency in Hz.
`time`	Time in milliseconds that the buzzer is on.
`volume`	Sets the buzzer volume. Valid values are 0–10. 0 = no buzzer. **Note:** This parameter is not used on the UX100.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_buzzerGetKeyboardBeepInfo()

Returns the defined frequency, volume, and duration settings for the beep on a key press.

Prototype

```
struct ux100BuzzerInfoStruct ux100_buzzerGetKeyboardBeepInfo(void);
```

Parameters

`frequency`	Sets the buzzer frequency in Hz.
`time`	Time in milliseconds that the buzzer is on.
`volume`	Sets the buzzer volume. Valid values are 0–10. 0 = no buzzer. **Note:** This parameter is not used on the UX100.

Return Values

0 =	The ux100BuzzerInfoStruct is filled, as shown in the following example.
-1 =	errno indicates the error type.

Example

```
struct ux100BuzzerInfoStruct {
unsigned int currentFrequency, current frequency in Hertz.
unsigned int currentTime, Time in milliseconds. 0 means no sound
unsigned int currentVolume, Volume. Range from 0 to 10.
}
```

PIN Entry Functions

This section presents the PIN management APIs that interface with the PINpad service `svc_ux100`.

NOTE It is recommended to use the ADK tools to set up PIN entry.

errno Values

Table 12 lists the errno values returned in the PIN management APIs:

Table 12 PIN Entry errno Descriptions

Code	Description
EINVAL	PIN error.
ETIME	Timeout.
ENODATA	PIN encryption key error
EIO	PIN length error
EPERM	Key press was ignored. See ux100_pinDelete().

ux100_pinStartPinEntry()

Starts the PIN entry process and enables PIN entry mode. This function checks that the minimum and the maximum PIN length(4 and 12, respectively) for the PIN entry is PCI compliant. It also sets the timeout for PIN entry. The maximum timeout allowed is 5 minutes. Timeout settings longer that 5 minutes are forced to meet this maximum. You can have only one open PIN session. An error returns if the PINpad is already in PIN entry mode. The PINpad must be in Field mode to allow PIN entry.

Prototype

```
int ux100_pinstartpinentry(unsigned char pinLenMin,
unsigned char pinLenMax, int timeout);
```

Parameters

`pinLenMin`	Sets the minimum number of digits allowed for a PIN entry.
`pinLenMax`	Sets the maximum number of digits allowed for a PIN entry.
`timeout`	PIN entry session timeout in milliseconds.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_pinGetNbCurrDigitEntered()

Returns the number of the PIN digits entered.

Prototype

```
int ux100_pinGetNbCurrDigitEntered(void);
```

Return Values

0 =	Returns the number of the PIN digits entered.
≠ 0	errno indicates the error type.

ux100_pinDelete()

Deletes the last PIN digits entered or all in the buffer. PIN digit deletion is handled by the GUI. Until the PINpad receives this call, key presses are ignored. As soon as the CORRECTION key is pressed, the PINpad no longer expects to receive any ux100_keybdGetKey() until it receives ux100_pinDelete(). If the PINpad receives a ux100_keybdGetKey() during this time, errno is immediately set to EBADRQC, and the key pressed is discarded from the PIN buffer.

Prototype

```
int ux100_pinDelete(int nbDigit);
```

Parameters

`nbDigit`	Sets the PIN digits to delete: • 0 = delete all entered PIN digits • 1 = delete the last digit entered

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_pinAbort()

Stops the current PIN entry process. When this call is sent, the PED erases the PIN buffer and resets the keyboard to non PIN entry mode. If a PIN is stopped, by the time the PINpad handles the request, it is possible that a false key is in the key’s FIFO. To avoid this, call ux100_keybdFlush().

Prototype

```
int ux100_pinAbort(void);
```

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_pinSendEncPinBlockToVault()

Sends the PIN to the vault. On a VAL key press, this function has an approximate 30-second timeout to get the PIN block. It always returns the VAL key value, but also generates the PIN block only when the PIN length is within the PIN digit length limit. The VAL and COR keys are used for PIN bypass. When one of these keys is pressed and the PIN buffer is empty, the firmware returns a PIN bypass code.

Prototype

```
int ux100_pinSendEncPinBlockToVault(struct ux100PinBlockStruct
pinBlockSt);
```

Parameters

```
struct ux100PinBlockStruct
{ void* data; // Pin block data.
int data_count; // Number of bytes in data.
};
```

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_pinEntrymode()

Retrieves the PIN mode information from the PINpad.

Prototype

```
int ux100_pinEntrymode(void);
```

Return Values

0 =	The PINpad is in non PIN mode.
1 =	The PINpad is in PIN mode.
-1 =	Cannot perform the request. errno indicates the error type.

Info Functions

This section presents the functions to drive the display using `svc_ux100`.

ux100_getVersion()

Returns the `svc_ux100` service version.

Prototype

```
struct version int ux100_getVersion(void);
```

Return Values

0 =	The version structure is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct version {
int major;
int minor;
int maint;
char build[16];
};
```

ux100_infoGetDeviceList()

Returns the list of the devices on the PINpad.

Prototype

```
int ux100_infoGetDeviceList(void);
```

Return Values

0 =	n = The bit field value, as shown in the following example.
≠ 0	errno indicates the error type.

Example

```
typedef enum
{
UX100_MASK_NONE = 0,
UX100_MASK_MAIN = 1 << 0, /**<Main secure target ID */
UX100_MASK_KYBD = 1 << 1, /**< keyboard functions in the target*/
UX100_MASK_SEC = 1 << 2, /**< Security functions present in the target */
UX100_MASK_DISP = 1 << 3, /**< display functions in the target*/
UX100_MASK_SWITCH = 1 << 4, /**< switch functions in the target*/
UX100_MASK_ALL = 0xFFFFFFFF,
} UX100_DEVICE_ID;
```

ux100_infoGetSecureVersion()

Obtains the software version of the firmware running on the PINpad.

Prototype

```
struct ux100SecureVersionStruct ux100_infoGetSecureVersion(void);
```

Return Values

0 =	The ux100SecureVersionStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100SecureVersionStruct
{
char id_code[50]; /** Software package name */
char family[10]; /** PCI or equivalent certification ID */
char version[12]; /** version */
};
```

ux100_infoGetPartNumber()

Returns the hardware part number of the PINpad.

Prototype

```
struct ux100PartNumberStruct ux100_infoGetPartNumber(void);
```

Return Values

0 =	The ux100PartNumberStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100PartNumberStruct
{
char part_number[32+1]; // ux100 part number
};
```

ux100_infoGetSerialNumber()

Returns the PINpad serial number.

Prototype

```
struct ux100SNStruct ux100_infoGetSerialNumber(void);
```

Return Values

0 =	The ux100SNStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100SerialNumberStruct
{
char serial_number[11+1]; // ux100 serial number
};
```

ux100_infoGetHardwareVersion()

Returns the PINpad hardware version.

Prototype

```
struct ux100HardVerStruct ux100_infoGetHardwareVersion(void);
```

Return Values

0 =	The ux10HardVerStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux10HardVerStruct
{
char hardware_version[15+1]; // ux100 hardware version.
};
```

ux100_infoGetPCIHardwareVersion()

Returns the PINpad PCI hardware version.

Prototype

```
struct ux100PCIHardVerStruct ux100_infoGetPCIHardwareVersion(void);
```

Return Values

0 =	The ux10PCIHardVerStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux10PCIHardVerStruct
{
char pci_hardware_version[4+1]; // ux100 PCI hardware version.
};
```

ux100_infoGetModelNumber()

Returns the PINpad model number.

Prototype

```
struct ux100ModelNumberStruct ux100_infoGetModelNumber(void);
```

Return Values

0 =	The ux100ModelNumberStruct is filled, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ModelNumberStruct
{
char model_number[12+1]; // ux100 model number
};
```

ux100_infoGetCountryName()

Returns the PINpad country name set at manufacture.

Prototype

```
struct ux100CountryNameStruct ux100_infoGetCountryName(void);
```

Return Values

0 =	The ux100CountryNameStruct is filled, as show in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100CountryNameStruct
{
char country_name [12+1]; //ux100 Country name
};
```

ux100_infoGetPairingStatus()

Returns the pairing status between the PINpad and the terminal.

Prototype

```
int ux100_infoGetPairingStatus(void);
```

Return Values

0 =	n = The status of the pairing: • 0 = The pairing is done and status is OK. • 1 = The pairing failed due to missing keys in the PINpad. • 2 = The pairing failed due to missing keys in the terminal. • 3 = The pairing failed due to bad certificate in the PINpad. • 4 = The pairing failed due to bad certificate in the terminal.
≠ 0	errno indicates the error type.

ux100_infoGetRemSwStatus()

Returns the status of the anti-removal switch (ARS) on the PINpad.

Prototype

```
int ux100_infoGetRemSwStatus(void);
```

Return Values

0 =	n = The status of the ARS: • 0 = The state of the ARS is armed. • 1 = The state of the ARS is triggered.
≠ 0	errno indicates the error type.

ux100_infoGetTamperStatus()

Returns the tamper status of the PINpad.

Prototype

```
int ux100_infoGetTamperStatus(void);
```

Return Values

0 =	n = The tamper status: • 0 = The status of the PINpad is OK. • 1 = The PINpad has been tampered with.
≠ 0	errno indicates the error type.

ux100_infoGetCoinChargingStatus()

Returns the coin battery charge status.

Prototype

```
int ux100_infoGetCoinChargingStatus(void);
```

Return Values

0 =	n = The status of the coin battery: • 1 = The charge status of the coin battery is good. • 0 = The charge status of the coin battery is bad.
≠ 0	errno indicates the error type.

ux100_infoGetSensorTemperature()

Obtains the temperature of the sensor.

Prototype

```
int ux100_infoGetSensorTemperature(void);
```

Return Values

0 =	n = The temperature of the sensor from -40°C to +125°C. The accuracy of the measure is 1 °C.
≠ 0	errno indicates the error type.

ux100_infoGetLogEvents()

Retrieves log events from the PINpad.

Prototype

```
struct ux100LogEventsStruct ux100_infoGetLogEvents(void);
```

Return Values

0 =	The ux100LogEventsStruct contains the log, as shown in the following example.
≠ 0	errno indicates the error type.

The application must free output structure data member when non-NULL. The function stores each event record in following format:

```
[DD-MM-YYYY HH:MM] EVT:<event name> STATUS:xxh DATA:<event data>LF
```

where,

- `<event name>` – The human readable event identifier.
- `xxh` – The event status code in hex format (zero, if not applicable).
- `<event data>` – Additional event information (can be empty).
- LF – A line feed (0x0A).

Example

```
struct ux100LogEventEventsStruct
{
void* data; // Log data in txt format.
int data_count; // Size of the data (in bytes).
};
```

ux100_infoSetIdCard()

Sets the identity card in the PINpad.

Prototype

```
int ux100_infoSetIdCard(struct uxIdCard idcard);
```

Parameters

Structure with the following format:

```
ux100IdCardStruct
{
unsigned long int device_mode; device mode
char part_number[32 + 1]; part number, max 33 bytes null-terminated string
char serial_number [11 + 1]; serial number, max 12 bytes null-terminated
string
char hardware_version[15 + 1]; hardware version, max 16 bytes null-
terminated string
char pci_hardware_version[4 + 1]; PCI hardware version, max 5 bytes null-
terminated string
char model_number[12 + 1]; model number, max 13 bytes null-terminated
string
char country_name [12 + 1]; country name, max 13 bytes null-terminated
string
char permanent_terminal_id[10 + 1]; permanent terminal ID, max 11 bytes
null-terminated string
};
```

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_infoGetConnectionStatus()

Checks if the PINpad is connected and ready to run commands.

Prototype

```
int ux100_infoGetConnectionStatus(unsigned int timeout);
```

Parameters

`timeout`	Time to wait for the PINpad to be ready. 0 = wait forever.

Return Values

0 =	Success. The PINpad is connected.
-1 =	errno indicates the error type.

ux100_infoGetDeviceMode()

Returns the device mode.

Prototype

```
int ux100_infoGetDeviceMode(void);
```

Return Values

0 =	Success. n = the device mode: • 0 = This PINpad is in manufacture mode. • 1 = The PINpad is in production mode. • 8 = The PINpad is in development mode.
-1 =	errno indicates the error type.

ux100_infoGetOperationalMode()

Returns the PINpad operational mode.

Prototype

```
int ux100_infoGetOperationalMode(void);
```

Return Values

0 =	Success. n = PINpad operational mode: • 0 = Field mode • 1 = Production mode #1 • 2 = Production mode #2 • 3 = Restricted mode • 4 = Controller Personalization mode
-1 =	errno indicates the error type.

ux100_infoGetPermanentTerminalID()

Returns the PINpad permanent ID.

Prototype

```
struct ux100PermanentTerminalIDStruct
ux100_infoGetPermanentTerminalID(void);
```

Return Values

0 =	The ux100PermanentTerminalIDStruct is filled, as shown in the following example.
-1 =	errno indicates the error type.

XML Structure

```
struct ux100PermanentTerminalIDStruct
{
char permanent_terminal_id[10 + 1]; /**< ux100 Permanent Terminal ID */
};
```

ux100_infoGetUnitType()

Returns the PINpad model type for example, UX100 or UX110.

Prototype

```
int ux100_infoGetUnitType(void);
```

Return Values

0 =	Unknown PINpad model type.
100 =	The PINpad is the UX100.
110 =	The PINpad is the UX110.
-1 =	errno indicates the error type.

ux100_infoResetLogHeader()

Resets the PINpad events log header.

NOTE

All stored events are lost.

Prototype

```
int ux100_infoResetLogHeader(void);
```

Return Values

0 =	Success. The log header was deleted.
-1 =	errno indicates the error type.

ux100_infoSetLogEvent()

Stores controller-related information in the PINpad event log.

Prototype

```
int ux100_infoSetLogEvent(char* event, int size);
```

Parameters

`event`	Data to store in the PINpad event log.
`size`	Size of data to store.

Return Values

0 =	Success. The new event is stored in the PINpad event log.
-1 =	errno indicates the error type.

ux100_infoGetAtmelSerialNumber()

Returns the Atmel CPU serial number.

Prototype

```
struct ux100AtmelSerialNumber ux100_infoGetAtmelSerialNumber(void);
```

Return Values

0 =	the ux100AtmelSerialNumber structure contains the Atmel S/N, as shown in the following example
-1 =	errno indicates the error type.

XML Structure

```
struct ux100AtmelSerialNumber
{
char atmel_sn [UX100_ATMEL_SERIAL_NUMBER_MAX_LEN + 1]; // ux100 Atmel SN
};
```

ux100_infoSetRTC()

Sets the PINpad RTC value from a given timestamp. The timestamp must be converted to an ASCII string in YYYY-MM-DD HH:MM:SS format.

Prototype

```
int ux100_infoSetRTC(char* s);
```

Parameters

s	Timestamp string to send to the PINpad RTC.

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_infoGetRTC()

Returns the PINpad RTC value as ASCII string in YYYY-MM-DD HH:MM:SS format.

Prototype

```
struct ux100DateTime ux100_infoGetRTC(void);
```

Return Values

0 =	Success. The ux100DateTime structure contains the actual RTC value, as shown in the following example.
-1 =	errno indicates the error type.

XML Structure

```
struct ux100DateTime
{
char rtc [UX100_DATE_TIME_MAX_LEN + 1]; // ux100 RTC value
};
```

ux100_infoGetHeaterMode()

Retrieves the UX100 heater mode.

Prototype

```
struct ux100_infoGetHeaterMode(void);
```

ux100_infoGetHeaterStatus()

UX100 only: Retrieves the heater hardware status. Heater ON/OFF is controlled in hardware only. For UX110: Always returns 0 (not supported).

Prototype

```
struct ux100_infoGetHeaterStatus(void);
```

Return Values

1 =	(heater on) Indicates that the LCD is being heated by the heater hardware.
0 =	(heater off) Indicates that the LCD is not being heated.
-1 =	errno indicates the error type.

ux100_infoGetBootVersion()

Returns the ux100 BOOT version.

Prototype

```
struct ux100BootVersion ux100_infoGetBootVersion(void);
```

Return Values

0 =	ux10BootVersion structure is filled.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100BootVersion
```

ux100_infoGetDiagCounter()

Retrieves the UX100 diagnostic counter in `index`.

Prototype

```
struct ux100DiagCounter ux100_infoGetDiagCounter(int index);
```

Parameters

`index`	Index of the diagnostic counter to retrieve.

Return Values

0 =	The ux100DiagCounter structure is filled.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100DiagCounter
{
                char         text[17];  // Max of 16 chars + 1 null
                unsigned int flags;
                int          value;
                int          index;
};
```

ux100_infoSetDiagCounter()

Sets the UX100 diagnostic counter in `index`.

Prototype

```
struct ux100DiagCounter infoSetDiagCounter(int index, int value);
```

Parameters

`index`	Index of the diagnostic counter to retrieve.
`value`	New value of the diagnostic counter in `index`.

Return Values

0 =	The ux100DiagCounter structure is filled.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100DiagCounter
{
                char         text[17];  // Max of 16 chars + 1 null
                unsigned int flags;
                int          value;
                int          index;
};
```

Security Functions

This section presents security-related APIs.

ux100_secGenerateRandom()

Returns a random number generated by the PINpad.

Prototype

```
Struct ux100ByteBuffer ux100_secGenerateRandom(int size);
```

Parameters

`size`	Indicates the size in bytes of the random number.

Return Values

0 =	Success. The number is returned in the ux100ByteBuffer structure, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ByteBuffer
{
void* data;
int data_count;
};
```

ux100_secSetSecureDate()

Set a trusted date certificate validation.

Prototype

```
int ux100_secSetSecureDate(unsigned int date);
```

Parameters

`date`	BDC coded date (yyyymmdd).

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_secLoadCertificate()

Loads a certificate into the PINpad.

Prototype

```
int ux100_secLoadCertificate(int type,
struct ux100ByteBuffer certificateSt);
```

Parameters

`type`	Type of certificate.

Return Values

0 =	Success. The certificate type is in struct ux100ByteBuffer, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ByteBuffer
{
void* data; // pointer on the data of the certificate
int data_count; // size in byte of the data
};
```

ux100_secLoadPublicKey()

Loads a public key in the PINpad.

Prototype

```
int ux100_secLoadPublicKey(in type, struct ux100ByteBuffer publicKeySt);
```

Parameters

`type`	Type of public key into struct ux100ByteBuffer, as shown in the following example.

Return Values

0 =	Success. The certificate type is in struct ux100ByteBuffer, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ByteBuffer
{
void* data; // pointer on the data of the public key
int data_count; // size in byte of the data
};
```

ux100_secReadPublicKey()

Reads the public key from the PINpad.

Prototype

```
struct ux100ByteBuffer ux100_secReadPublicKey(int type);
```

Parameters

`type`	Type of public key.

Return Values

0 =	Success. The public key is in struct ux100ByteBuffer, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ByteBuffer
{
void* data; // pointer on the data of the public key
int data_count; // size in byte of the data
};
```

ux100_secReadCertificate()

Reads a certificate from the PINpad.

Prototype

```
struct ux100ByteBuffer ux100_secReadCertificate(int type);
```

Parameters

`type`	Type of certificate.

Return Values

0 =	Success. The certificate is in struct ux100ByteBuffer, as shown in the following example.
≠ 0	errno indicates the error type.

XML Structure

```
struct ux100ByteBuffer
{
void* data; // pointer on the data of the certificate
int data_count; // size in byte of the data
};
```

ux100_secGetKeyList()

Lists all symmetric and asymmetric keys in the PINpad.

Prototype

```
struct ux100KeyList ux100_secGetKeyList(void);
```

Return Values

0 =	Success. The keys are in struct ux100KeyList, as shown in the following example
-1 =	errno indicates the error type.

XML Structure

```
struct ux100KeyList
{
void* data; // pointer on the keys data. Structure to be defined.
int data_count; // size in byte of the data
};
```

ux100_secGetCurrSECIRQ()

Reads the current value in the security register.

Prototype

```
int ux100_secGetCurrSECIRQ(void);
```

Return Values

0 = Success. SECIRQ value meanings are coded (for example, 0xAABBxxDD):

- AA ORed values:
  - 0x01: MESH: Problem with the grid mesh
  - 0x02: SWITCH: Problem with the switch
- BB ORed values:
  - 0x08: VBACKUPSML: Low voltage in VBACKUP pin.
  - 0x10: VBACKUPSMH: High voltage in VBACKUP pin.
- DD value:
  - 0x04: CRC: Bad CRC in user key registers.

ux100_secGetSavedSECIRQ()

Reads the value of the security register on the last SECIRQ or when any attack lost keys.

Prototype

```
int ux100_secGetSavedSECIRQ(void);
```

Return Values

0 = Success. SECIRQ value meanings are coded (for example, 0xAABBxxDD):

- AA ORed values:
  - 0x01: MESH: Problem with the grid mesh
  - 0x02: SWITCH: Problem with the switch
- BB ORed values:
  - 0x08: VBACKUPSML: Low voltage in VBACKUP pin.
  - 0x10: VBACKUPSMH: High voltage in VBACKUP pin.
- DD value:
  - 0x04: CRC: Bad CRC in user key registers.

ux100_secGetCurrentUKSR()

Reads the current value of the User Key Status register.

Prototype

```
int ux100_secGetCurrentUKSR(void);
```

Parameters

`exponent`	The public key exponent.
`size`	The key size in bytes.
`forceRenewal`	Forces mutual authentication regardless of the key’s existence.

Return Values

0 = Success. UKSR value meanings are coded (for example, 0xAABBCCDD):

- AA ORed values:
  - 0x01: MESH – Problem with the grid mesh.
  - 0x02: SWITCH – Problem with the switch.
  - 0x04: TAMPER – Tamper attack on the chip (for example, a shield attack).
- BB ORed values:
  - 0x01: CK32LFM – 32KHz, low frequency detected.
  - 0x02: TSL – Low temperature detected.
  - 0x04: TSH – High temperature detected.
  - 0x08: VBACKUPSML – Low voltage in VBACKUP pin.
  - 0x10: VBACKUPSMH – High voltage in VBACKUP pin.
- CC value:
  - Test and JTAG pins.
- DD value:
  - 0x01 – NCLEAR (usually set to 1) No key erase since the last load a User Key register.
  - 0x02 – SOFT: User key register erased by software.
  - 0x04 – CRC: User key register erased by software.
  - 0x08 – VBACKUPRST: User key register erased to a reset on VBACKUP pin.

ux100_secGetSavedUKSR()

Reads the value of the security register the last time a UKSR arrived or any attack that lost the keys.

Prototype

```
int ux100_secGetSavedUKSR(void);
```

Return Values

0 = Success. UKSR value meanings are coded (for example, 0xAABBCCDD):

- AA ORed values:
  - 0x01: MESH – Problem with the grid mesh.
  - 0x02: SWITCH – Problem with the switch.
  - 0x04: TAMPER – Tamper attack on the chip (for example, a shield attack).
- BB ORed values:
  - 0x01: CK32LFM – 32KHz, low frequency detected.
  - 0x02: TSL – Low temperature detected.
  - 0x04: TSH – High temperature detected.
  - 0x08: VBACKUPSML – Low voltage in VBACKUP pin.
  - 0x10: VBACKUPSMH – High voltage in VBACKUP pin.
- CC value:
  - Test and JTAG pins.
- DD value:
  - 0x01 – NCLEAR (usually set to 1) No key erase since the last load a User Key register.
  - 0x02 – SOFT: User key register erased by software.
  - 0x04 – CRC: User key register erased by software.
  - 0x08 – VBACKUPRST: User key register erased to a reset on VBACKUP pin.

ux100_secTamperNumEntHandl()

Handles the numeric entry in tamper mode.

Prototype

```
int ux100_secTamperNumEntHandl(int function);
```

Parameters

`function`	Function type: • 0x00 – Deactivate numeric entry in tamper mode. • 0x01 – Activate numeric entry in tamper mode. • 0x02 – Get status of numeric entry in tamper mode.

Return Values

- 1 =	An error occurred. Check the errno value.
0 =	Numeric entry in tamper mode not activated.
1 =	Numeric entry in tamper mode activated.

Manufacturer Functions

This section presents the APIs that drive user tests using `svc_ux100`.

ux100_mfgRemovalSwitchReset()

Resets the anti-removal switch without issuing a cryptographic challenge.

Prototype

```
int ux100_mfgRemovalSwitchReset(void);
```

Return Values

0 =	Success.
-1 =	errno indicates the error type.

ux100_mfgClearTamper()

Detampers the PINpad and regenerates the TRK.

Prototype

```
int ux100_mfgClearTamper(void);
```

Return Values

0 =	Success.
-1 =	errno indicates the error type.

Performing a Card Reader Reset

The UX300 card reader and UX100 and UX110 PINpads are updated using a USB thumb drive with a FAT partition. If you need to update the card reader, ask your Verifone representative for the *Updating a UX300/UX100 Device over USB Manual*.

Multi Drop Bus

This section presents information on the UX Multi Drop Bus (MDB) service for vending machines, including XML commands and structures, #defines, and APIs.

NOTE

Reference the svc_mdb.h header file to supplement the information in this chapter.

MDB Functions

This section contains the MDB APIs, XML structs, and return format for the MDB service.

mdb_setupDevices()

Initializes MDB devices, the standby MCU and UART.

C Prototype

```
int mdb_setupDevices(void);
```

Parameters

`optFeatures`	Optional feature bits (can be ORed): • MDB_OPTFEAT_FTL – The supported file transport layer • MDB_OPTFEAT_32BIT – 32-bit monetary format • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY – Data entry. • MDB_OPTFEAT_NEGVEND – Negative vend.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <setupDevices/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <setupDevices>
            <return>int</return>
        </setupDevices>
    </mdb>
</svc_rtn>
```

mdb_closeDevices()

Releases all MDB device resources in V/OS (in standby MCU and UART).

