

Verix eVo Volume I: Operating System

Programmers Manual



Verix eVo Volume I: Operating System Programmers Manual
© 2014 VeriFone, Inc.

All rights reserved. No part of the contents of this document may be reproduced or transmitted in any form without the written permission of VeriFone, Inc.

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

VeriFone, the VeriFone logo, VeriCentre, Verix V, Verix eVo, VeriShield, VeriFind, VeriSign, VeriFont, and ZonTalk are registered trademarks of VeriFone. Other brand names or trademarks associated with VeriFone's products and services are trademarks of VeriFone, Inc.

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

Acknowledgments

RealView is a registered trademark of ARM Ltd. For information and ARM documentation, visit: www.arm.com.

VISA is a registered trademark of VISA USA, Inc.

All other brand names and trademarks appearing in this manual are the property of their respective holders.

VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA.
1-800-VeriFone
www.verifone.com



CONTENTS

PREFACE	25
Organization	25
Audience	26
Assumptions About the Reader	26
Conventions and Acronyms	27
Conventions	27
Acronyms	28
Related Documentation	31
 CHAPTER 1	
Programmers	
Quick Reference	
Function Calls	33
Function Call Error Codes	58
DBMON Abort Codes	59
Event Codes Returned by wait_event()	59
Managing Application Data—Effective Use of Memory Space	61
Communications Buffer Space	61
Erase Flash	61
Keypad	62
 CHAPTER 2	
Application	
Programming	
Environment	
Hardware	66
General Principles	66
Virtual Memory	67
Memory Management Unit (MMU)	67
Virtual Memory Map	67
Shared Memory	68
Configuring Number of Shared Regions	68
shm_open()	69
shm_close()	71
memory_access()	72
File System	73
File Locations	73
File Groups	74
Record-Structured Files	74
CONFIG.SYS File	74
Power-fail File Protection	75
Handles	75
Device APIs	75
Devices	77
Console	77
Verix Terminal Manager	78
Customizable Application Launcher	78
Verix Battery Management	79
Battery Conditioner	80
start_battery_conditioner()	81
battery_conditioner_status()	82

Verix Power Management	83
Sleep Mode	86
Hibernate	86
Wakeup Events	86
Power Management System	86
Serial Port Power Control	87
Bluetooth	87
WiFi	87
3G	87
Audio	89
Smart Card	89
ScHibernateNotification().	90
ScWakeNotification().	90
Printer, Battery, and Radio Interaction	91
Application Interface	92
Function Calls	92
BatteryRegs().	93
BatterySN().	93
battery_remain_charge().	94
cs_set_sleep_state().	94
disable_host_power().	95
enable_host_power().	95
get_battery_initialization_status().	96
pause_battery_monitor().	96
get_battery_sts().	97
get_battery_value().	98
get_dock_sts().	99
get_performance_counter().	99
get_performance_frequency().	99
get_powersw_sts().	100
set_battery_value().	100
set_com_pwr().	101
set_com1_pwr().	101
SVC_INFO_BAT_REQ().	102
SVC_SLEEP().	102
SVC_SHUTDOWN().	103
USB_COM2_RESET().	103

CHAPTER 3

File Management

Verix eVo File Systems	105
File Conventions	106
File Storage	106
Filenames	107
File Groups	107
Update Procedure	108
Protecting Files From Removal	108
Default System Files	109
.out Files	109
File Handles	109
Generic Files	109
Variable-Length Record (VLR) Files	110
Compressed Variable-Length Record (CVLR) Files	110

Keyed Files	110
Variable-Length Records.	111
Compressed Variable-Length Records	111
File Access Function Calls	112
Open Files	112
open()	116
read()	120
read_vlr()	124
read_cvlr()	125
write()	126
write_vlr()	129
write_cvlr()	130
lseek(), seek_vlr(), and seek_cvlr()	131
insert(), insert_vlr(), and insert_cvlr()	132
delete(), delete_vlr(), and delete_cvlr()	133
delete_()	134
get_bundle_id()	134
get_file_size()	135
get_file_date()	135
get_protocol_string()	136
SVC_CHECKFILE()	136
_remove()	137
lock()	138
unlock()	138
_rename()	139
get_file_attributes()	140
get_file_max()	140
reset_file_attributes()	141
set_file_attributes()	142
set_file_max()	142
close()	143
Keyed File Reads and Writes	144
getkey()	145
putkey()	146
dir_get_all_attributes()	147
dir_get_attributes()	147
dir_get_file_date()	148
dir_get_file_size()	148
dir_get_file_sz()	149
dir_get_first()	149
dir_get_next()	150
dir_get_sizes()	151
dir_put_file_date()	152
dir_reset_attributes()	153
dir_set_attributes()	154
file_copy()	155
put_file_date()	156
SVC_RAM_SIZE()	156
unzip()	157
Flash File System - Special Case	158
dir_flash_coalesce()	159
dir_flash_coalesce_size()	160

SVC_FLASH_SIZE()	160
Supporting Subdirectory	161
Naming Restrictions	161
Tokenizing Rules and Options	162
Inheritance Rules and Options	163
Subdirectories and Groups	164
mkdir()	167
chdir()	168
rmdir()	169
getcwd()	170

CHAPTER 4

System Configuration File

Environment Variable Descriptions	171
*AKM	172
*APNAME	172
*ARG—Arguments	172
*B—Communication Device Buffers	172
*BCM	172
CHKSUM—Checksum Control	172
*COM1RB	173
*COM2RB	173
*COM3RB	173
*COM8RB	173
*COMBO	173
COM2HW	173
COM3HW	173
*CPAD	173
*DARK	173
*DBMON	174
*DEBUG	174
*DEBUGO	174
*DEBUGT	174
*DEFRAG—Defragment Flash	175
*DIAG	175
*DIRCOM	175
*DOT0	175
*DOT1	175
*ESDMON	175
*ETHSPD	175
*FA—File Authentication	176
*FILE	176
*FKEY	176
*FK_FKEY	176
*FK_HKEY	176
*FK_TITLE	176
*GO—Startup Executable Code File	176
*GKE	177
*GUARD	177
*HEAP	177
*IPPMKI—Internal PIN Pad Communications Parameters	177
*KEYBOARD	178
*LOG	178

*LOGP	178
*MA	178
*MAXSEM	178
*MAXSH	178
*MENUx	178
*MENUxy	178
*MERR	178
*MN	179
*OFF	179
*OFFD	179
*PIPE	179
*POW	179
*PRNT	179
*PROT	179
*PRTFNT	180
*PRTLGO	180
*PW—Password	180
*RKLMKI	180
*SCTOUT	180
*SMDEF	180
*SMDL—VTM Download	180
*SMGIDS	180
*SMPW—VTM Password	181
*SMU2U	181
*SMUI	181
*SOFT_PWR_OFF	181
*SYSCHK	181
*TIME—Set Timers	181
*TMA	181
*TURNOFF	181
*TURNON	181
*TZ	181
*TZRULE	181
*UNZIP2	181
*UNZIP—Decompress.ZIP	182
UNZIP—Determine Decompress Results	182
*USBCLIENT	182
*USBGDEV	182
*USBGPID	182
*USBGVID	182
*USBRESET	182
*VALID—List Groups to Search	182
*VSOPATH	183
*WEAR	183
*Z Series—ZonTalk 2000 Control	183
*ZB	183
*ZDLTYPE	183
*ZDLY	183
*ZH	183
*ZINIT	183
*ZN	184
*ZR	184

*ZRDL	184
*ZRESUME	184
*ZRESULT	184
*ZRESP	184
*ZS	184
*ZSSL	184
*ZSWESC	184
*ZTCP	184
*ZTRY	184
*ZX	184
Device Variables	185
Search/Update CONFIG.SYS	186
get_env()	187
put_env()	188

CHAPTER 5 Multitasking

Verix eVo Application Architecture	189
Tasks	189
Task Startup	189
Task Termination	190
Device Ownership	190
Temporary Device Ownership	191
Sharing the Console	192
grant_owner()	193
revoke_owner()	193
File Sharing and File Locking	194
Task Function Calls	194
_exit()	195
*get_errno_ptr()	195
get_group()	196
get_native_group()	197
get_task_id()	197
get_task_info()	198
run()	199
run_thread()	200
set_errno_ptr()	201
set_group()	201
Application Threads and Semaphores	202
Semaphore Example with Deterministic Scheduling	203
Semaphore Example with User Events	203
Global Semaphore	203
Thread Synchronization	204
sem_init()	205
sem_open()	206
sem_close()	207
sem_post()	208
sem_prop()	209
sem_wait()	209
thread_cancel()	210
thread_join()	211
thread_origin()	212
Pipes	213

Pipe Header Format	214
Pipe Interface	214
Configure the Pipe	214
Pipe Function Calls	215
pipe_connect()	216
pipe_init_char()	218
pipe_init_msg()	218
pipe_init_msgX()	219
pipe_pending()	219
Restart Capability	220
SVC_RESTART()	220

CHAPTER 6 Event Handling

Pre-Sleep Event	223
reg_presleep()	224
Event Function Calls	225
clr_timer()	226
peek_event()	226
post_user_event()	227
read_event()	227
read_evt()	228
read_user_event()	228
set_host_power_event()	228
set_timer()	229
get_com1_event_bit()	230
set_com1_event_bit()	230
set_signal_events()	231
SVC_WAIT()	232
wait_event()	233
wait_evt()	234
Connection Fail-Over Management	235
Failover support for Dual SIM	235
Failover from GPRS to Wifi/BT	235

CHAPTER 7 Console Device

Display	237
Default Font	237
Font Files	237
Big Font Files	238
Gray Scale Character Display	238
get_touchscreen()	239
Keypad	240
Keypress Scan Codes	242
Alpha Key Support	243
CELL_PHONE Mode	245
setAlphaMode()	248
getAlphaMode()	248
alpha_multi_shift()	249
Dual Keypress	250
Hidden Function Support	250
Enter VTM	250
Auto-Repeating Keys	251

Backward Compatibility Mode.	251
Console Ownership	253
Manufacturer Information Block (MIB)	253
Console Function Calls.	254
activate_task()	255
alpha_shift()	256
clreol()	257
clrscr()	257
contrast_down()	258
contrast_up()	258
copy_pixel_block()	259
cs_hard_reset	260
cs_overlay_scan()	260
cs_read_temperature()	261
cs_set_baseline()	261
cs_set_sleep_state()	262
cs_soft_reset()	262
cs_spi_read()	263
cs_spi_cmd_data()	263
cs_spi_cmd_status()	264
cs_spi_write()	264
enable_kbd()	265
delline()	265
disable_hot_key()	265
disable_kbd()	266
disable_key_beeps()	266
display_frame_buffer()	267
draw_line()	269
enable_hot_key()	270
enable_key_beeps()	270
get_backlight_level()	271
get_battery_icon()	271
getcontrast()	272
getfont()	272
getgrid()	273
getinverse()	273
get_kb_backlight()	274
getscrollmode()	274
get_BMP()	275
get_character_size()	276
get_console()	276
get_display_color()	277
get_display_coordinate_mode()	278
get_font()	278
get_font_mode()	279
get_iap_state()	279
get_hot_key_sts()	280
get_host_status()	280
get_ipod_status()	281
gotoxy()	281
iap_control_function()	282
iap_get_keypad_info()	282

inline()	283
inverse_toggle()	283
invert_pixel_block()	284
is_keypad_secure()	285
kbd_pending_count()	285
kbd_pending_test()	286
key_beeps()	286
lock_kbd()	287
putpixelcol()	287
put_BMP()	288
put_BMP_at()	290
put_graphic()	290
resetdisplay()	291
reset_ipod_pins()	291
screen_size()	292
set_hot_key()	292
set_ipod_pins()	293
set_kb_backlight()	293
setcontrast()	294
setfont()	294
setinverse()	295
setscrollmode()	295
set_backlight()	296
set_backlight_level()	296
set_cursor()	297
set_display_coordinate_mode()	297
set_display_color()	298
set_font()	299
set_fkey_panel()	300
set_touchscreen_keymap()	301
sts_kbd_lock()	301
SVC_INFO_DISPLAY()	302
SVC_INFO_DISPLAY_EXT()	302
SVC_INFO_KBD()	303
wherrecur()	303
wherewin()	304
wherewincur()	304
window()	305
write_at()	305
write_pixels()	306

CHAPTER 8

Service Function Calls

get_component_vars()	308
set_combo_mode()	309
SVC_CHK_PASSWORD()	310
SVC_INFO_BAT_REQ()	310
SVC_INFO_COUNTRY()	311
SVC_INFO_COUNTRY_EXT()	311
SVC_INFO_HW_VERS()	312
SVC_INFO_HW_VERS_EXT()	312
SVC_INFO_KEYBRD_TYPE()	313
SVC_INFO_LOTNO()	313

SVC_INFO_LOTNO_EXT()	314
SVC_INFO_MAG()	314
SVC_INFO_MFG_BLK()	315
SVC_INFO_MFG_BLK_EXT()	316
SVC_INFO_MOD_ID()	316
SVC_INFO_MODULE_ID()	317
SVC_INFO_MODELNO()	320
SVC_INFO_MODELNO_EXT()	320
SVC_INFO_MODEM_TYPE()	321
SVC_INFO_MODEM_TYPE_EX()	321
SVC_INFO_OS_HASH()	321
SVC_INFO_OS_HMAC_SHA1()	322
SVC_INFO_OS_HMAC_SHA256()	322
SVC_INFO_PARTNO()	323
SVC_INFO_PARTNO_EXT()	323
SVC_INFO_PIN_PAD()	324
SVC_INFO_PORT_IR()	324
SVC_INFO_PORT_MODEM()	325
SVC_INFO_PORTABLE()	325
SVC_INFO_PRNTR()	326
SVC_INFO_READ_MIB()	326
SVC_INFO_RELEASED_OS()	327
SVC_INFO_RESET()	327
SVC_INFO_SBI_VER()	328
SVC_INFO_SERLNO()	328
SVC_INFO_SERLNO_EXT()	329
SVC_INFO_URT0_TYPE()	329
SVC_INFO_URT1_TYPE()	330
SVC_INFO_URT2_TYPE()	331
SVC_INFO_URT3_TYPE()	331
SVC_LED()	332
SVC_LEDS()	332
SVC_SHUTDOWN()	333
SVC_INFO_LIFETIME()	333
SVC_INFO_CRASH()	334
SVC_INFO_EPROM()	335
SVC_INFO_PTID()	336
SVC_VERSION_INFO()	336
SVC_LRC_CALC()	337
SVC_MEMSUM()	337
SVC_MOD_CK()	338
FIFOs.	340
First-in First-out (FIFO) Buffers.	341
SVC_CHK_FIFO()	342
SVC_CLR_FIFO()	342
SVC_GET_FIFO()	343
SVC_PUT_FIFO()	343
SVC_READ_FIFO()	344
SVC_WRITE_FIFO()	344
CRCs.	345
CRC Function Calls	346
SVC_CRC_CALC()	347

SVC_CRC_CALC_L()	347
SVC_CRC_CCITT_L()	348
SVC_CRC_CCITT_M()	348
SVC_CRC_CRC16_L()	349
SVC_CRC_CRC16_M()	350
SVC_CRC_CRC32_L()	351
Configuration Information Block (CIB)	352
SVC_INFO_CIB_ID()	353
SVC_INFO_CIB_VER()	353
SVC_INFO_DEV()	354
SVC_INFO_DEV_TYPE()	355
SVC_INFO_DUAL_SIM()	356
SVC_INFO_PRESENT()	356
SVC_INFO_USB_BITS()	357
Definitions in SVC.h	358

CHAPTER 9 System Devices

Device Management Function Calls	363
get_name()	364
get_owner()	364
set_owner()	366
set_owner_all()	366
Magnetic Card Reader	367
Hybrid Card Reader	367
No Data Characters on Track 3 MSR	367
Increased Buffer Size	367
Testing the MSR	367
Magnetic Card Reader Function Calls	367
card_magprint_count()	369
card_magprint_data()	369
card_magprint_stat()	370
card_mode()	370
card_pending()	371
card_raw_data()	371
magprt_mode_control()	372
VeriShield Protect (VSP)	373
VSP Encryption	373
Verix eVo Implementation	374
Data Formats	376
Internal OS Calls	377
VTM Menu	378
VSP_Status()	380
VSP_Crypto()	380
VSP_Disable()	381
VSP_Enable()	381
VSP_Encrypt_MSR()	382
VSP_Encrypt_KBD()	382
VSP_Decrypt_PAN()	383
VSP_Decrypt_MSR()	383
VSP_Init()	384
VSP_Passthru()	384
VSP_Reset()	385

VSP_Result().	385
VSP_Status_Ex().	386
VSP_Xeq().	386
Smart Card Reader	387
ICC Socket Locations	387
Smart Card API.	387
PCI PED Requirement	388
Administrative Services.	388
IFD_Get_Capabilities().	390
IFD_Set_Capabilities().	390
IFD_Set_Protocol_Parameters().	391
IFD_Power_ICC().	392
Mechanical Characteristics.	393
IFD_Swallow_ICC().	394
IFD_Eject_IC().	394
IFD_Confiscate_ICC().	395
Communication Services	396
IFD_Transmit_to_ICC().	397
ICC Insertion and Removal.	398
IFD_Is_ICC_Present().	399
IFD_Is_ICC_Absent().	399
Enumeration of the Device Capabilities	400
ICC Interface Management.	402
Protocol Support.	403
Specific Features for the Smart Card API.	405
Smart Card Code Examples	409
Multi-Application Support	410
Common Function Calls	411
Contactless Reader	411
Real-Time Clock	411
Related Clock Function Calls	411
SVC_VALID_DATE().	413
Real-Time Clock Function Calls	413
get_bits_per_second().	414
read_clock().	414
read_RTC().	415
Timer Function Calls	416
read_ticks().	417
Beeper	418
Beeper Function Calls.	418
play_RTTTL().	419
beeper_off().	420
error_tone().	420
normal_tone().	421
sound().	421
Speaker Audio System.	423
Speaker with Volume Control	423
Buzzer Emulation	423
Permanent System Sounds	423
Simultaneous Audio and Video.	423
License and Royalties.	423
Synchronized Video and Audio.	423

Device Driver APIs	424
set_pcm_blk()	425
get_pcm_blk()	426
set_pcm_playback()	427
get_pcm_status()	428
set_pcm_volume()	429
get_pcm_volume()	429
System Sounds and Audio Playback	430
Device Detection	430
Internal Printer	430
Internal Printer Function Calls	430
get_opn_blk()	431
set_opn_blk()	432
get_port_status()	433
reset_port_error()	435
barcode_pending()	436
barcode_raw_data()	436
Special Items	437
<ESC>a<n>;	442
<ESC>b<n>,<m>;	442
<ESC>c	443
<ESC>d	443
<ESC>e<n>;	444
<ESC>f<n>;	444
<ESC>g	445
<ESC>h<n>;	445
<ESC>K<n>;	446
<ESC>i	446
<ESC>s	446
<ESC>F<n>;	446
<ESC>l<s><t>;	447
<ESC>m<c><r1>...<rn>;	449
<ESC>H<hh1>...<hhn>;	453
<ESC>GL<f>,<t>,<w>,<h>;<b1>...<bn>;	454
<ESC>GP<t>[,<m>];	456
<ESC>w<n>;	457
<ESC>CS;	458
Dot Graphics Mode	459
Download Fonts and Logos	461
Support for Paper-out LED	463
SDIO	464
SDIO Device Firmware Download	464
SDIO API	466
reset_ctls()	469
set_event_bit()	470
get_event_bit()	472
get_sd_device_bits()	472
Biometric Module (VX 520 GPRS)	473
Libraries	473
SVC_INFO_BIO()	474
USB Barcode Scanner	475
Configuring D130 as USB-COM	475

Enabled Codes on Heron D130 Device	476
Operating Test	477
USB Keyboard	477
USB to RS-232 Converter	478
USB Device Bits	478
Power	479
USB Keyboard Scan Codes	479
Support for Windows Keys	482
Calculating Make and Break Scan Codes	482
Metrologic Barcode Scanner	483
Configuring the Barcode Scanner	483
Keyboard Driver Data Output	483
Metrologic USB Barcode Scanner API	484
Processing Events	485
Device Bits	485
Power	485
Operating Test	485
USB Device Driver APIs	487
USB Device Driver Events	487
MC5727 CDMA Radio	487
AT and Data Port (COM2)	487
Status and Control Port (COM9)	488
Event Handling	488
APIs	489

CHAPTER 10 **Communications**

The Opn_Blk() Structure	493
Character Mode	493
Character Mode Initialization	493
Simultaneous Communication	
Channels	493
Communication Ports	494
RS-232 Serial Port (COM1 and COM2)	494
Communication Port Flow Control (COM1 and COM3)	494
Selecting Non-Standard Bit Rates (COM1 and COM2)	495
Determining Actual Bit Rate (COM1 and COM2)	495
USB Dongles (COM3 and COM6)	495
RS-232 Serial Port (COM1)	497
Serial Port (COM2)	497
get_battery_value().	500
Modem Port (COM3)	502
VX 675 Full Feature Base and VX 820 DUET Modem Device API	503
Conexant Modems (Banshee and Eisenhower)	503
Silicon Laboratories Modem (Si24xx)	504
Asynchronous	505
Synchronous	505
SDLC Protocol	506
Enhanced SDLC Protocol	507
SDLC Status	507
Country Profiles	508
Modem Recommendations	510
Modem Functions	511

modem_sleep().	513
modem_wake().	513
SVC_ZONTALK().	514
Serial Printer Port (COM4).	517
C Code Applet for COM4 Driver	517
Internal PIN Pad (IPP) Port (COM5).	517
Clear IPP Keys Upon Certificate Tree Removal	517
IPP Function Calls	518
select_pinpad().	519
IPP_power().	519
TerminatePinEntry().	520
PINentryStatus().	520
USB External Serial (COM6).	521
set_usb_multi_device().	523
get_usb_device_bits().	524
USB External Ethernet (ETH1).	525
USB Internal WiFi (WLN1).	525
Firmware Loading	525
USB_WIFI_POWER().	528
WiFi Control and Status Management	529
Sending PIMFOR Requests	530
Receiving PIMFOR Responses and Traps	530
Country/Region Configuration	530
Management Information Block (MIB)	531
9-Bit Serial Interface	531
MDB Physical Interface	532
MDB Dongle Version Number	532
9-Bit API	532
write_9bit().	533
read_9bit().	533
Cinterion PHS8-P Radio Support	534
3G Radio.	534
GPS Receiver.	534
Predator Bluetooth Modem Support	534
Modem Profile Loading	535
BT_Si2434_profile_load().	537
SDLC and V.80 Support	538
SDLC Packet Posting	539
Bluetooth Firmware Update	539
Ethernet Device	539
Dial Device	539
Trident Bluetooth Support	540
Devices	540
APIs	543
set_event_bit().	546
get_event_bit().	547
bt_peek_event().	548
bt_clear_event().	548
bt_enable_events().	549
bt_disable_events().	549
bt_perform_inquiry().	551
bt_bluetooth_configuration().	551

bt_spp(1)_configuration()	552
bt_dun_configuration()	553
bt_obex_configuration()	553
bt_obex_push_file()	553
bt_pan_configuration()	554
bt_sdp_search_services()	555
bt_version_string()	555
bt_mdm_profile_load()	556
General Communication Device Functions	557
download()	558
set_serial_lines()	559
set_fifo_config()	561
get_fifo_config()	562

CHAPTER 11

Security/Crypto Library

Security Services Functions	564
File Encryption Support Functions	564
crypto_read()	566
crypto_write()	567
Crypto Functions	569
AES()	570
DES()	571
GenerateRandom()	572
isAttacked()	572
get_tamper()	573
CheckKeyAreaIntegrity()	573
rsa_calc()	574
SHA1()	575
SHA256()	575
VeriShield Security Script Functions	576
pcPS_GetVSSVersion()	578
iPS_GetScriptStatus()	578
iPS_InstallScript()	579
iPS_ExecuteScript()	580
iPS_UninstallScript()	581
VSS PIN Entry Functions	582
iPS_CancelPIN()	583
iPS_GetPINResponse()	583
iPS_RequestPINEntry()	585
iPS_SelectPINAlgo()	586
iPS_SetPINBypassKey()	587
iPS_SetPINParameter()	588
Key Loading Functions	590
iPS_CheckMasterKey()	591
iPS_DeleteKeys()	592
iPS_LoadMasterClearKey()	593
iPS_LoadMasterEncKey()	594
iPS_LoadSysClearKey()	595
iPS_LoadSysEncKey()	596
get_rkl_krd_cert()	597

APPENDIX A		
Verix Terminal Manager	When to Use VTM	599
	Local and Remote Operations	599
	Verifying Terminal Status	600
	Entering Verix Terminal Manager	600
	Passwords	600
	System Password	601
	File Group Passwords	601
	VTM Menu	601
	Features	603
APPENDIX B		
VeriShield Security Scripts	Verishield Security Script Implementation	605
APPENDIX C		
IPP Key Loading	Data Passthrough	607
	User Interface	607
	Error Codes	609
	Master Key Protection	609
	PCI PED Enhancements	610
	Password Requirements	610
	Changing Passwords Manually	611
	Passwords Shorter than Required	611
	Passwords Longer than Required	612
	Download Password Change	612
	OS Upgrade	613
	Default Password	613
	IPP Key Load	614
APPENDIX D		
IPP Communications Packets	Advanced Programming in IPP	615
	Minor Differences by Packet	615
	Packets	616
	Packet Acknowledgement and Timing	618
	Encryption	618
	Constraints	620
	NAKs	620
	Time Outs	620
	Key Insertion	620
	Entering a PIN	621
	Restrict Speed of the PIN Encryption Operation	621
	IPP7	621
	GISKE	622
	Key Management Switching	622
	Using a Session Key	623
	Rules for Loading the Master Key (MS only)	624
	KLK	625
	3DES	626
	1DES	626
	Master Key Addressing	627
	Clear Text GISKE Key Block Loading Rule	627

Common Packets	628
Packet 01: Interactive Diagnostic Routine	628
Packet 05: Transfer Serial Number.	628
Packet 06: Request PIN Pad Serial Number	629
Packets 09 and 14: Response Packet to Packet 01.	631
Packet 11: PIN Pad Connection Test	635
Packets 7 and 12: Dummy Packets	636
Packet 13: Select Baud Rate	636
Packet 15: Set IPP Key Management Mode.	637
Packet 17: Set IPP Key Management Mode.	639
Packet 18: Check IPP Key Management Mode	645
Packet Z60: Accept and Encrypt PIN (VISA Mode)	649
Packet Z63: Accept and Encrypt PIN–Custom PIN Entry Requirements (VISA Mode)	651
Packet M04: Read Permanent Unit Serial Number	653
MS-Specific Packets	654
Packet 02: Transfer Master Key	654
Packet 04: Check Master Key.	657
MS Packet 08: Select a Master Key	662
MS Packet 71: Transfer PIN Block	663
Packet 07: Dummy Packet	664
DUKPT-Specific Packets.	665
Packet 19: Select a DUKPT Engine	665
Packet 25: Check the DUKPT Engine	666
DUKPT Packet 73: Transfer PIN Block (for Packets Z60 or Z63).	667
DUKPT Packet 90: Load Initial Key Request	668
DUKPT Packet 91: Load Initial Key Response.	670
DUKPT Packet 76: PIN Entry Test Request.	671
DUKPT Packet 71: Transfer PIN Block - (for Packet 76)	671
DUKPT Packets 92 and 93.	673
DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request . 673	
DUKPT Packet 75: DUKPT Accept and Encrypt PIN/Data Authentication Response.	674
Packet 78: DUKPT Accept and Encrypt PIN/Data Authentication Test Request 676	
MAC-Specific Packets	677
MAC Packet Z66: Request MAC	677
MAC Packet Z67: Return MAC.	680
Packet 72: Cancel MAC Session	681
MAC Module Design.	681

APPENDIX E

Account Data Encryption

ADE Key Loading	683
Unique Key Enforcement	683
Cleartext Key Loading via VTM	683
Encrypted Key Loading via VRK.	685
ADE APIs.	685
ade_encrypt().	687
ade_status()	689
ade_active()	690
ADE State Controls	691

Turning ADE On or Off	691
Feature Enablement	691
ADE Status Displays	691
Startup Display	691
Non-transient Visual Indication	691
VTM ADE Status Menu	691
ADE VTM Menus	692
ADE Status Menu	692
ADE Key Load Menu	692
ADE On/Off Menu	692
ADE Menu Flowchart	692

APPENDIX F Special Topics

Smart Card PIN Entry using an External PIN Pad	693
Master-Session Key Management Function Calls and Smart Card PIN Entry	694
Master/Session Functions	695
decrypt_session_data()	696
gen_master_key()	697
gen_session_key()	698
test_master_key()	698
Support for APACS40 Cryptographic Functions	699
Software Block Diagrams	699
Calc_Auth_Parm()	700
Calc_MAC()	700
Create_MAC_Key()	701
Init_MAC()	701
New_Host_Key()	702
Reset_Key()	702
Term_MAC()	703
String Utilities	704
dtoa()	705
ltoa()	706
ultoa()	706
strnlwr()	707
strnupr()	707
SVC_HEX_2_DSP()	708
SVC_DSP_2_HEX()	709
SVC_PACK4()	710
SVC_UNPK4()	711
Counted Strings	712
SVC_CS2AZ()	713
SVC_AZ2CS()	713
SVC_INT2()	714
SVC_2INT()	714
DLLs	715
Removing DLLs	715
unload_DLL()	716
Diagnostic Logging	717
dbprintf Method	717
Memory-Based Logging	717
Log Reset	717

Configurable OpSys Logging	718
dbdump()	719
Network Device Drivers	720
GPRS modems GSM 7.10	720
USB Ethernet	721
WiFi	721
3G	723
Device Ports	723
Device Presence	724
Network Device Status	725
openaux()	726
closeaux()	726
TCP/IP Network Support	727
TCP/IP Module	727
Socket Events	727
Invoking the OS Socket Interface	728
Bypassing the OS Socket Interface	728
IP Stack Control and Configuration	728
Dual SIM Design	729
SIM_SELECT()	730
set_SIM_slot()	731
get_SIM_slot()	731
Feature Enablement (FE)	732
GPS	732
HSPA+	732
feature_license_total()	733
feature_license_expiration()	733
feature_license_is_enabled()	734
feature_license_get_tag()	734
feature_license_get_detail()	735

APPENDIX G

Download Operations

OS Download	738
SVC_ZONTALK_NET()	739
download_net()	741
fpNotify()	743
shRegisterNotifyCB()	744
shSetParam()	745
shRegisterExtendedProcessing()	746
fpshPostProcess()	747
fpshPreProcess()	748
OS Download Precautions	749
Preparing a Terminal to Accept Downloads	750
File Name Extensions and GIDs	752
Error Messages	752
File Authentication	754
File Authentication and Downloads	754
Run-Time Authentication	755
authenticate()	756
file_copy_auth_bit()	757
Support for Compressed Files	758
Format for Compressed Files	758

Application Interface to Decompression Service	759
Verix eVo Support for File Groups	759
Determine UNZIP Results	760
User Interface	760
Error Codes	760
Performance	761
Download Result Messages	761
Back-to-Back Downloads	762
Hardware Requirements	762
Special Considerations	762
Initiating a Back-To-Back Download	765
USB Flash Memory Download	766
USB VTM Download	766
Logical File Transfers	767
CONFIG.SYS Files	767
Date, Time, and Passwords	767
USB Flash Auto-Download	768
Multiple ZIP file Downloads	770
SD Memory Download	771
SD Memory Read	772
SD Memory Write	772
User Mode TCP/IP Download Support	772
Resumable Download	774
Split File Naming Convention	774
Combining Files	775
CONFIG. SYS Variables	775
Maximizing Free Flash Space	776
File Removal Specification Syntax	778
Merging CONFIG.\$\$\$ into CONFIG.SYS	779
Zip Files Inside Zip Files	779
Automatic File Removal	780
SVC_ZONTALK()/download()	780
IP Persistence	781
Setting Up a Protected GID	781
Deleting *GUARD variable	781
Protecting the Application Files	781
VeriCentre Downloads	782
File Protection Application	782

APPENDIX H WWAN

Radio Modem Function Calls	785
set_gsm_break()	786
set_gsm_powersave()	786
set_radio_ctl()	787
get_radio_sts()	788
CDMA	789
Hardware Interface	789
SVC.H Symbols	790
Wireless Module ID EEPROM	790
GPRS	791
VX 680 GPRS	791
VX 520 GPRS	792

GPRS modems GSM 7.10	793
USB EM660 Radio Modem Power Management	793
3G	793
WiFi Module	793
Bluetooth Radio Module	794

APPENDIX I USB Support

USB Flash Drive	795
USB Ethernet (ETH1)	795
USB Ethernet Functions	796
get_enet_status().	797
get_enet_MAC().	797
set_enet_rx_control().	798
USB Client	799
HID Client	799
RS-232 Client	799
USB Client API	799
get_usbd_status().	801
usb_pending_out().	801
USB Host.	802
USB Thermal Printer	802
Font Memory.	802
Logo Memory	802
Printer ID.	802
Firmware Version	802
Print Buffer Management	805

APPENDIX J ASCII Table

Control Characters	807
------------------------------	-----

GLOSSARY	809
--------------------	-----

INDEX	811
-----------------	-----



This programmers manual supports the Development Toolkit (DTK) for the Verix eVo transaction terminals with at least 6 MB memory. This manual:

- Describes the programming environment.
- Provides descriptions of the `CONFIG.SYS` variables.
- Provides Application Programmer Interface (API) descriptions and code examples.
- Provides discussion on system and communication devices.
- Provides descriptions of the security features.
- Describes working with the IPP (integrated PIN pad).
- Provides information on downloading applications into a Verix eVo terminal.

The Verix eVo OS is designed to enhance the current core OS applications to provide application services. Applications are written in the C programming language and run in conjunction with the Verix eVo operating system. The Verix eVo supports an improved communication architecture for the current VX products on Predator and Trident.

This manual also contains information regarding the API with the Verix eVo operating system, and with optional peripheral or internal devices.

NOTE

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

Organization

This guide is organized as follows:

- | | |
|-----------|--|
| Chapter 1 | Provides a quick reference to function calls, error codes, <code>CONFIG.SYS</code> variables, download procedures, and flash erasing instructions for the Verix eVo operating system. |
| Chapter 2 | Presents a high-level overview of the hardware and software environment in which application programs run. |
| Chapter 3 | Describes the Verix eVo file systems, file conventions, default system files, file access function calls, file attribute function calls, keyed file reads and writes, variable-length records and compressed variable-length records, and file directory function calls. |
| Chapter 4 | Describes the default system file, <code>CONFIG.SYS</code> . |
| Chapter 5 | Describes the Verix eVo multiple application architecture. |
| Chapter 6 | Presents an overview of the Verix eVo event-oriented services. |

Chapter 7	Describes the console interface, including the APIs used to control the console.
Chapter 8	Presents the function calls used to retrieve information about the Verix eVo operating system and Verix eVo-based terminal device settings.
Chapter 9	Lists the APIs for Verix eVo-based terminal supported devices.
Chapter 10	Lists the APIs for Verix eVo-based terminal supported communication devices.
Chapter 11	Lists the APIs related to security and the crypto libraries.
Appendix a	Describes the VTM operations, when to use VTM menus and the type of operations.
Appendix b	Discusses the VeriShield Security Script (VSS) concept for creating and customizing the security modules to support different key management schemes.
Appendix c	Describes IPP key loading and memory area of the IPP.
Appendix d	Lists the required packet commands of the IPP for MS or DUKPT operations.
Appendix e	Provides options for account data protection.
Appendix f	Describes support for smart card PIN entry from an external PIN pad, Master/Session key management functions, APACS40 crypto functions and string utilities.
Appendix g	Provides procedures to download applications to the Verix eVo-based terminal.
Appendix h	Lists the radio modem function calls.
Appendix i	Describes support for USB host interface and function calls.
Appendix j	Provides an ASCII table for Verix eVo-based terminal displays.
Glossary	Provides definitions of industry terminology used in this manual.

Audience

This document is of interest to **Application developers** creating applications for use on Verix eVo-based terminals.

Assumptions About the Reader

It is assumed that the reader:

- Understands C programming concepts and terminology.
- Has access to a PC running Windows 2000, Windows XP, or Windows 7.
- Has installed the VVDTK on this machine.
- Has access to Verix eVo-based terminal.

Conventions and Acronyms

This section provides reference to conventions and acronyms used in this manual, and discusses how to access the text files associated with code examples.

Conventions

The following conventions help the reader distinguish between different types of information:

- The `courier` typeface is used for code entries, filenames and extensions, and anything that requires typing at the DOS prompt or from the terminal keypad.
- Text references in [blue](#) are links in online documentation. Click on the text to jump to the topic.

NOTE



Notes point out interesting and useful information.

CAUTION



Cautions point out potential programming problems.

The various conventions used throughout this manual are listed in [Table 1](#).

Table 1 Conventions

Abbreviation	Definition
A	ampere
b	binary
bps	bits per second
dB	decibel
dBm	decibel meter
h	hexadecimal
hr	hours
KB	kilobytes
kbps	kilobits per second
kHz	kilohertz
mA	milliampere
MAX	maximum (value)
MB	megabytes
MHz	megahertz
min	minutes
MIN	minimum (value)
ms	milliseconds
pps	pulse per second
Rx	Receive
s	seconds

Table 1 Conventions

Abbreviation	Definition
Tx	Transmit
V	volts

Acronyms

The acronyms used in this manual are listed in [Table 2](#).

Table 2 Acronyms

Acronym	Definition
2TDEA	Double-length TDEA key
ADE	Account Data Encryption
ANSI	American National Standards Institute
APDU	Application Protocol Data Units
API	Application Programmer Interface
ASCII	American Standard Code For Information Interchange
APACS	Association For Payment Clearing Services: Standards Setting Committee; A Member Of The European Committee For Banking Standards (Ecbs)
ATR	Answer To Reset
BCD	Binary Coded Decimal
BCLib	Biometric Data Conversion Library
BD	Bluetooth Driver
BIOS	Basic Input Output System
BRK	Break
BT	Bluetooth
BWT	Block Waiting Time
CDC	Communications Device Class
CDMA	Code Division Multiple Access
CIB	Configuration Information Block
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
CRLF	Carriage Return Line Feed
CTS	Clear to Send
CVLR	Compressed Variable-length Record
CWT	Character Waiting Time
DDL	Direct Download Utility
DLL	Dynamic Link Library
DSR	Data Send Ready
DTK	Development Toolkit. See <i>Vvdtk</i> .
DTR	Data Terminal Ready
DUKPT	Derived Unique Key Per Transaction
DUN	Dial-Up Networking
EBS	European Banking Standard

Table 2 **Acronyms** (continued)

Acronym	Definition
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMV	Europay Mastercard and Visa
EOF	End-of-file
EPP	External PIN Pad
FE	Feature Enablement
FIFO	First In, First Out
HID	Human Interface Device
ICC	Integrated Circuit Card; Smart Card
IEEE	Institute Of Electrical And Electronics Engineers
IFD	Smart Card Interface Device
IFSC	Information Field Size Card
IFSD	Information Field Size Reader
ILV	Identifier Length Value
IPP	Integrated PIN Pad
ISR	Interrupt Service Routine
KSN	Key Serial Number
LAN	Local Area Network
LCD	Liquid Crystal Display
LRC	Longitudinal Redundancy Check
MAC	Message Authentication Code
MCU	Microcontroller
MDB	Multi Drop Bus
MIB	Manufacturer Information Block
MMU	Memory Management Unit
MPLA	Modem Profile Loading Application
MSAM	Micromodule-Size Security Access Module
MSR	Magnetic Stripe Reader
MUX	Multiplexor
NCP	Network Control Panel
NMI	Nonmaskable Interrupt
OS	Operating System
OTP	One-Time Password
PAN	Personal Area Network
PCI PED	Payment Card Industry PIN Entry Devices
PED	PIN Entry Devices
PIN	Personal Identification Number
PKCS	Public Key Cryptography Standards
PLL	Phased Locked Loop
PMIC	Power Management Integrated Circuit
POS	Point-of-Sale

Table 2 **Acronyms** (continued)

Acronym	Definition
PSCR	Primary Smart Card Reader
PTID	Permanent Terminal Identification Number
PTS	Protocol Type Selection
RAM	Random Access Memory
RFID	Radio Frequency Identification
RFU	Reserved for Future Use
RI	Ring Indicator
ROM	Read-Only Memory
RTC	Real-Time Clock
RTTTL	Ring Tone Text Transfer Language
RTS	Request To Send
SAM	Security Access Module
SBI	Secure Boot Image
SCC	Serial Communication Control
SCC Buffer	Storage Connecting Circuit Buffer
SCL	Simple Communications Library
SCR	Swipe Card Reader
SDIO	Secure Digital Input/Output
SDLC	Synchronous Data Link Control
SDP	Sockets Direct Protocol
SMS	Small Message Service
SOC	System On Chip
SPP	Sequenced Packet Protocol
TCB	Task Control Block
UART	Universal Asynchronous Receiver Transmitter
UI	User Interface
UPT	Unattended Payment Terminal
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VLR	Variable-length Record
VPN	VeriFone Part Number
VSS	VeriShield Secure Script
VTM	VeriFone Terminal Manager, formerly System Mode
VVDTK	Verix V Development Toolkit
WTX	Workstation Technology Extended
WWAN	Wireless Wide Area Network

Related Documentation

To learn more about the Verix eVo OS, refer to the following set of documents:

- *Verix eVo Volume II Operating System and Communication Programmers Guide*, VPN - DOC00302.
- *Verix eVo Volume III: Operating System Programming Tools Reference Guide*, VPN - DOC00303.
- *Verix eVo Porting Guide*, VPN - DOC00305.

Detailed operating information can be found in the Reference Manual for your terminal. For equipment connection information refer to the Reference Manual or Installation Guides.



Programmers 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, and keypad key return values for the Verix eVo operating system.

Function Calls

The functions listed in [Table 3](#) 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, the page number can be clicked to jump to the function description.

Table 3 Function Calls

Function Call	Description	Page
Update CONFIG.SYS Entries with Applications		
<code>int get_env(const char *key, char *buf, int bufsize);</code>	Access CONFIG.SYS entries.	187
<code>int put_env(const char *key, const char *val, int len);</code>	Stores an environment variable in CONFIG.SYS.	188
Miscellaneous Service Calls		
<code>int get_component_vars(int hdl, char *buf, int len);</code>	Returns information about an OS component (typically a driver).	308
<code>int set_combo_mode(int mode);</code>	Sets the module specified by mode. (either in conventional telephone modem or as a TCP/IP adapter).	309
<code>int SVC_CHK_PASSWORD(char *buf);</code>	Compares the counted string in buffer to the password for the current group.	310
<code>int SVC_INFO_BAT_REQ(char *char1);</code>	Returns information from the configuration file (CIB) that indicates if a battery is required.	310
<code>int SVC_INFO_CIB_ID(char *CIBid11, int bufLen);</code>	Returns the part number of the CIB.	311
<code>int SVC_INFO_CIB_VER(char *CIBver5, int bufLen);</code>	Returns the CIB version.	353
<code>int SVC_INFO_COUNTRY(char *buf_12);</code>	Stores 12 bytes of factory-defined country variant data in the caller's buffer.	310
<code>int SVC_INFO_COUNTRY_EXT(char *buf, int len);</code>	Returns the country field of the MIB.	311

Table 3 Function Calls

Function Call	Description	Page
<pre>int SVC_INFO_DEV(int type, char *device); int SVC_INFO_DEV(INFO_PRES_SPEAKER, devname);</pre>	<p>Returns the COM device.</p> <p>On VX 680 3G, this function call returns /dev/pcm in devname when INFO_PRES_SPEAKER is passed.</p>	353
<pre>int SVC_INFO_DEV_TYPE(int type); int SVC_INFO_DEV_TYPE(INFO_PRES_SPEAKER);</pre>	<p>Returns the device type.</p> <p>On VX 680 3G, this function call returns MID_SPEAKER when INFO_PRES_SPEAKER is passed.</p>	355
<pre>int SVC_INFO_DISPLAY(char *buf_6);</pre>	Stores display type and size information in the caller's buffer.	312
<pre>int SVC_INFO_HW_VERS(char *buf_2);</pre>	Stores a 2-byte factory-defined hardware version in the caller's buffer.	312
<pre>int SVC_INFO_HW_VERS_EXT(char *buf, int len);</pre>	Returns the HW version field of the MIB.	312
<pre>int SVC_INFO_KEYBRD_TYPE(void);</pre>	Returns the keyboard style as defined in the CIB.	313
<pre>int SVC_INFO_LOTNO(char *buf_6);</pre>	Stores a 6-byte factory-defined manufacturing lot number in the caller's buffer.	313
<pre>int SVC_INFO_LOTNO_EXT(char *buf_6);</pre>	Returns the lot number field of the MIB.	314
<pre>int SVC_INFO_MFG_BLK(char *buf_30);</pre>	Stores 30 bytes of factory-defined manufacturing data in the caller's buffer.	315
<pre>int SVC_INFO_MFG_BLK_EXT(char *buf, int len);</pre>	Returns the manufacturing block data.	316
<pre>int SVC_INFO_MOD_ID(void);</pre>	Returns a code indicating the type of modem installed.	316
<pre>int SVC_INFO_MODULE_ID(int port);</pre>	Detects the type of radio used such as the WiFi, GPRS, CDMA, and Ethernet.	317
<pre>int SVC_INFO_MODELNO(char *buf_12);</pre>	Stores a 12-byte factory-defined model number in the caller's buffer.	320
<pre>int SVC_INFO_MODELNO_EXT(char *buf, int len);</pre>	Returns the model number field of the MIB.	320
<pre>int SVC_INFO_MODEM_TYPE(void);</pre>	Returns the modem type used by the terminal as defined in the CIB.	321
<pre>int SVC_INFO_MODEM_TYPE_EX(void);</pre>	Returns the modem type used by the terminal	321
<pre>int SVC_INFO_OS_HASH (U8* hashout20, U8* keyin, int keysz);</pre>	Allows the application to compute a checksum for the entire OS.	340
<pre>int SVC_INFO_OS_HMAC_SHA1(char *hashout20, char *keyin, int keyinsz);</pre>	Computes the HMAC SHA hash value for the OS and certificate tree.	322

Table 3 Function Calls

Function Call	Description	Page
<code>int SVC_INFO_OS_HMAC_SHA256(char *hashout32, char *keyin, int keyinsz, int mode);</code>	Computes the HMAC SHA 256 hash value for the OS.	322
<code>int SVC_INFO_PARTNO(char *buf_12);</code>	Stores a 12-byte factory-defined part number in the caller's buffer.	320
<code>int SVC_INFO_PARTNO_EXT(char *buf, int len);</code>	Returns the part number of the MIB.	323
<code>int SVC_INFO_PIN_PAD(char *buf_1);</code>	Stores a 1-byte PIN pad type code in the caller's buffer.	324
<code>int SVC_INFO_PORT_IR(void);</code>	Returns the serial port number for infrared communication (if supported).	324
<code>int SVC_INFO_PORT_MODEM(void);</code>	Returns the serial port number connected to the modem.	325
<code>int SVC_INFO_PORTABLE(char *char1);</code>	Indicates the presence of battery power.	325
<code>int SVC_INFO_PRESENT(void);</code>	Returns a bit map of devices present in a terminal.	356
<code>int SVC_INFO_PRNTR(char *buf_1);</code>	Stores a 1-byte printer type code in the caller's buffer.	326
<code>int SVC_INFO_READ_MIB(char *buf, int len);</code>	Returns the MIB.	326
<code>int SVC_INFO_RELEASED_OS(void);</code>	Returns the OS format.	327
<code>int SVC_INFO_RESET(char *buf_12);</code>	Stores the time of the last terminal reset in the caller's buffer and returns the total number of resets in the terminal's lifetime (since the current OS was loaded).	327
<code>int SVC_INFO_SBI_VER(char *SBIver5, int bufLen);</code>	Returns the version of the SBI.	328
<code>int SVC_INFO_SERLNO(char *buf_11);</code>	Stores an 11-byte factory-set serial number in the caller's buffer.	328
<code>int SVC_INFO_SERLNO_EXT(char *buf, int len);</code>	Returns the serial number field of the MIB.	329
<code>int SVC_INFO_URT0_TYPE(char *char1);</code>	Returns the SOC UART0 (COM1) type, or the number of wires in the interface.	329
<code>int SVC_INFO_URT1_TYPE(char *char1);</code>	Returns the SOC UART1 (COM2) type, or the number of wires in the interface.	330
<code>int SVC_INFO_URT2_TYPE(char *char1);</code>	Returns the SOC UART2 (COM3) type, or the number of wires in the interface.	331
<code>int SVC_INFO_URT3_TYPE(char *char1);</code>	Returns the SOC UART3 (COM7) type, or the number of wires in the interface.	331
<code>int SVC_LED(int id, int mode);</code>	Sets the light-emitting diode specified by ID on or off.	329

Table 3 Function Calls

Function Call	Description	Page
<code>int SVC_LEDS(int mode, void *param);</code>	Reads/writes to the LEDs depending on the selected mode.	332
<code>long SVC_INFO_LIFETIME(void);</code>	Returns the total number of seconds the terminal has been in operation.	313
<code>void SVC_INFO_CRASH(struct info_crash_t *results);</code>	Retrieves diagnostic information about the most recent fatal exception.	311
<code>void SVC_INFO_EPROM(char *buf_9);</code>	Stores a counted string that contains an 8-byte firmware version in the caller's buffer.	312
<code>void SVC_INFO_PTID(char *buf);</code>	Stores a counted string that contains an 8-byte terminal identification number in the caller's buffer.	326
<code>void SVC_VERSION_INFO(char *buf);</code>	Stores a counted string that contains the OS version information.	333
Application Programming Environment		
Shared Memory		
<code>int memory_access (const void *buf, int len);</code>	Tests if a region of memory can be read or written.	72
<code>int shm_close (void *region_address);</code>	Frees up the shared memory region previously obtained by calling <code>shm_open</code> .	71
<code>void *shm_open (const char *name, int flag, int size);</code>	Creates a new shared memory object, or opens an existing one for use by the calling application.	69
Battery Conditioner		
<code>int battery_conditioner_status(void);</code>	Gets the battery conditioner status.	82
<code>int start_battery_conditioner(void);</code>	Initiates the battery conditioner and returns the status.	81
File Management		
File Access Function Calls		
<code>int chdir(const char* name);</code>	Restores the current directory.	168
<code>int close(int hdl);</code>	Closes file when access to the file is no longer needed.	143
<code>int delete(int hdl, unsigned int count);</code> <code>int delete_vlr(int hdl, unsigned int count);</code> <code>int delete_cvlr(int hdl, unsigned int count);</code> <code>int delete_(int hdl, unsigned int count);</code>	Deletes data from a file opened for write access at the location of the file position pointer. Note: <code>delete_()</code> provides an alternate name for the <code>delete</code> function so as not to conflict with the <i>delete</i> keyword in C++.	134
<code>int get_bundle_id(int hdl, char *buf);</code>	Copies the Apple <code>bundle_id</code> string into the given buffer.	134

Table 3 Function Calls

Function Call	Description	Page
<code>int getcwd(char* name, int len);</code>	Obtains full “path” of the working directory.	170
<code>int get_file_date(int hdl, char *yymmddhhmmss);</code>	Returns information about the last update to the file.	135
<code>int get_file_size(int hdl, long *filesize);</code>	Returns information about the file size.	134
<code>int get_protocol_string(int iap_channel, char *buf);</code>	Copies the Apple <code>bundle_id</code> string into the given buffer. This is a V*600 only function.	136
<code>int insert(int hdl, const char *buf,int size);</code> <code>int insert_vlr(int hdl, const char *buf,int size);</code> <code>int insert_cvlr(int hdl, const char *buf,int size);</code>	Inserts data into a file opened for write access at the location of the file position pointer.	132
<code>int lock(int hdl, long reserved1, long reserved2);</code>	Locks the open file associated with handle, preventing it from being accessed through any other handle.	138
<code>int lseek(int hdl, long offset, int origin);</code> <code>int seek_vlr(int hdl, long offset, int origin);</code> <code>int seek_cvlr(int hdl, long offset, int origin);</code>	Sets the file position pointer of an open file to a specified location.	131
<code>int mkdir(const char* name);</code>	Creates a new subdirectory.	167
<code>int open(const char *id, int flags);</code>	Allocates and returns an integer file handle used in all subsequent file operations.	116
<code>int read(int hdl, char *buf, int count);</code> <code>int read_vlr(int hdl, char *buf, int count);</code> <code>int read_cvlr(int hdl, char *buf, int count);</code>	Transfer data from a file opened for reading, to a buffer within the application's data area.	120, 124, 125
<code>int _remove(const char *filename);</code>	Deletes a specified file in the directory.	137
<code>int _rename(const char *oldname, const char *newname);</code>	Points the caller's pointer to a pair of pointers to ASCII pathnames.	139
<code>int rmdir(const char* name);</code>	Removes the newly-created subdirectory.	169
<code>int SVC_CHECKFILE(char *filename);</code>	Calculates the checksum for the specified file.	136
<code>int unlock(int hdl, long reserved1, long reserved2);</code>	Removes a lock set by <code>lock()</code> from the open file associated with handle.	138
<code>int write(int hdl, const char *buf, int count);</code> <code>int write_vlr(int hdl, const char *buf, int count);</code> <code>int write_cvlr(int hdl, const char *buf, int count);</code>	Transfer data from an application's buffer to a file that is open for writing.	126, 129, 130
File Attribute Function Calls		
<code>int get_file_attributes(int hdl);</code>	Returns the attribute byte of an open file.	140
<code>long get_file_max(int hdl);</code>	Returns the maximum file data size set in <code>set_file_max()</code> .	140

Table 3 Function Calls

Function Call	Description	Page
<code>int reset_file_attributes(int hdl, int attributes);</code>	Clears attribute flags for an opened file.	141
<code>int set_file_attributes(int hdl, int attributes);</code>	Sets selected attribute flags for an open file.	142
<code>int set_file_max(int hdl, long maxsize);</code>	Sets the maximum data size allowed for a file.	142
Keyed File Reads and Writes		
<code>int getkey(const char *key, char *buf, int size, char *filename);</code>	Retrieves data associated with a given key value.	145
<code>int putkey(const char *key, const char *buf, int size, char *filename);</code>	Stores the data for a given key.	146
File Directory Function Calls		
<code>int dir_get_all_attributes(const char *file);</code>	Retrieves the file attributes of the requested file.	147
<code>int dir_get_attributes(const char *filename);</code>	Provides access to the file attribute bits that the file system maintains.	147
<code>int dir_get_file_date(const char *filename, char *yyyymmddhhmmss);</code>	Retrieves the file date.	148
<code>long dir_get_file_size(const char *filename);</code>	Returns the size of the file.	148
<code>long dir_get_file_sz(const char *filename);</code>	Returns the number of data bytes in the file.	149
<code>int dir_get_first(char *drive);</code>	Returns a NULL-terminated string containing the name of the first file in the directory (usually CONFIG.SYS).	149
<code>int dir_get_next(char *buf);</code>	Takes the current filename in the specified directory and retrieves the following filenames.	150
<code>int dir_get_sizes(const char *drive, struct fs_size *fs);</code>	Returns general information about the specified directory.	151
<code>int dir_put_file_date(const char *filename, const char *yyyymmddhhss);</code>	Attaches a date to the file.	152
<code>int dir_set_attributes(const char *filename, int attributes);</code>	Turns on one or more of the attribute bits for the specified file.	154
<code>int dir_reset_attributes(const char *filename, int attributes);</code>	Turns off the identified attributes.	153
<code>int file_copy(const char *source, const char *target);</code>	Copies the file named by source to target.	155
<code>int put_file_date(int hdl, const char *buf);</code>	Writes the date format <code>yyyymmddhhmmss</code> to the file defined by the file handle.	156
<code>int SVC_RAM_SIZE(void);</code>	Returns the amount of memory, in kilobytes, installed in the terminal.	156
<code>int unzip(const char *zipfile);</code>	Decompresses a VeriFone zip format file.	157

Table 3 Function Calls

Function Call	Description	Page
Flash File System - Special Case		
<code>int dir_flash_coalesce(void);</code>	Erases all flash file system files tagged for deletion and pushes the current files down, recovering memory space.	159
<code>int dir_flash_coalesce_size(long *size);</code>	Returns the number of bytes to reclaim with a coalesce.	160
<code>int SVC_FLASH_SIZE(void);</code>	Returns the amount of flash memory, in kilobytes, installed in the device.	160
String Utilities		
<code>int dtoa (double d, char *buf, int buf_size, int format, int precision);</code>	Converts a floating point value to a string.	705
<code>char *ltoa(long value, char *buf, int radix);</code>	Converts a number to a string.	706
<code>char *ultoa (unsigned long value, char *buf, int radix);</code>	Converts a number to a string.	706
<code>void SVC_INT2(unsigned int value, char *buf);</code>	Converts a number to a string.	714
<code>unsigned int SVC_2INT(const char *source);</code>	Converts a counted ASCII string containing decimal digits to a binary value.	714
<code>int SVC_CS2AZ(char *zstring, const char *cstring);</code>	Converts counted string, <code>cstring</code> , to a standard C zero-terminated string, <code>zstring</code> .	715
<code>int SVC_AZ2CS(char *cstring, const char *zstring);</code>	Converts a zero-terminated string, <code>zstring</code> , to a counted string, <code>cstring</code> .	713
<code>void strnlwr(char *dest, const char *source, int size);</code>	Converts string to lowercase.	707
<code>void strnupr(char *dest, const char *source, int size);</code>	Converts string to uppercase.	707
<code>void SVC_HEX_2_DSP(const char *hex, char *dsp, int n);</code>	Converts binary data to ASCII text.	708
<code>void SVC_DSP_2_HEX(const char *dsp, char *hex, int n);</code>	Converts ASCII hexadecimal data to binary.	709
<code>int SVC_PACK4(char *dest, const char *source, int size);</code>	Compresses ASCII data.	710
<code>int SVC_UNPK4(char *dest, const char *source, int size);</code>	Decompresses ASCII data compressed by <code>SVC_PACK4()</code> .	711
Event Handling		
Pre-Sleep Event		
<code>int reg_presleep(int bitnum);</code>	Allows tasks to register for a pre-sleep event that will be sent by the OpSys before the OpSys enters sleep mode.	224
Event Function Calls		
<code>int clr_timer(int timer_id);</code>	Cancels a timer before it expires.	226
<code>int get_com1_event_bit(int hdl);</code>	Returns the event bit used for COM1 events.	230

Table 3 Function Calls

Function Call	Description	Page
<code>int set_com1_event_bit(int hdl, long flag);</code>	Sets the event bit to use for COM1 events.	230
<code>int set_host_power_event(long evt);</code>	Defines the event bit to use for notification of changes in power or charging.	228
<code>int set_timer(long msec, long eventmask);</code>	Schedules an event to occur after a specified delay.	229
<code>int SVC_WAIT(unsigned int msec);</code>	Suspends the calling task for a specified amount of time.	232
<code>long peek_event(void);</code>	Examines pending events.	226
<code>post_user_event(int user_task_id, int user_bits);</code>	Allows to post an immediate event from a thread to another thread or task.	227
<code>long read_event(void);</code>	Reads and clears pending events.	227
<code>long read_evt(int needed_events);</code>	Reads and clears pending events. This function is similar to <code>read_event</code> function except that only events listed in the bit mask <code>needed_events</code> will be reported to the caller.	228
<code>int read_user_event(void);</code>	Reads and clears the new “user events” field for the calling task. It also resets the new <code>EVT_USER</code> bit in the calling task’s main event.	228
<code>long wait_event(void);</code>	Waits for an event to occur. See Table 7 for event codes returned by <code>wait_event()</code> .	233
<code>int wait_evt(int needed_events);</code>	Waits for an event to occur. The events that are listed in the bit mask <code>needed_events</code> will only cause the task to awake.	234
<code>int set_signal_events(int hdl, char *signal);</code>	Allows the radio to enable an event to occur when one or more input signal lines changes.	231
Console Device		
<code>int activate_task(int task_id);</code>	Allows the current console owner to pass control of the console to the specified task.	255
<code>int alpha_shift(int c);</code>	Returns the character that follows <code>char</code> in the ALPHA key shift sequence.	256

Table 3 Function Calls

Function Call	Description	Page
<code>int clreol(void);</code>	Clears the display line from the current cursor position to the end of the line, relative to the current window.	257
<code>int clrscr(void);</code>	Clears the current display window and places the cursor in the upper-left corner of the window (column 1, line 1).	257
<code>int contrast_down(void);</code>	Decrements the current contrast setting.	258
<code>int contrast_up(void);</code>	Increments the current contrast setting.	258
<code>int copy_pixel_block(int srcStartX, int srcStartY, int srcEndX, int srcEndY, int dstStartX, int dstStartY, int dstEndX, int dstEndY);</code>	Copies the source block specified to the destination block.	259
<code>int cs_hard_reset(void);</code>	Commands the OS to start the hardware reset input of the keypad processor.	260
<code>int cs_overlay_scan(int scantype);</code>	Allows the application to start an overlay scan.	260
<code>int cs_read_temperature(void);</code>	Returns the temperature used in keypad overlay detection.	261
<code>int cs_set_baseline(int scantype);</code>	Used in factory calibration of the CPESM.	261
<code>int cs_set_sleep_state(int sleep);</code>	Allows the application to put the keypad into a low power sleep mode and wake it back up.	262
<code>int cs_soft_reset(void);</code>	Commands the OS to send a software command to the keypad processor that requests a reset.	262
<code>int cs_spi_cmd_data(char *data);</code>	Allows an application to poll the status of the most recent command sent by this API.	263
<code>int cs_spi_cmd_status(void);</code>	Allows an application to poll the status of the most recent command sent by this API.	264
<code>int cs_spi_read(int encrypted, int command);</code>	Commands the OS to send a read command to the keypad processor.	263
<code>int cs_spi_write(int encrypted, int command, unsigned char *data);</code>	Commands the OS to send a write command to the keypad processor.	264
<code>int enable_kbd(void);</code>	Unlocks the keyboard.	265
<code>int delline(void);</code>	Deletes the display line that contains the cursor and moves all lines below it up one line.	265
<code>int disable_hot_key(void);</code>	Disables the hotkey feature.	265
<code>int disable_kbd(void);</code>	Locks the keyboard.	266

Table 3 Function Calls

Function Call	Description	Page
<code>void disable_key_beeps(void);</code>	Disables beeps when keys are pressed.	266
<code>int display_frame_buffer(int x, int y, int w, int h, short * buffer);</code>	Displays the top-leftmost pixel at display location.	267
<code>int draw_line(int startX, int startY, int endX, int endY, int width, int color);</code>	Draws a line of specified width and color from the start point to the end point.	269
<code>void enable_hot_key(void);</code>	Re-enables the hotkey feature after it is disabled by <code>disable_hot_key()</code> .	270
<code>int enable_key_beeps(void);</code>	Enables audible key beeps on a keypress.	270
<code>int get_backlight_level(void);</code>	Returns the backlight level.	271
<code>int get_battery_icon(char* buff3);</code>	Displays the battery icon.	271
<code>int get_BMP(int startX, int startY, int endX, int endY, char *buf, int color);</code>	Creates a BMP-formatted string from a portion of the display specified by the coordinates and color depth, and places the string in the specified buffer.	275
<code>int get_character_size(int* pixelRows, int* pixelColumns);</code>	Stores the current character size as the number pixel rows and columns.	276
<code>int get_console(int clear_keys);</code>	Returns the handle for the console if the current task owns the console.	276
<code>int get_display_color(int type);</code>	Returns the current value for the color type specified.	277
<code>int get_display_coordinate_mode(void);</code>	Returns the current positioning mode.	278
<code>int get_font(char *font_name);</code>	Returns the filename of the current font or the string <code>DEFAULT</code> if the default font is in use.	272
<code>int get_font_mode(void);</code>	Returns the mode of the current font setting.	279
<code>int get_iap_state(int hdl);</code>	Returns the IAP state machine state.	279
<code>int get_ipod_status(int hdl);</code>	Returns the state of the IPOD host.	281
<code>void getfont(char *font);</code>	Returns the display font.	272
<code>int getcontrast(void);</code>	Returns the current display contrast setting.	272
<code>int getgrid(void);</code>	Returns the current grid setting.	273
<code>long get_hot_key_sts(void);</code>	Retrieves hot key status.	280
<code>int getinverse(void);</code>	Returns the current inverse video setting.	273
<code>int get_host_status(void);</code>	Returns the connection and battery state of the host device.	280

Table 3 Function Calls

Function Call	Description	Page
<code>int get_kb_backlight(void);</code>	Retrieves the current keyboard backlight value.	274
<code>int get_touchscreen(int *x, int *y);</code>	Returns the coordinates of the touch in pixels.	239
<code>int getscrollmode(void);</code>	Returns the current scroll mode setting.	295
<code>int gotoxy(int x, int y);</code>	Positions the cursor at the screen relative to the character position specified.	281
<code>int iap_control_function(int hdl, int func);</code>	Allows the application to control the keypad state.	282
<code>int iap_get_keypad_info(int hdl, char *buf);</code>	Copies the keypad status to the given buffer.	282
<code>int insline(void);</code>	Inserts a blank line following the line containing the cursor.	283
<code>int is_keypad_secure(void);</code>	Allows the application to check whether the OS considers the keypad firmware to be authentic.	285
<code>int invert_pixel_block(int startX, int startY, int endX, int endY);</code>	Inverts the colors within the specified pixel block.	284
<code>int kbd_pending_count(void);</code>	Returns the number of keystrokes available for reading.	285
<code>int kbd_pending_test(int t);</code>	Tests if key t is in the keyboard buffer.	286
<code>int key_beeps(int flag);</code>	Turns on beeps when keys are pressed.	286
<code>int lock_kbd(int flag);</code>	Locks/unlocks the keyboard.	287
<code>int put_BMP(char *file);</code>	Displays small, icon-sized graphics used as key labels or pushbuttons for the touch panel.	288
<code>int put_BMP_at(int x, int y, char *file);</code>	Writes the bit mapped image found in the file to the current window starting at the defined x-y coordinates.	290
<code>int put_graphic(const char *buf, int len, int x1, int y1, int x2, int y2);</code>	Displays graphic images (for example, icons) within a specified area.	290
<code>int putpixelcol(char *buf, int len);</code>	Displays graphic images on a byte-by-byte basis.	287
<code>int resetdisplay(const char *font, int grid_id);</code>	Sets the font.	291
<code>int reset_ipod_pins(int hdl, int mask);</code>	Sets the state to 0 for the pins defined.	291
<code>int screen_size(char *buf);</code>	Stores the screen size in *buf.	292
<code>int set_backlight(int mode);</code>	Turns the backlight on/off .	296

Table 3 Function Calls

Function Call	Description	Page
<code>int set_backlight_level(int level);</code>	Sets the backlight brightness level	296
<code>int set_cursor(int flag);</code>	Turns the cursor on and off.	297
<code>int set_display_color(int type, int value);</code>	Sets the type to the specified color for all subsequent characters.	298
<code>int set_display_coordinate_mode(int setting);</code>	Sets the positioning mode as specified.	297
<code>int set_font(const char *font_name);</code>	Sets the font to the specified font file.	294
<code>int set_hot_key(int keycode);</code>	Defines a key to be used as a hot key.	292
<code>int set_ipod_pins(int hdl, int mask);</code>	Sets the state to 1 for the pins defined.	293
<code>int set_kb_backlight(int value);</code>	Sets the keyboard backlight to a level between 0 and 100.	293
<code>int setcontrast(int value);</code>	Sets the display contrast level to the specified value.	294
<code>int setfont(const char *font);</code>	Sets the display font.	294
<code>int setscrollmode(int mode);</code>	Sets the scroll mode.	295
<code>int sts_kbd_lock(void);</code>	Returns the lock status of the keyboard.	301
<code>int SVC_INFO_DISPLAY(char *stuff_6x);</code>	Stores display type and size information in the caller's buffer.	302
<code>int SVC_INFO_DISPLAY_EXT(displayInfo_t dispInfo);</code>	Retrieves information about the display module.	302
<code>int SVC_INFO_KBD(char *stuff_1x);</code>	Fills the caller's buffer with the two-byte keyboard type from the manufacturing block.	303
<code>int wheretur(int *x, int *y);</code>	Returns the current cursor position relative to the physical display not the current window.	303
<code>int wherewin(int *x1, int *y1, int *x2, int *y2);</code>	Returns the current display window coordinates into the four integer variables.	304
<code>int wherewincur(int *x, int *y);</code>	Returns the current cursor position relative to the current window <i>not</i> the physical display.	304
<code>int window(int x1, int y1, int x2, int y2);</code>	Defines a logical window within the physical display.	305
<code>int write_at(char *buf, int len, int x, int y);</code>	Similar to <code>write()</code> , except that the cursor is positioned prior to writing the data in the current font.	305
<code>int write_pixels(int startX, int startY, int endX, int endY, int color);</code>	Fills the specified pixel block with the specified color.	306
<code>int getinverse(void);</code>	Returns the current video setting.	273

Table 3 Function Calls

Function Call	Description	Page
<code>int inverse_toggle(void);</code>	Toggles the current inverse video setting. Equivalent to <code>setinverse(3)</code> .	283
<code>int setinverse(int value);</code>	Selects the inverse video setting based on the two LSBs of <i>value</i> .	295
<code>int setAlphaMode(classic);</code>	On VX 820, switches the keyboard operation to classic mode.	248
<code>int setAlphaMode(cell_phone);</code>	On VX 820, switches the keyboard operation to cell phone mode.	248
<code>int getAlphaMode(void);</code>	On VX 820, returns the current mode of the keyboard.	248
<code>int alpha_multi_shift(int key, int *shift);</code>	On VX 820, switches the keyboard operation to classic mode (default) or cell phone mode.	249
System Devices		
User APIs		
<code>int VSP_Crypto();</code>	Internal Use Only, DO NOT USE.	380
<code>int VSP_Decrypt_MSR(char *MSRe, char *MSRc);</code>	Decrypts the encrypted MSR image <code>MSRe[]</code> , and returns the results in <code>MSRc[]</code> .	383
<code>int VSP_Decrypt_PAN(char *PANE, char *PANc);</code>	Decrypts the encrypted PAN in <code>PANE[]</code> , and returns the results in <code>PANc[]</code> .	383
<code>int VSP_Disable(char *st);</code>	Disables VSP encryption, and returns VSP status in <code>st[]</code> .	381
<code>int VSP_Enable(char *st);</code>	Enables VSP encryption, and returns VSP status in <code>st[]</code> .	381
<code>int VSP_Encrypt_KBD(char *PANc, char *EXPc, char *PANE, char *EXPe);</code>	Encrypts the cleartext PAN in <code>PANc[]</code> and expiration date in <code>EXPc[]</code> , and returns the results in <code>PANE[]</code> and <code>EXPe[]</code> , respectively.	382
<code>int VSP_Encrypt_MSR(char *MSRc, char *MSRe);</code>	Encrypts the cleartext MSR image in <code>MSRc[]</code> , and returns the results in <code>MSRe[]</code> .	382
<code>int VSP_Init();</code>	Internal Use Only, DO NOT USE.	384
<code>int VSP_Passthru(char *in, int iLen, char *out, int oLe);</code>	Allows a raw form of the VSP commands to be used.	384
<code>int VSP_Reset();</code>	Internal Use Only, DO NOT USE.	385
<code>int VSP_Result(void);</code>	Determines if a previously executed VSP function is completed or still running.	385
<code>int VSP_Xeq();</code>	Internal Use Only, DO NOT USE.	386
<code>int VSP_Status(char *st);</code>	Returns VSP status in <code>st[]</code> .	380
<code>int VSP_Status_Ex(char *xst);</code>	Returns VSP extended status in <code>xst[]</code> .	386

Table 3 Function Calls

Function Call	Description	Page
<i>Device Management Function Calls</i>		
<code>long get_event_bit(int hdl);</code>	Determines the event generated by the console device.	472
<code>int barcode_pending(void);</code>	Returns the number of samples for the last scan attempt.	436
<code>int barcode_raw_data(char *buf);</code>	Copies the raw barcode buffer to buffer and returns the count.	436
<code>int get_fifo_config(int hdl, char *buf);</code>	Retrieves current FIFO configuration.	562
<code>int get_name(int hdl, char *name_20);</code>	Retrieves the device name associated with <code>handle</code> .	364
<code>int get_opn_blk(int hdl, Opn_Blks *ob);</code>	Copies the current <code>Opn_Blks</code> structure into the caller's <code>ob</code> structure.	431
<code>int get_owner(const char *id, int *task_id);</code>	Retrieves owning task and handle for a device or pipe.	364
<code>int get_port_status(int hdl, char *buf);</code>	Copies current port status information to caller's 4-byte buffer.	433
<code>unsigned long get_sd_device_bits(void);</code>	Determines if an CTLS card is present.	472
<code>int reset_ctls(void);</code>	Reboots the contactless device micro-controller.	469
<code>int set_event_bit(int hdl, long flag);</code>	Allows the application to select one of the unused event bits and assigns it to the console device.	470
<code>int set_fifo_config(int hdl, const char *buf);</code>	Sets transmit FIFO length.	561
<code>int set_opn_blk (int hdl, Opn_Blks *ob);</code>	Configures the port using the parameters specified in the provided open block structure.	432
<code>int set_owner(int hdl, int task);</code>	Transfers ownership of an open device to another task.	366
<code>int set_owner_all(int hdl);</code>	Allows all of the owner's siblings to use the device for read, write, status, control, or close.	366
<i>Magnetic Card Reader</i>		
<code>int card_magprint_count(void);</code>	Returns the number of interrupts since the last RESTART.	369
<code>int card_magprint_data(char *buf);</code>	Copies the magnetic card reader buffer to the buffer.	369
<code>int card_magprint_stat(char *buf);</code>	Returns the status of the magnetic card reader.	370
<code>int card_mode(int mode);</code>	Sets the magnetic card reader mode flags.	370

Table 3 Function Calls

Function Call	Description	Page
<code>int card_pending(void);</code>	Determines if there is unread data in the card reader buffer.	371
<code>int card_raw_data(char *buf);</code>	Returns the unformatted raw track data for the magnetic card reader.	371
<code>int magprt_mode_control(int mode);</code>	Sets the magprt flag for the magnetic card reader mode flags.	372
<code>int SVC_INFO_MAG(char *buf_1);</code>	Stores a 1-byte magnetic card reader type code in the caller's buffer.	314
<i>Clock and Timer Function Calls</i>		
<code>void date2days(const char *yyyymmdd, long *days);</code>	Sets *days to the number of days elapsed from January 1, 1980, to the specified date.	411
<code>void datetime2seconds(const char *yyyymmddhhmmss, long *secs);</code>	Converts a date/time string to the number of seconds since January 1, 1980 (that is, midnight Dec. 31, 1979).	411
<code>int days2date(long *days, char *yyyymmdd);</code>	Determines the date that is the given number of days past January 1, 1980.	411
<code>int get_bits_per_second(int hdl);</code>	Returns the actual bits per second based on the clock rate and the baud rate constant for the processor.	414
<code>unsigned long read_ticks(void);</code>	Returns the number of clock ticks elapsed since the terminal was powered up or reset.	417
<code>int read_clock(char *yyyymmddhhmmssw);</code>	Stores the current time and date in the caller-provided buffer.	414
<code>int read_RTC(int hdl, char *yyyymmddhhmmssw);</code>	Returns the current time and date.	415
<code>int seconds2datetime (const unsigned long *seconds, char *yyyymmddhhmmss);</code>	Converts the number of seconds since January 1, 1980 (that is, midnight Dec. 31, 1979) to a date/time string.	414
<code>void secs2time (const long *secs, char *hhmmss);</code>	Converts the number of seconds to a time string.	411
<code>int SVC_VALID_DATE(const char *yyyymmddhhmmss);</code>	Verifies that its argument represents a valid date and time.	412
<code>void time2secs (const char *hhmmss, long *secs);</code>	Converts the time into a seconds string.	411
<i>Beeper</i>		
<code>void beeper_off(void);</code>	Squelches the beeper.	420
<code>int close(int hdl);</code>	Releases the handle associated with the beeper.	420
<code>void error_tone(void);</code>	Produces a 100-ms tone at 880 Hz.	420

Table 3 Function Calls

Function Call	Description	Page
<code>void normal_tone(void);</code>	Produces a 50-ms tone at 1245 Hz.	421
<code>void play_RTTTL(char *music);</code>	Invokes the RTTTL interpreter and it returns allowing the calling application to continue with the other tasks. Meantime, the RTTTL interpreter running as a separate thread, will play the tune.	419
<code>int sound(int note, int milliseconds);</code>	Causes the beeper to generate one of the 96 standard tones at a specified time.	421
Device Drivers		
<code>int set_pcm_blk(int hdl, struct *pcm_blk);</code>	Sets the audio sample rate and the format of the samples.	425
<code>int get_pcm_blk(int hdl, struct *pcm_blk);</code>	Reads the current <code>pcm_blk</code> set by <code>set_pcm_blk()</code> .	426
<code>int set_pcm_playback(int hdl, int action);</code>	Allows the application to control the audio playback.	427
<code>int get_pcm_status(int hdl, struct *pcm_status);</code>	Returns the status of the <code>/dev/pcm</code> audio FIFO.	428
<code>int set_pcm_volume(int hdl, int vol);</code>	Sets the gain of the audio amplifier chip.	429
<code>int get_pcm_volume(int hdl, int &vol);</code>	Returns the current volume setting.	429
Communication Devices		
<code>int read_9bit(int com, short *buf, int count);</code>	Accepts 8-bit internal protocol data from the MDB Dongle and returns the data to the application in the buffer.	533
<code>int write_9bit(int com, short *buf, int count);</code>	Takes words in the buffer and sends them to the MDB Dongle for 9-bit conversion.	533
<code>int bt_peek_event(int hdl, DWord_t *event_mask);</code>	Allows applications to determine the cause or causes of Bluetooth events.	548
<code>int bt_clear_event(int hdl, DWord_t event_mask);</code>	Allows applications to clear the Bluetooth events.	548
<code>int bt_enable_events(int hdl, long events);</code>	Allows the owner of each Bluetooth device to choose which Bluetooth events it wishes to receive events notification from.	549
<code>int bt_disable_events(int hdl, long events);</code>	Disables Bluetooth events	549
<code>int bt_perform_inquiry(device_table_t *device_table);</code>	Performs a device inquiry and returns the number of Bluetooth devices found.	551
<code>int bt_bluetooth_configuration(GAP_Pairability_Mode_t pair-Mode);</code>	Configures the bluetooth connection character.	551

Table 3 Function Calls

Function Call	Description	Page
<code>int bt_spp_configuration(BD_ADDR address, unsigned int port, char *service_name, boolean server);</code> <code>int bt_sppl_configuration(BD_ADDR address, unsigned int port, char *service_name, boolean server);</code>	Configures the connection established when an application opens the DEV_SPP_BT or DEV_SPPL_BT device.	552
<code>int bt_dun_configuration(BD_ADDR address, unsigned int port);</code>	Configures the Bluetooth address and port number of the DUN port.	553
<code>int bt_obex_configuration(BD_ADDR address, unsigned int port);</code>	Configures the Bluetooth address and port number of the access point port.	553
<code>int bt_pan_configuration(BD_ADDR address, int portNumber, PAN_Service_Type_t localType, PAN_Service_Type_t remoteType);</code>	Configures the Bluetooth address of the access point	554
<code>int bt_sdp_search_services(BD_ADDR address);</code>	Returns the services supported by a remote connection.	555
<code>int bt_version_string (bt_component_t component, char *versionString, int allocatedSize);</code>	Retrieves various version strings available from either the Verix Bluetooth Driver or the Bluetooth Stack.	555
<code>int bt_mdm_profile_load (int hdl);</code>	Loads the modem profile to the modem.	556
<code>int modem_sleep(int hdl);</code>	Places the Conexant Harley modem chip into Deep Sleep mode.	513
<code>int modem_wake(int hdl);</code>	Wakes modem from Deep Sleep mode.	513
<code>int set_event_bit(int hdl, long flag);</code>	Selects a Verix event bit to use to notify the owner of DEV_SPP_BT of events.	546
<code>long get_event_bit(int hdl);</code>	Returns the event bit selected by <code>set_event_bit()</code> .	547
FIFOs		
<code>int SVC_CLR_FIFO(fifo_t *fifo, int datasize);</code>	Initializes the FIFO data structure pointed to by <code>fifo</code> with a capacity of <code>datasize</code> bytes.	342
<code>int SVC_CHK_FIFO(const fifo_t *fifo);</code>	Returns the number of bytes currently stored in the FIFO (that is, those written to it, but not yet read).	342
<code>int SVC_GET_FIFO(fifo_t *fifo);</code>	Retrieves byte from FIFO.	343
<code>int SVC_PUT_FIFO(fifo_t *fifo, int val);</code>	Add a byte to FIFO.	343
<code>int SVC_READ_FIFO(fifo_t *fifo, char *buf, int size);</code>	Reads bytes from FIFO.	344
<code>int SVC_WRITE_FIFO (fifo_t *fifo, const char *buf, int size);</code>	Writes bytes to FIFO.	344
CRCs		
<code>unsigned int SVC_CRC_CALC(int type, const char *buf, int size);</code>	Calculates a CRC value for <code>size</code> bytes of data in <code>buffer</code> .	347
<code>unsigned long SVC_CRC_CALC_L (int type, const char *buf, int size);</code>	Identical to <code>SVC_CRC_CALC()</code> , except that it returns a 32-bit result.	347

Table 3 Function Calls

Function Call	Description	Page
<code>unsigned char SVC_LRC_CALC(void const *buf, int size, unsigned char seed);</code>	Calculates the LRC (longitudinal redundancy check) value for <i>size</i> bytes of data in <i>buffer</i> .	352
<code>unsigned int SVC_CRC_CRC16_L(void const *buf, int sz, unsigned int seed);</code>	Calculates a standard CRC16 CRC value for <i>size</i> bytes of data in <i>buffer</i> .	349
<code>unsigned int SVC_CRC_CRC16_M(void const *buf, int sz, unsigned int seed);</code>	Calculates a standard CRC16 CRC value for <i>size</i> bytes of data in <i>buffer</i> .	350
<code>unsigned int SVC_CRC_CCITT_L(void const *buf, int sz, unsigned int seed);</code>	Calculates a 16-bit CRC for <i>size</i> bytes of data in <i>buffer</i> using the CCITT polynomial	348
<code>unsigned int SVC_CRC_CCITT_M(void const *buf, int sz, unsigned int seed);</code>	Calculates a 16-bit CRC for <i>size</i> bytes of data in <i>buffer</i> using the CCITT polynomial	348
<code>unsigned long SVC_CRC_CRC32_L(void const *buf, int sz, unsigned long seed);</code>	Calculates a 32-bit CRC32 CRC value for <i>size</i> bytes of data in <i>buffer</i> .	351
<code>unsigned int SVC_MEMSUM(const char *buf, long size);</code>	Computes the sum of <i>size</i> bytes from <i>buffer</i> , treating both the bytes and the sum as unsigned, and ignoring overflows.	337
<code>unsigned int SVC_MOD_CK(const char *acct);</code>	Generates a Luhn check digit for a sequence of digits or validates a sequence of digits containing a check digit.	338

General Communication Device Functions

Note: The calls listed in this section apply to all COM devices. The calls listed in specific COM sections apply only to that device.

<code>int BT_Si2434_profile_load(void);</code>	Verifies base station type and downloads updated profile version when version is different on existing modem.	537
<code>int download(int hdl, void *parms);</code>	Receives a download through the open serial port.	558
<code>int IPP_power(int type);</code>	There is no IPP power on/off hardware.	519
<code>unsigned long get_usb_device_bits (void)</code>	Gets the status of all supported USB devices.	524
<code>int reset_port_error(int port);</code>	Resets error conditions for parity, framing, and overrun.	435
<code>int select_pinpad(int type);</code>	Always returns zero as there is no port multiplexing hardware.	519
<code>int set_serial_lines(int hdl, char *buf);</code>	Normally sets or resets DTR, RTS, and BRK based on <i>buf</i> .	436

Table 3 Function Calls

Function Call	Description	Page
<code>int set_usb_multi_device(int hdl, int onOff);</code>	Sets handle to an open USB port.	523
USB Ethernet (ETH1)		
<code>int get_enet_status(int hdl, char *status4);</code>	Checks whether Ethernet link is live or not.	797
<code>int get_enet_MAC(int eth_hdl, char *MACbuf);</code>	Returns the MAC (Media Access Control) address.	797
<code>int set_enet_rx_control(int hdl, int rx_control);</code>	Enables/disables multicast and broadcast.	798
Modem Port (COM3)		
<code>int SVC_ZONTALK(unsigned char type);</code>	Receives a download through the terminal modem.	514
Internal PIN Pad (IPP) Port (COM5)		
<code>int PINentryStatus(void);</code>	Returns the PIN entry status and can be used to infer when the console belongs to the PIN-entry background task.	520
<code>int SVC_INFO_PIN_PAD(char *buf_lx);</code>	Fills the caller's buffer with the one-byte internal PIN pad availability information from the manufacturing block.	521
<code>int TerminatePinEntry(void);</code>	Ends the PIN entry session.	520
USB Internal WiFi (WLN1) (on VX 680 only)		
<code>int USB_WIFI_POWER(int power);</code>	Used to power the WiFi on or off.	525
USB Client (on VX 820 only)		
<code>int get_usbd_status(int hdl);</code>	Checks whether the USB initialization is complete	801
<code>int usb_pending_out(int hdl);</code>	Returns the amount of written but unsent data in the driver's buffers.	801
Multitasking		
Task Function Calls		
<code>int _exit(int status);</code>	Terminates the calling task.	195
<code>int *get_errno_ptr(void);</code>	Returns the address of the errno value for the task.	195
<code>int get_group(void);</code>	Returns the effective file group membership of the calling task.	196
<code>int get_native_group(void);</code>	Returns the group of the currently executing task.	197
<code>int get_task_id(void);</code>	Retrieves the task number.	197
<code>int get_task_info(int id, struct task_info *info);</code>	Stores information about a specified task in the info structure.	198
<code>int grant_owner(int devhdl, int task_id);</code>	Grants temporary ownership of the device.	193

Table 3 Function Calls

Function Call	Description	Page
<code>int revoke_owner(int devhdl);</code>	Revokes the temporary ownership granted by <code>grant_owner()</code> .	193
<code>int run(const char *file, const char *args);</code>	Executes the specified program file as a new task.	199
<code>int set_errno_ptr(int *ptr);</code>	Allows the program to change the pointer to a location in the current task.	201
<code>int set_group(int group);</code>	Changes the effective file group membership of the calling task.	201
<code>int run_thread(int routine, int parameter, int stacksize);</code>	Executes the specified thread as a new task.	200
Semaphore Application Function Calls		
<code>int sem_close(sem_t *hdl);</code>	Returns the global semaphore handle to the system.	207
<code>int sem_init(sem_t *sem, unsigned int value);</code>	Initializes a semaphore to the value given.	205
<code>int sem_wait(sem_t *sem);</code>	Causes the calling thread to be suspended if the semaphore is unavailable.	207
<code>int sem_post(sem_t *sem);</code>	Frees the semaphore for general use and returns the handle immediately.	207
<code>int sem_prop(sem_t *semaphore, unsigned short mode);</code>	Allows visibility of global semaphores to the calling task.	209
<code>sem_t * sem_open(char *id, int sbz);</code>	Opens a global semaphore before it can be used.	206
Pipes		
<code>int pipe_connect(int pipehandle, int targetpipehandle);</code>	Allows data to be written using one handle, and subsequently to be read using the other handle.	216
<code>int pipe_init_char(int hdl, int max_input_chars);</code>	Initializes the character mode pipe.	218
<code>int pipe_init_msg(int hdl, int max_input_msgs);</code>	Initializes a message mode pipe.	218
<code>int pipe_init_msgX(int pipe_handle, int max_messages_pending);</code>	Configures the pipe to receive the <code>pipe_extension_t</code>	219
<code>int pipe_pending(int hdl);</code>	Tests pipe data availability.	219
<code>int SVC_RESTART(const char *filename);</code>	Performs a complete terminal reset.	220
Support for APACS40 Cryptographic Functions		
<code>int Calc_Auth_Parm(const char *TranData, char *AuthParm);</code>	Computes the authentication parameter based on the provided transaction data.	700
<code>int Calc_MAC(const char *buf, int len, char *mac8);</code>	Computes the standard ANSI X9.19 message authentication code for the designated buffer.	700

Table 3 Function Calls

Function Call	Description	Page
<code>int Create_MAC_Key(int hostkey, const char *A, const char *B);</code>	Sets the current MAC key based upon use of the One Way function.	701
<code>int Init_MAC(void);</code>	Allows multiple tasks to use APACS40 features (one at a time).	701
<code>int New_Host_Key(int hostkey, const char *rqst_residue, const char *resp_residue);</code>	Updates the current host key in the APACS40.KEY Group 0 file for the designated host.	702
<code>int Reset_Key(int hostkey);</code>	Resets the current host key in the APACS40.KEY Group 0 file for the designated host.	702
<code>int Term_MAC(void);</code>	Clears the current owner variable of the APACS40 feature set so that another task can use the feature.	703
Special Topics		
Stack Interface		
<code>int closeaux(int devhdl);</code>	Closes the device status specified by the device handle.	726
<code>int dbdump(void *buf, int bufsize);</code>	Dumps memory into the system log.	719
<code>int openaux(int devhdl);</code>	Returns a Verix device handle for a limited device to be used for link monitoring.	726
Dual Sim		
<code>int SIM_SELECT(int simindex);</code>	On VX 520 GPRS selects which SIM card to use.	730
<code>int set_SIM_slot(int hdl, int slotNumber);</code>	On VX 680 3G this API selects the specified SIM card slot.	731
<code>int get_SIM_slot(int hdl);</code>	On VX 680 3G this API returns the index of the currently selected SIM card slot.	731
DLL		
<code>int unload_DLL (int load_address);</code>	Allows the OpSys to recover memory allocated to the program.	716
Feature Enablement		
<code>int feature_license_total(void);</code>	Returns the number of feature licenses installed.	733
<code>int feature_license_expiration(featureLicenseTag* pTag, int* pUnit);</code>	Returns license expiration status details.	733
<code>int feature_license_is_enabled(featureLicenseTag* pTag);</code>	Returns boolean indicating if license is currently enabled.	734
<code>int feature_license_get_tag(int index, featureLicenseTag* pTag);</code>	Returns installed license tags based on index.	734
<code>int feature_license_get_detail(featureLicenseTag* pTag, char* licenseFieldName, char* buff, int buffLength);</code>	Returns the details of a specific feature license Name=Value pair.	735

Table 3 Function Calls

Function Call	Description	Page
Security/Crypto Functions		
<code>int crypto_read(int hdl, char *buf, int count);</code>	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.	566
<code>int crypto_write(int hdl, const char *buf, int count);</code>	Encrypts and writes count bytes of data from buffer to the open file associated with handle.	567
<code>int DES(unsigned char ucDeaOption, unsigned char *pucDeaKey8N, unsigned char *pucInputData, unsigned char *pucOutputData);</code>	Performs DES, DESX, and triple-DES computations.	571
<code>int GenerateRandom(unsigned char * random8);</code>	Returns an 8-byte random value.	572
<code>int AES(unsigned char ucAesOption, unsigned char *pucAesKey8N, unsigned char *pucInputData, unsigned char *pucOutputData);</code>	Performs AES computations on 128-bit data block.	570
<code>int isAttacked(void);</code>	Indicates if an attack occurred, causing the loss of the transaction keys or encrypted files.	572
<code>int rsa_calc(unsigned short * msg, unsigned short *mod, int wds, int exp, unsigned short * result);</code>	Performs a public key RSA computation.	574
<code>int SHA1(unsigned char * unused, unsigned char *input_buffer, unsigned long nb, unsigned char * sha20);</code>	Performs an SHA-1 computation as described in FIPS PUB 180-2.	575
<code>int iPS_GetScriptStatus(unsigned char ucScriptNumber, unsigned char *pucINName);</code>	Checks if a VeriShield security script file is installed in the terminal and if so, returns the name of the script.	578
<code>int iPS_InstallScript(char *pucINName);</code>	Installs a VeriShield security script file in the unit.	579
<code>int iPS_ExecuteScript(unsigned char ucScriptNumber, unsigned char ucMacroID, unsigned short usINDataSize, unsigned char *pucINData, unsigned short usMaximumOUTDataSize, unsigned short *pusOUTDataSize, unsigned char *pucOUTData);</code>	Starts the execution of a given macro from a given loaded VeriShield security script.	580
<code>int iPS_UninstallScript(unsigned char ucScriptNumber);</code>	Uninstalls the specified VeriShield security script from the unit.	581
<code>int iPS_CancelPIN(void);</code>	Cancels the PIN processing.	583
<code>int iPS_GetPINResponse(int *piStatus, PINRESULT *pOUTData);</code>	Checks the status of the PIN session.	583
<code>int iPS_RequestPINEntry(unsigned char ucPANDataSize, unsigned char *pucINPANData);</code>	Initiates the PIN collection.	585
<code>int iPS_SelectPINAlgo(unsigned char ucPinFormat);</code>	Selects the PIN algorithm used during the next PIN session.	586
<code>int iPS_SetPINParameter(PINPARAMETER *psKeypadSetup);</code>	Configures several parameters for the upcoming PIN session.	588
<code>int iPS_CheckMasterKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKVC);</code>	Indicates if a key is present in the specified location.	591
<code>int iPS_DeleteKeys(unsigned long ulKeyType);</code>	Deletes the specified set of keys.	592

Table 3 Function Calls

Function Call	Description	Page
<code>int iPS_LoadMasterClearKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKeyValue);</code>	Loads the security script's master keys.	593
<code>int iPS_LoadMasterEncKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKeyValue);</code>	Loads the security script's master keys without deleting the keys already loaded.	594
<code>int iPS_LoadSysClearKey(unsigned char ucKeyID, unsigned char *pucINKeyValue);</code>	Loads the KLK (system keys).	595
<code>int iPS_LoadSysEncKey(unsigned char ucKeyID, unsigned char *pucINKeyValue);</code>	Loads the system keys.	596
<code>int get_rkl_krd_cert(char *filename);</code>	Retrieves the Key RecFailureng Device certificate and writes the data to filename.	597
WWAN Functions		
<code>int set_gsm_break(int hdl);</code>	Sends a break signal to the GSM radio.	786
<code>int set_gsm_powersave(int hdl, unsigned int power);</code>	Sends the GSM radio power save command setting.	786
<code>int set_radio_ctl(int hdl, const char *sigs);</code>	Sets and resets a control line to the radio module.	787
<code>int get_radio_sts(int hdl, char *sigs);</code>	Returns the status of RAD_INT and RAD_INT2.	788
Download Operations		
<code>int authenticate(const char *signature_file_name);</code>	Marks the file as authentic.	756
<code>int file_copy_auth_bit(const char *source, const char *target);</code>	Allows applications to copy the authenticated bit of a source file to a target file.	757
Power Management Functions		
<code>int BatteryRegs(char *buf);</code>	Returns the registers in the buffer.	93
<code>int battery_remain_charge(void);</code>	Returns the percentage of the remaining battery charge.	94
<code>int cs_set_sleep_state(int sleep);</code>	Allows the application to put the keypad into a low power sleep mode and wake it back up.	94
<code>int disable_host_power(void);</code>	Disables the V*600 host's power.	95
<code>int enable_host_power(void);</code>	Enables the V*600 host's power.	95
<code>int get_battery_initialization_status(void);</code>	Returns the initialization status.	96
<code>int get_battery_sts(void);</code>	Indicates battery status.	97
<code>int get_battery_value(int type);</code>	Returns the requested battery values.	98
<code>int get_dock_sts(void);</code>	Indicates if the unit is docked or undocked.	99

Table 3 **Function Calls**

Function Call	Description	Page
<code>int get_performance_counter(void);</code>	Returns the current number of cycles since the last power cycle.	99
<code>int get_performace_frequency(void);</code>	Returns the current value of FCLK for the terminal main processor.	99
<code>int get_powersw_sts(void);</code>	Indicates if the power switch is being held down.	100
<code>int pause_battery_monitor(long on_off);</code>	If the parameter is 1 the battery monitor is blocked, if 0 the battery monitor block is removed.	96
<code>int set_battery_value(int type, int value);</code>	Allows the user to set the type of the call and the parameter, if relevant.	100
<code>int set_com_pwr(int port, cons char *sig);</code>	Controls the power on both COM2 and COM8 ports simultaneously.	101
<code>int set_com1_pwr(const char *sigs);</code>	Controls terminal power.	101
<code>int SVC_INFO_BAT_REQ(char *char1);</code>	Returns '2', indicating that battery is required for printing, but not for GPRS SIM protection.	102
<code>int SVC_SLEEP(void);</code>	The terminal goes to sleep after 50 ms if all applications are idle.	102
<code>int SVC_SHUTDOWN(void);</code>	Commands the terminal to turn itself off.	103
<code>int USB_COM2_RESET(void);</code>	Commands USB EM660 radio modem to power off/on.	103

Table 4 displays module IDs and device.

Table 4 **Module ID**

Module ID	Logical Name	Device
0	MID_NO_MODEM	No modem on COM3
2	MID_UNKNOWN_MODEM	Unknown modem on COM3
4	MID_BANSHEE_ONLY	Conexant Banshee modem
6	MID_CO561_ONLY	Connect One Ethernet 10BaseT only
7	MID_CARLOS_CO561	Connect One Ethernet 10BaseT and Carlos combo
8	MID_MC56_ONLY	Siemens GSM/GPRS US only
9	MID_MC55_ONLY	Siemens GSM/GPRS International only
10	MID_EM3420_ONLY	Sierra CDMA 1xRTT only
11	MID_CO710_ONLY	Connect one WiFi 802.11b only
12	MID_CARLOS_MC56	Siemens GSM/GPRS US and Carlos combo
13	MID_CARLOS_MC55	Siemens GSM/GPRS International and Carlos combo
14	MID_CARLOS_EM3420	Sierra CDMA 1xRTT and Carlos combo
15	MID_CARLOS_CO710	Connect One 802.11b WiFi and Carlos combo

Table 4 **Module ID** (continued)

Module ID	Logical Name	Device
16	MID_EISENHOWER_ONLY	Predator Conexant Eisenhower modem
17	MID_EISEN_USB_ETHERNET	Eisenhower/USB Ethernet combo
18	MID_EISEN_EM3420	Eisenhower/CDMA combo
19	MID_EISEN_MC56	Eisenhower/GPRS USA combo
20	MID_EISEN_MC55	Eisenhower/GPRS International combo
21	MID_EISEN_USB_WIFI	Eisenhower/USB WiFi combo
22	MID_BANSHEE_CO210	Banshee/CO210 Ethernet combo
23	MID_CO210_ONLY	CO210 Ethernet
24	MID_ISDN_ONLY	ISDN
25	MID_BANSHEE_USB_ETHER	Banshee/USB Ethernet combo
40	MID_HARLEY_MODEM	Trident Harley Modem
42	MID_COM2_UART	COM2 is configured as 2 wire UART
50	MID_USB_MODEM	USB modem
60	MID_BTEZ1	BT Ezurio brand module 1
61	MID_BTEZ2	BT Ezurio brand module 2
62	MID_BTEZ3	BT Ezurio brand module 3
63	MID_BTEZ4	BT Ezurio brand module 4
64	MID_BTAA1	BT alternate vendor module 1
65	MID_BTAA2	BT alternate vendor module 2
66	MID_BTAA3	BT alternate vendor module 3
67	MID_BTAA4	BT alternate vendor module 4
70	MID_M200	Kyocera M200 CDMA
72	MID_MC55i_ONLY	Sierra MC55i GPRS
73	MID_MC5727	Sierra MC5727 CDMA
74	MID_SOC_ETH	Internal Ethernet
75	MID_USB_HOST_PWR	Powered USB Host
76	MID_USB_HOST_NO_PWR	USB Host not powered
77	MID_USB_HOST_HUB	USB with internal hub
78	MID_USB_DEV	USB device
79	MID_CTL5	Contactless for the VX 680
80	MID_SD_A	SD Slot A
81	MID_SD_B	SD Slot B
82	MID_TOUCH_RES	Touchscreen type - resistive
83	MID_TOUCH_CAP	Touchscreen type - capacitive
84	MID_HAUWEI_EM660	HAUWEI EVDO radio
85	MID_DUET820_MCU	VX 820 DUET base
86	MID_BCM432291_WiFi	BCM432291 WiFi
87	MID_BCM43291_BT	BCM432291 BlueTooth

Table 4 **Module ID** (continued)

Module ID	Logical Name	Device
88	MID_BGS2	Cinterion BGS2 GSM/GPRS radio
99	MID_TBD	To Be Determined

CAUTION

Any value for COM2HW and COM3HW variables other than those listed in [Table 4](#) may cause applications and OS to incorrectly handle the module.

Function Call Error Codes

Error codes for the Verix eVo OS are listed in [Table 5](#). These are reported by returning a result of `-1` with `errno` set to a specific standard error code.

NOTE

These values are defined in `errno.h` included in the folder of your Verix eVo SDK installation.

Table 5 **Error Codes Set by Function Calls**

Code	Value	Description
EPERM	1	Caller does not have necessary privileges.
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 over SPI.
ENXIO	6	No such device or address (or beyond limit).
E2BIG	7	Arguments list is too long; EXEC list > 5120 bytes.
ENOEXEC	8	EXEC format error (file has no “magic” number).
EBADF	9	All functions other than <code>open()</code> ; Console owned by another task; invalid file handle.
ECHILD	10	No child processes (for <code>WAIT</code>).
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).
EFAULT	14	Bad address (hardware fault using argument).
ENOTBLK	15	Block device required (for example, for <code>MOUNT</code>).
EBUSY	16	A write or control function was issued before the previous function is completed (device or directory in use).
EEXIST	17	File already exists.
EXDEV	18	Cross-device link; link to another device.
ENODEV	19	<code>open()</code> function: Console currently owned by another task. Other tasks: No such device or inappropriate call.
ENOTDIR	20	No a directory (for example, in a path prefix).

Table 5 Error Codes Set by Function Calls

Code	Value	Description
EISDIR	21	Is a directory; cannot write to a directory.
EINVAL	22	Invalid function parameter (argument).
ENFILE	23	File table overflow; no more OPENs allowed.
EMFILE	24	Too many file handles in use.
ENOTTY	25	Not a typewriter.
ETXTBSY	26	Text file busy; cannot EXEC open file.
EFBIG	27	File too large.
ENOSPC	28	No space remains or other write error on device.
ESPIPE	29	Illegal seek; cannot LSEEK to a pipe.
EROFS	30	Read-only file system; device is read only.
EMLINK	31	Too many links to a file.
EPIPE	32	Broken pipe.
EDOM	33	Input value to math function not in domain.
ERANGE	34	Output value from math function out of range.

DBMON Abort Codes

Table 6 lists common debug abort codes.

Table 6 *DBMON Abort Codes

Code	Description
1	Unable to open device for PC communication.
2	*DBMON value invalid.
3	USB device failure.

Event Codes Returned by wait_event()

Event codes returned by `wait_event()` are listed in Table 7. For more information, see `wait_event()`. There are more device drivers and devices and there are events in the 32 bit event mask. These events can be overloaded, see `set_event_bit()` for more information.

Table 7 Defined Events

Name	Device	Description
EVT_ACTIVATE	Console	Console ownership returned to application.
EVT_BAR	Bar Code Reader	Input available.
EVT_CLK	Clock	Generated once per second for the task that owns DEV_CLOCK.
EVT_COM1	COM1	Input available on COM1.
EVT_COM2	COM2	Input available on COM2.
EVT_COM3	COM3	Input available on COM3.
EVT_COM4	COM4	Input available on COM4.
EVT_COM5	COM5	Input available on COM5.

Table 7 **Defined Events** (continued)

Name	Device	Description
EVT_COM6	COM6	Input available on COM6 on VX 680 terminal. This is also available on the converter module of the Qx120 Contactless device.
EVT_COM8	COM8	External serial, /* com 8 I/O */
EVT_BIO	USB Device	Issued when there is an incoming data from the fingerprint reader.
EVT_USER	CTLS	/* post_user_event summary bit */
EVT_CONSOLE	Console	Display output complete.
EVT_DEACTIVATE	Console	Console ownership lost.
EVT_ICC1_INS	Smart Card	Customer card inserted.
EVT_ICC1_REM	Smart Card	Customer card removed.
EVT_IFD_READY	Interface Device	Read complete from the IFD. Issued to the current owner of the IFD channel.
EVT_IFD_TIMEOUT	Interface Device	Read time-out on the IFD. Issued to the current owner of the IFD channel.
EVT_KBD	Console	Keyboard input available.
EVT_MAG	Card Reader	Input available; signaled on a card swipe. To set this trap, the card device must be open and the operating system card swipe buffer is empty.
EVT_NETWORK	Network	Input available on Ethernet port.
EVT_PIPE	Pipe	Input arrived on a pipe.
EVT_SHUTDOWN	Terminal	The terminal turns off.
EVT_SOKT	Socket I/O	
EVT_SYSTEM	System	Universally interesting event.
EVT_TIMER	Timer	User-defined through the <code>set_timer()</code> function.
EVT_USB	USB	Input available on USB port.
EVT_WLN	USB WiFi	Incoming data and PIMFOR management packets set this event.
EVT_USB_CLIENT	USB Client	Reports client events. The OS determines the type of USB client device it presents to a USB host at boot time.
EVT_REMOVED	Case Removal Latch	Notifies the OS that the keypad unit has been removed from its host system by monitoring the case removal switch.

Managing Application Data—Effective Use of Memory Space

An important design issue for Verix eVo-based terminals is efficient use of memory, especially memory space. Use the following tips in managing data space:

- Minimize the use of global variables. Unlike local variables, global variables are not dynamic, and they *always* occupy space in memory. Local variables only occupy stack space when they are used.
- Avoid deep nesting of functions. Nesting function calls uses large amounts of stack space.
- Avoid using literals in `#define` because they tend to use large amounts of stack space. When declared as a `#define`, the string is duplicated in the memory each time it is referenced in a function call.
- Design the application to rely on downloaded data files and table-driven techniques to make it more flexible and easier to maintain and update.

Communications Buffer Space

Communications system buffers are used by the data communication device drivers (modem, RS-232, PIN pad, and so on). They are also used for inter-program communication when writing to a message-type pipe.

- In Predator, buffers are 64 bytes in length and can contain up to 63 bytes of data. Trident buffers are 1024 bytes in length and can contain up to 1023 bytes of data.
- The maximum number of allocated buffers is 255, and the minimum number is 8. By default, the maximum number is allocated.
- The environment variable `*B` in `CONFIG.SYS` contains the number of buffers allocated by the system for Predator terminals. `*B` is ignored for Trident.
- Buffers are automatically joined together as needed to store messages of up to 4 KB.

Erase Flash

The only way to erase flash is to call `dir_flash_coalesce()`. Files in flash are tagged as deleted, but are not erased until `dir_flash_coalesce()` is called. `dir_flash_coalesce()` does not completely erase flash. This call is invalid for Trident. Trident has an automatic wear leveling function built in.

Keypad

Figure 1 illustrates the keypad layout of VX 520.

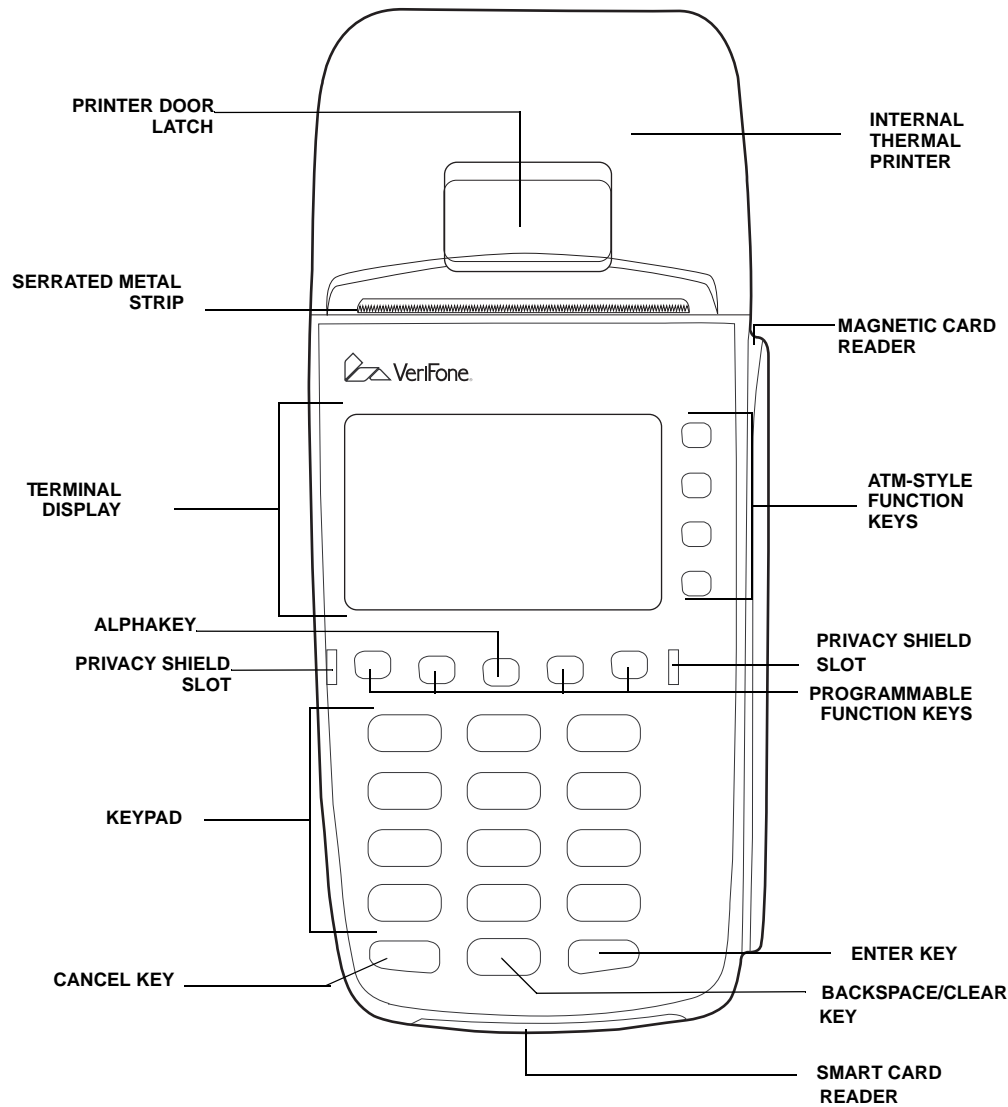


Figure 1 VX 520 Keys

NOTE



Some terminals have discrete function keys next to or below the display. Other terminals have touch screens with ATM keys and function keys implemented as part of the display touch panel.

Table 8 lists the keypress scan codes.

Table 8 Keypress Scan Codes

Keypress	Scan Code	Description
1	0xB1	1 with the high-order bit set.
2	0xB2	2 with the high-order bit set.
3	0xB3	3 with the high-order bit set.
4	0xB4	4 with the high-order bit set.
5	0XB5	5 with the high-order bit set.
6	0xB6	6 with the high-order bit set.

Table 8 **Keypress Scan Codes** (continued)

Keypress	Scan Code	Description
7	0xB7	7 with the high-order bit set.
8	0xB8	8 with the high-order bit set.
9	0xB9	9 with the high-order bit set.
*	0xAA	* with the high-order bit set.
0	0xB0	0 with the high-order bit set.
#	0xA3	# with the high-order bit set.
CANCEL	0x9B	ESC with the high-order bit set.
BKSP	0x88	BS with the high-order bit set.
BKSP (long key press)	0x8E	SO with the high-order bit set.
ALPHA	0x8F	SI with the high-order bit set.
ENTER	0x8D	CR with the high-order bit set.
F0 ^a	0xEE	n with the high-order bit set.
F1 ^b	0xFA	z with the high-order bit set.
F2 ^b	0xFB	{ with the high-order bit set.
F3 ^b	0xFC	with the high-order bit set.
F4 ^b	0xFD	} with the high-order bit set.
F5 ^a	0xEF	o with the high-order bit set.
a (leftmost horizontal screen key)	0xE1	a with the high-order bit set.
b (mid-left horizontal screen key)	0xE2	b with the high-order bit set.
c (mid-right horizontal screen key)	0xE3	c with the high-order bit set.
d (rightmost horizontal screen key)	0xE4	d with the high-order bit set.

a. Applicable for 16 x 21 LCD (V*670).

b. The V*525 does not have F1-F4 keys. It has telco function keys on the left side of the telephone hand set. These keys are EA, EB, EC, and ED.

Application Programming Environment

This chapter presents a high-level overview of the hardware and software environment in which application programs run.

Verix eVo-based terminals provide most of the elements of a conventional computing environment. Application programs are written in C, compiled into native machine code, and then downloaded to the terminal. The Verix eVo operating system manages tasks, memory, files, input/output devices, and other system resources. Applications request Verix eVo operating system services through a trap mechanism encapsulated in a system call library.

The major components of the Verix eVo operating system are illustrated in Figure 2.

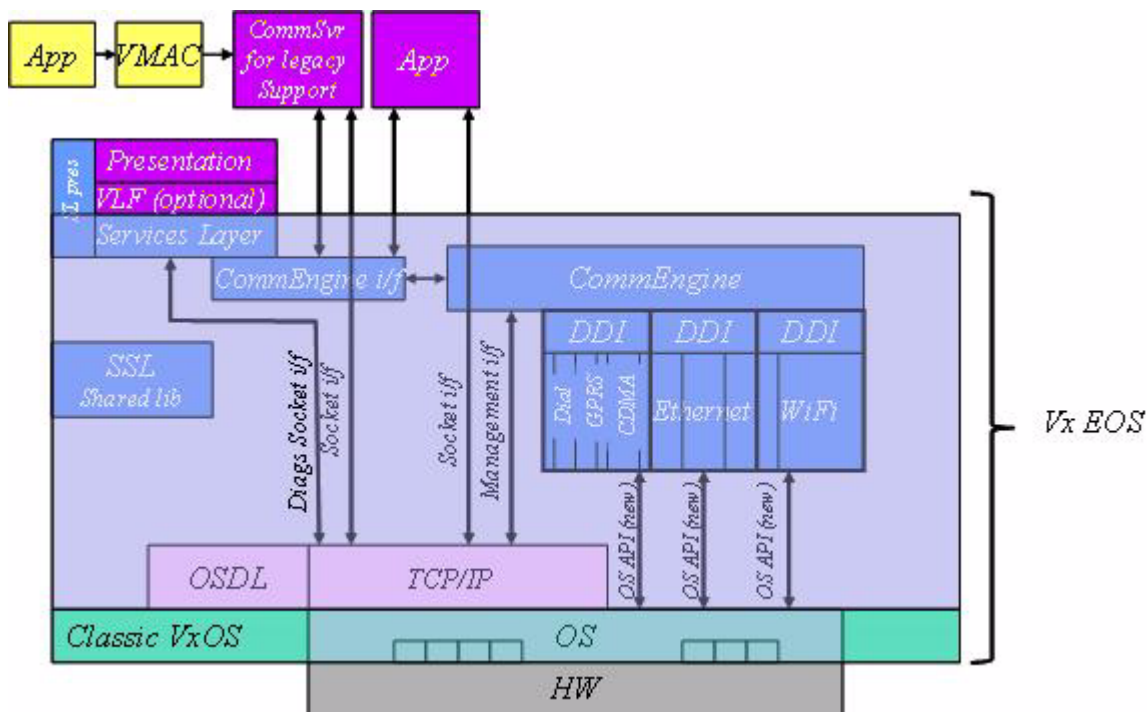


Figure 2 Verix eVo Block Diagram

Hardware

Verix eVo-based terminals are built around an ARM 11 processor. These terminals incorporate some of the following components:

- ARM 32-bit CPU
- A minimum of 160 MB memory.
- A Memory Management Unit (MMU) that affords a high degree of protection between separate application tasks, as well as between applications and the operating system
- An LCD display panel and keypad that provide the primary means of user interaction
- Additional input/output devices, which can include a beeper, real-time clock, internal thermal printer, modem, and serial communication ports to connect external devices

General Principles

The following are the general design principles for the Verix eVo operating system:

- Formal OS structure.

A layered operating system consisting of a low-level device layer and a high-level service layer. This is similar to the PC system software that consists of a low-level BIOS and a high-level BIOS.
- C-compatible interface for OS service calls.

For Verix eVo-based terminals, the OS conforms to the ARM-Thumb Procedure Call Standard (ATPCS). Refer to the ARM documentation for description of this interface.
- Consistent register conventions (mandated by the ARM compiler and ATPCS).

Since Verix eVo codes are generally written in the C programming language, these details are generally automatically handled by the C compiler.
- Application portability.

Applications are coded in standard third-generation languages (for example, ANSI C) with significant device and file features designed for transaction terminals.

ARM compiler option-b aligns the data section on a 4 KB boundary. This is required for Trident terminals and if used on a Predator terminal, the application can run on both platforms.

Virtual Memory

The VeriFone eVo terminal is composed of several types of memory: Flash, static, and volatile RAM. There are other specific memory regions, but for applications this description focuses on the file system memory and the execution of applications.

Trident terminals also have the I: and F: file systems, however, both systems are in flash. All files downloaded to a Trident terminal are copied to flash. All files created by applications are in flash. For execution speed, application executables are copied to volatile RAM and executed there. This also pretains to the OS. The differences of memory type with respect to F: and I: are transparent to applications and appears just like the Predator terminals. The MMU manages task separation as in the Predator terminals.

The file systems are persistent over power cycles. File writes are guaranteed to complete. If a power fail occurs during a file write, the OS checkpoints the operation and completes the write process after the terminal restarts.

Memory Management Unit (MMU)

All Verix eVo terminals support virtual memory using hardware-based memory management features. One goal of the MMU is to give each task its own “private” view of memory that cannot be seen or altered by any other user task. This goal is achieved by reloading key registers in the MMU each time a new task is scheduled to run. The `SCHEDULR` module performs the actual reloading of the MMU registers.

User memory, both code and data, is mapped as tiny pages (1 KB) in virtual memory space.

Codefiles, including library files, must always begin on 1 KB boundaries. The file manager performs this alignment automatically.

The default setting for the heap allocation (`*HEAP`) on VX 680 3G is set to support the maximum number of simultaneous networks which includes WiFi, BT/ETH, BT/DIAL, 2G/3G and GPS. The maximum heap allocation setting for `*HEAP` is increased above the default setting to allow additional HEAP allocation.

Virtual Memory Map

The MMU manages application execution and maps the application into virtual memory. User memory is allocated using the “7xx” sections, that is, all application memory lies in the range 0x70000000 through 0x7FFFFFFF.

Normal applications have a virtual memory map with the following general plan:

- 0x70000000..0x700FFFFFF: system global library. The library itself is approximately 20 KB; the remainder of this virtual megabyte is unavailable for users, since all users share the mapping of this megabyte using a common MMU table.
- 0x70100000..0x703FFFFFF: expansion space, usually for shared libraries, but can include expanded stack space.
- 0x70400000..0x7041FE00: high end of user stack. The stack limit is always 0x7041FE00, with the lower address computed based on the value in the

user's header. If the stack size exceeds 130560 bytes, then additional virtual megabytes must be mapped at 0x703xxxxx, extending to 0x702xxxxx, then to 0x701xxxxx. The entire stack cannot exceed 3276288 bytes (3.124 MB) and must be contiguous.

- 0x7041FE00..0x7041FFFF: part of global data area (reserved for system use: for example, errno)
- 0x70420000..0x71FFFFFF: user code; then user data; then space for shared libraries (which can also use whatever remains in the region 0x70100000..0x703FFFFFF not used by the stack). For most applications, whose virtual memory requirement is less than 1 MB, only one MMU section is required, corresponding to 0x70400000..0x704FFFFFF.

The virtual memory map is vital to application programmers trying to debug an application in a Verix terminal. System Error Log address displays the virtual address as well as the debugger.

Shared Memory

The eVo terminal supports multi-tasking. Given the MMU task separation some mechanism is needed to share data between tasks. Global shared memory is that mechanism but is only supported in Trident terminals.

The variable `*MAXSHM` controls the total number of active shared memory handles available in the system. `*MAXSHM` can be set to any number between 1 and 256. The default value is 16. Each shared segment is created with a specified size up to 1 MB. However, the shared memory segment is allocated out of the OS memory space and has practical limits. By default, each segment is accessible by all tasks. Access can be restricted by GID when the segments are created.

Typically shared memory is used for process control and passing of small amounts of data. There is global semaphore support but shared memory is not restricted semaphores. It is highly recommended that other techniques be used for sharing large amounts of data such as files in GID 15. This is a limited resource.

Configuring Number of Shared Regions

The variable `*MAXSHM` controls the total number of active shared memory handles available in the system. `*MAXSHM` can be set to any number between 1 and 256. The default value is 16.

shm_open()

This function call creates a new shared memory object, or opens an existing one for use by the calling application. If the application has multiple threads, it does not matter which thread opens the region, since all threads share the same virtual memory space.

Prototype

```
void *shm_open (const char *name, int flag, int size);
```

Parameters

char *name	<p>The name of the shared object, an area in the memory. By default this name is visible system wide—every task can share this memory.</p> <p>Note: The creator can restrict access to the memory object by using a name that defines a group restriction. Names that begin with a numeric value followed by slash are interpreted to represent a group restriction. Only tasks with access to the identified group can access this memory. For example, the name “FOO” is visible system wide, but the name “1/FOO” is visible only to programs with access to file group 1.</p>
flag	<p>Can be one or more of the following:</p> <p>O_CREAT, creates the object if it does not exist. When combined with O_EXCL, the call will fail if it already exists, returning a result of -1 with errno set to EEXIST. Access rights for a memory region depend on what was specified at the time the region was created. The creator is automatically granted read-write access. Other users, however, are restricted to read-only access unless the creator specifies write access.</p> <p>O_RDWR, the memory object can be both written and read. When shared memory can be written by multiple tasks, extra care may be needed.</p> <p>O_RDONLY, the memory can be read but not written. This restriction does not apply to the creator of the shared memory region.</p> <p>O_EXCL, useful when combined with O_CREAT to ensure that the caller is the one creating the shared memory object.</p>
size	<p>The amount of space needed in the shared area. This is ignored if the region already exists.</p> <p>Note: Only the creator can specify the size of the region, and that size cannot be changed. The maximum size of a shared memory region is 1 MB (1,048,576 bytes). If the call succeeds, the return value is a pointer to the shared memory region. All sibling threads in the caller’s process can use this memory, it does not matter which one called shm_open. If the call fails, the return value will be -1, and errno will be set to indicate the reason for failure.</p>

Return Values

Failure: -1 with `errno` set to:

- `EACCES`, the name cannot be read due to memory access rights.
- `EINVAL`, size is invalid: must be non-zero positive number not exceeding 1048576. Group is invalid if name includes group restriction.
- `ENOMEM`, too many shared memory regions, either process has already used all eight, or the system maximum has been reached.
- `ENOENT`, named region doesn't exist and caller didn't specify `O_CREAT`.
- `EEXIST`, named region already exists but caller specified `O_CREAT` and `O_EXCL`.
- `EROFS`, creator of this region specified `O_LOCK`, you must use `O_RDONLY`.

WARNING



Applications designed to share a memory region must name the region identically. For instance, the memory regions “FOO” and “1/FOO” are not the same. Both may exist simultaneously as distinct regions.

The maximum number of shared memory regions that a program may open is eight regions. This affects scheduling time and large numbers will adversely affect system performance. The maximum number is per process, not per thread—if four sibling threads in a process each open two shared memory regions, no more will be allowed.

shm_close()

This function call frees up the shared memory region previously obtained by calling `shm_open()`. The region will no longer be accessible to the caller, but if there are other users it will remain available to them; when the last user of the region calls `shm_close`, then the memory will be returned to the system.

Prototype `int shm_close (void *region_address);`

Parameters

<code>region_address</code>	The shared memory region.
-----------------------------	---------------------------

Return Value

Success:	0
Failure:	-1 with <code>errno</code> set to: <code>EINVAL</code> , indicates <code>region_address</code> does not support shared memory. <code>EBADF</code> , caller does not own the memory.

memory_access()

This function call is used to test if a region of memory can be read or written. This can be used for any kind of memory, not just shared memory.

Prototype `int memory_access (const void *buf, int len);`

Return Value If the OS supports the call, returns:

- 2, if the area can be read or written.
- 1, if the area is read-only.
- 0, if the area is not accessible at all.

If the OS does not support the call (old OS), this returns `-EINVAL`.

Two macros are defined:

```
#define readable_memory(buf,len) ( memory_access(buf,len) > 0 )  
#define writable_memory(buf,len) ( memory_access(buf,len) > 1 )
```


File System

The Verix eVo operating system implements a file system in FLASH memory. Applications can create and access files using function calls such as `open()`, `read()`, `write()`, and `seek()`. Though their use is not recommended, the C language standard input/output interface (`fopen`, `fprintf`, and so on) is also implemented.

To provide system and application access to the file space, the OS manages this memory region as a virtual device, much in the same manner as physical devices.

File Locations

For compatibility with earlier platforms, Verix eVo files can belong to several different "drives"—"I:" and "F:" drives are supported. The "I:" drive is the default drive. If you specify the name "FOO" it is assumed to be the same as the "I:FOO" file.

Both file systems are physically maintained in non-volatile (NAND) FLASH memory. In general, you place your files in either "drive" since they are treated the same in nearly all cases.

NOTE



If an application is intended to run on previous platforms as well as on Verix eVo platforms. It is best to follow the conventions for file placement that apply to the previous platforms.

File Guidelines

The underlying physical medium (NAND FLASH) has constraints that may affect the performance of your application and ultimately the lifetime of your terminal. For best results, these guidelines must be followed:

- 1 Data is stored in pages of 2048 bytes. When writing a large number of records (such as, when copying a file), performance will be best if the records are multiples of 2048 bytes, and if the records are aligned in multiples of this size.

For example, if you wish to write 100 KB, you have better performance if you write 50 records of 2 KB each, rather than 1024 records of 100 bytes each. The same is somewhat true for reads, although in this case, the OS maintains a cache that will serve to minimize physical reads.

- 2 NAND FLASH has a limited life. Once a page is written, it cannot be updated again until it has been erased.

Typically only 100,000 write/erase cycles are possible over the lifetime of the unit. This allows a very large number of writes, but it is NOT an infinite number. Similarly, NAND FLASH cannot be read indefinitely.

If you have "constant" files that must be read repeatedly, consider reading the data into memory when you first need it and then using this copy for future use.

NOTE

A major difference between Predator and Trident is that the file system in Trident is all flash. Creating files in RAM is not possible. The practice in Predator of creating a RAM file for temporary or transient data and accessing this data multiple times per transaction or creating a file and deleting per transaction will rapidly erode the FLASH in Trident and the terminal will see corrupted data in these files in a relatively short period.

File Groups

There is no hierarchical directory structure. Group 0 is reserved for the operating system and Groups 1–15 are available to applications. Groups are not specified as part of filenames. Normally, a task can only access files in its own group, that is, the group that contains its executable `.OUT` file. There are two exceptions to this rule:

- Group 1 tasks are allowed to change their effective group (by calling `set_group`) to any Group 2–15. This allows them to access files in other groups (but only one group at a time).
- Any task can change its group to 15. Thus, Group 15 serves as a global shared file group. Group 15 files can also be directly designated by a slash (/) prefix to the filename (for example, `/batch.dat` or `F:/cardlist`).

Group 46 is added to support all three file systems ("I:" and "F:" and the new "N:" drives). Most group 46 files are placed in "N:" drive but dynamic data files may be present in both "I:" and "F:" drives as needed. Group 46 is primarily used to hold the Verix eVo "middleware" files for configuring and managing various networks.

NOTE

Groups 16-45 are Verifone private protected groups which are saved for projects in the future.

Record-Structured Files

In addition to simple stream-of-bytes files, Verix eVo files also support several forms of record-structured files. These include variable-length record (VLR), compressed variable-length record (CVLR), and keyed files (a special type of CVLR file that maintains a collection of key-value pairs). See [Variable-Length Records](#).

CONFIG.SYS File

The `CONFIG.SYS` file is a keyed file residing in terminal memory that contains system options and parameters. It plays a role analogous to environment variables in other systems (and can be accessed through the C `get_env()` library function as well as by direct file reads). For example, in GID1, the record with key `*GO` designates the application program that automatically launches when the

terminal is powered up or reset. Variables can be set in the `CONFIG.SYS` file using the download utility, from the keypad, or from application programs using the `put_env()` library function. Use of the `CONFIG.SYS` file and the available variables are described in [System Configuration File](#).

Power-fail File Protection

Verix eVo OS ensures that file input/output operations are reliable in case of power failures. If a `write()` or other system call that modifies files is interrupted by a power failure, it completes on terminal restart.

Handles

Applications manipulate devices, files, and pipes with handles assigned during the `open()` call. Verix eVo OS can support up to 32 device handles.

The file manager assigns up to 30 handles (by default), nominally between 32 and 61. To change the maximum from 30 to some other number, set the `CONFIG.SYS` variable `*FILE`.

Similarly, the pipe manager assigns up to 256 pipe handles. To save memory, set the `CONFIG.SYS` variable `*PIPE` to a lower value.

Device APIs

Following are general functional definitions of the device services. The usage here is intended to conform to standard POSIX conventions.

- `write (int hdl, char const *buf, int count);`

This function is used to transfer application data from the calling application to the external world. Failure is reported using a result code of -1 and further identified by setting the caller's `errno` variable. Otherwise the result code indicates the number of characters transferred.

Note that a write operation returns immediately to the caller. Typically, however (especially for serial devices) the actual transfer of data occurs on a delayed basis. The caller must not conclude that a successful write operation means that the intended receiver has actually received the data.

- `read (int hdl, char const *buf, int count);`

This function is used to transfer application data from the external world to the calling application. Failure is reported using a result code of -1 and further identified by setting the caller's `errno` variable. Otherwise, the result code indicates the number of characters transferred.

Data usually transfers from the external world (for example, over a serial line) to an intermediate buffer maintained by the device driver. The read operation fetches the data from that intermediate buffer, not directly from the device. Thus read is an immediate operation which does not include device delays.

On the other hand, substantial amounts of processing may be involved. For example, when reading from the magnetic stripe reader, the raw data bits that the interrupt service routine stored are decoded at three levels to return nicely ordered ASCII data to the caller.

For files in the RAM-based or ROM-based file system, the handle identifies an open file and the data transfers directly from the file system to the caller's buffer.

- `open (char *pathname, int opentype);`

The open command prepares the device for operation. This call means different things to different devices. Generally, any initialization steps which must occur before the device can be useful, such as clock divisor programming, should be handled within this call. Also, in MOST cases the open call initiates the interrupt-driven, background mechanism of the device driver. This means that henceforth the device driver is periodically awakened by interrupts, so that it can function within the background, independently of the normal foreground activity of the system. The exception to all of this is found in COM device drivers, as previously mentioned.

If an error is detected during open processing, a return code of -1 is given to the caller, and the caller's `errno` variable is set to reflect the type of error detected. Otherwise the open call returns a file handle with which subsequent "device" calls reference the activated file.

- `close (int hdl);`

The close operation suspends device operations in an orderly manner. It is essentially the analogue of the open call. This call means a variety of things to different devices, but in all cases it deactivates the background interrupt processes of the device (if running).

If the handle represents a file in the ROM-based or RAM-based file system, then the close operation simply marks the corresponding file entry available for use.

- `lseek (int hdl, long offset, int origin);`

This entry point supports the standard `Unix lseek()` call for manipulating the file pointer of an open file. Although all drivers are required to support this entry, it is currently used only by the file manager. Refer to [File Management](#) for more information.

- `errno`

`errno` is a global variable declared in the `svc.h` file. `errno` is used by the OS to report details about a failure. Since it is global to the task environment, it should be cleared prior to an OS call to ensure that if a failure occurs, the error reported actually came from the current call.

Not all OS calls use `errno` and do not set the value so it is possible to get confusing results if not cleared by the application each time.

Devices

In addition to the console, Verix eVo-based terminals include a variety of other input/output devices. These vary among terminal models but typically include a beeper, real-time clock, magnetic card reader, and serial communication ports (that can be used to connect to a printer or external modem). Other devices that may be present include an internal modem, bar code reader, smart card reader, and internal thermal printer.

Application tasks access devices by opening them by name—for example, `open (DEV_CLOCK)`—and making read, write, and control function calls. The operating system arbitrates requests for device use by different tasks using an ownership model similar to that used for the console.

The table below presents a brief comparison of all devices available on different terminals.

Table 9 Comparison of Available Devices Across V^x Platforms

Device	VX 520	VX 520 GPRS	VX 680	VX 675	VX 820 / VX 805 DUET
COM1	Internal serial	Internal serial	External serial on Handy-Link	External serial on Handy-Link	N/A
COM2	Not available	Internal GPRS	Internal GPRS/CDMA/Bluetooth	Internal GPRS	USB serial (Duet only)
COM3	Internal dial	Internal dial	USB external dial	USB dial (charging base)	USB dial (Duet only)
COM4	Integrated printer	Integrated printer	Integrated printer	Internal printer	USB printer (Duet only)
COM5	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator
ETH1	internal Ethernet	internal Ethernet	internal Ethernet	internal Ethernet	N/A
ETH2	USB external Ethernet	USB external Ethernet	USB external Ethernet	USB external Ethernet	USB ETHERNET
COM6	USB external serial	USB external serial	USB external serial		USB external serial
COM8	N/A	N/A	N/A		N/A
WLN1	N/A	N/A	USB internal WiFi		N/A

Console

The console, besides having different display sizes has touch screen and color on some models. All terminals have the basic numeric keys, ENTER, CANCEL, and BACKSPACE but the other keys vary from terminal to terminal.

The VX 680 has no function or screen keys or alpha key.

NOTE



If the alpha key is not present, the user must use cell phone mode to enter alpha characters.

The console comprises the display and keypad. The LCD display size is 8 lines by 21 characters (8 x 21) for 168 characters on most terminals. An ASCII font is built into the unit. Other fonts can be downloaded as files. Graphic images can be displayed by creating custom font files. VX 680 and VX 820 terminals use 240 x 320 pixels and 20 x 30 characters.

The Verix eVo-based terminal keypads contain a 12-key numeric keypad and a set of additional *programmable* function keys under the screen, the functions are defined (as with *all* keys) by the application running on the terminal. The four ATM-style keys to the right of the 12-key numeric keypad are OS-defined. See keypad illustration.

Verix eVo-based terminal mediates sharing of the console among the application tasks. A task that successfully opens the console becomes its owner, preventing other tasks from using it. The owner task can relinquish the console either permanently or temporarily to allow other tasks to use it.

Some terminals incorporate a new key type called the Navigation key. This is a 4-way rocker switch that can be used to navigate up, down, right, left or whatever the application decides.

Verix Terminal Manager

Verix Terminal Manager (VTM) is the first task run when a terminal is powered up or reset. VTM is responsible for administrative tasks such as starting the initial application, receiving downloads, setting `CONFIG.SYS` variables, setting the clock, setting display contrast, starting the debug monitor, and so on. The VTM task remains active as long as the terminal runs, and can be reentered at any time by pressing F2 and F4 (or 7 and Enter) and entering a password. ENTER + 7 is the preferred password entry method. Not all terminals have the F2+F4 keys. Applications interrupted by entering VTM can be restarted only by a complete terminal reset. Refer to the Reference Manual of your terminal for a detailed description of VTM.

Customizable Application Launcher

The application program can be customized to run on VTM and is supported in any GID.

Adding the CONFIG.SYS variables

- Add `*MENUx` and `*MENUxy` variables in the `CONFIG.SYS` file.
 - x= menu title (1 - 9)
 - y=function key label, function name and parameter(s) (1 - 4)

The system searches for variables in the order of `*MENU1`, `*MENU2`, and so on until the next variable is defined. When the system finds the `*MENUx` variable, it searches for variables in the order of `*MENUx1`, `*MENUx2`, and so on. It displays the function label on the appropriate line and appends “Fy” where “y” is the function key number. It continues to search if the next variable `*MENUxy` is not found.

Working Mechanism

The following settings are used to configure new high-level screens that invoke specific user-written applications.

```
*MENU1=NETWORK DIAGNOSTICS
```

```
*MENU12=CONFIG,NETCONFIG.OUT,37
```

```
*MENU13=PING,PING.OUT,62
```

```
*MENU2=ICC DIAGNOSTICS
```

```
*MENU22=SLOT 1,ICCDIAG.OUT,1
```

```
*MENU23=SLOT 2,ICCDIAG.OUT,2
```

```
*MENU24=SLOT 3,ICCDIAG.OUT,3
```

```
*MENU5=MY DIAGNOSTICS
```

- 1 Press the **F2** key to launch the program **NETCONFIG.OUT** with parameter 37.
- 2 Press the **F3** key to launch the program **PING.OUT** with parameter 62.
- 3 Press the **F1** key to go to the previous menu; in this case, Menu 7.
- 4 Press the **F2** key to go to the next menu specified by *MENU(x+1); in this case, *MENU2.
- 5 Press the **F2** key to go to the next menu specified by *MENU(x+1); in this case, *MENU3. Since *MENU3 does not exist, the terminal goes to VTM menu 1.

When the user selects one of the user programs identified, VTM closes the console and invokes the program. When that program exits, VTM re-opens the console and displays the current menu again.

The exceptions—if the CONFIG.SYS variables are configured as described above and the user selects the PING option F3 on the NETWORK DIAGNOSTICS menu, it results in error. The following are the types of errors displayed:

- The specified program file may not exist.
- It may not be authenticated.
- Associated library files may not exist or may not be authenticated.
- There may be insufficient memory to run the program.

Any of the above types of errors cause the `run()` command to fail.

Verix Battery Management

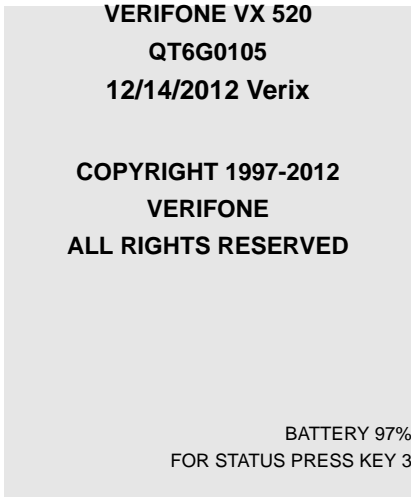
The Predator Vx610 and Vx670 and the Trident VX 680, VX 520 GPRS and VX 675 are portable devices. They can run on battery for some amount of time — the amount of time can vary considerably depending on how the application is designed and use. The battery must be charged periodically by connecting the terminal to wall power, either by plugging in the power pack or setting the terminal in a base station. The battery can be queried for status or conditioned, there are

several functions for each status which vary between Predator and Trident so read the battery functions carefully. Battery conditioning is a terminal specific feature based on the type of battery and charging circuit used. Read the conditioning functions to see if they are required for the terminal being used.

Without the battery pack, the printer cannot be used even if it is connected to AC power. Removing the battery pack while printing and connected to AC power causes a printer mechanism error. Refer to [Printer, Battery, and Radio Interaction](#) for more information.

The VTM Battery Support menus are listed in [Table 10](#):

Table 10 Battery Support

Display	Action
 <p>VERIFONE VX 520 QT6G0105 12/14/2012 Verix</p> <p>COPYRIGHT 1997-2012 VERIFONE ALL RIGHTS RESERVED</p> <p>BATTERY 97% FOR STATUS PRESS KEY 3</p>	<p>During startup, the terminal displays a copyright notice screen that shows the terminal model number, the OS version stored in the terminal's flash memory, the date the firmware was loaded into the terminal, and the copyright notice.</p> <p>This screen appears for three seconds, during which time you can enter Verix Terminal Manager by simultaneously pressing ENTER and 7 key.</p> <p>You can extend the display period of this screen by pressing any key during the initial three seconds. Each keypress extends the display period an additional three seconds.</p> <p>If the battery has not been initially charged, the screen displays BATTERY NOT CALIBRATED to inform the user to initialize and condition the battery.</p>

From the menu 2 of the VTM>VERIX DIAGS MGR screen, press down arrow to see additional entries, select 1>battery status. This will display the battery status information of the device.