C Prototype

```
int mdb_closeDevices(void);
```

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <closeDevices/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <closeDevices>
            <return>int</return>
        </closeDevices>
    </mdb>
</svc_rtn>
```

mdb_receive()

Receives the next available MDB command from the MDB buffer.

C Prototype

```
struct MdbCommand * mdb_receive(void);
```

Return Values

Returns the mdb_command struct when data is available. Returns NULL if the receive buffer is empty.

On error, errno is set to:

- EINVAL – Invalid parameters.
- EACCES– Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <receive/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <receive>
                <return type="list">
                    <item type="container">
                        <cmd type="list">
                            <item>unsigned char</item>
                            <!-- ... const list <item> count of [36] -->
                        </cmd>
                        <len>unsigned int</len>
                    </item>
                </return>
            </receive>
    </mdb>
</svc_rtn>
```

mdb_peek()

Peeks at the next available MDB command from the MDB buffer without advancing the mdb buffer pointer to the next available command.

C Prototype

```
struct MdbCommand * mdb_peek(void);
```

Return Values

Returns the mdb_command struct when data is available, Returns NULL if the receive buffer is empty.

On error, errno is set to:

- EINVAL – Invalid parameters.
- EACCES– Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <peek/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <peek>
         <return type="list">
            <item type="container">
               <cmd type="list">
                  <item>unsigned char</item>
                  <!-- ... const list <item> count of [36] -->
               </cmd>
               <len>unsigned int</len>
            </item>
         </return>
      </peek>
   </mdb>
</svc_rtn>
```

mdb_advance_buffer_tail()

Advances the mdb_buffer pointer to the next available command. Call this function to skip to the next available command, for example, after calling mdb_peek which does not advance the buffer pointer.

C Prototype

```
void mdb_advance_buffer_tail(void);
```

XML Command

```
<svc_cmd>
    <mdb>
        <advance_buffer_tail/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <advance_buffer_tail>
            <return/>
        </advance_buffer_tail>
    </mdb>
</svc_rtn>
```

mdb_getVersion()

Obtains the MDB service version.

C Prototype

```
struct version mdb_getVersion(void)’
```

Return Values

The version of the struct as defined in svcmgrSvcDef.h.

XML Command

```
<svc_cmd>
   <mdb>
      <getVersion/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <getVersion>
         <return type="container">
            <major>int</major>
            <minor>int</minor>
            <maint>int</maint>
            <build type="string">string [max len 16]</build>
         </return>
      </getVersion>
   </mdb>
</svc_rtn>
```

mdb_getConfig()

Obtains the current MDB configuration from the standby MCU. The configuration is permanently stored in the standby MCU and does not need to be set after a power fail.

C Prototype

```
struct MdbConfig * mdb_getConfig(void);
```

Return Values

0 =	Success
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <getConfig/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <getConfig>
            <return type="list">
                <item type="container">
                    <MdbOn>int</MdbOn>
                    <AutoACK>int</AutoACK>
                    <AutoSetup>int</AutoSetup>
                    <AutoReset>int</AutoReset>
                </item>
            </return>
        </getConfig>
    </mdb>
</svc_rtn>
```

mdb_setConfig()

Sets the new MDB configuration in the standby MCU. The configuration is permanently stored in the standby MCU and does not need to be set after a power fail.

C Prototype

```
int mdb_setConfig(struct MdbConfig config /*0*/);
```

Return Values

0 =	Polling is active.
1 =	Polling is inactive.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <setConfig>
            <config type="container">
                <MdbOn>int [default:0]</MdbOn>
                <AutoACK>int [default:0]</AutoACK>
                <AutoSetup>int [default:0]</AutoSetup>
                <AutoReset>int [default:0]</AutoReset>
            </config>
        </setConfig>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <setConfig>
            <return>int</return>
        </setConfig>
    </mdb>
</svc_rtn>
```

mdb_setBusTimings()

Sets the timing parameters (in milliseconds) for the MDB link.

C Prototype

```
int mdb_setBusTimings(int block_time /*0*/, int interbyte_time /*0*/);
```

Parameters

`block_time`	The maximum time allowed between a command and the first byte of the response message.
`interbyte_time`	The maximum time allowed betwenn two consecutive bytes within one command or reply message.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <setBusTimings>
         <block_time>int [default:0]</block_time>
         <interbyte_time>int [default:0]</interbyte_time>
      </setBusTimings>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <setBusTimings>
         <return>int</return>
      </setBusTimings>
   </mdb>
</svc_rtn>
```

mdb_setMdbLinkState()

Enables or disables the MDB link from the application’s point of view. All incoming MDB requests are discarded.

C Prototype

```
int mdb_setMdbLinkState(int disabled /*0*/);
```

Parameters

`disabled`	Sets the MDB link: • 1 – Disable the MDB link. • 0 – Enable the MDB link.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <setMdbLinkState>
         <disabled>int [default:0]</disabled>
      </setMdbLinkState>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <setMdbLinkState>
         <return>int</return>
      </setMdbLinkState>
   </mdb>
</svc_rtn>
```

mdb_getTimeSinceLastPoll()

Returns the time elapsed since the last poll from VMC. This can help detect if the VMC is operational. In battery operated systems, the VMC powers off after the specified idle time.

C Prototype

```
int mdb_getTimeSinceLastPoll(void);
```

Return Values

>=0	Success (centisec),
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES– Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <getTimeSinceLastPoll/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <getTimeSinceLastPoll>
            <return>int</return>
        </getTimeSinceLastPoll>
    </mdb>
</svc_rtn>
```

mdb_checkPolling()

Checks if the system is polled by the vending machine.

C Prototype

```
int mdb_checkPolling(int waittime /*0*/);
```

Return Values

0 =	Polling is active
1 =	Polling is inactive
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <checkPolling>
            <waittime>int [default:0]</waittime>
        </checkPolling>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <checkPolling>
            <return>int</return>
        </checkPolling>
    </mdb>
</svc_rtn>
```

mdb_setCardReaderAddress()

Sets the MDB cashless device address.

C Prototype

```
int mdb_setCardReaderAddress(int address /*MDB_ADR_CARD_READER*/)';
```

Parameters

`address`	This parameter is set to MDB_ADR_CARD_READER or MDB_ADR_CASHLESS_2.

Return Values

0 =	Polling is active.
1 =	Polling is inactive.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <setCardReaderAddress>
            <address>int [default:MDB_ADR_CARD_READER]</address>
        </setCardReaderAddress>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <setCardReaderAddress>
            <return>int</return>
        </setCardReaderAddress>
    </mdb>
</svc_rtn>
```

mdb_getCardReaderAddress()

Obtains the MDB cashless device address.

C Prototype

```
int mdb_getCardReaderAddress(void);
```

Parameters

`address`	Either MDB_ADR_CARD_READER or MDB_ADR_CASHLESS_-2.

Return Values

0 =	Either MDB_ADR_CARD_READER or MDB_ADR_CASHLESS_2.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES– Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <getCardReaderAddress/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <getCardReaderAddress>
         <return>int</return>
      </getCardReaderAddress>
   </mdb>
</svc_rtn>
```

mdb_sendJustReset()

Sends the JUST RESET MDB command to indicate that the device was reset and returned to the inactive state. If V/OS is in sleep mode and the AutoReset feature is enabled in MCU firmware (see mdb_setConfig()), the MCU automatically responds with a JUST RESET command for the first poll command immediately after a WAKEUP command is received from the VMC.

C Prototype

```
int mdb_sendJustReset(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendJustReset/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendJustReset>
            <return>int</return>
        </sendJustReset>
    </mdb>
</svc_rtn>
```

mdb_send()

Sends the MDB character string to the MCU.

C Prototype

```
int mdb_send(unsigned char * data /*NULL*/, unsigned int length /*0*/,
unsigned int timeout /*0*/);
```

Parameters

`data`	The character string send.
`length`	Length of the character string.
`timeout`	The time to wait in milliseconds for a response.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EHOSTDOWN – Part of the message was lost during the transaction or unable to transfer the data to the MCU.
- ETIMEDOUT – The request timed out.
- EBADF – The UART file descriptor is not open.

XML Command

```
<svc_cmd>
        <mdb>
        <send>
            <data type="list">
                <item>unsigned char</item>
            </data>
            <length>unsigned int [default:0]</length>
            <timeout>unsigned int [default:0]</timeout>
        </send>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <send>
            <return>int</return>
        </send>
    </mdb>
</svc_rtn>
```

mdb_flush()

Flushes the internal MDB service buffer.

C Prototype

```
int mdb_flush(void);
```

Return Values

0 =	Success; no other value returns. errno is always set to 0.

XML Command

```
<svc_cmd>
    <mdb>
        <flush/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <flush>
            <return>int</return>
        </flush>
    </mdb>
</svc_rtn>
```

mdb_cmdType()

Returns the MDB command type.

C Prototype

```
char mdb_cmdType(unsigned char * cmd /*NULL*/, unsigned int cmdLen /*0*/);
```

Parameters

`cmd`	Pointer to command.
`cmdLen`	Length of the command.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <cmdType>
         <cmd type="list">
            <item>unsigned char</item>
         </cmd>
         <cmdLen>unsigned int [default:0]</cmdLen>
      </cmdType>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <cmdType>
         <return>char</return>
      </cmdType>
   </mdb>
</svc_rtn>
```

mdb_displayText()

Sends the text to display on the vending machine. Language-specific characters may differ from ASCII definitions, and depend on the vending machine controller.

C Prototype

```
int mdb_displayText(char * text /*NULL*/, unsigned char time /*0*/);
```

Parameters

`text`	Display text as a 0-terminated ASCII-string. Maximum length supported is 32 bytes.
`time`	Maximum time in seconds that the text displays.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <displayText>
         <text type="string">string [default:NULL]</text>
         <time>unsigned char [default:0]</time>
      </displayText>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <displayText>
         <return>int</return>
      </displayText>
   </mdb>
</svc_rtn>
```

mdb_clearDisplay()

Clears the display of a vending machine.

C Prototype

```
int mdb_clearDisplay(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <clearDisplay/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <clearDisplay>
            <return>int</return>
        </clearDisplay>
    </mdb>
</svc_rtn>
```

mdb_sendReaderConfigData()

Sends the Reader Config Data MDB command in response to the SETUP Configuration Data MDB command received from the VMC.

C Prototype

```
int mdb_sendReaderConfigData(unsigned char readerLevel /*0*/, unsigned
char * countryCode /*NULL*/, unsigned char scaleFactor /*0*/, unsigned
char decimalPlaces /*0*/, unsigned char maxResponseTime /*0*/, unsigned
char miscOptions /*0*/, int battery /*0*/);
```

Parameters

`readerLevel`	The MDB reader level.
`countryCode`	The country/currency code (2 bytes packed BCD).
`scaleFactor`	The scale factor.
`decimalPlaces`	The number of decimal places.
`maxResponseTime`	The application maximum response time (in seconds).
`miscOptions`	These are miscellaneous options (can be ORed): • MDB_CONF_OPTIONS_REFUND – The reader can restore funds to the payment media. • MDB_CONF_OPTIONS_MULTIVEND – The reader is multivend capable. • MDB_CONF_OPTIONS_DISPLAY – The reader has its own display. • MDB_CONF_OPTIONS_VENDCASHSALE – The reader supports the VEND/CASH SALE subcommand.
`battery`	**Note:** A battery flag indicates that the system can work in battery mode and will switch off after each session. Battery mode is enabled when the VMC indicates an 8x feature level. This causes the `readerLevel` parameter to change to the battery operated level.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendReaderConfigData>
         <readerLevel>unsigned char [default:0]</readerLevel>
         <countryCode type="list">
```

```
            <item>unsigned char</item>
        </countryCode>
        <scaleFactor>unsigned char [default:0]</scaleFactor>
        <decimalPlaces>unsigned char [default:0]</decimalPlaces>
        <maxResponseTime>unsigned char [default:0]</maxResponseTime>
        <miscOptions>unsigned char [default:0]</miscOptions>
        <battery>int [default:0]</battery>
    </sendReaderConfigData>
  </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
  <mdb>
    <sendReaderConfigData>
        <return>int</return>
    </sendReaderConfigData>
  </mdb>
</svc_rtn>
```

mdb_sendPeripheralId()

Sends the Peripheral ID MDB command with peripheral ID information in response to the EXPANSION Request ID MDB command from the VMC.

C Prototype

```
int mdb_sendPeripheralId(unsigned char readerLevel /*0*/,
unsigned char * manufCode /*NULL*/, unsigned char modelNumber /*NULL*/,
unsigned char softwareVersion /*NULL*/,
unsigned char serialNumber /*NULL*/, unsigned int optFeatures /*0*/);
```

Parameters

`readerLevel`	The MDB reader level.
`manufCode`	The manufacturer code (3 bytes ASCII).
`modelNumber`	The model number (12 bytes ASCII).
`softwareVersion`	The software version (2 bytes; BCD coded).
`serialNumber`	The factory assigned serial number.
`optFeatures`	These are optional feature bits (used at reader Level 3; can be ORed): • MDB_OPTFEAT_FTL – The file transport layer supported. • MDB_OPTFEAT_32BIT – The 32-bit monetary format. • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY– The data entry. • MDB_ OPTFEAT_NEGVEND – The negative vend.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendPeripheralId>
            <readerLevel>unsigned char [default:0]</readerLevel>
            <manufCode type="list">
                <item>unsigned char</item>
            </manufCode>
            <modelNumber type="list">
                <item>unsigned char</item>
            </modelNumber>
            <softwareVersion type="list">
```

```
                <item>unsigned char</item>
            </softwareVersion>
            <serialNumber type="list">
                <item>unsigned char</item>
            </serialNumber>
            <optFeatures>unsigned int [default:0]</optFeatures>
        </sendPeripheralId>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendPeripheralId>
            <return>int</return>
        </sendPeripheralId>
    </mdb>
</svc_rtn>
```

mdb_setOptionalFeatures()

Activates the optional features in the MDB driver.

C Prototype

```
int mdb_setOptionalFeatures(unsigned int optFeatures /*0*/);
```

Parameters

`optFeatures`	Optional feature bits (can be ORed): • MDB_OPTFEAT_FTL – The supported file transport layer • MDB_OPTFEAT_32BIT – 32-bit monetary format • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY – Data entry. • MDB_OPTFEAT_NEGVEND – Negative vend.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <setOptionalFeatures>
            <optFeatures>unsigned int [default:0]</optFeatures>
        </setOptionalFeatures>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <setOptionalFeatures>
            <return>int</return>
        </setOptionalFeatures>
    </mdb>
</svc_rtn>
```

mdb_sendDateTime()

Sends the current system time through the Diagnostics Response Date/Time MDB command to the VMC (only to the Level 3 reader).

C Prototype

```
int mdb_sendDateTime(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendDateTime/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendDateTime>
            <return>int</return>
        </sendDateTime>
    </mdb>
</svc_rtn>
```

mdb_sendDateTimeRequest()

Sends the Time/date MDB command request to the VMC. This retrieves the current date/time from the VMC.

C Prototype

```
int mdb_sendDateTimeRequest(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendDateTimeRequest/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendDateTimeRequest>
            <return>int</return>
        </sendDateTimeRequest>
    </mdb>
</svc_rtn>
```

mdb_sendDravs()

Sends age verification status to the VMC.

C Prototype

```
int mdb_sendDravs(unsigned char ucFeatures /*0*/);
```

Return Values

Data can be any length less than or equal to MDB_WRITE_CMD_SIZE-1.

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendDravs>
         <ucFeatures>unsigned char [default:0]</ucFeatures>
      </sendDravs>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendDravs>
         <return>int</return>
      </sendDravs>
   </mdb>
</svc_rtn>
```

mdb_sendMalfunctionError()

Sends a malfunction error.

C Prototype

```
int mdb_sendMalfunctionError(unsigned char error /*0*/);
```

Parameters

`error`	The error code.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendMalfunctionError>
            <error>unsigned char [default:0]</error>
        </sendMalfunctionError>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendMalfunctionError>
            <return>int</return>
        </sendMalfunctionError>
    </mdb>
</svc_rtn>
```

mdb_sendDiagnosticResponse()

Sends device-specific diagnostic data.

C Prototype

```
int mdb_sendDiagnosticResponse(unsigned char * customData /*NULL*/,
int customData_len /*0*/);
```

Return Values

Data can be any length less than or equal to MDB_WRITE_CMD_SIZE-1.

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendDiagnosticResponse>
         <customData type="list">
            <item>unsigned char</item>
         </customData>
         <customData_len>int [default:0]</customData_len>
      </sendDiagnosticResponse>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendDiagnosticResponse>
         <return>int</return>
      </sendDiagnosticResponse>
   </mdb>
</svc_rtn>
```

mdb_sendSendDataEntryRequestResp()

Sends a data entry request response command.

C Prototype

```
int mdb_sendSendDataEntryRequestResp(unsigned char ucDataEntryLength
/*0*/, unsigned char bRepeatedReq, unsigned string pcText /*NULL*/,
unsigned char ucTime /*0*/);
```

Parameters

`ucDataEntryLength`	The length of the expected data entry.
`bRepeatedReq`	A flag indicating a repeated request.
`pcText`	This is optional display text; NULL can be passed.
`ucTime`	The diplay time. This field is only used if `pcText` is not NULL

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendSendDataEntryRequestResp>
         <ucDataEntryLength>
            unsigned char [default:0]
         </ucDataEntryLength>
         <bRepeatedReq>unsigned char [default:0]</bRepeatedReq>
         <pcText type="string">string [default:NULL]</pcText>
         <ucTime>unsigned char [default:0]</ucTime>
      </sendSendDataEntryRequestResp>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendSendDataEntryRequestResp>
         <return>int</return>
      </sendSendDataEntryRequestResp>
   </mdb>
</svc_rtn>
```

mdb_sendDataEntryCancel()

Sends a data entry cancel command.

C Prototype

```
int mdb_sendDataEntryCancel(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendDataEntryCancel/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendDataEntryCancel>
         <return>int</return>
      </sendDataEntryCancel>
   </mdb>
</svc_rtn>
```

mdb_sendFTLRetryDeny()

Sends a file transfer layer retry deny command.

C Prototype

```
int mdb_sendFTLRetryDeny(unsigned char ucDestination /*0*/,
unsigned char ucWaittime /*0*/, unsigned char bWithAck /*0*/);
```

Parameters

`ucDestination`	The address of the VMC.
`ucWaittime`	A time delay for the sender to wait before trying to re-send the data file.
`bWithAck`	Set to ‘1’.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendFTLRetryDeny>
         <ucDestination>unsigned char [default:0]</ucDestination>
         <ucWaittime>unsigned char [default:0]</ucWaittime>
         <bWithAck>unsigned char [default:0]</bWithAck>
      </sendFTLRetryDeny>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendFTLRetryDeny>
         <return>int</return>
      </sendFTLRetryDeny>
   </mdb>
</svc_rtn>
```

mdb_sendFTLSendBlock()

Sends a block of data from a file to the VMC.

C Prototype

```
int mdb_sendFTLSendBlock(unsigned char ucDestination /*0*/, unsigned char
ucBlockNum /*0*/, unsigned char ucDataLen /*0*/,
unsigned char pucData /*NULL*/, unsigned char bWithAck /*0*/);
```

Parameters

`ucDestination`	The address of the VMC.
`ucBlockNum`	The block number.
`ucDataLen`	The length of the data.
`pucData`	Pointer to the data.
`bWithAck`	Set to ‘1’.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendFTLSendBlock>
            <ucDestination>unsigned char [default:0]</ucDestination>
            <ucBlockNum>unsigned char [default:0]</ucBlockNum>
            <ucDataLen>unsigned char [default:0]</ucDataLen>
            <pucData type="list">
                <item>unsigned char</item>
            </pucData>
            <bWithAck>unsigned char [default:0]</bWithAck>
        </sendFTLSendBlock>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendFTLSendBlock>
            <return>int</return>
        </sendFTLSendBlock>
    </mdb>
</svc_rtn>
```

mdb_sendFTLOkToSend()

Confirms with the VMC that it is OK to send a file.

C Prototype

```
int mdb_sendFTLOkToSend(unsigned char ucDestination /*0*/);
```

Parameters

`ucDestination`	Address of the VMC.

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendFTLOkToSend>
         <ucDestination>unsigned char [default:0]</ucDestination>
      </sendFTLOkToSend>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendFTLOkToSend>
         <return>int</return>
      </sendFTLOkToSend>
   </mdb>
</svc_rtn>
```

mdb_sendFTLRequestToSend()

Sends a request to send a file to the VMC.

C Prototype

```
int mdb_sendFTLRequestToSend(unsigned char ucDestination /*0*/, unsigned
char ucFileID /*0*/, unsigned char ucMaxLength /*0*/,
unsigned char bLastSession /*0*/);
```

Parameters

`ucDestination`	The address of the VMC.
`ucFileID`	The ID of the file.
`ucMaxLength`	The maximum file length.
`bLastSession`	The control byte.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendFTLRequestToSend>
            <ucDestination>unsigned char [default:0]</ucDestination>
            <ucFileID>unsigned char [default:0]</ucFileID>
            <ucMaxLength>unsigned char [default:0]</ucMaxLength>
            <bLastSession>unsigned char [default:0]</bLastSession>
        </sendFTLRequestToSend>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendFTLRequestToSend>
            <return>int</return>
        </sendFTLRequestToSend>
    </mdb>
</svc_rtn>
```

mdb_FTLReceiveFile()

Receives a file from the VMC.

C Prototype

```
int mdb_FTLReceiveFile(unsigned char ucSourceAddress /*0*/,
unsigned char ucFileID /*0*/, char * pcFileName /*NULL*/,
unsigned char ucMaxBlocks /*0*/, unsigned char ucControlByte /*0*/,
unsigned int uiTimeout /*0*/);
```

Parameters

`ucSourceAddress`	Address of the source.
`ucFileID`	ID of the file.
`pcFileName`	Name of the file.
`ucMaxBlocks`	The maximum number of blocks, as received from the request command.
`ucControlByte`	The request command control byte.
`uiTimeout`	The maximum timeout between data blocks (in 100ths of a second; maximum valid value = 10000).

Return Values

0 = Success

-1 = Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <FTLReceiveFile>
         <ucSourceAddress>unsigned char [default:0]</ucSourceAddress>
         <ucFileID>unsigned char [default:0]</ucFileID>
         <pcFileName type="string">string [default:NULL]</pcFileName>
         <ucMaxBlocks>unsigned char [default:0]</ucMaxBlocks>
         <ucControlByte>unsigned char [default:0]</ucControlByte>
         <uiTimeout>unsigned int [default:0]</uiTimeout>
      </FTLReceiveFile>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <FTLReceiveFile>
         <return>int</return>
      </FTLReceiveFile>
   </mdb>
</svc_rtn>
```

mdb_FTLReceiveFileExt()

Receives a file from the VMC.

C Prototype

```
int mdb_FTLReceiveFileExt(unsigned char ucSourceAddress /*0*/,
unsigned char ucFileID /*0*/, char * pcFileName /*NULL*/,
unsigned char ucMaxBlocks /*0*/, unsigned char ucControlByte /*0*/,
unsigned int uiTimeout /*0*/, unsigned char ucRecData /*NULL*/,
unsigned int uiRecDataLen /*NULL*/);
```

Parameters

`ucSourceAddress`	Address of the source.
`ucFileID`	ID of the file.
`pcFileName`	Name of the file. If NULL, data is stored in `ucRecData`.
`ucMaxBlocks`	The maximum number of blocks, as received from the request command.
`ucControlByte`	The request command control byte.
`uiTimeout`	The maximum timeout between data blocks (in 100ths of a second; maximum valid value = 10000).
`ucRecData`	Pointer to file destination.
`uiRecDataLen`	The size of `ucRecData`.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <FTLReceiveFileExt>
         <ucSourceAddress>unsigned char [default:0]</ucSourceAddress>
         <ucFileID>unsigned char [default:0]</ucFileID>
         <pcFileName type="string">string [default:NULL]</pcFileName>
         <ucMaxBlocks>unsigned char [default:0]</ucMaxBlocks>
         <ucControlByte>unsigned char [default:0]</ucControlByte>
         <uiTimeout>unsigned int [default:0]</uiTimeout>
         <ucRecData type="list">
            <item>unsigned char</item>
         </ucRecData>
         <uiRecDataLen type="list">
            <item>unsigned int</item>
         </uiRecDataLen>
      </FTLReceiveFileExt>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <FTLReceiveFileExt>
         <return>int</return>
      </FTLReceiveFileExt>
   </mdb>
</svc_rtn>
```

mdb_FTLRespondToReceiveRequest()

Responds to a request to receive a file from the VMC.

C Prototype

```
int mdb_FTLRespondToReceiveRequest(unsigned char ucDestination /*0*/,
unsigned char ucFileID /*0*/, char * pcFileName /*NULL*/,
unsigned int uiAddDataHeaderLen /*0*/,
unsigned char pucAddDataHeader /*0*/,
unsigned int uiAddDataTrailerLen /*0*/,
unsigned char pucAddDataTrailer /*0*/, unsigned char ucMaxBlocks /*0*/,
unsigned char ucControlByte /*0*/);
```

Parameters

`ucDestination`	VMC address.
`ucFileID`	ID of the file.
`pcFileName`	Name of the file. If NULL, data is stored in `ucRecData`.
`uiAddDataHeaderLen`	Additional data header length (can be 0).
`pucAddDataHeader`	Additional data header (can be NULL).
`uiAddDataTrailerLen`	Additional data trailer length (can be 0).
`pucAddDataTrailer`	Additional data trailer (can be NULL).
`ucMaxBlocks`	The maximum number of blocks.
`ucControlByte`	The request command control byte.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <FTLRespondToReceiveRequest>
         <ucDestination>unsigned char [default:0]</ucDestination>
         <ucFileID>unsigned char [default:0]</ucFileID>
         <pcFileName type="string">string [default:NULL]</pcFileName>
            <uiAddDataHeaderLen>
            unsigned int [default:0]
            </uiAddDataHeaderLen>
         <pucAddDataHeader type="list">
         <item>unsigned char</item>
      </pucAddDataHeader>
         <uiAddDataTrailerLen>
            unsigned int [default:0]
            </uiAddDataTrailerLen>
         <pucAddDataTrailer type="list">
            <item>unsigned char</item>
         </pucAddDataTrailer>
         <ucMaxBlocks>unsigned char [default:0]</ucMaxBlocks>
         <ucControlByte>unsigned char [default:0]</ucControlByte>
      </FTLRespondToReceiveRequest>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <FTLRespondToReceiveRequest>
         <return>int</return>
      </FTLRespondToReceiveRequest>
   </mdb>
</svc_rtn>
```

mdb_FTLSendFile()

Receives a file from the VMC.

C Prototype

```
int mdb_FTLSendFile(unsigned char ucDestination /*0*/, unsigned char
ucFileID /*0*/, char * pcFileName /*NULL*/,
unsigned int uiAddDataHeaderLen /*0*/,
unsigned char pucAddDataHeader /*0*/,
unsigned int uiAddDataTrailerLen /*0*/,
unsigned char pucAddDataTrailer /*0*/, unsigned char ucMaxBlocks /*0*/,
unsigned char ucControlByte /*0*/);
```

Parameters

`ucDestination`	Address of the file destination.
`ucFileID`	ID of the file.
`pcFileName`	Name of the file.
`uiAddDataHeaderLen`	Additional data header length (can be 0).
`pucAddDataHeader`	Additional data header (can be NULL),
`uiAddDataTrailerLen`	Additional data trailer length (can be 0).
`pucAddDataTrailer`	Additional data trailer (can be NULL).

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- ENODEV – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <FTLSendFile>
         <ucDestination>unsigned char [default:0]</ucDestination>
         <ucFileID>unsigned char [default:0]</ucFileID>
         <pcFileName type="string">string [default:NULL]</pcFileName>
         <uiAddDataHeaderLen>
         unsigned int [default:0]
         </uiAddDataHeaderLen>
         <pucAddDataHeader type="list">
         <item>unsigned char</item>
         </pucAddDataHeader>
            <uiAddDataTrailerLen>
            unsigned int [default:0]
            </uiAddDataTrailerLen>
         <pucAddDataTrailer type="list">
            <item>unsigned char</item>
         </pucAddDataTrailer>
      </FTLSendFile>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <FTLSendFile>
         <return>int</return>
      </FTLSendFile>
   </mdb>
</svc_rtn>
```

mdb_sendBeginSession()

Sends the Begin Session MDB command to inform the VMC about the card amount, and prompts the user to make selection. This command only sends the card amount. There is no guarantee that this amount is available to debit from the card. The EXPANSION Request ID MDB command from the VMC. Configuration data is obtained from the VMC.

C Prototype

```
int mdb_sendBeginSession(unsigned char readerLevel /*0*/,
unsigned int amount /*0*/, unsigned char * paymentMediaId /*NULL*/,
unsigned char paymentType /*0*/, unsigned char * paymentData /*NULL*/,
unsigned char * userLanguage /*NULL*/,
unsigned char * currencyCode /*NULL*/, unsigned char cardOptions /*0*/);
```

Parameters

`readerLevel`	The MDB reader level.
`amount`	Available funds (credit; scaled) on the card.
`paymentMediaId`	Payment media ID (4 bytes).
`paymentType`	Type of payment.
`paymentData`	Payment data (2 bytes).
`userLanguage`	User language (2 bytes ASCII).
`currencyCode`	User currency code as defined in ISO 4217 (2 bytes packed BCD),
`cardOptions`	The card options (can be ORed): • MDB_CARDOPT_DISPLAY– The VMC displays the credit • MDB_CARDOPT_NO_DISPLAY – The VMC must not display the credit (user option) • MDB_ CARDOPT_NO_REFUND – The inserted card has no refund capability • MDB_CARDOPT_REFUND – The inserted card has refund capability. • MDB_CARDOPT_NO_REVALUE – The inserted card has no revalue capability. • MDB_CARDOPT_REVALUE – The inserted card has revalue capability.

Return Values

0 =	Success
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendBeginSession>
         <readerLevel>unsigned char [default:0]</readerLevel>
         <amount>unsigned int [default:0]</amount>
         <paymentMediaId type="list">
            <item>unsigned char</item>
         </paymentMediaId>
         <paymentType>unsigned char [default:0]</paymentType>
         <paymentData type="list">
            <item>unsigned char</item>
         </paymentData>
         <userLanguage type="list">
            <item>unsigned char</item>
         </userLanguage>
         <currencyCode type="list">
            <item>unsigned char</item>
         </currencyCode>
         <cardOptions>unsigned char [default:0]</cardOptions>
      </sendBeginSession>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendBeginSession>
         <return>int</return>
      </sendBeginSession>
   </mdb>
</svc_rtn>
```

mdb_sendSessionCancelRequest()

Sends the Session Cancel Request MDB command to the VMC. This cancels the session before the user makes a selection.

C Prototype

```
int mdb_sendSessionCancelRequest(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendSessionCancelRequest/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendSessionCancelRequest>
            <return>int</return>
        </sendSessionCancelRequest>
    </mdb>
</svc_rtn>
```

mdb_sendVendApproved()

Sends the Vend Approved MDB command in response to the VEND Vend Request MDB command from the VMC. This informs the VMC that the vend amount was deducted from the user's payment media. The VMC can dispense the product. Do not send this command if it is unsure if sufficient funds are available.

C Prototype

```
int mdb_sendVendApproved(unsigned char readerLevel /*0*/,
unsigned int amount /*0*/);
```

Parameters

`readerLevel`	the MDB reader level.
`amount`	The amount deducted (scaled).

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendVendApproved>
            <readerLevel>unsigned char [default:0]</readerLevel>
            <amount>unsigned int [default:0]</amount>
        </sendVendApproved>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendVendApproved>
            <return>int</return>
        </sendVendApproved>
    </mdb>
</svc_rtn>
```

mdb_sendVendDenied()

Sends the Vend Denied MDB command in response to the VEND Vend Request MDB command from the VMC. This denies approval of the patrons selection and prohibits the VMC to dispense any product.

C Prototype

```
int mdb_sendVendDenied(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendVendDenied/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendVendDenied>
         <return>int</return>
      </sendVendDenied>
   </mdb>
</svc_rtn>
```

mdb_sendEndSession()

Sends the End Session MDB command to indicate that the payment media was removed, the device has finished the vending process and has returned to the Enabled state. The End Session command is issued in response to the VEND Session Complete MDB command from the VMC.

C Prototype

```
int mdb_sendEndSession(void);
```

Return Values

0 =	Success
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendEndSession/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendEndSession>
            <return>int</return>
        </sendEndSession>
    </mdb>
</svc_rtn>
```

mdb_sendOutOfSequence()

Sends the Out of Sequence MDB command to reject forbidden commands. Use this command if the device cannot execute the received command.

C Prototype

```
int mdb_sendOutOfSequence(unsigned char readerLevel /*0*/,
unsigned char state /*0*/);
```

Parameters

`readerLevel`	The MDB reader level.
`state`	The state of the MDB state machine (used only for reader Level 2 or 3; cannot be ORed).

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendOutOfSequence>
            <readerLevel>unsigned char [default:0]</readerLevel>
            <state>unsigned char [default:0]</state>
        </sendOutOfSequence>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendOutOfSequence>
            <return>int</return>
        </sendOutOfSequence>
    </mdb>
</svc_rtn>
```

mdb_sendCancelled()

Sends the Cancelled MDB command in response to the READER Cancel MDB command from the VMC.

C Prototype

```
int mdb_sendCancelled(void);
```

Return Values

0 =	Success
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendCancelled/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendCancelled>
            <return>int</return>
        </sendCancelled>
    </mdb>
</svc_rtn>
```

mdb_sendRevalueApproved()

NOTE This call is only supported on Level 2 and 3 readers.

Sends the Revalue Approved MDB command in response to the REVALUE Revalue Request MDB command from the VMC to accept the revalue request. The reader must add the requested value to the payment media balance.

C Prototype

```
int mdb_sendRevalueApproved(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendRevalueApproved/>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendRevalueApproved>
         <return>int</return>
      </sendRevalueApproved>
   </mdb>
</svc_rtn>
```

mdb_sendRevalueDenied()

NOTE This call is only supported on Level 2 and 3 readers.

Sends the Revalue Denied MDB command in response to the REVALUE Revalue Request MDB command from the VMC to deny the revalue request.

C Prototype

```
int mdb_sendRevalueDenied(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
    <mdb>
        <sendRevalueDenied/>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <sendRevalueDenied>
            <return>int</return>
        </sendRevalueDenied>
    </mdb>
</svc_rtn>
```

mdb_sendRevalueLimitAmount()

Sends the Revalue Limit Amount MDB command in response to the REVALUE Revalue Limit Requested MDB command from the VMC. The revalue limit is the amount the reader will accept.

C Prototype

```
int mdb_sendRevalueLimitAmount(unsigned char readerLevel /*0*/,
unsigned int amount /*0*/);
```

Parameters

`readerLevel`	The MDB reader level.
`Revalue`	This is the card limit value (scaled).

Return Values

0 =	Success.
-1 =	Error with errno set to:

- EINVAL – Invalid parameters.
- EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <mdb>
      <sendRevalueLimitAmount>
         <readerLevel>unsigned char [default:0]</readerLevel>
         <amount>unsigned int [default:0]</amount>
      </sendRevalueLimitAmount>
   </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <mdb>
      <sendRevalueLimitAmount>
         <return>int</return>
      </sendRevalueLimitAmount>
   </mdb>
</svc_rtn>
```

mdb_setDisplayData()

Stores information about the vending machine display. This information is normally sent by the vending machine in the setup config data command.

C Prototype

```
int mdb_setDisplayData(unsigned char width /*0*/,
unsigned char height /*0*/, unsigned char dispInfo /*0*/);
```

Parameters

`width`	The number of characters per row (default 16).
`height`	The number of rows on the display (default 2).
`dispInfo`	The display type.

Return Values

0 =	Polling is active.
1 =	Polling is inactive.
-1 =	Error with errno set to: • EINVAL – Invalid parameters.

XML Command

```
<svc_cmd>
    <mdb>
        <setDisplayData>
            <width>unsigned char [default:0]</width>
            <height>unsigned char [default:0]</height>
            <dispInfo>unsigned char [default:0]</dispInfo>
        </setDisplayData>
    </mdb>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <mdb>
        <setDisplayData>
            <return>int</return>
        </setDisplayData>
    </mdb>
</svc_rtn>
```

MDB Data Structure Index

The following are the MDB data structures:

- MdbCommand–MDB command structure
- MdbConfig–MDB configuration structure

MdbCommand Structure

Peeks at the next available MDB command from MDB buffer without advancing the mdb_buffer pointer to the next available command.

C Prototype

```
struct MdbCommand mdb_peek (void);
```

Return Values

Returns the mdb_command struct when data is available. Returns NULL if the receive buffer is empty.

0 =	Success
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • ACCES – Cannot access the standby MCU interface.

Data Fields

`unsigned char cmd [MDB_CMD_SIZE]`

This is the MDB address, plus the data and checksum.

`unsigned int len`

This is the length of `cmd`.

Struct Reference

`unsigned char MdbCommand::cmd[MDB_CMD_SIZE]`

This is the MDB address, plus the data and checksum.

`unsigned int MdbCommand::len`

This is the length of `cmd`.

MdbConfig Structure

Obtains the current MDB configuration from the Standby MCU. The configuration is permanently stored in the Standby MCU, and does not need to be set after a power fail.

C Prototype

```
struct MdbConfig mdb_getConfig (void);
```

Return Values

0 =	Success
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • ACCES – Cannot access the standby MCU interface.

Data Fields

`int MdbOn`

This value is 1 if MDB mode is on, or 0 if MDB mode is off (must be 1 for this service to work).

`int AutoACK`

This value is 1 for MCU to Auto ACK all MDB commands from the VMC, or 0 for no auto-ACKing.

`int AutoSetup`

NOTE

Not currently implemented.

This value is 1 for MCU to send MDB device setup response automatically.

`int AutoReset`

This value is 1 for the MCU to automatically respond to the MDB Host RESET command when V/OS is in sleep mode, or 0 if not in sleep mode.

Struct Reference

`int MdbConfig::AutoACK`

This value is 1 for MCU to Auto ACK all MDB commands from the VMC, or 0 for no auto-ACKing.

`int MdbConfig::AutoSetup`

NOTE

Not currently implemented.

This value is 1 for MCU to send MDB device setup response automatically.

`int MdbConfig::AutoReset`

This value is 1 for the MCU to automatically respond to the MDB Host RESET command when V/OS is in sleep mode, or 0 if not in sleep mode.

#defines

```
#define MDB_CMD_SIZE 36
#define MDB_WRITE_CMD_SIZE (MDB_CMD_SIZE -1)
#define MDB_FTL_BYTES_PER_BLOCK 31U
#define MDB_FTL_LAST_SESSION_FLAG 0x02
```

```
#define MDB_FTL_MAX_BLOCKS_PER_SESSION 255UL
#define MDB_ADR_CARD_READER 0x10
#define MDB_ADR_CASHLESS_2 0x60
#define MDB_UNKNOWN -1
#define MDB_NO_COMMAND 0
#define MDB_RESET 1
#define MDB_SETUP_CONFDATA 2
#define MDB_IDENT 3
#define MDB_DISABLE 4
#define MDB_ENABLE 5
#define MDB_VEND_REQUEST 6
#define MDB_VEND_CANCEL 7
#define MDB_VEND_SUCCESS 8
#define MDB_VEND_FAILURE 9
#define MDB_SESSION_COMPLETE 10
#define MDB_VEND_REQUEST_NEG 11
#define MDB_VEND_CASH_SALE 12
#define MDB_SETUP_PRICE 13
#define MDB_REVALUE_LIMIT_REQ 14
#define MDB_REVALUE_REQ 15
#define MDB_EXP_DIAG 16
#define MDB_SET_DATE_TIME 17
#define MDB_ENABLE_OPT 18
#define MDB_CANCEL 19
#define MDB_FTLR_REQ_TO_RCV 20
#define MDB_FTLR_RETRY_DENY 21
#define MDB_FTLR_SEND_BLOCK 22
#define MDB_FTLR_OK_TO_SEND
#define MDB_FTLR_REQ_TO_SEND 24
#define MDB_DATA_ENTRY_RESPONSE 25
#define MDB_OPTFEAT_FTL 0x01
#define MDB_OPTFEAT_32BIT 0x02
#define MDB_OPTFEAT_MULTICUR 0x04
#define MDB_OPTFEAT_NEGVEND 0x08
#define MDB_OPTFEAT_DATAENTRY 0x10
#define MDB_OPTFEAT_SELECTIONFIRST 0x20
#define MDB_CONF_OPTIONS_REFUND 0x01
#define MDB_CONF_OPTIONS_MULTIVEND 0x02
#define MDB_CONF_OPTIONS_DISPLAY 0x04
#define MDB_CONF_OPTIONS_VENDCASHSALE 0x08
#define MDB_STATE_NONE 0
#define MDB_STATE_INACTIVE 1
#define MDB_STATE_DISABLE 2
#define MDB_STATE_ENABLE 3
#define MDB_STATE_IDLE 4
#define MDB_STATE_VEND 5
```

```
#define MDB_STATE_REVALUE 6
#define MDB_STATE_NEGATIVE_VEND 7
#define MDB_CARDOPT_DISPLAY 0x00
#define MDB_CARDOPT_NO_DISPLAY 0x01
#define MDB_CARDOPT_NO_REFUND 0x00
#define MDB_CARDOPT_REFUND 0x02
#define MDB_CARDOPT_NO_REVALUE 0x00
#define MDB_CARDOPT_REVALUE 0x04
```

Configuring PPP Over UX PSTN Modems

1. Use the `net_interfaceSet( struct netIfconfig )` command to write the PPP parameters to netconf.xml.
2. Use the `net_interfaceModemExtSet( NET_MODEM_UXPSTN, netModemInfo)` command to write UX PSTN modem in netconf.xml.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="modem_uxpstn"/>
</PPPPROTO_LIST>

<MODISDNLINK_LIST>
<MODISDNLINK name_id=" modem_uxpstn" … other parameters … />
</MODISDNLINK_LIST>

</SETTINGS>
```

Configuring PPP Over UX ISDN Modems

1. Use the `net_interfaceUp( "ppp0", NET_IP_V4 )` command to connect to the modem and then bring the PPP interface.
2. Use the `net_interfaceDown( "ppp0" )` command to disconnect the PPP interface and hang up.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="modem_uxisdn"/>
</PPPPROTO_LIST>

<MODISDNLINK_LIST>
<MODISDNLINK name_id=" modem_uxisdn" … other parameters … />
</MODISDNLINK_LIST>
```

```
</SETTINGS>
```

Configuring PPP Over the Serial Port

1 Use the `net_interfaceSet( struct netIfconfig )` command to write the PPP parameters to netconf.xml.

NOTE

Use `ppp0` as the PPP interface name.

2 Use the OPTION tag Convoy the PPP over serial port parameters.

Example

```
char p_option[NET_OPTION_MAX];

/*<** Define the parameters used for PPP connection */
#define PPP_OVER_SERIAL "asyncmap 0 noauth noipdefault crtscts
local_ip:remote_ip com_port speed

strncpy(p_option ", PPP_OVER_SERIAL, strlen(PPP_OVER_SERIAL) );
```

where,

- `local_ip` = terminal IP address
- `remote_ip` = server IP address
- `com_port` = serial port name (for exmple, /dev/ttyAMA1)

NOTE

Even if you configure the desired TTY serial interface for the printer in the API, always use /dev/ttyAMA2 for the Remote Printer.

- `speed` = communication speed (for exmple, 19200)

1 Use the `net_optionSet( "ppp0", p_option )` command to write the PPP over serial parameters to netconf.xml.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="serial_port"/>
</PPPPROTO_LIST>

<OPTION_LIST>
<OPTION name_id="ppp0" option=" asyncmap 0 noauth noipdefault crtscts
local_ip:remote_ip com_port speed" />
```

```
</OPTION_LIST>
</SETTINGS>
```

2 Use the `net_interfaceUp( "ppp0", NET_IP_V4 )` command to connect to the modem and then bring the PPP interface.

3 Use the `net_interfaceDown( "ppp0" )` command to disconnect the PPP interface and hang up.

Remote Printer

This section contains the UX remote printer service, including XML commands and structures, #defines, and APIs. The UX terminal supports two printer models:

- CUSTOMS TG 2460, TG 2460H
- ARTEMA unattended serial printer

Both printers exchange data over a serial connection. Printer type is automatically detected on connection.

NOTE Reference the svcmgrSvcDef.h header file to supplement the information in this section.

Supported Features

- Print interface is the same for supported printers
- Rendering–Done completely inside the printer; applications cannot print out a full framebuffer or large pre-rendered image files
- Plain text–Font size and print density formatting options
- Barcode–Predefined types; rendering done in printer H/W
- Icons and bitmap images–Limited file size
- Application-defined coordinates positioning

Remote Printer Functions

This section presents the remote printer APIs and XML structures. See #defines for valid parameter values.

NOTE Even if you configure the desired TTY serial interface for the printer in the API, always use /dev/ttyAMA2 for the Remote Printer.

remoteprinter_getVersion()

Obtains the remoter printer service version.

C Prototype

```
struct version remoteprinter_getVersion (void);
```

Parameters

`optFeatures`	Optional feature bits (can be ORed): • MDB_OPTFEAT_FTL – The supported file transport layer • MDB_OPTFEAT_32BIT – 32-bit monetary format • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY – Data entry. • MDB_OPTFEAT_NEGVEND – Negative vend.

Return Values

Returns the version struct as defined in svcmgrSvcDef.h.

0 =	Success.
-1 =	Error with errno set to: • EINVAL – Invalid parameters. • EACCES – Cannot access the standby MCU interface.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <getVersion/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <getVersion>
         <return type="container">
            <major>int</major>
            <minor>int</minor>
            <maint>int</maint>
            <build type="string">string [max len 16]</build>
         </return>
      </getVersion>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_Open()

Opens the remote printer service. Call remoteprinter_TurnOn() after using this command to enable printing.

C Prototype

```
int remoteprinter_Open (char comDev);
```

Parameters

`comDev`	The COM flag string.

Return Values

0 =	Success, and the COM device handle.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access the COM device.
- EINVAL – Invalid input parameter.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Open>
         <comDev type="string">string [default:NULL]</comDev>
      </Open>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <Open>
         <return>int</return>
      </Open>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_TurnOn()

Powers on the remote printer.

C Prototype

```
int remoteprinter_TurnOn(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <TurnOn/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <TurnOn>
         <return>int</return>
      </TurnOn>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_TurnOff()

Powers off the remote printer.