Battery Conditioner

The cells and a monitor chip are assembled into the battery pack. The monitor chip does not know how charged the battery is. The battery conditioner calibrates the monitor chip by putting the battery pack into a known state, fully charged as determined by a battery voltage greater than 8.0 volts and a charging current less than 40mA for over 1 minute. The conditioner then sets the current monitor register to the largest value and discharges the battery. When the battery is fully discharged, determined by the battery voltage falling below 6.5 Volts, the conditioner reads the amount of current discharged and calls that value as full charge capacity. The conditioner then clears the charge counter and fully charges the battery. When the battery is fully charged as determined by the conditions above, the conditioning cycle ends and the terminal will return to the battery status screen.

The battery initializer, initializes the full charge and remaining charge registers in the battery pack. It does it by constantly monitoring the battery charging current. When the initializer detects that the battery charger chip has stopped charging the battery, it checks the battery's full charge register.

If the value in the full charge register is 1701 or 1251, the initializer sets the full charge register to 1700 or 1250; otherwise, the initializer does not change the full charge register. Then the initializer writes the full charge register value into the remaining charge register.

Battery Conditioner APIs

`retval=start_battery_conditioner()` ; is a non-blocking call that initiates the battery conditioner and returns the status. The OS will condition the battery in the background. The application can get the battery conditioner status by using `retval=battery_conditioner_status()` ;

start_battery_conditioner()

A non-blocking call that initiates the battery conditioner and returns the status.

Prototype

```
int start_battery_conditioner(void);
```

Return Values

-22	Not a portable unit
-3	No battery
-2	Temperature out of range
-1	No external power pack; On VX 675, returns -1 with errno set to <code>ENODEV</code> .
1	First charge
2	Discharge
3	Second charge
4	Calibrate current

NOTE



On VX 675, there is no capacity measuring hardware. The battery capacity cannot be determined.

battery_conditioner_status()

Retrieves the status of the battery conditioner.

Prototype `int battery_conditioner_status(void);`

Return Values

-22	Not a portable unit
-1	On VX 675, returns -1 with errno set to ENODEV
0	Idle
1	First charge
2	Discharge
3	Second charge
4	Calibrate current

NOTE



On VX 675, there is no capacity measuring hardware. The battery capacity cannot be determined.

Verix Power Management

The Verix eVo operating system reduces power consumption whenever possible during sustained periods of application inactivity. Application inactivity is when all active tasks have been in a wait state for a specified period of time. The actions taken by Verix eVo operating system to reduce power include the following:

- Peripherals are powered off, including the magnetic card reader, UART, DUART, backlight for display and keys and printer.
- Other peripherals are placed in low-power mode, including the IPP microprocessor and the printer's microprocessor.
- The baud-rate-generator for the three internal serial ports is disabled.
- The microprocessor is placed in `Sleep` mode. This disables the phased-locked loop (PLL) circuitry for clock generation and the integrated microprocessor core is shut down, reducing power to about 500 μ A.

NOTE



The word "docked" implies the external power pack connected and "undocked" implies no external power pack connected.

VeriFone has many portable terminals and there are some very dramatic differences and many subtle differences in how this is implemented between Predator and Trident terminals and even within the various Trident terminals. First, to understand these differences we need to define some terms:

OFF	The terminal is in a powered off state. When powered on from the OFF state the OS and then the applications are restarted.
ACTIVE-RUN	The application is running. The processor is running full speed and the application is in control. Power saving can be made but it is the role of the application to manage power in this state.
ACTIVE-IDLE	<p>The processor is slowed due to the application being suspended. Suspension is likely caused by the application calling</p> <p><code>SVC_WAIT()</code> or <code>wait_event()</code>. Transition to ACTIVE-RUN occurs on the millisecond clock tick as the scheduler is still</p> <p>active, the system is just running at a slower clock rate, and the <code>wait_event()</code> or <code>SVC_WAIT()</code> are ready. Note that ACTIVE-IDLE state cannot be entered if certain devices are active such as Ethernet.</p>
DEEP SLEEP	<p>The least power consuming state without going to the OFF state. The terminal enters DEEP SLEEP and resume to ACTIVE-RUN instantly without restarting the terminal. Note that network and communication devices will be disconnected and have to re-establish connection upon resumption from DEEP SLEEP.</p> <p>DEEP SLEEP turns off the processor clocks so devices can not operate. There are mechanisms to assist in customizing device action when entering DEEP SLEEP. This mode is not supported in Predator.</p>

SLEEP	The term used for Predator terminals and is also known as SLOW MODE. This is the same state as ACTIVE-IDLE. However, when entering SLEEP state the display will show a standby screen.
HIBERNATE	This is a special Vx700 mode. It is similar to DEEP SLEEP where the CPU kernel is powered off, but the display shows the standby screen.
Standby time	The amount of time that a terminal can remain available for use before the battery is discharged and the terminal goes to the OFF state.

All terminals act the same when connected to wall power. Applications have options to conserve power but when on wall power the terminals are in ACTIVE-RUN state. The OS does not look to manage power. Once removed from wall power a terminal is dependent on its battery. How long the terminal can remain powered on with its battery is what power management is about.

One common aspect of Predator and Trident terminals and power management is that for the OS to manage power, the application must be in a quiet state. The application must be waiting for some event or timer to expire. If the application is processing, the terminal is in ACTIVE-RUN state.

When all applications are idle, Predator portable terminals transitions from ACTIVE-RUN to ACTIVE-IDLE (SLEEP) either waiting for events or in SVC_WAIT. Control of the timing is determined by various CONFIG.SYS variables such as *POW, *OFF and *OFFD.

When in ACTIVE-IDLE the processor clock is slowed and the display screen shows the standby image. If no activity is present for the time defined in *OFF the terminal will power off. Pressing the RED CANCEL button for a period of time will also cause the unit to power off. Once power is turned off, the user must press the GREEN Enter key or connect wall power to the unit to move to ACTIVE-RUN state.

The Trident portable terminals similarly transitions from ACTIVE-RUN to ACTIVE-IDLE state when all application are idle. However, the display will be blank—no standby image is shown. In ACTIVE-IDLE state, the clocks are slowed but will instantly go to ACTIVE-RUN once an event or wait state ends. If no events or timer expires after a configurable time, the terminal will go to DEEP-SLEEP where the clocks are turned off. Any device dependent on the terminal clock will need to be restarted.

After the configured time period elapsed, the terminal will go from DEEP-SLEEP to OFF, where the OFF state means the terminal is totally powered down and the terminal can be restarted to go from OFF to ACTIVE-RUN state.

The following CONFIG.SYS variables control terminal power consumption or behavior of the various modes. The first value T or P indicates if the variable is a Trident or Predator or found in both OS.

T	*WAKEUP	Alarm clock function to force the terminal to wake up from OFF state.
T	*PM	Master environmental setting for conveniently selecting a "standard power management profile.
T	*DSP_BL_DLY	Seconds to wait, while in ACTIVE IDLE, before dimming display backlight. After this time elapses again, minimize display backlight.
T	*EE	Energy-efficiency setting. Default varies by product, but typically is 1 for battery-powered devices and 0 otherwise.
T	*KBD_BL_DLY	Seconds to wait, while IDLE, before turning keyboard backlight off.
TP	*OFF	Seconds to wait before going from IDLE to OFF.
TP	*OFFD	Seconds to wait before powering off: visible countdown.
TP	*POW	Milliseconds to wait, while idle, before entering DEEP-SLEEP state.
TP	*DARK	Enables darker printing; non-default values require more power.
T	*DIMLED	Percentage: brightness of status and logo LEDs.
TP	*TURNOFF	Seconds to hold red key to force power off.
TP	*TURNON	Seconds to delay during startup, if no AC present; used to prevent inadvertent startup.
TP	*GKE	Green key event handling.

*PM will force the following values when set to:

MAX Performance	MAX Operational	MAX standby
*PM = 0	*PM = 1	*PM = 2
*DSP_BL_DLY		
0: Never dim	20: Default	2: Dim after 1 second
*EE		
0: No ACTIVE IDLE	1: Use ACTIVE IDLE	1: Use ACTIVE IDLE
*KBD_BL_DLY		
0: Never dim	7: Default	2: Dim after 1 second
*OFF		
36000: Maximum	300: Default	36000: Maximum
*POW		

0: No DEEP SLEEP 60000: Default 500: Minimum

Sleep Mode

Once in `Sleep` mode, the unit wakes periodically to determine if any applications are ready to run.

Hibernate

This state is specific to the Vx700 PIN pad. The CPU kernel is powered off while the display is on and has the standby screen from Sleep mode. When the Vx700 unit enters this state after *OFF seconds of sleep, all serial ports CTS lines become inactive.

Pressing ENTER wakes up the Vx700 PIN pad, as will any serial port CTS going active. After waking up from this mode, the application restarts. The Vx700 does not display the sign on screen. Upon restart, not all USB devices on Vx700 may have enumerated yet.

Wakeup Events

Verix eVo OS polls for various wakeup events, including expired timers in the application area, special key detection (CANCEL key to power OFF; ENTER key to wake up), low-battery detection (forces power OFF), or docking. Except for the low-battery condition, which requires immediate action, all other conditions require debouncing.

NOTE



The PIT continues to operate when the unit is running – whether docked or not – but the PIT interrupt is disabled.

Power Management System

The terminal power-management system can be categorized into a number of states. Key points to notice are:

- The terminal always powers up in a running state and assumes it is docked. During power-up it quickly determines if it is undocked, and if so, if the battery level is sufficient for operation. If not, it immediately powers off.

NOTE



Docked implies the external power pack is connected and undocked implies that no external power pack is connected.

- To conserve battery power, the unit enters a special `sleep` state when all applications have been idle for a specified period.
- When the unit enters low-power `sleep` mode, the backlight is turned off. In addition, the console displays a standard "idle" screen.
- The unit wakes from low-power `sleep` mode based on the expiration of an application timer, the use of the green (ENTER) key, or by docking the unit (applying AC power).
- Four external conditions will cause the unit to perform an orderly power-off sequence: loss of battery power (critically low charge), removal of the battery



pack, an attack on the security system (such as opening the case), or holding the red (CANCEL) key for two full seconds.

- Whenever the unit docks or undocks, each application receives the `EVT_SYSTEM` indicator. Interested applications can use `get_dock_sts()` to determine the current state of the unit.

NOTE

On VX 680, the LCD backlight ramps the brightness up when going from off to on, and ramps the brightness down when going from on to off. The time from minimum to maximum and maximum to minimum brightness is about 250 ms.

The soft on/off feature is present when entering and leaving low power sleep mode, and when using the existing `set_backlight()` function.

Serial Port Power Control

Power control of the COM2 and COM8 serial ports is required. The API `set_com_pwr()` may be used to turn the power pins on both COM ports on and off. Turning the power pin on or off affects both of the ports simultaneously. The power is either on for both ports, or off for both ports.

Bluetooth

Bluetooth is a low power device. When VX 680 3G goes idle, it enters a low power state to conserve power. There is no special handling required for Deep Sleep mode. The application closes the device before entering into Deep Sleep mode. When the device is closed, the BT stack is shut down and the power is turned off by the BT chip. When opened, the power is re-enabled requiring BT connections to be re-established.

The PMIC control for power to the chip is no longer available. Bluetooth is modified to control the `3V3_BT_POWER_ON` pin on GPC20. Both the WiFi and Bluetooth changes are designed to share the use of this pin.

WiFi

WiFi on a VX 680 3G terminal currently has 3 power save modes: `Mode 0` (always on), `Mode 1` (maximum power savings, but throughput is reduced), and `Mode 2` (maximize power savings while providing minimal impact to throughput). `Mode 2` is currently the default setting, but it can be changed to `Mode 1` to enforce better power saving.

The WiFi driver can be modified to configure use of deep sleep interrupts. The PMIC control for power to the chip is no longer available. The WiFi driver can be modified to control the `3V3_BT_POWER_ON` pin on GPC20. Both the WiFi and Bluetooth changes are designed to share the use of this pin.

3G

The Cinterion PHS8-P radio must be powered on on VX 680 3G even when it is in Deep Sleep mode to maintain a connection to the cellular network. The radio may take as long as 60 seconds to connect to the cellular network, thus, it is required to keep the radio on for as long as the terminal is on. The radio uses as little power as possible in power save/Sleep mode so it can be kept on even when the terminal is in Deep Sleep mode. In 2G mode (GSM/GPRS), radio “Sleep” current varies from 1.5-3.1 mA. In 3G mode (WCDMA), radio “Sleep” current varies from 1.2 to 3.1 mA.

The radio power save mode is enabled when the radio detects USB suspend on the USB bus. Radio power save is disabled when the radio detects USB resume on the USB bus. When the radio is in power save mode, it still monitors the cellular network for incoming data or radio network events such as losing the connection to the radio tower.

EOS must initialize the radio so it can wake up the system from “Deep Sleep mode”. The following commands are required.

```
AT^SCFG="MEopMode/RingOnData", "on"
```

```
AT^SCFG="URC/Ringline", "asc0"
```

```
AT^SCFG="URC/Datamode/Ringline", "on"
```

```
AT^SCFG="MEopMode/PowerMgmt/VExt", "low"
```

When the OS enters deep sleep, it calls the USB host driver sleep callback function, which calls the USB suspend callback function registered by the radio device driver (and for other USB devices, if needed). The radio suspend function sends a USB setup packet to the radio to enable USB remote wake up. At this point the radio driver is ready for “Deep Sleep” so it returns to the USB host stack. The USB host driver calls other USB device suspend callback functions, if needed. Once all the suspend callbacks are done, the USB host driver puts USB suspend on the bus. All attached USB devices including the radio enter USB power save mode.

The radio must be in what Cinterion calls “Sleep mode” and draw no more than 3.1 mA. The USB host driver returns to the OS.

The OS calls other sleep callback functions for other device drivers, if needed. When the OS is done with callback functions it can enter low power Deep Sleep mode. This is the lowest power state the terminal and radio can be in.

If the radio detects incoming data or a network event, the radio signals the deep sleeping CPU via an interrupt pin. The CPU exits Deep Sleep mode and the OS resumes running. The OS calls device driver wakeup callback functions including the one for the USB host driver. The USB host driver callback function puts USB resume on the bus then calls USB resume callback functions including the one for the radio. The radio USB resume function sends a USB setup packet to the radio to cancel USB remote wake up.

At this point, the incoming radio data or network event message (URC) is delivered over the USB bus to the radio device driver which delivers it to the appropriate COM port as usual.

Power Save: 2G versus 3G

Controlling 2G versus 3G cannot be done in the OS device driver so it must be done by EOS. Running in 2G mode may not save battery life. When the radio is on SLEEP mode, the radio draws less current in 3G mode than in 2G.

When the radio is transferring data, the radio draws more current in 3G mode. However, for a given file size, the radio in 3G mode can transfer the file much faster so it can go back to low power SLEEP mode sooner than when using 2G mode. The radio uses more power to transfer the same amount of data in 2G than in 3G mode.

Radio	Transfer Rate	Calculated Time to Download 1 MB file
BGS2, GPRS, 2G	7200 bps (900 bytes/s)	18.5 minutes
PHS8-P, WCDMA, 3G	1 MBPS (125,000 bytes/s)	8 seconds

Audio The Trident SoC on VX 680 3G includes DMU control bits for the I2S controller and the 10-bit DAC. The OS power manager stops the I2S and DAC clocks when entering Deep Sleep mode and restores the clocks when leaving.

The speaker amplifier chip has an `EN` (able) pin. When the chip is disabled, it draws 1 uA or less. The `AUDIO_EN` (GPC0) GPIO is connected to the amplifier `EN` pin so the OS driver can put the chip in lower power mode.

The speaker device driver sleep function puts the amplifier chip in low power mode using the `EN` pin. The speaker driver wakeup function enables the amplifier chip then restores the chip configuration registers. The contents of the registers revert to their default values when the chip is disabled.

Smart Card When OS11 enters and exits Deep Sleep mode, registered functions `ScHibernateNotification()` and `ScWakeNotification()` are invoked respectively. After all smart card slots are closed, the VF6101 puts the VF6101 chip in on a “reset” state. When opening a slot while VF6101 is in “reset”, the “reset” state is removed and the EMV L1 driver is initialized.

ScHibernateNotification()

Deactivates all smart card slots then puts the VFI6101 smart card chip into a reset state.

ScWakeNotification()

Removes reset state from VF6101 then initializes the smart card EMV L1 driver. The application prepares any slots previously powered before entering Deep Sleep mode.

Printer, Battery, and Radio Interaction

The Verix eVo battery system consists of a removable power pack and a charger chip in the terminal. The battery pack consists of two Li-ON cells, a DS2438 Smart Battery Monitor (fuel gauge) chip, and a charger chip VX 680 terminals support a battery capacity of 1800 mAH.

The DS2438 Smart Battery Monitor automatically measures and keeps track of the total amount of current entering and leaving (charging and discharging) the battery cells, as well as instantaneous current. These values are kept in registers in the DS2438 chip and are available to the OS.

The ISL6263 Battery Charger Chip on VX 680 is activated once the OS determines when the battery must be charged. The OS enables and disables this charger chip, which under normal operational mode is enabled when the battery voltage falls below 8.00 volts, and disabled when the charging current falls below 40 mA.

A thermistor (temperature sensor) has been placed in contact with the two cells of the battery. The VX 680 reads the thermistor via Channel 0. If the temperature is too low or too high, charging should be disabled.

On VX 680, the printer slows down when using GPRS or CDMA cellular modems, and when the radio is transmitting at high power. The battery cannot provide enough current for both to be running at maximum speed, simultaneously. The printer driver monitors a signal from the radio system that tells the printer driver when to slow down the printing.

NOTE



Unlike the GPRS/CDMA radio systems, there is no hardware indication from the USB WiFi system telling the printer to slow down.

The printer does not work without the battery pack, even when running on AC power. The devices that directly affect the amount of current available to the printer includes all smart card devices, GPRS/CDMA radio on COM2, USB WiFi, and USB host port on Handy-Link connector. Turning off as many of these devices as possible during printing will result in faster printing.

NOTE



On VX 675, terminal is powered by a single 18650 cell with a capacity of 2250 mAh that is not user-replaceable.

Applications can get the battery capacity by calling the function `BatteryRegs()`, and the COM2 device type by calling the function `SVC_INFO_MODULE(2)`.

If the application writes too many data to the printer, the OS buffers will eventually fill up and the `write()` function will return -1, and sets `errno` to `ENOSPC`.

If an application prints without a battery, printing will not occur. If the application writes too many data to the printer, the printer driver will discard the data to avoid overflowing the OS system buffers. If the application gets the printer status, the status indicates a mechanism error.

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. Flow control must support the following paradigm: Activity is detected, typically in the form of a keystroke or message, perform the task, then sleep again, to wait for more activity. The application indicates its readiness for sleep using the wait functions `wait_event()` or `SVC_WAIT()` available in the Verix eVo API.

Function Calls

This section details power management APIs.

BatteryRegs()

Returns the registers in the buffer. The return code is 1 if successful, or -1 if there is no battery. The input to this function is the address of a 36-byte buffer. This function is available on VX 680 and VX 520 GPRS (with battery config) terminals.

On VX 675, there are no battery registers, this API always returns -1.

Prototype `int BatteryRegs(char * buf);`

BatterySN()

On VX 675, there are no serial numbers, this API always returns -1.

Prototype `int BatterySN(char * buf);`

battery_remain_charge()

Returns the percentage of the remaining battery charge. This is a V*600 function call only.

Prototype `int battery_remain_charge(void);`

Return Values Success: Value from 0 to 100

cs_set_sleep_state()

This function call allows the application to put the keypad into a low power Sleep mode and wake it back up. The keypad will not function while on Sleep mode. When waking up the keypad, this function delays for 800mSec to allow the keypad processor time to wake up.

Prototype `int cs_set_sleep_state(int sleep);`

Parameters

<code>sleep</code>	1 - Puts the keypad to sleep
	0 - Wakes up the keypad

Return Values

Success:	0
Failure:	Non-zero

disable_host_power()

This function call disables the Vx600 host's power.

Prototype `int disable_host_power(void);`

Return Values Success: 0

enable_host_power()

This function call enables the Vx600 host's power.

Prototype `int enable_host_power(void);`

Return Values Success: 0

get_battery_initialization_status()

Returns the initialization status. The battery is considered initialized when it has been fully charged within a terminal and the Remaining Charge value is accurately calibrated. The return code is 1 if battery is initialized, 0 if not initialized, and -1 if there is no battery.

This function is available on VX 680 and VX 520 GPRS (with battery config) terminals.

Prototype `int get_battery_initialization_status(void);`

pause_battery_monitor()

If the parameter is 1 the battery monitor is blocked, if 0 the battery monitor block is removed. This is a Vx600 function call only.

Prototype `int pause_battery_monitor(long on_off);`

Parameters

- | | |
|---------------------|---|
| <code>on_off</code> | <ul style="list-style-type: none">• 1 = Block the battery monitor• 2 = Remove block of battery monitor |
|---------------------|---|

Return Values

Success: 0

get_battery_sts()

Indicates battery status. The application checks the battery status at the beginning of a transaction and refuses to allow another transaction to start if a level of 0 is returned. At the end of a transaction, the application typically checks the battery level and recommends docking or recharging, as appropriate.

If the battery desires a different threshold, the `get_battery_value` may be used.

Condition	State	Return Value/ Action
Battery voltage < 6.300 volts	Battery Critical	Unit will turn itself off
Battery Remaining Charge < 100 mAh	Battery Low	Return 0
None of the above	Battery OK	Return 1

Prototype

```
int get_battery_sts(void);
```

Return Values

1	Battery level is OK
0	Battery level is low.
-1	No battery.

get_battery_value()

Returns the requested battery values. The application may request Battery Full Capacity, Remaining Capacity, and Voltage.

Name	Value	Description
FULLCHARGE	0 - 2500	Theoretical MAX charge, in mAH
Note: On VX 675, always returns “theoretical FC” provided by the manufacturer.		
REMAININGCHARGE	0 - 2500	Remaining charge, in mAH.
Note: On VX 675, Charge Counter is not present. Direct Remaining Charge values cannot be provided. See get_battery_value() ; for more information.		
BATTERYVOLTAGE	6000-8500	Battery voltage, in mV.
Note: On VX 675, 3000 - 4300 battery voltage, in mV.		

Prototype

```
int get_battery_value(int type);
```

Return Values

- 1 No battery present
- 2 Battery Fault

get_dock_sts()

Retrieves docking status.

Prototype `int get_dock_sts(void);`

Return Values

0	Docked
-1	Undocked

get_performance_counter()

Returns the current number of cycles since the last power cycle. Note that this is instruction cycles not a clock. Cycles per second will vary depending on the current clock rate. The clock rate in battery powered Trident terminals will vary depending on the power saving state.

Prototype `int get_performance_counter(void);`

Return Values

Success: Number of cycles since last power cycle.

get_performance_frequency()

Returns the current value of FCLK for the terminal main processor. Note that FCLK can vary in battery powered Trident terminals depending on the power save state.

Prototype `int get_performance_frequency(void);`

Return Values

Success: Current FCLK value.

get_powersw_sts()

Indicates that the power switch key (red key) is being held down. If 1 is returned, the power switch key is being held down, and application should quickly finish any critical tasks as power will be turned off as soon as the key debounce is done. If 0 is returned, the power switch key is not being accessed.

Prototype `int get_powersw_sts(void);`

set_battery_value()

A generic battery function call that allows the user to set the type of call and parameter, if relevant. This call should not be used by most applications as the various battery functions have defined APIs already listed in the Programmer's Guide. This tool is generally used in debugging battery issues.

Prototype `int set_battery_value(int type, int value);`

Parameters

type	Defines the type of call selected.
value	Parameter type (if needed).

Return Values

Success:	0
Failure:	-1: EINVAL

set_com_pwr()

Controls the power on both COM2 and COM8 ports simultaneously. This is a bit-mapped parameter similar to the sigs parameter in [set_com1_pwr\(\)](#).

It returns 0 if the unit is powered on or off. -1 with errno set to EBADF if COM port is not opened, -1 with errno set to EINVAL, if open block is not set.

NOTE



For VX 520 terminals that do not have COM2, return value is -1 with errno set to ENODEV.

If the application executes `char sig=0x02;set_com_pwr(COM2, &sig)`, the OS turns on the power pin on both COM2 and COM8.

If the application executes `char sig=0x02;set_com_pwr(COM8, &sig)`, the power remains on in both ports.

If the application executes `char sig=0x00;set_com_pwr(COM2, &sig)`, the power pin on both COM2 and COM8 is turned off.

Prototype

```
int set_com_pwr(int port, cons char *sig);
```

set_com1_pwr()

This API controls power. This function will always return 0, sigs is a character that contains the following bit patterns:

<code>#define COM1_PWR_ON (1<<0)</code>	Turns on the COM1 level driver chip. Available on VX 680 terminals. Powerup default is ON.
<code>#define COM1_PWR_PIN_ON(1<<1)</code>	Turns on the power pin on the COM1 port. Powerup default is OFF.

Prototype

```
int set_com1_pwr(const char *sigs);
```

Example

(click on Example link to view attachment)

SVC_INFO_BAT_REQ()

Returns '2', indicating that battery is required for printing, but not for GPRS SIM protection.

Prototype `int SVC_INFO_BAT_REQ(char *char1);`

SVC_SLEEP()

This function returns 0 if the unit can go to sleep, or `EINVAL` if the unit cannot go to sleep because it is externally powered. This procedure temporarily sets the internal `*POW` variable to 50. If all applications are idle, the terminal goes into "Sleep mode" after 50 ms. Upon waking up, the internal `*POW` variable is restored. `*POW` in group1 `CONFIG.SYS` is never changed in this function.

This function is available on VX 680 and VX 520 GPRS.

Prototype `int SVC_SLEEP(void);`

SVC_SHUTDOWN()

This commands the terminal to turn itself off. If the function detects that it is not the correct terminal, it returns -1. If the VX 680/VX 520 GPRS terminal is being powered by an external power pack, it returns -1, otherwise this function returns 0.

Prototype `int SVC_SHUTDOWN(void);`

USB_COM2_RESET()

This function call is only used for the USB EM660 radio modem on Trident terminals. The call powers off the radio and then 550 milliseconds later turns the radio on. The radio is powered off 10.5 seconds after the non blocking call.

Prototype `int USB_COM2_RESET(void);`

Return Values

Success: 0
Failure: -1 - EINVAL

CONFIG.SYS Variables for Power Management

Three system settings that affect Sleep mode are listed below. The default values for these settings can be changed using variables in the CONFIG.SYS file in Group 1. The values are read at startup.

- ***POW** – indicates the amount of time (ms) that Verix eVo OS waits before attempting to place the unit in Sleep mode. The timer starts when all application tasks have become idle. A value of 0 indicates that the system will never enter low-power mode. The default value is 60,000 ms (60 s), and the maximum setting allowed is 600,000 ms (10 min).
- ***OFF** – an automatic transition from idle sleep to off occurs if the unit is continuously idle for 300 s (5 min). This parameter has a range of 1 s, to a maximum of 36,000 (10 hr). If *OFF is not present or its value is out of range, the terminal will use the default.
- ***GKE** – indicates what event, if any, is to be generated to notify applications that the user has woken up the unit with the green key. The default is for no event to be generated. Setting *GKE=1 triggers a console event for the current owner of the console. Setting *GKE=2 generates a system event for all applications.
- ***OFFD** – indicates the amount of time (s) that the OS delays between receiving the SVC_SHUTDOWN call and powering down the terminal. This variable is read on system restarts or reboots. The range is 2 seconds (default) to a maximum of 60 seconds.
- ***TURNON** – the number of seconds that the green key must be held down before the terminal turns itself on. Range is 1-5 seconds, and the default is 3 seconds.
- ***TURNOFF** – the number of seconds that the red key must be held down before the terminal turns itself off. Range is 1-9 seconds, and the default is 4 seconds.
- ***KBD_BL_DLY** – keypad backlight delay. Range is 0 to 60 seconds. Default is 7; 0 means that the keypad will not turn off during power save.
- ***DSP_BL_DLY** – display backlight delay. Range is 0 to 60 seconds. Default is 20; 0 means that display will not turn off during power save.



File Management

The Verix eVo-based terminal file system provides a flexible and efficient way to store and retrieve data. The system libraries provide functions to access a number of file types, each suitable for certain application requirements.

When designing an application, select file types based on the data to store and the access requirements for that data. The following sections describe each of the file access methods available and the type of data most suited to its use.

NOTE



Design file management routines to minimize the need to increase or decrease file size. Reuse record space when possible, instead of deleting and inserting new records, to significantly improve application performance.

The Verix eVo file management maintains all system environment information, user code, and data as memory files. Just as a PC application opens a file on disk, a Verix eVo application opens a file in memory — to the C language programmer, this is transparent.

To provide system and application access to the file space, Verix eVo OS manages this memory region as a *virtual* device, very much the same as the physical devices.

Verix eVo File Systems

Verix eVo-based terminals have several independent file systems. There is an updateable flash-based file system that behaves identically to the RAM-based system in nearly all respects (except, that it cannot be changed as easily), it is not explicitly mentioned, except in the few cases where its behavior is unique. Both I: and F: drives are on the same NAND flash memory, sharing the same pool of flash memory.

The OS automatically defragments NAND flash when a certain degree of fragmentation is detected.

NOTE



`dir_flash_coalesce()` and `dir_flash_coalesce_size()` are still provided. However, the application does not need to call these APIs as the OS routinely performs coalesce operations eliminating the need for an explicit `dir_flash_coalesce()`, and recoverable size obtained from `dir_flash_coalesce_size()` is always zero.

Since a new block is used every time a file is changed and other files in the system are not affected, `set_file_max()` is similarly provided for compatibility reasons.

In Predator, the `dir_flash_coalesce()` function call will force system restart. In Trident, this function always returns 0.

WARNING



NAND Flash has a definite life span and it is imperative that the application take action to minimize the number of writes and reads to flash.

In Predator, the I: drive, a battery backed RAM allows the application to store dynamic data in files for each transaction and then delete after the transaction is complete. This is highly discouraged in Trident as it will rapidly age the flash beyond the life expectancy of the terminal.

Note that while NAND Flash does have specifications for the number of writes, reads will also erode Flash, albeit a slower rate than the write.

The page/block size in this Nand Flash is 2 KB. When a file is modified the OS writes into a new segment to reduce the "wear" on the Nand Flash.

File Conventions

The following conventions for storing, naming, and opening files are used by the Verix eVo file system. Verix eVo-based terminals also use a set of system files for terminal and application maintenance.

File Storage

Files are stored in non-volatile (battery-backed) RAM or flash memory. For Predator, all files remain in memory even when power is removed from the terminal. Trident files are all in flash memory. Applications should store critical application data to files to preserve data during power failures. Use of the flash file system for application data files should be limited to files that rarely or never change, for example, font files.

Trident terminals support the standard FAT (FAT12, FAT16, FAT32) file system to access files on the external flash drive—M: is the external USB drive and L: is the external SD drive. Long file names are not supported.

If a file is created through the `open()`, the file attribute is always set to `0x20` (`ATTR_ARCHIVE` as defined in the FAT standard). The following basic file I/O functions are supported for these external drives, the first two letters in the file name must be M: or L: and the maximum number of open files on external drive is 10.

- `open`
- `read`
- `write`
- `lseek`
- `close`
- `remove`
- `dir_get_first`
- `dir_get_next`
- `dir_get_file_date`
- `dir_get_attributes` (always returns `ATTR_ARCHIVE` if the file exists).

- `dir_get_file_size`
- `dir_get_file_sz`

Filename

Filenames can be up to 32 characters long and *must* be terminated by a NULL character. Any non-NULL character can be used in the name. Filenames are not case sensitive. For example, TEST, test, TEst, and Test are considered the same file. The following file extensions are used within Verix eVo file system:

NOTE



The RealView® manuals use the `.out` suffix for what the Verix eVo OS calls the `.odb` file. The RealView naming convention is avoided on Verix eVo-based products as other VeriFone products and tools traditionally use `.out` to denote executable program files. See the *Verix eVo Operating System Programming Tools Reference Manual* for more information on the RealView compiling and debugging tools.

- ***.c**: C language source file. Compiled with `armcc` or `tcc`.
- ***.cpp**: C++ language source file. Compiled with `armcpp` or `tcpp`.
- ***.o**: Intermediate object file; valid only as an output file.
- ***.out**: Executable code file; valid only as an output file.
- ***.axf**: Debugger file.
- ***.a**: ARM static library file.
- ***.lib**: Shared library file
- ***.crt**: VeriShield certificate file; transferred to `CERTFILE.SYS` once processed
- ***.p7s**: VeriShield signature file; usually retained in the F: file system once processed

File Groups

Verix eVo-based terminals provide a method for isolating files that belong to one application so that programs in another application cannot update or otherwise gain access to them. Up to *45 groups* of files can be supported.

- Group 0 is reserved for files required by the operating system, such as device drivers.
- Group 1 is the *primary* user group. By default, files download into Group 1. The initial user program to run belongs to Group 1: The `*GO` variable is located in the Group 1 `CONFIG.SYS` file, and the named program *must* be a Group 1 file.

A non-writable file system ("N:") is supported in FLASH. If present, it will be dedicated to Verix eVo files. Every OS supports this new volume, which will only be enabled when the OpSys is installed in a terminal with the required memory space. Updates to the "N:" file system is performed only during OpSys startup as a consequence of downloading new Verix eVo files.

The “N:” file system is visible to all applications. In particular, the “N:” file system includes shared library files intended to be used by other applications. These files will be placed in group 15 in the “N:” file system. The Verix eVo network module will be loaded in “N:” in group 0.

Group 46 is added to support all three file systems (“I:” and “F:” and the new “N:”). Most group 46 files will be placed in “N:” but dynamic data files may be present in both “I:” and “F:” as needed. Code files running in group 46 have access to all other user groups. In contrast, no code file running from any other group will be able to see files in group 46. Group 46 is primarily used to hold the Verix eVo “middleware” files for configuring and managing various networks.

NOTE



Groups 16 - 45 are Verifone private protected groups which are saved for projects in the future.

Group 46 is reserved for EOS files only and requires VeriFone EOS file signing certificates. The move to GID 46 fails if a file is using the correct file extensions but is not signed with the VeriFone EOS file signing certificates.

Update Procedure

Files are never directly downloaded into the “N:” file system or into the user group 46. Instead, they are downloaded into group 1 or group 15. In order to be recognized as Verix eVo files, special naming conventions must be followed. The Verix eVo files will be downloaded with special suffixes to indicate their function. These suffixes are illustrated under the assumption that they will not conflict with any current usage. For example, when upgrading Verix eVo OS, VeriFone may distribute files named “VXEOS.OUT{!” and “SSL.LIB{#” and “TCPPIP.BIN{\$” and these files must be downloaded with these precise names.

NOTE



These special suffixes are used only temporarily to aid in file authentication and installation. Once the installation is complete, these suffixes will not be present.

See *Verix eVo Volume II Operating System and Communication Programmers Guide*, VPN - DOC00302 for more information on VXEOS.OUT and VXCE.out.

Protecting Files From Removal

For GID 1 and 15, file names that start with the “#” character are protected from deletion through the System Mode Clear Memory functions and Vericentre download removal directive with wildcards (for example, `-r *.*`). A file name that starts with the “#” character can only be explicitly removed through `_remove()` API or a file name-specific Vericentre download removal directive (for example, `-r #foo.dat`).

For GIDs 2-14, this same file protection feature can be enabled through the GID 1 *`PROT` variable. In addition, all files in a GID can be protected through the GID 1 *`GUARD` variable.

NOTE

The “#” character is part of the file name. For example, file `#foo.dat` and `foo.dat` can both exist in the same directory but only `#foo.dat` will be protected.

Default System Files

One system file is always present in the memory file system in Group 1:

<code>CONFIG.SYS</code>	Contains all environment variables used by the primary application.
-------------------------	---

User files are created, renamed, and deleted by the executing application program, as required. Verix eVo file system does not impose limits on the number of user files that can be created. However, expansion is limited by memory availability.

NOTE

It is *strongly* recommended that the application *never* attempt to use all “available” memory, since a certain amount of “slack space” is needed for efficient operation. A rule of thumb is to plan on leaving 10% of the total memory unused.

.out Files

When downloading `.out` files, it is important that the terminal performs a soft reset to enable execution of the newly downloaded `.out` file. Call the `SVC_RESTART` function to perform a terminal reset.

NOTE

To ensure that `.out` files execute after downloading, perform a terminal reset.

File Handles

In addition to opening any or all devices, applications can simultaneously open up to 30 user files.

It is possible, and occasionally useful, to open the same file more than once before closing it. Multiple *file handles* allow you to maintain multiple position indicators (seek pointers) within the file. Note that the limitation on the number of open files is based on the number of open handles, regardless of whether two handles happen to represent the same file. This limitation is system-wide. For example, if two applications each have nine files open, then a third application can open no more than twelve files. You can change the maximum value using the `CONFIG.SYS` variable *`FILE`, which can have a value between 10 and 224, default is 30.

Generic Files

Generic files can contain any type of data, making them especially suitable for binary data. Data is accessed by byte address within the file so that any quantity of data can be read or written at any time, from any location within the file. In most applications, a fixed record length is used, typically based on the size of a data structure or union. With careful planning, VLR files can also be used.

Typically, a generic file has a static record size determined by the total amount of space used by a data structure such as:

```
typedef struct MyRecord MYRECORD;
struct MyRecord
{
    int record_type;
    long first_field;
    ...

    int last_field;
};
#define MyRecLen sizeof(MYRECORD)
```

Each record in the file is defined by the structure `MyRecord`; the record length is simply the size of the structure, defined by the constant `MyRecLen`. Using generic files with variable-length records is discussed below.

Variable-Length Record (VLR) Files

VLR files allow data to be stored as records. The first byte in each record is a *count* byte (the number of bytes in the record, including the count byte), followed by data. Each record can have a maximum length of 254 data bytes.

VLR files are particularly suited to ASCII data and can contain arbitrary data, including embedded NULL characters. Data is accessed by record number rather than byte address, and requires that the file be processed from the beginning (first record, then subsequent records searched, until the correct record is found).

VLR files are best suited for chronologically sequenced records with little or no random access requirements.

Compressed Variable-Length Record (CVLR) Files

CVLR files are identical to VLR files with the addition of a compression algorithm applied to the data on reads and writes. This compression converts lowercase characters to uppercase, and stores numeric values (ASCII 0–9) as four bits. The compression and expansion of each record is handled by the file system and is transparent to the application.

NOTE



Byte values greater than 0x5F cannot be correctly translated when this compression is used and should not be included in the file. Access time for CVLR files is somewhat slower than for VLR files. File space savings are especially noticeable on data files containing a lot of numeric data stored as ASCII characters.

Keyed Files

Keyed files (also called *paired* files) are essentially CVLR files that use two records for every data record written. The first record (called the *key*) is followed by a data record. The key gives the data record an alphanumeric name or identifier, providing random access to the records. Keyed file access is considerably slower than VLR or CVLR files, as two records must be read for each data entry and a comparison of the key performed. From a timing performance perspective, keyed files are the least desirable.

Keyed files do have appropriate uses however. Since keyed files can be edited through the terminal keypad, they are ideal for holding data that requires modification, such as download parameters (telephone numbers and terminal ID), and customer-specific information, such as a customer address.

The `CONFIG.SYS` file is a keyed file that can hold downloaded parameters and can be accessed from VTM.

Keyed files can also create a database where information could be located by a key, such as an account number. Care must be taken when using keyed files in applications where speed and performance are critical.

NOTE

For increased performance, minimize the use of keyed files. After downloads, move frequently used data from `CONFIG.SYS` to more efficient files.

The following system calls operate on keyed files:

- `get_env()`
- `put_env()`
- `getkey()`
- `putkey()`

Variable-Length Records

Basic file input/output (I/O) in Verix eVo follows the Posix model in which files are treated as unstructured sequences of bytes. Verix eVo file system also supports record-structured I/O, which allows files to be accessed as a sequence of logical records. Functions are provided to write, read, insert, delete, and seek to records.

Records are stored as a one byte length followed by the data. The length includes itself so it is one greater than the number of data bytes. Since the maximum value which can be stored in a byte is 255 a record can contain at most 254 data bytes. Zero length records are allowed (however it can be tricky to distinguish reading a zero length record from an end of file).

Variable-length records (VLRs) are an access method, not a file type. Although it is common to refer to VLR files, the file system does not distinguish between file written as records and files written as bytes. In fact, it is possible to mix I/O methods within a single file. For example, a file could have a fixed length header written by `write()` followed by a set of records written by `write_vlr()` (see [write\(\)](#)). The application is responsible for ensuring that the file is positioned to the start of a record before calling record-oriented functions.

Compressed Variable-Length Records

Compressed variable length records are a special case of variable length records. The file structure is the same; the difference is that the data is compressed to save space. The length byte at the start of each record is the number of bytes of compressed data, not the original length. The uncompressed data size is restricted to 254 bytes. Attempting to write 300 bytes is an error even if it compresses to 200 bytes.

The compression algorithm is designed for data which consists mostly of numeric digits. Basically it replaces each decimal digit by a 4 bit nibble. Non-numeric characters are encoded as two nibbles, the first of which is in the range hexadecimal A – F to distinguish it from numeric digits. The encoding is shown in the following table. The two nibbles may be in the same byte or in two different bytes, depending on the alignment. If the compressed data contains an odd number of nibbles, the last byte is padded with hex F. Data containing only numeric digits is compressed to half its size; data containing no digits do not shrink at all.

For example: The 7-character string “\$642.98” is represented in ASCII as

24	36	34	32	2E	39	38
----	----	----	----	----	----	----

This compresses to 9 nibbles which are stored in 5 bytes:

C4	64	2C	E9	8F
----	----	----	----	----

Because several non-numeric characters are encoded identically the compression is not fully reversible. When the data is decompressed each non-numeric character expands to its first occurrence in the table. The result is that values greater than 0x5F are changed to different characters. For example, an open parenthesis (ASCII code 7B) is encoded as FB, which decodes to an open bracket (ASCII 5B) when decompressed. Lowercase letters are changed to uppercase.

File Access Function Calls

With the exception of keyed files, all file access methods use the same set of basic function calls. The method of access is determined by parameters passed to these functions. In general, each function returns a specified value ≥ 0 on success or a value of -1 on failure. On failure, `errno` is set to a specific error code to indicate the failure.

Open Files

The `open()` function uses attributes to accomplish the following:

- Create new files
- Open existing files for writing
- Position a seek pointer within an existing file

Create New Files

To create a new file, the `O_CREAT` attribute must be included in the call to `open()` specify `O_WRONLY` or `O_RDWR` to update the new file.

If the combination `O_CREAT | O_TRUNC` is used, an empty file is guaranteed. If you do not wish to write to the file following its opening, omit the `O_RDWR` or `O_WRONLY` flags and close the file immediately.

If you call `open()` with `O_CREAT` and specify an existing filename, `O_CREAT` is ignored and the file is simply opened using any additional read/write attributes specified in the call. In this case, a write operation may overwrite existing data in the file—a potentially destructive situation. To ensure that a new file is actually created, include the `O_EXCL` attribute in the `open()` call. If the file exists, an error value returns to the application with `errno` set to `EEXIST`.

To create a log file to record activity, specify the attributes `O_WRONLY | O_APPEND | O_CREAT` with the `open()` call. Log files do *not* have seek activity during recording.

**TIP**

When adding or deleting data from a file, it is important to remember that any files or data stored in memory after this file may move. While normally imperceptible, the amount of time required to perform this move increases with the amount of data being moved.

Place frequently updated files after large data-storage files and limit the number and frequency of operations that change the size of a file. When updating records of the same length, overwrite the previous data rather than deleting the old record and writing the new data. Fixed length records and padding can prevent changing the size of file records during management operations. This should be carefully considered if the overhead significantly impacts transaction storage requirements.

Open Files for Writing

Write access to a file is not implied. It must be specifically requested in the `open()` function by passing `O_WRONLY` or `O_RDWR`. `O_APPEND` can only be used in conjunction with a file that has requested write access.

Seeking with `lseek()`, `seek_cvlr()`, and `seek_vlr()` can move the file pointer away from the end of a file; however, each write to a file opened with `O_APPEND` performs a *seek to the end* before writing data. Files opened for `O_APPEND` have their seek pointers moved to the end of the file.

File Positioning

Positioning within a file is accomplished using an internal seek pointer—a long integer value—maintained by the file system that contains the byte or record address to use in the next `read()`, `write()`, `insert()`, or `delete()` operation. The seek pointer is allocated when the file is first opened and is therefore unique per handle. When a file is first opened, the seek pointer is set to a known state (typically zero) at the beginning of the file. As noted earlier, if `O_APPEND` is specified in the `open()` call, the seek pointer is positioned at the end of the file. Applications can modify the seek pointer using the `lseek()`, `seek_vlr()`, and `seek_cvlr()` functions.

Table 11 lists the file access function calls discussed in this chapter.

Table 11 File Access Function Calls

API Function	API
Open File	
	open()
Read Files	
	read()
	read_vlr()
	read_cvlr()
Write Files	
	write()
	write_vlr()
	write_cvlr()
File Positioning	
	lseek(), seek_vlr(), and seek_cvlr()
Insert and Delete Data	
Note: When adding or deleting data from a file, it is important to remember that any files or data stored in memory after this file may move.	insert(), insert_vlr(), and insert_cvlr() delete(), delete_vlr(), and delete_cvlr() delete_()
Place frequently updated files after large data-storage files and limit the number and frequency of operations that change the size of a file. When updating records of the same length, overwrite the previous data rather than deleting the old record and writing the new data.	
Retrieve File Information	
	get_bundle_id()
	get_file_size()
	get_file_date()
	get_protocol_string()
	SVC_CHECKFILE()
	_remove()
Lock and Unlock Files	
Note: Attempts to open or access a locked file fail with an <code>EACCESS</code> error. Note that locks are associated with handles, not tasks. The restrictions apply to the task which called lock as well as to other tasks. The lock can be removed by calling unlock or closing the handle.	lock() unlock()

Table 11 **File Access Function Calls** (continued)

API Function	API
Rename a File	<code>_rename()</code>
File Attribute Function Calls	<code>get_file_attributes()</code> <code>get_file_max()</code> <code>reset_file_attributes()</code> <code>set_file_attributes()</code> <code>set_file_max()</code>
Close Files	<code>close()</code>
Keyed Files Reads and Writes	<code>getkey()</code> <code>putkey()</code>
File Directory Function Calls	
Note: The file system uses a non-hierarchical directory. This means that there is no support for subdirectories. Filenames can be up to 32 characters long and must be terminated by a NULL character.	<code>dir_get_all_attributes()</code> <code>dir_get_attributes()</code> <code>dir_get_file_date()</code> <code>dir_get_file_size()</code> <code>dir_get_file_sz()</code> <code>dir_get_first()</code> <code>dir_get_next()</code> <code>dir_get_sizes()</code> <code>dir_put_file_date()</code> <code>dir_reset_attributes()</code> <code>dir_set_attributes()</code> <code>file_copy()</code> <code>put_file_date()</code> <code>SVC_RAM_SIZE()</code> <code>unzip()</code>
Flash File	<code>dir_flash_coalesce()</code> <code>dir_flash_coalesce_size()</code> <code>SVC_FLASH_SIZE()</code>
Subdirectories and Groups	<code>mkdir()</code> <code>chdir()</code> <code>rmdir()</code> <code>getcwd()</code>

open()

Allocates and returns an integer file handle used in all subsequent file operations. Before a file can be accessed, it must be open.

A single file can be opened multiple times (up to 30 files can be open simultaneously; multiple opens of the same file are included). The number of files that can be open is set by the `*FILE` variable in the `CONFIG.SYS` file.

The handle is owned only by the process which calls the `open()` function. Only the task or thread which opens the file or device will have access to the `open()` function. Although `open()` handle is stored in the memory that is available to any other threads, only the task (or thread) that opened the file or device will have access to it.

Each call to `open()` returns a unique handle with access attributes specified by that `open()`. Thus, a file can have multiple seek pointers in different locations in the file. The programmer is responsible for the consequences of adding or deleting data from a file that has been opened multiple times. The integrity of the file is maintained, but in some cases it may be difficult to predict where the seek pointers are positioned. Since file handles are a limited resource, care should be taken in their allocation and use.

`open()` also allocates a pipe control block and assigns the specified name. Only the first eight characters of the name are significant. The name is useful to other tasks that need to connect to this pipe. For an example, see the section on `get_owner()`. Sometimes the name is not important. In this case, the application can utilize an anonymous pipe by opening a pipe with the special name "P:" as the first parameter.

Pipe names must be unique. Use of P:PIPE is not recommended as the `get_owner()` call only detects the first use. Most pipes can be opened anonymously. Server tasks (for example, a print server) are the primary users of this feature.

There are two ways to identify the pipename:

- Statically defined: In the form "P:name," where name is the name (an ASCII string of up to 16 bytes) that uniquely identifies the pipe.
- Dynamically defined: Set to "P:" with no numeric suffix. In this case the OS assigns an unused pipe number. This is sometimes referred to as an anonymous pipe. The pipename parameter is not case sensitive. Only the first 16 characters of the name (not counting P:) are significant.

`open()` also returns the handle for writing (or reading or closing) the console device. This function must be called *before* any other functions can be called. The following operations are performed on calls to the `open()` function:

- Sets the font to the default
- Clears the display
- Clears the key buffer
- Turns off the cursor
- Sets the contrast from the default to the contrast setting in the console's MIB area, which contains the last user setting
- Sets window to the complete screen
- Turns backlight on

`open()` prepares the firmware to accept and store card reads. If the programmer does not make this call, all card reads are ignored by the terminal. Reopening the magnetic card device without an intervening `close()` succeeds and does not clear any pending card data.

This function call explicitly opens the clock/calendar device or sound-generating device, returning its associated device handle. It is *not* necessary to open the clock/calendar device just to read the current time, since any application can always call `read_clock()`. The only time the clock must be opened is to write to it.

Invoking `open()` explicitly opens the beeper's sound-generating device, returning its associated device handle. The beeper does not need to be explicitly opened. It is a shared device and any application can call the functions `normal_tone()` and `error_tone()`. The only advantage to opening the beeper is to prevent other applications from using it.

This function call also opens the printer device. On success, the printer handle is returned, and this handle can be used for `read()`, `write()`, `close()`, and other APIs. If an application opens the printer device when the VX 820 is connected to a DUET base station, the OS returns a valid handle.

NOTE

The VX 820, Vx805, Vx810 DUET terminals are used in conjunction with a base unit that contains a printer. This printer is treated just like an internal printer with the exception of added checks for the device being connected via USB from the base to the terminal.

If a printer is connected to the VX 820/Vx805/Vx810 DUET via the communication port, the OS has no knowledge of this device as a printer and it is up to the application to provide all printer functions.

This function call opens the IPP. On success, the IPP handle is returned and this handle can be used for `read()`, `write()`, and `close()`. The pointer must point to the string `"/dev/com5"`. On VX 820, returns -1 with several values of `errno`.

This function call opens the USB device. The API only succeeds in a single USB device mode if no external USB devices are currently open and the requested device is connected to a port that is powered on. In a multi-USB-device mode is requested, the API succeeds if the device is currently connected to a port that is powered on.

`open()` prepares the asynchronous RS-232 interface for operation. The port remains inactive until a call to `set_opn_blk()` is made to initialize the port with the selected baud, format, and protocol parameters.

This function also returns the handle for reading, writing, or closing the Ethernet or WiFi device.

On VX 680 3G, the PCM driver has a standard Verix `open()` API. Only one task may open/own this device at any time.

On BT, this function gives access to the SPP port connection created by the `DEV_ETHERNET` or `DEV_ETHERNET1` or `DEV_BLUETOOTH` or `DEV_BLUETOOTH1` device.

Prototype

```
int open (const char *filename, int flags);  
handle = open("/dev/pcm",0);
```

Parameters

filename	Pointer to a NULL-terminated string, up to 32 bytes long.	
flags	An integer that indicates the access attribute:	
	<code>O_RDONLY</code>	Opens the file for read-only access; the file cannot be written.
	<code>O_WRONLY</code>	Opens the file for write-only access; the file cannot be read.
	<code>O_RDWR</code>	Opens the file for read/write access; records can be read, written, inserted, or deleted.
	<code>O_APPEND</code>	Opens the file with the file position pointer initially set to the end of the file.
	<code>O_CREAT</code>	Opens a new file for write access.
	<code>O_TRUNC</code>	Opens an existing file, truncates its length to zero, deletes previous contents.
	<code>O_EXCL</code>	Returns error value if the file already exists; used with <code>O_CREAT</code> .
	<code>O_CODEFILE</code>	Specifies that the file contains executable code. Normally this attribute is used by the download module and is not used by applications.
	<code>O_SINGLE_DEV</code> <code>O_MULTI_DEV</code>	The device is to be opened in single-USB-device mode. This has no effect if used on a non-USB device. <code>O_MULTI_DEV</code> : also defined (as 0x0) and can be used to open USB devices in multi-USBdevice

Note: O_SINGLE_DEV and O_MULTI_DEV flags are from the USB secure mode. While secure mode is no longer supported in Trident some legacy still exists but can be ignored. Trident does not support secure mode at all. Only Vx670 supports secure mode in Predator. But, we do not push for the use of secure mode as it has limited usefulness.

Return Values

Success: A positive integer is a handle that can be used for subsequent access to the file: read, write, and so on.

Failure: -1 with errno set to ENODEV.

read()

This function call requires a valid handle to be open. A successful call copies up to `len` bytes from the file to the address specified by `buf`.

This function reads data from the open pipe handle. In a pipe handle if `len` is set to 0, 0 is also returned. If `len` is greater than 0 and the pipe is a character pipe, read from this handle's read FIFO either until `len` bytes have been read or until the FIFO is empty. If it is a message pipe, read the next message and transfer either the entire message or as much as fits into the passed buffer. In both cases, signal a normal completion trap to the other handle associated with this pipe.

Ensure that there is enough buffer space available to contain your message. The return value is the number of bytes read. Note that if the pipe is a message type and the message is bigger than the size of the passed buffer, only `count` bytes of the message are returned and the remainder of the message is lost. If this pipe has been configured using `pipe_init_msgX`, then the first 16 bytes of the input message will include the pipe header as described on [Pipe Header Format](#).

NOTE



It should be common programming practice to only execute when there is something to do. Polling a device or a PIPE is not recommended. Execute a `wait_event()` and wait for the `EVT_PIPE` event until data is ready in the PIPE the application. The `EVT_PIPE` event is sent by the OS when data is placed in the pipe.

In a console device, `read()` allows the current owner to retrieve the keys in the key buffer, whether secure or not. In general, all keypresses go into the key buffer until it is full; at which point, new keypresses are discarded without notification or error beep. The exceptions are VTM entry keys and the hot key. The size of the key buffer is 20 characters.

In MCR, each invocation of `read()` transfers data from a card reader swipe into the buffer. Only one card swipe can be queued at a time. If a new card swipe occurs before a previous swipe has been `read()`, the new swipe is discarded. Also, any card swipes that occur while the magnetic device is closed are discarded. Also, any card swipes that occur while the magnetic device is closed are discarded. The data returned in `len` is in the following format:

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 decoding channel 1 data

`d1` = Any data decoded successfully (may not exist)

`c2` = A one-byte size of `c2+s2+d2`

`s2` = A one-byte status of decoding channel 2 data

`d2` = Any data decoded successfully (may not exist)

`c3` = A one-byte size of `c3+s3+d3`

`s3` = A one-byte status of decoding channel 3 data

`d3` = Any data decoded successfully (may not exist)

Note that if the buffer supplied by the caller is too small, the card data must be truncated. This truncation is performed as follows:

The status and length bytes for each track are always returned. If a size argument of 6 is supplied (minimum), the following returns:

c1	s1	c2	s2	c3	s3
----	----	----	----	----	----

where, `c1`, `c2`, and `c3` are all set to 2.

If there is additional space in the buffer, as much track 1 data are returned as space allows. If there is additional space after all track 1 data are stored, as much track 2 data are returned as space allows. If there is still space allows after track 1 and track 2 data are stored, as much track 3 data are returned as space allows.

Some cards may contain no data for a given track (or a given track may not be supported by the reader), this leaves more space in the buffer for tracks that contain data.

The `status` byte can have one of the following values:

Status	Symbol	Meaning
0	MAG_NOERR	Valid data
1	MAG_NODATA	No data
2	MAG_NOSTX	Missing start sentinel or insufficient data
3	MAG_NOETX	Missing end sentinel or excessive data
4	MAG_BADLRC	Missing LRC or LRC error
5	MAG_PARITY	Parity error
6	MAG_REVETX	Reversed end sentinel found
7	MAG_BADJIS	Insufficient printable characters
8	MAG_BADTRK	Insufficient characters

For each track, a specific list of decode tables is used in the attempt to decode the track data. The track data and the address of the first entry in the decode table are passed to a low-level decoder. This decoder first attempts to decode the data in the forward direction. If decoding fails, the decoder then attempts to decode the data in the reverse direction. If both attempts fail, the `status` byte for the last attempt (reverse decode, in this case) returns, and the procedure repeats for the next entry in the list of decode tables. If the list is exhausted without a successful decode, the `status` byte of the final decode attempt returns. If at any time a decode attempt is successful, the decoder sets a flag to lock the direction of decode for the remaining channels and then exits the procedure, returning the success code of `MAG_NOERR`.

This procedure is repeated for the remaining tracks until all three are processed.

As each track is processed, the results (ci, si, di) are appended to buffer.

The order of the decode tables is set to the following sequence:

ISO 7811, AAMVA, JIS B 9561, CA DL/ID

For track 1, the list of decode tables contains information for the following entries:

ISO 7811 track 1, AAMVA track 1, CA DL/ID track 1

For track 2, the list of decode tables contains information for the following entry:

ISO 7811 track 2

For track 3, the list of decode tables contains information for the following entries:

ISO 7811 track 3, AAMVA track 3, JIS B 9561 Type II (front track), CA DL/ID track 3

Note that several decoding passes can be attempted for each track, but the return format allows only one error code (in the `status` byte) to be returned per track. Which decoding pass error code returns for bad tracks depends on the internal details of the device driver and may be subject to change.

NOTE



JIS B 9561 Type I track 1 is the same as ISO 7811 track 1. AAMVA track 2 and JIS B 9561 Type I track 2 are the same as ISO 7811 track 2; CA DL/ID track 2 is compatible with (but holds less data than) ISO 7811 track 2.

In a Real-Time clock, `read()` places the system date, time, and day of the week in an application buffer as a 15-byte ASCII character array (not a NULL-terminated string). The 15-byte ASCII character array is returned in `*buf` in the format `yyyymmddhhmmssd`.

`read()` retrieves input from the printer device. The printer supports various escape commands that generate responses. When the internal printer receives a command that requires a response, the response is stored and a COM4 event is generated. The printer response to an escape command is read through the `read()` API.

`read()` also transfers data from the card reader scan into the buffer.

Each invocation of `read()` transfers data from the internal port into the buffer and returns the number of bytes actually read or zero if no data are available.

`read()` also allows the user to read one packet (maximum of 1514 bytes) from the Ethernet or WiFi device. Any count less than 1514 is considered to be an error, the result will be -1 with `errno` set to `EINVAL`.

On VX 680 3G, this function call always returns 0. No input and no events are returned because `/dev/pcm` is an output only device.

On BT, this function reads characters received from the SPP port. This function also reads a received Ethernet packet.

Prototype

```
int read (int hdl, char *buf, int len);
```

Parameters

hdl	Handle of the calling device.
buf	Pointer to an array where the data is stored.
len	Determines the maximum value to read.

Return Values

Failure:	<ul style="list-style-type: none">–1 with errno set to <code>EBADF</code>: file not open (bad handle) or file currently locked by another user.–1 with errno set to <code>EACCES</code>: caller's buffer is not writable (for example, bad buffer address).–1 with errno set to <code>EINVAL</code>: the count parameter is too large.
Success:	Return value shows the actual number of bytes placed in the buffer.

NOTE

The keypad driver makes no distinction between the keypad buffer for secure and non-secure keys. Secure keys are keys used during getpin operations: 0-9, ENTER, BKSP and CANCEL, all other keys are non-secure. Secure keys can be differentiated from non-secure keys if the high order bit is set to 0x80. All keypad reads should remove the high order bit before trying to see what key is read. The getpin operation takes over the keypad during getpin operations so secure key designation has limited use for applications.

read_vlr()

A successful call copies up to `len` bytes from the file to the address specified by `buf`.

Prototype `int read_vlr (int hdl, char *buf, int len);`

Parameters

<code>hdl</code>	Handle of the calling device.
<code>buf</code>	Pointer to an array where the data is stored.
<code>len</code>	Determines the maximum value to read.
<code>read_vlr()</code>	Points to the next variable-length record in the file.

Return Values

Failure:	<ul style="list-style-type: none">–1 with <code>errno</code> set to <code>EBADF</code>: file not open (bad handle) or file currently locked by another user.–1 with <code>errno</code> set to <code>EACCES</code>: caller's buffer is not writable (for example, bad buffer address).–1 with <code>errno</code> set to <code>EINVAL</code>: the count parameter is too large.
Success:	Return value shows the actual number of bytes placed in the buffer; this value may be smaller if the end of file is encountered before the read reaches the <code>count</code> value.

read_cvlr()

A successful call copies up to `len` bytes from the file to the address specified by `buf`.

Prototype `int read_cvlr (int hdl, char *buf, int len);`

Parameters

<code>hdl</code>	Handle of the calling device.
<code>buf</code>	Pointer to an array where the data is stored.
<code>len</code>	Determines the maximum value to read.
<code>read_cvlr()</code>	Points to the compressed variable-length record in the file.

Return Values

Failure:	<ul style="list-style-type: none">–1 with <code>errno</code> set to <code>EBADF</code>: file not open (bad handle) or file currently locked by another user.–1 with <code>errno</code> set to <code>EACCES</code>: caller's buffer is not writable (for example, bad buffer address).–1 with <code>errno</code> set to <code>EINVAL</code>: the count parameter is too large.
Success:	Return value shows the actual number of bytes placed in the buffer; this value may be smaller if the end of file is encountered before the read reaches the <code>count</code> value.

write()

A successful call to these functions copies up to `len` bytes from the buffer into the file.

This function call writes data to the open pipe. For a character pipe, write `len` bytes or as many as fit to this handle's write FIFO. For a message pipe, allocate memory of count size, write the buffer to it, and add this message to the FIFO. The return value is the number of bytes written.

It also writes buffer to display. The characters stored in `buf` are retrieved from the current font and written to the display window. Unless `scroll_mode()` has been called, the write wraps within the current window and truncates at the end of the window.

If the current font is the default font, the following characters reposition the cursor:

- Line feed (`\n`): Move cursor to start of next row. ignore if on last row
- Form feed (`\f`): Clear window and move cursor to start of window
- Carriage return (`\r`): Move cursor to start of same row.
- Backspace (`\b`): Move cursor back one. Ignore if at top line, leftmost column

The cursor is always positioned at the next character after the end of the text. If the cursor falls outside the window, the cursor is not visible.

The Real-Time Clock sets the system date and time, the ASCII digit format should be the same as `read()`, except that only 14 bytes are actually used—the 15th byte is determined by the system. Errno value `EINVAL` is returned when the array has improper formatting. Both date and time are validated to ensure proper formatting. For example, if the array is not all ASCII digits or if the date is February 29 in a non-leap year, the clock does not update and a result of `-1` is returned with errno set to `EINVAL`.

This function call writes to the printer. Data written to the printer can include special printer control codes and escape sequences. See [Special Items](#) for more information.

This function also 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`.

`write()` transfers data from an application buffer into the device driver's buffer, only if the latter is empty. Once in the device buffer, the data are transferred to the transmitter each time the transmit buffer goes empty. This API also allows the user to write one packet (60 - 1514 bytes) for the Ethernet or WiFi device.

To play PCM sounds on a VX 680 3G terminal, applications write PCM samples to the `/dev/pcm` device using the `write()` API. Before writing to `/dev/pcm`, the application must set the audio parameters using the `set_pcm_blk()` API. The parameters set with `set_pcm_blk()` determine the format of the samples expected by the `write()` API. `write()` returns either a positive number of bytes accepted, or a negative error indication.

The `write()` API buffers the PCM samples into an internal audio FIFO. If the write request is larger than the space available in the FIFO, the write request will not write any data into the FIFO and fail with a `ENOSPC` error code. This allows the application to re-try the same write request later when FIFO space frees up.

If `/dev/pcm` is set for 8 bit mono mode, the driver accepts any length of bytes that fits into the FIFO. In 8 bit mode, every byte counts as one sample in the FIFO. If `/dev/pcm` is set for a 16 bit mono mode, the driver only accepts even numbers of bytes. Writes of odd numbers of bytes causes `write()` to fail with an `EINVAL` error code. The driver counts every two bytes as one FIFO sample. The sample format can be set for little endian or big endian samples.

If `/dev/pcm` is set for 16 bit stereo, the driver only accepts multiples of 4 bytes. Writes of non multiples of 4 bytes causes `write()` to fail with an `EINVAL` error code. The driver counts every four bytes as one FIFO sample.

On BT, this function sends characters over the SPP port. This function also writes an Ethernet packet to send.

Prototype

```
int write (int hdl, const char *buf, int len);
```

Parameters

hdl	Handle of the calling device.
buf	Pointer to an array where the data is stored.
len	Determines the maximum value to write.

Return Values

Failure:	<p>–1 with <code>errno</code> set to <code>EBADF</code>: File not open (bad handle) or file currently locked by another user. In pipe, this means Pipe not owned by caller or is not configured or not enough memory to allocate the message (message pipes only).</p> <p>–1 with <code>errno</code> set to <code>ENOSPC</code>: Not enough memory to complete the request.</p> <p>–1 with <code>errno</code> set to <code>EACCES</code>: Caller's buffer is not readable (for example, bad buffer address).</p> <p>–1 with <code>errno</code> set to <code>EINVAL</code>: <code>len</code> parameter is too large.</p> <p>–1 with <code>errno</code> set to <code>EFBIG</code>: File expansion area is full. See set_file_max().</p> <p>–1 with <code>errno</code> set to <code>ENOMEM</code>: Not enough memory to allocate for the FIFOs.</p>
----------	--

–1 and `errno` set to `EPIPE`: Pipe not connected or connected to a task that exited. –1 and `errno` set to `ENOSPC`: Pipe is full, or (message pipes only) not enough system buffers available to complete the operation.

–`EINVAL`: This function is not supported on Metrologic Barcode device.

Success: Shows the actual number of bytes written.

NOTE

The OS will automatically send `EVT_PIPE` to the task receiving the data.

NOTE

The `write()` operation is a non-block call. `write()` returns immediately and the actual writing of the data to memory is performed by the OS in the background. This operation is guaranteed to complete even over power failures.

write_vlr()

A successful call copies up to `len` bytes from the buffer into the file.

`write_vlr()` and `write_cvlr()` either create new records in the file or overwrite existing records, depending on the position of the seek pointer.

If the file was opened with the `O_APPEND` attribute bit set, all writes are done at the end of the file, regardless of prior calls to `lseek()`, `seek_vlr()`, or `seek_cvlr()`. When a read function follows a seek function, all data at that file location transfers to the application's buffer. `O_APPEND` means *always* append.

Prototype

```
int write_vlr (int hdl, const char *buf, int len);
```

Parameters

<code>hdl</code>	Handle of the calling device.
<code>buf</code>	Pointer to an array where the data is stored.
<code>len</code>	Determines the maximum value to write.
<code>write_vlr()</code>	Point to the next record in the file. If the seek pointer is positioned within the file (that is, when overwriting existing data), then the file manager assumes that the intention is to overwrite an existing VLR record at this point. The file manager reads the byte at the current location to determine the size of the existing record, deletes this record, then replaces it with the new VLR record.

Return Values

Failure:	<p>–1 with <code>errno</code> set to <code>EBADF</code>: File not open (bad handle) or file currently locked by another user.</p> <p>–1 with <code>errno</code> set to <code>ENOSPC</code>: Not enough memory to complete the request.</p> <p>–1 with <code>errno</code> set to <code>EACCES</code>: Caller's buffer is not readable (for example, bad buffer address).</p> <p>–1 with <code>errno</code> set to <code>EINVAL</code>: <code>len</code> parameter is too large.</p> <p>–1 with <code>errno</code> set to <code>EFBIG</code>: File expansion area is full. See set_file_max().</p>
Success:	Shows the actual number of bytes written.

write_cvlr()

A successful call copies up to `len` bytes from the buffer into the file. `write_vlr()` and `write_cvlr()` either create new records in the file or overwrite existing records, depending on the position of the seek pointer.

If the file was opened with the `O_APPEND` attribute bit set, all writes are done at the end of the file, regardless of prior calls to `lseek()`, `seek_vlr()`, or `seek_cvlr()`. When a read function follows a seek function, all data at that file location transfers to the application's buffer. `O_APPEND` means *a*lways append.

Prototype

```
int write_cvlr (int hdl, const char *buf, int len);
```

Parameters

<code>hdl</code>	Handle of the calling device.
<code>buf</code>	Pointer to an array where the data is stored.
<code>len</code>	Determines the maximum value to write.
<code>write_cvlr()</code>	Point to the next record in the file. If the seek pointer is positioned within the file (that is, when overwriting existing data), then the file manager assumes that the intention is to overwrite an existing CVLR record at this point. The file manager reads the byte at the current location to determine the size of the existing record, deletes this record, then replaces it with the new CVLR record.

Return Values

Failure:	<p>–1 with <code>errno</code> set to <code>EBADF</code>: File not open (bad handle) or file currently locked by another user.</p> <p>–1 with <code>errno</code> set to <code>ENOSPC</code>: Not enough memory to complete the request.</p> <p>–1 with <code>errno</code> set to <code>EACCES</code>: Caller's buffer is not readable (for example, bad buffer address).</p> <p>–1 with <code>errno</code> set to <code>EINVAL</code>: <code>count</code> parameter is too large.</p> <p>–1 with <code>errno</code> set to <code>EFBIG</code>: File expansion area is full. See set_file_max().</p>
Success:	Shows the actual number of bytes written; this value may be smaller when using <code>write_cvlr()</code> , as <code>write_cvlr()</code> uses compression.

lseek()*, *seek_vlr()*, and *seek_cvlr()

To position the seek pointer within a file, pass the start location and an offset (long integer) value to the function *lseek*, *seek_vlr()*, or *seek_cvlr()*, depending on which access method is used. Start locations can be:

- `SEEK_SET` — Beginning of file
- `SEEK_CUR` — Current seek pointer location
- `SEEK_END` — End of file

If `SEEK_SET` or `SEEK_END` is used, the system moves the seek pointer to this location and then moves it again, based on the offset value. If `SEEK_CUR` is used, the pointer is moved from its current location by the `offset` value.