C Prototype

```
int remoteprinter_TurnOff (void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <TurnOff/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <TurnOff>
         <return>int</return>
      </TurnOff>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_Flush()

Clears the printer buffer.

C Prototype

```
int remoteprinter_Flush(void);
```

Parameters

`optFeatures`	Optional feature bits (can be ORed): • MDB_OPTFEAT_FTL – The supported file transport layer • MDB_OPTFEAT_32BIT – 32-bit monetary format • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY – Data entry. • MDB_OPTFEAT_NEGVEND – Negative vend.

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Flush/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <Flush>
         <return>int</return>
      </Flush>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_GetStatus()

Obtains the remote printer status.

C Prototype

```
struct rmpBinData remoteprinter_GetStatus(int statusType /*0*/);
```

Parameters

`statusType`	Remoter printer status.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – Feature not supported by the connected printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <GetStatus>
         <statusType>int [default:0]</statusType>
      </GetStatus>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <GetStatus>
         <return type="container">
            <data type="binary">base64binary string</data>
         </return>
      </GetStatus>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_PrintBarCode()

Obtains the remote printer print barcode.

C Prototype

```
int remoteprinter_PrintBarCode(int barcodeType /*0*/,
struct rmpBinData binData /*0*/);
```

Parameters

`barcodeType`	The barcode system type to print.
`binData`	The barcode data, including the barcode check-digit bit.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – Printer cover is open or has a paper jam.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <PrintBarCode>
         <barcodeType>int [default:0]</barcodeType>
         <binData type="container">
            <data type="binary">
            base64binary string [default:NULL]
            </data>
         </binData>
      </PrintBarCode>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <PrintBarCode>
         <return>int</return>
      </PrintBarCode>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_UpdateFirmware()

Updates the remote printer firmware.

C Prototype

```
int remoteprinter_UpdateFirmware(char * pFileName /*NULL*/);
```

Parameters

`pFileName`	Firmware file name to download to the remote printer.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <UpdateFirmware>
         <pFileName type="string">string [default:NULL]</pFileName>
      </UpdateFirmware>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <UpdateFirmware>
         <return>int</return>
      </UpdateFirmware>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_DownloadLogo()

Downloads the specified logo image file(s) to the remote printer.

C Prototype

```
int remoteprinter_DownloadLogo(int sectorId /*0*/,
char * logoFileName /*NULL*/);
```

Parameters

`sectorId`	Display sector location for the logo file download.
`logoFileName`	Logo file name to download to the remoter printer.

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
         <DownloadLogo>
         <sectorId>int [default:0]</sectorId>
         <logoFileName type="string">
         string [default:NULL]
         </logoFileName>
      </DownloadLogo>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <DownloadLogo>
         <return>int</return>
      </DownloadLogo>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_DownloadFont()

Downloads the specified font file(s) to the remote printer.

C Prototype

```
int remoteprinter_DownloadFont(int fontId /*0*/,
char * fontFileName /*NULL*/);
```

Parameters

`fontId`	The font sector location for the font file download.
`fontFileName`	The font file name to download to the remote printer.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
    <remoteprinter>
        <DownloadFont>
            <fontId>int [default:0]</fontId>
            <fontFileName type="string">
            string [default:NULL]
            </fontFileName>
        </DownloadFont>
    </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <remoteprinter>
        <DownloadFont>
            <return>int</return>
        </DownloadFont>
    </remoteprinter>
</svc_rtn>
```

remoteprinter_PrintLogo()

Sends the print logo command to the remote printer.

C Prototype

```
int remoteprinter_PrintLogo(int logoId /*0*/,
struct rmpBinData binData /*0*/);
```

Parameters

`logoId`	The logo file to print.
`binData`	Additional data.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – The font download feature is not supported by the connected printer; or a paper jam or the printer cover is open.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <PrintLogo>
         <logoId>int [default:0]</logoId>
         <binData type="container">
            <data type="binary">
            base64binary string [default:NULL]
            </data>
         </binData>
      </PrintLogo>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <PrintLogo>
         <return>int</return>
      </PrintLogo>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_PrintBitImage()

Sends a prints BitImage command to the remote printer.

C Prototype

```
int remoteprinter_PrintBitImage(int modeId /*0*/,
struct rmpBinData binData /*0*/);
```

Parameters

`modeId`	The print mode.
`binData`	BitImage binary data for printing a one-line graphic.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – The bitImage print feature is not supported by the connected printer; or a paper jam or the printer cover is open.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <PrintBitImage>
         <modeId>int [default:0]</modeId>
         <binData type="container">
            <data type="binary">
            base64binary string [default:NULL]
            </data>
         </binData>
      </PrintBitImage>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <PrintBitImage>
         <return>int</return>
      </PrintBitImage>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_Command()

Sends a command to the remote printer.

C Prototype

```
struct rmpBinData remoteprinter_Command(int cmdID /*0*/,
struct rmpBinData binData /*0*/);
```

Parameters

`cmdID`	The command to execute.
`binData`	The command data to send to the remoter printer.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – The command is not supported by the connected printer; or the printer cover is open or a paper jam was detected.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Command>
         <cmdID>int [default:0]</cmdID>
         <binData type="container">
            <data type="binary">
            base64binary string [default:NULL]
            </data>
         </binData>
      </Command>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <Command>
         <return type="container">
            <data type="binary">base64binary string</data>
         </return>
      </Command>
```

```
                                        </remoteprinter>
                                    </svc_rtn>
```

remoteprinter_Detect()

Detects the remote printer.

C Prototype

```
int remoteprinter_Detect(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- EPERM – Printer cover is open or paper jam detected.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Detect/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <Detect>
         <return>int</return>
      </Detect>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_GetLibVersion()

Obtains the remote printer library version.

C Prototype

```
struct rmpBinData remoteprinter_GetLibVersion(void);
```

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <GetLibVersion/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <GetLibVersion>
         <return type="container">
            <data type="binary">base64binary string</data>
         </return>
      </GetLibVersion>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_SetDensity()

Sets the remote printer print density.

C Prototype

```
int remoteprinter_SetDensity(int level /*0*/);
```

Parameters

`level`	Print density level, where /*0*/ is: • M1470/Artema: [-10..10] • Default is 0. • -10 brightest • +10 darkest • TG2460: [0, 8] • Default is 4. • 0 brightest • 8 darkest. • TG2460H: [48 , 56] • Default is 4. • 48 brightest • 56 darkest.

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <SetDensity>
         <level>int [default:0]</level>
      </SetDensity>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <SetDensity>
         <return>int</return>
      </SetDensity>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_SetCharacterSize()

Sets the remote printer character print size width and height.

C Prototype

```
int remoteprinter_SetCharacterSize(int width /*0*/, int height /*0*/);
```

Parameters

`width`	Character print width.
`height`	Character print height.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <SetCharacterSize>
         <width>int [default:0]</width>
         <height>int [default:0]</height>
      </SetCharacterSize>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <SetCharacterSize>
         <return>int</return>
      </SetCharacterSize>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_Cut()

Sends the cut paper command to the remote printer.

NOTE

TG2460/TG2460H printers do not support this command.

C Prototype

```
int remoteprinter_Cut(int cutType /*0*/);
```

Parameters

`cutType`	Sets the cut type: valid values are PARTIAL or FULL.

Return Values

0 =	Success.
-1 =	Error with errno set to:

- ENOMEM – Cannot malloc space.
- ENXIO – Cannot access COM device.
- EINVAL – Invalid input parameter.
- ENODEV – Cannot access printer.
- EPERM – This feature is not supported by the connected printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Cut>
         <cutType>int [default:0]</cutType>
      </Cut>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
   <remoteprinter>
      <Cut>
         <return>int</return>
      </Cut>
   </remoteprinter>
</svc_rtn>
```

remoteprinter_Close()

Sends the Close command to the remote printer.

C Prototype

```
int remoteprinter_Close(void);
```

Parameters

`optFeatures`	Optional feature bits (can be ORed): • MDB_OPTFEAT_FTL – The supported file transport layer • MDB_OPTFEAT_32BIT – 32-bit monetary format • MDB_OPTFEAT_MULTICUR – Multi currency/multi lingual • MDB_OPTFEAT_DATAENTRY – Data entry. • MDB_OPTFEAT_NEGVEND – Negative vend.

Return Values

0 =	Success.
-1 =	Error with errno set to: • ENOMEM – Cannot malloc space. • ENXIO – Cannot access COM device. • EINVAL – Invalid input parameter. • ENODEV – Cannot access printer.

XML Command

```
<svc_cmd>
   <remoteprinter>
      <Close/>
   </remoteprinter>
</svc_cmd>
```

Return XML

```
<svc_rtn>
    <remoteprinter>
      <Close>
         <return>int</return>
      </Close>
   </remoteprinter>
</svc_rtn>
```

#defines

```
#define SVC_RMP_APP_BUFF_SIZE 1024
```

Remote Printer Size Definitions

```
#define SVC_RMPRINT_LIB_VERSION_SIZE 10
#define SVC_MAX_BIT_IMAGE_SIZE_PER_LINE 511
```

Remote Printer Cut Type

```
#define SVC_RMPRNT_PARTIAL_CUT 0
#define SVC_RMPRNT_FULL_CUT 1
```

Remote Printer Type

```
#define SVC_RMP_MODEL_NONE 0
#define SVC_RMP_MODEL_M1470 1
#define SVC_RMP_MODEL_TG2460 2
#define SVC_RMP_MODEL_TG2460H 3
```

Remote Printer Bit Image Mode

```
#define SVC_BIT_IMAGE_8DOTS_SINGLE_DENSITY 0
#define SVC_BIT_IMAGE_8DOTS_DOUBLE_DENSITY 1
#define SVC_BIT_IMAGE_24DOTS_SINGLE_DENSITY 32
#define SVC_BIT_IMAGE_24DOTS_DOUBLE_DENSITY 33
```

Printer On/Off Control

```
#define SVC_RMPRNT_DISABLE_PRINTER 0
#define SVC_RMPRNT_ENABLE_PRINTER 1
```

Printer Density Levels

```
#define SVC_RMPRNT_DENSITY_LIGHTER 0
#define SVC_RMPRNT_DENSITY_LIGHT 1
#define SVC_RMPRNT_DENSITY_NORMAL 2
#define SVC_RMPRNT_DENSITY_DARK 3• #define SVC_RMPRNT_DENSITY_DARKER 4
```

Printer Status ID

```
#define SVC_ID_RMPRNT_PRINTER_STAT 1
#define SVC_ID_RMPRNT_OFFLINE_STAT 2
#define SVC_ID_RMPRNT_ERROR_STAT 3
#define SVC_ID_RMPRNT_SENSOR_STAT 4
#define SVC_ID_RMPRNT_PAPER_STAT 17
#define SVC_ID_RMPRNT_FULL_STAT 20
#define SVC_ID_RMPRNT_PRNTER_ID 21
```

Printer Barcode System Type IDs

```
#define SVC_ID_BARCODE_SYSTEM_UPC_A 0
#define SVC_ID_BARCODE_SYSTEM_UPC_E 1
#define SVC_ID_BARCODE_SYSTEM_EAN13 2
#define SVC_ID_BARCODE_SYSTEM_EAN8 3
#define SVC_ID_BARCODE_SYSTEM_CODE39 4
#define SVC_ID_BARCODE_SYSTEM_ITF 5
#define SVC_ID_BARCODE_SYSTEM_CODABAR 6
#define SVC_ID_BARCODE_SYSTEM_CODE93 7
#define SVC_ID_BARCODE_SYSTEM_CODE128 8
#define SVC_ID_BARCODE_SYSTEM_CODE32 20
```

Extension Barcode Systems

```
#define SVC_ID_BARCODE_SYSTEM_UPC_A_EX 65
#define SVC_ID_BARCODE_SYSTEM_UPC_E_EX 66
#define SVC_ID_BARCODE_SYSTEM_EAN13_EX 67
#define SVC_ID_BARCODE_SYSTEM_EAN8_EX 68
#define SVC_ID_BARCODE_SYSTEM_CODE39_EX 69
#define SVC_ID_BARCODE_SYSTEM_ITF_EX 70
#define SVC_ID_BARCODE_SYSTEM_CODABAR_EX 71
#define SVC_ID_BARCODE_SYSTEM_CODE93_EX 72
#define SVC_ID_BARCODE_SYSTEM_CODE128_EX 73
#define SVC_ID_BARCODE_SYSTEM_CODE32_EX 90
```

Remote Printer Command IDs

```
#define SVC_RMPRINT_RAW_CMD 0
```

Raw commands. With this command ID the application will pass through the total number of commands, including command, parameters, and data. The command IDs are printer-specific. The application only needs to pass through variable command parameter(s) as necessary.

TG2460 printer-specific commands:

- rmpCommand pDataIn
- rmpCommand pDataOut

Generic TG Printer(TGP) commands:#define SVC_TGP_HORIZONTAL_TAB 1

Sets the horizontal tab. NULL NULL

```
#define SVC_TGP_LINE_FEED 2
```

Prints the job and inserts a line feed. NULL NULL.

```
#define SVC_TGP_CAR_RETURN 3
```

Prints the job and inserts a carriage return. NULL NULL.

```
#define SVC_TGP_SET_RIGHT_SPACE 4
```

Sets the right-side character spacing. 1 Byte: n NULL.

```
#define SVC_TGP_SET_PRINT_MODE 5
```

Sets the print mode. 1 Byte: n NULL.

```
#define SVC_TGP_SET_PRINT_POS 6
```

Sets the absolute print position. 2 Bytes: nL nH NULL.

```
#define SVC_TGP_UNDER_LINE_MODE 7
```

Enables or disables underline mode. 1 Byte: n NULL.

```
#define SVC_TGP_INCH8_SPACE 8
```

Selects 1/8-inch line spacing. NULL NULL.

```
#define SVC_TGP_INCH6_SPACE 9
```

Selects 1/6-inch line spacing. NULL NULL.

```
#define SVC_TGP_MIN_LINE_SPACE 10
```

Sets the line spacing using minimum units. 1 Byte: n NULL.

```
#define SVC_TGP_SET_SCRIPT_MODE 11
```

Sets or resets script mode. 1 Byte: n NULL.

```
#define SVC_TGP_INIT_PRINTER 12
```

Initializes the printer. NULL NULL.

```
#define SVC_TGP_SET_HOR_TAB_POS 13
```

Sets the horizontal tab position. n Bytes: n1....nk NUL NULL.

```
#define SVC_TGP_SET_BOLD_MODE 14
```

Enables or disables bold mode. 1 Byte: n NULL.

```
#define SVC_TGP_SET_STRIKE_MODE 15
```

Enables or disabls double-strike mode. 1 Byte: n NULLs.

```
#define SVC_TGP_PRINT_AND_FEED 16
```

Prints and paper feed. 1 Byte: n NULL.

```
#define SVC_TGP_SET_INTERN_CHAR 17
```

Selects the international character set. 1 Byte: n NULL.

```
#define SVC_TGP_SET_ROTATE_MODE 18
```

Sets the print mode rotated by 90 degrees. 1 Byte: n NULL.

```
#define SVC_TGP_SET_RELATIVE_POS 19
```

Sets the relative print position. 2 Bytes: nL nH NULL.

```
#define SVC_TGP_SEL_JUSTIFICATION 20
```

Selects justification type. 1 Byte: n NULL.

```
#define SVC_TGP_PANEL_KEYS 21
```

Enables or disables the panel keys. 1 Byte: n NULL

```
#define SVC_TGP_PRINT_FEED_LINES 22
```

Prints and feed paper *n* lines. 1 Byte: n NULL.

```
#define SVC_TGP_SEL_CODE_TABLE 23
```

Selects the character code table. 1 Byte: n NULL.

```
#define SVC_TGP_GET_PAPER_STATUS 24
```

Transmits the paper status. NULL 1 Byte.

```
#define SVC_TGP_SET_SPEED_MODE 25
```

Selects the speed and quality mode. 1 Byte: n NULL

```
#define SVC_TGP_SET_UP_SIDE_DOWN 26
```

Sets or cancels upside-down printing. 1 Byte: n NULL

```
#define SVC_TGP_CUT_MOVE_BACK 27
```

The total cut and paper moving back. NULL NULL

```
#define SVC_TGP_SET_WHITE_BLACK 28
```

Enables white/black reverse print. 1 Byte: n NULL

```
#define SVC_TGP_SET_HRI_POS 29
```

Selects the print position of HRI. 1 Byte: n NULL.

```
#define SVC_TGP_GET_MODE_ID 30
```

Transmits the printer mode ID. 1 Byte: n 1 Byte.

```
#define SVC_TGP_GET_TYPE_ID 31
```

Transmits the printer type ID. 1 Byte: n 1 Byte.

```
#define SVC_TGP_GET_VERSION_ID 32
```

Transmits the printer version ID. 1 Byte: n 4 Bytes.

```
#define SVC_TGP_SET_LEFT_MARGIN 33
```

Sets the left margin. 2 Bytes: nL nH NULL.

```
#define SVC_TGP_HORI_VERT_MOTION 34
```

Sets the horizontal and vertical motion units. 2 Bytes: x y NULL.

```
#define SVC_TGP_PRINT_AREA_WIDTH 35
```

Sets the print area width. 2 Bytes: nL nH NULL.

```
#define SVC_TGP_PRINT_COUNTER 36
```

Print counter. NULL NULL.

```
#define SVC_TGP_SEL_HRI_FONT 37
```

Selects the font for HRI characters. 1 Byte: n NULL.

```
#define SVC_TGP_BARCODE_HEIGHT 38
```

Sets the bar code height. 1 Byte: n NULL,

```
#define SVC_TGP_GET_TRANSMIT_STATUS 39
```

Transmits the paper sensor status. NULL 1 Byte.

```
#define SVC_TGP_SET_BARCODE_WIDTH 40
```

Sets the bar code width. 1 Byte: n NULL.

```
#define SVC_TG2460_MOVE_BACK 41
```

Moves back one character. NULL NULL.

```
#define SVC_TG2460_MACRO_DEFINITION 42
```

The start or end of the macro definition, NULL NULL.

```
#define SVC_TG2460_EXECUTE_MACRO 43
```

Executes a macro. 3 Bytes: r t m NULL.

```
#define SVC_TG2460_SEL_COUNTER_MODE 44
```

Selects the counter print mode. 2 Bytes: n m NULL.

```
#define SVC_TG2460_SEL_COUNT_MODEA 45
```

Selects count mode-A. 6 Bytes: aL aH bL bH n r NULL.

```
#define SVC_TG2460_SET_COUNTER 46
```

Sets the counter. 2 Bytes: nL nH NULL.

```
#define SVC_TG2460_SEL_COUNT_MODEB 47
```

Selects count mode-B. 10 Bytes: sa ; sb ; sn ; sr ; sc ; NULL.

```
#define SVC_TG2460_SET_SCRIPT 48
```

Sets the superscript and subscript parameters. 1 Byte: n NULL

```
#define SVC_TG2460H_SET_VERT_POS 49
```

Sets the absolute vertical print position in page mode. 2 Bytes: nL nH NULL.

```
#define SVC_TG2460H_POWER_LED_BAR 50
```

Powers on or off the LED bar. 1 Byte: n NULL.

```
#define SVC_TG2460H_SET_CPI_MODE 51
```

Sets or cancels CPI mode. 1 Byte: n NULL

```
#define SVC_TG2460H_SET_NOTCH_DISTANCE 52
```

Sets the notch distance. 2 Bytes: nH nL NULL.

```
#define SVC_TG2460H_ALIGN_HEAD 53
```

Aligns the print head with the notch. NULL NULL

```
#define SVC_TG2460H_ALIGN_AUTO_CUTTER 54
```

Aligns the auto cutter with the notch. NULL NULL

```
#define SVC_MCSP_DELETE_FONTS 55
```

Instantly deletes fonts. NULL NULL.

```
#define SVC_MCSP_GET_FONT_INFO 56
```

Queries the font info. 1 Byte: n 8 Bytes: Font info: see MCS printer spec

```
#define SVC_MCSP_GET_LOGO_CRC 57
```

Queries the logo CRC. logoIndex range[01–15] 1 Byte: n 2 Bytes.

```
#define SVC_MCSP_SET_FONT 58
```

Changes the font. fontIndex range[0–4].

```
#define SVC_MCSP_SET_BIG_LETTERS 59
```

Sets the print style to big letters. NULL NULL

```
#define SVC_MCSP_GET_PRINTER_ID 60
```

Obtains the printer ID. NULL 2 Bytes.

```
#define SVC_MCSP_SET_SMALL_LETTERS 61
```

Sets the print style to small letters.

```
#define SVC_MCSP_FORWARD_NEXT_MARK 62
```

Forwards the printer to next printer mark.

```
#define SVC_MCSP_SET_NORMAL_LETTERS 63
```

Sets the print style to normal letters.

```
#define SVC_MCSP_RECEIPT 64
```

Attendance test: NULL NULL

```
#define SVC_MCSP_READ_SERIAL 65
```

Returns the 12-digit serial number. NULL 12 Bytes.

```
#define SVC_MCSP_SET_POWEROFF_TIMEOUT 66
```

Sets the power off timeout in seconds, valid value[10–60], default 20: 1 Byte: n NULL.

```
#define SVC_MCSP_GET_POWEROFF_TIMEOUT 67
```

Queries the power off timeout NULL 2 Bytes: Power off timer value.

```
#define SVC_MCSP_GET_SENSOR_INFO 68
```

Obtains the temperature of print head sensor.

```
#define SVC_MCSP_DISPLAY_COUNTER 69
```

Displays counter NULL 8 Bytes: Printer counter number, little endian.

```
#define SVC_RMPRINT_NUM_CMD 70
```

Returns the total command number.

rmpBinData Structure

This is the remote printer binary data structure.

```
#include <svc_remoteprinter.h>
```

Data Fields

```
void  data
```

Binary data.

```
int data_count
```

Bytes of binary data.

```
void rmpBinData::data
```

Binary data.

```
int rmpBinData::data_count
```

Bytes of binary data.

rmpCmdParams Structure

This is the command parameters structure.

```
#include <svc_remoteprinter.h>
```

Data Fields

```
char id
char rmpCmdParams::id
```

The character ID.

```
int cmdLen
int rmpCmdParams::cmdLen
```

The length of the command data.

```
char cmdData [SVC_RMP_APP_BUFF_SIZE]
char rmpCmdParams::cmdData[SVC_RMP_APP_BUFF_SIZE]
```

The command buffer size.

rmpDataReturn Structure

This is the returned data structure.

```
#include <svc_remoteprinter.h>
```

Data Fields

```
int len
int rmpDataReturn::len
```

Bytes of return data.