Prototype

```
int lseek(int hdl, long offset, int origin);
int seek_vlr (int hdl, long offset, int origin);
int seek_cvlr (int hdl, long offset, int origin);
```

Parameters

<code>hdl</code>	Handle of the calling device.
<code>offset</code>	Specifies the number of bytes to move the seek pointer from the specified starting point; used with <i>lseek()</i> ; can be positive or negative. <i>lseek()</i> can be positive or negative, and it specifies the number of bytes to seek forward or backward from the specified origin. For the functions <i>seek_vlr()</i> and <i>seek_cvlr()</i> the offset value <i>must</i> be positive, and it specifies the number of records to seek into the file.
<code>origin</code>	Value <code>SEEK_SET</code> , <code>SEEK_CUR</code> , or <code>SEEK_END</code> as defined in <code>SVC.H</code> .

NOTE



Backwards seeks in record files are not supported.

Return Values

The return value from these functions is the *absolute number of bytes* (not records for *seek_vlr()* and *seek_cvlr()*) from the beginning of the file. In a generic file, this value coincides with the pointer position in the file. For other file types, this value is meaningless because it also counts bytes (which include record headers) instead of records. Fixed-length records can be randomly accessed using the record number as a key. The byte address passed to *lseek()* is simply the number of records multiplied by the size of each record.

Failure	-1 with <code>errno</code> set to <code>EBADF</code> : file not open (bad handle), or file currently locked by another user -1 with <code>errno</code> set to <code>EINVAL</code> : The origin is not <code>SEEK_SET</code> , <code>SEEK_CUR</code> , or <code>SEEK_END</code> , <code>offset</code> is too large, or (for <i>seek_vlr</i> or <i>seek_cvlr</i>) <code>offset</code> is negative.
---------	--

Example

```
bytes=lseek(handle,4L,SEEK_SET);
```

insert(), insert_vlr(), and insert_cvlr()

A successful call to these functions inserts up to `size` bytes from the buffer into the file. The file position pointer is moved to the end of the inserted data.

Prototype

```
int insert(int hdl, const char *buf, int size);
int insert_vlr (int hdl, const char *buf, int size);
int insert_cvlr (int hdl, const char *buf, int size);
```

Parameters

<code>hdl</code>	Handle of the calling device.
<code>buf</code>	Pointer to an array where the data is stored.
<code>size</code>	Determines the maximum value to insert; used with the <code>insert_cvlr()</code> function.

Return Values

<code>bytes_inserted</code>	Return value shows actual number of bytes inserted; this value can be smaller because <code>insert_cvlr()</code> uses compression.
Failure:	<p>–1 with <code>errno</code> set to <code>EBADF</code>: File not open (bad handle) or file currently locked by another user.</p> <p>–1 with <code>errno</code> set to <code>ENOSPC</code>: Not enough memory to complete the request.</p> <p>–1 with <code>errno</code> set to <code>EACCES</code>: Caller's buffer is not readable (for example, bad buffer address).</p> <p>–1 with <code>errno</code> set to <code>EINVAL</code>: <code>count</code> parameter is too large.</p> <p>–1 with <code>errno</code> set to <code>EFBIG</code>: File expansion area is full. See set_file_max().</p>

delete(), delete_vlr(), and delete_cvlr()

Deletes data from a file opened for write access at the location of the file position pointer. Any data following the deleted data is moved to fill the gap. The file position pointer is not modified by these functions. If fewer than `count` bytes follow the current position, all data to the end of file is deleted. The file size shrinks by the number of deleted bytes. The call is not valid for pipes or devices.

Prototype

```
int delete(int hdl, unsigned int count);
int delete_vlr (int hdl, unsigned int count);
int delete_cvlr (int hdl, unsigned int count);
```

Parameters

<code>hdl</code>	Handle of the calling device.
<code>count</code>	For <code>delete()</code> : The number of bytes to delete. For <code>delete_vlr()</code> and <code>delete_cvlr()</code> : The number of VLR (or CVLR) records to delete. The size of each record is read from the file.

Return Values

Success:	0
Failure	-1 and <code>errno</code> set to <code>EBADF</code> : file not open (bad handle), or file currently locked by another user. -1 and <code>errno</code> set to <code>EINVAL</code> : Invalid <code>count</code> value (negative).

delete_()

Identical to `delete()`, but providing an alternate name for the function that does not conflict with the C++ *delete* keyword.

Prototype `int delete_(int hdl, unsigned int count);`

Parameters

`hdl` Handle of the calling device.
`count` Determines the maximum value to delete.

Return Values

Success: 0
Failure -1 and `errno` set to `EBADF`: Invalid handle.
 -1 and `errno` set to `EINVAL`: Invalid `count` value (negative).

get_bundle_id()

Copies the Apple `bundle_id` string into the given buffer.

Prototype `int get_bundle_id(int hdl, char *buf);`

Parameters

`hdl` Device handle
`buf` Buffer pointer where the `bundle_id` is returned.

Return Values

Success: 0
Failure: -1: `EINVAL`

get_file_size()

This call retrieves the file size as an integer returned in the indicated buffer. The file must be open *before* calling this function.

Prototype `int get_file_size(int hdl, long *filesize);`

Parameters

<code>hdl</code>	Handle of the calling device.
<code>filesize</code>	The value returned reflects the size of the internal file header (currently 64 bytes), the amount of user data currently written, and the expansion area currently allocated for the file.

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : File not open. -1 with <code>errno</code> set to <code>EACCES</code> : The buffer specified in <code>filesize</code> is not writable.

get_file_date()

Returns information about the last update to the file. The file must be open *before* calling this function.

Prototype `int get_file_date(int hdl, char *yymmddhhmmss);`

Return Values

Returns a 12-byte timestamp in `buf`. The timestamp contains the date and time the file was last modified in the format: *yymmddhhmmss*, where:

- *yy* = year
- *mm* = month
- *dd* = day
- *hh* = hour
- *mm* = minutes
- *ss* = seconds

get_protocol_string()

Copies the Apple `bundle_id` string into the given buffer. This is a V*600 only function.

Prototype `int get_protocol_string(int iap_channel, char *buf);`

Parameters

<code>iap_channel</code>	Host channel ID
<code>buf</code>	Buffer pointer where the host string is returned.

Return Values

Success:	0
Failure:	-1: EINVAL

SVC_CHECKFILE()

Calculates the checksum for the specified file and compares it with the value stored in the file header.

Prototype `int SVC_CHECKFILE (const char *filename);`

Return Values

Success:	0 the values match, with the caller's buffer containing the requested timestamp.
Failure:	-1 and <code>errno</code> set to <code>EPERM</code> : Checksums do not match.
	-1 and <code>errno</code> set to <code>EBADF</code> : Invalid handle or file currently locked by another user.
	-1 and <code>errno</code> set to <code>ENOENT</code> : File does not exist.
	-1 and <code>errno</code> set to <code>EACCES</code> : Caller's buffer is not writable (for example, bad buffer address).

The use of `EPERM` (`errno` = 1) to indicate a bad checksum is for historical reasons. `<svc.h>` defines some alternate symbolic constants for use with this function:

```
FILE_OK = 0
FILE_BAD = 1
FILE_NOT_FOUND = 2
```

_remove()

The name of the file to delete is placed in the buffer as a NULL-terminated string.

NOTE



Prior to calling this function, all open handles for the target file *must* be closed. This includes any multiple opens of the target file.

Prototype

```
int _remove(const char *filename);
```

Parameter

`filename` Name of file to delete.

Return Values

Success: 0 = the specified file is found, the file is deleted and `result` is set to zero.

Failure: `result` set to -1 and `errno` is set to `ENOENT`: File not found.

lock()

Locks the open file associated with `handle`, preventing it from being accessed through any other handle. A locked file cannot be opened, and if it is already open under another handle it cannot be read or written through that handle.

Prototype `int lock (int hdl, long reserved1, long reserved2);`

Parameters

`reservedn` Reserved for future use.

Return Values

 Success: 0

 Failure: -1 and `errno` set to `EBADF` if the file is not writable.

unlock()

Removes a lock set by [lock\(\)](#) from the open file associated with `handle`.

Prototype `int unlock (int hdl, long reserved1, long reserved2);`

Return Values

 Success: 0

 Failure: -1 and `errno` set to `EBADF`: Invalid `handle`.

 -1 and `errno` set to `EACCESS`: The file is not locked through this handle.

_rename()

Points the caller's pointer to a pair of pointers to ASCII pathnames. If the first pathname exists in the directory and if the second pathname does *not*, the name of the file in question is renamed.

NOTE

In OS that support sub-directories, the `_rename()` function also renames internal signature files (extension .S1G) created when a file is accompanied by a P7S file authentication file during download.

This is important as when an authenticated file is renamed and the S1G file containing the HASH of the file is not renamed, the system integrity checks will cause a CALL CUSTOMER SERVICE error.

When a downloaded file is authenticated, the system created a filename.S1G file containing the HASH of the file. System integrity checks run each 24 hours to validate that authenticated files have not been changed. If the S1G file is not renamed the integrity check fails.

Prototype

```
int _rename(const char *oldname, const char *newname);
```

Parameters

`oldname` Name of the file to rename.
`newname` New name for the file.

Return Values

Success: 0 file renamed successfully.

Failure: -1 with `errno` set to `EACCES`: Either filename is not readable (for example, bad buffer address).

 -1 with `errno` set to `ENOENT`: file specified in `oldname` does not exist.

 -1 with `errno` set to `EEXIST`: file specified in `newname` already exists.

 -1 with `errno` set to `EINVAL`: Attempt to change group (for example, `rename("foo", "/goo");`).

get_file_attributes()

Returns the attribute byte for an open file. It is similar to [dir_get_all_attributes\(\)](#), except that the file is specified by `handle` rather than a name. See [Table 12](#) for a list of attribute bits.

See also, [dir_get_all_attributes\(\)](#), [set_file_attributes\(\)](#), [reset_file_attributes\(\)](#).

Prototype `int get_file_attributes (int hdl);`

Return Values

Failure: -1 and `errno` set to `EBADF`: Invalid file handle.

NOTE



If a flash file is open for writing, its attributes are not actually set until the file is closed. In this case, `get_file_attributes()` returns a meaningless value.

get_file_max()

Returns the maximum file data size set by `set_file_max()`. If a maximum size has not been set, `get_file_max()` returns the current data length.

NOTE



`set_file_max`, which sets the maximum data size for a file is supported only for backward compatibility.

Prototype `long get_file_max (int hdl);`

Return Values

Failure: -1 and `errno` set to `EBADF`: Invalid handle.

-1 and `errno` set to `EINVAL`: handle is not a file.

Example See [set_file_max\(\)](#) for linked code example.

reset_file_attributes()

Clears attribute flags for an opened file. It is similar to [dir_reset_attributes\(\)](#), except that the file is specified by a handle rather than a name.

NOTE



The file must be open with write access.

See also, [dir_reset_attributes\(\)](#), [get_file_attributes\(\)](#), [set_file_attributes\(\)](#).

Prototype

```
int reset_file_attributes (int hdl, int attributes);
```

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Invalid file handle or no write access.

-1 and errno set to EINVAL: Invalid flag (attempt to set ATTR_NOT_AUTH).

set_file_attributes()

Sets selected attribute flags for an open file. It is similar to [dir_set_attributes\(\)](#), except that the file is specified by a handle instead of a name. See also, [dir_set_attributes\(\)](#), [get_file_attributes\(\)](#), [reset_file_attributes\(\)](#).

NOTE

The file must be open with write access.

Setting ATTR_NO_GROW is not permitted on flash files.

Prototype

```
int set_file_attributes (int hdl, int attributes);
```

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Invalid file handle or no write access.

-1 and errno set to EINVAL: Invalid flag (attempt to set ATTR_NOT_AUTH or ATTR_NO_GROW on flash file).

set_file_max()

Sets the maximum data size for a file. The file must be in the memory and opened with write access.

NOTE

This function is a Predator function and has no affect in Trident. Trident OS always return 0.

close()

This function call closes the file specified in the `hdl` parameter. Each file opened by an application must also be closed when access to the file is no longer needed.

This function closes open pipe handle. If `hdl` is not found in the list of open pipes, only return 0; if found, marks the pipe as closed.

`close()` also releases ownership of the console device. Failure returns -1 with `errno` set to `EBADF`: Caller does not own console device.

On MCR, this function disables the card reader input, preventing the terminal from recognizing card reads, any card data queued on device are discarded. Any card swipes that occur before device `open()` or after device `close()` are discarded.

`close()` also releases the resources associated with the clock or beeper handle. It also releases printer or IPP ownership. Anything that is queued for printing or all unread data will be lost.

This function closes the USB device. Upon `close()`, the OS checks if all external USB are closed. If so, powers on all external USB ports.

`close()` disables the device; the application is responsible for ensuring that all data written to the port was transmitted before issuing `close()`. Issuing `close()` without checking [Prototype](#) or calling `SVC_WAIT()` may cause loss of transmitted characters.

This function also releases the USB Ethernet or USB WiFi device. -1 returns error set to `EBADF`. Failure only occurs if the caller is not the current owner.

On VX 680 3G, the PCM driver has a standard Verix `close()` API.

On BT, this function disconnects the `DEV_BLUETOOTH`, `DEV_DUN_BT` device from the SPP port. This function also disconnects the `DEV_PAN_BT` device from the PAN network.

Prototype `int close(int hdl);`

Parameters

<code>hdl</code>	Once the file is closed, the <code>handle</code> is no longer valid and all internal resources used by the handle are released.
------------------	---

Example

Click on the linked C code file for the beeper example.

Return Values

Success: 0

Failure: -1 with `errno` set to `EBADF`: File not open.

Keyed File Reads and Writes

Keyed files allow records to be accessed by unique character-based strings. In a keyed file, each record consists of two elements: a key value and its associated data.

The same rules for CVLRs apply for keyed files: Both key and data values *must* be text based; Both elements are compressed when stored; lowercase characters are converted to their uppercase equivalent.

In effect, the mechanism for the keyed file functions (`getkey()` and `putkey()`) are paired CVLR functions (for key and data).

NOTE

Use the `getkey()` and `putkey()` functions instead of the CVLR functions. These functions are more efficient and avoid file corruption.

The maximum length of a key is 32 bytes; data can be up to 128 bytes.

NOTE

The 32- and 128-byte maximums will always work, but it is possible to create and use files containing longer records with careful, well-planned write and access structure. While Verix eVo file system does not reject technically over-limit records, it also does not prohibit them.

Keyed files *must* be created by the system or the application prior to access with `putkey()` and `getkey()`. If the application creates a file to use as a keyed file with `open()`, it should immediately `close()` the file to terminate the file handle.

NOTE

When VeriCentre is used to download a keyed file, keys *must* be 7 bytes or less. VeriCentre does *not* restrict data size.

Unlike other file access methods, keyed files do not need to be opened and closed prior to each read or write. The `getkey()` and `putkey()` functions perform these operations internally.

getkey()

Retrieves data associated with a given key value.

Prototype `int getkey(const char *key, char *buf, int size, const char *filename);`

Parameters

key	A NULL-terminated string up to 32 bytes in length (7 bytes if used with VeriCentre downloads).
buf	Pointer to an array where the data associated with key is stored.
size	Specifies the size of buf.
filename	Pointer to a NULL-terminated string; up to 32 characters long. If filename = 0, the CONFIG.SYS file is searched for the specified key.

The number of bytes read does not necessarily have to match the actual record size. For example, you can read only the first 32 bytes of each record, even though the record is 120 bytes long. If the entire record needs to be read, pass the maximum value of 128 in the `max_bytes` parameter.

NOTE



The 32- and 128-byte maximums will always work, but it is possible to create and use files containing longer records with careful, well-planned write and access structure. While Verix eVo file system does not reject technically over-limit records, it also does not prohibit them.

Return Values

If the file does not contain a record-matching key, `bytes_read` is returned with a value of zero.

Success: 0

Failure: -1 with `errno` set to `EACCES`: Either `key` is not readable, `buf` is not writable, or `filename` is not readable. Typically, this means that the pointer is invalid.
-1 with `errno` set to `EBADF`: The file specified in `filename` does not exist.
-1 with `errno` set to `ENOENT`: The pair identified by `key` does not exist within the file.

putkey()

Stores data for a given key. Can also delete a key/record pair by setting the `count` parameter to zero.

Prototype

```
int putkey(const char *key, const char *buf, int const size,
char const *filename);
```

Parameters

<code>key</code>	A NULL-terminated string up to 32 bytes in length (7 bytes if used with VeriCentre downloads).
<code>buf</code>	Pointer to a character array.
<code>size</code>	Specifies the number of bytes to write.
<code>filename</code>	Pointer to a NULL-terminated string that is up to 32 characters long.

Return Values

NOTE



The file being written to *must* exist; this can be accomplished using `open()` with the `O_CREAT` attribute bit set.

Failure: -1 with `errno` set to `EBADF`: Non-existent file specified.

-1 with `errno` set to `EACCES`: Either `key` is not readable, `buf` is not writable, or `filename` is not readable. Typically, this means that the pointer is invalid.

-1 with `errno` set to `EINVAL`: `size = 0` for non-existent key.

dir_get_all_attributes()

Retrieves the file attributes of the requested file.

Prototype `int dir_get_all_attributes(const char *file);`

Parameters

<code>file</code>	Name of the file to retrieve attributes.
-------------------	--

Return Values

Success:	Bit mask of file attributes as defined in <code>SVC.H</code> .
Failure:	-1 with <code>errno</code> set to <code>ENOENT</code> : filename not found.

dir_get_attributes()

Provides access to the file attribute bits that the file system maintains. See [Table 12](#) for a list of attribute bits.

Prototype `int dir_get_attributes (const char *filename);`

Return Values One of the values from the list of attribute bits in [Table 12](#) is returned.

Failure:	-1 with <code>errno</code> set to <code>EACCES</code> : The file specified in <code>filename</code> is not readable.
	-1 with <code>errno</code> set to <code>ENOENT</code> : The file specified in <code>filename</code> file does not exist.

dir_get_file_date()

Retrieves the file date.

Prototype `int dir_get_file_date(const char *filename, char *yyyymmddhhmmss);`

Parameters

<code>filename</code>	Pointer to a valid name of file in the file group.
<code>yyyymmddhhmmss</code>	Pointer to a 14-character buffer that contains the date of either the file creation date or last modified date of the file after the function call.

Return Values

Success: 0

Failure: -1 with `errno` set to `EACCES`: The file specified in `filename` is not readable.
-1 with `errno` set to `ENOENT`: The file specified in `filename` file does not exist.

dir_get_file_size()

Returns the size of the file.

Prototype `long dir_get_file_size(const char *filename);`

Parameters

<code>filename</code>	Pointer to a valid name of a file in the file group.
-----------------------	--

Return Values

Success: > 0

Failure: -1 with `errno` set to `EBADF`: Invalid file handle.
-1 with `errno` set to `EACCES`: The file specified in `filename` is not readable.
-1 with `errno` set to `ENOENT`: The file specified in `filename` file does not exist.

dir_get_file_sz()

Returns the number of data bytes in the named file.

NOTE



There is no handle-based version of this function. The data size of an open file can be determined by using `lseek()`, `seek_vlr()`, and `seek_cvlr()` to seek to its end.

Prototype

```
long dir_get_file_sz (const char *filename);
```

Return Values

Failure: -1 with `errno` set to `EACCES`: The file specified in `filename` is not readable.
-1 with `errno` set to `ENOENT`: The file specified in `filename` file does not exist.

dir_get_first()

Returns a NULL-terminated string containing the name of the first file in the directory (usually `CONFIG.SYS`).

Prototype

```
int dir_get_first (char *drive);
```

Parameters

`drive` NULL-terminated name of the first file found in the directory.

NOTE



A file system identifier is passed in the `drive` parameter, such as `I:` for RAM or `F:` for flash memory.

Return Values

Success: 0
Failure: -1 with `errno` set to `EACCES`: Caller's buffer is not writable.
-1 with `errno` set to `ENOENT`: No files exist on this drive (for current group)

dir_get_next()

Retrieves the filename that follows the current file in the buffer.

Prototype `int dir_get_next (char *buf);`

Parameters

<code>buf</code>	Contains the name of the file returned from a prior call to <code>dir_get_first()</code> .
------------------	--

Return Values The directory is searched for the specified filename, and the name of the file in the following entry is returned.

Failure: -1 with `errno` set to `EACCES`: Caller's buffer is not writable

 -1 with `errno` set to `ENOENT`: Filename passed in buffer is not found or is the last entry in the directory.

NOTE



A file system identifier can be passed in the `buf` parameter, such as `I:` or `F:`

dir_get_sizes()

Returns general information about the directory—the number of files in the directory, the amount of memory used by the file system, and the amount of free space remaining. The caller must provide a buffer to hold the structure `fs_size`. The structure `fs_size` is defined in the header file `svc.h`.

NOTE

Trident drives I: and F: are both on the Nand flash, the “Avail” member of the resulting struct `fs_sizes` tells the total (I: + F:) available NAND flash memory at the time the function is called.

In Predator, the integer value returned is the available memory at the time of calling this function. This value is useful to applications in allocating / maintaining stack and heap. The application executable is moved into the memory prior to execution. The OS will use a portion of the memory for its own house keeping. The physical ceiling of all application executables (including the application's total heap and stack as set by `Vrxhdr`) and the Operating System's own housekeeping needs reflects the total Physical memory (some platforms have 32 MB, others have 64 MB).

Prototype

```
int dir_get_sizes (const char *drive, struct fs_size *fs);
```

Parameters

`drive` File system location.
`fs_size` Pointer to structure `fs_size`.

Return Values

The available memory in bytes.

Example

```
struct fs_size sizes;  
dir_get_sizes("F:", &sizes);  
printf("%d files in F:\n", sizes.Count);  
printf("%ld bytes used in F:\n", sizes.InUse);  
printf("%ld bytes NAND flash memory available\n", sizes.Avail);
```

dir_put_file_date()

Attaches a date to the file. The date of a file is normally updated to the current system time whenever a file is changed. One exception is files downloaded using [SVC_ZONTALK\(\)](#). The date for these files corresponds to the timestamp of the file downloaded from the host machine.

Prototype `int dir_put_file_date(const char *filename, const char *yyyymmddhhss);`

Parameters

<code>filename</code>	Pointer to a valid name of file in the file group.
<code>yyyymmddhhmmss</code>	Pointer to a 14-character buffer that contains the date to assign the file.

Return Values

Success: 0, no problems found.

Failure: -1 and `errno` set to `EBADF`: Invalid file handle.

 -1 and `errno` set to `ENOENT`: File does not exist.

 -1 and `errno` set to `EINVAL`: Specified date/time invalid.

 -1 and `errno` set to `EACCES`: One or both of the caller's parameters is not readable (bad pointer).

dir_reset_attributes()

Clears attribute flags for a file. The flags indicated by “1” bits in attributes are set to “0”. To set flags, call [dir_set_attributes\(\)](#). ATTR_NOT_AUTH cannot be changed, nor can the attributes of files in flash.

NOTE



Use [reset_file_attributes\(\)](#) to change flash file attributes.

See [Table 12](#) descriptions of the attribute bits, and [dir_set_attributes\(\)](#) for an example.

See also, [reset_file_attributes\(\)](#), [dir_get_all_attributes\(\)](#), [dir_set_attributes\(\)](#).

Prototype

```
int dir_reset_attributes (const char *filename, int attributes);
```

Return Values

Success: 0

Failure: -1 and errno set to ENOENT: File does not exist.
-1 and errno set to EINVAL: Invalid flag (attempt to clear ATTR_NOT_AUTH).
-1 and errno set to EBADF: Attempt to change flash file attributes.
-1 and errno set to EACCES: File specified in filename is not readable.

dir_set_attributes()

Turns on one or more of the attribute bits for the specified file. The attribute argument is ORed with the file attributes. See [Table 12](#) for a list of attribute bits.

Prototype `int dir_set_attributes (const char *filename, int attributes);`

Table 12 Attribute Bits and Function

Bit	Name	Function
0	ATTR_READONLY	Intended to mark files that cannot be written. Note: This function is not currently supported.
1	ATTR_NO_CKSUM	Verix eVo OS does not validate the checksum for a file with this bit set. By default, the bit is not set, and the OS attempts to validate the checksum of the file. If the checksum is erroneous, a system notification is generated. By setting this attribute bit, the application can validate the checksum and take appropriate action to handle errors.
2	ATTR_NO_GROW	This bit indicates the use of file extents. By default, the bit is <i>not</i> set. The bit is normally set/reset with the <code>set_file_max()</code> call.
3	ATTR_NOT_AUTH	This bit is set by default to indicate that the file is not authenticated. The bit is reset when Verix eVo OS authenticates the file. An application cannot reset this bit. Once the bit is reset, the file cannot be opened for update. Note that secure terminals <i>require</i> that application code files are authenticated to execute.

Return Values

Success: 0

Failure: -1 is returned with `errno` set to `ENOENT`: File does not exist.

-1 is returned with `errno` set to `EACCES`: File specified in `filename` is not readable.

-1 is returned with `errno` set to `EINVAL`: Invalid attribute setting.

Example

In the linked code example a task responsible for updating a critical file must minimize the chances of another task writing to it. It sets the read-only attribute bit, clearing it only long enough to do its own updates. (Note that another task could also reset the attribute bit, so protection is limited.) See also [lock\(\)](#) and [unlock\(\)](#) for another approach to this problem.

file_copy()

Copies the file named by source to target. `file_copy()` fails if the target file already exists or the source file is already open.

Prototype

```
int file_copy (const char *source, const char *target);
```

Return Values

Success: 0

Failure: -1 and `errno` set to `EEXIST`: Target file already exists.

-1 and `errno` set to `EACCES`: Either source or target filenames could not be read.

-1 and `errno` set to `ENOENT`: File specified in source does not exist.

-1 and `errno` set to any other value: An `open()`, `read()`, or `write()` call failed during copying; `errno` remains as set by the failed function.

Example

The linked code example copies a file to Group 15, where it is available to tasks running in any other group.

put_file_date()

Writes the date format `yyymmddhhmmss` to the file defined by the file handle.

Prototype `int put_file_date(int hdl, const char *buf);`

Parameters

<code>hdl</code>	File handle returned by open call.
<code>buf</code>	ASCII string formatted as <code>yyymmddhhmmss</code> .

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EINVAL</code> - Date is not correct.

SVC_RAM_SIZE()

Returns the amount of memory, in kilobytes, installed in the terminal. See also, [dir_get_sizes\(\)](#) and [SVC_FLASH_SIZE\(\)](#).

Prototype `int SVC_RAM_SIZE (void);`

Example Click the linked example to view the sample code.

unzip()

Decompresses a standard zip format file. Decompression runs as a separate task, and `unzip()` returns immediately after starting it. The caller can continue to execute while `unzip()` runs in the background. When the unzip task completes, an `EVT_SYSTEM` event is posted to the invoking task.

The `CONFIG.SYS` variable `UNZIP` (not to be confused with `*UNZIP`) is set to 0 when the unzip operation starts, and to 1 on successful completion. See [UNZIP—Determine Decompress Results](#).

No new tasks can start while the unzip task is running. Only one instance of `unzip()` can run at any time. Calls to `unzip()` or `run()` made while the unzip task is active, fail with an `ENOMEM` error.

NOTE

The `unzip()` function does only minimal validation. In particular, it does not verify the existence, accessibility, or format of the specified file. Any errors result in the failure of the unzip task, but are not directly visible to the caller.

Prototype

```
int unzip (const char *zipfile);
```

Return Values

Success: 0: The unzip task started successfully

Failure: -1 and `errno` set to `ENOMEM`: Unzip task already running or unzip utility not found.

Example

The linked example code invokes `unzip()` and waits for it to complete before proceeding.

Flash File System - Special Case

For compatibility with previous products, Verix eVo supports several different file "drives"—the default "I:" or "RAM" file system and the "F:" or "FLASH" file system. Both file systems behave and are managed similarly in nearly all respects. There is one exception, reflecting rather odd behavior that is necessary in order to be compatible with the previous products.

If you wish to copy a file from "FOO" to "GOO" possibly making changes along the way, you would typically open "FOO" as read-only, and you would open "GOO" as read-write or write-only, then proceed to read "FOO" records, modify them, and write "GOO" records (consider, for example, that "GOO" might be the same as "FOO" except every "carriage return" in "FOO" is to be replaced by "carriage return, line feed" in "GOO").

In Predator and Trident, a variation of the above could also be handled—your output file could sometimes have the same name as the input file. In particular, you could perform the following:

```
hdl1 = open ("F:FOO", O_RDONLY);  
hdl2 = open ("F:FOO", O_CREAT+O_WRONLY+O_TRUNC+O_APPEND);  
<read(hdl1, ), write(hdl2, ) etc >
```

You would expect that when hdl2 is delivered, the file "F:FOO" would be truncated and have length 0, so that nothing could be read using hdl1. In previous products, however, the data for "F:FOO" was stored in NOR FLASH, and it remained available until the next `dir_coalesce` call. For this reason, the above example worked the same as the initial example in which the output file was named "GOO".

For compatibility purposes, we support this behavior in Verix eVo. Specifically, if a user opens an output file (either `O_WRONLY` or `O_RDWR` or `O_CREAT`) in drive "F:" with attribute `O_TRUNC`, and if there are other users of the file, then the other users retain access to the original file as long as it is open. The original file, however, is now marked temporary and will disappear as soon as all users close it.

NOTE

Use of this feature is not recommended, and support will likely be withdrawn in a future release.

dir_flash_coalesce()

Erases all files tagged for deletion in the flash file system, and pushes the current files towards the beginning of the flash file system.

NOTE

Verix eVo automatically manages the retrieval of FLASH pages that are made available by writing or deleting files. No application intervention is normally necessary.

For effective use of FLASH memory, maintain a substantial number of free pages. As the number of free pages declines, the file manager becomes inefficient and begins to spend a significant amount of time retrieving available space. Efficiency declines as the amount of free space approaches 0.

In order to help ensure that free blocks are available when needed, you may choose to invoke the recovery process explicitly during "off hours" by calling `dir_flash_coalesce`, partially written blocks will be consolidated.

WARNING

In general, use of `dir_flash_coalesce` is **not recommended**, since it adds wear with little benefit. It is far better to reduce the size of files and to reduce the number of file writes.

This function call returns `-EINVAL` for Trident terminals. Trident has an automatic wear leveling system.

dir_flash_coalesce_size()

Returns the number of bytes to reclaim with a coalesce. Returns `EBUSY` if there is a file with write access not yet closed. This function should be called prior to the coalesce to determine if a coalesce is necessary and safe to perform.

`dir_flash_coalesce_size()` also checks for flash files open for writing.

These files are deleted if [dir_flash_coalesce\(\)](#) is called before they are closed.

See also [dir_flash_coalesce\(\)](#).

Prototype `int dir_flash_coalesce_size (long *size);`

Parameters

<code>size</code>	Set to the number of bytes of flash memory used by deleted files. This space can be recovered for use by new files through dir_flash_coalesce() .
-------------------	---

Return Values Recoverable size obtained from this call is always 0. Returns 0 for Trident.

Example See the linked code example in [dir_flash_coalesce\(\)](#).

SVC_FLASH_SIZE()

Returns the amount of flash memory installed in the device in kilobytes. See [SVC_RAM_SIZE\(\)](#) for an example. See also [dir_get_sizes\(\)](#), and [dir_flash_coalesce_size\(\)](#)

Prototype `int SVC_FLASH_SIZE (void);`

Supporting Subdirectory

Sub-directories are only supported on Trident terminals with OS version QTt01nn or higher. The original file support is continued with some modifications to the file naming conventions. Applications using the new OS versions must be evaluated for compliance to the new naming conventions. Note that subdirectory support does not change the GID restrictions. The GID is the first level directory for an application, thus, an application in GID 1 will see "1" as the root directory.

Naming Restrictions

Rules for file names are enforced to support subdirectories—"bizarre" names that conflict with common computer usage conventions are discouraged.

- Special characters are prohibited. Names for files and directories cannot contain certain special characters. Users will not be able to create a file/directory if it includes forward and reverse slash characters, colons, question marks, and asterisks. If an existing file contains such, the file will be renamed with the offending characters removed.
- The special directory name ".." consisting of two periods means "higher level directory" as it does in other well-known file systems, so it cannot be used for the name of either a file or a directory.
- Use of other special characters should be considered carefully in considering portability. The use of space and other "white space" characters, as well as the equal sign, plus sign, greater-than sign, less-than sign, percent sign, and other special characters that conflict with common programming usage on other operating systems such as MS-DOS, is strongly discouraged (although currently not prohibited).
- Lower-case letters are permitted, but they will be converted to upper case. The names "FOO" and "foo" continue to be equivalent, as they have been in the past.
- A name entirely composed of ASCII digits 0 through 9 must be at least three characters long. This is to identify a file's group as part of its name. This explicitly reserves all 110 such forms: 0-9 and 00-99.

WARNING



Mandatory conversion is implemented. This means that when the new OS supporting subdirectories is installed in a terminal, it will scan the existing file directories to make sure that existing names satisfy the restrictions noted above. Any conflicting file names will be modified to conform.

It is recommended that any applications that use non-conforming file names should be modified before installing the OS.

NOTE



File operations (such as open) that use the filename will accept the full pathname including the subdirectory.

*GO can start applications residing in a subdirectory. However, chidir() is not called by VTM, thus, the root for the application started is not the subdirectory but the GID.

- Subdirectory names must be no more than 32 characters long. The maximum path length, including delimiters for nested subdirectories, is 250 bytes. This absolute limit applies to the file name as viewed by the system; since the apparent root directory for a task may suppress part of this name, the limit may appear to be smaller.

Tokenizing Rules and Options

Following the usual rules for specifying file names in a hierarchical directory, a file's "path name" can include its drive name (followed by colon, 0x3A); its group assignment (one or two decimal digits, followed by forward slash); and subdirectory names separated by a special "subdirectory token" character. Two of such tokens are supported.

Consistent with MS-DOS usage, the reverse slash character ('\ or 0x5C) can always be used to delimit subdirectory names. If you are attempting to port an application from MS-DOS, its use of subdirectories should pose no problems. If a path name begins with the reverse slash character, the path name is considered absolute, beginning with the caller's root directory.

A forward slash (/ or 0x2F) can also be used but several restrictions apply. Names for files or directories cannot be a pure decimal number, since a decimal number is always interpreted as a group number. However, a name consisting of at least three characters in length is allowed although not entirely recommended.

Any file or directory name that begins with a decimal number followed by a slash always indicates a group number. The name is always considered to represent an absolute path—it is relative to the root directory, rather than to the current working directory. For example, "F:15/COMMON.LIB" always means the file "COMMON.LIB" in the "root" of the "F:" directory.

A file or directory name beginning with forward slash (without preceding numeric digits), means an absolute path for group 15. For example, "/FOO" means the group 15 file "FOO" in the root directory.

The application calls enumerated below are supported:

- `mkdir()`
- `chdir()`
- `rmdir()`
- `getcwd()`

WARNING



DO NOT attempt to use `mkdir()` and `rmdir()` on non-subdirectories OS. You will not receive any error messages, but this will cause the terminal to fail to reboot and may not be recoverable.

Example Below is an example of the services provided with these calls:

<code>mkdir ("S1");</code>	The name of the new subdirectory is S1. It will be created in the current directory.
<code>rmdir ("S1");</code>	Remove the subdirectory just created.
<code>chdir ("\\");</code>	Restore my current directory to the original "home" directory for this drive. Note that the C pre-processor requires that we use two slashes here.
<code>mkdir ("F:2/xyz\\abc");</code>	Make a new subdirectory below the existing subdirectory "xyz" which is an entry in the "top level" directory for group 2. It is not necessary to BE in group 2 to make this call, though you must have access to group 2 (in other words, this works for applications running in group 1 or 2, but not in groups 3, 4, etc). If "xyz" doesn't exist already, this will fail.
<code>chdir ("S1\\T2/U333");</code>	Beginning in the current directory, traverse subdirectories S1, then T2, finally ending in U333. This will become my "current directory" until another "chdir" is performed.
<code>chdir ("..");</code>	Go up one subdirectory level. If this command follows that in the previous example, our current working directory will be "S1/T2"
<code>mkdir ("/S1");</code>	Beginning in the absolute directory for group 15, create a new subdirectory called S1. Since group 15 is common to all applications, any application can make this call.
<code>chdir ("/S1");</code>	Beginning in the absolute directory for group 15, navigate to the subdirectory called S1 and make that my current directory. This call usually fails. It can only succeed if the caller is already in group 15. You cannot use chdir to change your group setting.
<code>chdir ("\\S1");</code>	Beginning in MY absolute directory, find the entry for subdirectory S1 and make that my current directory.
<code>chdir ("F:");</code>	Make "F:" the default drive. This does not change your current working directory for either "I:" or "F:" and it offers little benefit to offset its huge potential for confusion.

Inheritance Rules and Options

As an application changes its directory and drive settings, its view of the file system shift. This altered view is inherited by any program or thread that the application invokes.

If an application calls `setdrive('F')` and then `chdir("XYZ")`, and then invokes `unzip("ARCHIVE.ZIP")`, the unzip program will behave as if it, too, had first called `setdrive('F')` and `chdir("XYZ")`, and when it opens "ARCHIVE.ZIP", it expects the file to be found on the "F:" drive in the "XYZ" subdirectory. Furthermore, this initial setting of the current directory becomes the permanent root directory for the new task—it cannot see any files "above" its root directory. This mechanism effectively restricts the namespace for the new task.

WARNING



Historically, Verix applications assume that "FOO" was equivalent to "I:FOO" in all cases. Now that users can select subdirectories and a new default drive, this is no longer guaranteed.

NOTE



The same inheritance rules apply to both Threads and Tasks. The thread's permanent root directory for file access is determined based on the parent task's current working directory when the thread is started.

Library Names

The program header contains the names of libraries to be linked with the program code during startup. These names are always absolute. For example, when a program "F:1/BIN/PROG.OUT" is to be run and that its header indicates that library "X.LIB" is needed; the OS will look for the library file "I:1/X.LIB" in this case.

Environment Variables

For each group, there is only one `CONFIG.SYS` file, located in the top-level subdirectory for that group. Every user in that group has access to this file using the familiar `get_env` and `put_env` calls, even if the user is otherwise restricted from accessing this top-level directory.

Effect on Current Working Directory

If an application is invoked with a restricted view of the drive's directory structure, the result for a call to `getcwd` will reflect the restriction that it will not see the higher level directories. For example, if application "foo" is invoked and calls `getcwd`, its buffer will contain "I:". Let it navigate to subdirectory "BAR" and call `getcwd`. This time, its buffer will contain "I:\BAR". If it now invokes the application "footoo" and if "footoo" calls `getcwd`, then footoo will see only "I:" in its buffer.

Subdirectories and Groups

Each drive in the file system has its own top-level directory, generally considered the "root" directory. Within the root directory is a subdirectory named "1" created by the OpSys to hold all files in Group 1. Similar subdirectories will be created to hold files for other groups.

Impact of `set_group()`

The `set_group()` function has been modified internally to call `chdir()` and `chdir()` on behalf of the application. Thus, the current working directory will be lost each time the application calls `set_group()`. If the user calls `set_group()` to restore his own "permanent" group, he will always be placed in his "home" directory, which is not necessarily the top-level "root" directory for that group; this mechanism cannot be used to bypass the inheritance restrictions. A working directory cannot be set to another directory belonging to a different group. `set_group` must be called first to reset the working directories for both "I:" and "F:" drives.

WARNING



DO NOT DOWNGRADE A SUBDIRECTORY OS to a non-subdirectory OS. It is highly probable the file system will be corrupted and the terminal will not be recoverable outside of a service center.

You can use DDL to download to subdirectories, using the '@' feature to specify a different destination name. For example, "ddl foo@subdir/foo" will download "foo" from the PC into "subdir\foo" in the terminal. Note that when downloading, the terminal creates necessary subdirectory structure to hold the file. This "implicit mkdir" support is NOT part of the open logic. There are no current plans to upgrade VeriCentre with equivalent functionality. The use of ZIP archives is the preferred approach.

If you create a ZIP archive containing subdirectories and download that archive to a terminal, the subdirectory structure is recreated in the terminal. You must be careful where you place the archive, however, since unzip is subject to the same working-directory inheritance rules as other programs. For example, you download A.ZIP to the top level directory for group 1. If you unzip it there, the same subdirectory structure will be created at this point. If you were to navigate to a subdirectory below this, however, you would not be successful if you call unzip("./A.ZIP"), since the unzip program cannot see the input file.

The unit-to-unit "send" feature copies the subdirectory structure. Note, however, that if the receiving terminal is running an older OS that doesn't support subdirectories, the results will be unknown and probably result in a corrupted file system.

Below are important information on files/subdirectories that you should know:

- GID restrictions apply. Even if you are in a subdirectory, your access to files in other groups is limited by the same rules that have always applied.
- To get a list of sub-directories under a given directory, iterate with `dir_get_first` and `dir_get_next`, check their attributes in order to detect the subdirectories.

Example: If you call `dir_get_attributes`, test for `ATTR_SUBDIR`. If you call `dir_get_all_attributes`, check `ATTR__SUBDIR`. Whenever you are iterating through subdirectories, you inevitably need to "go back" to the next higher directory; this is where you will appreciate how handy it is to call `chdir("../")`.

- You will not have the equivalent of `dir_get_file_sz` to size a directory, thus, you iterate through the files.
- `Open()` accepts the full path name to open a file.

Example: If you are in group 1, in the `1/DATA/CONST` directory, you could open a file in the `1/DATA` subdirectory by calling `open("1/DATA/TEST")` or `open("../TEST")` or `open("../DATA/TEST")`.

- You can set `*GO` variable to a program in a subdirectory. Note that the Verix Terminal Manager program (or system mode) is the program which invokes the `*GO` application, and VTM will not call `chdir` prior to running this application. This means that the `*GO` application will not be restricted from seeing the higher level directories.

- If you upgraded a terminal to an OS that supports subdirectories, you can not downgrade to an OS that does not support subdirectories.
- The OS authenticates *.P7S files in subdirectories as long as the P7S file is in the “I:” file system. The corresponding target file must be in the same subdirectory.

Example: If you want to place the target file in the “F:” file system, you must create the same subdirectory structure in both “I:” and “F:” in order to authenticate it; this is because the file signing tool has no way to specify path names. Be sure to sign with the “-L” option in this case.

- The absolute path name is limited to approximately 250 bytes (the approximation comes into play because no application can see the entire path). If you use single-character subdirectory names specified by “slash” delimiters, you can create approximately 125 levels of subdirectories.
- As soon as you change groups, your working directory is reset to the appropriate top-level directory for that group. The OS does not remember the current working directory.
- Unzip operation supports subdirectories. If you create a ZIP archive containing subdirectories and download that archive to a terminal, the subdirectory structure will be recreated in the terminal.

NOTE



Exercise care in where you place the archive; unzip is subject to the same working-directory inheritance rules as other programs.

Example: When you download A.ZIP to the top level directory for group 1 and unzip it there, the same subdirectory structure will be created at this point. If you navigate to a subdirectory below this, however, you would not be successful if you call `unzip("../A.ZIP")`, since the unzip program cannot see the input file.

- You can use DDL to download to subdirectories. The ‘@’ feature can be used to specify a different destination name.

Example: `ddl foo@subdir/foo` will download “foo” from the PC into “subdir/foo” in the terminal. When downloading, the terminal will create any necessary subdirectory structure to hold the file. This “implicit mkdir” support is NOT part of the open logic.

There are no current plans to upgrade VeriCentre with equivalent functionality. The use of ZIP archives is the preferred approach.

- The unit-to-unit “send” feature copies the subdirectory structure. However, if the receiving terminal is running an older OS that does not support subdirectories, the results will be unsatisfactory.

mkdir()

Creates a new subdirectory.

WARNING

DO NOT attempt to use `mkdir()` and `rmdir()` on non-subdirectories OS. You will not receive any error messages, but this will cause the terminal to fail to reboot and may not be recoverable.

Prototype

```
int mkdir(const char* name);
```

Parameters

<code>char* name</code>	Name of the subdirectory affected.
-------------------------	------------------------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: ENOENT no such subdirectory. EACCES the user's buffer is not accessible. EEXIST the subdirectory specified for <code>mkdir</code> already exists. EPERM the caller does not have access to this drive. ENFILE no file handles available at this time. ENOSPC out of memory.

chdir()

Restores the current directory to the original “home” directory. Note that the C pre-processor requires the use of two slashes (“\”).

Prototype `int chdir(const char* name);`

Parameters

<code>char* name</code>	Name of the subdirectory affected.
-------------------------	------------------------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: ENOENT no such subdirectory. EACCES the user’s buffer is not accessible. EEXIST the subdirectory specified for <code>mkdir</code> already exists. EPERM the caller does not have access to this drive. ENFILE no file handles available at this time. ENOSPC out of memory.

rmdir()

Removes the newly-created subdirectory.

WARNING

DO NOT attempt to use `mkdir()` and `rmdir()` on non-subdirectories OS. You will not receive any error messages, but this will cause the terminal to fail to reboot and may not be recoverable.

Prototype

```
int rmdir(const char* name);
```

Parameters

<code>char* name</code>	Name of the subdirectory affected.
-------------------------	------------------------------------

Return Values

Success:	0
----------	---

Failure:	-1 with <code>errno</code> set to:
----------	------------------------------------

`ENOENT` no such subdirectory.

`EACCES` the user's buffer is not accessible.

`EEXIST` the subdirectory specified for `mkdir()` already exists.

`EPERM` the caller does not have access to this drive.

`ENFILE` no file handles available at this time.

`ENOSPC` out of memory.

`EINVAL` the directory specified for `rmdir()` is not empty, so it cannot be removed.

getcwd()

Used to obtain the full “path” for the current working directory. This includes the drive specifier (“I:\”) followed by the names of the relevant subdirectories, which are always delimited by reverse slashes (“\”) and the result always uses upper-case characters.

Prototype `int getcwd(char* name, int len);`

Parameters

<code>char* name</code>	Name of the subdirectory affected.
-------------------------	------------------------------------

Return Values

Success:	0
----------	---

Failure:	-1 with <code>errno</code> set to: ENOENT no such subdirectory. EACCES the user’s buffer isn’t accessible. EEXIST the subdirectory specified for <code>mkdir</code> already exists. EPERM the caller doesn’t have access to this drive. ENFILE no file handles available at this time. ENOSPC out of memory.
----------	--



System Configuration File

Verix eVo-based terminals use the default system file `CONFIG.SYS` to configure the system environment. A terminal's end user can add or change `CONFIG.SYS` entries using the VTM file editor or a download utility. An application can read and write to the file through the `get_env()` and `put_env()` library routines.

There can be up to 15 `CONFIG.SYS` files; one for each file group in use. Most variables that control terminal properties are in the Group 1 `CONFIG.SYS` file (applications running in other file groups cannot change these system properties). Unless otherwise noted, references in this chapter to the `CONFIG.SYS` file assume the Group 1 file.

`CONFIG.SYS` is a compressed ASCII format file maintained as a keyed file (refer to [Keyed File Reads and Writes](#)). Due to restrictions on compressed data, `CONFIG.SYS` values are limited to 128 characters, and can only use ASCII characters in the range 20h – 5Fh. This excludes lowercase letters and some punctuation characters.

NOTE



The 32- and 128-byte maximums will always work, but it is possible to create and use files containing longer records with careful, well-planned write and access structure. While Verix eVo file system does not reject technically over-limit records, it also does not prohibit them.

Ordinary `CONFIG.SYS` entries are erased when a full download to the terminal is performed.

NOTE



Entries with a key name that begin with a pound sign (#) are preserved during downloads. `CONFIG.SYS` entries that begin with an asterisk (*) are also preserved during full downloads.

Environment Variable Descriptions

This section discusses the `CONFIG.SYS` environment variables, their descriptions and usage. `CONFIG.SYS` files residing in other groups that are for application use only are also discussed here.

CAUTION



Do not create new '*' `CONFIG.SYS` variables. These variables are reserved for the OS, and new '*' variables may be developed that would overwrite those developed for an application. Existing '*' `CONFIG.SYS` variables can be modified to suit an application's purpose. Variables can be defined for use with an application, but should not begin with '*'.

***AKM** Sets cell phone mode available on terminals that support it. Values are CP (cell phone) or CLASSIC.

***APNAME** In the ENHANCED UI, this is the name of the application residing in a GID. During FULL download, *APNAME is displayed in the confirmation process. If *APNAME is not defined, APPLICATION is displayed, instead.

***ARG—Arguments** *ARG contains the `argc` and `argv` arguments passed to the main program specified by the *GO variable. Thus, it plays the role of the command line arguments of more conventional systems. Multiple arguments can be passed by separating them with spaces.

Example If *ARG = "-V 256", the main program receives `argc = 3`, `argv[1] = "-V"` and `argv[2] = "256"`. `argv[0]` contains the name of the program file.

The use of *ARG is optional.

***B—Communication Device Buffers** The system maintains a set of memory buffers for communications device I/O operations (RS-232, PIN pad, modem, and so on), minimum value is 1, maximum is 256 (default). All I/O operations simultaneously share the communications buffer pool. Increasing the number of communication buffers improves I/O function performance, but it also increases memory use. The programmer must decide which is more important to the application.

*B accepts a decimal number, indicating the number of communication buffers the system maintains. For example, *B = 24 assigns 24 buffers; each buffer is 64 bytes long. Non-communications devices (keyboard, card reader, and so on) do not use the buffers allocated by *B and are not affected by this variable.



WARNING Limiting *B may cause indeterminate side effects—under certain conditions, a known side effect of setting this variable is the failure of modem profiles to load properly. This CONFIG.SYS variable is meant for use in applications where memory is limited and should not be set if memory is not limited.

***BCM** Backward Compatibility Mode (BCM) for Predator terminals. This variable is not used in Trident. This is used on the V*670 terminal, which has a larger display, to emulate the display of the V*510 for compatibility between the applications that run on both terminals. Valid values are:

- 0 = No BCM
- 1 = BCM Mode
- 2 = BCM mode with arrows

CHKSUM—Checksum Control By default, each time the system starts the application, it validates the checksum of all the files in the file system (on power-up or following a restart from VTM). The user can bypass automatic checksum verification by assigning a value of 2 to CHKSUM.

If `CHKSUM = 2`, the system bypasses checksum validation at all times. All other `CHKSUM` values are undefined.

If automatic checksum validation is disabled, the application can check the integrity of its files using the `SVC_CHECKFILE()` function.

NOTE



`CHKSUM` does *not* begin with an asterisk and is deleted from `CONFIG.SYS` on a full download. This ensures that checksum validation is not disabled when a new application is loaded into the system. Disabling checksums removes an important check on system integrity and should be used only in very unusual circumstances.

- *COM1RB** The `*COMnRB` value specifies the size of the read buffer for the COM1, COM2, COM3, and COM8 UART devices for Predator terminals. Trident terminals are set at the max size and ignores the variable. When set, the value is the number of bytes for the FIFO received. The maximum size is 8192 bytes.
- *COM2RB** See description of `*COM1RB`.
- *COM3RB** See description of `*COM1RB`.
- *COM8RB** (Vx700 only). See description of `*COM1RB`.
- *COMBO** Sets the application group to use a modem or TCP/IP. 0=modem,1=TCP/IP
- COM2HW** This variable is not supported in Trident. In Predator, if this variable is not present, OS performs default module detection at power up. If the variable is present, the OS uses this value as the value represents the type of modem installed and bypasses the module detection process.

The value of this variable is erased by full downloads as it is an ordinary variable, this forces re-detection of the module when the module is replaced.
- COM3HW** This variable is not supported in Trident. In Predator, If this variable is not present, the OS performs default module detection at power up. If the variable is present, the OS uses this value as the value that represents the type of modem installed and bypasses the module detection process.

The value of this variable is erased by full downloads as it is an ordinary variable, this forces re-detection of the module when the module is replaced.
- *CPAD** Time between key presses (cell phone mode). Value is between 0 - 10000, default is 1500 milliseconds (1.5 sec).
- *DARK** Sets the computed print strobe time to control darkness for printing graphics or characters on the integrated thermal printer, refer to `<ESC>w<n>;`. The higher the number, the longer the print strobe activation time, the darker the print and the higher the print times, and power consumption.

***DBMON** Configuration for debug monitor. Format is `pb` where

- `p` is the port:
 - 0 = USB
 - 1 = COM1 (default)
 - 2 = COM2
- `b` is the baud rate as set with the following values:

Baud	Value	Baud	Value
300	0	38400	7
600	1	57600	8
1200	2	115200	9 default)
2400	3	12000	10
4800	4	14400	11
9600	5	28800	12
19200	6	33600	13

NOTE



The baud setting is ignored when `p` is set to 0 (USB).

***DEBUG** Sets viewing of OS debug trace.

***DEBUGO** May be used to enable additional logging, but should not be used in production since they will rapidly fill the log.

***DEBUGT** Filters for the `*DEBUGT` log output. The task-IDs to be watched can be restricted by specifying an upper bound, and the types of records that can be selected. For example:

`*DEBUGT=3-6:WEIRD`

For this setting, the "-6" restricts the range of task IDs to 4, 5, 6. The types of records to be monitored are selected here with "":WEIRD" which represents all currently available types:

- W = Write
- E = Environment (`get_env` and `put_env`)
- I = IOCTL (status and control)
- R = Read
- D = Directory (rename and remove)
- C = Close
- O = Open

Note that if you specify W, I, R, or C, you must also specify O, since the handle is pretty meaningless without the file name. The list of types can be given in any order. Setting `*DEBUGT=1:DOC` would capture all directory, open, and close calls for all tasks except system mode.

***DEFRAG— Defragment Flash**

Sets automatic flash defragmenting on terminal start up. This helps to avoid running out of usable flash memory. The flash file system is checked for deleted files and optionally coalesced (defragmented), according to the following rules:

- `*DEFRAG` is not defined or set to 0: Coalesce flash if deleted files found.
- `*DEFRAG` is > 0 : Coalesce flash if freed memory space would be \geq `*DEFRAG` (value in KB).
- `*DEFRAG` is < 0 : Never coalesce flash. In this case, the VTM flash defragment function is only available manually.

`*DEFRAG` only works in Predator. If `*DEFRAG` is used in Trident `*DEFRAG` returns `success` but no action.

Example

```
*DEFRAG = 0      Always coalesces flash when deleted files are detected.
*DEFRAG = 250    If coalesce will free 250 KB or more memory, then coalesce flash.
*DEFRAG = -1     Never coalesce flash.
```

***DIAG** Executes a diagnostic or key-loading program *once* in terminal VTM; allows a diagnostic program to run *once* when security is preventing access.

***DIRCOM** Defines the communication port for `tmdirlist` output.

***DOT0** This variable sets the maximum number of simultaneous dots printed when the `nRAD_HPWR` signal line is low which indicates that the radio is drawing high current and printer slows down its printing consuming less power, by default the variable is set to 16. This parameter has a range of 16 to 64.

***DOT1** This variable sets the maximum number of simultaneous dots printed when the `nRAD_HPWR` signal line is high which indicates that the radio is not drawing high current and the may print at full speed, by default the variable is set to 40. This parameter has a range of 16 to 64.

***ESDMON** This variable is currently available in Trident only. This variable turns on the client side ESD monitoring function, and works with `*USBRESET`. The settings are:

- 0 = No SOF reset, no `-EPIPE` and yes suspend reset.
- 1 = No SOF reset, yes `-EPIPE`, yes suspend reset.
- 2 = Yes SOF reset, yes `-EPIPE`, yes suspend reset.

***ETHSPD** This value configures the Ethernet port for either 10 MBPS or 100 MBPS and full or half duplex.

- 1 = 10 MBPS PHY rate, half duplex, auto-negotiation off.
- 2 = 100 MBPS PHY rate, half duplex, auto-negotiation off.
- 3 = 10 MBPS PHY rate, full duplex, auto-negotiation off.
- 4 = 100 MBPS PHY rate, full duplex, auto-negotiation off.

*FA—File Authentication

If this variable is present and set to 0, all signature files are removed from terminal memory. If this variable is set to 1, all signature files are retained in terminal memory. It is important to retain signature files if planning back-to-back downloads. Default is 1: retain signature files.

***FILE** Maximum number of files that can be simultaneously open. Minimum is 10; maximum is 224 (default: 30).

***FKEY** Defines Backward Compatibility mode for VX 680, VX 820, and VX 825 and creates a frame around the text area to emulate the F0-F5 and the horizontal function keys. If the area where the F0 key is defined in the frame is touched on the screen the F0 key value is returned. The same applies for the other function keys.

***FK_FKEY** The BMP image filename for the column to the right of the text area.

***FK_HKEY** The BMP image filename for the horizontal key area below the text area.

***FK_TITLE** The BMP image filename for the frame title the across the top of the text area.

*GO—Startup Executable Code File

On power up or system restart, the terminal decides which program to run by looking at the `*GO` entry in `CONFIG.SYS` in file Group 1. Each group can have a `CONFIG.SYS` file, but it is desirable to allow the application with the highest privilege to control the startup of other applications. VTM gives this privilege to the `CONFIG.SYS` file in Group 1, which is determined to be the *sponsoring* application. If there is an `*ARG CONFIG.SYS` entry, then its contents are passed as command line arguments to the process. `*GO` determines action as follows:

- If `*GO = APPL.OUT`, on system restart the terminal searches for the file `APPL.OUT` and attempts to execute it.
- If `*GO` is not set at system restart, the terminal displays `DOWNLOAD NEEDED`.
- If `*GO` is not found, the terminal displays `DOWNLOAD NEEDED NO *GO VARIABLE`.
- If `*GO` is set but the `run()` system call fails, possibly because the executable file is missing, then the terminal displays `DOWNLOAD NEEDED INVALID *GO VALUE`.

For Verix eVo, VTM is enhanced to invoke “`N:VXEOS.OUT`” in group 46, prior to launching the designated group 1 application, with respect to the following restrictions:

- If there is no group 1 application available to be run, then VTM will prompt for a download and the Verix eVo application(s) will not be invoked.

- As soon as VTM has invoked the Verix eVo application it will immediately proceed to invoke the designated group 1 application.

***GKE** Indicates the type of event that has to be generated when the user has pressed green key. The default is that no event will be generated. If `*GKE = 1`, pressing the green key will trigger a console event for the current owner of the console. If `*GKE = 2`, pressing the green key will generate a system event for all applications.

***GUARD** Works with the `dir_zap` type calls such as `dir_zap_I_allgps()`. `*GUARD` defines a global “guarded mask” used to specify which groups are “guarded” from “`dir_zap*`” calls. Only groups 1 and 15 can be guarded (set `*GUARD=8002`), but by default, no group is guarded.

***HEAP** Defines the HEAP size for the system. This value should not be used without recommendation from the OS team. The value range from 1 to 512 KB, and the default value is 256. This is used mainly for the USB driver.

***IPPMKI—Internal PIN Pad Communications Parameters**

This variable sets communications parameters for the internal PIN pad key loading from VTM (see [Appendix c, IPP Key Loading](#)). The value can specify a baud rate or the following flags:

- E A7E1 format
- O A7O1 format
- D Set DTR
- R Set RTS

Order does not matter. For example, “E1200R” sets the serial port to A7E1 at 1200 baud and turns on RTS. The default settings are A8N1 (ansync, 8-bit, no parity, one stop bit) at 19200 baud.

NOTE



`*IPPMKI` refers only to settings used for the external COM1 (RS232) port where the key loading system (usually a PC running either MKIXOR or SecureKIT) is physically connected. It does not affect the (internal) physical serial channel to the IPP itself (COM2), which is accessed by applications as “`/dev/com2`.”

E, O, D, and R also set `Fmt_A7E1`, `Fmt_A7O1`, `Fmt_DTR`, and `Fmt_RTS`, respectively.

The flags and rate can be intermixed in any order. Unrecognized characters are ignored. For example:

- 1200E = 1200 baud, even parity
- ER= 19200 baud (default), even parity, assert RTS
- R,9600,E = 9600 baud, even parity, assert RTS (the commas are ignored)

`*IPPMKI` is intended to support key loading software with fixed communication requirements; the baud rate probably has no significant effect on performance given the small amount of data involved. Note that the COM1 settings are independent of the COM2/IPP settings.

- *KEYBOARD** Tells the generic USB driver which type of keyboard is being used. The values can be 0, 1 or 2.
- *LOG** May be used to reserve an area in the file system for a circular, memory-based log in kilobytes. The allowable range is between 50 and 32 x 1024.
- *LOGP** Enables selection of another serial port when retrieving system logs.
- *MA** This is set when MULTI-APP download is chosen. *ZA is set to *MA when MULTI-APP is chosen in the ENHANCED UI.
- *MAXSEM** Controls the total number of global semaphore.
- *MAXSH** MControls the total number of active shared memory handles available in the system.
- *MENUx** Used to add menus to the VTM screen system. This variable is not supported in Trident.
- *MENUxy** Used to add menus to the VTM screen system. This variable is not supported in Trident.
- *MERR** The modem profile load operation loads a file pointed to by *MN. If this file fails to load, the error code is saved in *MERR. Verify that the profile loaded correctly by checking the SYSTEM INFO system information menu. If the expected version does not display in the VER field, access the EDIT menu and review the CONFIG.SYS file for the variable *MERR. *MERR shows error codes on operation failure.

Table 13 lists the *MERR error code values.

Table 13 *MERR Values

*MERR Values	Display	Descriptions
1	EXTENSION NOT .ZIP	File defined by *MN is not a zip file
2	NOT AUTHENTICATED	File defined by *MN is not authenticated
3	*MN FILE ZERO LEN	File defined by *MN has a length of zero
4	FILE COPYING ERROR	MODPROF.ZIP does not exist
5	FILE UNZIP ERROR	ZIP file fails unzip operation
6	NAME NOT MODPROF.S37	The file within the *MN zip file is not named MODPROF.S37
7	MODEM COMM ERROR	Modem Communication error such as: <ul style="list-style-type: none"> Modem fails to respond with OK when download is completed

Table 13 *MERR Values

*MERR Values	Display	Descriptions
		<ul style="list-style-type: none"> Modem does not respond with “.” For each record written Modem does not respond as expected (AT** does not cause the “Download initiated” message)
8	MDM PROFILE MISMATCH	Modem profile does not match modem type
9	*MD UNZIP ERROR	Illegal profiles or other file types in the file pointed to by *MD

***MN** Verix eVo OS requires a country-specific configuration file. Parameters and patches to the modem firmware are included in this file. The file downloads to the modem on each power cycle or whenever a new file downloads to the terminal. *MN is set to the new filename and, once successfully loaded, *MN is removed from the CONFIG.SYS file and the configuration file is moved into memory GID0.

***OFF** An automatic transition from Idle (sleep) to OFF occurs if the unit is continuously idle for five minutes (default value), OFF variable is set to adjust this time period. This parameter has a range of 1 s to a maximum of 36,000 s (10 hr). The terminal sets to the default value of five minutes if *OFF is not present or its value is out of range.

***OFFD** Indicates the delay (in seconds) between receiving the SVC_SHUTDOWN call and powering down the terminal by default the time taken is 2 seconds. This variable is read on system restart or reboot. The range is 2 seconds (default) to a maximum of 60 seconds.

***PIPE** Controls the number of pipe handles available. Minimum is 0; maximum is 256 (default).

***POW** Indicates the amount of time (milliseconds) that Verix eVo OS waits before attempting to place the unit in sleep mode. The timer starts when all application tasks have become idle. A value of 0 indicates that the system will never enter low-power mode. The default value is sixty seconds; the maximum setting is 600000 (ten minutes).

***PRNT** If set to zero in group 1, will prevent download errors from printing.

***PROT** Similar to *GUARD but protects files beginning with # from dir_zap calls. The mask is a bit map of GIDs 1-15. Default is 8002, meaning that groups 1 and 15 are protected.

***PRTFNT** Specifies the amount of memory allocated for the printer fonts in increments of 1-KB. The number of font pages ranges from 0 to 256, and the default is 64 for backward compatibility. If *PRTFNT is set to 256, then the program can load and print font pages using 1 to 256 in the existing printer commands.

***PRTLGO** Specifies the amount of memory allocated for the printer logos in increments of 12-KB. The number of logos ranges from 0 to 10, and the default is 1 for backward compatibility. If *PRTLGO is set to 10, then the program can load and print logos using 0 to 9 in the existing printer commands.

***PW—Password** File group access password. VTM requires the entry of this password to permit access to files within the group. *PW is defined separately for each file group. *PW is not an actual CONFIG.SYS variable, although it may be set like one during downloads.

NOTE



For security, passwords are *not* stored in CONFIG.SYS.

***RKLMKI** Defines communication port options for the RKL application. The use of this parameter is the same as *IPPMKI.

***SCTOUT** Smart card time-out control. Value is 1 ms - 86400000 (1 day).

***SMDEF** Used to define a 3 key press sequence for password entry.

- 1 = ENTER+8+4 key press is password entry, ENTER+7 is disabled
- 2 = ENTER+7+5 key press is password entry, ENTER+7 is disabled

***SMDL—VTM Download** This flag enables polling for direct download during the start-up sequence before displaying the copyright screen. Supported values are:

- *SMDL=0 do not poll (default)
- *SMDL=1 (for developers only) poll for download

Other values are reserved for future use.

If set, a direct download is attempted during startup. The system looks for ENQs on the line, trying both 115200 and 19200 bps. If no data is detected, normal startup resumes.

NOTE



This option is provided as a convenience for developers. Do not enable for terminals placed into service.

***SMGIDS** Used in ENHANCED download UI to store a comma-separated list of GIDS. It is the list that is last chosen by the user during a FULL MULTI-APP download for application deletion. This can be edited by the user.

Example: `*SMGIDS = 1,3,5,11`

*SMPW—VTM Password	Setting this variable sets the VTM entry password. As with <code>*PW</code> , <code>*SMPW</code> is not stored in <code>CONFIG.SYS</code> .
*SMU2U	Defines the baud rate used for back to back downloads. The default rate is <code>Rt_115200</code> . Legal values for communication port rates are defined in <code>svc.h</code> .
*SMUI	Indicates which VTM is in use. This can be set by selecting either ORIGINAL UI or ENHANCED UI from the VTM Menu. This can also be edited directly. The values are: <ul style="list-style-type: none"> • 0 = ORIGINAL (default) • 1 = ENHANCED
*SOFT_PWR_OFF	This variable applies to V*520 Sprocket only. All other platforms will ignore this variable, the terminal will not SHUT DOWN when you press and hold the RED key when the external power pack is connected.
*SYSCHK	Specifies time (format: hhmm) for running the daily system integrity check. Default is 0314, meaning 3:14 am.
*TIME—Set Timers	Sets the number of system timers. By default, 100 timers are available. This number can be increased to 200 (maximum) by setting <code>*TIME=200</code> . Timers are shared by the OS and all user tasks. See set_timer() .
*TMA	Name of the Remote Diag to run.
*TURNOFF	Number of second it takes to press the CANCEL key to turn off battery-powered units. Values range from a minimum of 1 second to a maximum of 9 seconds. Default is 4 seconds.
*TURNON	Number of second it takes to hold the ENTER key to turn on battery-powered units. In Trident, values range from a minimum of 3 second to a maximum of 7 seconds. Default is 3 seconds.
*TZ	Allows user to specify timezone using standard codes, such as <code>*TZ=PST</code> .
*TZRULE	Allows user to specify timezone using explicit rules for daylight savings, and others.
*UNZIP2	On startup, system mode will look for <code>*UNZIP2</code> . <code>*UNZIP2</code> is similar to <code>*UNZIP</code> but has extra directives that are included to remove files. If <code>*UNZIP2</code> exists, <code>*UNZIP</code> is ignored. Both can be set for backwards compatibility. When both are downloaded to a terminal running an OS that does not include RDL enhancements, <code>*UNZIP2</code> will be ignored. When both are downloaded to a terminal running an OS with RDL enhancements, <code>*UNZIP</code> is ignored. <code>*UNZIP</code> is

processed as usual if `*UNZIP2` does not exist. The `*UNZIP2` remove file directive and remove file list are similar to the download 'R' packet which removes files during a download. These feature however, triggers file removal during `*UNZIP2` processing, not during a download.

***UNZIP— Decompress.ZIP**

During terminal startup, the terminal checks for the environment variable `*UNZIP` in Groups 1 – 15 `CONFIG.SYS` files. If `*UNZIP` is set, a zip archive file decompresses during startup. For example, if the archive `MYSTUFF.ZIP` downloads and the `*UNZIP` variable is set to `MYSTUFF.ZIP` during the download, when the terminal restarts the archive `MYSTUFF.ZIP` is decompressed. The environment variable is then deleted and the archive `MYSTUFF.ZIP` removed.

NOTE



The terminal decompresses the ZIP file regardless of the extension being used.

UNZIP—Determine Decompress Results

Limited results about the decompression can be obtained using the variable `UNZIP` in `CONFIG.SYS`, which is set to 0 when `UNZIP.OUT` starts, and to 1 on successful conclusion. Note that `*UNZIP` AND `UNZIP` are two different variables. See [Determine UNZIP Results](#) for more information.

***USBCLIENT**

Determines the type of USB client device. This is allowed to have two values — HID and RS-232. If no value is defined for the `*USBCLIENT` environment variable, the default is set to HID for Trident terminals, and RS-232 for PIN pads.

***USBGDEV**

Defines the device name used by the generic USB driver. Up to 4 device names can be listed, separated by commas. The total string must be less than 40 characters in length.

***USBGPID**

Defines the Product ID used the generic USB driver. Up to 64 character PIDs can be defined, separated by commas.

***USBGVID**

Defines the Vendor ID used by the generic USB driver. Up to 64 character VIDs can be defined, separated by commas.

***USBRESET**

Controls the reset hold time of `nuSB_DEVICE_RSET`. The hold time defaults to `*USBRESET=1` and the value can range from 1 to 20000, each unit represents 300uS hold time, for example `*USBRESET=20` is 20*300uS.

***VALID—List Groups to Search**

List additional groups to search on terminal startup as part of the VeriShield file authentication process. By default, the Verix eVo operating system looks in all groups for new certificate files and signature files. `*VALID` can limit the search to only the specified groups. Use `*VALID` to request that other groups be searched by providing a comma-delimited list. For example, to search Groups 2, 6, and 15, use: `*VALID=2,6,15`.

***VSOPATH** The OS looks for the VSO libraries in the same place and group where the application is, if not found, the OS searches the path specified by the `*VSOPATH` config.sys variable of the application's group. The `*VSOPATH` follows the same rules for a valid path and maximum length as specified in "VxEOS Core OpSys Enhancements – Phase 2", separated by semicolon. For example:

```
*VSOPATH="I:15/;F:15/;I:10/subdir1/"
```

The OS searches the root directory of drive I: group 15, then, the root directory of drive F: group 15, then, subdir1 of drive I: in group 10.

***WEAR** This variable is for internal use only. This variable is in `CONFIG.SYS` GID1. Please do not use.

***Z Series— ZonTalk 2000 Control**

Sets the COM port rate for downloads (see [Appendix 10, Conexant Modems \(Banshee and Eisenhower\)](#)). The following four entries in the `CONFIG.SYS` file controls baud rates for application downloads (*only*) with VeriCentre download application:

- `*ZA=xxxxxx` VeriCentre application ID (name).
- `*ZP=xxxxxx` VeriCentre download host telephone number. Can use embedded dialing control characters. This variable is used to hold phone numbers (must be a valid phone number. For an IP download, a valid IP address, including port number, is required) for modem downloads or TCP/IP downloads. The `*ZP` variable should be assigned appropriately prior to selecting either modem or TCP/IP download.
- `*ZRESET` String stored in this call is the command to reset the modem to a known state before initialization or dialing out. If `*ZRESET` is empty, the VeriCentre operating system uses the `ATZ0` command as default. `Z0` will restore the modem to the profile last saved using `AZT&W0` (most likely the same as Factory profile 0, but could be anything last saved with that AT command).
Note: Use `AT&F0` to restore the modem to Factory profile 0.
- `*ZT=xxxxxx` VeriCentre terminal ID.

***ZB** Sets the maximum block size on download.

***ZDLTYPE** An OS managed variable and should not be changed by the application.

***ZDLY** A retry attempt delay, in seconds, when resumeable downloads are enabled. The default value is 5 seconds.

***ZH** This variable defines the packet size for VeriCentre downloads. The range is 1024 to 4096 bytes. If this variable is used, the buffer is allocated in the caller's heap, adequate heap space must exist.

***ZINIT** External modem initialization string.
Default is `ATM0V0&D2`.

- V0 - sets terse mode (numeric responses),
 - &D2 - drop DTR to hang up.
- *ZN** Defines the network setting for IP-based downloads.
Format: *ZN=a.b.c.d:p where the first four integers are the “dotted quad” network address and the final integer is the port number.
- *ZR** Defines the modem AT command reset code. The default is "AT&F\r". Note the presence of the AT and carriage return.
- *ZRDL** If present, this variable signals VTM to display resumeable download screens.
- *ZRESUME** Managed by the terminal and VeriCentre and should not be changed by the application.
- *ZRESULT** This variable is set at the end of a ZONTALK download to save the completion status of the download. This is useful for full downloads that restarts after the download is completed and does not return to the calling application.
- *ZRESP** Specifies the expected response from the modem on connection. Default is “CONNECT 2400”. The terminal first tries to convert the response as a numeric value, then compares it to 1, 5, or 10 to indicate success.
- *ZS** An OS managed variable and should not be changed by the application.
- *ZSSL** If set to 1 an IP download will use SSL.
- *ZSWESC** External modem flag to use ‘+++’ to escape into command mode, rather than DTR transitions. Default is DTR.
- *ZTCP** Specifies the name of an application file to run at the time of TCP/IP download.
- *ZTRY** Specifies the number of retries for resumable downloads. The range is from 0 to 30 with the default set at 5.
- *ZX** If this variable is not present and has a value other than 1, the terminal retains the last download message on the screen (including “COMM ERRORS”, “APPLICATION NOT FOUND”, “INVALID TERMINAL ID” or other VeriCentre messages). If the variable is present and *ZX=1, it subsequently checks for the existence of the message “DOWNLOAD DONE” in the final message packet sent from the download host. It reboots without waiting for the user to press a key.

Device Variables For convenience, the system library defines a set of global variables containing device names. Use these variables in place of previous `/dev` names. The device names and corresponding handles are shown in [Table 14](#).

Table 14 Verix eVo Device Handles

Device	<code>/dev</code> Name	<svc.h> Variable
Magnetic card reader	<code>/dev/mag</code>	<code>DEV_CARD</code>
Real-time clock	<code>/dev/clock</code>	<code>DEV_CLOCK</code>
Beeper	<code>/dev/stderr</code>	<code>DEV_BEEPER</code>
Console (keypad and display)	<code>/dev/console</code>	<code>DEV_CONSOLE</code>
COM port 1	<code>/dev/com1</code>	<code>DEV_COM1</code>
COM port 2	<code>/dev/com2</code>	<code>DEV_COM2</code>
COM port 3	<code>/dev/com3</code>	<code>DEV_COM3</code>
COM port 4/Integrated thermal printer	<code>/dev/com4</code>	<code>DEV_COM4</code>
COM port 5	<code>/dev/com5</code>	<code>DEV_COM5</code>
COM port 6	<code>/* com 6 */</code>	<code>DEV_COM6</code>
COM port 8	<code>/* com 8 */</code>	<code>DEV_COM8</code>
COM port 9	<code>/* com 9 */</code>	<code>DEV_COM9</code>
COM port 10	<code>/* com 10 */</code>	<code>DEV_COM10</code>
Mag card	<code>/* mag card */</code>	<code>DEV_CARD</code>
Barcode reader	<code>/* bar code reader */</code>	<code>DEV_BAR</code>
CTLS	<code>/* Contactless device */</code>	<code>DEV_CTLS</code>
USB keyboard	<code>/* USB Keyboard HID converted to make and break code*/</code>	<code>DEV_KYBD[</code>
USB host	<code>/* PP1000SE and V*810 device */</code>	<code>DEV_USBSE</code>
Semtek device driver	<code>/* Semtek device driver */</code>	<code>DEV_SEMTEK</code>
Customer smart card	<code>/dev/icc1</code>	<code>DEV_ICC1</code>
Merchant SAM	<code>/dev/icc2</code>	<code>DEV_ICC2</code>
Merchant SAM	<code>/dev/icc3</code>	<code>DEV_ICC3</code>
Merchant SAM	<code>/dev/icc4</code>	<code>DEV_ICC4</code>
Merchant SAM	<code>/dev/icc5</code>	<code>DEV_ICC5</code>
Merchant SAM	<code>/dev/icc6</code>	<code>DEV_ICC6</code>
USB External Ethernet	<code>/dev/eth1</code>	<code>DEV_ETH1</code>
USB Internal WiFi	<code>/dev/wln1</code>	<code>DEV_WLN1</code>
USB Client	<code>/dev/usbd</code>	<code>DEV_USBD</code>

Search/Update CONFIG.SYS

The system provides functions to search and update CONFIG.SYS entries:

`get_env()` Retrieves a given environment variable and its value from CONFIG.SYS.

`put_env()` Stores an environment variable and its value in CONFIG.SYS. The following restrictions apply:

- VeriCentre downloads only: Keys must be ≤ 7 bytes.
- Entries prefixed with an asterisk (*) are reserved for system use only.
- Do *not* use control codes (values between 0x00 and 0x1F).

get_env()

Retrieves the current setting of an environment variable from CONFIG.SYS.

Prototype

```
int get_env(const char *key, char *buf, int bufsize);
```

Parameters

*key	Zero-terminated string that contains the name of the environment variable to retrieve. Maximum characters allowed is 32.
*buf	Array where the current setting of the environment variable is stored.
bufsize	Maximum number of characters to store in <code>buffer</code> . Values longer than defined are truncated. The maximum length of an environment variable is 128 characters.

NOTE



The 32-byte and 128-byte maximums will always work, but it is possible to create and use files containing longer records with careful, well-planned write and access structure. While Verix eVo file system does not reject technically over-limit records, it also does not prohibit them.

Return Values

The number of characters added to the caller's buffer is returned. This is the length of the current setting, unless truncated. If no variable `key` exists, 0 returns. -1 returns on error.

The current setting is returned in the buffer passed. This buffer is not zero-terminated. The current setting is truncated if the `bufsize` passed is lesser than the length of the setting.

Notes

Each file group has a separate CONFIG.SYS file. CONFIG.SYS *always* exists in Group 1, but exists in other groups only after a password is set. An error returns when CONFIG.SYS does not exist in the group and when bad pointers are encountered.

Example

The linked code, also shown below, segment displays the value of the *GO environment variable.

```
char buf[33]; /* 32 characters + terminator */
int n;
...
n = get_env("*GO", buf, 32);
/* Terminate the value. Note that this works even if the variable */
/* does not exist and n==0. If we were not in group 1 we should */
/* also check for n==1. */
buf[n] = '\0';
printf("*GO = %s", buf);
```

put_env()

Stores an environment variable in CONFIG.SYS.

Prototype

```
int put_env(const char *key, const char *val, int len);
```

Parameters

***key** Zero-terminated string that contains the name of the environment variable to store.

***val** Array that contains the value to store. Does not need to be zero-terminated.

len Length of the value (excluding terminator).

Return Values

Success: Length of value stored.

Failure: -1, error.

Notes

If the variable already exists, its setting is changed; if not, it is created. Each file group has a separate CONFIG.SYS file. CONFIG.SYS *always* exists in Group 1, but is created in other groups only when a password is set for that group. An error is generated if CONFIG.SYS does not exist in the current group.

Example

The following code segment sets the variable VERBOSE to OFF.

```
int n;  
...  
n = put_env("VERBOSE", "OFF", 3);  
if (n < 0) error (...)
```



Multitasking

The Verix eVo operating system allows multiple application tasks to share the CPU, using simple “round-robin” scheduling. See [Figure 2](#) for a block diagram of the Verix eVo operating system.

This chapter provides an overview of multitasking and describes incorporation in your applications. It also discusses the following:

- [Tasks](#)
- [Device Ownership](#)
- [File Sharing and File Locking](#)
- [Application Threads and Semaphores](#)
- [Pipes](#)
- [Restart Capability](#)

Verix eVo Application Architecture

In the Verix eVo environment, the work performed by an application can be divided among tasks. Normally, each task performs a specialized function such as printing receipts, handling device input and output, modem communications, or controlling the overall program flow (a *main* task). When properly designed, tasks become independent, reusable *objects* that can be used as building blocks in other applications.

Tasks

Tasks within an application communicate through *pipes*, a type of software-based serial channel. For example, the application might have a main task that communicates with three other tasks: one that handles the card reader, keyboard, and PIN pad; one that prints summary receipts; and one that accesses a authorization confirmation file and obtains authorizations over a LAN.

Task Startup

In Verix eVo-based terminals, the first main task is initiated as set in the `*GO` parameter in the Group 1 `CONFIG.SYS` file. For example, if `*GO` is set to `F:MYMAIN.OUT`, the code file `MYMAIN.OUT` in the flash file system executes at startup.

NOTE



Executable files must have a valid signature file. The signature file is created when a file is authenticated during download when the P7S file is processed. The P7S file is created by the file signing process. If the file pointed to by `*GO` is not properly authenticated, the operation fails and an error is displayed on the display.

Additional tasks can be started using the `run()` system call, as follows:

```
int run(const char *codefilename, const char *arguments);
```

*codefilename must correspond to a valid code file in the *current* file group. If the file is part of the flash file system, the `F:` prefix *must* be specified. The extension `.out` *must* also be specified. The NULL-terminated ASCII argument string is parsed and converted to an array, and passed to the `main()` routine in the code file.

If an error occurs during processing of the `run()` call, a result of `-1` is returned. Otherwise, the result returned to the calling task is the task identifier for the newly created task.

Task Termination

In Verix eVo-based terminals, most tasks never terminate. In some cases however, it may be appropriate for a task to relinquish the processor permanently. The system call is:

```
void _exit(int unused);
```

NOTE



The application task should bring its activities to an *orderly state* prior to exiting. While Verix eVo OS closes any open devices and files and recovers the memory assigned to the task, it does *not* perform such application-dependent activities as logging off a network or ensuring that all data sent to the printer is printed.

Device Ownership

In Verix eVo-based terminals, if one task has opened a device and another task attempts to open that device, the second open request fails with the calling task's `errno` variable set to `EBADF`. There are several exceptions however.

First, `normal_tone()` and `error_tone()` can be called without opening the beeper device (`DEV_BEEPER`) and can be used by any task. If, however, a task opens the beeper device, it owns the beeper and no other task is allowed to call `normal_tone()` or `error_tone()` until the owning task releases the device using the `close()` call.

Similarly, the clock device (`DEV_CLOCK`) can be read at any time using the `read_clock()` or `read_ticks()` calls. This is true even if another task has issued an `open()` call for the clock device.

With the two exceptions above, the system strictly enforces device ownership. Any attempt by one task to access a device currently owned by another task results in an error, with the calling task's `errno` variable set to `EBADF`. The following mechanisms are available to the application developer to help facilitate cooperative device sharing.

- 1 First, a mechanism for the application to determine the devices present in the system is provided:

```
int get_name (int dvic_nbr, char *dev_id);
```

For devices numbered `0..31`, this call returns the name of the device as used in an `open()` statement. If the device exists, the name is placed as a NULL-terminated string in the caller's `dev_id` buffer. Otherwise, a result of `-1` is returned with the calling task's `errno` variable set to `EBADF` or `EACCES` if the caller's `dev_id` buffer is not writable.

- 2 The next service allows a task to determine which task currently owns a particular device and the handle on which it is open:

```
int get_owner (const char *dev_id, int *ownertaskid);
```

`dev_id` specifies the device (for example, `DEV_CONSOLE`). If the device is owned, it returns the handle of the open device and stores the ID of the task that owns it in the location pointed to by `ownertaskid`; if it fails it returns `-1` with `errno` set to `EACCES`, if `dev_id` is not readable or `ownertaskid` is not writable, or `ENODEV`, if the device does not exist.

- 3 Finally, the analogous service allows a task to surrender ownership of an owned device, and pass it immediately to another task:

```
int set_owner (int hdl, int task_id);
```

Once again, the `handle` parameter should represent a device (other than the console) that the calling task currently has open, and the `task_id` parameter should represent another task in the system. Ownership transfer occurs transparently, without any change in the device itself. For example, a buffered input that the surrendering task has not read is readable by the new owner.

A successful transfer is indicated by a 0 result. Failure is indicated by a `-1` result and causes the caller's `errno` variable set as follows:

`errno` set to `EBADF` Either the handle was invalid or the calling task did not own the device at the time of the call.

`errno` set to `EINVAL` The `task_id` parameter was invalid.

```
int set_owner_all(int device_handle);
```

This returns 0 if successful, and 1 if failure with `errno` set to `EBADF`. Upon successful execution, all of the owner's siblings (threads sharing the same address space) will be allowed to use the device for read, write, status, control, or close. Only the actual owner receives events. If the owner thread transfers ownership using the existing `set_owner()` call to one of its sibling threads, then the transferee will receive events for the device. If the device is closed, or if ownership is transferred to a non-sibling, then the shared ownership ceases. Normal ownership (exclusive to one thread) resumes when the owner calls either `set_owner()` or `open()`.

Temporary Device Ownership

There can only be one temporary owner. The temporary owner can only call `read()` and `write()` with the handle. It cannot use `grant_owner()`, `revoke_owner()`, or `set_owner()`. Only the original owner that called `open()` can use `grant_owner()` and `revoke_owner()`. The temporary owner cannot use any control or status functions. This prevents the temporary owner from changing the device configuration.

The owner of the device specified by `devhdl` can temporarily grant the task specified by `task_id` ownership of the device. The function is identical to `set_owner()` but the ownership is temporary and limited. The caller retains ownership control of the device, unlike using `set_owner()`.

See [grant_owner\(\)](#) and [revoke_owner\(\)](#) for more information.

Sharing the Console

The console device (display and keyboard) is not an exception to device ownership rules. Only one task at a time can use it. However, `set_owner()` is not used for the console. Instead the current owner can transfer the console to another task by calling `activate_task()`. In addition, there is a *hot key* mechanism that can transfer ownership when the user presses specified keys. See [Console Device](#) for details.

NOTE



If a hot key is defined it is used to switch to the defined task. This key is not stored in the keypad buffer. The VTM password entry key press is an example of a special hot key.

grant_owner()

Grants temporary ownership of the device.

Prototype `int grant_owner(int devhdl, int task_id);`

Parameters

<code>devhdl</code>	Verix device handle returned from calling <code>open()</code> .
<code>task_id</code>	Verix task ID of the temporary owner.

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> if the caller is not the original owner of the device, or <code>EINVAL</code> if the task is invalid.

revoke_owner()

Revokes the temporary ownership granted by `grant_owner()`. The original owner can use this function but the temporary owner cannot.

Prototype `int revoke_owner(int devhdl);`

Parameters

<code>devhdl</code>	Verix device handle returned from calling <code>open()</code> .
<code>task_id</code>	Verix task ID of the temporary owner.

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> if the caller is not the original owner of the device.

File Sharing and File Locking

One way for tasks to share information is through common files. By default, there are no restrictions on concurrent use of files. Each task has its own access (through a unique handle) to a file.

In some cases, coordination among tasks may be necessary so that, for example, one task does not attempt to read a file while it is in the middle of being updated by another task. Verix eVo-based terminals support file locking through the `lock()` function. Reads and writes to a locked file are restricted to the task that locked the file. Read or write attempts by any other task fail until the file is unlocked (`unlock()`). Failure is indicated by a `-1` return value with `errno` set to `EACCES`.

File operations that require read or write access are: `read()`, `write()`, `lseek()`, `insert()`, `delete()`, their VLR (variable-length record) and CVLR (compressed variable-length record) equivalents, and the keyed-file access methods `getkey()`, `putkey()`, `get_env()`, and `put_env()`. A call to `open()` with the `O_TRUNC` file attribute bit set also fails on a locked file. Any call to `open()` on a locked file fails with `errno` set to `EACCES`.

File operations not requiring read or write access are unaffected by file locks, including `close()`.

As good programming practice, unlock any locked files before closing them. However, when a file is closed, any lock on that handle (by the closing task) is removed.

NOTE



All file locks are cleared after a power failure or system restart. File locks apply to the entire file; record locking is *not* supported.

Task Function Calls

The function calls listed in this section control tasks.

- `lock()`
- `unlock()`

_exit()

Terminates the calling task. Task memory is reclaimed by the system; any open files or devices are closed. If called by a thread process (that is, one created by the `run_thread` call), the thread process disappears along with its stack area. When a thread process “returns” from the routine which launched it, it implicitly calls `_exit()`.

If called by a “main” process (that is, one created by the `run` call), then (a) any active threads it may have created will be terminated, and (b) the main process disappears along with all its memory (code and data).

Prototype `int _exit(int unused);`

Return Values

Success: 0

Failure: Any other value.

Example

Click the linked example to view the sample code.

***get_errno_ptr()**

The Task Control Block (TCB) contains a pointer to the `errno` value for the task. `*get_errno_ptr` returns the address of this value. Note that no check is made on the address and using the address to access `errno` may result in an error if the address is not a valid address for the task. Note the default address in the TCB is in system space and not a valid address for the task.

Prototype `int *get_errno_ptr(void);`

Return Values

Success: Pointer to `errno` value.

get_group()

Returns the effective file group membership of the calling task. Two group numbers are associated with each task:

- the original permanent group that was assigned when the process started, and
- the effective group that is initially the same, but can be changed by `set_group()`.

The effective group determines which files the task can access. There is no separate function for getting the permanent group, but it is included in the structure returned by `get_task_info()`. See also `set_group()` and `get_task_info()`.

NOTE



When process A starts process B by calling `run()`, the permanent group assigned to B is the same as the current effective group of A. Normally this means that the permanent group ID of B is the same as the group ID of its codefile, but this is not always the case.

For example, suppose process A is in group 3 and wants to execute the group 15 codefile `foo.out` using `run()`.

Method 1:

```
set_group (15);
run ("foo.out", "");
set_group (3);
```

Method 2:

```
run ("/foo.out", "");
```

If Method 1 is used, the new process has a permanent group ID of 15. If Method 2 is used, the new process has permanent group ID of 3.

Method 2 allows the group 15 codefile to see files in group 3; Method 1 does not.

Prototype `int get_group (void);`

Example Click the linked example to view the sample code.

get_native_group()

Returns the group number of the currently executing task.

Prototype `int get_native_group(void);`

Return Values Success: Group number of the current task.

get_task_id()

Retrieves the task number. Task numbers are assigned sequentially as new tasks are created. These numbers not reused when a task exits. Task 1 is the VTM program. The first application task is Task 2.

Prototype `int get_task_id (void);`

Return Values Returns the ID number of the calling task.

Example Click the linked example to view the sample code.

get_task_info()

Stores information about a specified task in the `info` structure.

`struct task_info` is defined as:

```
struct task_info {
    short id;
    char group_id;
    signed char sts;
    long stacksize;
    long datasize;
    long msecs;
    long ct;
    char path[33];
}
```

The `status` field contains one of the following codes:

Symbol	Value	Definition
TASK_READY	0	Task is ready to run when processor is available.
TASK_WAIT_EVENT	1	Task is waiting in <code>event_wait</code> .
TASK_WAIT_TIME	2	Task is sleeping in <code>SVC_WAIT()</code> .
TASK_EXIT	-1	Task has exited.
TASK_DEBUG	-128 (0x80)	Task is under control of the debugger (see TASK_DEBUG).
TASK_WAIT_SEM	32	Thread is blocked on a <code>SEM_WAIT</code> call.

TASK_DEBUG

`TASK_DEBUG` is not a status by itself; it is ORed with one of the other codes, setting the high bit of the status byte. For example, if a task being debugged is waiting for an event, the status is hexadecimal 81 = decimal -127 = 129 if cast to unsigned.

Prototype

```
int get_task_info (int id, struct task_info *info);
```

Return Values

Success: Returns the number of bytes written to `info` if successful

Failure: -1 with `errno` set to `EINVAL`: The task ID is invalid.

-1 with `errno` set to `EACCES`: The caller's `info` address is invalid (not writable).

Example

The linked code example displays the filename for all running tasks.

run()

Executes the specified program file as a new task. The string pointed to by `args` is split at whitespace characters into a list of strings that are passed to the new task through the `argc` and `argv` parameters of its main function. Following the usual C convention, the first entry in `argv` is the filename. For example, the call

```
run("test.out", "-v card.dat");
```

calls the main routine of `TEST.OUT` with the following arguments:

```
argc = 3  
argv[0] = "test.out"  
argv[1] = "-v"  
argv[2] = "card.dat"
```

There is no way to specify arguments with embedded space characters. (Quotes are treated like any other non-blank character.) If there are no arguments, `args` should point to an empty string, and not `NULL`.

The Verix eVo operating system does not record the origin of tasks and there is no special relationship between the new task and the task that started it.

Prototype

```
int run (const char *file, const char *args);
```

Return Values

Success: Returns the task number of the new task.

Failure: -1 with `errno` set to `EACCES`: Either of the two parameters cannot be read.
-1 with `errno` set to `ENOENT`: The codefile does not exist or has not been authenticated.
-1 with `errno` set to `EINVAL`: The codefile is improperly formatted or aligned.
-1 with `errno` set to `ENOMEM`: There is not enough free memory available to run it.

Note: If the codefile specifies that shared libraries are needed, all libraries must be present, properly authenticated, formatted, and aligned. A failure of any of them causes the run to fail with the appropriate `errno` setting.

Example

The linked code example runs the program specified by environment variable `*CHILD` with arguments from `*CHILD_ARGS`.

run_thread()

Executes the specified thread as a new task. Successful invocation means that the caller's virtual address space is expanded by number of bytes equal to the stacksize bytes (rounded up to a multiple of 1024 bytes), and the entire address space is visible to the caller and the new process, the new region of memory will be used for local variables by the new process.

Prototype

```
int run_thread(int routine, int parameter, int stacksize);
```

Parameters

<code>routine</code>	Name of the routine.
<code>parameter</code>	Parameter to the routine which you want to run.
<code>stacksize</code>	Size of the stack required to run the specified routine.

Return Values

Success:	Positive value: Indicates the process ID of the newly created thread.
Failure:	Negative value: Indicates various failure conditions.

set_errno_ptr()

The TCB contains an entry for a pointer to the errno value for the task. set_errno_ptr allows the application to change the pointer to a location in the current task. Note that the pointer passed in to the call must point to a valid address in the task memory space.

Prototype `int set_errno_ptr(int *ptr);`

Parameters

*ptr	Pointer to a location in the task memory space for errno.
------	---

Return Values

Success:	0
Failure:	-1: EACCES

set_group()

Changes the effective file group membership of the calling task. Tasks whose permanent group is 1, can change to any group. Other tasks can only change to Group 15 and back. See also [get_group\(\)](#).

Prototype `int set_group (int group);`

Return Values

Success:	Returns the new group number.
Failure:	1: Group is invalid (either because it is out of range or the change is not allowed). errno is not set.

Example Click the linked example to view the sample code.

Application Threads and Semaphores

Trident terminals support the scheduling mechanism that allows multiple tasks (or processes) to share the same virtual address. Since this introduces the possibility of collisions when accessing variables, semaphores are supported and the event mechanism is generalized to allow further optimization of processor usage.

The main thread executed initially via `run()`, declares the semaphore in a global memory and initializes it to 1 ("available") during startup. The name of the semaphore reflects the resource it guards.

```
sem_t sem_tran;

sem_init (&sem_tran, 1);
```

When one thread is about to update the resource, it calls `sem_wait` to ensure that no other thread is currently updating it. If necessary, it waits until the resource is free and then gains exclusive ownership and proceeds. Once it is updated, it releases the resource using `sem_post()`.

```
int process_transaction (...)
{
    sem_wait (&sem_tran);

    ...// Perform the critical updates here

    sem_post (&sem_tran);
}
```

While a thread is waiting, its Task Control Block will reflect a status of "TASK_WAIT_SEM." If multiple threads are waiting for the same resource (semaphore), the one which has waited the longest will obtain ownership when the owning thread releases the semaphore with `sem_post`.

Since threads share the same address space, to ensure that another thread is not interrupted, semaphore is used. A thread may create a semaphore which is then used, and other threads to control access.

CAUTION



- Semaphores should not be created in a thread's "local" variables (that is, within the thread's stack), as the stack will disappear when the thread exits. Also, a thread which acquires a semaphore via `sem_wait()` should be set free using `sem_post()` prior to exiting so that another semaphore which is waiting can be scheduled.
- Semaphore should never be updated directly. While the system does not prevent such updates, the integrity of the mechanism will be compromised by such access.

Semaphore Example with Deterministic Scheduling

This example uses four semaphores so that two tasks use the resource alternately. Each task uses the resource exactly once before the other task can use it.

[Example](#)

Click the linked example to view the sample code.

Semaphore Example with User Events

This example shows that the user events can be an effective synchronization tool either between threads sharing memory or between independent tasks.

[Example](#)

Click the linked example to view the sample code.

Global Semaphore

Existing semaphore calls are generalized to support arbitrary user tasks.

`sem_open()` facilitates use of the new semaphore features. Verix-style “application group” protection is implemented so that programs can remain isolated from other applications.

- `sem_open()` creates global semaphore.
- `sem_close()` closes global semaphore.

Determining Semaphore Properties

Since global semaphores are not directly accessible in user space, several system calls allow the user to “read” the current settings of the semaphore:

- `int sem_ownr_id (sem_t *sem);`
- `int sem_next_id (sem_t *sem);`
- `int sem_value (sem_t *sem);`

The “sem” parameter in each case must be an existing global semaphore that is previously opened to return the current setting of the selected field in the semaphore structure. If not, the result is -1.

If the “value” field is 0, an application which tries to wait on the semaphore will be suspended—if one or more tasks are suspended on the semaphore, the “owner” and “next” fields are used by the scheduler to manage the suspended tasks.

Each of the above case will call one new OpSys call, `int sem_prop (sem_t *sem, int type);` and this OpSys call will be used to implement the three user-level calls. Note that the `sem_prop()` call and its higher-level aliases can be used for all types of semaphores.

Configuring Global Semaphore Numbers

By default the maximum number of global semaphores is 64. The system variable `*MAXSEM` can be set to any value between 1 and 1024 in `CONFIG.SYS` for group 1.

Side Effects

Any global semaphore that a task may have opened need to be returned to the system when that task exits:

- 1 The semaphore handle must be freed and made available to others.
- 2 If the semaphore itself is not being used by any other tasks, the semaphore must be freed so that it can be opened again.
- 3 If the semaphore is being used by other tasks, then ensure that the exiting task is not waiting on the semaphore. If it is waiting, then it must be removed from the semaphore waiting structure so that the semaphore can still be used by others.

This situation can occur if the exiting task has sibling threads—if one of the siblings cancels this thread, or if the original thread exits, then this thread will be forced to exit even though it is waiting on a semaphore. Note that `thread_cancel` feature can also require non-global semaphores to be cleaned in a similar manner.

Thread Synchronization

The following calls are used for thread synchronization:

- `thread_cancel()` forces a thread to exit.
- `thread_join()` waits until a thread exits.
- `thread_origin()` determines the oldest thread in the process.

APIs

The following APIs are also discussed below:

- `open()`
- `sem_init()`
- `sem_open()`
- `sem_close()`
- `sem_post()`
- `sem_prop()`
- `sem_wait()`

sem_init()

Initializes a semaphore to the value given. Typical value for mutex-style semaphores is “1” indicating that the semaphore is currently available, or “0” which allows for immediate blocking.

Prototype `int sem_init(sem_t *sem, unsigned int value);`

Parameters

<code>sem_t *sem</code>	Pointer to a semaphore structure.
<code>value</code>	The initial value for semaphore.

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: EACCES if the "sem" is not in user space. EINVAL if the "sem" is improperly aligned.

CAUTION



Use `errno` with caution as it is a global variable shared by other threads.

sem_open()

Opens a global semaphore before it can be used. Global semaphores are maintained in system memory outside of user space.

Prototype

```
sem_t * sem_open(char *id, int sbz);
```

Parameters

id	Identifies the semaphore. This is a standard null-terminated string of up to 16 characters excluding the terminating null. This is not case-sensitive.
sbz	Currently not in use, should be set to zero (0).

The name space of the identifier is universal. Any program in any group may open and use the semaphore. However, the creator of the name may restrict its use based on application group. This is achieved by specifying a numeric group restriction prior to the name, followed by a forward slash (/) character. For instance, the semaphore “1/foo” is visible only to applications having access to group 1; the semaphore “foo” is visible to all applications.

NOTE



The semaphore “/s” is not the same as the semaphore “15/s” though both are private semaphores restricted to those with access to group 15. Similarly, semaphore “1/foo” and 0001/foo” are distinct private semaphores restricted to those with access to group 1.

The return code is of type `sem_t*`, and can be used when calling the functions `sem_post`, and `sem_wait`. This is not a conventional pointer to memory, so users should not attempt to access the elements of the semaphore directly. Such access will cause the program to fail with a data abort.

The function `sem_init` should not be used to initialize a global semaphore. When first created a global semaphore will have a “value” of 1; an application calls `sem_wait` will not wait but proceed immediately.

Return Values

Failure:	-1 with <code>errno</code> set to:
	<code>ENOMEM</code> , all global semaphores have already been used.
	<code>EACCESS</code> , user’s “id” parameter is not accessible.
	<code>EINVAL</code> , user has requested a global semaphore for an improper group, or the user’s ‘id’ is too long.

The value returned by `sem_open()` is a kind of “handle” for the global semaphore. If two callers open the same global semaphore, two different handles will be provided. This is similar to the situation when two applications open the same file and obtain two different file handles.

sem_close()

Returns the global semaphore handle to the system. It is good practice to close the handle once it is no longer since there are only limited number of semaphore handles. The global semaphore itself will remain as long as there are active semaphore handles for it.

Prototype `int sem_close(sem_t *handle);`

Parameters

handle	The semaphore handle.
--------	-----------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: EINVAL if it returned an invalid semaphore. EBADF, if the caller does not own the semaphore.

sem_post()

Allows a waiting task to proceed, if that task is waiting on this semaphore. If no task is waiting, it increments the semaphore.

Prototype `int sem_post(sem_t *sem);`

Parameters

<code>sem_t *sem</code>	Pointer to a semaphore structure.
-------------------------	-----------------------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: EACCES if the "sem" is not in user space. EINVAL if the "sem" is improperly aligned.

CAUTION

Use `errno` with caution as it is a global variable shared by other threads.

sem_prop()

A Trident only function that allows visibility of global semaphores to the calling task. The call also works on private semaphores.

Prototype `int sem_prop(sem_t *semaphore, unsigned short mode);`

Parameters

semaphore	Semaphore to be operated on.
mode	operation to perform on the select semaphore: <ul style="list-style-type: none"> • 0: return sem->ownr_id; • 1: return sem->next_id; • 2: return sem->value;

Return Values

Success:	Dependent on the mode selected.
Failure:	-1

sem_wait()

Causes the calling thread to be suspended if the semaphore is unavailable. Once the semaphore is available, it gives the control of the semaphore to the caller and returns.

Prototype `int sem_wait(sem_t *sem);`

Parameters

sem_t *sem	Pointer to a semaphore structure.
------------	-----------------------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to: <ul style="list-style-type: none"> EACCES if the "sem" is not in user space. EINVAL if the "sem" is improperly aligned.

Note: `Errno` is a global variable shared by other threads.

CAUTION



Use `errno` with caution as it is a global variable shared by other threads.

thread_cancel()

This can be used to force a sibling thread to exit.

WARNING

Use this call only rarely, if at all, as there are likely to be unintended consequences and undesirable side effects. If a multi-threaded application is sufficiently complex that this kind of feature appears to be needed, we recommend that a more orderly mechanism be included with the application—user events, pipes, or even shared variables offer ways for one thread to request an orderly shutdown from another.

Prototype `int thread_cancel(int thread_id);`

Parameters

<code>thread_id</code>	ID of thread to cancel.
------------------------	-------------------------

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>ESRCH</code> , no such thread is found.

thread_join()

This can be used to wait until a sibling thread exits. When the targeted sibling thread exits, its exit code can be provided to the caller of `thread_join`. If this is desired, the caller should use the `retval` parameter to specify where the exit code should be delivered. If `retval` is 0, the exit code will not be provided.

It is recommended that the exiting thread explicitly call `exit` in order to pass a return value to the joining thread.

NOTE

The location of `retval` may be in scope at the time the call is invoked but not at the time the call completes. In this case, the exit code cannot be returned to the joining thread.

Verix threads return integers, not void pointers, so this call is not completely POSIX-compliant.

Prototype

```
int thread_join(int thread_id, int *retval);
```

Parameters

<code>thread_id</code>	ID of thread to cancel.
<code>retval</code>	The return value of the function.

Return Values

Success:	0, thread has exited.
Failure:	-1 with <code>errno</code> set to: <code>EDEADLK</code> if thread X tries to join thread Y, while thread Y has already joined thread X. <code>ESRCH</code> if thread does not exist. <code>EINVAL</code> if thread is not a sibling or another thread has already joined.

thread_origin()

This can be used to determine the task identifier for the first thread in a process, that is, the one which was started using `run()` instead of `run_thread()`.

Prototype `int thread_origin(void);`

Return Values The return code is the task identifier for the original thread in the caller's process.

Pipes

A pipe is a bidirectional communication channel connecting two tasks. One task communicates with another by writing to and reading from the pipe that connects them. A pipe behaves much like an RS-232 cable connecting two terminals: Data written into the pipe at one end are read out at the other end. Both ends can be read from and written to simultaneously (full-duplex operation). Pipes provide a clearly defined communication mechanism that avoids hidden interactions between tasks.

Pipes are *dynamic*. They are created when opened and deleted when closed. They do *not* remain intact through terminal resets or power cycles. Any data in a pipe are lost after a terminal reset or power cycle. When the application restarts and reopens the pipe, the pipe is recreated. Two types of pipes are available:

- Character
- Message

A *character* pipe treats the data going through it as individual bytes. When a task writes a buffer to the pipe, the operating system writes one byte at a time into the pipe, and reads from the pipe in the same way. At the application level, the task can (and should for efficiency) read and write buffers of several characters at a time.

A *message* pipe treats each write buffer as a single entity; each item in the pipe is a *complete* message. When a task writes a buffer, the operating system writes the entire buffer at once. When a task reads from the pipe, an entire message is removed from the pipe and copied to the read buffer. If the read buffer is not large enough to hold the message, the operating system transfers as much of the message as possible; the rest is discarded. One feature of the message pipe is that the size of the message is preserved from the writer to the reader.

The number of pipes allocated by the operating system is determined by the environment variable `*PIPE` in the `CONFIG.SYS` file (see [System Configuration File](#)). By default, if `*PIPE` is not set or is set to an invalid value, the number of pipes allocated is at maximum 256.

Always check result codes to detect *insufficient buffers*. It is well worth intensive planning and testing to verify that the design has adequate buffers. Leaving `*B` set to the maximum default usually provides sufficient buffers, but the system can still consume the maximum buffers if, for example, a task is using all buffers to write a very long receipt. This may take a few milliseconds, but consumes many buffers. Actually printing the data frees buffers, but prints take longer, which frees the buffers more slowly as printers are relatively slow.

NOTE



Buffers are the same resource used in UART in other devices. There are a total of 256 buffers, where each buffer is 1024 bytes in Trident terminals and 256 bytes in Predator terminals. There is some overhead in the buffers to provide linkage to chained buffers so not all of that space is available.

In practice, most tasks do not need more than one pipe, and some tasks do not need any. To conserve memory space, determine on a system basis how many pipes are required and ensure that the `*PIPE` variable is set accordingly.

In a multi-application environment it may be necessary to know reliably which application sent an inter-process message. The Verix eVo environment is provided with an “extended message” option so that incoming messages automatically include a header generated by the OpSys at the time the message is sent.

Pipe Header Format

An “extended message” option is provided for the Verix eVo OS so that incoming messages automatically include a header generated by the OpSys at the time the message is sent. The standard header file `svc.h` defines the structure type `pipe_extension_t` as follows:

```
typedef struct {
    short sndr_pipe_id;
    char sndr_task_id;
    char sndr_group_id;
    unsigned long sndr_time;
    char reserved[8];
} pipe_extension_t; // Prefixed when writing to extended msg pipes
#define PIPE_EXT_SZ sizeof(pipe_extension_t)
```

Refer to [pipe_init_msgX\(\)](#) and [read\(\)](#) for more information on pipe header.

Pipe Interface

The application interface to pipes is a subset of the standard file system interface. It supports the standard `open()`, `close()`, `read()`, and `write()` function calls. There are also pipe-specific function calls, that allow configuration of the type of pipe needed and connect one pipe to another.

Configure the Pipe

Below are functions used to configure the pipes.

Prototype

```
int status = pipe_init_msg (int pipehandle, int maxmessages);
int status = pipe_init_char (int pipehandle, int maxcharacters);
```

Parameters

<code>maxmessages</code>	Specifies the maximum number of <code>write()</code> operations that can be made to the message-type pipe at one time (that is, before any of the buffered messages are read).
<code>maxcharacters</code>	For character-type pipes, determines the maximum number of characters that can be written to the pipe before any characters are read.

Return Values

Success:	0
Failure:	−1, with <code>errno</code> set to <code>EBADF</code> : The pipe is not open.

Pipe Function Calls

This section presents the pipe-specific function calls, they are as listed below:

- `pipe_connect()`
- `pipe_init_char()`
- `pipe_init_msg()`
- `pipe_init_msgX()`
- `pipe_pending()`

You may also refer to these APIs:

- `open()`
- `close()`
- `read()`
- `write()`

Error Conditions and Error Codes

Errors are reported by returning a result of -1 with `errno` set to a specific standard error code. The caller will receive error codes in the following situations:

ENOMEM	Not enough memory to allocate for the FIFOs.
EBADF	Pipe not owned by caller or is not configured. (message pipes only): Not enough memory to allocate for the message.
EPIPE	Pipe not connected or connected to a task that exited.
ENOSPC	Pipe is full, or (message pipes only) not enough system buffers available to complete the operation.

pipe_connect()

This is the *plumbing* operation. It allows data to be written using one handle, and subsequently to be read using the other handle.

Prototype `int status = pipe_connect(int pipehandle, int targetpipehandle);`

The second handle must be obtained from a separate `open()` command.

As an example, consider the configuration in [Figure 3](#). Suppose that each task A, B, and C generate one request message to Task D and wait for a single-message response from D before proceeding.

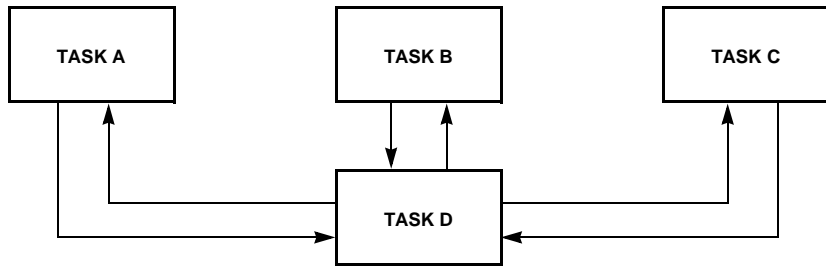


Figure 3 Tasks Opening Pipe Handles

For simplicity, assume that TASK D starts first and issues the following:

```
p0 = open ("P:REQUESTS", 0);  
pipe_init_msg (p0, 3); /* Up to three messages may be received */
```

To allow the other tasks to connect to pipe `p0`, the handle value *must* be communicated to them. There are several ways this can be done:

- 1 If the tasks know the name used to open the pipe they can call `get_owner()`.
- 2 If the other tasks are started by TASK D, it can pass the value of `p0` to them as a program argument.
- 3 TASK D could write the value to a file with an agreed upon name that the other tasks could read.
- 4 TASK D could set a `CONFIG.SYS` variable to the handle value that the other tasks could retrieve.

Each of the other tasks can then perform the following operations:

```
p = open ("P:", 0);      /* Obtain anonymous pipe handle */
pipe_init_msg (p, 1);    /* Only needs to be able to
                           receive one message */
pipe_connect (p, p0);     /* When we write to our pipe, it
                           will be read using p0 */
```

Finally, when one of the tasks (A, B, or C) is ready to send a request message to TASK D, it writes to its handle a message identifying its source that is, it includes the handle within the message. Then, for TASK D to reply, it first connects the two pipes:

```
pipe_connect (p0, p);
```

It then writes to p0, and the data is read by the corresponding task.

Parameters

pipehandle	Handle received in response to open().
targetpipehandle	Destination (the handle of a pipe opened by another task).

NOTE



In some instances it can be useful to write messages for future retrieval using a pipe. It is possible to write pipe messages to yourself by connecting one of your pipe handles to the same pipe or to another one owned by your task.

Return Values

Success:	0
Failure:	-1 with errno set to EINVAL: Target is invalid or the pipe is not configured. -1 with errno set to EBADF: Caller does not own the pipehandle, targetpipe is not open or is not configured.

Example

Click on the linked example to view the sample code.

pipe_init_char()

Initializes the character mode pipe. `pipe_init_char()` configures the open pipe associated with `handle` as a character mode pipe, and sets the maximum number of unread bytes it can hold.

See also, [pipe_init_msg\(\)](#).

Prototype `int pipe_init_char (int hdl, int max_input_chars);`

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: Invalid pipe handle.

-1 and `errno` set to `EINVAL`: Invalid buffer size.

-1 and `errno` set to `ENOSPC`: Insufficient buffer space available.

pipe_init_msg()

Configures a message mode pipe. `pipe_init_msg()` configures the open pipe associated with `handle` as a message mode pipe, and sets the maximum number of unread messages it can hold. (The number of bytes in those messages is limited only by internal buffer space.)

See also, [pipe_init_char\(\)](#).

Prototype `int pipe_init_msg (int hdl, int max_input_msgs);`

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: Invalid pipe handle.

-1 and `errno` set to `EINVAL`: Invalid buffer size (negative).

-1 and `errno` set to `ENOSPC`: Insufficient buffer space available.

pipe_init_msgX()

This call functions exactly as the existing call `pipe_init_msg()` except that it also configures the pipe to receive the `pipe_extension_t`. See [Pipe Header Format](#).

A pipe configured to receive these headers can connect to any other message pipe, whether or not that pipe is configured to receive these headers. Similarly, any other message pipe can connect to this pipe.

Whenever another pipe is used to write a message to this pipe, the OpSys adds the new 16-byte header. The fields in the header is filled in by the OpSys to reflect the sender's data and the time of the write. The time corresponds to the value returned by the `read_ticks()` system call, that is, time in milliseconds since the system was started.

Prototype

```
int pipe_init_msgX(int pipe_handle, int max_messages_pending);
```

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: Invalid pipe handle.

-1 and `errno` set to `EINVAL`: Invalid buffer size (negative).

-1 and `errno` set to `ENOSPC`: Insufficient buffer space available.

pipe_pending()

Tests pipe data availability and returns the number of bytes available to be read from the pipe specified by `handle`.

Prototype

```
int pipe_pending(int hdl);
```

Return Values

Success: Number of bytes currently in the buffer.

Failure: -1 and `errno` set to `EBADF`: The handle is not valid, if is not owned by the caller, or is not configured.

Example

Click on the linked example to view the sample code.

Restart Capability

Although it is not specifically related to multitasking, the `SVC_RESTART` function provides another way for one task to start another. In this case, a complete terminal reset is performed, so all pending tasks are destroyed before a new one begins.

SVC_RESTART()

Performs a complete terminal reset.

Prototype

```
int SVC_RESTART(const char *filename);
```

Parameters

`filename` Complete name of an executable file (including the `.out` suffix) or null string.

Return Values

Success: If the specified filename is found, the function does not return, a system restart is performed and the specified file executes.

If the specified filename is a null string (""), `SVC_RESTART()` behaves like a power reset. That is, the specified application executes through the `*GO` variable in `CONFIG.SYS`.

Failure: `-1`: The specified filename was not found.
`-EINVAL`: The caller's filename buffer is not readable.

Note: For historical reasons, `errno` is not set in this call. Note too, that at this point it is not verified that the indicated program *can* be run -- this is just a rudimentary check.

For example, suppose the user calls

```
SVC_RESTART ("config.sys", "");
```

The call is accepted, but during the restart when the system attempts to `run("config.sys", "")`, it can fail for a variety of reasons. (See [run\(\)](#) for details on all that can go wrong!)

For example, it's possible that if it was verified at this point, there might not be enough memory for it to run, but there will be enough on restart.

NOTE



`SVC_RESTART()` can not be used to start a rouge application. The application started must meet the usual authentication requirements to start. It is not advisable to restart if all tasks have not been given an opportunity to close properly. `SVC_RESTART()` stops any executing task, close all resources no matter what state they are in and restarts the operating system. Pending communication operations, radios, and the like will be restarted and any data that has not been read is lost.



Event Handling

Verix eVo applications can synchronize with external events using several event-oriented services of the operating system. An *event* can be posted to an application for various reasons—most commonly to indicate that input data arrived and can now be read.

One of the most significant benefits of the event mechanism is that it allows an application with no work pending to *sleep* safely. Once work arrives, the application “wakes up” and performs the required tasks. This frees the processor so that it is available for other tasks that may have work to perform.

The operating system maintains a 32-bit event *word* for each application program. Individual bits within this word are set to 1 whenever the device corresponding to that bit posts an event—generally speaking, when input has arrived from that device. The event is only posted to the program that opened the device.

The function most directly involved in this process is `wait_event()`. Normally, an application issues the `wait_event()` call when it is ready for more input. The application is suspended until some device receives input, then it resumes processing, having received the event word that identifies which devices received input.

The `wait_event` clears the event word and waits until an event is posted. Once an event is received, the application resumes at the point of the `wait_event` call. The application immediately examines the event word to see what event was received. If not the expected event then the application can return to the `wait_event` state.

In some cases, the application can opt to issue a `peek_event()` call first. This retrieves a copy of the event word, but does *not* clear it to 0. Also, it does not suspend the application even if no event occurred (in this case, the result is 0).

The third member of this family is `read_event()`. `read_event()` returns and clears the event word much like `wait_event()`, but does not wait if no event is pending.

Table 15 lists events that may be reported through a `wait_event()` call. Event names and the corresponding event word bits are defined in `<svc.h>`.

Table 15 **Defined Events**

Name	Device	Description
EVT_ACTIVATE	Console	Console ownership returned to application.
EVT_BAR	Bar Code Reader	Input available.

Table 15 **Defined Events** (continued)

Name	Device	Description
EVT_CLK	Clock	1-second event (only available to the task that opened the clock).
EVT_COM1	COM1	Input available on COM1.
EVT_COM2	COM2	Input available on COM2.
EVT_COM3	COM3	Input available on COM3.
EVT_COM4	COM4	Input available on COM4.
EVT_COM5	COM5	Input available on COM5.
EVT_COM6	COM6	Input available on COM6 on terminals that support USB to RS-232 module and the converter module of the Qx120 Contactless device.
EVT_CONSOLE	Console	Display output complete.
EVT_DEACTIVATE	Console	Console ownership lost.
EVT_ICC1_INS	Smart Card	Customer card inserted.
EVT_ICC1_REM	Smart Card	Customer card removed.
EVT_IFD_READY	Interface Device	Read complete from the IFD. Issued to the current owner of the IFD channel.
EVT_IFD_TIMEOUT	Interface Device	Read time-out on the IFD. Issued to the current owner of the IFD channel.
EVT_KBD	Console	Keyboard input available.
EVT_MAG	Card Reader	Input available; signaled on a card swipe. To set this trap, the card device must be open and the operating system card swipe buffer empty.
EVT_NETWORK	Network	Input available on Ethernet port.
EVT_PIPE	Pipe	Input arrived on a pipe.
EVT_SYSTEM	System	Universally interesting event.
EVT_TIMER	Timer	User-defined through the <code>set_timer()</code> function.
EVT_USB	USB	Input available on USB port. This is set in the event bit mask of all tasks whenever a USB device is connected or disconnected. The event bit is set even with no open USB device.
EVT_WLN	USB WiFi	Incoming data and PIMFOR management packets set this event.
EVT_USB_CLIENT	USB Client	Reports client events. The OS determines the type of USB client device it presents to a USB host at boot time.
EVT_REMOVED	Case Removal Latch	Notifies the OS that the keypad unit has been removed from its host system by monitoring the case removal switch.

At this time, there are more events in the 32 bit event mask in the OS. Since there is no space to include these in the standard event list, these events can be overloaded. The call, `set_event_bit()` allows the user to define an event and bit position. Some care must be taken in selecting what device event to overwrite. Some COM events may occur (for example, COM events) even if you have previously defined an event. For example, you want to create an event for the Bluetooth DUN profile and set this to `EVT_COM1`. If `DEV_COM1` is active, the application will receive events on `EVT_COM1` for both the `DEV_COM1` device and the BT DUN device.

Pre-Sleep Event

Applications/libraries need a way to put communication devices in low power mode before the terminal goes to sleep. The commands and protocols to put devices in low power mode vary from device to device.

The pre-sleep event feature allows tasks to register for an event that occurs before the terminal goes to sleep. When a task wakes up from a pre-sleep event, it can quickly send the power mode commands then call `wait_event()`. After `PRESLEEP_IDLE` ms of additional idle time, the OS enters sleep mode. The event bit is selectable per task.

The `PRESLEEP_IDLE` time is tentatively set at 100 ms. This means after the `*POW` idle time has occurred, tasks that registered for the pre-sleep event may continue to execute until all tasks are idle for `PRESLEEP_IDLE` time. See `reg_presleep()` for more information.

reg_presleep()

Allows tasks to register for a pre-sleep event that will be sent by the OpSys before the OpSys enters sleep mode.

Prototype `int reg_presleep(int bitnum);`

Parameters

`bitnum` [0..31] The event bit that the OpSys should set in the bit mask returned by `wait_event()`.

More than one task can register and each can select its own `bitnum` value. Some event bits are reserved so cannot be selected such as `EVT_SYSTEM`.

Each task so registered will receive one pre-sleep event per sleep cycle. After the pre-sleep events have been sent, the OpSys expects all awakened tasks to become idle again quickly so that the OpSys can continue the process of entering sleep mode. Once pre-sleep has started, there is no way to go out except for the OpSys to enter Sleep mode. The OpSys takes control of the keyboard once pre-sleep mode starts, so pressing keys will not prevent the terminal from entering sleep mode.

When tasks receive the pre-sleep event, they should perform only necessary work to put devices in low power mode, and then they should call `wait_event()`. Lengthy operations or waking up other tasks must not be done. After an additional period of `PRESLEEP_IDLE` milliseconds, the OpSys enters Sleep mode. This means tasks executing in pre-sleep mode should execute at least once every `PRESLEEP_IDLE` ms when performing pre-sleep work.

The `PRESLEEP_IDLE` time is the time duration when all tasks have been suspended (having called `SVC_WAIT` or `wait_event()`). When the OS determines all tasks have been waiting (idle) for 100 ms, the OS enters Sleep mode.

Example Assume that `PRESLEEP_IDLE` is 100 milliseconds. A task sends a command to a communications device and waits 250 milliseconds for the response. In this case, the OpSys puts the terminal in sleep mode after 100 milliseconds. To prevent the system from sleeping during this period, the task could instead check for the response every 50 milliseconds. This prevents the OpSys from entering Sleep mode until the response is received.

Event Function Calls

The following functions implement the exception-handling system.

- `clr_timer()`
- `peek_event()`
- `post_user_event()`
- `read_event()`
- `read_evt()`
- `read_user_event()`
- `set_host_power_event()`
- `set_timer()`
- `get_com1_event_bit()`
- `set_com1_event_bit()`
- `set_signal_events()`
- `SVC_WAIT()`
- `wait_event()`
- `wait_evt()`

clr_timer()

Cancels a timer before it expires.

Prototype `int clr_timer(int timer_id);`

Parameters

timer_id	ID of timer to cancel. This is the value returned in <code>set_timer()</code> .
----------	---

Return Values

Success:	0
Failure:	-EINVAL; This function fails when the timer ID is invalid or is in use by another task, or the associated timer has expired. errno is not set.

Example See `set_timer()` for linked code example.

peek_event()

Examines pending events.

Prototype `long peek_event(void);`

Return Values Returns the event mask for the calling task.

Notes `peek_event()` differs from `wait_event()` in the following two ways:

- If no events are pending it returns 0 instead of waiting for an event to occur.
- It does not clear the event mask.

post_user_event()

This function ORs the bits set in the `user_bits` variable with the user mask in the TCB. This causes the OS to generate an event, `EVT_USER`, indicating the user mask has changed. The user mask is 32 bits and are user defined. See [read_user_event\(\)](#) for more information.

Prototype `post_user_event(int user_task_id, int user_bits);`

Parameters

<code>user_task_id</code>	User task ID.
<code>user_bits</code>	A bit which is sent by one task to another task.

Return Values

Success:	0
Failure:	-1 with caller's <code>errno</code> to <code>EINVAL</code> , indicates that the destination task does not exist, or no bits were selected in the <code>user_event</code> parameter.

read_event()

Reads and clears pending events.

Prototype `long read_event(void);`

Return Values Returns the event mask for the calling task.

NOTE



`read_event()` differs from `wait_event()` in that if no events are pending, it returns 0 instead of waiting for an event to occur. `read_event()` differs from `peek_event()` in that it clears the mask.

read_evt()

Reads and clears pending events. This function is similar to `read_event` except that only events listed in the bit mask, `needed_events`, will be reported to the caller. Events not listed will remain available for subsequent access using `wait_event`, `peak_event`, `read_event`, `wait_evt` and `read_evt` calls.

Prototype `long read_evt(int needed_events);`

Parameters

`needed_events` It is a bit mask and the events in the bit mask causes the task to wake up.

Return Values The needed event is obtained and it returns a positive value.

read_user_event()

This call reads and clears the “user events” field for the calling task. It also resets the new `EVT_USER` bit in the calling task’s main event.

Prototype `int read_user_event(void);`

set_host_power_event()

This function call defines the event bit to use for notification of changes in power or charging. This is a Vx600 function call only.

Prototype `int set_host_power_event(long evt);`

Parameters

`evt` Event mask bit to use for power or charging status change.

set_timer()

Schedules an event to occur after a specified delay.

Prototype `int set_timer(long msecs, long eventmask);`

Parameters

<code>long msecs</code>	Delay before the event occurs. Maximum value is one day (86,400,000 ms).
<code>long eventmask</code>	Event to post when the timer expires. Normally, this is <code>EVT_TIMER</code> , which is reserved for this purpose. However, any event or combination of events can be specified. This value is ORed into the task's pending event word.

Return Values

Success	If the timer is successfully set, a non-negative timer ID is returned.
Failure:	-EINVAL invalid delay (longer than 24 hr) -ENOSPC all timers busy. errno is not set by this call.

NOTE



By default, 100 timers are available. This number can be increased to 200 (maximum) by setting the `CONFIG.SYS` variable `*TIME`. Timers are shared by the OS and all user tasks, so the number available to a particular application is less than `*TIME`.

The number of timers provided by default (100) is in fact very generous in that very few systems would ever run out of timers with this setting—if the applications are functioning correctly! Always check the result returned from this call to verify that a timer was obtained; if not, either debug the application to find why it is using *so many* timers, or increase the number using `*TIME`.

If the system *does* run out of timers, operation of the terminal may become erratic until the next restart. For example, the beeper may not turn off or protocol time-out conditions may not be handled correctly.

Example

The linked code segment waits for a key press, but times out if no key is pressed within 30 seconds; other events are ignored.

get_com1_event_bit()

This function returns the event bit used for COM1 events. This is a Vx600 only function.

Prototype `int get_com1_event_bit(int hdl);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

set_com1_event_bit()

This function call is similar to [set_event_bit\(\)](#). This sets the event bit to use for COM1 events. This is a Vx600 only function.

Prototype `int set_com1_event_bit(int hdl, long flag);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

set_signal_events()

Allows the application owning the radio to select which signal will generate an event when it changes.

Prototype `int set_signal_events(int hdl, char * signal);`

Parameters

hdl	Handle of the COM port.
signal	Pointer to the signals of interest. <ul style="list-style-type: none">• 1 = CTS• 2 = DCD

Return Values

Success:	0
Failure:	-1 <ul style="list-style-type: none">• If invalid pointer, <code>errno</code> is set to <code>EACCES</code>.• If executed on a COM port that does not support it, <code>errno</code> is set to <code>EINVAL</code>.• If open block is not set, <code>errno</code> is set to <code>EINVAL</code>.

SVC_WAIT()

Suspends the calling task for the specified number of milliseconds. It does not return until the time has passed. The function uses one of the system timers. See also [set_timer\(\)](#).

The maximum delay is 65535 ms, or a little over 1 minute. To set a longer delay, use [set_timer\(\)](#) and [wait_event\(\)](#).

`SVC_WAIT(0)` is sometimes used as a dummy system call to give the kernel the opportunity to task switch..

Prototype

```
int SVC_WAIT(unsigned int msec);
```

Parameters

msec	Delay time. Since this is a 16-bit value, the maximum delay is a little over 1 minute.
------	--

Return Values

Success: 0

Failure: -ENOSPC: All timers busy. (`SVC_WAIT()` uses one of the system timers, as does `set_timer()`.)

-EINVAL: The caller specified a value greater than 65535.

NOTE



For delays longer than 65 seconds, use `set_timer()` and `wait_event()`. `SVC_WAIT(0)` is sometimes used as a dummy system call to give the OS the opportunity to task-switch.

Example

The linked example code file shows that a common use of `SVC_WAIT()` is to briefly display a message. Note that the code does not check the return value. This is typical, although as noted above, it is possible for the call to fail.

wait_event()

Waits for an event.

Prototype `long wait_event(void);`

Return Values Returns and clears the *event mask* for the calling task (Table 15 lists event bits).

NOTE



A bit is set in the event mask for each event that occurred since the last call to `wait_event()` or `read_event()`, or the start of the program if this is the first call. Event bits are defined in `<svc.h>`.

Callers must assume that multiple event bits could be set in the mask.

Comparisons (such as, `if (mask == EVT_KBD)`) are usually a mistake. The proper test is `if (mask & EVT_KBD)`.

Robust code should consider an event as an indication that something interesting happened on a device. It must not make assumptions as to what the event was. For example, after receiving an `EVT_KBD`, the program should be prepared to handle zero, one, or multiple input characters. The following sequence is an example of the type of subtle race conditions that can arise:

- 1 User presses a key, setting `EVT_KBD` in event mask.
- 2 Program returns from `wait_event()` and clears event mask.
- 3 User presses another key, resetting `EVT_KBD`.
- 4 Program reads keyboard input and retrieves both characters.
- 5 Program calls `wait_event()`. It immediately returns with `EVT_KBD` set (from 3), but there is no data in the keyboard buffer.

Example

Event-driven programs typically have a central event loop that waits for events and dispatches them to the proper handler. The linked code example illustrates a simple event loop.

wait_evt()

This event functions similar to the existing `wait_event` except that:

- Only the events listed in the bit mask “`needed_events`” will cause the task to awake.
- Only the events listed in the bit mask will be reported to the caller. Events that are not listed will be recorded as they occur and will remain available for subsequent access using `wait_event`, `peek_event`, `read_event`, `wait_evt` and `read_evt` calls.

Prototype `int wait_evt(int needed_events);`

Parameters

<code>needed_events</code>	It is a bit mask, the events in the bit mask causes the task to wake up.
----------------------------	--

Return Values The needed event is obtained and positive value is returned.

Connection Fail-Over Management

On VX 680 3G, the OS does not guarantee that after fail-over, extraneous packets with the old IP address from an existing connection are not seen on a network. The OS does not guarantee that if the IP address remains the same after fail-over, an existing connection is maintained.

Failover support for Dual SIM

When TCP/IP is used, and network fail-over occurs from one SIM to another and the IP address changes, the OS ensures that the configured socket timeouts still apply.

Failover from GPRS to Wifi/BT

It is required to have failover from one radio technology to another, the most likely usage case is using WiFi or BT, as the primary and would failover to GPRS/3G only if / when necessary.

When TCP/IP is used, and network fail-over occurs from one radio to another and the IP address changes, the OS ensures that the configured socket timeouts still apply.



Console Device

The console device is composed of the keypad and display. These devices have many variants, for more information you may refer to the terminal Reference Manuals. The keypad is composed of the standard numeric keys (0-9), ENTER, Backspace, and CANCEL, all keypads have these keys. Key pads may also include, *, #, +, -, function keys aligned on the right hand side of the display, the screen keys along the bottom of the display, ALPHA, or a toggle key. Positioning of the screen or function keys vary in some terminals. As for the display the size and orientation varies between the terminal types. A display may or may not have touchscreen or graphic capability, it can also be colored or black and white.

Display

The VX 520 terminal has a 128 x 64 monochrome display, and a color variant in a 2.8 inches 320 x 240 x 16bit RGB color LCD, while the VX 680 has a 240 x 320 x 16bit RGB display (color). The VX 680 display is also a touchscreen. The capabilities of these displays are different and as functions used for the displays will have limitations or should not be used.

Default Font

There is one built-in font (the default font) that is in effect until an application sets another font. The appearance of this font depends on the pixel size of the terminal display. The VX 680 screen has denser pixel packing so the characters appear smaller compared to the VX 520. This font is the 6 × 8 character size of the ISO 8851 character set from 0x20 to 0xf, with the following exceptions:

- Character 0x80 is the Euro symbol (€).
- Characters 0x81 – 0x8A are used to display the battery icon on hand held units.
- Characters 0x14 – 0x15 combine to form an up arrow (similar to the arrow used in VTM).
- Characters 0x16 – 0x17 combine to form a down arrow (similar to the arrow used in VTM).
- Characters 0x18P – 0x1F are various single character arrows.

Other characters that are not defined display as a checkerboard block.

Font Files

The console driver uses the standard font files created by the Font Designer tool included in the VVDTK. An attempt to access a character not defined for the font (for example, 0xFF and font only contains 128 characters), a checkerboard character displays.

Font files have extension `.vft` or `.fon`. The kernel ensures that these files cannot shift during execution. Font files without this extension can cause unexpected behavior.

NOTE



Note that the font size 8×10 is *not* supported.

Table 16 lists the standard Verix eVo-based font files supported by the console driver, and their data and file sizes.

Table 16 Verix eVo Font Files Supported

Font Character Size (pixel col x row)	Characters per Screen (row x col)				Bytes per Character
	8-Line LCD	15-Line LCD	16-Line LCD	20-Line LCD	
6×8	8×21	30×53	16×21	40×40	6
8×16	4×16	15×20	8×16	20×30	16
16×16	4×8	15×20	8×8	20×15	32

Big Font Files

The console driver supports font files up to 65,536 characters. The VVDTK Font Designer application provides an option to merge multiple font files to create a font file with more than 256 characters.

If the font file is 256 characters or less, the console driver can retrieve characters with a one-byte index (that is, one byte written to the console results in one character displayed). If the specified font file size is greater than 256 characters, the console driver uses two bytes for every character displayed: The first byte is the high-order byte of the index. For example, in the following example code, the first `write()` displays the single character at offset 0x0101 in the font (the 257th character). The second `write()` displays the single character at offset 0x00ff (the 255th character) in the font.

```
write(h, "\x1\01", 2);    257th
write(h, "\x00\xff", 2);  255th
```

If the application writes an odd number of bytes to the console, the last byte is ignored. If the specified index is out of range, a checkerboard character displays.

Gray Scale Character Display

This allows applications to set shades of gray on the display. The application would do this by setting the background and/or foreground color for subsequent data writes to the display. Some Trident terminals have the hardware to support this feature. Others, like the VX 675 only supports monochrome mode. See the following functions `set_display_color()` and `get_display_color()`.

get_touchscreen()

This returns the coordinates of the touch in pixels. The VX 680 terminal comes with a “good enough” default calibration. A system mode menu item is provided to allow users to calibrate the screen — the calibration function displays three targets where user is expected to touch the center of the target, then the new calibration data is saved.

Prototype `int get_touchscreen(int *x, int *y);`

Parameters

- x The horizontal or column coordinate.
- y The vertical or row coordinate.

Return Values Values range from (0,0) upper left corner of the display to the pixel resolution of the display. On VX 680, (240, 320) the lower right corner of the display.

If the touchscreen is not “touched”, returns 0, X and Y are invalid.

Example This sample code allows the user to draw on the screen or display a signature.

Keypad

As with the display, Verix V terminals have several keypad configurations. Although all terminals will have the 12 key pad, ENTER, CANCEL and Backspace. The arrangement of the 12 keys from telco to calculator styles may vary as well as whether * and # or + and - are used. Screen, function and alpha keys or a toggle key vary from terminal to terminal.

Figure shows the typical VX 520 keypad layout.

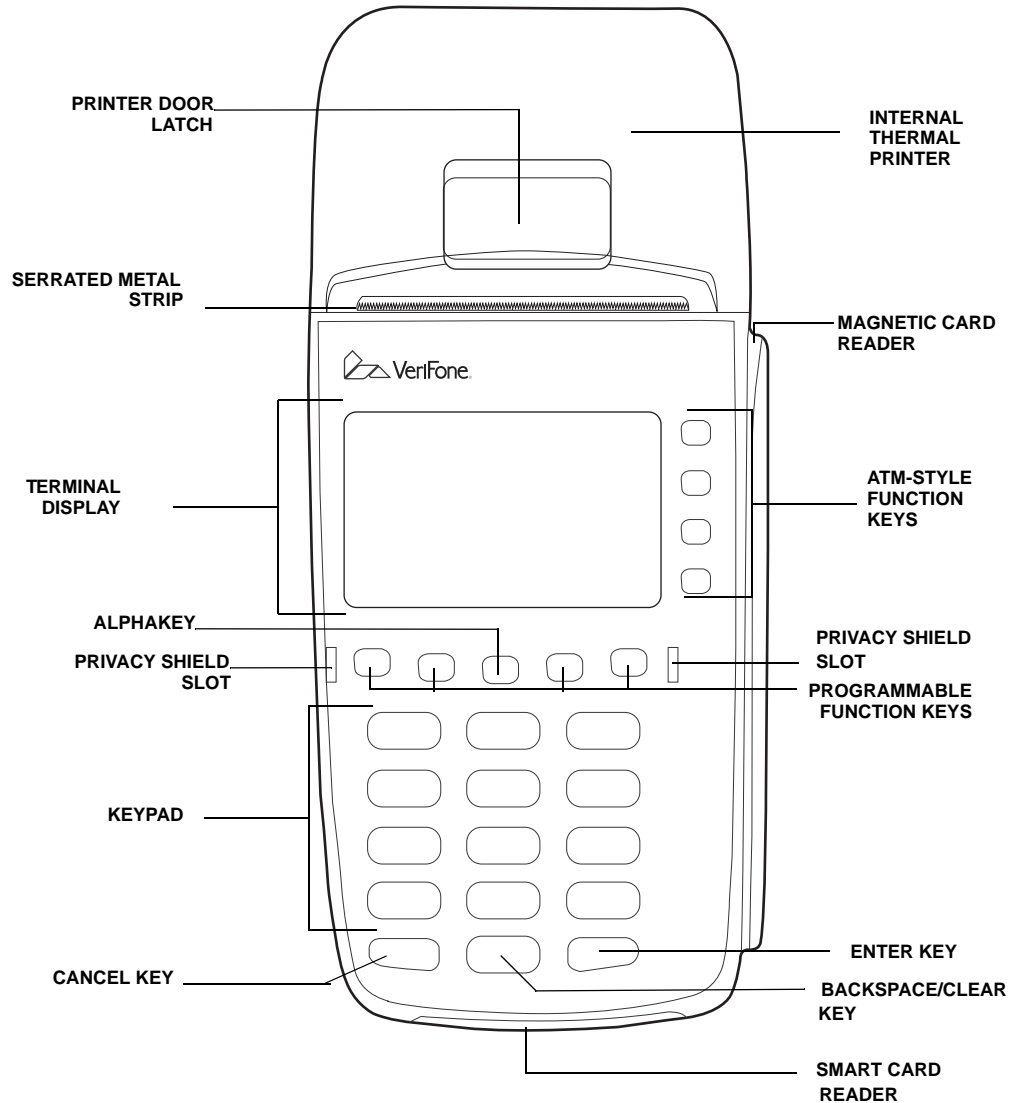


Figure 4 VX 520 Keys

NOTE



Some terminals have discrete function keys next to or below the display. Other terminals have touch screens with ATM keys and function keys implemented as part of the display touch panel. In VX 520 color, there are no ATM keys on the right side of the VX 520 color display as the scan codes map to the navigation keys. There is no alpha key, as the scan codes map to the select/confirm key. The default password is 166831.

A new keypad type and keymap are supported, and the existing APIs return these new keycodes (same as the VX 675):

Old Key Code	New Key Code
F1 ('z')	Nav. Up ('Z')
F2 ('{')	Nav. Down ('[')
F3 (' ')	Nav. Left ('\')
F4 ('}')'	Nav. Right (']')
Alpha (0x0F)	Select/Confirm ('^')

`SVC_INFO_KEYBRD_TYPE()` will return '3'.