```
char data [64]
char rmpDataReturn::data[64]
```

The return data.

V/OS on the UX300 Terminal

This section contains V/OS information specific to Verifone UX300 unattended terminals. This information supersedes that in the main body of this document. Topics include:

- Updating the Terminal
- Performing a Card Reader Reset
- Sysmode Features

Refer to Device Configuration for Development for UX300 connections to make for application development.

Graphical User Interface (GUI)

The UX300 GUI is covered in the ADK DirectGUI component. The GUI is similar to VX 520, but has a different keyboard layout. Access key definitions (key mapping) in HTML files may need modification to match the UX keyboard. The GUI also discusses PIN handling using the UX100 PIN Entry Functions. Documentation and sample code is in the ADK in the DOC releases package. The use of keyboard and display on the remote PIN entry device is fully transparent.

Updating the Terminal

To update the UX300:

1. Select the desired tar package from the **Update** menu (for example, dl.release-30XX0101-buildallux300. tgz).

CAUTION

The buildall update package erases every installed system-signed package. You must reinstall the application developer console package and any system-signed packages required for applications.

Installation takes approximately 5 minutes

2. Choose **Yes** at the restart prompt.

   You can manually reboot the machine using a power-cycle or from the menu. The terminal update initiates on power up.

NOTE

The terminal may reboot during installation.

   The menu displays on update success.

3. Log in as supervisor, and select Information > Basic Info and scroll to verify the current version.

The last entry is the current build, which should correspond to the package chosen in Step 1.

I/O Module

The UX300 I/O module is not customer serviceable. Do not remove the I/O module.

CAUTION

Removing the UX300 I/O module triggers a hardware tamper. All cryptographic keys are erased, including keys required for the secure USB link. The numeric keys are disabled, and the terminal must be returned to Verifone for repair.

Performing a Card Reader Reset

The UX300 terminal and UX100 PINpad are updated using a USB thumb drive with a FAT partition. If you need to update the card reader, ask your Verifone representative for the *Updating a UX300/UX100 Device over USB Manual*.

Sysmode Features

- When Sysmode launches, the orange LED lights solid to indicate that system mode is executing.
- If launching Sysmode manually (by continually pressing the sysmode button), the update begins automatically when the USB flash memory device is connected. All packages in the V/OS-auto-update USB package directory (see Secure Installer). The yellow LED blinks during installation. The green LED blinks three times for a successful installation. The red LED blinks three times if the installation failed.

VX 520 and UX300 Sysmode Application

This section presents Sysmode for the UX300 and VX 520 terminals.

Enter Sysmode

1 Enter Sysmode:

- Press the Sysmode button on the back panel on the UX300 (Figure 17).

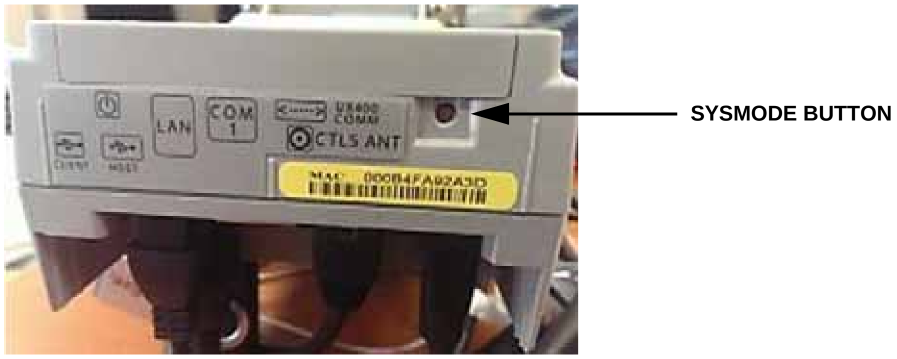


Figure 17 UX300 Sysmode Button

- Simultaneously press the 1, 5, and 9 keys on the VX 520 terminal.

2 On the keypad, select the user, and press **OK**.

3 Enter the user password.

The appropriate menu for the user displays.

Password Management

The UX terminal supports three password groups:

- USERS – Log in password for the supervisor, level1, and level2 users
  - Supervisor – Access to all Sysmode menus; this user is reserved for Verifone service operations
  - level1, level2, and maintenance – Can only access the Sysmode menus allowed by the Security Policy File
- SWITCH – switch1 and switch2 passwords to activate devices when an Anti-Removal Switch is pressed

You are prompted to set passwords when the device first boots. Use the keypad and press **OK** to set the new password. Each password has an expiration set at manufacture. Save the password in a secure location.

NOTE level2 passwords cannot be recovered. UX terminals must be returned to Verifone or an authorized repair center if the level2 password (customer's supervisor) is lost.

Security Policy File

This file contains the following definitions for level1 and level2 user access:

- level1 has access to the ARS menu and can change ASR passwords
- level2 (usually the customer's supervisor user) has level1 access, plus access to the Expire Switch Password menu to reset ASR passwords to their default value
- Maintenance (customer's operator for on-site maintenance to check device status) only has access to the Information and Diagnostic menus

Change Passwords

Use the following procedure to reset the USER password.

Use the arrow keys to navigate through the Sysmode menus. Press **OK** after making each selection.

1. Select **Login >** ***user***.
2. Enter the current password for the selected user.
3. Select **Security > Password Mgr**.
4. Select **Change > Switch**.
5. Select the password to modify (for example, switch1 for UX300 ARS).
6. Enter the current password for the selected user.
7. Enter and confirm the new password.

   Save the password in a secure location.

Sysmode Menus

Sysmode menus are configuration specific. For example, when an UX300 is connected to a UX100 PINpad, an UX100 Sysmode menu displays; or if you have VX 520 with GPRS, a GPRS modem menu is available. Depending on your configuration, the following menus are available when the Sysmode application is running:

Information

- Basic info: Display general information on the unit (S/N, P/N, revision, model, and so on), and low-level software (Linux kernel, U-Boot, SBI, and so on)
- Ports/Options (UX300): Display COM ports states
- Software: List of installed software
- FW Version: Display the firmware version.

- Modem: Display modem information (for example, code, FW, or library version)
- Memory: Display flash memory and RAM usage
- Logs: Display the tamper and installation logs
- Counters (UX300): Display the internal counters (PIN entries, SC insertions, and so on)
- UX Info: Display information about the UX100 or UX110 connected to the UX300

Administration

- Touch Panel: Run UX200 touch panel calibration
- Config: Edit the user or system config file
- Communication: Configure the ECR, USB, Network, and VHQ Estate Manager
- Date/Time: Configure the system date and time
- Network: Configure and display information on network interfaces
- File Manager: Browse system files
- Power Settings
  - Display: Configure UX200 display
  - Audio: Set UX200 volume
- Remove/Uninstall: Remove selected or all user bundles.

NOTE

This feature cannot remove OS/system bundles.

- MMS: Configure the remote MMS server

Update

- USB Memory: Browse the USB device and install selected software
- Serial: Install update over the serial link (only with the Mx800Downloader application on COM1 only)
- Netloader: Install applications over Ethernet (only with the Mx800Downloader application)
- MMS: Initiate a session with the MagIC Management system to update S/W configuration remotely

Transfer

- Serial/USB: Install bundle using serial link (using the Mx800Downloader application)

- USB/SD: Install a bundle from the USB memory device or an SD card
- ERC download
  - Netloader: Install a bundle over the Ethernet port (using the Mx800Downloader application)
  - FTP/SFTP: Install a bundle using FTP or from an SFTP server
  - PAYware: Install a bundle from a PAYware server

Security

- Key loading
  - Load Bank/VRK keys using serial link (COM1 only). Access is restricted with two key loader passwords
  - Perso Pinpad: Create key pair and build certificate using a smartcard for mutual authentication V/OS PINpad
- Tamper status: Display security status
- Key list: Display loaded secure keys/scripts (RKL, VSS, DUKPT, and MS), and also displays status on P-Series authentication
- Password Mgr: Change user and key load passwords or expired password
- Verishield tree: List all installed Verifshield keys
- Security policy: List the security policy file name
- VRP status:
- Service switch: Display ARS information
- Removal Switch: (ARS management) Display information about removal switch status and reset removal switch status

Diagnostic

- Battery: Display battery status and voltage
- Display: Perform display diagnostics, which attempts to display a bitmap
- Touch Panel: Perform touch panel diagnostics of touch and signature response
- Keyboard: Display the matrix with the keys. Press a key to increase the counter number for that key value. There is no button to return to a previous menu. If no keys are pressed in 5 seconds, the diagnostic test automatically exits
- Card
  - Smart Card: Perform smart card device diagnostics
  - Magstripe: Perform magnetic stripe device diagnostics
  - Contactless: Perform Contactless device diagnostics

- Communications
  - Ethernet: Perform Ethernet connection diagnostics (ping test)
  - Serial: Perform serial loop-back diagnostics. For this test to work correctly, the loop-back cable shown in Figure 18 is required.

NOTE

Insert the loop-back cable into serial port 1 (COM1) with NO console enabled.

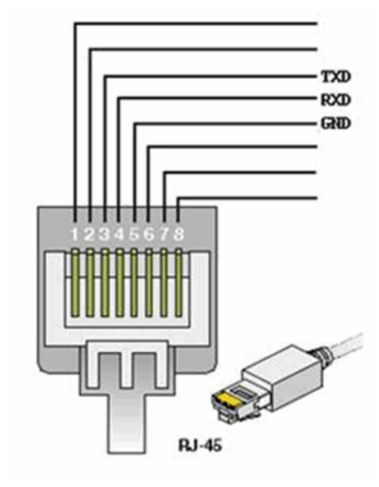


Figure 18 Loop-back Cable Pin Out

  - Modem: Perform modem diagnostics
  - USB: Perform USB diagnostics
- Printer
  - Printer (VX 520): Perform diagnostics for the printer. It prints the numbers 123456789012345678901234, and then a black square
  - Remote Printer (UX300): Perform diagnostics for the remote printer. It prints the string 123456789012345678901234, a barcode, and then displays "**DIAGNOSTIC TEST FINISHED**" and cuts the paper
- Buzzer/Audio: Perform diagnostics for the buzzer
- LEDs
  - VX 520 LEDs: Perform diagnostics for the LOGO LED and keyboard LED
  - UX300 LEDs: Perform diagnostics for the three LEDs (green, orange, red) on the front panel
- (P-Series only) PINpad
  - Basic info: Display general PINpad information (P/N, S/N, PCI code, model)

- Security:
  - Status: Display security status (tamper)
  - Key list: Display the PINpad key list
  - Unlock: Detamper the PINpad using the “UNLOCK” smartcard
  - Reset: Remove PINpad keys using the “RESET” smartcard
  - Perso: Generate key pair and certificate using the “AUTH_PP” smartcard. This is required for mutual authentication V/OS PINpad
- Diagnostic: Perform diagnostics for all PINpad devices separately

UX100 (UX300 with UX110 attached)

- Basic info
  - Unit type: Display information about the unit
  - Operation Mode: Display the operation mode of the unit
  - Heater Mode: Display information about the heater
  - ID Card Info: Display information about the ID card
- Software: Display the software version
- Security
  - Status: Display security status (tamper)
  - Key list: Display PINpad key list
  - Reset: Remove the PINpad keys using the “RESET” smartcard
  - Perso: Generate a key pair and certificate using the “AUTH_PP” smartcard, which is required for mutual authentication V/OS PINpad
- Configure
  - Contrast: Set the screen contrast
  - Backlight: Set the timeout and mode of the backlight (On/Off/Auto)
  - Keyboard beep: Set the Frequency, Duration, and Volume (currently not used) of the UX100 keyboard
- Start (Run App on UX300): Launch user applications (not available when unit tampered)
- Exit
  - Reboot: Restart the device
  - Logout: Return to the log in menu
- Manufacturing (only visible on PRODUCTION or MANUFATURING units)
  - Load MIB: Initiate communication with the MIBDownload PC tool
- Dev (only visible on DEVELOPMENT units)
  - Execute: Start the selected application

- Update local: Install package(s) located in `/mnt/flash/install/dl`
- Remove bundle: Remove the selected bundle

NOTE

There is no restriction on the bundle type that can be removed on this screen. Ensure you have selected the proper bundle.

- Load MIB: Initiate communication with the MIBDownload PC tool
- Detamper: Call vfi_deTamper()
- Exit: Close the Sysmode application (no front-end application running)

Special Features

- (VX 520) When Sysmode launches, the keyboard buzzer is disabled
- (UX300) When Sysmode launches, the orange LED is on steady to indicate that system mode is executing
- (UX300) If Sysmode is launched manually, an automatic update from a USB flash memory drive starts. All packages in the /VOS-auto-update USB package directory install. An automatic update is indicated with a blinking yellow LED during installation. On successful installation, the green LED blinks three times; on failure the red LED blinks three times.

APPENDIX G

V/OS on the VX Terminal

This section contains V/OS information specific to VX terminals. For these terminals, this information supersedes that in the main body of this document.

NET Services

This section presents the VX terminal-specific NET Services used to configure IPv4 Ethernet and Wi-Fi devices. For more information, see Chapter 21, NET Service Ethernet and Wi-Fi Configuration.

NOTE

NET Service supports both IPv4 and IPv6, however, in V/OS only IPv4 is currently supported. Parameters related to IPv6 will be ignored.

NET Service uses an XML file to store the parameters. To configure IPv4 Ethernet and Wi-Fi communications device (terminal specific) and support various protocols (IP, PPP, SSL, and so on). It provides calls to:

1. Write Ethernet Parameters to netconf.xml.
2. Write netIfConfig Structure to netconf.xml file using net_interfaceSet().
3. Read Ethernet Parameters from netconf.xml.
4. Write Wi-Fi Parameters to netconf.xml.
5. Write netIfConfig structure to netconf.xml file using net_interfaceSet().
6. Read Wi-Fi Parameters from netconf.xml.
7. Use net_interfaceSet() and the `netIfconfig` structure to write the PPP parameters to netconf.xml file:

   a. Create the `netIfconfig` structure and provide the PPP parameters:

```
struct netIfconfig config;
memset( &config, 0, sizeof(struct netIfconfig) );
strncpy( config.interface, "ppp0", strlen("ppp0"));
/* Set interface parameter to "ppp0" for PPP */
strncpy( config.user, "myusername", strlen("myusername"));
strncpy( config.password, "mypassword", strlen("mypassword"));
strncpy( config.pppPort, "com_name",  strlen("com_name") );
```

NOTE

`com_name` refers to the communication device (modem, serial port, GPRS, and so on) used to connect the PPP. Possible values are shown in Table 13.

Table 13 pppPort Values

Media Name	Value	Description
NET_MONDEM_PSTN	"modem_uart"	Default internal PSTN modem name.
NET_GPRS_LAYER	"gprs_layer"	GPRS interface name.
NET_SERIAL_PORT	"serial_port"	Serial port interface name.

b Use net_interfaceSet() to write PPP parameters to netconf.xml:

```
int ret = net_interfaceSet( config );
```

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="comname"/>
</PPPPROTO_LIST>
<!-other tags -- >
</SETTINGS>
```

`PPPPROTO` is the communication tag for PPP parameters.

NOTE

"`ppp1`" must be used as interface name when connecting PPP over GPRS.

"`ppp0`" is reserved for modems and serial ports.

c Use net_interfaceGet() to read the `netIfconfig` structure from netconf.xml:

```
struct netIfConfig config = net_interfaceGet( "ppp0/ppp1" );
```

“`ppp0/ppp1`" is passed to indicate the PPP parameters to read.

On success (errno=0), `netIfconfig` contains the PPP parameters read from the netconf.xml file.

Modem and GPRS Parameters

The PPP layer requires a modem or GPRS device connection. To store modem and GPRS parameters in netconf.xml.

1 Use net_interfaceModemExtSet() and the `netModemInfo` structure to write V/OS PSTN modem parameters to the netconf.xml file:

```
int net_interfaceModemExtSet( char *iface, struct netModemInfo mdnInfo );
```

where,

- `iface` is NET_MONDEM_PSTN for a V/OS standard modem.
- `struct netModemInfo` is the general structure used to Convoy the modem parameters.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

```
<SETTINGS>

<MODLINK_LIST>
<MODLINK name_id="modem_uart" number="0123456789" dial_type="1" mode="2"
max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1"/>
</MODLINK_LIST>
<!-other tags -- >
</SETTINGS>
```

`MODLINK` is the communication tag for PSTN modem parameters (for standard PSTN modems).

2 Use net_interfaceModemExtGet() and the `netModemInfo` structure to read the PSTN modem parameters from netconf.xml:

```
struct netModemInfo net_interfaceModemExtGet( char *iface );
```

where,

- `iface` is NET_MONDEM_PSTN for a V/OS standard modem.

On success (errno=0), `netModemInfo` contains the PSTN parameters read from the netconf.xml file.

3 Use net_interfaceModemExtSet() and the `netModemInfo` structure to write PSTN modem parameters to netconf.xml:

```
int net_interfaceModemExtSet( char *iface, struct netModemInfo mdnInfo );
```

where,

- `struct netModemInfo` is the general structure used to Convoy the modem parameters.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<MODLINK_LIST>
<MODLINK name_id="modem_pstn" number="0123456789" dial_type="1" mode="2"
max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1"/>
</MODLINK_LIST>
<!-other tags -- >
</SETTINGS>
```

`MODLINK` is the communication tag for PSTN modem parameters (for standard PSTN modems).

4 Use net_interfaceModemExtGet() and the `netModemInfo` structure to read PSTN modem parameters from netconf.xml:

```
struct netModemInfo net_interfaceModemExtGet( char *iface );
```

where,

- `iface` is NET_MONDEM_PSTN for a V/OS standard modem.

On success (errno=0), `netModemInfo` contains the PSTN parameters read from the netconf.xml file.

5 Use net_interfaceModemIsdnSet() and the `netModemIsdnInfo` structure to write ISDN modem parameters to netconf.xml:

```
int net_interfaceModemIsdnSet( struct netModemIsdnInfo mdnInfo );
```

where,

- `struct netModemInfo` is the general structure used to Convoy the modem parameters.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<!-other tags -- >

<MODISDNLINK_LIST>
<MODISDNLINK name_id="modem_isdn" isdn_prot="10" number="959"
x25_packet_size="0" facilities="" nui="" x25_number="" user_data=""
max_retries="2" connect_timeout="30" country_code="FD"/>
</MODISDNLINK_LIST>

</SETTINGS>
```

`MODISDNLINK` is the communication tag for PSTN modem parameters (for standard PSTN and ISDN modems).

6 Use net_interfaceModemIsdnGet() and the `netModemIsdnInfo` structure to read ISDN modem parameters from the netconf.xml file:

```
struct netModemIsdnInfo net_interfaceModemIsdnGet();
```

On success (errno=0), `netModemIsdnInfo` contains the ISDN parameters read from the netconf.xml file.

7 Use net_interfaceGprsSet() and the `netGprsInfo` structure to write GPRS parameters to the netconf.xml file:

```
int net_interfaceGprsSet( struct netGprsInfo gprsInfo );
```

where,

- `struct netGprsInfo` is the general structure used to Convoy the GPRS parameters.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<!-other tags -- >

<GPRS_LIST>
```

```
<GPRS name_id="gprs_layer" APN="orange.fr" />
</GPRS_LIST>

</SETTINGS>
```

GPRS is the communication tag for GPRS parameters.

8 Use net_interfaceGprsGet() and the `netGprsInfo` structure to read GPRS parameters from the netconf.xml file:

```
struct netModemIsdnInfo net_interfaceModemIsdnGet();
```

On success (errno=0), `netGprsInfo` contains the GPRS parameters read from the netconf.xml file.

9 Bring Up a Specific Network Interface.

10 Configure the Ethernet Interface.

11 Configure the Wi-Fi Interface.

12 Configure the PPP interface:

Configuring PPP Over V/OS PSTN Modems

a Use the net_interfaceSet() command and the `netIfconfig` structure to write the PPP parameters in netconf.xml.

b Use the `net_interfaceModemExtSet( NET_MONDEM_PSTN, netModemInfo)` command to write the V/OS PSTN modem to netconf.xml.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="modem_uart"/>
</PPPPROTO_LIST>

<MODLINK_LIST>
<MODLINK name_id=" modem_uart" … other parameters … />
</MODLINK_LIST>

</SETTINGS>
```

c Use the `net_interfaceUp( "ppp0", NET_IP_V4 )` command to connect the modem and bring the PPP interface up.

- Use the `net_interfaceDown( "ppp0" )` command to disconnect the PPP interface and hang up.

Configuring PPP Over GPRS

a Use the net_interfaceSet() command and the `netIfconfig` structure to write the PPP parameters in netconf.xml.

NOTE

Use `ppp1` as the PPP interface name for the GPRS interface.

b Use the `net_interfaceGprsSet(netGprsInfo830)` command to write GPRS parameters in netconf.xml.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp1" user="username" password="password" auth="pap"
next_media="gprs_layer"/>
</PPPPROTO_LIST>

<GPRS_LIST>
<GPRS name_id="gprs_layer" … other parameters … />
</GPRS_LIST>

</SETTINGS>
```

c Use the `net_interfaceUp( "ppp1", NET_IP_V4 )` command to connect to the modem and then bring the PPP interface.

d Use the `net_interfaceDown( "ppp1" )` command to disconnect the PPP interface and hang up.

Configuring PPP Over the Serial Port

a Use the `net_interfaceSet( struct netIfconfig )` command to write the PPP parameters to netconf.xml.

NOTE

Use `ppp0` as the PPP interface name.

b Use the OPTION tag Convoy the PPP over serial port parameters.

Example:

```
char p_option[NET_OPTION_MAX];

/*<** Define the parameters used for PPP connection */
#define PPP_OVER_SERIAL "asyncmap 0 noauth noipdefault crtscts
local_ip:remote_ip com_port speed
```

```
strncpy(p_option ", PPP_OVER_SERIAL, strlen(PPP_OVER_SERIAL) );
```

where,

- `local_ip` = terminal IP address
- `remote_ip` = server IP address
- `com_port` = serial port name (for exmple, /dev/ttyAMA1)
- `speed` = communication speed (for exmple, 19200)

c Use the `net_optionSet( "ppp0", p_option )` command to write the PPP over serial parameters to netconf.xml.

The result netconf.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>

<PPPPROTO_LIST>
<PPPPROTO name_id="ppp0" user="username" password="password" auth="pap"
next_media="serial_port"/>
</PPPPROTO_LIST>

<OPTION_LIST>
<OPTION name_id="ppp0" option=" asyncmap 0 noauth noipdefault crtscts
local_ip:remote_ip com_port speed" />
</OPTION_LIST>
</SETTINGS>
```

d Use the `net_interfaceUp( "ppp0", NET_IP_V4 )` command to connect to the modem and then bring the PPP interface.

e Use the `net_interfaceDown( "ppp0" )` command to disconnect the PPP interface and hang up.

Management Communication Sessions for Applications

NET Service establishes communication sessions for applications. The following are supported session types:

- TCP/IP (or SSL) over Ethernet
- TCP/IP (or SSL) over WIFI
- TCP/IP (or SSL) over PPP over Modem (PSTN, ISDN)
- TCP/IP (or SSL) over PPP over Serial Port
- TCP/IP (or SSL) over PPP over GPRS
- Modem direct calls

The application uses a configuration file (XML format) to perform communication sessions through the NET Service. The configuration file uses communication *tags* to describe the session type such as SSL over PPP over modem. It also stores the corresponding parameters such as SSL certificates, PPP user name and password, the phone number, and so on. Figure 19 shows how NET Services facilitate application communications.

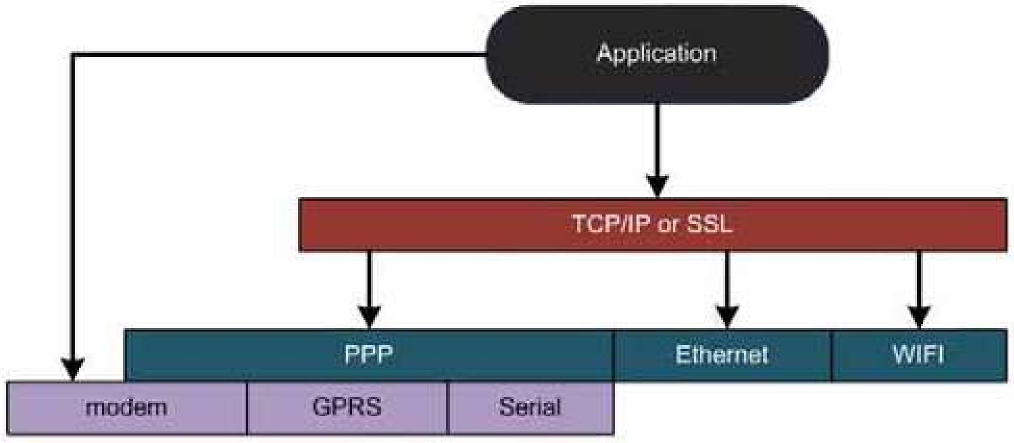


Figure 19 NET Service Communication Sessions

The application uses a NET Service API to load its session file to perform a connection. It then calls another NET Service API to start the session and retrieves an output value session ID (handle) that identifies the current session. The application uses this handle for all subsequent calls (read/write/disconnect).

Establishing Communication Sessions

An application must provide a session file in XML format to perform a communication session using the NET Service. Each supported communication type is described in this session and how to build them from the application context.

Figure 20 summarizes the main steps for an application to establish a communication session using the NET Service.

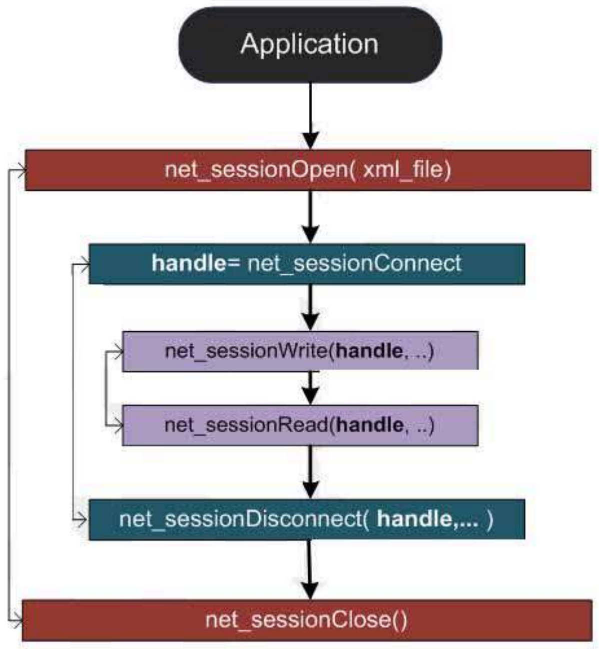

Application
net_sessionOpen(xml_file)
handle= net_sessionConnect
net_sessionWrite(handle, ..)
net_sessionRead(handle, ..)
net_sessionDisconnect(handle,...)
net_sessionClose()

Figure 20 NET Service Communication Session Flow

1 Use net_sessionOpen() to load the session file.

```
int net_sessionOpen( char *p_xmlFile );
```

This function takes as parameter the configuration file in XML format, and returns 0 on success or -1 on error.

2 Use net_sessionConnect() to start the communication session.

```
int handle = net_sessionConnect( int sessionType );
```

A session handle (>0) returns on success.

You can define more than one communication session (for example, SSL over Wi-Fi and SSL over Ethernet) in the same XML configuration file. The `sessionType` parameter selects which session to use first.

NOTE `handle`–the value returned by `net_sessoionConnect (>0))`–uniquely identifies the current session. The application uses `handle` to exchange data with the remote server.

3 Use the net_sessionWrite() to send data to the remote server.

```
int net_sessionWrite( int handle, struct netDataTransfer data );
```

where,

- `handle` = the value returned by net_sessionConnect()
- `data` = the data to send to the peer. The structure definition is:

```
struct netDataTransfer
{
   void *data;/**< array of bytes to transfer */
   int data_count; /**< number of bytes to transfer */
};
```

4 Use the net_sessionRead() to read data from the remote server.

```
struct netDataTransfer net_sessionRead(int handle, int count, int
timeout);
```

where,

- `handle` = the value returned by net_sessionConnect()
- `count` = the number of bytes to read
- `timeout` = the number of seconds to wait for data

A `netDataTransfer` structure returns. The structure definition is:

```
struct netDataTransfer
{
   void *data;/**< array of bytes to transfer */
   int data_count; /**< number of bytes to transfer */
};
```

NOTE On success (errno=0), the application must free the data in the `netDataTransfer` structure.

5 Use net_sessionDisconnect() to terminate the current communication session.

```
int net_sessionDisconnect(int handle );
```

where,

- `handle` = the value returned by net_sessionConnect()

The handle value is passed as a parameter to indicate which session to close.

NOTE Once net_sessionDisconnect() is called, the handle becomes invalid. To obtain a new handle, the application can start a new session by calling net_sessionConnect().

6 Use net_sessionClose() to unload the XML configuration file.

```
int net_sessionClose( void );
```

This call releases the context. All handles are no longer valid.

NOTE Once net_sessionDisconnect() is called, the handle becomes invalid. To obtain a new handle, the application can start a new session by calling net_sessionConnect().

XML Communication Files

The following is the general structure of a NET Service communication files:

```
<?XML VERSION="1.0" ENCODING="UTF-8" STANDALONE="YES" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="session name" next_media="tag1 name" />
</SESSION_LIST>

<TAG1_LIST>
<TAG1 name_id="tag1 name" param1a="value" …param1n="value"
next_media="tag2 name"/>
</TAG1_LIST>

<TAG2_LIST>
<TAG2 name_id="tag2 name" param2a="value" …param2n="value" />
</TAG2_LIST>

</SETTINGS>
```

Every NET Service XML communication file begins with a SESSION tag. This is mandatory. Each SESSION tag entry is uniquely identified by the `name_id` parameter within the entire XML file (here session name). The `next_media` parameters (`tag1 name`) links the SESSION tag to another tag in the XML file (TAG1).

TAG1 indicates what the purpose of this session (for example, TCP/IP, SSL, modem, and so on). TAG1 parameters (`param1a` ... `param1n`) Convoy data for TAG1 (for example, the IP address for a TCP/IP session or SSL certificates for SSL).

TAG1 is uniquely identified by its `name_id` (`tag1 name`) within the entire XML file. `next_media` refers to another tag (TAG2) in the XML file may be required to fully work. For example, if TAG1 is TCP/IP, TAG2 can store parameters for Ethernet, Wi-Fi, or PPP. Some tags such as SESSION require a `next_media` parameter, and some don't such as Ethernet, Wi-Fi.

Every NET Service communication file follows this structure.

NET Service Communication Tags

This section lists all supported communication tags and corresponding attributes. NET Service tags are grouped into five categories:

- Session tags define an entry point for an XML file.

TAG Name	Description
SESSION	Entry point for a communication file.

- Protocols tags define SSL and TCP/IP layers in the configuration file.

TAG Name	Description
SOCKPROTO	TCP/IP socket parameters.
SSL	SSL parameters.

- IP Interfaces tags define the IP interfaces used by the protocol tags.

TAG Name	Description
ETHLINK	Ethernet parameters.
WIFI	Wi-Fi parameters.
PPPPROTO	PPP parameters.

- Modem tags define tags for modems and the GPRS layer. Modem tags can link to a PPP layer or be used directly by a SESSION entry.

TAG Name	Description
MODLINK	PSTN modem parameters.
MODISDNLINK	ISDN modem parameters.
GPRS	GPRS parameters.

- Option tags define tags for OPTION.

Tag Name	Description
OPTION	Option parameters.

NOTE

A tag is defined by a list of attributes and their values. Not all attributes are required. Mandatory attributes are written in bold in the following tables.

Tag name: SESSION

Use this tag to configure a communication session.

Attribute Name	Type	Accepted Values	Description
name_id	***char***[32]	media name	Identification name of the media
Priority	int	1 …	Priority of the session
Timeout	double	1 …	Connection timeout (RFU)
Retry	int	0 …	Number of connection tries (RFU)
next_media	***char***[32]	Next media name	Identifies the protocol used by the session.

Tag name: SOCKPROTO (TCP/IP)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identification name of the media.
protocol	char[8]	tcp	Socket type (for example, one possible value is TCP).
remote_ip	char[128]	IP:PORT or DNS:PORT	Server address and port number.
next_media	char[32]	Next media name	Identifies the physical layer used by this protocol such as Wi-Fi, ETHERNET, or PPP.

Tag name: SSL

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the SSL layer.
remote_ip	char[128]	IP:PORT or DNS:PORT	Socket protocol.
server_profile	char[128]	Full path + file name	Identifies the place where the server CA is stored.
client_profile	char[128]	Full path + file name	Identifies the place where the client certificate and the client private key are stored.
session_ttl	int	0…	Indicates in seconds how long to keep the Resume Session before performing a complete handshake.
next_media	char[32]	Next media name	Identifies the linked media.

Tag name: ETHLINK (Ethernet)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
usedhcp	int	0, 1	1=DHCP is used 0 = DHCP not used
local_ip	char[16]	ip range	Identifies the network adapter IP address.
netmask	char[16]	ip range	Network mask.
broadcast	char[16]	ip range	Broadcast.
gateway	uint8[16]	ip range	Gateway IP address.
Activate	int	0,1	Indicates if this interface must be activated.
Dhcpid	char[16]		The client id string for DHCP.
Clienthostname	char[128]		The client hostname string for DHCP.
Speed	char[16]	(0, Auto), (1,100F), (2, 10F), (3,100H), (4,10H)	Defines the link speed as 10F, 100F, 10T, 100T, or AUTO (default is AUTO).
dns1	char[16]	ip range	Domain Name Server number 1.
dns2	char[16]	ip range	Domain Name Server number 2.

Tag name: WIFI

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
usedhcp	int	0, 1	1=DHCP is used 0 = DHCP not used
local_ip	char[16]	ip range	Identifies the network adapter IP address
netmask	char[16]	ip range	Network mask.
broadcast	char[16]	ip range	Broadcast.
gateway	uint8[16]	ip range	Gateway IP address.
activate	int	0,1	Indicates if this interface must be activated.
suppliconf	char[16]		Wi-Fi configuration security file location (standard format).
dns1	char[16]	ip range	Domain Name Server number 1.
dns2	char[16]	ip range	Domain Name Server number 2.

Tag name: PPPPROTO (PPP)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies name of the media.
user	char[32]		User name for PPP.
password	char[32]		Password for PPP.
auth	char[32]	auto, chap, pap	Authentication mode for PPP.
pppOption	char[16]		RFU.
next_media	char[16]	ip range	Indicates the media used to connect the PPP layer

Tag name: MODLINK (PSTN Modems)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
number	char[32]		The phone number to dial.
dial_type	char	1=Tone, 2=Pulse	Identifies the dial tone type.
mode	char	1=Sync, 2=Async	Identifies the mode as asynchronous or synchronous.
max_retries	char	0…	Identifies the number of dial retries.
max_line_rate	char	(1=**300**), (2=**300_B**), (3=**1200**), (4=**1200_FAST**), (6=**1200_B**), (7=**2400**), (8=**4800**), (12=**14400**), (14=**33600**), (15=**56000**),	Identifies the modem connection rate.
connect_timeout	char		Identifies the whether the modem is waiting for an answer. If expired, there are no more retries.
error_correction	char	1 = Activate correction error 0 = otherwise	Indicates whether to activate the modem connection error.
compression	char	1	error_correction must be enabled to use compression.

Tag name: MODISDNLINK (Modem ISDN)

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
number	char[32]		The phone number to dial
isdn_prot	char	(3, HDLC_SYNC_TO_ASYNC), (4, HDLC_TRANSPARENT), (10, X75), (20, X31_B_CHL), (21, X31_D_CHL)	ISDN transmission protocols.
x25_packet_size	char[5]	64 - 2048	Identifies the mode as asynchronous or synchronous.
facilities	char[20]	starting with 'G'	Access to X.25 Closed User Group.
nui	char[20]	a-z, A-Z, 0-9	NUI (Network User Identification) and password with call setup.
x25_number	char[20]		X.25 call number, (X.25 B channel only).
user_data	char[20]		User data starting with D (without protocol ID, data length max. 16 char). P with protocol ID, data length max. 12 characters.
max_retries	char	0…	The maximum number of dial retries.
connect_timeout	char		Identifies the whether the modem is waiting for an answer. If expired, there are no more retries.
country_code	char		Identifies the country code.

Tag name: GPRS

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	media name	Identifies the name of the media.
APN	char[32]		GPRS access point name.
operator_id	char		Operator LAI number.
reg_mode	char[5]	1 = AUTO 2 = MANUAL 3 = AUTO_MANUAL	Network registration mode.
number	char[32]		Phone number for GSM data calls (RFU).
bearer	char[32]		GSM Data modulation (RFU).

Tag name: OPTION

Attribute Name	Type	Accepted Values	Description
name_id	char[32]	Media name of another tag in the XML file for which this option applies	Identifies the name of the media.
option	char[256]		List of options.

NET Service Communication File Examples

This section provides XML example files for NET Service communication structures.

TCP/IP Over Ethernet

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="eth-session" priority="1" timeout="10000" retry = "3"
next_media="tcp-layer" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id="tcp-layer" protocol="tcp"
remote_ip="www.verifone.com:80" next_media="eth-layer" />
</SOCKPROTO_LIST>
<ETHLINK_LIST>
<ETHLINK name_id="eth-layer" interface="eth0" usedhcp="1"/>
</ETHLINK_LIST>
 </SETTINGS>
```

TCP/IP Over Wi-Fi

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
```

```
<SESSION_LIST>
<SESSION name_id="wifi-session" priority="1" next_media="tcp-layer"
timeout="10000" retry = "3"/>
</SESSION_LIST>
 <SOCKPROTO_LIST>
SOCKPROTO name_id="tcp-layer" protocol="tcp"
remote_ip="www.google.com:80" next_media="wifi-iface" />
</SOCKPROTO_LIST>
 <WIFI_LIST>
<WIFI name_id="wifi-iface" interface="eth0" usedhcp="yes" suppliconf="/
mnt/flash/etc/config/svcnet/wificfg.conf"/>
</WIFI_LIST>
 </SETTINGS>
```

SSL Over Ethernet

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION name_id="ssl-session" priority="1" next_media="ssl-eth"
timeout="10000" retry = "3"/>
</SESSION_LIST>
 <SSL_LIST>
<SSL name_id="ssl-eth" remote_ip="mmsgto.fr:443"  server_profile="/tmp/"
client_profile="/tmp/" next_media="eth-iface" />
</SSL_LIST>
 <ETHLINK_LIST>
<ETHLINK name_id="eth-iface" interface="eth0" usedhcp="1"/>
</ETHLINK_LIST>
 </SETTINGS>
```

SSL is the tag for an SSL entry used with the following attributes:

- remote_ip: defines the server IP address to connect to.
- server_profile: specifies the path that contains the server CA root.

- client_profile: specifies the path that contains the client CA root and the Private key.

NOTE

- For server_profile, the server CA root name must be CA_Key.pem. For example:

  ```
  /tmp/CA_Key.pem
  ```
- For client_profile, the client CA root name must be CC_Key.pem and the Private Key name is CP_Key.pem. For example:

  ```
  /tmp/CA_Key.pem
  /tmp/CP_Key.pem
  ```
- client_profile must be provided only when SSL mutual authentication is required.

Private Key Protection

When the private key is password protected, the application must use net_sessionStorePassword() to provide its plain text value to NET Service. This API can be called at any time.

```
int net_sessionStorePassword(char *profile, char *password, int gshared);
```

where,

- profile = client_profile
- password = the passphrase that protects the private key in plain text
- gshared is 1 if the passphrase is shared within the group, or 0 if it is private to the application

Overriding the Default PCI - Cipher List

In some cases the application may need to use its own cipher list to connect to a specific server. The application uses the net_sessionCipherList() API to indicate to the NET Service layer which cipher suite applies for a specific profile. This API can be called at any time.

```
o  int net_sessionCipherList(char *profile, char *cipher, int gshared);
```

where,

- profile = server_profile
- cipher = the cipher suite used for a given server profile.
- gshared = 1 if the cipher suite is shared within the group, or 0 if it is private to the application.

Default PCI - Cipher List

An SSL/PCI configuration file restricts the SSL ciphers to only those that are PCI compatible. It allows defining the cipher suite used for SSL sessions if the application is not providing one. The file is located at `/mnt/flash/etc/config/svcnet/cipher_pci.xml`

This file is loaded only when the application is not providing its own cipher suite.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<CIPHER_SUITE_LIST>
<CIPHER_SUITE cipher_id="1" cipher="pciciphersuite" />
</CIPHER_SUITE_LIST>
</SETTINGS>
```

where,

- pciciphersuite = the PCI cipher suite string.
- cipher_id="1" This is reserved for future use, and is always 1.

NOTE

- If the cipher_pci.xml file is not present and the application does not provide one, the default OpenSSL cipher list applies.
- Applications cannot change the PCI cipher suite.

TCP/IP over PPP over PSTN Modem

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-modem" protocol="TCP"
remote_ip="www.verifone.com:80" />
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
<PPPPROTO name_id=" ppp-modem" user="magic" password="isdn" auth="pap"
pppOption="" next_media="modem_uart"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<MODLINK name_id="modem_uart" number="0820903002" dial_type="1" mode="2"
max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1"/>
</MODLINK_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-modem" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
<OPTION name_id="modem_uart" option="dial_no_wait no prefix 8"/>
</OPTION_LIST>
</SETTINGS>
```

TCP/IP over PPP over ISDN Modem

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
```

```
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-modem" protocol="TCP"
remote_ip="www.verifone.com:80" />
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
<PPPPROTO name_id=" ppp-modem" user="magic" password="isdn" auth="pap"
pppOption="" next_media="modem_isdn"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<MODLINK name_id="modem_isdn" number="0820903002" dial_type="1" mode="2"
max_retries="1" max_line_rate="12" connect_timeout="30"
error_correction="1" compression="1"/>
</MODLINK_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-modem" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
<OPTION name_id="modem_uart" option="file /tmp/usrcmdfile_pstn1.txt"/>
</OPTION_LIST>
</SETTINGS>
```

TCP/IP over PPP over GPRS

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
<SESSION_LIST>
<SESSION  name_id="ppp-session" next_media="tcp-layer" priority="1"
timeout="3000" retry="1" />
</SESSION_LIST>
<SOCKPROTO_LIST>
<SOCKPROTO name_id=" tcp-layer " next_media="ppp-gprs" protocol="TCP"
remote_ip="www.verifone.com:80" />
</SOCKPROTO_LIST>
<PPPPROTO_LIST>
<PPPPROTO name_id=" ppp-gprs" user="magic" password="isdn" auth="pap"
pppOption="" next_media="gprs_layer"/>
</PPPPROTO_LIST>
<MODLINK_LIST>
<GPRS_LIST>
<GPRS name_id="gprs_layer" APN="m2minternet"  />
</GPRS_LIST>
<OPTION_LIST>
<OPTION name_id=" ppp-gprs" option="asyncmap 0 noauth noipdefault
usepeerdns lcp-echo-interval 3 lcp-echo-failure 2"/>
</OPTION_LIST>
</SETTINGS>
```

Creating a Communication File Dynamically

Applications may need to create an XML communication file. This section describes how to do that using the NET Service API.

Each communication type (TCP/IP, SSL, Wi-Fi) has a corresponding descriptor structure defined in svc_net.h. This structure writes data in the XML communication file or reads from it. NET Service associates each communication family (Socket, SSL, Wi-Fi) with a fixed communication type value, COM Type, which are also defined in svc_net.h. Descriptors and COM types create and modify the XML files.

Table 14 lists all NET Service communication descriptors and corresponding COM types.

Table 14 NET Service COM Types

Name	Tag	COM Type	Description
sessionDescriptor	SESSION	NETCOM_SESSION	Parameters for SESSION tags.
socketDescriptor	SOCKPROTO	NETCOM_SOCKPROTO	Parameters for socket tags.
sslDescriptor	SSL	NETCOM_SSL	Parameters for SSL tags.
ethernetDescriptor	ETHLINK	NETCOM_ETHLINK	Parameters for Ethernet tags.
wifiDescriptor	WIFI	NETCOM_WIFI	Parameters for Wi-Fi tags.
pppDescriptor	PPPPROTO	NETCOM_PPPPROTO	Parameters for PPP tags.
modemDescriptor	MODLINK	NETCOM_MODLINK	Parameters for PSTN modem tags.
modemIsdnDescriptor	MODISDNLINK	NETCOM_MODISDNLINK	Parameters for ISDN modem tags.
gprsDescriptor	GPRS	NETCOM_GPRS	Parameters for GPRS tags.
gprsDescriptor	OPTION	NETCOM_OPTION	Parameters for Option tags.

- net_sessionSetTag() writes a descriptor to the XML file.

```
int net_sessionSetTag(int comtype, char *nameid, struct netDataTransfer
data);
```

- net_sessionGetTag() reads a descriptor from the XML file.

```
struct netDataTransfer net_sessionGetTag( int comtype, char *nameid );
```

Writing a Communication Descriptor to an XML File

Use the steps in this section to add communication tags to a NET Service communication file. The following example adds a TCP/IP layer. The socketDescriptor structure and NETCOM_SOCKPROTO are required to add the TCP/IP layer. Adding any other tags follows the same path, but replaces socketDescriptor and NETCOM_SOCKPROTO with the desired quantities.

1 Create the data transfer structures:

```
struct socketDescriptor sckDesc;
/** Create a socket descriptor structure */
struct netDataTransfer sckData;
/** Create a general data transfer structure */
memset( &sckDesc, 0, sizeof(struct socketDescriptor));
memset( &sckData, 0, sizeof(struct netDataTransfer));
```

2 Open the file to store the data:

```
int ret = net_sessionOpen("/tmp/com_file.xml");
```

If the path /tmp/com_file.xml does not exist, it is created at the end of the process.

3 Fill the socket descriptor parameters:

```
strncpy( sckDesc.name, "socket", strlen("socket") );
/** value for name_id*/
strncpy( sckDesc.protocol, "tcp", strlen("tcp") );
strncpy( sckDesc.remote_ip, "www.verifone.com:80",
strlen("www.verifone.com:80") );
strncpy( sckDesc. nextMedia, "eth-layer", strlen("eth-layer") );
/* value for next_media */
```

4 Copy the socket descriptor structure in the `netDataTransfer` structure:

```
sckData.data = &sckDesc;
sckDesc.data_count = sizeof(struct socketDescriptor );
```

5 Use net_sessionSetTag() to add `netDataTransfer struct (sckData)` in the context:

```
int net_sessionSetTag(NETCOM_SOCKPROTO, "socket", sckData );
```

The data is stored in the context, but is not saved in the destination file until the next step.

6 Save the file.

```
int ret = net_sessionSave( "/tmp/com_file.xml" );
```

You can use a different filename. The existing file is not modified.

The result /tmp/com_file.xml is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<SETTINGS>
 <SOCKPROTO_LIST>
SOCKPROTO name_id="socket" protocol="tcp" remote_ip="www.verifone.com:80"
next_media="eth-layer" />
</SOCKPROTO_LIST>
 </SETTINGS>
```

Reading a Communication Descriptor from an XML File

Use the steps in this section to read communication tags from a NET Service communication file. We will read a TCP/IP layer as an example. The socketDescriptor structure and NETCOM_SOCKPROTO COM type are required to read a TCP/IP layer. All other tags follow the same path, but replace socketDescriptor and NETCOM_SOCKPROTO with the desired quantities.

Use the following steps to read a communication descriptor from an XML file:

1 Create the data transfer structures:

```
struct socketDescriptor sckDesc;
/** Create a socket descriptor structure */
struct netDataTransfer sckData;
/** Create a general data transfer structure */
memset( &sckDesc, 0, sizeof(struct socketDescriptor));
memset( &sckData, 0, sizeof(struct netDataTransfer));
```

2 Open the file from which to read the data:

```
int ret = net_sessionOpen("/tmp/com_file.xml");
```

NOTE

/tmp/com_file.xml must be an existing file.

3 Fill the socket descriptor parameters:

```
sckData = net_sessionGetTag( NETCOM_SOCKPROTO, "socket" ); /* indicates
the name_id you want to read */
if ( errno == 0)
{/* In case of success errno is 0 */

    If (sckData.data_count == sizeof(struct socketDescriptor) )
    {
        memcpy( &sckDesc, sckData.data, sckData.data_count);
    }
    free(sckData.data ); /* The caller must free the data*/
}
```

NOTE

If errno is 0 (success), the caller must free the `netDataTransfer` structure.

NET Service API

- See NET Service API for API descriptions.
- See NET Service Constants for information on constants.
- See NET Service Structures for information on structures.

Sysmode Application

The following are menus are available when the sysmode application is running:

Information

- Basic info: Displays general information on the unit (S/N, P/N, revision, model, and so on), and low-level software (Linux kernel, U-Boot, SBI, and so on)
- Ports/Options: Displays COM ports states
- Software: Lists installed software
- Memory: Displays flash memory and RAM usage
- Logs: Displays the tamper and installation logs
- Counters: Displays the internal counters (PIN entries, SC insertions, and so on)

Administration

- Date/Time: Configure the system date and time
- Network: Configure and display information on network interfaces
- MMS: Configure the remote MMS server
- Remove: Remove selected or all user bundles.

NOTE

This feature cannot remove OS/system bundles.

Update

- USB Memory: Browse the USB device and install selected software
- Serial: Install update over the serial link (only with the Mx800Downloader application on COM1 only)
- Netloader: Install applications over Ethernet (only with the Mx800Downloader application)
- MMS: Initiate a session with the MagIC Management system to update S/W configuration remotely

Security

- Key loading
  - Load Bank/VRK keys using serial link (COM1 only). Access is restricted with two key loader passwords.
  - Perso Pinpad: Create key pair and build certificate using a smartcard for mutual authentication V/OS-PINpad
- Tamper status: Display security status

- Key list: Display loaded secure keys/scripts (RKL, VSS, DUKPT, and MS), and also displays status on P-Series authentication
- Password Mgr: Change user and key load passwords or expired password
- Verishield tree: List all installed Verifshield keys
- Removal Switch: (ARS management) Displays information about removal switch status and reset removal switch status

Diagnostic

- Display: Performs the display diagnostic test - attempting to display the bitmap on it.
- (VX 520) Keyboard: Displays the matrix with the keys. Pressing a key increases the counter number for that key value. There is no button to return to previous menu. If no keys are pressed in 5 seconds, the diagnostic test automatically exits.
- Card
  - Smart Card: Test performs the smart card device diagnostics
  - Magstripe: Test performs the magnetic stripe device diagnostics
  - Contactless: Test performs the Contactless device diagnostics (if connected)
- Communications
  - Ethernet: Ethernet connection diagnostics (ping test)
  - Serial: Serial loop-back diagnostics. For this test to work correctly, the loop-back cable shown in Figure 21 is required.

NOTE

Insert the loop-back cable into serial port 1 (COM1) with NO console enabled.