The application controls the function of the screen-addressable keys. The following are VTM definitions of the color coded function keys and the ALPHA key:



The operating characteristics of this key is application dependent. When this key is pressed in VTM, the current menu option is escaped and the user returns to the previous menu level.



This is a dual function key. A quick keypress erases the previously entered single character. When this key is held down, the entire line of data is erased.



Use when data entry is complete.



This key is for entering alphabetic information using the limited keypad. Refer to [Alpha Key Support](#) for details on entering alphanumeric information.

NOTE



The keypad has an auto-repeat capability. If the key is pressed continuously each 750ms results in a new character placed in the keypad buffer. The BKSP key is handled in a special manner — If the BKSP key is pressed such that auto-repeat is engaged the result of the key press changes from 0x88 to 0x8E. This is traditionally interpreted as a single backspace then changes to erase field.

NOTE



A new feature that is now in VX 675, and will be added in future terminals is the navigation key. This 4-way navigation key returns up, down, left, right, and has a select or confirm center key.

Keypress Scan Codes

Table 17 lists the keypress scan codes.

Table 17 Keypress Scan Codes

Keypress	Scan Code	Description
1	0xB1	1 with the high-order bit set.
2	0xB2	2 with the high-order bit set.
3	0xB3	3 with the high-order bit set.
4	0xB4	4 with the high-order bit set.
5	0xB5	5 with the high-order bit set.
6	0xB6	6 with the high-order bit set.
7	0xB7	7 with the high-order bit set.
8	0xB8	8 with the high-order bit set.
9	0xB9	9 with the high-order bit set.
*	0xAA	* with the high-order bit set.
0	0xB0	0 with the high-order bit set.
#	0xA3	# with the high-order bit set.
CANCEL	0x9B	ESC with the high-order bit set.
BKSP	0x88	BS with the high-order bit set.
BKSP (long key press)	0x8E	SO with the high-order bit set.
ALPHA	0x8F	SI with the high-order bit set.
ENTER	0x8D	CR with the high-order bit set.
F0 ^a	0xEE	n with the high-order bit set.
F1 ^b	0xFA	z with the high-order bit set.
F2 ^b	0xFB	{ with the high-order bit set.
F3 ^b	0xFC	with the high-order bit set.
F4 ^b	0xFD	} with the high-order bit set.
F5 ^a	0xEF	0 with the high-order bit set.
a (leftmost horizontal screen key)	0xE1	a with the high-order bit set.
b (mid-left horizontal screen key)	0xE2	b with the high-order bit set.
c (mid-right horizontal screen key)	0xE3	c with the high-order bit set.
d (rightmost horizontal screen key)	0xE4	d with the high-order bit set.
up	0xDA	Z with the high-order bit set.
down	0xDB	[with the high-order bit set.
left	0xDC	\ with the high-order bit set.
right	0xDD] with the high-order bit set.
select	0xDE	^ with the high-order bit set.

a. Applicable for 16 x 21 LCD (V×670).

b. The V×525 does not have F1-F4 keys. It has telco function keys on the left side of the telephone hand set. These keys are EA, EB, EC, and ED.

Alpha Key Support

Normal system usage, as well as some VTM operations, requires entering alphanumeric information. The ALPHA key is provided to support alphanumeric entries. This section describes VTM ALPHA key functionality. The application must implement ALPHA key functionality. [alpha_shift\(\)](#) helps applications implement ALPHA key support.

NOTE



Some terminal do not have an ALPHA key. In this case cell phone mode is the default. Cell phone mode allows the selection of the alpha shifted characters by continually pressing the same key. If you wish to enter a second character with the same physical key you must wait 1 second before pressing the key again. The 1 second time interval is adjustable using the *CPAD variable.

The terminal keypad has 12 keys that can be used to enter as many as 52 different characters. These characters are the letters A through Z, the numbers 0 through 9, and the following special characters:

*	:	;	-	=
,	!	.	&	\$
'	+	#	space	"
@	%	/	\	

ALPHA Key Entry

Table 18 illustrates how to enter the string: 2 A E S ! space.

Table 18 **Alphanumeric Key Entry Example**








Desired Character	Keypress	Notes
2	Press 	
A	Press  Press  once	The ALPHA key performs a function similar to the shift key on a typewriter. The ALPHA key selects one of several different characters assigned to a single terminal key. Press the key with the desired character, then press the ALPHA key as many times as required until the correct character appears.
E	Press  Press  twice	
S	Press  Press  three times	

Table 18 **Alphanumeric Key Entry Example** (continued)





Desired Character	Keypress	Notes
!	Press 	
	Press  once	
space	Press 	
	Press  twice	

Table 19 shows different characters and how to access them from the keypad:

Table 19 **Alphanumeric Characters and Shift Entries**

























Key Press	No  Key Press	Press  One Time	Press  Two Times	Press  Three Times	Press  Four Times	Press  Five Times	Press  Six Times	Press  Seven Times	Press  Eight Times	Press  Nine Times	Press  Ten Times	Press  Eleven Times
	1	Q	Z	.	1	Q	Z	.	1	Q	Z	.
	2	A	B	C	2	A	B	C	2	A	B	C
	3	D	E	F	3	D	E	F	3	D	E	F
	4	G	H	I	4	G	H	I	4	G	H	I
	5	J	K	L	5	J	K	L	5	J	K	L
	6	M	N	O	6	M	N	O	6	M	N	O
	7	P	R	S	7	P	R	S	7	P	R	S
	8	T	U	V	8	T	U	V	8	T	U	V
	9	W	X	Y	9	W	X	Y	9	W	X	Y
	0	-	space	+	0	-	space	+	0	-	space	+
	*	,	'	\	"	*	,	'	\	"	*	,
	#	!	:	;	@	&	/	\	%	\$	_	#

Table 19 is for the default alpha shift. If the keypad is EBS100 type 4 or type 6 there is a different mapping:

EBS100 type 4:

```

1  * # 1
2  A B C 2
3  D E F 3
4  G H I 4
5  J K L 5
6  M N O 6
7  P Q R S 7
8  T U V 8
9  W X Y Z 9
0  - , ' " . + ! : ; @ = & / \ % $ _ 0

```

EBS100 type 6:

```

2  A B C 2
3  D E F 3
4  G H I 4
5  J K L 5
6  M N O 6
7  P Q R S 7
8  T U V 8
9  W X Y Z 9
-  * , ' " . -
+  # ! : ; @ = & / \ % $ _ +

```

CELL_PHONE Mode

Apart from the classic method where the user enters alpha characters the same way it is done on all current Verix eVo-based terminals. Terminals also supports the CELL_PHONE mode. If the terminal does not have an ALPHA key the default mode of the terminal is CELL_PHONE mode.

CELL_PHONE mode allows the user to select cell phone style (multiple key press mode) alpha character entry. When a terminal operates in cell phone mode, the alpha characters are obtained by pressing specific keys repeatedly until the desired character is displayed (such as, pressing “2” key four times displays the letter “C”).

The ALPHA key is always available to the user in `CELL_PHONE` mode, however the user must use either the ALPHA key or the `CELL_PHONE` method for any single key sequence. That is, if “C” is desired, press the “2” key, then the ALPHA key three times (for the classic mode), or press the “2” key four times (for `CELL_PHONE` method) with each subsequent key press within the inter-key timeout described in `(*CPAD)`. Similarly, if the “2” key is pressed twice within the inter-key timeout, and then followed by a single ALPHA key press, the letter “B” is displayed. If the “2” key is pressed again, a new key sequence is started and “2” is returned, even if the “2” key was pressed within the inter-key timeout after the ALPHA key was pressed.

ALPHA Mode Application Interface

Two `CONFIG.SYS` variables are associated with the Alpha Mode.

The first `CONFIG.SYS` variable sets the alpha character entry mode, `*AKM`. If `*AKM=CP`, the OS operates in cell phone alpha character mode. If `*AKM` has any other value or is not present, the OS operates in classic mode by default.

The `setAlphMode(CELL_PHONE)` sets the environment variable `*AKM=CP`. The parameter `CLASSIC` or an invalid parameter removes the environment variable `*AKM`, resulting in default behavior. The programmatic interface takes precedence over the `CONFIG.SYS` setting. The `CELL_PHONE` mode (multiple key press mode) is dependent on the time between key strokes. The application reads a single character at a time from the console. It then calls `alpha_multi_shift()` and displays the character returned. The application is responsible for character placement on the screen, as with all current Verix eVo platforms, and for managing the alpha mode in a multi-application environment, since each application may have unique requirements regarding its console input. Each application initializes to the value set in the group's `CONFIG.SYS`. If `*AKM` does not exist, it defaults to `CLASSIC` mode.

The second `CONFIG.SYS` variable is `*CPAD`. When a terminal operates in cell phone alpha mode, the OS translates the repeated key presses that take place within 1 second of each other (default) to the next alpha character in the alpha shift sequence for that key. If a key is pressed more than 1 second after the last, the OS returns the key as input.

NOTE



Pressing the “2” key twice in a row with a 1 second (or more) interval between key presses displays the string 22.

The time delay for cell phone alpha mode is adjusted using `*CPAD=milliseconds`. If `*CPAD` is present, the OS uses its value for the alpha shift delay. If `*CPAD` is not present or does not contain a valid numeric value between 0 and 10000, the OS uses the value 1000 or 1 seconds for the alpha shift delay. This range includes 0 and 10000 as valid values.

NOTE



If `*CPAD=1750`, the OS treats the repeated key presses that are more than 1.75 seconds apart as two different key presses. Repeated key presses that are less than 1.75 seconds apart are treated as alpha shift key presses.

Numeric truncation at the alpha character happens when alpha characters are within the environment variable value (standard C library behavior for alpha to `int` conversion). For example, when 900A00 is entered as a number value for the key and the key is converted, the A00 is truncated and ignored, thus, resulting to an internal value 900.

NOTE



`CELL_PHONE` mode is supported for both standard and EBS100 keyboard configurations.

The OS always turns off the cell phone mode for pin entry functions and VTM password entry, regardless of the mode the application has requested.

setAlphaMode()

Switches the keyboard operation to CLASSIC mode (default) or CELL_PHONE mode.

Prototype

```
int setAlphaMode(classic);  
int setAlphaMode(cell_phone);
```

Parameter

classic	Classic keyboard mode.
cell_phone	Cell phone mode.

Return Values

Success: Classic mode or cell phone mode is switched on.

getAlphaMode()

Returns the current mode of the keyboard.

Prototype

```
int getAlphaMode(void);
```

Return Values

Success: The current keyboard mode.

alpha_multi_shift()

Switches the keyboard operation to CLASSIC mode (default) or CELL_PHONE mode.

Prototype `int alpha_multi_shift(int key, int *shift);`

Parameter

key	The value read from the console. The high bit should be masked before passing the value to alpha_multi_shift().
shift	The pointer to a local int variable. This variable must be zero on the first call. The function manages the value as needed for subsequent calls.

NOTE



The shift variable should be reset to zero after reverting to CLASSIC mode and before returning to CELL_PHONE mode.

Return Values

Success: The shifted character.
The value as input if the time between key strokes exceeds *CPAD, or the next character in the alpha sequence for the input key.

Dual Keypress

The console driver detects when certain pairs of keys are simultaneously pressed and returns a combined scan code. The following are the dual keypresses recognized by the console driver:

- The ENTER, BKSP, CANCEL and if present the ALPHA and the right most screen key can be combined with one of the numeric keys (0-9) to create a "dual" keypress. The result is a single character placed in the keypad buffer.

Dual keypresses are debounced for the same amount of time as single keys (two scans in a row). Dual keypresses do not auto repeat. The scan codes returned for dual keypresses are shown in Table 20.

Table 20 Dual Keypress Scan Codes

Key Pair	Scan Code
d + 0..9	0xd0..0xd9
CANCEL + 0..9	0xc0..0xc9
BKSP + 0..9	0xa0..0xa9
ALPHA + 0..9	0xf0..0xf9
ENTER + 0..9	0xe0..0xe9

NOTE



Some dual keypresses return codes overlap with normal single keypress return codes. Specifically, dual keypresses Enter+1 through Enter+4 overlap with single keypresses a through d; Clear+3 overlaps with #.

The special key pairs F2+F4 and Enter+7 are used to enter VTM. These are the only keypresses that do not follow the restrictions defined above.

Hidden Function Support

There are VTM functions called hidden functions. These hidden functions are activated by specific key strokes that are not shown on any menu. They are available only while a particular menu is viewed. To invoke a back-to-back download, press 4 or 5 instead of '*' or '#' in Menus 1, 2, or 3.

Enter VTM

To enter VTM, simultaneously press the F2 and F4 keys. The dual keypress Enter+7 also enters VTM. The console driver detects the keys for VTM entry when they are pressed simultaneously and debounced for the normal duration.

NOTE



The dual key press to enter VTM can be changed from ENTER+7 to a 3 key press of either ENTER+7+5 or ENTER+8+4 by setting the CONFIG.SYS variable *SMDEF. If *SMDEF=1 ENTER+8+4 is the new password entry key press sequence and ENTER+7 is disabled. If *SMDEF=2 then ENTER+7+5 is the new system mode password entry key press sequence.

When this key sequence is detected and debounced, the console driver deactivates the current task (if it is not already the VTM task) and activates the VTM task.

The console driver does not save and restore the display on entry and exit of VTM. If a user tries to enter VTM and fails, the current application will still run but the user will be prompted for password entry. It is the application's responsibility to restore the display (if desired). VTM will generate an event, `EVT_ACTIVATE`, to notify the application that something happened.

VTM on VX 520 color is modified such that the key layout matches the 4-way navigation keys found below the display. The menu items "fill" the screen and take advantage of the longer display. VTM menus are displayed in 3-line mode until VTM is modified to use the full screen height.

Auto-Repeating Keys

If the user holds down a key, after a short debounce, the console posts an `EVT_KBD` event and passes the key's return code to the key buffer. If the user continues to hold the key down for another 750 ms, auto-repeating begins, at which point another event and key code are returned to the application. After this initial repeat, if the same key is still being held down, the event and key code are returned every 750 ms while the key is being held.

Backward Compatibility Mode

Backward compatibility mode is available on VX 680, which emulates Vx670 and also on VX 820, which emulates Vx810. Set `*FKEY=1` in group 1 `CONFIG.SYS` file to activate compatibility mode on both platforms.

You will see a framework on the top, right, and bottom of the screen. The top is a title frame, the right-hand vertical panel displays the screen keys (6 keys for VX 680, 4 keys for VX 820). The panel at the bottom of the screen shows four screen keys with the Verix traditional ALPHA key in the middle. The vertical and horizontal keys emulate the same key arrangement of the screen keys on Vx670 and Vx810. The remainder of the screen is the application display area and is 192 x 192 pixels. The default font is 9 x 12.

NOTE



BCM is not supported on VX 520 color terminals.

Replacing the Frame Panels

Each of the three frame panels—title panel, right-hand vertical panel and bottom/horizontal panel can be replaced by the application programmer. They are bmp files that are read only on console startup and will remain on the screen `*FKEY=1` is in the group 1 `CONFIG.SYS` file. The key arrangement cannot be changed.

For title panel:

- 1 Define `*FK_TITLE` to the name of the bmp file you've loaded.
- 2 The file should be 240 x 62 pixels.

For the right-hand/vertical panel:

- 1 Define `*FK_FKEY` to the name of the bmp file you've loaded.

2 The file should be 48 x 199 pixels.

For the bottom / horizontal panel:

1 Define *FK_HKEY to the name of the bmp file you've loaded.

2 The file should be 240 x 66 pixels.

Note that these frame panels can be replaced at will if [set_fkey_panel\(\)](#) is used.

VX 680 Compatibility Mode

The vertical panel should contain six button images within the following pixel dimensions {x1, y1, x2, y2}:

- Key 0 {200, 56, 233, 74 }
- Key 1 {200, 90, 233, 108}
- Key 2 {200, 124, 233, 142}
- Key 3 {200, 158, 233, 176}
- Key 4 {200, 192, 233, 215}
- Key 5 {200, 226, 233, 244}

VX 820 Compatibility Mode

The vertical panel should contain four vertical button images within the following pixel dimensions {x1, y1, x2, y2}:

- Key 1 {200, 56, 233, 74}
- Key 2 {200, 113, 233, 131}
- Key 3 {200, 170, 233, 198}
- Key 4 {200, 226, 233, 244}

NOTE



The horizontal panel button images (left to right)/ for both platforms should be within the following pixel dimensions:

{5, 263, 30, 301}

{44, 263, 69, 301}

{122, 263, 147, 301}

{161, 263, 186, 301}

ALPHA key {79, 273, 112, 291}

Console Ownership

Verix eVo mediates sharing of the console among the application tasks. A task that successfully opens the console becomes its owner, preventing other tasks from using it. The owner task can relinquish the console either permanently or temporarily to allow other tasks to use it.

There are three ways to transfer ownership of the console from one task to another:

- Press the hot key
- Press the VTM keys (F2+F4)
- Call `activate_task()` (console owner only)

In all cases prior to transfer of ownership to the new task, the console driver closes any open font file and resets to the default font. The keypad buffer is not cleared. In each case, it is the responsibility of the new task to reestablish its own context (with regard to each of these items) on activation. An application receives the `EVT_ACTIVATE` event when activated.



CAUTION

The console driver does *not* clear any console events (`EVT_KBD`) as a result of ownership transfer. It is the application's responsibility to manage its events properly prior to ownership transfer.

Manufacturer Information Block (MIB)

The console driver maintains its MIB in a non-volatile area of RAM (as do all drivers). The contents of the MIB are listed in [Table 21](#). This information can be retrieved by an application (usually the terminal management agent) using the `get_component_vars()` function.

Table 21 Manufacturer Information Block (MIB) Format

Description	Offset	Length
Number of keys pressed	0–3	4 bytes
State information:	4–8	5 bytes
contrast setting	4	
key beep setting	5	
hot key status	6	
backlight setting	7	
hot key code	8	

NOTE



Functions exist to retrieve the information in the MIB. Using the `get_componet_vars()` function is considered a primitive method and not recommended.

On VX 680 3G terminals, the OS sends the MIB `MODEL ID` field to VeriCentre to identify the terminal type for download. The model ID is also displayed on the copyright message screen during startup. The `MODEL ID` field must be set to VX6803G, this field is not determined by the OS or included in the OS. The `MODEL ID` must be added to the MIB at the factory before it is downloaded into the terminal.

Console Function Calls

This section presents descriptions of the function calls used to control the console.

You may also refer to the APIs listed below:

- `open()`
- `read()`
- `write()`
- `close()`

activate_task()

Allows the current console owner to pass control of the console to the specified task. The caller *must* be the current owner of the console to make the call. Three actions occur as a result of this call:

- An event code `EVT_DEACTIVATE` is posted to the current owner.
- Ownership of the console is passed to the new task, as if a `set_owner()` call was made. The new owner does not need to open the console, but immediately can begin writing to it using the `STDIN` or `STDOUT` handle.
- The font is set to the default font. Note that this may affect window size and cursor placement.
- An event code `EVT_ACTIVATE` is posted to the new owner of the console.

See [Console Ownership](#) for information on console ownership transfer.

WARNING



The keypad buffer is not cleared. The task receiving ownership should clear the buffer if new input is expected. The display retains whatever is on the display before passing ownership.

Prototype

```
int activate_task(int task_id);
```

Parameters

`task_id` The task ID for the new owner. If the task ID is invalid, console behavior is undefined. If the current owner is the same as the new owner, the ownership transfer proceeds (that is, the owner task receives both events, the font is reset, and so on).

Return Values

Success: 0

Failure: -1 with `errno` set to `EBADF`: The task does not own the console.

alpha_shift()

Returns the character that *follows* *c* in the ALPHA key shift sequence as performed on the terminal keypad and used in the VTM editor. Ownership of the console is not required to use this function.

Typically, the user presses a key followed by a number of ALPHA keypresses (see [Table 18](#)). If ALPHA key entry is required, the application is responsible for implementing the functionality.

Prototype `int alpha_shift(int c);`

Return Values Shift sequences correspond to the letters on the keypad, plus 16 special characters, as described in [Table 19](#).

If a given character is not part of any shift sequence, it is returned unchanged.

Example

<code>alpha_shift(2)</code>	Returns "A", which is the character that follows "2" in the ALPHA key shift sequence in Table 19 .
<code>alpha_shift(A)</code>	Returns "B", which is the character that follows "A" in the ALPHA key shift sequence in Table 19 .
<code>alpha_shift(B)</code>	Returns "C", which is the character that follows "B" in the ALPHA key shift sequence in Table 19 .
<code>alpha_shift(C)</code>	Returns "2", which is the character that follows "C" when the ALPHA key shift sequence wraps around to the beginning for that particular key in Table 19 .

clreol()

Clears the display line from the current cursor position to the end of the line based on the current window.

Prototype `int clreol(void);`

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own console device.

clrscr()

Clears the current window. If no window is set, the window defaults to the complete screen. The cursor is positioned at the start of the window.

Prototype `int clrscr(void);`

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own console device.

NOTE



A window may not be the entire screen. Other functions define a window to be anything from the entire screen to some portion. If no window is defined, `clrscr()` will clear the display.

contrast_down()

Decrements the current contrast setting, allowing it to “wrap” within the range of valid values. If the value goes below the minimum, then it is set to the maximum. Equivalent to `setcontrast(65)`.

NOTE

This API can only be used in monochrome terminals.

Prototype `int contrast_down(void);`

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: Calling task does not own the console.

contrast_up()

Increments the current contrast setting, allowing it to “wrap” within the range of valid values. If the value goes above the maximum, then it is set to the minimum. Equivalent to `setcontrast(64)`.

NOTE

This API can only be used in monochrome terminals.

Prototype `int contrast_up(void);`

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: Calling task does not own the console.

copy_pixel_block()

Copies the source block specified to the destination block. If a block violates a window boundary, the appropriate parameter will be adjusted. If the blocks are of different sizes, the block will be reduced to the smaller of the two dimensions. The source and destination block may overlap.

For example, executing the command:

```
copy_pixel_block(0,0,22,13,64,64,86,77);
```

will result in the system copying the upper left shaded area to the center of the display.

Prototype

```
int copy_pixel_block(
    int srcStartX, int srcStartY, int srcEndX, int srcEndY,
    int dstStartX, int dstStartY, int dstEndX, int dstEndY);
```

Parameters

srcStartX	The source block leftmost pixel column.
srcStartY	The source block uppermost pixel row.
srcEndX	The source block rightmost pixel column.
srcEndY	The source block bottommost pixel row.
dstStartX	The destination block leftmost pixel column.
dstStartY	The destination block uppermost pixel row.
dstEndX	The destination block rightmost pixel column.
dstEndY	The destination block bottommost pixel row.

Return Values

Success:	0
Failure:	-1 and errno set to <code>EBADF</code> if the application does not own the console or the console is not open. -1 and errno set to <code>EINVAL</code> for other errors.

cs_hard_reset

A non-blocking function call that commands the OS to start the hardware reset input of the keypad processor. The OS then re-initializes the keypad and driver then returns to normal operation. This function can be used to restore the keypad operation if it crashes. The `cs_spi_cmd_status()` function returns 0 while the reset is in progress, then returns ACK (0x06) when the reset is complete. The driver waits for about 5 seconds to let the keypad recover, then it repeats the hash verification. The keypad will be non-responsive until both the reset and the hash verification are complete.

Prototype `int cs_hard_reset(void);`

Return Values

Success: 0

Failure: -1 with errno set to ENODEV if call is made to the wrong keypad type.

cs_overlay_scan()

This function allows the application to start an overlay scan. The CPESM specification describes the overlay scan type values. The return value indicates whether the scan mode was entered successfully or not. If an overlay is detected, the keypad is considered not secure and PIN entry will be disabled.

Prototype `int cs_overlay_scan(int scantype);`

Parameters

scantype Scan type defined in the CPESM specification.

Return Values

Success: 0

Failure: Non-zero

cs_read_temperature()

Returns the temperature used in keypad overlay detection. It is used for factory calibration of the keypad capacitance.

Prototype `int cs_read_temperature(void);`

Return Values The temperature in degrees Celsius rounded to the nearest integer.

cs_set_baseline()

A non-blocking function call used in factory calibration of the CPESM. This commands the OS to send a software command to the keypad processor that requests the keypad to take a baseline capacitance measurement for the given type of scan. The processor stores the measurement in non-volatile storage.

The [cs_spi_cmd_status\(\)](#) function returns 0 while the measurement is in progress, then returns the ACK byte sent by the keypad when the command is complete. After the command is complete, the keypad returns to normal operation.

Prototype `int cs_set_baseline(int scantype);`

Parameters

<code>scantype</code>	Scan type byte defined in the CPESM specification.
-----------------------	--

Return Values

Success:	0
Failure:	-1: ENODEV if call is made to the wrong keyboard type.

cs_set_sleep_state()

Allows the application to put the keypad into a low power sleep mode and wake it back up. The keypad will not function when in `sleep` mode. When waking up the keypad, this function delays for 800mSec to allow the keypad processor time to wake up.

Prototype `int cs_set_sleep_state(int sleep);`

Parameters

<code>sleep</code>	1 to put the keypad to sleep. 0 to wake up the keypad.
--------------------	---

Return Values

Success:	0
Failure:	Non-zero

cs_soft_reset()

A non-blocking function call that commands the OS to send a software command to the keypad processor that requests a reset. The OS then re-initializes the keypad and driver then returns to normal operation. This reset type can be used after download to initiate the execution of the new code. The `cs_spi_cmd_status()` function returns 0 while the reset is in progress, then returns ACK (0x06) when the reset is complete. The driver waits for about 5 seconds to let the keypad recover, then it repeats the hash verification. The keypad will be non-responsive until both the reset and the hash verification are complete.

Prototype `int cs_soft_reset(void);`

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>ENODEV</code> if call is made to the wrong keypad type.

cs_spi_read()

This non-blocking function call is used in testing the CPESM. This function call commands the OS to send a read command to the keypad processor. The `cs_spi_cmd_status()` function returns 0 while the read is in progress, then returns either ACK or NAK when the command is complete. An ACK indicates that the LRC is good. A NAK indicates a bad LRC. After an ACK or NAK is received, the read data is available to be read with the `cs_spi_cmd_status()` function. After the read command is complete, the keypad returns to normal operation.

Prototype

```
int cs_spi_read(int encrypted, int command);
```

Parameters

encrypted	<ul style="list-style-type: none">• True to decrypt• False to clear text
command	Read command byte defined in the CPESM specification.

Return Values

Success:	0
Failure:	-1: ENODEV if call is made to the wrong keyboard type.

cs_spi_cmd_data()

This function allows an application to poll the status of the most recent command sent by this API. While the command is in progress, this function returns 0. When the command is complete, this function returns ACK (0x06) or NAK (0x15). If the CPESM returns a different ACK byte, this function returns that byte. This function also returns the packet data from the most recent read command. This function does not interfere with the normal operation of the keypad.

Prototype

```
int cs_spi_cmd_data(char *data);
```

Parameters

data	Pointer to an array that will hold the returned data.
------	---

Return Values

Zero while busy, ACK or NAK when complete.

cs_spi_cmd_status()

This function allows an application to poll the status of the most recent command sent by this API. While the command is in progress, this function returns 0. When the command is complete, this function returns ACK (0x06) or NAK (0x15). If the CPESM returns a different ACK byte, this function returns that byte.

Prototype `int cs_spi_cmd_status(void);`

Return Values Zero while busy, ACK or NAK when complete.

cs_spi_write()

A non-blocking function call that is used in testing the CPESM. This function call commands the OS to send a write command to the keypad processor. The [cs_spi_cmd_status\(\)](#) function returns 0 while the write is in progress, then returns the ACK byte sent by the keypad when the command is complete. After the command is complete, the keypad returns to normal operation.

Prototype `int cs_spi_write(int encrypted, int command, unsigned char *data);`

Parameters

<code>encrypted</code>	<ul style="list-style-type: none"> • True to encrypt • False to clear text
<code>command</code>	The write command byte defined in the CPESM specification.
<code>data</code>	The data to be written in the keypad processor.

Return Values

Success:	0
Failure:	-1: ENODEV if call is made to the wrong keypad type.

enable_kbd()

Unlocks the keyboard.

Prototype `int enable_kbd(void);`

Return Values

Success: 0

Failure: -1

delline()

Deletes the window display line containing the cursor and moves all following lines in the window up one line. The cursor position does not change.

Prototype `int delline(void);`

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own console device.

disable_hot_key()

Disables the hot key feature. If disabled, a hot key press is treated as a normal key press and queued for the console owner to read. See the hot key definition in [get_hot_key_sts\(\)](#) for more information.

Prototype `int disable_hot_key(void);`

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own console device.

disable_kbd()

Locks the keyboard.

Prototype `int disable_kbd(void);`

Return Values

Success: 0

Failure: -1

disable_key_beeps()

Disables audible key beep on key press.

Prototype `void disable_key_beeps(void);`

display_frame_buffer()

Takes the pointer to the application frame “buffer” of width “w” and height “h” and display the top-leftmost pixel at display location (x,y). The frame buffer must contain the appropriate number of bits per pixel. Currently the only supported pixel format is 16-bit 5:6:5.

NOTE



VeriFone recommends using display_frame_buffer() for animation or slideshow display.

Prototype

```
int display_frame_buffer(int x, int y, int w, int h, short * buf);
```

Parameters

x	Column coordinate
y	Row coordinate
w	Width
h	Height
buf	Pointer to a short array

Return Values

Success: 0

Failure: -1 and errno set to EBADF if the application does not own the console or the console is not open.

-1 and errno set to EINVAL for other errors.

Example

```
/* create 16 bit 5/6/5 format pixel from RGB triplet */
#define RGB2PIXEL565(r,g,b) \
    (((r) & 0xf8) << 8) | (((g) & 0xfc) << 3) | (((b) & 0xf8) >> 3))

Int main(void) {
    short * frameBuffer;
    frameBuffer=(short*)malloc(8);

    con = open (DEV_CONSOLE, 0);
    frameBuffer[0]= RGB2PIXEL565 (0xFF, 0, 0 ); // red
    frameBuffer[1]= RGB2PIXEL565 (0, 0xFF, 0 ); // green
    frameBuffer[2]= RGB2PIXEL565 (0, 0, 0xFF); // blue
    frameBuffer[3]= RGB2PIXEL565 (0xFF, 0, 0xFF); // purple
```

```
display_frame_buffer(0,0,4,1,frameBuffer); // will display the
pixel row
```

Red	Green	Blue	Purple	

```
display_frame_buffer(1,2,2,2,frameBuffer); // will display the
pixel block
```

Red	Green	Blue	Purple	
	Red	Green		
	Blue	Purple		

```
display_frame_buffer(5,1,1,4,frameBuffer); // will display the
pixel column
```

Red	Green	Blue	Purple			
					Red	
	Red	Green			Green	
	Blue	Purple			Blue	
					Purple	

draw_line()

Draws a line of specified width and color from the start point to the end point.

Prototype `int draw_line(int startX, int startY, int endX, int endY, int width, int color);`

Parameters The parameters describe the pixel line where width is the line width and color is the line color.

startX	The column of the starting pixel.
startY	The row of the starting pixel.
endX	The column of the ending pixel.
endY	The row of the ending pixel.

Return Values

Success: 0

Failure: -1 and errno set to `EBADF` if the application does not own the console or the console is not open.

-1 and errno set to `EINVAL` for other errors.

enable_hot_key()

Enables the hotkey feature. If a hot key has been defined, it is enabled and a hot key press transfers console ownership to the hot key owner.

Prototype `void enable_hot_key(void);`

enable_key_beeps()

Enables audible key beeps on a keypress.

Prototype `void enable_key_beeps(void);`

Return Values

Success: 0

Failure: -1 and errno set to `EBADF`: Caller does not own console device.

get_backlight_level()

Returns the backlight level set by [set_backlight_level\(\)](#).

Prototype `int get_backlight_level(void);`

Return Values The return value ranges from zero to 100 (0 - 100).

get_battery_icon()

Returns the pointer to the battery icon.

Prototype `int get_battery_icon(char* buff3);`

Parameters

buff3	Pointer to a buffer size of 3.
-------	--------------------------------

Return Values

Success:	1
Failure:	0

getcontrast()

Returns the current contrast setting for the display. See also, [setcontrast\(\)](#).

Prototype `int getcontrast(void);`

Return Values

Success: 1–15 Valid contrast setting

Failure: -1 and errno set to `EBADF`: Callings task does not own the console.

getfont()

Returns the filename of the current font. If the font in use is the default font, the string “DEFAULT” is stored in `font_name`. See also, [get_font\(\)](#).

Prototype `void getfont(char *font_name);`

Parameters

`font_name` Pointer to the null-terminated font name.

getgrid()

Returns the current grid setting. The font file character size determines the grid.

Prototype `int getgrid(void);`

Return Values

Success Code:	Font size
2	6 × 8
0	8 × 16 and 16 × 16
Failure:	-1 and errno set to <code>EBADF</code> : Caller does not own console device.

getinverse()

Returns the current inverse video setting.

Prototype `int getinverse(void);`

Return Values

0-1	Inverse video setting (0 = Off, 1 = On).
-1	If calling task does not own the console, <code>errno</code> is set to <code>EBADF</code> .

get_kb_backlight()

Retrieves the current keyboard backlight value.

Prototype `int get_kb_backlight(void);`

Return Values

Success: 0 - 100

Failure: -1

getscrollmode()

Returns the current scroll mode setting. The scroll mode setting determines how the display behaves when the cursor is at the end of the window. See also [setscrollmode\(\)](#).

Prototype `int getscrollmode(void);`

Return Values

Returns a code indicating the current display scrolling behavior, as set by [setscrollmode\(\)](#).

- | | |
|----|---|
| 0 | No scrolling (default). |
| 1 | Horizontal scrolling. Window will scroll right to left and up so writing a character past the end of the window will move each line over and up and the first character in the window will scroll off the window. |
| 2 | Vertical scrolling. Window will scroll up. Writing past the end of the window causes the first line in the window to be deleted. |
| -1 | with errno set to <code>EBADF</code> : Caller does not own console device. |

get_BMP()

This creates a BMP-formatted string from a portion of the display specified by the coordinates and color depth, and places the string in the specified buffer. The application must then take the buffer and write it to a file.

Prototype

```
int get_BMP
(int startX, int startY, int endX, int endY, char *buf, int color);
```

Parameters

startX	Leftmost pixel column of the block.
startY	Uppermost pixel row of the block.
endX	Rightmost pixel column of the block.
endY	Bottommost pixel row of the block.
buf	Address of the buffer to store data.
color	Number of bits per pixel. Only 1 and 24 bits are supported.

Return Values

Success:	Positive, number of bytes of data.
Failure:	Negative, error occurred and errno is set appropriately.

The buffer must be large enough to contain the data. The formula for calculating the buffer size is dependent upon the block size and color depth. For a 1 bit BMP, the formula is:

$$\text{Size} = 40 + (\text{endY} - \text{startY} + 1) * ((\text{endX} - \text{startX} + 1 + 31) / 32) * 4$$

NOTE



Use integer math. Ignore the fractional portion of the division.

For a 24 bit BMP, the formula is:

$$\text{Size} = 32 + (\text{endY} - \text{startY} + 1) * (((\text{endX} - \text{startX} + 1) * 3) + 3) / 4 * 4$$

NOTE



Use integer math. Ignore the fractional portion of the division. Do NOT simplify the equation.

get_character_size()

Stores the current character size as the number pixel rows and columns. The selected font determines the current character size.

Prototype `int get_character_size(int* pixelRows, int* pixelColumns);`

Parameters

`pixelRows` Pointer to store rows information.
`pixelColumns` Pointer to store column information.

Return Values

Success: 0
 Failure: -1 and errno set to EBADF: If the application does not own the console or the console is not open.

get_console()

Returns the handle for the console if the current task owns the console and optionally clears the keyboard FIFO.

Prototype `int get_console(int clear_keys);`

Parameters

`clear_keys` If > 0, the keyboard FIFO is cleared.

Return Values

Success: ≥ 0 : Console handle
 Failure: -1 and errno set to EBADF: Caller does not own console device.

get_display_color()

Returns the current value for the color type specified.

Prototype `int get_display_color(int type);`

Parameters

type FOREGROUND_COLOR. Return values are:
 0 = WHITE (monochrome and 4 level gray only)
 1 = BLACK (monochrome and 4 level gray only)
 2 = LIGHTGRAY (4 level gray only)
 3 = DARKGRAY (4 level gray only) 16-bit RGB value (RGB mode only)

BACKGROUND_COLOR. Return values are:
 0 = WHITE (monochrome and 4 level gray only)
 1 = BLACK (monochrome and 4 level gray only)
 2 = LIGHTGRAY (4 level gray only)
 3 = DARKGRAY (4 level gray only) 16-bit RGB value (RGB mode only)

CURRENT_PALETTE (the currently selected palette). Return values are:
 1 = RGB
 2 = MONOCHROME (black and white)
 4 = GRAY4 (4 level gray scale)

AVAILABLE_PALETTE (the number of palettes supported by the terminal).
 Return values are:
 1 = Monochrome only (510, 570, 610)
 2 = Monochrome and 4 level gray (670)
 3 = RGB (580, 680)

Return Values

Success: Current value for the specified color type.

Failure: -1 and errno set to EBADF: If the application does not own the console or
 the console is not open.
 -1 and errno set to EINVAL for other errors.

get_display_coordinate_mode()

Returns the current positioning mode (CHARACTER_MODE or PIXEL_MODE).

Prototype `int get_display_coordinate_mode (void);`

Return Values

Success: The current positioning mode.

Failure: -1 and errno set to EBADF: If the application does not own the console or the console is not open.

get_font()

Returns the filename of the current font or the string DEFAULT if using the default font.

Prototype `int get_font(char *font_name);`

Parameters

`font_name` Pointer to a buffer to store the null-terminated font name.

Return Values

Success: > 0: Length of `font_name` string.

Failure: -1 and errno set to EBADF: Caller does not own console device.

-1 and errno set to EACCES: `font_name` is an invalid pointer.

get_font_mode()

Returns the number of index bytes (1 or 2) used for the current font. Font files with less than 256 characters have 1-byte indices. Otherwise, the font file requires 2 bytes to index a character.

Prototype `int get_font_mode(void);`

Return Values

Success: 1 or 2: Number of bytes to index current font file.

Failure: -1 and errno set to EBADF: Caller does not own console device.

get_iap_state()

This function returns the IAP state machine state. This is a Vx600 only function.

Prototype `int get_iap_state(int hdl);`

Parameters

hdl Device handle

Return Values

Success: 0

Failure: -1: EINVAL

get_hot_key_sts()

Returns the current status of the hot key. If the hot key is defined, the return value can be interpreted based on the following structure.

```
struct {
    short currentConsoleOwnerId;
    char hotKeyEnabled;
    char hotKeyCode;
    } hotKeyStatus;
```

NOTE



The returned hot key code is the “secured” key code value. In general, this is the normal key code value with the high bit off.

See also [disable_hot_key\(\)](#), [enable_hot_key\(\)](#), and [Table 17](#).

Prototype

```
long get_hot_key_sts(void);
```

Return Values

Success: ≥ 0 : A hot key has been defined and is currently enabled

Failure: -1 with errno set to `EBADF`: Caller does not own console device.

get_host_status()

Returns the connection and battery state of the host device. This is a Vx600 function call only.

Prototype

```
int get_host_status(void);
```

Return Values

Success - Bit mask where:

bit 15	0 - Ipod connected
	1 - Ipod not connected
bit 13	0 - Host not connected
	1 - Host connected
bit 11	0 - No external power
	1 - External power

get_ipod_status()

This is a Vx600 only function call that returns the state of the IPOD host. The bit mask is defined as follows:

549K_RESISTOR	0x01
IPAY_PWR_IPOD	0x02
ACC_DETECT	0x04
IPOD_PRESENT	0x08
ACC_POWERED	0x10

Prototype int get_ipod_status(int hdl);

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1 : EINVAL

gotoxy()

Moves the cursor to the position specified. The coordinates are character oriented, 1 based and window relative (for example gotoxy(1,1) always positions the cursor at the top right corner of the window). If the coordinates specified are outside the current window, they are forced into the window.

Prototype int gotoxy(int x, int y);

Parameters

x	x row coordinate.
y	y column coordinate.

Return Values

Success:	0
Failure:	-1 and errno set to EBADF: Caller does not own console device.

iap_control_function()

This function call allows the application to control the keypad state. This is a V*600 only function.

Prototype `int iap_control_function(int hdl, int func);`

Parameters

- | | |
|------|--|
| hdl | Device handle |
| func | <ul style="list-style-type: none"> • IAP_CONTROL_KEYPAD_SLEEP turns off the keypad. • IAP_CONTROL_KEYPAD_WAKE turns the keypad on (takes 800mSec). • IAP_CONTROL_DISABLE_KEYBEEP turns off the keypad beeps. • IAP_CONTROL_ENABLE_KEYBEEP turns on the keypad beeps. |

Return Values

Success:	0
Failure:	-1

iap_get_keypad_info()

This is a V*600 only function call that copies the keypad status to the given buffer. The first byte is the “Enable State”. The second byte is the “Beep State”.

Prototype `int iap_get_keypad_info(int hdl, char *buf);`

Parameters

- | | | | | | |
|--------------------|--|--------------------|--|------------------|--|
| hdl | Device handle | | | | |
| buf | <p>The two status bytes are as follows</p> <table border="0"> <tr> <td style="padding-right: 10px;">Enable State byte:</td> <td> <p>0 if keypad is disabled.</p> <p>1 if keypad is enabled and awake.</p> <p>2 if keypad is enabled, still waking up.</p> </td> </tr> <tr> <td style="padding-right: 10px;">Beep State byte:</td> <td> <p>0 if beeps are disabled.</p> <p>1 if beeps are enabled.</p> </td> </tr> </table> | Enable State byte: | <p>0 if keypad is disabled.</p> <p>1 if keypad is enabled and awake.</p> <p>2 if keypad is enabled, still waking up.</p> | Beep State byte: | <p>0 if beeps are disabled.</p> <p>1 if beeps are enabled.</p> |
| Enable State byte: | <p>0 if keypad is disabled.</p> <p>1 if keypad is enabled and awake.</p> <p>2 if keypad is enabled, still waking up.</p> | | | | |
| Beep State byte: | <p>0 if beeps are disabled.</p> <p>1 if beeps are enabled.</p> | | | | |

Return Values

Success:	0
Failure:	-1

insline()

Inserts a blank line following the line containing the cursor. All lines following it move down one line. See also [enable_kbd\(\)](#).

Prototype `int insline(void);`

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own console device.

inverse_toggle()

Toggles the current inverse video setting. Equivalent to `setinverse(3)`.

Prototype `int inverse_toggle(void);`

Return Values

Success: 0

Failure: -1 and errno is set to EBADF if calling task does not own the console.

invert_pixel_block()

Inverts the colors within the specified pixel block—black is inverted to white (and vice versa) and light gray is inverted to dark gray (and vice versa). A block can be as small as one pixel, up to the size of the display.

Prototype `int invert_pixel_block (int startX, int startY, int endX, int endY);`

Parameters The parameters describe the pixel block to invert.

<code>startX</code>	The leftmost column.
<code>startY</code>	The uppermost row.
<code>endX</code>	The rightmost column.
<code>endY</code>	The bottommost row.

Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF` if the application does not own the console or the console is not open.
 -1 and `errno` set to `EINVAL` for other errors.

is_keypad_secure()

This function call allows the application to check whether the OS considers the keypad firmware to be authentic. If the firmware is not authentic, then the PIN entry will be disabled.

Prototype `int is_keypad_secure(void);`

Return Values

- 0 Secure
- 1 Not secure
- 2 Security check is in progress.

kbd_pending_count()

Returns the number of keystrokes available for reading. The maximum number of keystrokes is 20.

Prototype `int kbd_pending_count(void);`

Return Values

- Success: Number of key presses queued for reading (0–20).
- Failure: -1 and errno set to EBADF: Caller does not own console device.

kbd_pending_test()

Tests if a specific key is currently queued in the keyboard FIFO.

Prototype `int kbd_pending_test(int targetchar);`

Parameters

targetchar The key to look for.

Return Values

Success: 0: targetchar not present in the keyboard buffer.
 1: targetchar present in the keyboard buffer.

Failure: -1 and errno set to EBADF: Caller does not own console device.

key_beeps()

Turns on beeps when keys are pressed if `flag = 1`; turns off beeps if `flag = 0`. When key beeps are on, the normal tone is emitted for 50 ms, starting from initial key-down debounce. If the key pressed is not enabled, the keypress is ignored and an error beep sounds.

When key beeps are off, there are no beeps when keys are pressed.

Prototype `int key_beeps(int flag);`

Parameters

flag Key beep setting (1 or 0).

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own the console device.

lock_kbd()

Locks/unlocks the keyboard.

Prototype `int lock_kbd(int flag);`

Parameters

flag	0: Unlocks the keyboard. 1: Locks the keyboard.
------	--

Return Values

Success: 0
Failure: -1

putpixelcol()

Displays graphic images on a byte-by-byte basis. `putpixelcol()` works on the current window, *not* on the entire screen. Data wraps at the end of the current line of the current window to fit the data in the specified window. As with `write()` functions, the cursor is positioned at the next character after the end of the displayed text.

`putpixelcol()` writes `buf` in columns of 8 pixels horizontally. If the current character size is 8 x 16 or 16 x 16, data wraps at the end of each character to fill each character of the current grid before going to the next character.

Prototype `int putpixelcol(char *buf, int len);`

Parameters

buf	Pointer to data to write.
len	Number of bytes in <code>buf</code> to write.

Return Values

Success: > 0: Number of characters written to display. This should be the same as `len`.
Failure: -1 with `errno` is set to `EACCES`: `buf` is an invalid pointer.
 -1 with `errno` is set to `EINVAL`: Negative length parameters.
 -1 and `errno` set to `EBADF`: Caller does not own the console device.

put_BMP()

Facilitates the display of small, icon-sized graphical constructs that might be used as key labels or pushbuttons for the touch panel.

It loads BMP graphic files into memory cache, until system available memory drops to approximately 1MB of free memory. Once files are memory-cached, they will remain cached without further cache management. Once the 1MB free space limit is exceeded, additional graphics displayed with `put_BMP()` will be read from NAND FLASH.

These changes to `put_BMP()` allow well-behaved applications to benefit from cached bmp files, while preserving functionality for applications in low memory situations.

WARNING

This function call is available for both Predator and Trident. VeriFone strongly discourages the use of this API to display extensive animation, video, or slideshow. Overuse or misuse of `put_BMP()` can result in excessive NAND access, potential for read disturb errors that can corrupt files and the subsequent need to process the unit through a repair depot.

To avoid excessive NAND access, use `display_frame_buffer()` as an alternative. NAND Flash is a Trident feature.

NOTE

The Trident operating systems are enhanced to minimize read disturb effects. They include supplemental debug messages to notify developers when the cache limit is reached. Use these message to identify when applications are over-using `put_BMP()` and need corrective action to reduce NAND FLASH reads.

The OS DEBUG message for QT680015 and QT520016

“`put_BMP(bmp_file_name): NAND`” indicates that the cache is full, therefore BMP is being read from NAND FLASH. For OS versions subsequent to QT680015 and QT520016, this debug message will change to “`SYS_MAP_FILE(bmp_file_name): return ENOMEM.`”

Disproportionate use of `put_BMP()` risks circumventing these measures because the OS may not have the available time required for these enhancements to detect and recover errors before the read disturb threshold is exceeded. This effect will increase over time.

Bit Mapped File Creation

We use MicroSoft (MS) Paint to create BMP files. While other tools can be used, the file format, size, and parameters from Paint must be followed. First, BMP files must not be compressed. Second, BMP files are either monochrome, grey scale, or color. 4 level grey scale is supported for displays that can show grey scale. Color displays can support 16 (4 bit), 256 (16 bit) or 16,777,216 (24 bit) colors. The maximum BMP display size is defined by the terminal display used.

An example is the VX 520 which supports a 128 x 64 bit display (320 x 240 bit display for color variant) or the VX 680 which supports a 240 x 320 bit display. BMP files can be smaller.

To create a BMP file in Windows Paint, go to Paint and select a New file. There are two methods to adjust the image size:

- 1 Use the toolbar.
 - Click on Image, then Attributes.
 - Set the Width and Height to 128 and 64 or 128 x 128.
- 2 Grab the corner and pull to the desired size. Note that here the pixel addressing is zero-based so a 128 x 128 file should be 127,127 in status window on the bottom right of the window.
 - Click File, then Save As.
 - Name the file, then select Monochrome Bitmap in the “save as type” window, then click Save the file. To create a four-level gray scale image using paint, do a Save As and select 16 color bitmap. Use only black, dark gray, light gray and white from the palette. Other colors may be selected, but may NOT display correctly on the display.

Prototype `int put_BMP(char *file);`

Parameters

<code>file</code>	The standard format BMP file.
-------------------	-------------------------------

Return Values

Success: 0

Failure: -1 and errno set to `EBADF`. If the application does not own the console or the console is not open.

-1 and errno set to `EINVAL` for other errors.

put_BMP_at()

Writes the bitmap image found in the file to the current window starting at the defined x-y coordinates.

Prototype `int put_BMP_at(int x, int y, char *file);`

Parameters

<code>x</code>	X coordinate of current window.
<code>y</code>	Y coordinate of current window.
<code>file</code>	Filename containing the bitmap image.

Return Values

Success:	0
Failure:	-1: EINVAL - Invalid parameter.

put_graphic()

Writes `buf` to the rectangular area specified by `x1`, `y1`, `x2`, and `y2`. If required, the write wraps within the area. If the write is too big for the area, it is truncated. The coordinates are character based and window relative. If the coordinates specified are outside the bounds of the window, they are forced in bounds. If either of the second coordinate pair is less than the first pair, it is set to the first coordinate.

The cursor is positioned at the next character after the end of the data.

Prototype `int put_graphic(const char *buf, int len, int x1, int y1, int x2, int y2);`

Parameters

<code>buf</code>	Pointer to data to write.
<code>length</code>	Number of bytes in <code>buf</code> to write.
<code>x1</code>	Top-left x (column) coordinate of put graphic write area.
<code>y1</code>	Top-left y (row) coordinate of put graphic write area.
<code>x2</code>	Bottom-right x (column) coordinate of put graphic write area.
<code>y2</code>	Bottom-right y (row) coordinate of put graphic write area.

Return Values

Success:	> 0: Value is the same as <code>length</code> regardless of number of characters written to the display (for example, if <code>buf</code> is truncated, <code>length</code> is still returned).
Failure:	-1 with <code>errno</code> set to EACCES: Invalid <code>buf</code> pointer. -1 with <code>errno</code> set to EINVAL: <code>length</code> is negative. -1 and <code>errno</code> set to EBADF: Caller does not own the console device.

resetdisplay()

Sets the font for a pixel type display. If the character size for the new font file is different from the previous font file, the window and cursor may be affected. The window is changed to the closest valid display coordinates for the new character size. Likewise, the cursor is moved to the closest valid display location for the new font.

NOTE



resetdisplay() is provided for backward compatibility with TXO terminals.

Prototype

```
int resetdisplay(const char *font, int grid_id);
```

Parameters

font	Font definition file used for the font. To select the default font, use a null string.
grid_id	Not used.

Return Values

Success:	0
Failure:	-1 with errno set to EBADF: Caller does not own console. -1 with errno set to EACCES: font is an invalid pointer. Other ernnos may be set by the file manager.

reset_ipod_pins()

This is a Vx600 only function. This function call sets the state to 0 for the pins defined:

549K_RESISTOR	0x01
IPAY_PWR_IPOD	0x02
ACC_DETECT	0x04

Prototype

```
int reset_ipod_pins(int hdl, int mask);
```

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

screen_size()

Retrieves the screen size. On VX 680, this accepts a two char array. A char is not large enough to describe VX 680's 240 x 320-pixel display. It is advised that instead of using this function, applications can use either [SVC_INFO_DISPLAY\(\)](#) or [SVC_INFO_DISPLAY_EXT\(\)](#).

Prototype `int screen_size(char *buf);`

Parameters

`buf` Pointer to store the screen size:

- `buf[0]` = number of rows in the current screen in the default font
- `buf[1]` = number of columns

Return Values

Success: 0

Failure: -1 with `errno` set to `EBADF`: Caller does not own the console device.
 -1 with `errno` set to `EACCES`: `buf` is an invalid pointer.

Example

The linked code example sets the screen size.

set_hot_key()

Defines a key to be used as a hot key. Note that the keycode value must have the high order bit set.

Prototype `int set_hot_key(int keycode);`

Parameters

`keycode` Valid key value with a high order bit set.

Return Values

Success: 0

Failure: -1: `EINVAL` if `keycode` is not a valid key.

set_ipod_pins()

Sets the state to 1 for the pins defined:

549K_RESISTOR	0x01
IPAY_PWR_IPOD	0x02
ACCT_DETECT	0x04

Prototype `int set_ipod_pins(int hdl, int mask);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

set_kb_backlight()

Sets the keyboard backlight level between 0 and 100.

Prototype `int set_kb_backlight(int value);`

Parameters

value	Value from 0 to 100.
-------	----------------------

Return Values

Success:	0
Failure:	-1

setcontrast()

Sets the display contrast based on `value`. The range of valid contrast settings is 1-15. If `value` is 0, then the contrast setting is set to the default value of 8. If `value` is 64, the contrast setting is incremented to the next value in the range (incrementing 15 causes the value to wrap around to 1). If `value` is 65, the contrast setting is decremented to the previous value in the range (decrementing 1 causes the value to wrap around to 15). All other values are ignored.

NOTE

This API can only be used in monochrome terminals.

Prototype `int setcontrast(int value);`

Parameters

- | | |
|--------------------|--|
| <code>value</code> | Contrast setting: <ul style="list-style-type: none">• 1–15: Set contrast to this value.• 0: Set contrast to 8.• 64: Increment contrast setting by one; 15 wraps to 1.• 65: Decrement contrast setting by one; 1 wraps to 15.• All other values are ignored. |
|--------------------|--|

Return Values

- | | |
|----------|---|
| Success: | 0 |
| Failure: | –1 and <code>errno</code> set to <code>EBADF</code> : Caller does not own the console device. |

Example The linked code example demonstrates making the display one step darker.

setfont()

Has the same functionality as [set_font\(\)](#).

setinverse()

Selects the inverse video setting based on the two LSBs of *value*.

Value	Meaning
0	Turn off inverse video.
1	Turn on inverse video.
2	No change.
3	Toggle inverse video setting.

Prototype `int setinverse(int value);`

Parameters

`value` Inverse video option.

Return Values

0 Success.
-1 If calling task does not own the console, `errno` is set to `EBADF`.

setscrollmode()

Sets the scroll mode. The scroll mode setting determines how the display behaves when the cursor is at the end of the window.

Prototype `int setscrollmode(int mode);`

Parameters

`mode`

- 0 = No scrolling (default)
- 1 = Horizontal scrolling: Window will scroll right-to-left and up, so that writing a character past the end of the window will move each line over and up, and the first character in the window will scroll off the window.
- 2 = Vertical scrolling: Window will scroll up. Writing past the end of the window will cause the first line in the window to be deleted.

Return Values

Success: 0
Failure: -1 with `errno` set to `EBDAF`: Caller does not own console.
-1 with `errno` set to `EINVAL`: mode setting is invalid.

Example

The linked code example demonstrates setting the scroll mode.

set_backlight()

Turns the display backlight on/off. The state of the backlight is preserved across power cycles so the unit will have the same backlight level unless specifically changed.

This interface allows the application to control the state of the backlight immediately. If the application has set the backlight on then the OS power-management logic will turn it off when going to sleep and turn it back on when waking up. If the application has turned the backlight off with the interface then OS-initiated sleep mode will not affect the backlight.

Prototype `int set_backlight(int mode);`

Parameters

- mode
- 1 = backlight on (default).
 - 0 = backlight off.

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own the console device.

set_backlight_level()

This API sets the display backlight brightness to a specific level — a percentage value between 0 (OFF) and 100 (ON).

NOTE



Backlight may not have 100 discrete levels — certain adjacent percentage values may result to the same brightness level.

Prototype `int set_backlight_level(int level);`

Parameters

- level
- 100 = Backlight on.
 - 0 = Backlight off.

Return Values

Success: 0

Failure: -1 and errno set to EINVAL: The specified brightness level is out of range.

set_cursor()

Turns the cursor on or off. When visible, the cursor is displayed as a blinking reverse video image of the underlying character.

Prototype `int set_cursor(int flag);`

Parameters

flag • 1 = Cursor on.
 • 0 = Cursor off.

Return Values

Success: 0

Failure: -1 and errno set to EBADF: Caller does not own the console device.
 If flag is not 0 or 1, errno is set to EINVAL.

set_display_coordinate_mode()

Sets the positioning mode as specified. The current window will be set back to the default of the entire screen and the cursor will be positioned to the upper left side of the display. Changing the coordinate mode does not result in any observable change in the display.

Prototype `int set_display_coordinate_mode (int setting);`

Parameters

setting • CHARACTER_MODE
 • PIXEL_MODE

Return Values

Success: 0

Failure: -1 and errno set to EBADF: If the application does not own the console or the console is not open.
 -1 and errno set to EINVAL for other errors.

set_display_color()

Sets the type to the specified color for all subsequent characters. The settings will remain in effect until changed or the console is `Open()`.

Prototype `int set_display_color (int type, int value);`

Parameters

- | | |
|-------|---|
| type | <ul style="list-style-type: none"> • FOREGROUND_COLOR • BACKGROUND_COLOR • CURRENT_PALETTE |
| value | <p>FOREGROUND_COLOR:</p> <p>0 = WHITE (monochrome and 4 level gray only)</p> <p>1 = BLACK (monochrome and 4 level gray only)</p> <p>2 = LIGHTGRAY (4 level gray only)</p> <p>3 = DARKGRAY (4 level gray only) 16-bit RGB value if in RGB mode</p>
<p>BACKGROUND_COLOR:</p> <p>0 = WHITE (monochrome and level 4 gray only)</p> <p>1 = BLACK (monochrome and level 4 gray only)</p> <p>2 = LIGHTGRAY (4 level gray only)</p> <p>3 = DARKGRAY (4 level gray only)16-bit RGB value if in RGB mode</p>
<p>CURRENT_PALETTE:</p> <p>1 = RGB</p> <p>2 = MONOCHROME (black and white)</p> <p>4 = GRAY4 (4 level gray scale)</p> |

Return Values

- | | |
|----------|---|
| Success: | 0 |
| Failure: | <p>-1 and <code>errno</code> set to <code>EBADF</code>: If the application does not own the console or the console is not open.</p> <p>-1 and <code>errno</code> set to <code>EINVAL</code> for other errors.</p> |

set_font()

Sets the font to the specified font file. The font file is generated by the Font Generation tool included in the VVDTK.

This function only changes `font`—it does not clear the screen. It is the application's responsibility to issue a `clrscr()` command *prior* to switching fonts of differing sizes (if that is desired). That is, there is *no* limitation on fonts of different sizes displayed at the same time. The application can manage this.

Note that there can be side effects from changing the font. For example, the cursor may move. The cursor *must always* be positioned on a character in the current font. This means that when the font changes if the cursor is not on a character boundary for the new font, it positions at the next closest character to the right. It is the user's responsibility to reposition the cursor to the desired location, if necessary, when changing fonts. Similarly, changing the font may affect the window. If so, the window is changed to the closest valid display coordinates for the new font size.

Prototype

```
int set_font(const char *font_name);
```

Parameters

`font_name` A null-terminated string containing the font filename. To set the font to the default font, pass a null string.

Return Values

Success: 0

Failure:

- 1 with `errno` set to `EBADF`: Specified font file not found.
- 1 with `errno` set to `EINVAL`: Font file not in correct format.
- 1 with `errno` set to `EACCES`: `font_name` is an invalid pointer.
- 1 and `errno` set to `EBADF`: Caller does not own the console device.

Other `errno` values may be set by the file manager.

set_fkey_panel()

Adds the ability for applications to set the compatibility mode frame panels on demand.

Prototype

```
set_fkey_panel(char *bmp_filename, int which_panel);
```

Parameters

`*bmp_filename`

The full name of the .bmp file to load.

`which_panel`

Any of the following panels:

`TITLE_PANEL` - top section of the compatibility frame. Panel size is 236 x 61 pixels.

`FKEY_PANEL` - right-hand vertical section of the frame. Panel size is 48 x 199 pixels.

`HKEY_PANEL` - bottom section of the frame. Panel size is 240 x 66 pixels.

set_touchscreen_keymap()

This gives applications the ability to set up touch keys on the screen. The size, number, and placement are the developers' choice. Each `touchkey_map_item` represents the placement of one key. The function reads up to 50 keys.

Prototype

```
set_touchscreen_keymap(tkm_t *map, int ct);
```

Parameters

`map` The definition of the key as follows:

```
typedef struct touchkey_map_item { // For consistency with window() etc
    unsigned short x1; // Upper left column, 0..239
    unsigned short y1; // Upper left row, 0..319
    unsigned short x2; // Lower right column, 0..239
    unsigned short y2; // Lower right row, 0..319
    unsigned int key; // Encoded key value
} tkm_t;

ct: size of map
```

sts_kbd_lock()

Returns the lock status of the keyboard.

Prototype

```
int sts_kbd_lock(void);
```

Parameters

None

Return Values

- 0 Keyboard is not locked.
- 1 Keyboard is locked.

SVC_INFO_DISPLAY()

Fills the caller's buffer with six bytes, representing the display type dimensions.

Prototype `int SVC_INFO_DISPLAY(char *stuff_6x);`

Parameters

<code>stuff_6x</code>	Pointer to the buffer.
-----------------------	------------------------

Return Values

<code>0x30 0x30 0x30 0x30 0x30 0x30</code>	= 1 line (7 segment)
<code>0x30 0x30 0x30 0x30 0x30 0x31</code>	= 2 line (7 segment)
<code>0xFF</code>	no data, since manufacturing block has not been loaded.

SVC_INFO_DISPLAY_EXT()

Retrieves information about the display module. The structure definition is found in `SVC.H`.

Prototype `int SVC_INFO_DISPLAY_EXT(displayInfo_t dispInfo);`

Example

```
// Display Information data structure
typedef struct {
    char moduleName[20]; // the display module name
    char controllerName[20]; // the display controller chip
    int driverVersion; // ie. version 2.52.142 = 0x00 02 34 8E
    int width; // display width, in pixels
    int height; // display height, in pixels
    int bitsPerPixel; // number of bits per pixel
    int pixelFormat; // the pixel format
} displayInfo_t;
```

SVC_INFO_KBD()

Fills the caller's buffer with the one-byte keyboard type from the manufacturing block.

Prototype `int SVC_INFO_KBD(char *buf);`

Parameters

<code>buf</code>	Pointer to the 2 byte buffer.
------------------	-------------------------------

Return Values

0x30	Telco type keypad (123,456,789,*0#)
0x31	Calculator type keypad (789,456,123, *0#)
0x32	Singapore calculator type keypad (789,456,123, *0#)
0x36	EBS100 keypad (123,456,789,-0+)
0x37	Vx525 same as Telco with function keys changed to telephone functions.
0xFF	No data; manufacturing block not loaded.

wherecur()

Returns the current cursor position relative to the physical display, *not* the current window.

NOTE



If the last character written is at the last position of the window, the cursor remains at the last window position.

Prototype `int wherecur(int *x, int *y);`

Parameters

<code>x</code>	Location to return the horizontal (or column) coordinates of the cursor.
<code>y</code>	Location to return the vertical (or row) coordinates of the cursor.

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : Caller does not own console.
	-1 with <code>errno</code> set to <code>EACCES</code> : Either <code>x</code> or <code>y</code> is an invalid pointer.

wherewin()

Returns the display coordinates for the current window.

`x1`, `y1`, `x2`, and `y2` are the locations to return the current window position. The current display window coordinates are copied into the four integer variables, where (`x1`, `y1`) are the column (character size) and row coordinates respectively of the upper-left corner of the window; (`x2`, `y2`) are the window's lower-right corner column (character size) and row coordinates, respectively.

Prototypes

```
int wherewin(int *x1, int *y1, int *x2, int *y2);
```

Parameters

- `x1` Pointer to location to store window top-left x (horizontal or column) coordinate.
- `y1` Pointer to location to store window top-left y (vertical or row) coordinate.
- `x2` Pointer to location to store window bottom-right x (horizontal or column) coordinate.
- `y2` Pointer to location to store window bottom-right y (vertical or row) coordinate.

Return Values

- Success: 0
- Failure: -1 with `errno` set to `EBADF`: Caller does not own console.
- 1 with `errno` set to `EACCES`: One or more argument is invalid.

wherewincur()

Returns the current cursor position relative to the current window, *not* the physical display.

NOTE



If the last character written is at the last position of the window, the cursor remains at the last window position.

Prototype

```
int wherewincur(int *x, int *y);
```

Parameters

- `x` Pointer location to store cursor x (horizontal or column) coordinate.
- `y` Pointer location to store cursor y (vertical or row) coordinate.

Return Values

- Success: 0
- Failure: -1 with `errno` set to `EBADF`: Caller does not own console.
- 1 with `errno` set to `EACCES`: One or more argument is invalid.

window()

Defines a logical window within the physical display.

If any coordinates fall outside the physical display dimensions, the minimum/maximum values are used. The cursor is placed in the home position (1,1) of the window.

Prototype `int window(int x1, int y1, int x2, int y2);`

Parameters

`x1` Window top-left x (horizontal or column) coordinate.
`y1` Window top-left y (vertical or row) coordinate.
`x2` Window bottom-right x (horizontal or column) coordinate.
`y2` Window bottom-right y (vertical or row) coordinate.

Return Values

Success: 0
 Failure -1 with `errno` set to `EBADF`: Caller does not own console.

write_at()

Similar to [write_at\(\)](#), except that the cursor is positioned prior to writing the data in the current font. The position is specified in character, 1-based, window relative coordinates. If the location is outside the window, the coordinates are forced in to the window.

As with the `write()` function, data is wrapped within the window (unless [setscrollmode\(\)](#) has been called). The cursor is positioned at the next character after the end of the text.

Prototype `int write_at(char *buf, int len, int x, int y);`

Parameters

`buf` Pointer to data to write
`len` Number of bytes in `buf` to write
`x` x (column) coordinate to start write
`y` y (row) coordinate to start write

Return Values

Success. > 0 Value will be the same as `len` regardless of the number of characters written to display (that is, if `buf` is truncated, `len` is still returned).
 Failure: -1 with `errno` set to `EBADF`: Caller does not own console device.
 -1 with `errno` set to `EACCES`: Invalid `buf` pointer.
 -1 with `errno` set to `EINVAL`: A negative length parameter was passed.

write_pixels()

Fills the specified pixel block with the specified color. The can be as small as one pixel up to the size of the display.

Prototype `int write_pixels (int startX, int startY, int endX, int endY, int color);`

Parameters The parameters describe the pixel block, where color is the color used to fill the block.

startX The leftmost column.

startY The uppermost row.

endX The rightmost column.

endY The bottommost row.

Return Values

Success: 0

Failure: -1 and errno set to `EBADF` if the application does not own the console or the console is not open.

 -1 and errno set to `EINVAL` for other errors.



CHAPTER 8

Service Function Calls

The function calls listed in this chapter retrieve information about the Verix eVo-based terminal's operating system and device settings.

get_component_vars()

Returns information about an OS component (typically a driver). This includes the component file name and timestamp that identifies the version. In addition, drivers can maintain a small amount of non-volatile data that can be used to save configuration settings or diagnostic information. For example, the console driver stores the user's contrast setting here. See also [SVC_INFO_LIFETIME\(\)](#).

For drivers associated with a device, *hdl* is the handle returned when the device is opened (however, the device does not need to be open to call this function). Components not associated with a device that can be opened have a fixed number in the range 0–31. Since the components can change with operating system releases, they are not listed. The result is stored in *buf*, with length *len*. The format is as follows:

Bytes	Description
1–12	File name. Padded with zeros (it may not be zero-terminated if the name uses all 12 bytes).
13–18	File timestamp, <i>ymdhms</i> , where each byte contains two BCD digits.
19– <i>len</i>	Variable data defined by component.

The buffer size must be at least 18 bytes. If the available data is longer than the buffer, it is truncated.

`get_component_vars()` returns the non-volatile data for the communication port.

NOTE



On VX 820 DUET, this returns a file name of “com4_usb.bin” for the USB thermal printer device — the Micro-controller firmware.

Prototype

```
int get_component_vars(int hdl, char *buf, int len);
```

Return Values

Success:	Information about OS component
Failure:	<p>–1 and <i>errno</i> set to <code>EACCESS</code>: Invalid buffer pointer.</p> <p>–1 and <i>errno</i> set to <code>EINVAL</code>: Invalid component number or <i>len</i> is less than 18.</p> <p>–1 and <i>errno</i> set to <code>EBADF</code>: <i>comm_handle</i> is not a valid open file descriptor.</p> <p>–1 and <i>errno</i> set to <code>ENXIO</code>: USB device not present.</p>
com3.usb.bin	On VX 820, <code>get_component_vars()</code> returns this file for the USB modem device.

Example

The linked example program displays the file name and build time of each OS component.

set_combo_mode()

Sets the module specified by mode either in conventional telephone modem or as a TCP/IP adapter. This function is not supported in Trident. The API exists and will return -EINVAL.

Prototype `int set_combo_mode(int mode);`

Parameter

mode	0: Landline.
	1: Alternate communication device such as TCP/IP.

Return Values

Success:	4: Conexant Banshee
	22: Conexant Banshee/CO210 combo
Failure:	-1 and errno set to:
	ENODEV: Device requested not available (modem or TCP/IP not present)
	EINVAL: Caller's option is not 0 or 1
	EBADF: COM 3 not open.

SVC_CHK_PASSWORD()

Compares the counted string in buffer to the password for the current group. See [DLLs](#) for a description of counted strings.

Prototype `int SVC_CHK_PASSWORD (const char *buf);`

Return Values

1: Contents of buffer match password.

0: Contents of buffer do not match password.

-EACCES: Caller's buffer is not readable.

SVC_INFO_BAT_REQ()

Returns information from the configuration file (CIB) that indicates if a battery is required. `Char1` returns 1 if battery is required; 0 if not. This function is not supported in Predator.

Prototype `SVC_INFO_BAT_REQ(char *char1);`

Parameters

`char1` Pointer to a char.