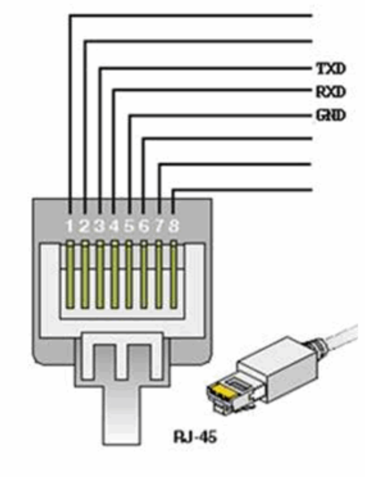


Figure 21 Loop-back Cable Pin Out

  - Modem: Performs the modem diagnostics
  - USB: The USB diagnostics test.
- Printer: Perform the diagnostic test for the printer. It prints the numbers 1234567890123456789 01234, and then a black square.
- Buzzer: Performs the diagnostic test of the buzzer.
- LEDs: Tests the LOGO LED and keyboard LED.
- (P-Series only) PINpad
  - Basic info: Displays general PINpad information (P/N, S/N, PCI code, model)
  - Security:
    - Status: Displays security status (tamper
    - Key list: Displays the PINpad key list
    - Unlock: Detampers the PINpad using the “UNLOCK” smartcard
    - Reset: Removes PINpad keys using the “RESET” smartcard
    - Perso: Generate key pair and certificate using the “AUTH_PP” smartcard. This is required for mutual authentication V/OS-pinpad
  - Diagnostic: Tests all PINpad devices separately

VX 520 Special Features

This section describes OS variants for the VX 520 terminal, including:

NOTE When the Sysmode Application launches, the keyboard buzzer is disabled.

Color LCD

You must make the following changes to the CIB and LCD driver files to support the VX 520 terminal:

- Change the resolution entry to 320*240(landscape)
- The ATM keys are replaced with a 4-way navigation key

Keypad

The following are VX 525 keypad criteria:

- There are 4 function keys under the display.
- Between the second and third keys is a 4-way navigation key with a select/confirm center key.
- There is no alpha key (see Table 15)
- The default password is 166831.

The VX 525 keypad type and keymap includes 4-way navigation keycodes. Table 15 compares the VX 525 keymap to the standard V/OS keymap:

Table 15 VX 525 Keymap

V/OS Standard Key Code	VX 525 Key Code
F1 ('z')	Nav. Up ('Z')
F2 ('{')	Nav. Down ('[')
F3 ('|')	Nav. Left ('\')
F4 ('}')	Nav. Right (']')
Alpha (0x0F)	Select/Confirm ('^')

SVC_INFO_KEYBRD_TYPE() always returns 3.

GPRS Radio

The VX 525 3G terminal has the following criteria for the GPRS radio, SIM, and UART:

- GPRS_Type = GPRS_INTERFACE_TYPE_USB_HOST "2"
- GPRS_ControllerID = GPRS_CONTROLLER_ID_PHS8P "3"
- GPRS_ControllerID = GPRS_CONTROLLER_ID_PHS8P_NOGPS
- DUAL_SIM = DUAL_SIM_WITH_DETECT "1", or
- DUAL_SIM = DUAL_SIM_NO_DETECT "2"
- UART_1_type 0x30 UART_1_INTERFACE_NONE

VX 675 Special Features

This section describes OS variants for the VX 675 terminal, including:

- Comm Engine
- Network Control Panel
- Drivers
- Battery
- PMIC

- Keypad
- Color LCD
- Terminal Types

Comm Engine

The Comm Engine uses the HWCIB.ini file to determine supported communications devices in the terminal. Ensure that the following changes are made to the HWCIB.ini:

- Add the following section to define GPRS, GSM-PPP, and GSM dial support:

```
    [GPRS-88-GPRS]
    commtech= GPRS
    devdrvname = GPCTBGS2
    drvtype = PPP
    signal = 1
    devname =
    startmode =
    shared =
[GPRS-88-PPPGSM]
    commtech = PPP/GSM
    devdrvname = GSCTBGS2
    drvtype = PPP
    signal = 1
    devname =
    startmode =
    shared =
[GPRS-88-GSM]
    commtech= GSM
    devdrvname = GSCTBGS2
    drvtype = DIAL
    devname =
    shared =
```

Network Control Panel

The Network Control Panel (NCP) identifies the user interface such as display and keyboard layout that are available in terminal using the CIB information in APIs. The following are VX 675 terminal specific criteria:

- The battery icon is “Five-value battery icon.” A numeric value does not display in get_battery_value(), and the REMAININGCHARGE parameter only returns 0%–75%.
- The NCP supports the VX 675 keypad with 5-way navigation keys that navigate NCP menu
- The VX 675 has no alpha key, but has a phone type character input.
- The NCP relies on SVC_INFO_KEYBRD_TYPE() to determine the terminal keypad. Return values are:

- 0 = No function keys and no alpha keys
- 1 = Both function and alpha keys
- 2 = Alpha keys are present, but there are no function keys
- 3 = The terminal has a 5-way navigation key

The behavior of following APIs is critical for supporting VX 675 terminal:

SVC_INFO_DEV_TYPE(INFO_PRES_TOUCH)

The VX 675 does not support a touch screen.

Prototype

```
SVC_INFO_DEV_TYPE( INFO_PRES_TOUCH );
```

Return Values

The return value is 0.

SVC_INFO_PRNTR()

This call provides information on the built-in printer.

Prototype

```
SVC_INFO_PRNTR(<output parameter/printer ID>);
```

Parameters

output parameter	Success.
printer ID	Printer ID is

Return Values

Drivers

The following are the DDI modules:

- GPCTBGS2.lib, where:
  - GP = This DDI is for GPRS communication
  - CT = The device manufacturer is Cinterion
  - BGS2 = The device model
- GSCTBGS2.lib, this is the GSM driver, where:

- GS = This DDI is for GSM communication
- CT = the device manufacturer is Cinterion
- BGS2 = The device model

The INI and MTD files have the same name as that of the DDI downloadable module.

Battery

The following are the battery state APIs required by DDI GPRS library to perform power management routines:

SVC_INFO_BAT_REQ()

Use this call when a battery is required by the terminal to operate the GPRS radio and for SIM protection. Use the powermngt_get_sufficient_battery() call to determine battery charge state.

NOTE When the terminal is running on battery, not all the devices can run at the same time (for example, printer, CTLS, GPRS, and so on). This is managed by the application choosing which devices should run, and which should be off.

Prototype

```
SVC_INFO_BAT_REQ(char * one_char);
```

Parameters

Need definitions

char

one_char

Return Values

? 0	The battery is required by the terminal to operate the GPRS radio only.
= 0	The battery is not required for peripherals.
= 1	The battery is required for the printer and GPRS radio.
= 2	The battery is required for the printer only.

PMIC

What does PMIC denote? Power Management ? ?

You must modify the following modules to use the PMIC driver:

- kmod-vf210x_video/bcm5892_lcd.c
- kmod-vf210x_video/bcm5892_lcd_mono.c

- kmod-vf210x_video/bcm5892_bl.c
- kmod-vf210x_radio/radio_hw.c
- kmod-vf210x_led/led.c (if required)
- u-boot/drivers/misc/vf210x_led.c (if required)

The VX 675 uses the V/OS Power Management Functions, but has the following enhancements:

- Client applications can report busy/idle status
- Client applications can create timers to wake up the system

Keypad

The VX 675 terminal does not support a touch screen, but has a phone style keypad layout. There is a 5-way navigation key with a confirm button. Table 16 compares the VX 675 keymap to the V/OS standard keymap.

Table 16 VX 675 Keymap

V/OS Standard Key Code	VX 525 Key Code
F1 ('z')	Nav. Up ('Z')
F2 ('{')	Nav. Down ('[')
F3 ('|')	Nav. Left ('\')
F4 ('}')	Nav. Right (']')
Alpha (0x0F)	Select/Confirm ('^')

Color LCD

You must make the following changes to the CIB and LCD driver files to support the VX 675 terminal:

- Change the resolution entry from 240*320(portrait) to 320*240(landscape)
- Modify kmod-vf210x_video/bcm5892_lcd.c *how?*

Terminal Types

The VX 675 has three base station types:

- FF (full featured) base
    - Charger
    - Dial-up modem
    - USB host
    - Ethernet
    - RS232
- USB base
    - Charger
    - USB host
- BT base
    - Charger

- Dial-up modem
- Ethernet

You must make the following changes to support your base configuration:

- Modem library: Modify this library to include the USB modem.
- Bluetooth connectivity is based on the btDevice Struct Reference.

VX 820 Touch Panel

VX 820 touch capture technology supports Stylus Priority, which means that if a finger and the stylus are placed on the panel, the position input comes from the stylus and the finger is ignored. Stylus Priority mode is enabled by default.

Touch Panel Specifications

	VX
X-axis values	0 … 1023
Y-axis values	0 … 1023
Z-axis values	0 … 1
Sample Rate is always	200
Raw capture resolution (x,y)	1023 x 1024
Display Dimensions (x,y)	2.10 x 2.80
Display resolution (x,y)	240 x 320
Capture resolution (x,y) dpi:	488 x 366
Signature Capture Rate pps	200
Non-signature Capture Rate pps	200

Touch Panel Calibration and Compensation

VX Scaling Calibration

VX terminal touch panel calibration is normally performed at manufacture. VX terminals uses interactive scaling calibration. The scaling function calculates uses a set of 6 constants saved to a flash file. VX scaling calibration is called from V/OS, not directly by an application. The system tracks scaling calibration calls.

VX Terminal Signature Capture Functions

Signature Capture is implemented as a widget. The widget supports the definition of a signing region as well as the ability to track stylus movement. Note that the strokes returned by the Signature Capture widget are scaled to 320x234 (72dpi).

Use the following calls to read and process the raw data for higher resolution images.

NOTE

During signature capture with a stylus connected, the unit switches to stylus-only mode.

```
#include "svc.h"
#include "sig.h"
typedef struct {
long x : 12; // X co-ordinate 0…1880 of touched point
long y : 12; // Y co-ordinate 0…1360 of touched point
long z :   8;   //  Z co-ordinate/pressure 0…63 of touched point
} __attribute__((packed)) xyz_t;
```

This is a representation of the x,y,z value of a single touched point packed into 32-bits – x,y, and z are signed quantities.

Also defines `PENUP`, which has the value:

```
{.x = -1, .y = -1, .z = -1}
```

These functions return -1 on error with the errno set.

SigCapCount()

Returns the number of available signature points.

Prototype

```
int SigCapCount(void);
```

SigCapGet()

Copies up to `maxPoints` of data from the kernel buffer to the user buffer. Returns the number of points actually copied, which is less than `maxPoints` if fewer points are available.

Use `SigCapCount()` to retrieve the number of points captured. Ensure that the buffer is large enough to hold the number of points captured. For example, if `SigCapCount` returns 1000, then the buffer must be (`xyz_t`)*1000 bytes.

Prototype

```
int SigCapGet(void *data, int maxPoints);
```

SigCapBoxApply()

Applies a signature box to the data in the results of `SigCapGet()`. Data outside the box is replaced by `PENUP`. The data is also compressed to remove adjacent duplicate points and adjacent `PENUP`s. The function returns the new number of unique points.

The application supplies the signature box to the function call.

The box coordinates are in screen format, that is:

- x = 0...479 and y = 0...271 for Mx915 terminals
- x = 0...799, y = 0...479 for Mx925 terminals

Signature data is in touch panel coordinates 0…1880 and 0…1360 for best resolution. It is not necessary to call this function before calling `SigCap2Tiff()`.

SigCapGet and SigCapCount

By design, a single `PENUP` (-1,-1,-1) is inserted at the beginning of the buffer as soon as the signature capture starts. This is why a count of 1 is returned and the point is returned as (-1,-1,-1). As long as the pen stays up, no points are inserted. Once the pen rests, new data (x,y,z) are added. A `PENUP` is also always inserted at the end of the buffer for quick the pen movements of just one down point to register. The count is 3 (initial pen up, the down point, and another pen up).

`SigCapGet` and `SigCapCount` always return the total number of raw points collected. The longer the pen is in contact with the touch panel, the higher the count – it does not matter if the points are in the box. Part of this count is residue from the initial pressing of the STROKES button in the test program. Those points are replaced with -1,-1,-1 when `SigCapBoxApply` is called. Positive results of the above function should be passed as the `count` parameter to `SigCapBoxApply`.

`SigCapBoxApply` does two things on the fly:

- Replaces all points outside the box by -1,-1,-1
- Compresses (shuffles toward the beginning of the buffer) the data so that all adjacent duplicates (for example several -1,-1,-1 in a row after applying the box) are replaced by a single copy of the data (in this case a single -1,-1,-1).

This function returns the new total number of points after compression. The original and compressed counts will differ. When displaying compressed data, use the return of `SigCapBoxApply` as the new size unless it is desired to display the raw data.

Prototype

```
int SigCapBoxApply(xyz_t *Sig, int count, SigCapBox_t *box);
```

vf_sig_cature_options()

Sets the rendering options (signature thickness bits) according to the union pointed to by the argument. It controls only how the signature looks on the screen. Note that a separate method in the TIFF function calls control thickening of signature lines in the TIFF file.

To retrieve the call and the `union` definitions:

```
#include vf_sig_capture.h
```

Prototype

```
void vf_sig_cature_options(union renderOptions *ro);
union renderOptions
{
   struct

      int xminus1yminus1 : 1;
      int xyminus1 : 1;
      int xplus1yminus1 : 1;
      int xminus1y : 1;
      int xy : 1;
      int xplus1y : 1;
      int xminus1yplus1 : 1;
      int xyplus1 : 1;
      int xplus1yplus1 : 1;
   } __attribute__((packed));
   int options;
};
```

`union` structure represents a bit map of pixels to plot in addition to the original pixel when a line is rendered using Bresenham's algorithm. The options are specified as either a bit field or an absolute value from 0 ... 511. This is how the bits are mapped to pixels:

0(x-1,y-1)	1(x,y-1)	2(x+1,y-1)
3(x-1,y)	4(x,y)	5(x+1,y)
6(x-1,y+1)	7(x,y+1)	8(x+1,y+1)

A null pointer causes a local static copy of `union` to be cleared; otherwise the `union` pointed to is copied into the local static copy. A cleared `union` is the power-up default and means that only the original point x,y is plotted with no thickening. This also provides runtime-savings as there is no scanning of individual bits when the `union` is clear.

To plot point (x,y) as well as (x+1,y) and (x,y+1) to get a thicker line use:

```
vf_sig_capture_render_options(&(union renderOptions){.options = 0x0b0});
```

which is equivalent to (note the double braces):

```
vf_sig_capture_render_options (&(union renderOptions)
{{.xy=1,.xyplus1=1,.xplus1y=1}});
```

VX Terminal Thermal Printer

VX terminals have a thermal printer. To access the printer, link to the libvfiltp.so shared library. The application uses the printer.h header file to access the library.

Thermal Printer APIs

This section presents the VX thermal printer calls.

Definitions

```
#define PRINTER_OK                     0 /* Successful result */
#define PRINTER_OUT_OF_PAPER          -1 /* No Paper */
#define PRINTER_FIFO_FULL             -2 /* FIFO is full */
#define PRINTER_TIMEOUT               -3 /* timeout */
#define PRINTER_ERROR                 -4 /* Unrecoverable Error */
#define PRINTER_FONT_NOT_AVAILABLE    -5 /* No such font */
#define PRINTER_FIFO_OVERFLOW         -6 /* Data to print exceed FIFO MAX
                                         /* size 61440 bytes Linux fixed
*/

enum printer_e_inverse_color
{
   PRINT_WHITE = 0,
   PRINT_BLACK = 0xFF,
};

typedef struct
{
   int OutOfPaper;
   int JobsInFifo;
   int CurrentJob;
} PrintStatus;
```

Printer_PrintText()

Create a monochrome bitmap of text and send data to the FIFO. This call is non-blocking.

Prototype

```
int Printer_PrintText(const char *buf, int length);
```

Parameters

IN const char *buf	NULL terminated text string.
IN int length	String length: <= strlen(string)

Return Values

Job ID	Successful write to printer FIFO.
PRINTER_OUT_OF_PAPER	No paper.
PRINTER_FIFO_FULL	FIFO full.
PRINTER_FIFO_OVERFLOW	Data to print exceeds FIFO maximum size.

Printer_PrintLine()

Create a monochrome bitmap of pixel lines and write a file to the FIFO. This call is non-blocking.

Prototype

```
int Printer_PrintLine(int line_num , int inverse);
```

Parameters

IN int line_num	Number of pixel lines.
IN char inverse	PRINT_WHITE/PRINT_BLACK

Return Values

Job ID	Successful write to printer FIFO.
PRINTER_OUT_OF_PAPER	No paper.
PRINTER_FIFO_FULL	FIFO full.
PRINTER_FIFO_OVERFLOW	Data to print exceeds FIFO maximum size.

Printer_PrintBitMap()

Send a monochrome bitmap to FIFO. This call is non-blocking.

Prototype

```
int Printer_PrintBitMap(const char *buf , int Length);
```

Parameters

IN const char *buf	Bitmap data
IN int length	Bitmap length

Return Values

Job ID	Successful write to printer FIFO.
PRINTER_OUT_OF_PAPER	No paper.
PRINTER_FIFO_FULL	FIFO full.
PRINTER_FIFO_OVERFLOW	Data to print exceeds FIFO maximum size.
PRINTER_ERROR	Invalid parameter, unrecoverable error, or memory full.

Printer_SetFont()

Choose an available ttf font file. The default printer font file is Vera.ttf.

Prototype

```
int PrinterSetFont(char * FontName);
```

Parameters

IN char *font_name	Font filename; or a NULL terminated text string.

Return Values

PRINTER_OK	Success
PRINTER_ERROR	ttf Font file not found.

Printer_SetFontSize()

Set available ttf font size in mm. Default font size is 4mm.

Prototype

```
int Printer_SetFontSize(int size);
```

Parameters

int size	Font size in millimeters.

Return Values

PRINTER_OK	Success
PRINTER_ERROR	Font size unavailable.

Printer_GetStatus()

Fill the PrinterStatusStr structure.

Prototype

```
int Printer_GetStatus(PrinterStatus *Status);
```

Parameters

IN const PrinterStatus *Status	Pointer to the PrinterStatus structure.

Return Values

PRINTER_OK	Success
PRINTER_ERROR	Error

Printer_WaitReady()

Wait for print job to finish. This is a blocking call.

Prototype

```
int Printer_WaitReady(unsigned int job_id, unsigned int timeout);
```

Parameters

IN unsigned int timeout	Printer timeout in seconds. This is always a positive number.
IN unsigned int job_id	job_id to wait until complete. This is always a positive number: • == 0: Wait for all print jobs in the queue.

Return Values

PRINTER_OK	Successful print.
PRINTER_OUT_OF_PAPER	No paper.
PRINTER_TIMEOUT	Print job in progress.
PRINTER_ERROR	Invalid parameter or unrecoverable error.

Printer_WaitPaper()

Wait until paper is loaded. This is a blocking call.

Prototype

```
int Printer_WaitPaper(unsigned int timeout);
```

Parameters

int timeout	Timeout until paper is loaded.

Return Values

PRINTER_OK	Paper loaded.
PRINTER_OUT_OF_PAPER	No paper.
PRINTER_TIMEOUT	Print job in progress.
PRINTER_ERROR	Invalid parameter or unrecoverable error.

Printer_CancelPrint()

Cancel all print jobs in queue. This is a non-blocking call.

Prototype

```
void Printer_CancelPrint(void);
```

Printer_OutOfPaperEvent()

Call function callback when paper runs out.

```
int Printer_OutOfPaperEvent(void(*callback)(int));
```

Parameters

IN function pointer	Callback

Return Values

PRINTER_OK	Success
PRINTER_ERROR	Unrecoverable error.

APPENDIX H

PAYware Vision

The PAYware Vision shared library is available for the application.

Note the following caveats for PAYware Vision functionality:

- The TERMINALAPPLICATIONNAME and TERMINALAPPLICATIONVERSION use the values in the `*VPAPPNAME` and `*VPAPPVERSION` environment variables, respectively.
- For FTP commands, the directory paths may be embedded in the filename.
- The FTP server is responsible for any '/' to '\' translations, as required.
- When initiated in System mode, the terminal only supports `XFTPGET` with `filetype os` and `app`.
- Down Channel is for PAYware Vision Server-initiated commands; Up Channel is for terminal-initiated commands such as `XTRMCFG` and `sigcap`.
- ApplyOnDate logic must be handled by the application. System mode will not support future versions of ApplyOnDate.

PAYware Vision Library Functions

The following calls support PAYware Vision.

vpInit()

Connects to VP and starts a thread to monitor the connection.

Prototype

```
int vpInit(vp_parm_t *pstVP);

typedef struct {
 int   iOptions;
 char chSeperator;
 void *fnUpData;
 void *fnUpDisconnect;
 int   *fnDownReq;
 void *fnDownFileStatus;
 void *fnTimedOut;
} vp_parm_t;
```

Parameters

`vp_parm_t`	
`*pstVP`	
`iOptions`	Bit-defined options: • 0x0008 – Don't send XTRMCFG • 0x0004 – Enable SSL • 0x0002 – Disable Auto ACKs • 0x0001 – Force download
`chSeparator`	Field separator character.

Environment Variables for XTRMCFG

`*VPAPPNAME`	String for TERMINALAPPLICATIONNAME
`*VPAPPVERSION`	String for TERMINALAPPLICATION
`*VPDOWNPORT`	String for Down Channel port number. Default is 5016.
`*VPSERVERADDRESS`	String for VP server address, in form of an IP address "xxx.xxx.xxx.xxx" or fully qualified domain name "www.vfi-vp.com."
`*VPSERVERPORT`	String for VP server port. Default is 5014.
`*VPDOWNTIMEOUT`	String for Down Channel Idle Timeout, in seconds. Default is 60 seconds.

Return Values

The Up Channel socket fd.

< 0 Error where errno can be set to:

- -ENOLINK
- -ENOENT
- -EBUSY
- -ENOMEN
- -TIMEDOUT

vpParseFields()

Parses the input buffer into the field arrays. STX should not be in the input buffer.

Prototype

```
int vpParseFields(char *pchInBuf, vp_field_t *pszField, char *pszSep);

typedef struct {
 key[];
 value[];
} vp_field_t;
```

Parameters

`*pchInBuf`	Input data stream to parse.
`*pszField`	Pointer to array of `vp_field_t` to store parsed strings.
`*pszSep`	Pointer to field separator a character string.

Return Values

Returns items in the array or error.

The first index usually returns the command in `key` with no value.

vpSendPacket()

Sends the packet to VP server after adding the wrappers. This call is used for both Up and Down Channel messages.

Prototype

```
int vpSendPacket(int iFd, int iOptions, unsigned short ushMsgNum,
char *pchOutBuf, unsigned short ushLength);
```

Parameters

`iFd`	Socket fd.
`iOptions`	Bit-defined options: • 0x0004 – Secure Socket • 0x0002 – Wait for ACK
`ushMsgNum`	• 0 = Message number included. • if > 0 Insert this message number.
`*pchOutBuf`	Pointer to data buffer to send.
`ushLength`	Length of payload data.

Return Values

Bytes sent or error.

The VP library waits for ACK and then retries, if required, before returning.

vpCreateXTRMCFGString()

Creates the XTRMCFG message and returns it to the application.

Prototype

```
void vpCreateXTRMCFGString (char *pchOutBuf, char *pchEcho, int iOptions);
```

Parameters

`*pchOutBuf`	Pointer to data buffer to store XTRMCFG message.
`*pchEcho`	Pointer to data containing the Echo string. If null, the ECHOSTRING field is not sent.
`iOptions`	Bit-defined options, where only bit 2 is checked to determine if the data is secure or not.

vpExit()

Closes the VP connections, frees memory, and exits the application. Applications should call vpExit() to gracefully exit.

Prototype

```
int vpExit(int iFd);
```

Parameters

`iFd`	Socket fd.

vpCloseUp()

Closes the VP Up Channel connection.

Prototype

```
int vpCloseUp(int iFd);
```

Parameters

`iFd`	Socket fd.

vpVersion()

Returns the VP library version string in the form “xx.xx.xx”.

Prototype

```
void vpVersion(char *pchVersion);
```

Parameters

```
*pchVersion
```

Return Values

VP Library version string.

vpCloseDown()

Closes the VP Down Channel.

Prototype

```
Int vpCloseDown(void);
```

Callback Functions

These calls provide PAYware Vision callback functionality.

fnDownReq()

Called after a REQ packet is received on the Down Channel. The application can:

- Allow or disallow this command to proceed
- Perform the request and detail the result

If this callback function is not provided, all REQ commands are allowed to run. The VP library sends an ACK prior to issuing this callback.

System mode may use call this to display download status.

Prototype

```
int fnDownReq(int iDownFd, int iFieldCount, vp_field_t *pszFields);

typedef struct
{
 char  key[MAX_KEY_SIZE];
 char value[MAX_VALUE_SIZE];
} vp_field_t;
```

Parameters

`iDownFd`	Socket fd of Down Channel.
`iFieldCount`	Number of fields in the array.
`*pszFields`	Pointer to array of the field structure.

Return Values

> 0	Allow
-1	Disallow and don't ask again.
-2	Disallow, terminal busy, try again.
-3	Failure: application performed the request.
-4	Failure: invalid file or form type.

fnDownFileStatus()

Called after a file successfully received from the XFTPGET command on the Down Channel is to be acted on. The application must handle the download file and ApplyOnDate logic.

Prototype

```
void fnDownFileStatus(int iStatus, int iFieldCount,
vp_field_t *pszFields);
```

Parameters

`iStatus`	Status of FTP operation: • < 0 Error • = 0 Done • > 0 bytes or percentage of completion
`iFieldCount`	Number of fields in the array.
`*pszFields`	Pointer to array of field structure.

fnUpData()

Called when response data is received on the Up Channel. The VP library sends an ACK prior to issuing this callback.

Prototype

```
int fnUpData(unsigned short ushDataSize, unsigned short ushMsgNum,
char *pchData);
```

Parameters

`ushDataSize`	Contains the size of data in `*pchData`.
`ushMsgNum`	Contains the message number.
`*pchData`	Contains the response data.

fnUpDisconnect()

Called when the Up Channel is disconnected. The VP library closes and cleans up the socket connection.

Prototype

```
void fnUpDisconnect(void);
```

fnTimedOut()

Called when ACKs are not received within the timeout period for the Down Channel Response message.

NOTE

For the Up Channel, the application is responsible for the protocol timeouts.

Prototype

```
void fnTimedOut(void);
```

IPP Legacy Library

This chapter describes the IPP support functions ported from the TXO platforms:

- ipp_getpin()
- ipp_read()
- ipp_abort()
- ipp_diag()
- ipp_mac()
- select_key_mgmnt()
- get_key_mgmnt()
- getKeyStatus()

These functions are the front end to the IPP functions. All limitations of IPP emulation listed in the Chapter 16 apply to this set of functions.

There are several differences between the Omni 7xxx functions and the V/OS terminal functions due to the underlying architecture:

- PIN exhaustion protection implemented differently (see ippWrite(), page 245).
  - In V/OS terminals, a token must be available when starting the PIN session. If no token is available, `ipp_getpin()` returns -2.
  - In the Omni 7xxx, a token must be available when returning the encrypted PIN block. `ipp_read()` returns -5 until one is available.
- In V/OS terminals, some parameter checking is done initially. For example, `ipp_getpin()` returns -3 for invalid minimum PIN length, invalid maximum PIN length, invalid master key number, or invalid working key string. In the Omni 7xxx, those errors are reported by `ipp_read()`.
- In V/OS terminals, the `ipp_read()` return value does not go through all the intermediate states that the Omni 7xxx's does. For example, the values -3, -4, and -5 are never returned by `ipp_read()` on the V/OS terminal.
- There is no IPP trap mechanism on V/OS terminals.
- Interac mode support functions are not implemented on V/OS terminals. Interac support is through the Security Script.

All legacy IPP functions are defined in `ippleg.h`.

Applications must link with the `libvfileg.so` and `libvfisec.so` libraries using `-lvfileg` and `-lvfisec`.

Applications must call `ippOpen()` before using any legacy IPP functions.

ipp_getpin()

Pass the appropriate parameters to define PIN entry. Before issuing this command, `setSecurePINDisplayParameters()` must be used to register the PIN feedback callback function. See SetSecurePINDisplayParameters(), page 246.

Once `ipp_getpin()` executes touch panel data is no longer accessible to the application and is routed directly to the internal software. As each digit of the PIN is entered, an event echoes to the application through the callback function. Illegal keys are ignored. Pressing [CLEAR] aborts the session

Encryption of the PIN is performed internally by the system hardware immediately after the PIN is entered and [ENTER] pressed. Raw PIN data is never available to the application. After issuing this command, use `ipp_read()` to collect the encrypted PIN information.

Prototype

```
result = ipp_getpin(key_type, disp_line, min_pin_len, max_pin_len,
zero_pin_ok, max_time, pan, working_key, master_key);

int result, key_type, max_time, master_key;

char dsp_line, min_pin_len, max_pin_len, zero_pin_ok, *working_key, *pan);
```

Parameters

`key_type`	0 = Master Key Management 1 = DUKPT	
`dsp_line`	Unused on the V/OS terminal.	
`min_pin_len`	Minimum length of the PIN (4 .. `max_pin_len`)	
`max_pin_len`	Maximum length of the PIN (`min_pin_len` ..12)	
`zero_pin_len`	0 = not permitted 1 = permitted	
`max_time`	1...300 max. time, in seconds, for timeout abort	
`pan`	pointer to Personal Account Number 8...19 characters - null terminated	
`working_key`	1DES Mode	16 characters; null terminated for Master Key Session, ignored for DUKP.
	3DES Mode	120 characters; null terminated with GISKE data block for Master Key Session, ignored for DUKPT.

`master_key`	1 character: (0...9) for Master Key Session, (0...2) for DUKPT to select DUKPT engine.

Return Values

• = 0	Success
• -1	PIN Entry Occurring
• -2	Too many PIN entry requests in a short period - Retry later.
• -3	Invalid parameter or IPP communication error.

ipp_read()

Return the encrypted PIN block generated by `ipp_getpin()`.

Prototype

```
result = ipp_read(buffer, size);

intresult;
unsigned int size;
char *buffer;
```

Parameters

`buffer`	User-defined buffer in the application space to store the data.
`size`	The number of bytes to read.

Return Values

• >0	Number of bytes read
• = 0	IPP idle
• -1	Waiting for PIN entry
• -2	PIN entry occurring
• -6	Abort PIN entry by Timeout
• -7	Abort PIN entry by Program
• -8	Abort PIN entry by CLEAR key
• -9	Abort with data
• -10	IPP Communication error
• -11	IPP Command error
• -12	Zero PIN length error

NOTE

Some events occur so quickly that it may not be possible to see the status code.

Return values >0 indicate that the packet is interpreted as described below.

Master Session Key Management buffer contents:

STX	1H	Start of text, value 02h
packet type	2AN	value ‘71'
delimiter	1A	value '.'
Function key	1N	value 0 if function key feature not implemented
PIN length	2N	value 00 or 04 -12
PIN Block Format	2N	value 01 - format prior to encryption
Encrypted PIN Block	16H	encrypted PIN
ETX	1H	End of text, value 03h

Input error packet contents:

STX	1H	Start of text, value 02h
packet type	2AN	value ‘71'
error code	1N	1 = No Master Key 2 = account / working key error 3 = PIN length over max 4 = PIN length under min/non-decimal digit in PIN 6 = Master Key Attributes error 7 = KOF/GISKE Working Key Attributes error
ETX	1H	End of text; 03h value

DUKPT Key Management packet contents:

STX	1H	Start of text; 02h value
packet type	2AN	value ‘73'
packet delimiter	1A	value '.'
00000	5N	value: ‘00000'
KSN	10-20H	Key Serial Number: Hex (Leading F's suppressed) Presented only if PIN entered Length is 0 if no PIN entered
Encrypted PIN	16H	The 64-bit encrypted PIN block represented as 16 hexadecimal digits.
ETX	1H	End of text, value 03h

Input error packet contents:

STX	1H	Start of text, value 02h
packet type	2AN	value ‘73'

error code	1N	1 = No key
		2 = Key serial number error
		3 = PIN length over max
		4 = PIN length under min
		6 = Over 1 million transactions
error ETX	1H	End of text; 03h value

ipp_mac()

Perform a MAC calculation on the user data in Hex ASCII format to build Z66 MAC packets. A Z66 packet with a “6” flag indicates the last binary data packet; Z66 with a “7” flag indicates a first or middle binary data packet.

Prototype

```
return = ipp_mac(master_key, working_key, second_key, message,
message_length, result);
int return;
unsigned int message_length;
char master_key, second_key, *working_key, *message, *result;
```

Parameters

`master_key`	ASCII "0...9" selects master key number. Pass as null (0) to indicate there is no master key.
`working_key`	1DES Mode — 16 characters, null terminated 3DES Mode — 20 characters, null terminated GISKE data block
`second_key`	ASCII "0...9" selects second key number. If this parameter is non-zero, MAC generation starts with `master_key`. If the message is longer than one 32-byte block, the final block is processed using `second_key` for enhanced security. Pass as NULL (0) to indicate there is no second key.
`message`	The message on which to perform the MAC calculation. This parameter in 32-byte blocks, and a running calculation is performed on each block. Maximum number of blocks is 100, therefore the maximum message length is 3200 bytes.
`message_length`	Number of characters in message; max 3200.
`result`	Buffer pointer to contain the MAC value result.

Return Values

1	Success
-1	Master Key Pointer error
-2	Second Key Pointer error
-3	Message length too large
-4	Wrong block Size
-5	Communication error

The MAC value result is placed in the users buffer, formatted as:

Process code field	One character, indicates status • 0 = no error and MAC follows
MAC field	16 character MAC value

ipp_abort()

Abort PIN collection.

Prototype

```
return = ipp_abort(void);
int return;
```

Return Values

= 0	Success
-1	Abort failed

On success the application has control of the keyboard and display.

NOTE 0 returns if PIN collection is not running when this function is called.

ipp_diag()

Execute various IPP diagnostics. The application uses this function to check information on the IPP firmware and perform 1DES test encryptions.

Prototype

```
return = ipp_diag(test_type, result, master_key);
int  return, test_type, master_key;
char *result;
```

Parameters

`test_type`	Type of test to perform:	
	0	ROM checksum test
	1	Serial number test
	2	ROM version number test
	3	Master Session encryption test
	4	DUKPT encryption test
`result`	Pointer to a buffer big enough to hold the results: min 17 bytes for tests 0–2, and 150 bytes for tests 3–4.	
`master_key`	Master Key slot number for performing test 3. DUKPT engine for performing test 4.	

Return Values

- = 0 Success
- -1 Error

The `result` buffer contains the following, as specified in `test_type`:

test_type	result buffer
0	Contains checksum
1	Serial number, if present
2	Version number
3-4	Refer to ipp_read().

All strings are null terminated. For DUKPT and Master Session Encryption tests, the usual prompts for necessary information are hard coded for the purpose of this diagnostic. The values for this information are:

M/S working_key	1234567890123456
DUKPT working_key	DUKPT ENCRYPTION
Card Number	4012345678901

PIN	1234

NOTE

Keys loaded for Master Session and DUKPT must be 1DES compatible.

select_key_mgmnt()

Selects the Key Management method, and allows selection of Master Session and DUKPT methods: either Single DES (1DES) or Triple DES (3DES). Also Secure Messaging and Zero Key support.

Prototype

```
return = select_key_mgmnt(kmm, demf);
int return;
unsigned char kmm, demf;
```

Parameters

`kmm` 1 character binary encoded: 7 6 5 4 3 2 1 0

- - - - - 0 0 0	1DES Master/Session (default)
- - - - - 0 0 1	Mixed Mode (1DES & 3DES GISKE)
- - - - - 0 1 0	3DES GISKE Master/Session
- - - - - 0 1 1	Secure Messaging (not supported on V/OS)
- - - - 0 - - -	DUKPT Engine “0” 1DES (default)
- - - - 1 - - -	DUKPT Engine “0” 3DES
- - - 0 - - - -	Zero Key Support OFF (default)
- - - 1 - - - -	Zero Key Support ON
- - 0 - - - - -	Empty GISKE session key support OFF (default)
- - 1 - - - - -	Empty GISKE session key support ON
- 0 - - - - - -	Do not clear the keys
- 1 - - - - - -	Clear all MS keys and KLK key
0 - - - - - - -	MAC Empty Working Key Support OFF (default)
1 - - - - - - -	MAC Empty Working Key Support ON

`demf` 1 character binary encoded: 7 6 5 4 3 2 1 0

DUKPT Engine “1”	
- - - - - - - 0	1DES DUKPT (default)
- - - - - - - 1	3DES DUKPT
DUKPT Engine “2”	3DES GISKE Master/Session
- - - - - - 0 -	1DES DUKPT (default)
- - - - - - 1 -	3DES DUKPT
X X X X X X - -	Reserved

Return Values

• = 0	Success
• -10	IPP communication error
• -11	IPP command error

get_key_mgmnt()

Check current Key Management settings.

Prototype

```
return = get_key_mgmnt(kmm, demf);
int return;
char *kmm, *demf;
```

Parameters

`kmm` 1 character binary encoded: 7 6 5 4 3 2 1 0

- - - - - 0 0 0	1DES Master/Session (default)
- - - - - 0 0 1	Mixed Mode (1DES & 3DES GISKE)
- - - - - 0 1 0	3DES GISKE Master/Session
- - - - - 0 1 1	Secure Messaging (not supported on V/OS)
- - - - 0 - - -	1DES DUKPT (default)
- - - - 1 - - -	3DES DUKPT
- - - 0 - - - -	Zero Key Support OFF (default)
- - - 1 - - - -	Zero Key Support ON
- - 0 - - - - -	Empty GISKE session key support OFF (default)
- - 1 - - - - -	Empty GISKE session key support ON
- 0 - - - - - -	MS keys or KLK key present
- 1 - - - - - -	All MS keys and KLK key are clear
0 - - - - - - -	MAC Empty Working Key Support OFF (default)
1 - - - - - - -	MAC Empty Working Key Support ON

`demf`	1 character binary encoded: 7 6 5 4 3 2 1 0	
	DUKPT Engine “1”	
	- - - - - - - 0	1DES DUKPT (default)
	- - - - - - - 1	3DES DUKPT
	DUKPT Engine “2”	
	- - - - - - 0 -	1DES DUKPT (default)
	- - - - - - 1 -	3DES DUKPT
	X X X X X X - -	Reserved

Return Values

- = 0 Success
- -10 IPP communication error
- -11 IPP command error

The returned values are in the users variables `kmm` and `demf`.

getKeyStatus()

Return the status of the following keys:

- Master Session
- DUKPT
- RKL
- User

Prototype

```
return = getKeyStatus(int *keyStatus, int length);
int  return;
int *keyStatus;
int length;
```

Parameters

`keyStatus`	Pointer to an array of integers
`length`	Number of integers in the array

A maximum of 23 integers returns. Specifying a length less than 23 returns a subset of key statuses. Each integer value has a status of one or more keys:

Index	Key	Values
0	Master Session 0	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
1	Master Session 1	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
2	Master Session 2	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
3	Master Session 3	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
4	Master Session 4	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded

Index	Key	Values
5	Master Session 5	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
6	Master Session 6	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
7	Master Session 7	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
8	Master Session 8	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
9	Master Session 9	-1 = Error 00 = No Key 01 = Non GISKE key loaded 02 = GISKE key loaded
10	DUKPT 0	-2 = DUKPT Key End-of-Life ERROR -1 = ERROR 00 = No Key 05 = 1DES Key Loaded 06 = 3DES Key Loaded
11	DUKPT 1	-2 = DUKPT Key End-of-Life ERROR -1 = ERROR 00 = No Key 05 = 1DES Key Loaded 06 = 3DES Key Loaded
12	DUKPT 2	-2 = DUKPT Key End-of-Life ERROR -1 = ERROR 00 = No Key 05 = 1DES Key Loaded 06 = 3DES Key Loaded
13	RKL	Bit fields define state of Private/Public RKL keys 0x01: Private Key loaded 0x02: Public Key loaded 1 = Key loaded 0 = No key

Index	Key	Values
14	Root User Keys	Bit fields define which key slots are loaded. 0x001:Slot 0 0x002:Slot 1 0x004:Slot 2 0x008:Slot 3 0x010:Slot 4 0x020:Slot 5 0x040:Slot 6 0x080:Slot 7 0x100:Slot 8 0x200:Slot 9 1 = Key loaded 0 = No key loaded
15–30	USR1–16 User Keys	Bit fields define which key slots are loaded. 0x001:Slot 0 0x002:Slot 1 0x004:Slot 2 0x008:Slot 3 0x010:Slot 4 0x020:Slot 5 0x040:Slot 6 0x080:Slot 7 0x100:Slot 8 0x200:Slot 9 1 = Key loaded 0 = No key loaded

PCI 4 Compliance

V/OS terminals support PCI compliance level 4 (PCI 4). This chapter discusses the changes that impact the people who manufacture, deploy, and repair V/OS PCI 4 devices; the changes that must be made at the application level; and the changes that are recommended at the application level.

NOTE

MX9 series terminals are the first PCI 4-certified devices for Verifone. This chapter focuses on PCI 4 support on MX9 terminals.

Identifying PCI 4 Compliance Devices

The hardware P/N can be displayed by accessing System Mode via **INFORMATION>BASIC SYSTEMS** tab. On OS version 3014xxxx and later, MX9 supports a new API to identify if hardware is PCI 4 compliant:

```
include <platforminfo_api.h>
unsigned int hardware_id = 0;
unsigned long size = 0;
if (platforminfo_get(PI_ADC0_HARDWARE_ID, &hardware_id, sizeof
hardware_id, &size) == PI_OK)
{
If (hardware_id & PI_ADC0_HARDARE_ID_PCI4)
printf(“This is a PCI 4 unit!\n”);
else
printf(“This is not a PCI 4 unit\n”);
}
```

Applications can use this API to adjust their behavior depending on the PCI level supported by the hardware platform.

Sensitive Passwords

PCI 4 imposes the following rules for sensitive password handling:

- Passwords must be a minimum of seven (7) characters.
- Passwords must be unique in the system.

Keyloading passwords and ARS clearing passwords (not used on MX) are all sensitive passwords (System Mode entry passwords are not considered sensitive passwords). The two keyloading passwords have a minimum of seven (7) characters. No two passwords can be set to the same value.

This rule applies to both sensitive passwords and non-sensitive passwords. When passwords are changed, the system compares the new password with all the existing passwords and if the password already exists, the new password is rejected. This forces the user to enter a different password.

CAUTION PCI 4 now prohibits deployment and repair to enter the default value and change the password to the default value reversed, when injecting keys on the MX9 product line.

Operating System Downgrades

Downgrading the OS to a version prior to 30140200 is not allowed. It is a PCI violation to allow a PCI 4-compliant device to run an operating system that is not PCI 4-compliant. Attempting to perform a downgrade may result in a unit that no longer boots. If this occurs, the unit must be re-burned with OS version 30140200, or newer, for it to become operational.

Application Level Changes

PCI-4 requires some changes that must be made to the V/OS that in turn require changes to the application.

24-Hour Reboot

This requirement ensures that all memory in the device is re-initialized at least once every 24 hours. This means that if an existing application is put on a PCI 4 V/OS device, that application will reboot approximately 24 hours from the time it is powered up or from the time of the last reboot.

CAUTION The critical part of this rule is that the reboot may occur in the middle of a transaction or some other critical task, hence, proper timing of reboot should be considered.

How the Application Should Manage the 24-Hour Reboot?

There are 2 options in ensuring that the reboot will not take place in the middle of a transaction or an important task.

1 MX9 System Mode supports the ability to set the time of day to perform the 24-hour reboot. This forces a reboot at a specific time each day. The advantage is if a given installation has a known period where a reboot will not cause an issue, then no change to the application is necessary.

For Example, a retailer that closes at 9:00 in the evening could set the reboot time for 12:00 A.M., and the unit will safely reboot each day. Powering off the unit when the store is closed will work too and helps conserve a little energy.

To set the 24-hour reboot time from System Mode, go to **ADMINISTRATION>DATE/TIME** tab. Enter the time (in 24-hour time) and check the **ENABLE** option.

The 24-hour reboot time can also be set via an application downloaded configuration parameter. The variable name is `*24hrreboot` and the value must be formatted as `hh:mm:ss` (example: `*24hrreboot = 00:00:00` would be 12:00 A.M.).

NOTE The unit must be rebooted after setting the reboot time in order for it to take effect. Be sure that the clock on the unit is set accurately for the location of the device.

2 The application can monitor the time when the forced reboot will occur and it can perform a preemptive reboot at a time that will have no customer impact. The 24-hour reboot mechanism uses the Linux uptime count to determine when to perform the forced reboot. The application can read the same value and know when the reboot will occur and issue a controlled reboot during an idle period where it is unlikely to be noticed.

To read the current system up time, the application can call the following:

```
#include <time.h>

int clock_gettime(CLOCK_MONOTONIC, struct timespec *tp);
```

Unsigned Packages' New Destination

The destination for unsigned packages had been moved to a new location to achieve PCI 4 certification. The original destination for unsigned packages used to be the user's flash directory (i.e. `/mnt/flash/userdata/usr<x>`). For example, `usr1` would have a flash directory on `/mnt/flash/userdata/usr1`.

All users have a symbolic link named 'flash' in their home directory. The symbolic link will point to the user's specific flash directory.

Unsigned Packages Expanded to New Destination

Unsigned packages are expanded to a subdirectory under the user's flash directory. This directory is called `unsigned`. For example, `usr1`'s unsigned packages will be expanded to `/mnt/flash/userdata/usr1/unsigned`. The application will look for the files included in any unsigned package in the new location. The files can then be moved to a more appropriate location or simply accessed from the unsigned directory.

Multi-Application Access to Magnetic Stripe Reader (MSR) Disabled

PCI 4 allows only one application to have access to the MSR at any given time. Prior to this, a legacy feature of MX8 would have allowed multiple applications to open and read the data returned from the MSR. This lets multiple applications access card data independently and then decide which application would process the transaction depending on the card data. This behavior is now unacceptable under PCI 4.

This means that on PCI 4 devices, only one application can open the MSR device (via `msrOpen()` API) at a time. A second call to open the MSR will result in a return value of EBUSY, which signals that the device is open and not available. One suggestion is for the application to create a process that reads the MSR data and then routes it to the appropriate application for further processing.

Application Recommendations

Application developers should review and implement the following recommendations in order to improve overall application security, robustness and integrity.

Hardening

Compiler and linker flags can be set to improve the robustness and integrity of the application. Verifone recommends using the following options:

```
CFLAGS = -O2 -fstack-protector-all -D_FORTIFY_SOURCE=2 -fpie
-fPIE -Wl,-z,relro,-z,now -Wall -Wextra -Wformat=2 -Wswitch-
default -Wswitch-enum.
```

Security Functionality	CFLAGS
Stack protector	`-fstack-protector-all`
Fortify source	`-D_FORTIFY_SOURCE=2`
User space ASLR	`-fpie -fPIE`
GOT protection	`-z,relro,-z,now`
Compile time warning	`-O2 -Wextra -Wformat=2 -Wswitch-default -Wswitch-enum -Wconversion`

Refer to the following information, available on Confluence, for a detailed discussion on GCC hardening features:

- GCC Hardening - CFLAGS
- GCC Hardening - FORTIFY_SOURCE
- GCC Hardening - GOT protection and early binding
- GCC Hardening - Position Independent Executable (PIE)
- GCC Hardening - Stack-protector
- GCC Hardening - Warning flags

CAUTION

Enabling the recommended compiler and linker hardening options may result in new warnings and build errors. Carefully examine and correct errors and warnings and apply fixes as needed. This can be challenging on open source code, hence, smart discretion is advised.

Bugs in applications (such as buffer overruns, illegal access, etc.) will be caught at run time when these protections are enabled. A process with these potentially serious problems will be terminated immediately versus cause sporadic crashes and bad behavior. It is strongly recommended that a complete and thorough SQA cycle be performed on all code that is built with the hardening options.

Random Numbers

Applications should use a PCI-approved Random Number Generator (RNG). This means that applications needing random numbers directly should call either of the following:

- The Vault Random Number Generator API `generate_random()`
- `/dev/urandom` or `/dev/random` where `generate_random()`is called to get the random bytes.

OpenSSL

It is recommended that TLSv1.2 be used where possible. Applications that are using OpenSSL for SSL/TLS directly or as part of SSH should:

- Call `vfiSec_set_vault_rand_method()` in `libvfiVaultapi.so` (see `vficrypt.h`), to set the RNG used by OpenSSL to be the Vault Random Number Generator.
- This impacts the calling application only (other applications will be unaffected since each application gets a separate instance of a library it links against).

Documentation

Refer to Security Policy document, which contains security guidance for users and application developers.

Index

[redacted]

2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA
[redacted]

V/OS

Programmers Manual

Verifone Part Number DOC00501, Revision G