Return Values

1: Battery is required

0: Battery is not required

2: Battery is required for printing, but not for GPRS SIM protection.

SVC_INFO_COUNTRY()

Stores 12 bytes of factory-defined country variant data in the caller's buffer. The data is ASCII but not zero-terminated. There is no uniform convention for its contents.

Country data is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_COUNTRY (char *buf_12);`

Return Values

Success: 0

Failure: -1 with errno set to -EACCES: The caller's buffer is not writable.

SVC_INFO_COUNTRY_EXT()

This is a Trident only function. `SVC_INFO_COUNTRY_EXT` is the same as `SVC_INFO_COUNTRY` with the exception of the added length of the `buf` parameter. This function returns the country field of the MIB.

Prototype `int SVC_INFO_COUNTRY_EXT(char *buf, int len);`

Parameters

`buf` Buffer that receives the MIB data.

`len` Maximum buffer length.

Return Values

Success: 0

SVC_INFO_HW_VERS()

Stores a 2-byte factory-defined hardware version in the caller's buffer. The data is ASCII, but not zero-terminated. The hardware version is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_HW_VERS (char *buf_2);`

Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_HW_VERS_EXT()

This is a Trident only function call. `SVC_INFO_HW_VERS_EXT` is the same as `SVC_INFO_HW_VERS` with the exception of the added buffer parameter length. This function returns the HW version field of the MIB.

Prototype `int SVC_INFO_HW_VERS_EXT(char *buf, int len);`

Parameters

buf Buffer that receives the MIB data.

len Maximum buffer length

Return Values

Success: 0

SVC_INFO_KEYBRD_TYPE()

Trident CIB field operation. This function returns the keyboard style as defined in the CIB.

On VX 520 Color D/E/CTLS, this function call will return '3'.

Prototype `int SVC_INFO_KEYBRD_TYPE(void);`

Return Values

Success: 0 = No ATM or Function or ALPHA keys
 1 = ATM and Function keys present
 2 = No ATM or Function keys but ALPHA key present

SVC_INFO_LOTNO()

Stores a 6-byte factory-defined manufacturing lot number in the caller's buffer. The data is ASCII, but not zero-terminated. The lot number is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_LOTNO (char *buf_6);`

Return Values

Success: 0
 Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_LOTNO_EXT()

This is a Trident only function call. SVC_INFO_LOTNO_EXT is the same as SVC_INFO_LOTNO with the exception of the added buffer parameter length. This function returns the lot number field of the MIB.

Prototype `int SVC_INFO_LOTNO_EXT(char *buf, int len);`

Parameters

buf	Buffer that receives the MIB data.
len	Maximum buffer length.

Return Values

Success:	0
----------	---

SVC_INFO_MAG()

Stores a 1-byte magnetic card reader type code in the caller's buffer. The codes are defined as follows:

- 0: No card reader present.
- B: Triple track MSR device based on the scanning ADC of the SoC (such as, SW data separator/decoder).

The card reader type is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_MAG(char *buf_1);`

Return Values The result is an ASCII character, not a binary number.

Success: 0

Failure: Non-zero, with the only failure condition an invalid buffer pointer. errno is unchanged.

SVC_INFO_MFG_BLK()

Stores 30 bytes of factory-defined manufacturing data in the caller's buffer. The data is ASCII text, but not zero-terminated. There is no standard format for its contents. Manufacturing data is part of a 128-byte *manufacturing block* stored in flash memory, which is set by the factory. In addition to the manufacturing data, this block describes various options such as, the keypad layout, display size, the availability of an internal PIN pad, and so on. Applications can query this information to adjust behavior for different terminals.

Table 22 summarizes the contents of the manufacturing block and the function used to retrieve each field. See the individual function descriptions for more details.

In general, all data is ASCII text but is not zero-terminated. Since the block is loaded during manufacturing, the OS cannot guarantee it is used as described. If the block has not been loaded, all bytes have the hexadecimal value 0xFF.

Table 22 **Manufacturing Block Contents**

Field	Size	Related Function
Manufacturing Data	30	SVC_INFO_MFG_BLK()
Model Number	12	SVC_INFO_MODELNO()
Country	12	SVC_INFO_BAT_REQ()
Part Number	12	SVC_INFO_MODELNO_EXT()
Hardware version	2	SVC_INFO_HW_VERS()
Lot Number	5	SVC_INFO_LOTNO()
Serial Number	11	SVC_INFO_SBI_VER()
Permanent Terminal ID	8	SVC_INFO_READ_MIB()
Keypad Type	1	SVC_INFO_HW_VERS_EXT()
Mag Card Reader Type	1	SVC_INFO_LOTNO_EXT()
Display Type	6	SVC_INFO_HW_VERS()
Printer Type	1	SVC_INFO_PRNTR()
PIN pad Type	1	SVC_INFO_PIN_PAD()

Prototype `int SVC_INFO_MFG_BLK (char *buf_30);`

Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

Example The linked example presents code typically used to retrieve manufacturing block fields. Note a check of return values is unnecessary since it is known that the buffer pointer is valid.

SVC_INFO_MFG_BLK_EXT()

This is a Trident only function. SVC_INFO_MFG_BLK_EXT is the same as SVC_INFO_MFG_BLK with the exception of the added length of the buffer parameter. This function returns the manufacturing block data.

Prototype `int SVC_INFO_MFG_BLK_EXT(char *buf, int len);`

Parameters

buf	Buffer to receive the MIB data
len	Maximum buffer length

Return Values

Success: 0

SVC_INFO_MOD_ID()

Returns a code indicating the type of modem installed. This is required to support radio-only terminals where PSTN modem is not populated.

Prototype `int SVC_INFO_MOD_ID(void);`

Return Values

Success: Value of installed modem. Should be opened as
"/dev/com3."

Failure: -1 and ERRNO set to EINVAL, function not supported.

NOTE



On VX 680/V*670 this function always returns 50. This is also true for VX 820 / V*810 PIN pad whether it operates as a stand-alone device or is connected to DUET base station.

SVC_INFO_MODULE_ID()

Takes a COM port number such as 2 for COM2 or 3 for COM3 and returns the module ID for that COM port.

NOTE

The V^x610 terminal now uses the Kyocera M200 CDMA radio module in place of the Sierra EM3420 CDMA radio module. The return value for `SVC_INFO_MODULE(2);` is 70.

On VX 820, this always returns a value 50. This is true whether the VX 820 PIN pad is operating as a stand-alone device or is connected to a DUET. This is similar to how the VX 680 terminal operates.

Prototype

```
int SVC_INFO_MODULE_ID(int port);
```

Return Values

Success: Module ID of COM port.

Failure: If the port parameter is other than 2 or 3:

- `EINVAL`, returns a negative value. If this call is made on an OS that doesn't support the call, it returns a negative value.

The following list is found in `SVC.H` and describes all device MID values. Some of these may not be returned by this function but are used or returned in others.

<code>MID_NO_MODEM</code>	0	No modem on COM3
<code>MID_UNKNOWN_MODEM</code>	2	Unknown modem on COM3
<code>MID_TDK_ONLY</code>	3	TDK 2.4 modem only
<code>MID_BANSHEE_ONLY</code>	4	BANSHEE 144 modem, also in <code>_info.c</code>
<code>MID_CARLOS_ONLY</code>	5	Carlos (Aspen144) modem only
<code>MID_CO561_ONLY</code>	6	Connect One 10BaseT only
<code>MID_CARLOS_CO561</code>	7	Carlos/CO 10baseT combo
<code>MID_MC56_ONLY</code>	8	GSM/GPRS US only
<code>MID_MC55_ONLY</code>	9	GSM/GPRS International only
<code>MID_EM3420_ONLY</code>	10	CDMA 1xRTT only
<code>MID_CO710_ONLY</code>	11	Connect One WiFi
<code>MID_CARLOS_MC56</code>	12	Carlos/GSM US combo
<code>MID_CARLOS_MC55</code>	13	Carlos/GSM Int combo
<code>MID_CARLOS_EM3420</code>	14	Carlos/CDMA combo
<code>MID_CARLOS_CO710</code>	15	Carlos/WiFi combo
<code>MID_EISENHOWER_ONLY</code>	16	Conexant Eisenhower modem only
<code>MID_EISEN_USB_ETHERNET</code>	17	Eisenhower/USB Ethernet combo

MID_EISEN_EM3420	18	Eisenhower/CDMA combo
MID_EISEN_MC56	19	Eisenhower/GPRS USA combo
MID_EISEN_MC55	20	Eisenhower/GPRS International combo
MID_EISEN_USB_WIFI	21	Eisenhower/USB WiFi combo
MID_BANSHEE_CO210	22	Banshee/CO210 Ethernet combo
MID_CO210_ONLY	23	CO210 Ethernet
MID_ISDN_ONLY	24	ISDN
MID_BANSHEE_USB_ETHER	25	Banshee/USB Ethernet combo
MID_HARLEY_MODEM	40	Trident Harley Modem
MID_COM2_UART	42	COM2 is configured as 2 wire UART
MID_USB_MODEM	50	USB modem
MID_TELEPHONE	51	Vx525 TelePOS
MID_BTEZ1	60	BT Ezurio brand module 1
MID_BTEZ2	61	BT Ezurio brand module 2
MID_BTEZ3	62	BT Ezurio brand module 3
MID_BTEZ4	63	BT Ezurio brand module 4
MID_BTAA1	64	BT alternate vendor module 1
MID_BTAA2	65	BT alternate vendor module 2
MID_BTAA3	66	BT alternate vendor module 3
MID_BTAA4	67	BT alternate vendor module 4
MID_M200	70	Kyocera M200 CDMA
MID_MC55i_ONLY	72	Sierra MC55i GPRS
MID_MC5727	73	Sierra MC5727 CDMA
MID_SOC_ETH	74	Internal Ethernet
MID_USB_HOST_PWR	75	Powered USB Host
MID_USB_HOST_NO_PWR	76	USB Host not powered
MID_USB_HOST_HUB	77	USB with internal hub
MID_USB_DEV	78	USB device
MID_CTL5	79	Contactless for the Vx680
MID_SD_A	80	SD Slot A
MID_SD_B	81	SD Slot B
MID_TOUCH_RES	82	Touchscreen type - resistive
MID_TOUCH_CAP	83	Ttouchscreen type - capacitive
MID_HAUWEI_EM660	84	HAUWEI EVDO radio

MID_DUET820_MCU	85	Vx820 DUET base
MID_BCM43291_WIFI	86	BCM432291 WiFi
MID_BCM43291_BT	87	BCM432291 BlueTooth
MID_BGS2	88	Cinterion BGS2 GSM/GPRS radio
MID_PHS8P	89	Cinterion PHS8-P 3G radio + GPS
MID_PHS8P_NOGPS	90	Cinterion PHS8-P 3G radio - GPS
MID_TBD	99	To Be Determined

SVC_INFO_MODELNO()

Stores a 12-byte factory-defined model number in the caller's buffer. The data is ASCII, but not zero-terminated. No standard format for model numbers is defined. The model number is stored in the manufacturing block.

See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_MODELNO (char *buf_12);`

Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_MODELNO_EXT()

This is a Trident only call. `SVC_INFO_MODELNO_EXT` is the same as `SVC_INFO_MODELNO` with the exception of the added length of the buffer parameter. This function returns the model number field of the MIB.

Prototype `int SVC_INFO_MODELNO_EXT(char *buf, int len);`

Parameters

buf Buffer that receives the MIB data

len Maximum buffer length

Return Values

Success: 0

SVC_INFO_MODEM_TYPE()

Trident CIB field operation, returns the modem type used by the terminal as defined in the CIB, either the Conexant Harley modem or the Silicon Laboratories modem.

Prototype `int SVC_INFO_MODEM_TYPE(void);`

Return Values Success: MID_HARLEY_MODEM or MID_USB_MODEM as defined in SVC.H.

SVC_INFO_MODEM_TYPE_EX()

Trident CIB field operation, returns the modem type used by the terminal as defined in the CIB, either the Conexant Harley internal modem/USB modem (used on VX 675 charging base) or the Silicon Laboratories modem.

Prototype `modem_type_tSVC_INFO_MODEM_TYPE_EX(void);`

Return Values Success: ModemNone Or ModemHarleyInternal Or ModemHarleyRemovable Or
ModemSiLabsRemovable as defined in SVC.H

SVC_INFO_OS_HASH()

Allows an application to compute a checksum for the entire operating system. This computes the 20-byte HMAC-SHA1 hash value for the operating system in the terminal, with the specified key used as a seed. See Internet RFC 2104 for a description of the algorithm used.

Prototype `int SVC_INFO_OS_HASH (U8* hashout, U8* keyin, int keysz);`

SVC_INFO_OS_HMAC_SHA1()

Computes the HMAC SHA hash value for the OS and certificate tree.

Prototype `int SVC_INFO_OS_HMAC_SHA1(char *hashout20,char *keyin,int keyinsz);`

Parameters

hashout20	Pointer to a buffer to receive the hash.
keyin	Pointer to the initial key for the computation
keyinsz	Size of the key

Return Values

Success: 0
Failure: -1 with errno set to EACCES

SVC_INFO_OS_HMAC_SHA256()

Computes the HMAC SHA 256 hash value for the OS. At present the mode is ignored.

Prototype `int SVC_INFO_OS_HMAC_SHA256(char *hashout32,char *keyin,int keyinsz, int mode);`

Parameters

hashout20	Pointer to a buffer to receive the hash.
keyin	Pointer to the initial key for the computation
keyinsz	Size of the key
mode	Reserved

Return Values

Success: 0
Failure: -1 with errno set to EACCES

SVC_INFO_PARTNO()

Stores a 12-byte factory-defined part number in the caller's buffer. The data is ASCII, but not zero-terminated. The part number is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_PARTNO (char *buf_12);`

Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_PARTNO_EXT()

A Trident only function call. SVC_INFO_PARTNO_EXT is the same as SVC_INFO_PARTNO with the exception of the added length of the buffer parameter. This function returns the part number of the MIB.

Prototype `int SVC_INFO_PARTNO_EXT(char *buf, int len);`

Parameters

buf Buffer that receives the MIB data.

len Maximum buffer length.

Return Values

Success: 0

SVC_INFO_PIN_PAD()

Stores a 1-byte PIN pad type code in the caller's buffer, as follows:

- 1 indicates that an internal PIN pad is installed.
- 0 indicates none installed.

The PIN pad type is stored in the manufacturing block.

See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

The result is an ASCII character, not a binary number.

Prototype `int SVC_INFO_PIN_PAD (char *buf_1);`

Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_PORT_IR()

Returns the serial port number for infrared communication, if the terminal supports infrared communications. See [SVC_INFO_PORT_MODEM\(\)](#) for an example of the use of a similar function.

Prototype `int SVC_INFO_PORT_IR (void);`

Return Values

Failure: -1: Infrared communications not supported in this terminal.

SVC_INFO_PORT_MODEM()

Returns the serial port number connected to the modem. This applies only to terminals with a built-in external modem (that is, a modem inside the case accessed through an internally wired serial port). It does not apply to separate external modems connected to the terminal by a cable.

Prototype `int SVC_INFO_PORT_MODEM (void);`

Return Values

Success: 3: The modem device should be opened as `"/dev/com3"`.

Failure: -1: No modem included.

Example

The function example in the linked file opens the modem port and returns its handle.

NOTE



On VX 820 DUET, these values are true even if the VX 820 is operating as stand-alone device or connected to the DUET.

SVC_INFO_PORTABLE()

Indicates the presence of battery power. It returns 1 if the unit can run on battery power alone, and 0 if otherwise.

Prototype `int SVC_INFO_PORTABLE(char *char1);`

Parameters

`char1` Pointer to a char.

SVC_INFO_PRNTR()

Stores a one-byte ASCII printer type code in the caller's buffer, as follows:

- 1 indicates an internal printer is installed.
- 0 indicates no internal printer is installed.

Note that the function cannot determine if an external printer is connected to one of the serial ports.

The printer type is stored in the manufacturing block.

See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples.

Prototype `int SVC_INFO_PRNTR (char *buf_1);`

Return Values The result is an ASCII character, not a binary number.

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.
 255: The manufacturing block is not set.

SVC_INFO_READ_MIB()

This is a Trident function call only. This function call returns the MIB.

Prototype `int SVC_INFO_READ_MIB(char *buf, int len);`

Parameters

buf	Pointer to buffer to receive the MIB.
len	Maximum buffer length

Return Values Number of bytes copied.

SVC_INFO_RELEASED_OS()

Examines the OS name. If the name is formatted as a released OS, it returns “OS is released”. Otherwise, it returns “not released.”

Prototype `int SVC_INFO_RELEASED_OS(void);`

Return Values

Success: 0 - OS is not a released version.
 1 - OS is a released version.

Failure: -1

SVC_INFO_RESET()

Stores the time of the last terminal reset in the caller's buffer and returns the total number of resets in the terminal's lifetime (since the current OS was loaded). The time is 12 bytes of ASCII data, *YYMMDDHHMMSS*, year, month, day, hour, minute, and second. It is not terminated.

Prototype `int SVC_INFO_RESET (char *buf_12);`

Return Values

Success: 0

Failure: -1 with errno set to `EACCES`: Invalid buffer pointer provided.

Example Click on the linked example to view the sample code.

SVC_INFO_SBI_VER()

SBI is a Trident terminal executable file that is started by the terminal processor. The SBI in turn validates and starts the OS. SVC_INFO_SBI_VER returns the version of the SBI.

Prototype `int SVC_INFO_SBI_VER(char *SBIver5, int bufLen);`

Parameters

SBIver5	Pointer to a buffer that receives the SBI version.
bufLen	Buffer length, must be at least the size of the version

Return Values

Success:	0
Failure:	-1 with errno set to EINVAL.

SVC_INFO_SERLNO()

Stores an 11-byte factory-set serial number in the caller's buffer. Injecting serial numbers is a manufacturing option so this data may be blank or set to a default value. The data is ASCII but not zero-terminated. The serial number is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and examples. See also [SVC_INFO_READ_MIB\(\)](#).

Prototype `int SVC_INFO_SERLNO (char *buf_11);`

Return Values

Success:	0
Failure:	-1 with errno set to EACCES: Invalid buffer pointer provided.

SVC_INFO_SERLNO_EXT()

This is a Trident only function call. SVC_INFO_SERLNO_EXT is the same as SVC_INFO_SERLNO with the exception of the added buffer parameter length. This function returns the serial number field of the MIB.

Prototype `int SVC_INFO_SERLNO_EXT(char *buf, int len);`

Parameters

buf	Buffer that receives the MIB data.
len	Maximum buffer length.

Return Values

Success: 0

SVC_INFO_URT0_TYPE()

This is a Trident only function call. SVC_INFO_URT0_TYPE returns the SOC UART0 (COM1) type, or the number of wires in the interface as listed below:

- A 2 wire interface only has Tx and Rx lines.
- A 4 wire interface has Tx, Rx, CTS, and RTS lines.
- A 6 wire interface has Tx, Rx, CTS, RTS, DTR, and DCD lines
- A 8 wire interface has Tx, Rx, CTS, RTS, DTR, DCD, DSR, and RI lines.

Prototype `int SVC_INFO_URT0_TYPE(char *char1);`

Parameters

char1	Pointer to a 1-byte buffer. Returns 0x32, 0x34, 0x36, or 0x38.
-------	--

Return Values

Success: 0

SVC_INFO_URT1_TYPE()

This is a Trident only function call. SVC_INFO_URT1_TYPE returns the SOC UART1 (COM2) type, or the number of wires in the interface as listed below:

- A 2 wire interface only has Tx and Rx lines.
- A 4 wire interface has Tx, Rx, CTS, and RTS lines.
- A 6 wire interface has Tx, Rx, CTS, RTS, DTR, and DCD lines.
- A 8 wire interface has Tx, Rx, CTS, RTS, DTR, DCD, DSR, and RI lines.

Prototype `int SVC_INFO_URT1_TYPE(char *char1);`

Parameters

<code>char1</code>	Pointer to a 1-byte buffer. Returns 0x32, 0x34, 0x36, or 0x38.
--------------------	--

Return Values

Success:	0
----------	---

SVC_INFO_URT2_TYPE()

This is a Trident only function call. SVC_INFO_URT2_TYPE returns the SOC UART2 (COM3) type, or the number of wires in the interface as listed below:

- A 2 wire interface only has Tx and Rx lines.
- A 4 wire interface has Tx, Rx, CTS, and RTS lines.
- A 6 wire interface has Tx, Rx, CTS, RTS, DTR, and DCD lines.
- A 8 wire interface has Tx, Rx, CTS, RTS, DTR, DCD, DSR, and RI lines.

Prototype `int SVC_INFO_URT2_TYPE(char *char1);`

Parameters

char1 Pointer to a 1-byte buffer. Returns 0x32, 0x34, 0x36, or 0x38.

Return Values

Success: 0

SVC_INFO_URT3_TYPE()

This is a Trident only function call. SVC_INFO_URT3_TYPE returns the SOC UART3 (COM7) type, or the number of wires in the interface as listed below:

- A 2 wire interface only has Tx and Rx lines.
- A 4 wire interface has Tx, Rx, CTS, and RTS lines.
- A 6 wire interface has Tx, Rx, CTS, RTS, DTR, and DCD lines.
- A 8 wire interface has Tx, Rx, CTS, RTS, DTR, DCD, DSR, and RI lines.

Prototype `int SVC_INFO_URT3_TYPE(char *char1);`

Parameters

char1 Pointer to a 1-byte buffer. Returns 0x32, 0x34, 0x36, or 0x38.

Return Values

Success: 0

SVC_LED()

Sets the light-emitting diode specified by ID on or off (on if mode = 1 or off if mode = 0). The number and location of application-controllable LEDs varies in different terminal models. Some Verix eVo-based terminals have no LEDs. Referencing a non-existent LED number may not be recognized as an error because the OS does not always know which LEDs are connected.

Prototype `int SVC_LED (int id, int mode);`

Return Values

Success: 0

Failure: -1 with errno set to EINVAL.

NOTE



The VX 820 PIN pad does not support a printer and a battery. No LEDs are supported to show their status.

SVC_LEDS()

This Trident function reads/writes to the LEDs depending on the selected mode.

Prototype `int SVC_LEDS(int mode, void *param);`

Parameters

mode 0: LED_READ_MONO (defined in SVC.H)
 1: LED_WRIETE_MONO (defined in SVC.H)
 param Buffer that receives read data.

Return Values

Success: 0

Failure: -1 with errno set to EINVAL.

SVC_SHUTDOWN()

Commands the portable or battery powered terminals such as V*670 or VX 680 to turn off. SVC_SHUTDOWN gives an audible notification to the user via a half-second “buzz” from the beeper. The terminal issues a system event, EVT_SHUTDOWN, every second until it turns itself off. All applications should take the event and shut down by closing sockets, logging off networks, and closing files, among others. The system displays SHUTTING DOWN xx message, where xx is a countdown starting from *OFFD. When the count reaches 0, the terminal shuts off. Applications that take the event should exit and not use SVC_SHUTDOWN.

Prototype `int SVC_SHUTDOWN (void);`

Return Values

Success: 0

Failure: -22 if the function detects that it is not VX 680.

SVC_INFO_LIFETIME()

Returns the total number of seconds the terminal has been in operation. The counter increments once per second when the terminal is powered on, and is reset only if a new operating system is loaded.

Although the result is declared as long, it is maintained by Verix eVo OS as a 32-bit unsigned value, and applications that expect to be in use for more than 68 years should cast it to `unsigned long`. It wraps back to 0 after approximately 136 years of operating time.

Prototype `long SVC_INFO_LIFETIME (void);`

SVC_INFO_CRASH()

Retrieves diagnostic information about the most recent fatal exception. Fatal exceptions include address errors, division by zero, illegal instructions, and so on. If an application causes these types of errors, Verix eVo OS traps it and saves some diagnostic information in a *crash* log (also known as an *error* log). The information can be viewed through a VTM screen or the debugger, and can be retrieved by an application through this function. It is retained until an overwrite by another exception.

Prototype

```
void SVC_INFO_CRASH (struct info_crash_t *results);
```

The crash log format is described by the `info_crash` structure:

```
struct info_crash {  
    unsigned long usr_regs[16]; /* R0-R15 */  
    unsigned long cpsr;        /* CPSR */  
    unsigned long und_regs[3]; /* SPSR_und, R13_und, R14_und */  
    unsigned long abt_regs[3]; /* SPSR_abt, R13_abt, R14_abt */  
    unsigned long irq_regs[3]; /* SPSR_irq, R13_irq, R14_irq */  
    unsigned long fiq_regs[8]; /* SPSR_fiq, R08_fiq..R14_fiq */  
    unsigned long svc_regs[3]; /* SPSR_svc, R13_svc, R14_svc */  
    unsigned long fault_addr; /* bad address for data abort */  
    int abort_type;           /* 1 = data, 2 = prog, 3 = undef */  
    int task_id;              /* which task */  
    char time[6];             /* time of crash: BCD format, yymmddhhmmss */  
};
```

SVC_INFO_EPROM()

Stores a counted string that contains an 8-byte firmware version in the caller's buffer. This value identifies the OS version. Note however, that if updated OS components are downloaded to flash, the change is not reflected in the version number. See also [SVC_INFO_LIFETIME\(\)](#).

Verix eVo OS version numbers are in the form *QThhvmm*, where:

- Q Designates the Verix eVo operating system
- T T - If the terminal is Trident, A-V×510, B- V×610, C - V×570, D - V×670, E - V5, G - V×810, H - V×700, J - V×510 GPRS
- hh Hardware definition, currently 00. In Trident 00 means the OS does not support directories if 01 or greater then directories are supported. Predator 20 means the OS supports RKL.
- vv Released version, beginning with 00
- mm Minor release, usually A0. Predator release will always have alpha character, Trident will always have number.

Prototype `void SVC_INFO_EPROM (char *buf_9);`

Example See the linked example in [SVC_INFO_LIFETIME\(\)](#).

SVC_INFO_PTID()

Stores a counted string that contains an 8-byte terminal identification number in the caller's buffer. The value is either an identifier unique to the individual terminal, or the default value, 12000000.

The PTID is stored in the manufacturing block. See [SVC_INFO_MFG_BLK\(\)](#) for additional notes and an example. Note however, that this function differs from the other manufacturing block query functions in that it returns the result as a counted string. (for backwards compatibility.)

See also [SVC_INFO_SBI_VER\(\)](#).

Prototype `void SVC_INFO_PTID (char *buf);`

SVC_VERSION_INFO()

Stores a counted string that contains the OS version information, which supplements the version number returned by [SVC_INFO_HW_VERS\(\)](#). This is typically about 16 characters and includes the build date. Note that if updated OS components are downloaded into flash, the change is not reflected in the information returned. See also [SVC_INFO_HW_VERS\(\)](#) and [get_component_vars\(\)](#).

Prototype `void SVC_VERSION_INFO (char *buf);`

Example The output buffer will typically contain a formatted date indicating when the OS was built, plus the name of the OS.

For example, the first byte contains binary 17, followed by "01/23/2004 Verix."

SVC_LRC_CALC()

Calculates the LRC (longitudinal redundancy check) value for `size` bytes of data in `buf`. The LRC is simply the XOR of the bytes. `seed` is a starting value with which the bytes are XORed. Set `seed` to zero to get the LRC for just the data in the buffer. See also [SVC_CRC_CALC\(\)](#).

Prototype `unsigned char SVC_LRC_CALC`
 `(const void *buf, int size, unsigned char seed);`

Example The linked code example sends a packet that consists of a fixed header, followed by variable data, followed by an LRC covering both. `send()` is an assumed application function.

SVC_MEMSUM()

Computes the sum of `size` bytes from `buf`, treating both the bytes and the sum as unsigned, and ignoring overflows. Used as a checksum, this is significantly faster than a CRC, but less sensitive to errors. See also [SVC_CRC_CALC\(\)](#).

Prototype `unsigned int SVC_MEMSUM (const char *buf, long size);`

SVC_MOD_CHK()

Generates a *Luhn check* digit for a sequence of digits or validates a sequence of digits containing a check digit. See also [SVC_CRC_CALC\(\)](#).

This function answers two possible questions -- although normally the caller is interested in only one of the answers. Both answers are encoded in the low-order 16 bits of the integer result returned.

The input array `acct` is a counted string (see [SVC_AZ2CS\(\)](#)) containing a sequence of ASCII digits. The two questions of interest are:

- Is this account number valid?
- What number must be appended to create a valid account number?

The low-order 8 bits contain the calculated check digit which should be appended to the existing array to create a *valid* account number.

Bits 8-15 contain a boolean result indicating whether the existing array is a valid account number. It is zero if the sequence is valid or 0xFF if not. Normally only one of the two bytes is used by the caller. The result is undefined if the string contains any non-digit characters.

The Luhn check is a standard scheme for detecting data entry and transmission errors in account numbers. Letting the least-significant (right-most) digit be digit 1, each odd-numbered digit is added to the sum, and each even-numbered digit is “double-added.” Double-adding means that the digit is doubled, then if that produces a two digit result, they are added and the result is added to the main sum. The string is valid if the final sum is a multiple of 10. For example, [Table 23](#) considers the 16-digit account number 4168641234567890:

Table 23 Luhn Check Example

Odd Digits	Even Digits	Evens Double-Added
1	4	8
8	6	3
4	6	3
2	1	2
4	3	6
6	5	1
8	7	5
0	9	9
33		37

Since the grand total of $33 + 37 = 70$, and 70 is a multiple of 10, the number is valid.

Prototype unsigned int SVC_MOD_CHK (const char *acct);

Example 1 The function in the linked file checks if an account number is valid. The argument is assumed to be a zero-terminated string.

Example 2 In this linked file example, the function appends a Luhn-check digit to a digit string. The argument is assumed to be a counted string.

FIFOs

This section presents function calls to use for FIFOs for all communication devices. The use of these routines is completely optional. First-in-first-out data structures, called FIFOs, are often used for communications, but they can also be generally useful in applications. The Verix eVo linkable library contains a number of routines to support FIFOs. This section describes the routines available in the library. In Verix eVo OS, a FIFO is a first-in-first-out queue of bytes, typically used as a buffer. It is stored in a `fifo_t` structure, defined as follows:

```
typedef struct fifo {  
    long ldata [3];  
    char cdata [1]; /* dummy - actual length varies */  
} fifo_t;
```

To create a FIFO with a capacity of `datasize` bytes, you must allocate `fifo_t` with at least `datasize+1` bytes in its `cdata` array. Note that you cannot simply declare `fifo_t` because it is defined with a dummy length. One way to create a FIFO is to `malloc` it:

```
fifo_t *my_fifo = malloc(sizeof(fifo_t)+SIZE);  
clr_fifo(my_fifo, SIZE);
```

To create a local or global FIFO, use the following type of declaration:

```
struct {  
    fifo_t fifo;  
    char buf[SIZE];  
} my_fifo;  
SVC_CLR_FIFO(&my_fifo.fifo, SIZE);
```

NOTE



Applications should not directly access `fifo_t` fields.

First-in First-out (FIFO) Buffers

A FIFO is a “first-in-first-out” circular buffer used to efficiently manage buffer space. FIFOs are character oriented. They are formatted as shown below, assuming that up to N bytes need to be stored at one time:

Offset	Description
0	ENDPTR: Address of first byte beyond this FIFO
4	RDPTR: Address where first byte of user data can be read, if RDPTR != WRPTR
8	WRPTR: Address where next byte of user data can be written
12	Up to N bytes of user data
12+N+1	(first byte beyond this FIFO)

FIFOs are used primarily by communications device drivers to:

- 1 Store incoming and outgoing data streams in character-oriented communications.
- 2 Manage dynamic allocation of System Pool buffers. The applications level user of a device driver is not aware of these buffer structures.

SVC_CHK_FIFO()

Returns the number of bytes currently stored in the FIFO (that is, those written to it, but not yet read). See also [SVC_CLR_FIFO\(\)](#).

Prototype `int SVC_CHK_FIFO (const fifo_t *fifo);`

SVC_CLR_FIFO()

Initializes the FIFO data structure pointed to by `fifo` with a capacity of `datasize` bytes.

NOTE



This function *must* be called to initialize a FIFO before any of the other FIFO routines are called to use the FIFO.

Prototype `int SVC_CLR_FIFO(fifo_t *fifo, int datasize);`

Return Values This function call always returns 0.

SVC_GET_FIFO()

Retrieves byte from FIFO. This function removes and returns the next unread byte from the specified `fifo`. It is returned as an unsigned character value (0-255). See also [SVC_READ_FIFO\(\)](#) and [SVC_CLR_FIFO\(\)](#).

Prototype `int SVC_GET_FIFO (fifo_t *fifo);`

Return Values Returns -1 if the FIFO is empty.

SVC_PUT_FIFO()

Add a byte to FIFO. This function appends the unsigned character, `val`, to the specified FIFO. If `val` is not in the unsigned character range, it is truncated. See also [SVC_WRITE_FIFO\(\)](#) and [SVC_CLR_FIFO\(\)](#).

Prototype `int SVC_PUT_FIFO (fifo_t *fifo, int val);`

Return Values Success: 1
Failure: 0: FIFO full.

SVC_READ_FIFO()

Reads bytes from FIFO. This function removes the next `size` bytes from the specified FIFO and stores them in `buf`. If there are fewer than `size` bytes in the FIFO, all remaining bytes are returned. See also [SVC_GET_FIFO\(\)](#) and [SVC_CLR_FIFO\(\)](#).

Prototype `int SVC_READ_FIFO (fifo_t *fifo, char *buf, int size);`

Return Values The function returns the number of bytes read.

SVC_WRITE_FIFO()

Writes bytes to FIFO. This function appends `size` bytes from buffer to the specified FIFO. If there is insufficient capacity to add `size` bytes, as many as can fit are written. See also [SVC_PUT_FIFO\(\)](#) and [SVC_CLR_FIFO\(\)](#).

Prototype `int SVC_WRITE_FIFO (fifo_t *fifo, const char *buf, int size);`

Return Values Returns the number of bytes added.

Example Click the linked example to view the sample code.

CRCs

Cyclic Redundancy Checks (CRCs) are a form of checksum used to detect data communication errors. The sender typically computes a 16- or 32-bit CRC for a packet, and appends it to the data. The receiver computes the same CRC function on the received data to verify that it transmitted correctly. A convenient property of CRCs is that if the CRC for some data is appended to it, the CRC computed over the resulting data plus CRC is zero.

Conceptually the data is processed as a stream of bits. In hardware, the CRC can be updated as each bit transmits or is received. Software implementations usually process data a byte at a time. Each step combines the CRC for the previous bytes with the next byte to calculate an updated CRC. The starting “seed” value for the first byte is usually 0 or -1 (all 1 bits). A CRC can be computed piece wise by using the CRC for the first part of a packet as the seed for the next part.

CRCs are based on a polynomial function of the data bits. It is not necessary to understand the mathematics involved to use CRCs, but it is obviously important that senders and receivers agree on the function to use. Algorithms are characterized by the polynomial and the size of the result (usually 16 or 32 bits). In addition, CRC implementations can vary in:

- The order that they process the bits. Hardware implementations commonly start with the least-significant bit of each byte, while software implementations often start with the most-significant bit.
- The byte order of the result. CRC bytes are numbered CRC1, CRC2, and so on, in the order they are transmitted. If the result is returned as a 2- or 4-byte integer, the bytes may be in forward or reverse order. Furthermore, integers can be stored most-significant byte first (for big-endian processors, for example the 68000), or least-significant byte first (for little-endian processors, for example the Z81 or Pentium).
- The seed value, usually all 0s or all 1s. The all 1s case is often specified as “-1”, although “~0” would be better C usage.

Table 24 summarizes the CRC algorithms supported in the Verix eVo environment. The “Result” and “Seed” columns give the byte order (for example, 2,1 indicates that CRC2 is the most-significant byte and CRC1 is the least-significant byte). “Type” indicates the type code used to invoke the function through `SVC_CRC_CALC()`. Additional details are in the individual function descriptions.

Table 24 CRC Algorithms Supported by Verix eVo

Function	Algorithm	Size	Bit Order	Seed	Result	Default Seed	Type
<code>SVC_CRC_CALC()</code>	LRC	8	—	—	—	0	0
<code>SVC_CRC_CRC16_L()</code>	CRC16	16	LSB first	1,2	2,1	0	1
<code>SVC_CRC_CRC16_M()</code>	CRC16	16	MSB first	2,1	2,1	0	2
<code>SVC_CRC_CCITT_L()</code>	CCITT	16	LSB first	1,2	2,1	-1	3
<code>SVC_CRC_CCITT_M()</code>	CCITT	16	MSB first	2,1	2,1	-1	4
<code>SVC_CRC_CRC32_L()</code>	CCITT	32	LSB first	4,3,2,1	4,3,2,1	-1	5

CRC Function Calls

This section presents the CRC function calls.

NOTE



Verify the protocol and what is included in the CRC. For instance in a typical ACK/NAK protocol with <STX>packet<ETX><LRC> the <STX> character is NOT included in the LRC but the <ETX> character is included.

SVC_CRC_CALC()

Calculates a CRC value for `size` bytes of data in `buf`. This function provides a common interface to several different CRC algorithms, using `type` to select which one to use. Separate function calls are provided for each algorithm. [Table 24](#) lists the specific function for each type.

See also [Configuration Information Block \(CIB\)](#), [SVC_CRC_CCITT_L\(\)](#), [SVC_CRC_CCITT_M\(\)](#), [SVC_CRC_CRC16_L\(\)](#), [SVC_CRC_CRC16_M\(\)](#), and [SVC_CRC_CALC_L\(\)](#).

Prototype `unsigned int SVC_CRC_CALC (int type, const char *buf, int size);`

SVC_CRC_CALC_L()

Identical to [SVC_CRC_CALC\(\)](#), except that it returns a 32-bit result. 32 bits are required when `type = 5`. See also [SVC_CRC_CALC\(\)](#).

Prototype `unsigned long SVC_CRC_CALC_L (int type, const char *buf, int size);`

SVC_CRC_CCITT_L()

Calculates a 16-bit CRC for `size` bytes of data in `buf` using the CCITT polynomial:

$$x^{16} + x^{12} + x^5 + 1.$$

`SVC_CRC_CALC(3, buf, size)` is equivalent to `SVC_CRC_CCITT16_L(buf, size, -1)`

Prototype `unsigned int SVC_CRC_CCITT_L`
 `(const void *buf, int sz, unsigned int seed);`

SVC_CRC_CCITT_M()

Calculates a 16-bit CRC for `size` bytes of data in `buf` using the CCITT polynomial:

$$x^{16} + x^{12} + x^5 + 1$$

`SVC_CRC_CALC(4, buf, size)` is equivalent to `SVC_CRC_CCITT_M(buf, size, -1)`

Prototype `unsigned int SVC_CRC_CCITT_M`
 `(const void *buf, int sz, unsigned int seed);`

SVC_CRC_CRC16_L()

Calculates a standard CRC16 CRC value for `size` bytes of data in `buf`. CRC16 is based on the polynomial:

$$x^{16} + x^{15} + x^2 + 1$$

The data is processed least-significant bit first. `seed` is a starting value, normally zero. If `seed` is non-zero, CRC1 is in the high (most-significant) byte, and CRC2 is the low byte.

The result CRC value contains CRC1 in the low byte, and CRC2 in the high byte. Note that this is the opposite of the seed, therefore the bytes must be swapped if the result is used as the seed to another call. On a big-endian processor you must to reverse the byte order when appending the CRC to a message (see example below).

`SVC_CRC_CALC(1, buf, size)` is equivalent to `SVC_CRC_CRC16_L(buf, size, 0)`.

See also [SVC_CRC_CRC16_M\(\)](#) and [SVC_CRC_CALC\(\)](#).

Prototype

```
unsigned int SVC_CRC_CRC16_L  
(const void *buf, int sz, unsigned int seed);
```

Example 1

The linked code example calculates the CRC for a packet and appends it to the end.

Example 2

This linked file contains an alternate implementation that works correctly regardless of the processor's byte order.

SVC_CRC_CRC16_M()

Calculates a standard CRC16 CRC value for `size` bytes of data in `buf`. It is the same as [SVC_CRC_CRC16_L\(\)](#), except that bits are processed most-significant bit first. Also, the seed expects CRC1 in the low byte, and CRC2 in the high, so no byte swap is required to use the result of one call as the seed for the next. The linked examples with [SVC_CRC_CRC16_L\(\)](#) also work for this function. See also [SVC_CRC_CRC16_L\(\)](#) and [SVC_CRC_CALC\(\)](#).

`SVC_CRC_CALC(2, buf, size)` is equivalent to `SVC_CRC_CRC16_M(buf, size, 0)`

Prototype

```
unsigned int SVC_CRC_CRC16_M  
(const void *buf, int sz, unsigned int seed);
```

SVC_CRC_CRC32_L()

Calculates a 32-bit CRC32 CRC value for `size` bytes of data in `buf`. CRC32 is based on the polynomial:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Bits are processed least-significant bit first. `seed` is a starting value, with CRC1 in the low (least-significant) byte and CRC4 in the high byte.

The result CRC value is (CRC4, CRC3, CRC2, CRC1) from most-significant to least-significant byte. On a big-endian processor, it may be necessary to reverse the byte order when appending the CRC to a message. See also [SVC_CRC_CALC\(\)](#).

`SVC_CRC_CRC32_L(buf, size, -1)` is equivalent to `SVC_CRC_CALC_L(5, buf, size)`

Prototype

```
unsigned long SVC_CRC_CRC32_L  
(const void *buf, int sz, unsigned long seed);
```

Example

Click the linked example to view the sample code.

Configuration Information Block (CIB)

The Configuration Information Block describes the hardware configuration of a terminal and provides an indicator for things such as memory type, display type, keyboard type, and communication devices present among others. The OS uses this block of data to determine what drivers to load or how certain operations behave.

NOTE



This is a Trident feature and is not used in Predator.

The following CIB fields are modified to accommodate changes in the VX 520 color terminals:

VX 520 DE and VX 520 3G:

- SD socket "A" present (enables SD card driver)
- Display type and size (same as VX 675)
- Keypad type and key map (same as VX 675)

KBD_Style = '3'

VX 520 3G:

·GPRS modem:

GPRS_Type = GPRS_INTERFACE_TYPE_USB_HOST "2"

GPRS_ControllerID = GPRS_CONTROLLER_ID_PHS8P "3"

GPRS_ControllerID = GPRS_CONTROLLER_ID_PHS8P_NOGPS "4"

Dual SIM:

DUAL_SIM = DUAL_SIM_WITH_DETECT "1" <or>

DUAL_SIM = DUAL_SIM_NO_DETECT "2"

·UART

UART_1_type0x30 UART_1_INTERFACE_NONE

SVC_INFO_CIB_ID()

This is a Trident only function. SVC_INFO_CIB_ID returns the part number of the CIB.

Prototype `int SVC_INFO_CIB_ID(char *CIBid11, int bufLen);`

Parameters

CIBid11	Pointer to a char.
bufLen	Buffer length, must be long enough to hold the part number.

Return Values

Success:	0
Failure:	-1: EINVAL

SVC_INFO_CIB_VER()

The CIB is a Trident only feature. The CIB version is returned by this call.

Prototype `int SVC_INFO_CIB_VER(char *CIBver5, int bufLen);`

Parameters

CIBver5	Pointer to buffer that receives the version.
bufLen	Buffer length, must be at least the size of the version.

Return Values

Success:	0
Failure:	-1 with errno set to EINVAL.

SVC_INFO_DEV()

If the `INFO_PRES_XXXX` bit value is passed as an argument, `SVC_INFO_DEV` returns the COM device for that device.

Example If `INFO_PRES_MODEM` is passed, then `/DEV/COM3` is returned.

Note that for `SVC_INFO_DEV` and `SVC_INFO_DEV_TYPE`, the `INFO_PRES_XXX` value should be a single device and not the return of `SVC_INFO_PRESENT`. `SVC_INFO_PRESENT` returns a bit map of all devices. Where, `SVC_INFO_DEV` and `SVC_INFO_DEV_TYPE` require a single device request.

On VX 820, if `INFO_PRES_PRINTER` is passed, then `/DEV/COM4` is returned.

On VX 680 3G, this function call returns `/dev/pcm` in `devname` when `INFO_PRES_SPEAKER` is passed.

Prototype

```
int SVC_INFO_DEV(int type, char *device);  
int SVC_INFO_DEV(INFO_PRES_SPEAKER, devname);
```

Parameters

<code>type</code>	The device type.
<code>device</code>	The COM device.

Return Values Returns the COM device for the device type.

SVC_INFO_DEV_TYPE()

If the `INFO_PRES_xxxx` bit value is passed as an argument, `SVC_INFO_DEV_TYPE` returns the device type if that device is present or -1 if the device is not present.

On VX 680 3G, this function call returns `MID_SPEAKER` when `INFO_PRES_SPEAKER` is passed.

Example If `INFO_PRES_MODEM` is passed, the function returns `MID_HARLEY_MODEM` or `MID_USB_MODEM` if a modem is present. On VX 675, when `INFO_PRES_GPRS` is passed, this returns `MID_BGS2`.

Prototype

```
int SVC_INFO_DEV_TYPE(int type);
int SVC_INFO_DEV_TYPE(INFO_PRES_SPEAKER);
```

Parameters

type	The device type.
------	------------------

Return Values

Success	Device type, if device is present.
-1	If device is not present.

SVC_INFO_DUAL_SIM()

The application can call this function to get DUALSIM field in CIB.

Prototype `int SVC_INFO_DUAL_SIM(void);`

Return Values

Success	0x31	Dual SIM present, no SIM2 detection.
	0x32	Dual SIM present with both detection.
	0xFF or 0x0	No dual SIM.

Example

```
// Switch from SIM1 to SIM2 then back to SIM1
int SIM_type;
SIM_type = SVC_INFO_DUAL_SIM();
```

SVC_INFO_PRESENT()

This returns a bit map of devices present in a terminal. See [Definitions in SVC.h](#) for the defines for the bit map (INFO_PRES_XXXX). If the bit is 1 then that device is present.

On VX 680 3G, this function call returns the bit INFO_PRES_SPEAKER set if the speaker hardware is present. For terminals that has a removable USB modem or internal modem on the charging base (such as VX 675, VX 680, VX 805 or VX 820) call returns the bit INFO_PRES_MODEM set if the removable usb modem has been connected to a terminal or terminal is placed on its charging base.

Prototype `int SVC_INFO_PRESENT(void);`

Return Values

1	Bitmap of device is present.
---	------------------------------

SVC_INFO_USB_BITS()

If the `INFO_PRES_xxxx` bit value is passed as an argument, `SVC_INFO_USB_BITS` returns the device type if that device is present or -1 if the device is not present.

Prototype `int SVC_INFO_USB_BITS(int type);`


Parameters

<code>type</code>	The device type.
-------------------	------------------

Return Values

Success	Device type, if device is present.
-1	If device is not present.

NOTE



On VX 820 DUET, when `INFO_PRES_PRINTER` is passed, this returns `UDB_COM4`.

Definitions in SVC.h

Shown are the SVC.h definitions:

#define MID_COM2_UART	(42)	// COM2 is configured as 2 wire UART
#define MID_USB_MODEM	(50)	// USB modem
#define MID_BTEZ1	(60)	// BT Ezurio brand module 1
#define MID_BTEZ2	(61)	// BT Ezurio brand module 2
#define MID_BTEZ3	(62)	// BT Ezurio brand module 3
#define MID_BTEZ4	(63)	// BT Ezurio brand module 4
#define MID_BTAA1	(64)	// BT alternate vendor module 1
#define MID_BTAA2	(65)	// BT alternate vendor module 2
#define MID_BTAA3	(66)	// BT alternate vendor module 3
#define MID_BTAA4	(67)	// BT alternate vendor module 4
#define MID_M200	(70)	// Kyocera M200 CDMA
#define MID_MC55i_ONLY	(72)	// Sierra MC55i GPRS
#define MID_MC5727	(73)	// Sierra MC5727 CDMA
#define MID_SOC_ETH	(74)	// Internal Ethernet
#define MID_USB_HOST_PWR	(75)	// Powered USB Host
#define MID_USB_HOST_NO_PWR	(76)	// USB Host not powered
#define MID_USB_HOST_HUB	(77)	// USB with internal hub
#define MID_USB_DEV	(78)	// USB device
#define MID_CTL5	(79)	// Contactless for the VX 680
#define MID_SD_A	(80)	// SD Slot A
#define MID_SD_B	(81)	// SD Slot B
#define MID_TOUCH_RES	(82)	// touchscreen type - resistive
#define MID_TOUCH_CAP	(83)	// touchscreen type - capacitive
#define MID_TBD	(99)	// To Be Determined
#define MID_PHS8P	(89)	//Cinterion PHS8-P radio with GPS
#define MID_PHS8P_NOGPS	(90)	//Cinterion PHS8-P radio without GPS
#define INFO_PRE5_MODEM	0x01	//Internal Landline modem presence
#define INFO_PRE5_GPRS	0x02	// GPRS modem presence
#define INFO_PRE5_CDMA	0x04	// CDMA modem presence
#define INFO_PRE5_WIFI	0x08	// WIFI presence
#define INFO_PRE5_BT	0x10	// Bluetooth presence
#define INFO_PRE5_SOC_ETH	0x20	// Internal Ethernet presence

#define INFO_PRESENCE_USB_HOST	0x40	// USB Host port presence
#define INFO_PRESENCE_USB_DEV	0x80	// USB Device port presence
#define INFO_PRESENCE_CTLN	0x100	// Contactless Device presence
#define INFO_PRESENCE_SD_A	0x200	// SD Device presence
#define INFO_PRESENCE_SD_B	0x400	// SD Device presence
#define INFO_PRESENCE_TOUCH	0x800	// Touchscreen presence
#define INFO_PRESENCE_PRIMARY	0x1000	// Primary smartcard presence
#define INFO_PRESENCE_MSAM	0x2000	// MSAM slots presence
#define INFO_PRESENCE_BARCODE	0x4000	// Internal BarCode Reader presence
#define INFO_PRESENCE_DSPLITE	0x8000	// Display backlight presence
#define INFO_PRESENCE_KBDLITE	0x10000	// Keypad backlight presence
#define INFO_PRESENCE_TELEPHONE	0x20000	// Vx525 TelePOS



System Devices

This chapter describes the Application Programming Interface (API) for the following system devices:

- Magnetic Card Reader
- Smart Card Reader
- Contactless Reader
- Real-Time Clock/Calendar
- Beeper
- Speaker Audio System
- Internal Printer
- SDIO
- Biometric Module (VX 520 GPRS)
- USB Barcode Scanner
- USB Keyboard
- Metrologic Barcode Scanner
- USB to RS-232 Converter
- MC5727 CDMA Radio

System devices are accessed in the same way as files, by using the same basic set of function calls: `open()`, `read()`, `write()`, and `close()`. Like files, system devices are specified by name, prefixed with `DEV_`. For example, to open the magnetic card reader device, use the device name `DEV_CARD`. Like filenames, device names are not case-sensitive.

Basic function calls always return an error code of `-1` when an error condition is encountered and set `errno` to a specific error code.

Terminal standard device names defined in the `<svc.h>` file are listed in [Table 25](#).

Normally, all devices must be opened explicitly to be used. Two exceptions are the beeper and the clock. For example, normally tasks use `normal_tone()`, `error_tone()`, and `read_clock()` without opening the associated device.

For convenience, the system library defines a set of global variables containing the device names. Use these variables in place of previous `/dev` names. The device names and corresponding handles names are shown in [Table 25](#).

Table 25 Verix eVo Device Handles

Device	<code>/dev</code> Name	<code><svc.h></code> Variable
Magnetic card reader	<code>/dev/mag</code>	<code>DEV_CARD</code>
Bar code reader	<code>/dev/bar</code>	<code>DEV_BAR</code>
Real-time clock	<code>/dev/clock</code>	<code>DEV_CLOCK</code>
Beeper	<code>/dev/stderr</code>	<code>DEV_BEEPER</code>
Console (keypad and display)	<code>/dev/console</code>	<code>DEV_CONSOLE</code>
USB fingerprint reader device	<code>/dev/bio</code>	<code>DEV_BIO</code>
COM port 1	<code>/dev/com1</code>	<code>DEV_COM1</code>
COM port 2	<code>/dev/com2</code>	<code>DEV_COM2</code>
COM port 3	<code>/dev/com3</code>	<code>DEV_COM3</code>
COM port 4	<code>/dev/com4</code>	<code>DEV_COM4</code>
COM port 5	<code>/dev/com5</code>	<code>DEV_COM5</code>
COM port 6	<code>/* com 6 */</code>	<code>DEV_COM6</code>
COM port 8	<code>/* com 8 */</code>	<code>DEV_COM8</code>
COM port 9	<code>/* com 9 */</code>	<code>DEV_COM9</code>
COM port 10	<code>/* com 10 */</code>	<code>DEV_COM10</code>
Mag card	<code>/* mag card */</code>	<code>DEV_CARD</code>
Barcode reader	<code>/* bar code reader */</code>	<code>DEV_BAR</code>
CTLS	<code>/* Contactless device */</code>	<code>DEV_CTL5</code>
USB keyboard	<code>/* USB Keyboard HID converted to make and break code*/</code>	<code>DEV_KYBD</code>
USB host	<code>/* PP1000SE and VX 820 device */</code>	<code>DEV_USBSER</code>
Semtek device driver	<code>/* Semtek device driver */</code>	<code>DEV_SEMTEK</code>
Customer smart card	<code>/dev/icc1</code>	<code>DEV_ICC1</code>
Merchant SAM	<code>/dev/icc2</code>	<code>DEV_ICC2</code>
Merchant SAM	<code>/dev/icc3</code>	<code>DEV_ICC3</code>
Merchant SAM	<code>/dev/icc4</code>	<code>DEV_ICC4</code>
Merchant SAM	<code>/dev/icc5</code>	<code>DEV_ICC5</code>
Merchant SAM	<code>/dev/icc6</code>	<code>DEV_ICC6</code>

Device Management Function Calls

Each device is associated with a fixed handle ([Table 25](#)). For example the console handle is always 1. However Verix eVo OS reserves the right to change the handle assignments, so applications should use the handles returned by `open()`.

Successfully opening a device gives the calling task exclusive use of it (the task is said to *own* the device). Tasks can read the real-time clock ([read_clock Example](#)) and sound the beeper (`sound`) without opening the device. However, if a task does open a device, other tasks cannot access that device.

This section lists the function calls used to manage system devices.

NOTE



Using the fixed handle numbers does not provide an advantage as the device must still be opened to be used and the OS enforces device ownership by task.

get_name()

Retrieves the device name associated with `dvic_nbr`. A zero-terminated string is stored in the buffer pointed to by `dev_id`. The caller must supply a 20-byte buffer, regardless of the length of the expected result.

The device does not need to be open. If the handle is in the device handle range (from 0 – 255) but is not assigned to a device, an empty string returns. This is not considered an error.

See also, [get_owner\(\)](#).

Prototype

```
int get_name (int dvic_nbr, char *dev_id);
```

Return Values

Success: 0: Returns the name of the device used in `open()`.

Failure: A negated `errno` value. `errno` is not set. The error conditions are as follows:

EBADF: handle not in device handle range (0 – 255).

EACCESS: Invalid buffer pointer.

Example

The linked code example displays the device name for each handle (or as many as will fit on screen).

get_owner()

Retrieves owning task and handle for a device or pipe. `get_owner()` reports the task that currently owns a device or named pipe. A common use of `get_owner()` is for setting up pipe connections. For more information, refer to [Pipes](#).

NOTE



`get_owner()` does not work for files (these can be opened by multiple tasks simultaneously).

See also, [get_name\(\)](#) and [set_owner\(\)](#).

Prototype

```
int get_owner (char *device, int *task);
```

Parameters

`device` The name used in a call to `open()` for example, `/DEV/COM1` or `P:MYPIPE`. Names are not case-sensitive.

`*task` Set to the task ID of the task that owns it. Usually the task that opened it, although a task can also obtain ownership through [set_owner\(\)](#). If the device or pipe is not open, `*task` is set to 0.

Return Values The return value is the device handle. This allows a caller to determine the handle associated with a device without opening it. Normally devices must be open to use a handle, but there are exceptions, such as [get_port_status\(\)](#).

Failure: -1 and errno set to ENODEV: Invalid device or pipe name.

-1 and errno set to EACCESS: Invalid argument pointer.

Example The linked code example returns the owner of the console.

set_owner()

Transfers ownership of an open device to another task. Following this call, the caller cannot access the device. No changes to the device state are made and buffers are not cleared. (The caller may wish to do this before transferring control.) Pending events for the device are not transferred to the new task.

NOTE



Do not use `set_owner()` to transfer console ownership. Instead use `activate_task()`.

See also, [get_owner\(\)](#), [activate_task\(\)](#).

Prototype

```
int set_owner (int hdl, int taskid);
```

Return Values

Success: 0

Failure: A negated errno value. errno is not set. Error conditions are:

EBADF: Invalid handle or caller does not own device.

EINVAL: Invalid task number.

Example

The linked code example considers an application to monitor a serial port for service requests. When one is received, it starts a new task to handle it and transfers control of the port to it.

set_owner_all()

Upon successful execution, all of the owner's siblings (threads sharing the same address space) will be allowed to use the device for `read`, `write`, `status`, `control`, or `close`. Only the actual owner receives events. If the owner thread transfers ownership using the existing [set_owner\(\)](#) call to one of its sibling threads, then the transferee receives events for the device. If the device is closed, or if ownership is transferred to a non-sibling, the shared ownership ceases. Normal ownership (exclusive to one thread) resumes when the owner calls either [set_owner\(\)](#) or `open()`.

Prototype

```
int set_owner_all(int hdl);
```

Parameters

hdl Device handle

Return Values

Success: 0

Failure: -1: EINVAL

Magnetic Card Reader

The Verix eVo-based terminal's magnetic-stripe card reader can read data simultaneously from tracks 1, 2, and 3 of ISO 7811, AAMVA, and California DL/ID cards. This is in accordance with the *Visa® Second Generation "Full Service" Manufacturer's Specification Manual*. Trident terminals support a triple track magnetic card reader. Predator terminals generally support 2 tracks but some models have triple track readers. The Mag Card device information is also directly available to the application through the `SVC_INFO_MAG()` system call.

Hybrid Card Reader

On some terminals a hybrid card reader that reads mag card/smart card is attached to the unit via the COM2 serial interface.

No Data Characters on Track 3 MSR

Track 3 of the magnetic stripe card is used to store the enciphered PIN, country code, currency units, amount authorized, subsidiary account information, and other account restrictions.

When present, data characters should be stored on track 3 in compliance with ISO 7811 standards. The Track 3 encoding is as follows:

```
<Start Sentinel><data>...<data><End Sentinel><LRC>
```

Normally, the OS device driver discards the Start/End sentinels and LRC that bracket the data characters, such that only the data field is returned. If the data field is empty, the driver returns a count byte of 2 (1 count byte, 1 status byte, 0 characters) and a track decode status of 0 (VALID): { 2, 0 }. `card_mode(0)` forces the driver to return the Start/End sentinels and LRC—the driver returns a count byte of 5 (1 count byte, 1 status byte, 3 characters), a decode status of 0 (VALID), and the framing characters: { 5, 0, ' ', '?' , '4' }.

A track that lacks any bits or contains only clocking ('0') bits holds no information. The OS device driver returns a count byte of 2 (1 count byte, 1 status byte, 0 characters) and a track decode status of 1 (NO DATA): { 2, 1 }. In this context, "NO DATA" means "error, no information on track" and not "valid, but no characters in the data field."

Increased Buffer Size

The buffer sizes of the magnetic stripe reader (MSR) raw data for channels 1/2/3 are increased to 98/98/98 bytes for Trident from Predator's 92/32/92 bytes. The buffer allocation in the VX 680 terminal and allows the other terminals to read cards with track 2 data encoded at 210 BPI, similar to tracks 1 and 3. The extra 457 bits are clock bits that appear before and/or after the data on track 2.

Testing the MSR

Use this [Example](#) to test the magnetic strip reader by swiping a card. The number of data bytes read per track and the error state for each track is then displayed.

Magnetic Card Reader Function Calls

This section presents general functions for the magnetic card driver API.

You may also refer to the following APIs:

- `open()`
- `read()`

- `write()`
- `close()`

card_magprint_count()

A special purpose function for a MAGNEPRINT enabled terminal only. This function returns the number of interrupts since the last RESTART.

Prototype `int card_magprint_count(void);`

Return Values

Success: Number of interrupts.

Failure: -1: EINVAL

card_magprint_data()

A special purpose function for a MAGNEPRINT enabled terminal only. This function copies the magnetic card reader buffer to the buffer.

Prototype `int card_magprint_data(char *buf);`

Parameters

buf Buffer to receive data.

Return Values

Success: Size of the data returned.

Failure: -1: EINVAL

card_magprint_stat()

Returns the status of the magnetic card reader. A special purpose function for a MAGNEPRINT enabled terminal only.

Prototype `int card_magprint_stat(char *buf);`

Parameters

<code>buf</code>	Buffer that contains the returned status.
------------------	---

Return Values

Success:	0
Failure:	-1: EINVAL

card_mode()

Sets the magnetic card reader mode flags.

Prototype `int card_mode(int mode);`

Parameters

<code>mode</code>	Mask for the flags to set. See magnetic card reader specification for definition.
-------------------	---

Return Values

Success:	0
Failure:	-1

card_pending()

Determines if there is unread data in the card reader buffer.

Prototype `int card_pending(void);`

Return Values

Success: 0 if no card swipe is available for reading; 1 if a card swipe is available.

Failure: -1 and errno set to `EBADF: DEV_CARD` not open.

card_raw_data()

Returns the unformatted raw track data for the magnetic card reader.

Prototype `int card_raw_data(char *buf);`

Parameters

buf Buffer that receives the data.

Return Values

Success: 0

Failure: -1 with errno set to `EACCES`.

magprt_mode_control()

A special form of `card_mode` that sets the `magprt` flag for the magnetic card reader mode flags.

Prototype `int magprt_mode_control(int mode);`

Parameters

<code>mode</code>	Turns the <code>magprt</code> flag on or off.
-------------------	---

Return Values

Success:	0
Failure:	-1: <code>EINVAL</code>

VeriShield Protect (VSP)

VeriShield Protect (VSP) is a program that aims to secure the MSR card data from the moment it is presented to the terminal application to the time it is unscrambled by the decryption appliance at the host site.

No unencrypted MSR card data is seen in the communication channels between the terminal and the decryption appliance. Therefore, no PAN and Discretionary Data can be collected for unauthorized uses.

VSP Encryption

The software-based card data encryption, H-TDES Lite, is integrated in Verix eVo OS for the initial phase of the VSP program. Table 26 shows the MSR data in its original form and after encryption.

NOTE



The encrypted form retains the general format of the cleartext form, making it possible for an existing certified application to utilize VSP, but need not be modified to be “VSP-aware” and recertified.

The information is separated into the individual track data records for convenience. Only the bytes in the Count, Stat, and Data fields are present. The Count byte counts the number of data characters, plus itself and the Stat byte. The Stat byte is 0 for No Decode Error, 1 for No Track Data, and something else for a Decode Error. The Data bytes are shown in ASCII and the double quotes are not part of the data; the field is empty if the Stat byte is non-zero.

Table 26 MSR Card Data

Cleartext MSR Card Data			
Track	Count	Stat	Data
1	68	0	"%B4150000000000000^TESTCARD/VISA^1112101000004567000000123000000?K"
2	42	0	";4150000000000000=11121011234567000000?:"
3	2	1	
Encrypted MSR Data			
Track	Count	Stat	Data
1	70	0	"%B4150001882680000^TESTCARD/ VISA_____000000^43121019494664794988?V"
2	42	0	";4150001882680000=43121019494664794988?7"
3	2	1	

In comparing the track data in cleartext and encrypted forms, many—but not all—bytes are changed: middle digits of the PAN, last characters of the Name field, expiration date, discretionary data, and LRC.

- For the PAN, the first 6 digits (BIN value) and the last 4 digits are preserved. The remaining digits are altered, but one of those digits is chosen such that the LUHN checkdigit test will pass.

Cleartext: "4150000000000000" $(0+0+0+0+0+0) \bmod 10=0$

Encrypted: "4150001882680000" $(2+8+7+2+3+8) \bmod 10=0$

- On track 1, the Cardholder Name field is padded with spaces until it is 26 characters long, and the last 6 characters are set to a base-40 ID code. In this example, it is 000000.

Cleartext: "TESTCARD/VISA"

Encrypted: "TESTCARD/VISA_____000000"

- The Expiration Date (the 4 digits after Field Separator) is advanced by 32 years.

Cleartext: "1112"

Encrypted: "4312"

- The track 2 Discretionary Data field is encrypted, excluding the Service/Restriction Code (the 3 digits after Expiration Date) of 101. The result is used for both tracks 1 and 2.

Cleartext: "1011234567000000"

Encrypted: "1019494664794988"

- The number of characters in tracks 1 and 2 do not exceed ISO limits of 79 and 40, respectively.
- The LRCs are changed such that LRC checks will pass.

Verix eVo Implementation

The Verix eVo implementation requires two main pieces of software—the VSP driver and the set of special OS support features. The VSP driver handles a limited set of commands and requests related to the MSR encryption technology. It behaves like a device driver, but it is a special application that runs in an upper system GID. This application is supported by special OS services to access system resources not available to other applications.

VSP Driver

This driver consists of four files, the executable code (#SEMTEK.OUT), its data/configuration file (#SEMINIT.DAT), and their respective signatures (#SEMTEK.OUT.P7S, #SEMINIT.DAT.P7S). They are downloaded into an upper system GID, which is available in the VSP-capable OS. The VSP driver/application is launched when the MSR device driver is opened for the first time after system restart.

NOTE



The upper system GID must be enabled beforehand. This is done only once via a pair of downloads—the first download loads the Certificate Tree files (VXPAROSR.CRT, VMSPART.CRT, VMSSIGN1.CRT, VMSSIGN2.CRT), and the second download loads the Certificate files (SPONSORCERT.CRT, CERTIF.CRT, VXOWNROS.CRT, VXSIGNOS.CRT).

VSP must be coordinated with the transaction processor and requires configuration and keys to be useful. It is recommended that you discuss this with your VeriFone representative to receive the necessary files for download.

User APIs

The terminal application has a set of APIs to access the commands/requests supported by the VSP driver. Each API generates inline code, which executes a specific control function of the MSR device driver. The MSR device driver should remain open while the APIs are being used by the terminal application. If there was no error, the return code is 0. Otherwise, the return code is -1, with errno set to a specific error code.

NOTE



Error codes and state information within the VSP driver are returned in the status and extended status requests, and do not affect the terminal API error codes.

If the OS accepts the command/request, the application must poll for the response because the APIs are non-blocking—i.e., the OS issues a message to the VSP driver, which responds after it wakes up and processes the message. The API used to retrieve the response is `int VSP_Result(void)`. The return code is zero, if the response is still pending. Negative, if an error is found in the processing of the command/request; errno is set accordingly. The return code is positive, if there is no error; the value is the number of bytes in the response to the command/request.

Return Code	Error Condition
0	Response still pending.
< 0	Error—errno set accordingly.
> 0	Number of bytes in response.

Example:

The code fragment below illustrates how an application can get the VSP driver's extended status:

```
int rc; char buf[1024];

if ((rc = VSP_Status_Ex(buf)) >= 0)
    while (!(rc = VSP_Result()))
        SVC_WAIT(0); // allow other tasks to run
if (rc < 0)
    printf("Error: %d, errno = %d\n", rc, errno);
```

```

else
    printf("OK:  %d\n%s\n", rc, buf);

```

User APIs include:

- VSP_Status()
- VSP_Status_Ex()
- VSP_Crypto()
- VSP_Disable()
- VSP_Enable()
- VSP_Encrypt_MSR()
- VSP_Encrypt_KBD()
- VSP_Decrypt_PAN()
- VSP_Decrypt_MSR()
- VSP_Init()
- VSP_Passthru()
- VSP_Reset()
- VSP_Result()
- VSP_Xeq()

Data Formats

There are five data types passed via the parameter lists of the APIs:

- char st[8 + 9+1];
- char MSRC[248], MSRE[248];
- char PANc[19+1], PANe[19+1];
- char EXPc[4+1], EXPe[4+1];
- char xst[1023+1];

The status of the VSP driver is returned in array st. The first 8 bytes hold status codes for 8 things such as internal VSP keys and overall state. The last 10 bytes contain a null-terminated string for the driver version code, having a format of X.YY.ZZZZ. The output buffer must accommodate the 18-byte result. Below is the status for a partially initialized driver:

```

char st[8 + 9+1] = {
    0x00, 0x00, 0x00, 0x02, 0x02, 0x02, 0x02, 0x00,
    '2', '.', '2', '.', '0', '0', '2', '0', 0, 0
};

```


MSR data images are passed via arrays of characters that are formatted in the manner shown in the sample encryption. The cleartext and encrypted data are passed via MSRC and MSRE, respectively. The output buffer must accommodate the longest result, which is 248 bytes. The shortest data image is 6 bytes.

Example Click example to see sample encryption.

PAN data are null-terminated ASCII strings of digits. Based on ANS X9.24, the shortest PAN is 8 digits, and the longest is 19. The LUHN checkdigit is not verified by the terminal API. The cleartext and encrypted PANs are passed via PANc and PANe, respectively. The output buffer must accommodate the longest result, which is 20 bytes.

```
char PANc[19+1] = "4150000000000000";
char PANe[19+1] = "4150001882680000";
```

Expiration dates are null-terminated ASCII strings of digits consisting of a 2-digit year followed by a 2-digit month. The date is not validated by the terminal API. The cleartext and encrypted expiration dates are passed via EXPc and EXPe, respectively. The output buffer must accommodate the result, which is 5 bytes.

```
char EXPc[4+1] = "1112";
char EXPe[4+1] = "4312";
```

The extended status of the VSP driver is returned in array xst. It holds an ASCII null-terminated string. The string may contain new line characters, '\n'. The output buffer must accommodate the longest result, which is 1024 bytes. Below is the extended status for a partially initialized driver.

```
char xst[1023+1] = "No errors, state:0, encryption: Disabled";
```

Internal OS Calls

The most straightforward use of the encrypted MSR data is as a replacement for the cleartext MSR data passed to the host. The original, unencrypted MSR data is used by the application in all places where the various data fields are input to computations such as in PIN encryption blocks and MAC calculations, among others. Note that this requires the application to be updated as it must distinguish between cleartext and encrypted MSR data, and process them accordingly.

To minimize the need to modify existing applications to be “VSP-aware” and to recertify them, the OS automatically calls the internal versions of `VSP_Encrypt_MSR()` and `VSP_Decrypt_PAN()` under specific conditions.

- When the MSR device driver collects data from a card swipe, the cleartext MSR image is encrypted via `VSP_Encrypt_MSR()`. Only the encrypted results are passed to the application via `read()`. Thus, the application unknowingly is dealing with encrypted MSR data.
- When the VSS or IPP code performs a Master/Session Key or DUKPT PIN block encryption, or DUKPT MAC calculation, the encrypted PAN is decrypted via `VSP_Decrypt_PAN()` prior to the computation. Thus, the OS attempts to

replace the encrypted PAN with the cleartext PAN so that the application will get the correct results.

In cases where the OS cannot determine whether or not to adjust the application data automatically, the application will be required to access the cleartext MSR data explicitly, and to process it accordingly. For example, the Master/Session Key MAC calculation accepts any kind of data as its input. If an application includes the MSR data in the MAC, then it must be updated to explicitly fetch the cleartext MSR data and pass the relevant parts into the MAC computation.

VTM Menu

The VTM MAG CARD DIAG test screens list the VTM decode status message for each of the three tracks. The three status messages are five characters long:

- “VALID”
- “EMPTY”
- “ERROR”

Below are sample screens from an eight-line display:

```
VERIX TERMINAL MGR

TRK 1:EMPTY
TRK 2:EMPTY
TRK 3:EMPTY
```

This screen appears when the VSP-capable OS does not have a fully installed VSP driver. If the VSP driver is running, the function keys are labeled accordingly.

```
VERIX TERMINAL MGR
TRK 1:EMPTY
TRK 2:EMPTY
TRK 3:EMPTY
1> STAT
2> STATX
```

The function keys are mapped as follows:

KEY	FUNCTION
1	int VSP_Status (char *st);
2	int VSP_Status_Ex (char *xst);

If one of the first three function keys is pressed or if a card is swiped, then the VSP status is displayed as shown on the screen below:

```
VERIX TERMINAL MGR
TRK 1:EMPTY
TRK 2:EMPTY
TRK 3:EMPTY
1> STAT      2.2.0020
2> STATX
```

Note that an early OS version may display the hex status bytes on the same line. If the last function key is pressed, then the VSP extended status is sent to the primary serial port, which is COM2 for PIN pads (VX 820) and COM1 for all others. The serial format is async, 8N1, 115.2 KBPS.

VSP_Status()

Returns VSP status in `st[]`.

Prototype `int VSP_Status(char *st);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Crypto()

Internal Use Only. DO NOT USE.

Prototype `int VSP_Crypto();`

VSP_Disable()

Disables VSP encryption, and returns VSP status in `st[]`.

Prototype `int VSP_Disable(char *st);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

 ENODEV - VSP driver not loaded/not running.

 EACCESS - Argument access error.

 EINVAL - Invalid argument.

 EBUSY - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Enable()

Enables VSP encryption, and returns VSP status in `st[]`. Encryption is not enabled if the driver is not fully initialized and properly activated.

Prototype `int VSP_Enable(char *st);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

 ENODEV - VSP driver not loaded/not running.

 EACCESS - Argument access error.

 EINVAL - Invalid argument.

 EBUSY - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Encrypt_MSR()

Encrypts the cleartext MSR image in `MSRc[]`, and returns the results in `MSRe[]`. The VSP driver encrypts the cleartext data only for cards that meet a rigid set of criteria; otherwise, it returns the cleartext data as its output. This API can be used to encrypt MSR data from a smartcard, including a contactless card.

NOTE



The MSR device must be open even if the data source is not the MSR.

Prototype `int VSP_Encrypt_MSR(char *MSRc, char *MSRe);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Encrypt_KBD()

Encrypts the cleartext PAN in `PANc[]` and expiration date in `EXPc[]`, and returns the results in `PANe[]` and `EXPe[]`, respectively. The VSP driver encrypts the cleartext data only for cards that meet a rigid set of criteria; otherwise, it returns the cleartext data as its outputs. The final return code from `VSP_Result()` is the sum of the string lengths `PANe` and `EXPe`.

NOTE



The MSR device must be open even if the data source is not the MSR.

Prototype `int VSP_Encrypt_KBD(char *PANc, char *EXPc, char *PANe, char *EXPe);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Decrypt_PAN()

Decrypts the encrypted PAN in `PANe[]`, and returns the results in `PANc[]`. The VSP driver does not actually decrypt the PAN. It returns the last cleartext PAN recorded, if `PANe` matches the last encrypted PAN result it had saved. Otherwise, it returns the input data as its output.

Prototype `int VSP_Decrypt_PAN(char *PANe, char *PANc);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Decrypt_MSR()

Decrypts the encrypted MSR image `MSRe[]`, and returns the results in `MSRc[]`. The VSP driver does not actually decrypt the MSR image. It returns the last cleartext MSR image recorded, if `MSRe` matches the last encrypted MSR image result it had saved. Otherwise, it returns the input data as its output.

Prototype `int VSP_Decrypt_MSR(char *MSRe, char *MSRc);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Init()

Internal Use Only, DO NOT USE.

Prototype `int VSP_Init();`

VSP_Passthru()

Allows a raw form of the VSP commands to be used. It is not intended for general application use. The commands are not included in this document.

Prototype `int VSP_Passthru(char *in, int iLen, char *out, int oLe);`

Parameters

<code>in</code>	Input buffer containing the command to be executed.
<code>iLen</code>	Input command length
<code>out</code>	Buffer that accepts the command result.
<code>oLe</code>	Response length, usually includes the response byte, the response string, and the NULL terminator.

Return Values

Success:	0
Failure:	-1

VSP_Reset()

Internal Use Only, DO NOT USE.

Prototype `int VSP_Reset();`

VSP_Result()

Used to determine if a previously executed VSP function is completed or still running.

Prototype `int VSP_Result(void);`

Return Values

Success: > 0 - The number of bytes in the result
 0 - The process is still pending
Failure: -1

VSP_Status_Ex()

Returns VSP extended status in `xst []`.

Prototype `int VSP_Status_Ex(char *xst);`

Return Values

Success: 0

Failure: -1 with `errno` set to a specific error code:

`ENODEV` - VSP driver not loaded/not running.

`EACCESS` - Argument access error.

`EINVAL` - Invalid argument.

`EBUSY` - Device is in use (cannot encrypt new data if MSR has pending input data).

VSP_Xeq()

Internal Use Only, DO NOT USE.

Prototype `int VSP_Xeq();`

Smart Card Reader

The smart card reader can communicate with both EMV (Europay®, MasterCard®, Visa®) 4.0 and ISO/IES 7816 compliant asynchronous cards capable of 1.8V, 3V, or 5V operation.

NOTE



Developers can write applications for supported synchronous cards. Contact your VeriFone representative for specific synch card support.

Trident terminals supporting this comply with the smart card specifications requirement. One customer card (PSCR) and up to three SAM cards can be present. The OS detects the physical smart card configuration and operates normally in versions with or without smart card hardware. It also restricts the number of smart cards and SAMs that can be powered up at any one time. No more than two smart card interfaces (a smart card is defined as either a SAM or a PSCR) can be powered simultaneously.

The VTM diagnostic menus do not change for variants with no smart card, the menu items simply return an error if the user attempts to run them.

The smart card interface is provided by the VeriFone CardSlot Library. See the documentation with the CardSlot library for smart card support. Contact your VeriFone representative for more information.

NOTE



Terminals have various numbers of MSAMs or even the PSCR. The user should check the smartcard HW configuration before using. The smartcard driver will automatically configure itself to the number of PSCR and MSAMs.

ICC Socket Locations

ICC socket locations are on the back of the terminals, and may vary for each terminal.

Smart Card API

Smart card support for Verix eVo operating system applications is implemented by a combination of an operating system driver and a library, `LIBVOY.A`. Applications should access it only through the library interface, as defined in the `LIBVOY.H` header file.

The library interface is based on the *Interoperability Specification for ICCs and Personal Computer Systems*, usually referred to as the PC/SC standard. The PC/SC specifications are available on the World Wide Web at:

<http://www.pcscworkgroup.com>

Verix eVo operating system supports only the low-level interface device (IFD) specification described in part 3. High-level resource and application management function calls are left to the Verix eVo operating system application layer.

PCI PED Requirement

The following EPP functions have been deleted from Verix eVo terminals in conformity to the PCI PED certification. If any of these functions are called, an error will be returned, the return values for each function call are provided below.

API	Return Values
<code>decrypt_session_data()</code>	-1 with <code>errno</code> set to any of the following: <ul style="list-style-type: none"> • <code>ENOSYS</code> • <code>EACCESS</code>: Invalid key pointer. • <code>ENOENT</code>: No master key loaded. • <code>EPERM</code>: Session key not set or set by a different task.
<code>gen_master_key()</code>	-1 with <code>errno</code> set to any of the following: <ul style="list-style-type: none"> • <code>ENOSYS</code> • <code>EACCESS</code>: Invalid key pointer. • <code>EPERM</code>: Calling task is not Group 1.
<code>gen_session_key()</code>	-1 with <code>errno</code> set to any of the following: <ul style="list-style-type: none"> • <code>ENOSYS</code> • <code>EACCESS</code>: Invalid key pointer. • <code>ENOENT</code>: No master key loaded.
<code>test_master_key()</code>	-1 with <code>errno</code> set to <code>ENOSYS</code> .

Administrative Services

This section describes smart card administrative services and their syntax. The following services are discussed:

Retrieve IFD Capability

- `IFD_Get_Capabilities()`

Set IFD Capability

- `IFD_Set_Capabilities()`

Protocol Information and Negotiation

- `IFD_Set_Protocol_Parameters()`

ICC Power Management

- `IFD_Power_ICC()`

Syntax

Each service is described using notation similar to the following:

```
RESPONSECODE Name_Of_Service( IN DWORD param1, IN/OUTBYTE[] param2, OUT
WORD param3);
```

In this notation the following type alias are used:

BYTE	unsigned char
WORD	unsigned short
DWORD	unsigned long
RESPONSECODE	unsigned long (as a return value)

Each parameter is specified as either incoming (IN means to the card), outgoing (OUT means from the card), or both (IN/OUT).

IFD_Get_Capabilities()

This function instructs the smart card API to retrieve the value corresponding to the specified `Tag` parameter. This enables the calling application to retrieve any of the information described from the following TLV (tag-length-value) structures:

- Reader capabilities (see example code in [Set/Get Capabilities, page 410](#))
- ICC interface state (see example code in [Get Capabilities, page 410](#))
- Protocol parameters (see example code in [Get Capabilities, page 410](#))
- Specific smart card API features (see [Specific Features for the Smart Card API, page 405](#))

Prototype `RESPONSECODE IFD_Get_Capabilities(INDWORD Tag,OUT BYTE[]Value);`

Return Values `RESPONSECODE` can be one of the following:

<code>IFD_Success</code>	Value successfully retrieved.
<code>IFD_Error_Tag</code>	Tag does not exist.
<code>IFD_Error_Not_Supported</code>	Tag not supported.

IFD_Set_Capabilities()

This smart card API attempts to set the parameter specified by `Tag` to value. This function can be used by the application to set parameters such as the current IFSD, or to request an extension of the BWT.

Prototype `RESPONSECODE IFD_Set_Capabilities(INDWORD Tag,IN BYTE[] Value);`

Return Values `RESPONSECODE` can be one of the following:

<code>IFD_Success</code>	Parameter successfully set.
<code>IFD_Error_Set_Failure</code>	Operation failed.
<code>IFD_Error_Tag</code>	Tag does not exist.
<code>IFD_Error_Value_Read_Only</code>	The value cannot be modified.
<code>IFD_Error_Not_Supported</code>	Tag not supported.

IFD_Set_Protocol_Parameters()

The smart card API attempts to set the parameter specified by `Tag` to `Value`. This function can be used by the application to set parameters such as the current IFSD, or to request an extension of the BWT. A sample code is shown below.

```
RESPONSECODE IFD_Set_Protocol_Parameters(
    IN DWORD   ProtocolType
    IN BYTE    SelectionFlags
    IN BYTE    PTS1    // Encodes Clock Conversion
                        // and bit duration factors
    IN BYTE    PTS2    // RFU according to Iso7816-3
    IN BYTE    PTS3    // RFU according to Iso7816-3
);
```

An application specifies its preferred protocols and protocol parameters.

The `ProtocolType` parameter can be:

- a list of protocol types, coded in the same way as for tag 0x0120 and 0x0126.
- the special value `IFD_DEFAULT_PROTOCOL` (defined as 0x80000000).

`SelectionFlags` indicates which of the optional parameters (`PTS1`, `PTS2` and `PTS3`), if any, must be negotiated and included in the PTS request. Performing a bitwise OR operation on the following flags obtains the parameter:

- `IFD_NEGOTIATE_PTS1`: 1
- `IFD_NEGOTIATE_PTS2`: 2
- `IFD_NEGOTIATE_PTS3`: 4

The `PTS1`, `PTS2`, and `PTS3` bytes are the parameter characters as defined in the ISO 7816-3.

Prototype

```
RESPONSECODE IFD_Set_Capabilities(IN DWORD Tag, IN BYTE[]Value);
```

Return Values

`RESPONSECODE` can be one of the following:

<code>IFD_Success</code>	Parameter successfully set or PTS succeeded.
<code>IFD_Error_Set_Failure</code>	Operation failed.
<code>IFD_Error_PTS_Failure</code>	PTS failed.
<code>IFD_Error_Tag</code>	Tag does not exist.
<code>IFD_Error_Value_Read_Only</code>	The value cannot be modified.
<code>IFD_Error_Not_Supported</code>	Tag / PTS not supported.
<code>IFD_Protocol_Not_supported</code>	Protocol not supported.

IFD_Power_ICC()

This function is used to power up, power down, or reset the ICC. The desired action is specified by the `ActionRequested` parameter. The following actions are permitted:

- `IFD_POWER_UP`: Requests activation of the contact (cold ATR).
- `IFD_POWER_DOWN`: Requests deactivation of the contact.
- `IFD_RESET`: Requests a warm reset of the ICC (warm ATR).

If the function reports success and the action requested was either a reset or a power up, the ATR returned by the card and the protocol parameters can be accessed through the `IFD_Get_Capabilities` function.

Note that the ATR string, and so on, is available only after issuing the `IFD_Power_ICC()` command (cold or warm ATR). Also, note that [Get Capabilities](#) and [Get Capabilities](#) are updated by the `IFD_Power_ICC()` command (cold or warm ATR).

The smart card API cannot determine if the inserted card is synchronous or asynchronous. If an application supports both card types, the application must provide the necessary control. For example, the application can perform a power on for an asynchronous card, assuming it is an asynchronous card. If the ATR fails, the application can then perform a power on for a synchronous card.

Prototype

```
RESPONSECODE IFD_Power_ICC(IN WORD ActionRequested);
```

Return Values

`RESPONSECODE` can be one of the following:

`IFD_Success`

`IFD_Error_Power_Action`

`IFD_Error_Not_supported`

The requested action could not be carried out.

One of the requested actions is not supported.

Mechanical Characteristics

Support for the following three function calls is optional.

Swallow the ICC

- IFD_Swallow_ICC ()

Eject the ICC

- IFD_Eject_ICC ()

Confiscate the ICC

- IFD_Confiscate_ICC()

Syntax

Each service is described using notation similar to the following:

```
RESPONSECODE Name_Of_Service( IN DWORD param1, IN/OUTBYTE[] param2, OUT
WORD param3);
```

In this notation the following type alias are used:

BYTE	unsigned char
WORD	unsigned short
DWORD	unsigned long
RESPONSECODE	unsigned long (as a return value)

Each parameter is specified as either incoming (IN means to the card), outgoing (OUT means from the card), or both (IN/OUT).

IFD_Swallow_ICC()

This function causes a mechanical swallow of the ICC, if the IFD supports this feature.

Prototype	RESPONSECODE IFD_Swallow_ICC();	
Return Values	RESPONSECODE can be one of the following:	
	IFD_Success	Card successfully swallowed.
	IFD_Error_Swallow	Card not swallowed.
	IFD_Error_Not_supported	Function not supported

IFD_Eject_ICC()

This function causes a mechanical ejection of the ICC, if the IFD supports this feature.

Prototype	RESPONSECODE IFD_Eject_ICC();	
Return Values	RESPONSECODE can be one of the following:	
	IFD_Success	Card successfully ejected.
	IFD_Error_Eject	Card not ejected.
	IFD_Error_Not_supported	Function not supported

IFD_Confiscate_ICC()

This function causes the IFD to confiscate the ICC, if the IFD supports this feature.

NOTE

Currently, the Verix eVo-based terminals support this feature. The function always returns `IFD_Error_Not_supported`.

Prototype

```
RESPONSECODE IFD_Confiscate_ICC();
```

Return Values

RESPONSECODE can be one of the following:

<code>IFD_Success</code>	Card successfully confiscated.
<code>IFD_Error_Confiscate</code>	Card not confiscated.
<code>IFD_Error_Not_supported</code>	Function not supported

Communication Services

These function calls provide the mechanism for data exchange between the application interface and the smart card. Both synchronous and asynchronous smart cards are supported. You may refer to this API for more information.

Data Exchange with the ICC

- IFD_Transmit_to_ICC()

Syntax

Each service is described using notation similar to the following:

```
RESPONSECODE Name_Of_Service( IN DWORD param1, IN/OUTBYTE[] param2, OUT  
WORD param3);
```

In this notation the following type alias are used:

BYTE	unsigned char
WORD	unsigned short
DWORD	unsigned long
RESPONSECODE	unsigned long (as a return value)

Each parameter is specified as either incoming (IN means to the card), outgoing (OUT means from the card), or both (IN/OUT).

IFD_Transmit_to_ICC()

This function instructs the smart card API to send the command specified in the `CommandData` parameter to the ICC and return the response in the `ResponseData` parameter. This function also supports the data exchange for synchronous and asynchronous smart cards.

For the asynchronous cards, this function follows the ISO 7816-4 level using the APDU communication data exchange. Therefore, this function hides the use of the communication protocol (T=0 or T=1). The APDU needs to be formatted as described in ISO 7816-4.

Only the short format of *Lc* and *Le* is supported (one byte long). The `CommandData` parameter is a binary array structured as follows:

SCARD_IO_HEADER	Protocol Data
-----------------	---------------

where `SCARD_IO_HEADER` is defined as follows:

```
dword protocol;
dword length;
```

`Protocol Data` contains the APDU to send to the card. The `ResponseData` parameter contains optional data returned by the ICC, followed by two status words, SW1-SW2.

LENGTH	ReturnedData + SW1-SW2
--------	------------------------

where, `ResponseData` is defined as follows:

```
word LENGTH;

//defines the total length of the ReturnedData plus the SW1-SW2 bytes
byte[] ReturnedData;

byte SW1;

byte SW2;
```

Prototype

```
RESPONSECODE IFD_Transmit_to_ICC(INBYTE []CommandData, OUT BYTE []
ResponseData);
```

Return Values

`RESPONSECODE` can be one of the following:

<code>IFD_Success</code>	The request was successfully sent to the ICC.
<code>IFD_Communication_Error</code>	The request could not be sent to the ICC.
<code>IFD_Response_TimeOut</code>	The IFD timed out waiting for the response from the ICC.
<code>IFD_Error_BadFormat</code>	Input message is in a bad format.

ICC Insertion and Removal

The smart card API does not include an interrupt-based mechanism to indicate to the application if a card was inserted or removed. The application must poll using either `IFD_Is_ICC_Present()` or `IFD_Is_ICC_Absent()`. You may refer to these APIs for more information.

ICC Present

- `IFD_Is_ICC_Present()`

ICC Removed

- `IFD_Is_ICC_Absent()`

Syntax

Each service is described using notation similar to the following:

```
RESPONSECODE Name_Of_Service( IN DWORD param1, IN/OUTBYTE[] param2, OUT  
WORD param3 );
```

In this notation the following type alias are used:

BYTE	unsigned char
WORD	unsigned short
DWORD	unsigned long
RESPONSECODE	unsigned long (as a return value)

Each parameter is specified as either incoming (IN means to the card), outgoing (OUT means from the card), or both (IN/OUT).

IFD_Is_ICC_Present()

Asynchronously signals insertion of an ICC into the interface device.

NOTE



VeriFone SAM sockets do not have a card insertion switch; the card detect signal is hard-wired to Vcc and always indicates card present. As a result, IFD_Is_ICC_Present() and IFD_Is_Card_Absent() always return "Card Present" when the selected ICC is one of the SAM slots.

Prototype

```
RESPONSECODE IFD_Is_ICC_Present();
```

Return Values

RESPONSECODE can be one of the following:

IFD_Success	ICC present.
IFD_Failure	ICC not present.

IFD_Is_ICC_Absent()

Asynchronously signals removal of the ICC from the interface device.

Prototype

```
RESPONSECODE IFD_Is_ICC_Absent();
```

Return Values

RESPONSECODE can be one of the following:

IFD_Success	ICC present.
IFD_Failure	ICC not present.

NOTE



For information on Synchronous Card Communication Refer to *SC5000 CardSlot Library Programmers Guide*, VPN - 22564.

Enumeration of the Device Capabilities

The smart card API provides an interface that supports enumeration of the functionality. Information is returned using a TLV (tag-length-value) structure.

Note that [Set/Get Capabilities](#) Example is set when the commands `open-ICC` (tag 0x0188) and `select-a-particular-ICC` (tag 0x0190) are performed. The smart card API returns the ICC state of the selected ICC. All tags listed in [Table 27](#) can be set and read.

Table 27 Codes for Enumerating Interface Device Capabilities

Data Element	Tag	MAX Length	Data Encoding
Communications			
Channel ID	0x0110	4 bytes	Dword encoded as 0x <i>DDDDCCCC</i> , where: <ul style="list-style-type: none"> <i>DDDD</i> = data channel type <i>CCCC</i> = channel number The following encodings are defined for <i>DDDD</i> : <ul style="list-style-type: none"> 0x01 serial I/O; <i>CCCC</i> is port number. 0x02 parallel I/O; <i>CCCC</i> is port number. 0x04 PS/2 keyboard port; <i>CCCC</i> is zero. 0x08 SCSI; <i>CCCC</i> is SCSI ID number 0x10 IDE; <i>CCCC</i> is device number. 0x20 USB; <i>CCCC</i> is device number. 0xFy vendor-defined interface, with y in the range 0-15; <i>CCCC</i> is vendor defined.
Mechanical Characteristics			
Mechanical characteristics supported	0x0150	4 bytes	Dword result of a bitwise OR operation performed on the following values: <ul style="list-style-type: none"> 0x00000000 No special characteristics. 0x00000001 Card swallow mechanism. 0x00000002 Card eject mechanism. 0x00000004 Card capture mechanism. All other values are RFU.
Protocol (see PC/SC Part 2 of this specification)			
Asynchronous protocol types supported	0x0120	4 bytes	Dword encoded as 0x0 <i>RRRPPPP</i> , where: <ul style="list-style-type: none"> <i>RRR</i> is RFU and should be 0x000. <i>PPPP</i> encodes the supported protocol types. A '1' in a given bit position indicates support for the associated ISO protocol. Example: 0x00000003 indicates support for T=0 and T=1. This is the only compliant value that currently may be returned by devices. All other values (T=2, T=14, T=15, and so on) are outside this specification and must be handled by vendor-supplied drivers.

Table 27 Codes for Enumerating Interface Device Capabilities (continued)

Data Element	Tag	MAX Length	Data Encoding
Default CLK	0x0121	4 bytes	Default ICC CLK frequency in kHz encoded as little-endian integer value. Example: 3.58 MHz is encoded as the integer value 3580.
MAX CLK	0x0122	4 bytes	Maximum supported ICC CLK frequency in kHz, encoded as little-endian integer value.
Default data rate	0x0123	4 bytes	Default ICC I/O data rate in bps encoded as little endian integer.
MAX data rate	0x0124	4 bytes	MAX supported ICC I/O data rate in bps.
MAX IFSD	0x0125	4 bytes	Dword indicating MAX IFSD supported by IFD. MIN 32,254 is recommended.
Synchronous protocol types supported	0x0126	4 bytes	Dword encoded as 0x4RRRPPPP where: <ul style="list-style-type: none"> • RRR is RFU and should be 0x000. • PPPP encodes the supported protocol types. A '1' in a given bit position indicates support for the associated protocol. • 0x0001 indicates support for 2-wire protocol. • 0x0002 indicates support for 3-wire protocol. • 0x0004 indicates support for I²C-Bus protocol. All other values are outside this specification, and must be handled by vendor-supplied drivers.
Security Assurance Features			
User-to-card authentication devices	0x0140	4 bytes	Dword result of a bitwise OR operation performed on the following values: <ul style="list-style-type: none"> • 0x00000000: No devices. • 0x00000001: RFU. • 0x00000002: Numeric (that is, PIN) pad. • 0x00000004: Keyboard. • 0x00000008: Fingerprint scanner. • 0x00000010: Retinal scanner. • 0x00000020: Image scanner. • 0x00000040: Voice print scanner. • 0x00000080: Display device. • 0x0000dd00: dd is vendor selected for a vendor-defined device.

Table 27 Codes for Enumerating Interface Device Capabilities (continued)

Data Element	Tag	MAX Length	Data Encoding
User authentication input device	0x0142	4 bytes	Dword result of a bitwise OR operation performed on the following values: <ul style="list-style-type: none"> • 0x00000000: No devices. • 0x00000001: RFU. • 0x00000002: Numeric (that is, PIN) pad. • 0x00000004: Keyboard. • 0x00000008: Fingerprint scanner. • 0x00000010: Retinal scanner. • 0x00000020: Image scanner. • 0x00000040: Voice print scanner. • 0x00000080: Display device. • 0x0000dd00: <i>dd</i> in the range 0x01-0x40 is vendor selected for a vendor-defined device. • 0x00008000: Indicates encrypted input supported.
Power Management			
Power mgmt. supported	0x0131	4 bytes	<ul style="list-style-type: none"> • If 0, device does not support power down while ICC inserted. • If non-zero, device supports power down while ICC inserted.
Vendor			
Vendor name	0x0100	32 bytes	ASCII string.
Vendor-specified IFD type	0x0101	32 bytes	ASCII string.
Vendor-specified IFD version number	0x0102	4 bytes	Dword encoded as 0XMMmmbbbb where: <ul style="list-style-type: none"> • <i>MM</i> = Major version. • <i>mm</i> = Minor version. • <i>bbbb</i> = Build number.
IFD serial number	0x0103	32 bytes	ASCII string.
Vendor Defined Features			
Vendor defined features	May use values in range 0x0180-0x01F0	--	Refer to Specific Features for the Smart Card API .

ICC Interface Management

Tags listed in [Table 28](#) are set by the smart card API. These tags should not be set by the application. Tags listed in [Table 28](#) are set when:

- the commands `open-ICC` (tag 0x0188) and `select-a-particular-ICC` (tag 0x0190) are performed,
- and the `IFD_Power_ICC()` command is issued (cold or warm ATR).

Therefore, open the reader, then send `select-a-particular ICC` and issue the `IFD_Power_ICC()` command before attempting to read any value mentioned in [Table 28](#). The smart card API returns the state of the selected ICC.

Table 28 Codes for Enumerating ICC State

Information	Tag	MAX Length	Responses (return as integer)
ICC present	0x0300	1 byte	<ul style="list-style-type: none"> 0 = Not present. 1 = Card present but not swallowed (applies only if the IFD supports ICC swallow). 2 = Card present (and swallowed if the IFD supports ICC swallow). 4 = Card confiscated.
ICC interface status	0x0301	1 byte	Boolean: <ul style="list-style-type: none"> 0 = Contact inactive. 1 = Contact active.
ATR string	0x0303	33 bytes	Contains the ATR string as returned by the ICC.
ICC type, based on ATR sequence	0x0304	1 byte	ISO/IEC 7816 or unknown: <ul style="list-style-type: none"> 0 = Unknown ICC type. 1 = 7816 asynchronous. 2 = 7816 synchronous. Other values RFU.
New Features:			
Length of ATR string	0x0305	2 bytes	Contains the ATR length.

Protocol Support

The smart card API hides all protocol-related details from the application level and presents a standard interface based on the ISO 7816-4 commands/responses structure. Tags listed in [Table 29](#) are set when:

- the commands `open-ICC` (tag 0x0188) and `select-a-particular-ICC` (tag 0x0190) are performed.
- the `IFD_Power_ICC()` command is issued (cold or warm ATR).

Therefore, open the reader, then `select-a-particular ICC` and issue the `IFD_Power_ICC()` command before attempting to read any value mentioned in this [Table 29](#). The smart card API returns the ICC state of the selected ICC.

Tags listed in Table 29 are read-only, except tag 0x208. Only tag 0x208 can be set by the application.

Table 29 Codes for Enumerating Interface Device Protocol Options

Data Element	Tag	MAX Length	Read-Only	Comments
Current protocol type	0x0201	4 bytes	4	D encoded in the same manner as available protocol types. It is illegal to specify more than a one protocol in this value.
Current CLK	0x0202	4 bytes	4	Current ICC CLK frequency in kHz, encoded as a little-endian integer value. Example: 3.58 MHz is encoded as the integer value 3580.
Current F (clock conversion factor)	0x0203	4 bytes	4	F encoded as a little-endian integer (can be modified through PTS).
Current D (bit rate conversion factor)	0x0204	4 bytes	4	D encoded as a little-endian integer (can be modified through PTS).
Current N (guard time factor)	0x0205	4 bytes	4	N encoded as a little-endian integer (can be modified through PTS).
Current W (work waiting time)	0x0206	4 bytes	4	W encoded as a little-endian integer. Only valid if current protocol is T=0.
Current IFSC (information field size card)	0x0207	4 bytes	4	IFSC encoded as a little-endian integer. Only valid if current protocol is T=1.
Current IFSD (information field size reader)	0x0208	4 bytes		If the application does not set/change the IFSD, a default value of 32 is used. The MAX IFSD allowed is 0xFE. If the reader does not support changing this, an error is returned. IFSD encoded as a little-endian integer. Only valid if current protocol is T=1.
Current BWT (block waiting time)	0x0209	4 bytes	4	BWT encoded as a little-endian integer. Only valid if current protocol is T=1.
Current CWT (character waiting time)	0x020A	4 bytes	4	CWT encoded as a little-endian integer. Only valid if current protocol is T=1.
Current EBC encoding	0x020B	4 bytes	4	EBC encoded as: <ul style="list-style-type: none"> • 0 = LRC • 1 = CRC Only valid if current protocol is T=1.

Specific Features for the Smart Card API

Table 30 describes specific features for the smart card API.

Table 30 Codes for Enumerating Specific Features for the Smart Card API

Data Element	Tag	MAX Length	Data Encoding
Note: These new features are not a part of the PC/SC standard, but are required for the smart card API.			
Open the smart card reader	0x0180	4 bytes	<p>This tag opens the smart card reader. Before reading any tag value mentioned in Table 27, select the reader using tag 0x0180. Read values pertain to the attributes of the selected reader.</p> <p>This tag is for exclusive use by one application. Tag 0x181 must be called before calling this tag again.</p> <ul style="list-style-type: none"> 0x00000001 reader 1 is selected. <p>This tag can only be set.</p> <p>Note: This tag is supported as of 12/2000, but will eventually be phased out. Avoid using this tag.</p>
Close the smart card reader	0x0181	4 bytes	<p>This tag closes the smart card reader.</p> <p>This tag can only be set.</p> <p>Note: This tag is supported as of 12/2000, but will eventually be phased out. Avoid using this tag.</p>
Open ICC	0x188	4 bytes	<p>This tag opens one ICC.</p> <p>One “open one ICC” must be performed per card slot.</p> <ul style="list-style-type: none"> The least-significant word defines the selected ICC. The most-significant word defines the standard. <p>Valid Values:</p> <ul style="list-style-type: none"> CUSTOMER_SLOT MERCHANT_SLOT_1 MERCHANT_SLOT_2 MERCHANT_SLOT_3 MERCHANT_SLOT_4 <p>See LIBVOY.H for device definition.</p> <p>Setting this tag opens the device associated with the specified slot. Ownership of that smart card device is given to the calling task.</p> <p>Note: Some terminals may have a limited number of MSAM slots or may not have customer slots. Not all slots may be present.</p> <p>This tag can only be set.</p>

Table 30 Codes for Enumerating Specific Features for the Smart Card API (continued)

Data Element	Tag	MAX Length	Data Encoding
Close ICC	0x189	4 bytes	<p>This tag closes one ICC. At the end of a communication, call this tag to close the open ICC.</p> <p>One “close ICC” must be performed per card.</p> <p>Valid values:</p> <ul style="list-style-type: none"> CUSTOMER_SLOT MERCHANT_SLOT_1 MERCHANT_SLOT_2 MERCHANT_SLOT_3 MERCHANT_SLOT_4 <p>This tag must be set in order to close the smart card device releasing ownership.</p> <p>This tag can only be set.</p>
Select ICC	0x190	4 bytes	<p>This tag selects one ICC. This tag must be called before any data communication exchange with the card.</p> <p>Select a particular ICC, using tag 0x0190 before attempting to read any value mentioned in Table 28 or Table 29.</p> <p>Valid values:</p> <ul style="list-style-type: none"> CUSTOMER_SLOT MERCHANT_SLOT_1 MERCHANT_SLOT_2 MERCHANT_SLOT_3 MERCHANT_SLOT_4 <p>This tag can only be set.</p>
NAD management	0x0191	4 bytes	<p>This tag manages the NAD, if supported.</p> <ul style="list-style-type: none"> 0x00000000: Requests not to manage the NAD. 0x0000XX01: Requests to manage the NAD. XX contains the SAD and DAD as described in ISO-7816. <p>This tag can be set and read.</p> <p>Note: This is only valid for T=1 protocol when the Visa Cash standard is selected.</p>
Convention (direct or inverse convention)	0x0192	4 bytes	<ul style="list-style-type: none"> If read value is zero, then the convention is direct. If read value is non zero, then the convention is inverse. <p>This tag is read-only.</p> <p>Note: Only valid for asynchronous cards.</p>
WTX management	0x0193	4 bytes	<p>WTX Management is always enabled. Setting this tag will have no effect on WTX Management.</p> <p>Note: Only valid for T=1 protocol.</p> <ul style="list-style-type: none"> 0x00000000: Requests not to manage the WTX. 0x00000001: Requests to manage the WTX. <p>This tag can only be set.</p>

Table 30 Codes for Enumerating Specific Features for the Smart Card API (continued)

Data Element	Tag	MAX Length	Data Encoding
Power on: class selection	0x0194	4 bytes	<p>IMPORTANT: This tag <i>must</i> be called before a cold ATR order is issued to select the correct voltage and card type (asynchronous/synchronous).</p> <ul style="list-style-type: none"> 0x00000001: ICC is a class A card (5V). 0x00000002: ICC is a class B card (3V). 0x00000003: ICC is a class AB card. 0x00000004: ICC is a 1.8V card. <p>This tag can be read and set.</p>
PTS management	0x0195	4 bytes	PTS Management is always automatic. Setting this tag will have no effect.
Error code	0x0196	4 bytes	<p>This tag allows the error codes to be read. This tag is expected to provide specific information about the last error that occurred, which depends on the implemented hardware module.</p> <p>Error code = 0x100000XX: where XX = Error Status Code (see Table 31 for ESC code values). Error code = 0x000000XX: where XX error code defined in errno.h.</p> <ul style="list-style-type: none"> 0x00000001: W time-out (T0). 0x00000002: CWT time-out (T1). 0x00000004: BWT time-out (T1). <p>This tag is read-only.</p>
Select the standard	0x0197	4 bytes	<p>This tag allows the standard to be selected. Default is the EMV 3.1.1 standard.</p> <ul style="list-style-type: none"> 0x00000000: EMV 3.1.1. 0x00000001: ISO7816-3. <p>This tag can be read and set.</p>
Type of synchronous card	0x01A0	4 bytes	<p>Indicates the type of synchronous card, or if asynchronous.</p> <ul style="list-style-type: none"> 0x00000000: An asynchronous card is indicated. Default value. 0x000000XX <p>This tag can be read and set. See <i>SC5000 CardSlot Library Programmers Guide</i>, VPN - 22564, for more details.</p>
ICC structure	0x01B0	4 bytes	<p>This tag enables an application program to read the internal structure. A controller task can power up and transfer control of the card and relevant structures to another task without the other task requiring ICC power up.</p> <p>See Test if ICC Present or Absent.</p>
ICC structure size	0x01B2	4 bytes	<p>This tag is related to the 0x01B0 tag. The application can request the size of the ICC structure before issuing the 0x01B0 tag.</p>
IFM Version	0x183	4 bytes	<p>The tag retrieves the current software version of the smart card IFM.</p> <p>This tag can only be read.</p>

Table 30 Codes for Enumerating Specific Features for the Smart Card API (continued)

Data Element	Tag	MAX Length	Data Encoding
Override ATR	0x184	Max 32 bytes	Immediately following the receipt of the ATR string, the application may override the ATR string from the card by using this tag to send an alternate ATR string to the reader for the selected slot. This tag can only be set.
Reader IFSD	0x185	4 bytes	This tag will allow the application to read the current IFSD of the smart card reader. This tag can only be read.

Table 31 displays the ESC code values.

Table 31 ESC Code Values

Escape Code	Value
NO_ERROR	0
Generic Error Definitions	
CARD_DEACTIVATED	0x01
CARD_MOVED	0x02
CARD_NOT_PRESENT	0x03
Error Definitions for ATR	
ATR_MUTE	0x10
EARLY_ANSWER	0x11
ATR_PARITY_ERROR	0x12
ATR_WWT_EXCEEDED	0x13
ATR_DURATION_EXCEEDED	0x14
TB1_NOT_ZERO	0x15
TB1_NOT_PRESENT	0x16
NO_T0_NO_T1	0x17
B5_OF_TA2_SET	0x18
TB2_PRESENT	0x19
WRONG_WI	0x1A
PROTOCOL_MISMATCH	0x1B
WRONG_IFSC	0x1C
WRONG_CWI	0x1D
WRONG_BWI	0x1E
WRONG_TC1_CWT	0x1F
TB3_ABSENT	0x20
WRONG_TC3	0x21
BAD_FiDi	0x22
ATR_CHECKSUM_ERROR	0x23
ATR_LEN_EXCEEDED	0x24
TS_NEITHER_3B_OR_3F	0x25

Table 31 **ESC Code Values** (continued)

Escape Code	Value
ATR_NOT_SUPPORTED	0x26
Error Definitions for T=0 Protocol	
TOO_SHORT_APDU	0x30
WRONG_APDU	0x31
WWT_EXCEEDED	0x32
INS_ERROR	0x33
T0_PARITY_ERROR	0x34
Error Definitions for T=1 Protocol	
CARD_MUTE_NOW	0x50
RESYNCHRONISED	0x51
CHAIN_ABORT	0x52
BAD_NAD	0x53
IFSD_NOT_ACCEPTED	0x54
Error Definitions for PPS Negotiation	
PARAM_ERROR	0x70
PPS_NOT_ACCEPTED	0x71
RESPONSE_ERROR	0x72
PCK_ERROR	0x73
PPS_PARITY_ERROR	0x74
Hardware Errors	
CARD_ERROR	0xE0
BAD_CLOCK_CARD	0xE1
UART_OVERFLOW	0xE2
SUPERVISOR_ACTIVATED	0xE3
TEMPERATURE_ALARM	0xE4
FRAMING_ERROR	0xE9
Additional Errors not from EMV Library	
TOO_MANY_CARDS_POWERED	0x109 (PIN pad only)

Smart Card Code Examples

This section provides code examples for both asynchronous and synchronous smart cards.

Asynchronous Cards

Complete Program

Example

The linked example code file is for a 5V asynchronous card. Subsequent sections provide additional details.

Select Used Cards

Example

In the linked example code file, an application uses cards PSCR and MSAM1.

Cold ATR for PSCR

Example The linked example code file is for a 5V asynchronous card.

Warm ATR for PSCR

Example The linked example code file is for a 5V asynchronous card.

Switch Off PSCR

Example The linked example code file is for a 5V asynchronous card.

APDU Exchange on PSCR

Example The linked example code file is for asynchronous cards using the T=0 or T=1 protocols. The ATR bytes define the protocol.

Another APDU Exchange

Example The linked example code file is for asynchronous cards using the T=1 protocol.

Manual Protocol Type Selection

Example The linked example code file is for asynchronous cards using the T=0 protocol.

FSD Request

Example The linked example code file is only for T=1 cards.

Get Capabilities

Example The linked example code file returns general information about the card or the reader.

Get Capabilities

Example The linked example code file returns general information about the card.

Multi-Application Support

This section has links to example code files with new tags added for Multi-application support.

Example Click on the link to view the code for Application 1.

Example Click on the link to view the code for Application 2.

Set/Get Capabilities

Example The linked example code file returns general information about the card.

Common Function Calls

This section has links to example code files for common function calls for both synchronous and asynchronous cards.

Test if ICC Present or Absent

Example

The linked example code file is for either asynchronous or synchronous cards.

Swallow/Eject/Confiscate

Example

The linked example code file is for either asynchronous or synchronous cards.

Contactless Reader

VX 680 3G uses NXP PN512 contactless smart card reader. The upgraded C2 version of the chip is used in conjunction with EMV L1 protocol driver version 03.03.09 to comply with EMV 2.1 contactless L1 standard. Requirements to load 03.03.09 support includes a CIB update specifying the CTLS Configuration ID equal to "0x32". In addition a CTLS Definition file is required to specify the CTLS antenna analog character.

Real-Time Clock

Verix eVo-based terminals support a real-time clock device that maintains the current date and time, and provides a source of periodic interrupts for system timing. Trident terminals RTC is part of the embedded SOC.

The function `read_clock` allows the real-time clock to be read without a handle. Because of this, many applications do not need to open the real-time clock device. The application does not specify a buffer size and only one application can open the clock at a time. After an application opens the clock, it is then allowed to write to and read from the clock.

Related Clock Function Calls

The following are clock functions. Unlike some other operating systems, Verix eVo reads and writes date and time information using ASCII arrays. This format is ideally suited for printing or display purposes. Occasionally, however, it may be useful to convert the date or time to binary numbers. The following related functions may be useful in this regard:

```
void date2days(char *yyyymmdd, long *days);
int days2date (long *days, char *yyyymmdd);
void datetime2seconds (char *yyyymmddhhmmss, unsigned long *seconds);
int get_bits_per_second(int hdl);
int seconds2datetime (const unsigned long *seconds, char *yyyymmddhhmmss);
int SVC_VALID_DATE (const char *yyyymmddhhmmss);
void secs2time (const long *secs, char *hhmmss);
void time2secs (const char *hhmmss, long *secs);
```

Parameters

date	Indicates the ASCII date in eight bytes, <i>yyyymmdd</i> , where:
yyyy	Indicates year.
mm	Indicates month.
dd	Indicates day.

days	Indicates the number of days since January 1, 1980 as a 32-bit binary number.												
time	Indicates the ASCII time in six bytes, hhmmss in 24-hour format (for example, <i>hh</i> values are in the range 00-23), where: <table> <tr> <td>hh</td><td>Indicates hour.</td></tr> <tr> <td>mm</td><td>Indicates minutes.</td></tr> <tr> <td>ss</td><td>Indicates seconds.</td></tr> </table>	hh	Indicates hour.	mm	Indicates minutes.	ss	Indicates seconds.						
hh	Indicates hour.												
mm	Indicates minutes.												
ss	Indicates seconds.												
secs	Indicates the number of seconds since midnight as a 32-bit binary number.												
seconds	Indicates the number of seconds since January 1, 1980 as a 32-bit binary number.												
datetime	Indicates the ASCII date and time in 14 bytes, <i>yyyymmddhhmmss</i> , where: <table> <tr> <td>yyyy</td><td>Indicates year.</td></tr> <tr> <td>mm</td><td>Indicates month.</td></tr> <tr> <td>dd</td><td>Indicates day.</td></tr> <tr> <td>hh</td><td>Indicates hour.</td></tr> <tr> <td>mm</td><td>Indicates minutes.</td></tr> <tr> <td>ss</td><td>Indicates seconds.</td></tr> </table>	yyyy	Indicates year.	mm	Indicates month.	dd	Indicates day.	hh	Indicates hour.	mm	Indicates minutes.	ss	Indicates seconds.
yyyy	Indicates year.												
mm	Indicates month.												
dd	Indicates day.												
hh	Indicates hour.												
mm	Indicates minutes.												
ss	Indicates seconds.												
gets	hdl Device handle.												

SVC_VALID_DATE()

Verifies that the caller's 14-byte buffer contains a valid ASCII date and time in the range 19800101000000-20791231235959. A valid result indicates the day of the week (0=Sunday...6=Saturday). A return result of -1 indicates an invalid ASCII format.

Prototype

```
int SVC_VALID_DATE (const char *yyyymmddhhmmss);
```

Clock Example

The linked pseudocode file is a clock example.

read_clock Example

The linked code example displays a clock on the screen and updates it when the time changes.

date2days Example

The linked code example presents common use of `date2days`, which is to calculate the number of days between two dates.

datetime2seconds Example

The linked code example illustrates how a program could compute the elapsed time between two events.

days2date Example

The linked example code adds `ndays` to `date`, and returns the result in the same array.

seconds2datetime Example

The linked code example advances the given date/time by the given number of seconds.

SVC_VALID_DATE Example

The linked code example returns a string containing the day of the week for a given date.

Real-Time Clock Function Calls

This section presents the real-time clock function calls. You may also refer to the following generic APIs:

- `open()`
- `read()`
- `write()`
- `close()`

The following calls are related to the date and time information supported by the real-time clock.

get_bits_per_second()

Returns the actual bits per second based on the clock rate and the baud rate constant for the processor. This is not valid for COM3.

On BT, this function returns the baud rate requested in the [set_opn_blk\(\)](#) call for backward compatibility.

Prototype `int get_bits_per_second(int hdl);`

Parameters

hdl = Device handle

Return Values

Success: The actual bits per second

Failure: -1 with errno set to EINVAL.

read_clock()

Allows the real-time clock to be read without a handle, thus allowing it to be used regardless of current ownership of this device. Note that the caller does not specify a buffer size; fifteen bytes always returns if a valid buffer is specified. The simplest way to read the current time is using the `read_clock()` call.

Prototype `int read_clock(char *yyyymmddhhmmssW);`

The 15-byte ASCII character array is returned in the format: *yyyymmddhhmmssW*

where, *yyyy* = year

mm = month

dd = day

hh = hour

mm = minutes

ss = seconds

W = weekday (a number between 0–6 where, 0 = Sunday ... 6 = Saturday)

Return Values

A result of -1 indicates a valid buffer: if the caller does not own the fifteen-byte buffer provided, a result of 0 returns.

read_RTC()

Returns the current time and date.

Prototype `int read_RTC(int hdl, char *yyyymmddhhmmssW);`

Parameters The 15-byte ASCII character array is returned in the format: *yyyymmddhhmmssW*

where: *yyyy* = year
 mm = month (1 - 12)
 dd = day (1 - 31)
 hh = hour (0 - 23)
 mm = minutes (0 - 59)
 ss = seconds (0 - 59)
 W = day of the week (1 - 7, where, 1 = Sunday ... 7 = Saturday)

Return Values

Success: 0
Failure: -1: EINVAL

Timer Function Calls

Timers as system features are related to the clock. See the following calls for more timer-related functionality:

- `clr_timer()` - See function description in [Event Function Calls](#).
- `set_timer()` - See function description in [Event Function Calls](#).
- `SVC_WAIT()` - See function description in [Event Function Calls](#).
- `read_ticks()`

read_ticks()

Returns the number of clock ticks elapsed since the terminal was powered up or reset. The tick rate is defined by the constant `TICKS_PER_SEC`. On current platforms it is 1000 (that is, one tick per millisecond). At this rate the count wraps back to zero after 49.7 days. In most cases, it is the difference between two tick values—rather than the absolute value—that is of interest. See also, [read_clock Example](#) and [set_timer\(\)](#).

Prototype `unsigned long read_ticks (void);`

Return Values Returns the current millisecond timer tick from last power cycle.

Example The linked code example illustrates a common use of `read_ticks` to implement a timeout. Note however that an event-based approach using [set_timer\(\)](#) is often preferable to the polling in the example.

Also note that the code may not work correctly if it happens to be called just before the tick counter rolls over to 0. In such a case `timeout` can wrap around to a small number, causing a premature timeout. If the application is prepared to respond gracefully to a time out (for example, with a retry), this low probability event may be an acceptable risk. If not, a more sophisticated time comparison is required.

Beeper

The Verix eVo-based terminal beeper is a device that generates audible tones to aid the end user. Two types of sounds are defined: a *normal* beep (1245 Hz for 50 ms) and an *error* beep (880 Hz for 100 ms).

By default, terminal key presses are accompanied by a normal beep (key beep). The application can disable this feature by calling `int key_beeps(flag = 0)`.

When the USB device is connected while the terminal is running, the OS recognizes it and allows an application to open for it. Whenever a USB device is connected to the external USB port, a special “connected” tone is generated. Similarly, when a device is disconnected, a special “Disconnected” tone is generated.

During terminal startup, the terminal may use the beeper to play a welcome tune. This is accomplished using the `play_RTTTL()` routine that is described in the following section.

NOTE



On VX 820 PIN pad, the OS does not contain any pre-defined tunes. It inherits the tune playing feature of the VX 680 terminal, although the notes may sound differently depending on the buzzer hardware and case design. The application causes the VX 820 unit to play any sequence of single notes to create any desired tune by using the `play_RTTTL()` API. The tune is specified by a string of ASCII characters in the Ring Tones Text Transfer Language (RTTTL).

Beeper Function Calls

This section presents the beeper function calls.

Error Conditions and Error Codes

Errors are reported by returning a result of -1 with `errno` set to a specific standard error code. The caller will receive error codes in the following situations:

ENODEV open: beeper is currently owned by another task.

EINVAL control: note is negative or greater than 95, or negative duration `read`, `write`, `status`, `lseek`: any call.

EBADF read, write, control, status, lseek, close, dir: device not owned by caller (DVIC_MGR).

You may also refer to the following generic APIs:

- `open()`
- `close()`

play_RTTTL()

This library function is used to invoke the RTTTL interpreter and it returns allowing the calling application to continue with other tasks. Meantime, the RTTTL interpreter running as a separate thread, will play the tune. When the tune has been played, the interpreter provides a variable `RTTTL_RET` in the caller's `CONFIG.SYS` file. The return values in the `CONFIG.SYS` are:

- 0 = Tune finished normally
- 1 = Invalid default specifier
- 2 = No "=" in default setting
- 3 = No ',' to separate default setting
- 4 = Invalid note
- 5 = Invalid data
- 6 = Invalid state

Prototype `void play_RTTTL(char *music);`

Parameters

`*music` A buffer containing the music to be played. This string contains characters defined for `sound()`.

Return Values

Success: 0, tune finished normally.
Failure 1, invalid default specifier.
 2, no "=" in default setting.

Example This example file is for RTTTL-format ringtone sequence for the theme from Star Wars.

Example This is an example file is for RTTTL code.

beeper_off()

Immediately squelches the beeper.

Prototype `void beeper_off (void);`

error_tone()

Produces a 100-ms tone at 880 Hz. Control immediately returns to the caller.

Prototype `void error_tone (void);`

normal_tone()

Produces a 50-ms tone at 1245 Hz. Control immediately returns to the caller.

Prototype `void normal_tone (void);`

sound()

Causes the beeper to generate one of the 96 standard tones at a specified time.

Prototype `int sound(int note, int milliseconds);`

The beeper device supports 96 distinct tones designed to approximate eight octaves of the equal tempered musical scale of standard international pitch, with “treble A” having a frequency of 440 Hz. Actual frequencies generated are shown in the following table along with the corresponding musical notes and variations therefrom. The table reflects a system frequency of 200 MHz, the maximum duration is 10,000 ms or 10 seconds. Other values may appear in future platforms for Verix eVo.

The column labels indicate the following characteristics for each of the 96 notes:

- Note - standard “do-re-mi” designation for the musical note.
- N# - the note number, used as a parameter to the sound() function.
- Nominal - frequency in Hertz for the standard musical note.

Table 32 Beeper Tones

Note	N#	Nominal	Note	N#	Nominal
A	0	55.00	A	48	880
A#	1	58.27	A#	49	932
B	2	61.74	B	50	988
C	3	65.41	C	51	1047
C#	4	69.30	C#	52	1109
D	5	73.42	D	53	1175
D#	6	77.78	D#	54	1245
E	7	82.41	E	55	1319
F	8	87.31	F	56	1397
F#	9	92.50	F#	57	1480
G	10	98.00	G	58	1568
G#	11	103.83	G#	59	1661
A	12	110.00	A	60	1760
A#	13	116.54	A#	61	1865
B	14	123.47	B	62	1976
C	15	130.81	C	63	2093
C#	16	138.59	C#	64	2217

Table 32 **Beeper Tones** (continued)

Note	N#	Nominal	Note	N#	Nominal
D	17	146.83	D	65	2349
D#	18	155.56	D#	66	2489
E	19	164.81	E	67	2637
F	20	174.61	F	68	2794
F#	21	185.00	F#	69	2960
G	22	196.00	G	70	3136
G#	23	207.65	G#	71	3322
A	24	220.00	A	72	3520
A#	25	233.08	A#	73	3729
B	26	246.94	B	74	3951
C	27	261.63	C	75	4186
C#	28	277.18	C#	76	4435
D	29	293.66	D	77	4699
D#	30	311.13	D#	78	4978
E	31	329.63	E	79	5274
F	32	349.23	F	80	5588
F#	33	369.99	F#	81	5920
G	34	392.00	G	82	6272
G#	35	415.30	G#	83	6645
A	36	440.00	A	84	7040
A#	37	466.16	A#	85	7459
B	38	493.88	B	86	7902
C	39	523.25	C	87	8372
C#	40	554.37	C#	88	8870
D	41	587.33	D	89	9397
D#	42	622.25	D#	90	9956
E	43	659.26	E	91	10548
F	44	698.46	F	92	11175
F#	45	739.99	F#	93	11840
G	46	783.99	G	94	12544
G#	47	830.61	G#	95	13290

Speaker Audio System

VX 680 3G supports a standalone 1 Watt speaker (one speaker only) with an external audio amplifier. Inside the Trident SoC, the I2S (also known as the IIS) controller drives the 10-bit DAC (Digital to Analog Converter). The DAC analog output pin is connected to a Texas Instruments TPA2028D1 audio amplifier chip with volume control. The output of the amplifier is connected to the speaker.

Speaker with Volume Control

The OS device driver for the speaker supports mono-audio with software controlled volume settings / adjustments function. See [Device Driver APIs](#) for more information.

Buzzer Emulation

The hardware provides the original buzzer circuit in addition to the PCM sound circuit. All existing system buzzer sounds such as key beep, error tone, normal tone, contactless beeps, USB plug/unplug tones, and external power on/off tones will continue to use the buzzer hardware, not the new audio system. Buzzer emulation is not needed. Customizable system sounds is not implemented. System sounds refer to the sounds generated by the OS using existing API functions.

Permanent System Sounds

System sounds for security related features (for example, CTLS, PIN entry, EMV, and others) will continue to use the buzzer hardware.

Simultaneous Audio and Video

The OS provides a device driver for the speaker that a media player can use to play audio while playing video. Note that the media player is not part of the OS.

License and Royalties

The OS speaker device driver only supports uncompressed PCM encoded audio. Since the OS never handles compressed audio, no licensing or royalty fees are required.

Applications may play audio files by reading audio data from the files then writing the raw PCM data to the Verix device `/dev/pcm`. If the audio data is compressed, applications must decompress the audio before writing the raw PCM data to `/dev/pcm`.

The OS does not play audio from a file. The OS plays uncompressed audio data so there is no need to download drivers for other audio types. If an application supports proprietary audio compression algorithms such as those used in MP3 audio, the application developer is responsible for licensing or royalties.

Synchronized Video and Audio

The OS speaker and video device drivers assumes that synchronization is handled by the media player application software. Note that the media player is not an OS deliverable.

Device Driver APIs

This section presents the system audio device driver function calls. You may also refer to the following APIs:

- `open()`
- `read()`
- `write()`
- `close()`

set_pcm_blk()

set_pcm_blk() must be called after open() before calling any other PCM functions. This function call sets the audio sample rate and the format of the samples. It must not be called while audio is being played.

```
struct pcm_blk {
    unsigned int sample_rate;
    da_format_t format;
    char reserved[8]; // Just in case there are other parameters
}
```

The supported samples rates per second are:

- 8000
- 8021
- 11025
- 12000
- 16000
- 22050
- 24000
- 32000
- 44100
- 48000

The supported sample formats are defined in the following enum:

```
typedef enum{
    DA_INT16_BE,           // Native Verix format
    DA_INT16_LE_MONO,      // 16 bit WAV file format
    DA_INT16_LE_STEREO,    // 16 bit WAV file format
    DA_UINT8,              // 8 bit WAV file format
}da_format_t;
```

The sample formats are defined below:

DA_INT16_BE	This is the natural format used for computed sounds. It corresponds to the format of a C array of signed shorts. Samples are 16 bit signed big endian integers (from -32768 to 32767).
DA_INT16_LE_MONO	This is the format used by 16 bit mono WAV file samples. Samples are 16 bit signed little endian integers (from -32768 to 32767).

DA_INT16_LE_STEREO	This is the format used by 16 bit stereo WAV file samples. Samples are pairs of 16 bit signed little endian integers (from -32768 to 32767). The driver mixes the right and left channels so that both are heard.
DA_UINT8	This is the format used by 8 bit mono WAV files samples. Samples are 8 bit unsigned integers (from 0 to 255).

Prototype

```
int set_pcm_blk(int hdl, struct *pcm_blk);
```

Return Values

Success: 0

Failure: Negative error code is returned if an error occurs.

get_pcm_blk()

This API reads the current `pcm_blk` set by `set_pcm_blk()`. It returns 0 for success, or a negative error code if an error occurred.

Prototype

```
int get_pcm_blk(int hdl, struct *pcm_blk);
```

set_pcm_playback()

This API allows the application to control the audio playback. The following actions are possible:

Pause playback	This action causes the audio to stop playing within 192 samples. Audio will remain stopped until the application sends a resume action or <code>/dev/pcm</code> is closed and opened again. Writing samples to <code>/dev/pcm</code> does not resume a paused sound. Sending a pause action when already paused does nothing. Sending a pause action before starting a sound will prevent the new sound from starting until a resume action is sent.
Resume playback	This action resumes a previously stopped sound. Sending a resume action when the audio is not paused does nothing.
Abort playback (discards any buffered data)	This action removes all samples from the audio FIFO and stops the sound within 192 samples. It does not set <code>/dev/pcm</code> to the paused state. If more samples are written to <code>/dev/pcm</code> after an abort action, those samples are played. An abort action while <code>/dev/pcm</code> is paused empties the FIFO and leave <code>/dev/pcm</code> in the paused state.

Prototype `int set_pcm_playback(int hdl, int action);`

get_pcm_status()

This API returns the status of the `/dev/pcm` audio FIFO. The "buffered" field returns the number of samples in the audio FIFO. The "free_space" field returns the number of additional samples that can be written to fill up the FIFO. The size of the FIFO is the sum of the two fields. `pause_state` is 0 if not paused and 1 if paused.

```
struct pcm_status {  
    unsigned int buffered;  
    unsigned int free_space;  
    char pause_state;  
    char reserved[8];  
}
```

Prototype `int get_pcm_status(int hdl, struct *pcm_status);`

set_pcm_volume()

This API sets the gain of the audio amplifier chip. The hardware supports 64 gain levels but try to avoid locking this function to a specific amplifier chip. Other chips may have more gain levels. The driver maps the volume value from 0..255 to 0..63 hardware levels.

The volume level will be persistent across power cycles and device open/close so the user is not required to reset the volume level.

Prototype `int set_pcm_volume(int hdl, int vol);`

get_pcm_volume()

Returns the current volume setting.

Prototype `int get_pcm_volume(int hdl, int &vol);`

System Sounds and Audio Playback

System sounds such as key beep, normal tone, error tone, USB plug/unplug tones, external power on/off tones, and contactless sounds may occur at any time including during audio file playback. There is separate buzzer hardware in VX 680 3G terminals so these sounds can exist with PCM sounds.

Device Detection

```
#define MID_SPEAKER                (52)        // Speaker
#define INFO_PRES_SPEAKER          (0x40000)    // Speaker
```

Device detection APIs

See the following APIs for more information:

- `SVC_INFO_PRESENT()`
- `SVC_INFO_CIB_ID()`
- `SVC_INFO_DEV_TYPE()`

Internal Printer

The internal printer communications are affected by the components described below.

The internal printer is a major user of power in Verix V terminals. Some terminals such as V*670 and VX 680 will not allow the printer to be used unless the battery is inserted since the printer is powered from the battery and not from the external wall power. Power usage can be affected by how dark the printer prints. This is controlled by `CONFIG.SYS` variables such as `*DARK`.

Internal Printer Function Calls

The functions in this section control the internal printer communications.

You may also refer to the following APIs:

- `open()`
- `read()`
- `write()`
- `close()`

Error Conditions and Error Codes

Errors are reported by returning a result of -1 with `errno` set to a specific standard error code. The caller will receive error codes in the following situations:

<code>EBADF</code>	Handle is invalid.
<code>EACCESS</code>	Buffer is an invalid pointer.
<code>ENOSPC</code>	Length is negative.

get_opn_blk()

Copies the current `Opn_Blkc` structure into the caller's `ob` structure. See [set_opn_blk\(\)](#) for additional details on the `opn_blk()` for the printer. The caller is responsible for ensuring that the buffer is large enough to hold the `Opn_Blkc` structure.

On BT, this function returns the serial communication parameter block.

Prototype

```
int get_opn_blk(int hdl, struct Opn_Blkc *ob);
```

Parameters

<code>hdl</code>	Printer handle.
<code>ob</code>	Pointer to open block structure containing configuration.

Return Values

Success:	0: Successful close.
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : with <code>h</code> handle is invalid.
	-1 with <code>errno</code> set to <code>ENXIO</code> : USB device not present.

If the `opn_blk` structure is not previously set, `get_opn_blk()` returns zero and the caller's `OB` structure is set to zero.

NOTE



The difference between Predator and Trident terminals is that the open block parameters for a device are preserved over power cycles in Predator but are not saved in Trident. Using `get_opn_blk()` to retrieve open block values after a power cycle will have differing results between the two terminal types.

set_opn_blk()

Configures the port using the parameters specified in the provided open block structure. Since the printer is not a separate device, setting the open block is not a requirement. If called, `set_opn_blk()` saves the provided `Opn_Bl` information (for `get_opn_blk()`), but not validate the parameters.

There is no UART hardware, this function stores the contents of `ob` in transient memory. Calling `get_opn_blk()` returns the stored values.

When doing `set_serial_lines()` or `set_opn_blk` to USB devices, there may be a brief glitch in the state of the DTR or RTS signals. This can cause the modem to disconnect even if the desired state of the signal is not changed. If the state of DTR or RTS is not changed, do not include the option in the call.

Setting the DTR or RTS in the USB device is caused by sending a low level USB command to the USB device. This command then sets the state as applicable. Setting the state of the signal will cause the state to be unknown for a brief time, and is a function of the HW being used. Thus, when doing a synchronous connection and it is time to change from Asynchronous to Synchronous using the `set_opn_blk` call, **do not** include the `Fmt_DTR` or `Fmt_RTS` options as the modem will disconnect. The same goes for `set_serial_lines`.

On BT, this function sets the serial communication parameters for the SPP serial connection. Most of these parameters will be ignored by the Bluetooth stack. But, for backwards compatibility, the `set_opn_blk()` function must be called before communicating over `DEV_SPP_BT`. Supported bit rates are: 2400, 4800, 9600, 19200, 38400, 57600, and 115200. While data formats supported are: 7 bit and 8 bit data with even, odd, or no Parity. For the Dione base only 7 even and odd parity and 8 no parity are supported.

Prototype

```
int set_opn_blk (int hdl, const struct Opn_Bl *ob);
```

Parameters

<code>hdl</code>	Printer handle.
<code>ob</code>	Pointer to open block structure containing configuration.

Return Values

Success:	0: Successful close.
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : with <code>h</code> handle is invalid.

get_port_status()

Copies current port status information to caller's 4-byte buffer and returns a result code indicating whether or not any output is currently queued for the printer.

The ARM architecture does not support parity or break detection. Note that frame, overrun, and parity errors are latched until reset by an intelligent protocol or by the application calling `reset_port_error(port)`. The result code returned indicates if any output is currently queued or being transmitted. Use the following to close the port without truncating data:

```
while(get_port_status(comm_hdl, &buf) != 0)
    printf("\fOutput pending");
close (comm_handle);
```

When this function is sent to the IPP port, the returned result is 0 (success); however, the result is not viable as this port has no flow control (that is, Tx and Rx are the only lines available on this port).

4-byte Buffer Contents

1st byte	The amount of input messages pending. If the receive buffer contains more than 255 bytes, byte 0 is set to 255.
2nd byte	Not used. This is always set to 0. In IPP: Number of failed output messages pending (always 0).
3rd byte	Number of output slots available. This is computed as maximum slots less slots in use. Be aware that there may not be enough buffers available for all the slots available. Bit 1 of byte 3 is a flag to indicate overrun errors. In IPP: Always 1.
4th byte	Constant. CTS detected and DCD present. In IPP: Current signal information. Always 0 because there is no UART.

Signal information byte:

7	Set if break/abort detected (always 0)
6	Set if DSR detected (COM2 only)
5	Set if CTS detected (COM1, COM2 and COM3)
4	Set if RI (ring indicator) present (always 0)
3	Set if DCD present (COM2 and COM3 only)
2	Set if frame error detected (always 0)
1	Set if overrun error detected (always 0)
0	Set if parity error detected (always 0)

On BT, this function returns status information for the SPP port including error status, and receive data available.

Prototype

```
int get_port_status(int hdl, char *buf);
```

Parameters

hdl	Printer handle.
buf	Pointer to buffer to store printer status

Return Values

Success:	0: No output pending. > 0: Output pending.
Failure:	-1 with errno set to <code>EBADF</code> : hdl is invalid. -1 with errno set to <code>EACCES</code> : buf is an invalid printer. -1 with errno set to <code>ENXIO</code> : USB device not present.

reset_port_error()

Has no effect and the corresponding error indicators are always 0. In general for Verix eVo communication ports, `reset_port_error()` resets the error indicators for parity, framing, and overrun errors, and the break indicator.

On BT, this function resets the indicators for parity errors, framing errors, overrun errors, and break.

Prototype `int reset_port_error(int hdl);`

Return Values

Success: 0

Failure: -1 with `errno` set to `EBADF`: `hdl` is invalid.

-1 with `errno` set to `ENXIO`: USB device not present.

barcode_pending()

This is a Predator call only. This function returns the number of samples for the last scan attempt.

Prototype `int barcode_pending(void);`

Return Values

Success	Number of samples.
Failure	-1: EINVAL

barcode_raw_data()

This is a Predator call only. This function copies the raw barcode buffer to buffer and returns the count.

Prototype `int barcode_raw_data(char *buf);`

Parameters

buf	Buffer to receive the raw data.
-----	---------------------------------

Return Values

Success	Data count.
Failure	-1

Special Items

This section specifies the command set and operation of the system firmware that operates the internal thermal printer. Specifically, the command set covers control codes and escape sequences. Also specified is dot graphic mode operation, and character set and font data organization. Specific character sets must be documented separately and should consist of the following:

- Drawings of all font images and their reference numbers
- A cross reference for ASCII characters showing their mappings for specific countries
- Number of characters in the set (128 or 256) as well as number of countries, and so on

Printer Function

The system information function, `SVC_INFO_PRNTR()`, returns the printer-type code of the installed printer, if the terminal is equipped with an internal printer.

Control Codes and Command Interface

On command format/parsing errors, results on some terminals may be different from earlier printers. When a parsing error is detected on Verix eVo-based terminals, the character that caused the parsing error is thrown away. On earlier printers, the character that causes the error is still processed as printer input.

Printable Characters

Printable characters have hex codes from 20h to FFh. Code 7Fh (DEL) does not denote double-width code. It can be one of the printable characters.

When a printable character is received, it is placed in the print buffer, increasing the buffer pointer by one or two, depending on if double width mode is active (see [Set Double-Width Attribute](#)). If incrementing the buffer pointer causes it to exceed the right margin (`<ESC>e<n>;`), the line automatically prints.

Some of the codes from 20h to 7Eh can be remapped, depending on the currently selected country (see `<ESC>h<n>;`), so that certain characters in that range do not print as their ASCII equivalents. In particular, application programmers coding for international customers should *avoid* using the following characters:

- # instead of the appropriate abbreviation for *number*
- {
- [
- }
-]
- |
- ~
- `

- \

Control Codes

Control codes are hex codes from 00h to 1Fh. [Table 33](#) lists the printer device driver control codes by name, hex code, and function. Note that some are specifically listed as ignored.

Table 33 Control Codes and Function

Name	Code	Operation
NUL	0x00	Ignored
LF	0x0A	Print contents of buffer and advance to next line
VT	0x0B	Advances the paper to the next top of form. It works in conjunction with the new Set Top of Form and Form Length command.
FF	0x0C	Print contents of buffer and advance paper about 20mm
CR	0x0D	Ignored
SO	0x0E	Ignored
SI	0x0F	Ignored
DC1	0x11	Select/Deselect double height
DC2	0x12	Select/Deselect inverse printing
DC3	0x13	Ignored
DC4	0x14	Ignored
CAN	0x18	Empty print buffer character attributes and cancel
ESC	0x1B	Signals start of escape sequence
FS	0x1C	Ignored
GS	0x1D	Ignored
RS	0x1E	Select double width
US	0x1F	Select normal width

New Line

When Line Feed (0Ah) code is received, the buffer prints if it is not empty, and the carriage advances to the beginning of the next line, as specified by the line spacing command (see [<ESC>a<n>;](#)).

Form Feed

When Form Feed code (0Ch) is received, the buffer prints (the same as the Line Feed command), and paper advances to a pre-defined position regardless of the setting of the current line height. This command ensures that the last line of text is visible and that an adequate margin exists for tearing the paper.

Select High Page Character Set

SO code (0Eh) is ignored.

Select ASCII Character Set

SI code (0Fh) is ignored.

Toggle Inverse Printing

When DC2 code (12h) is received, the current print is toggled from normal to inverse, or vice-versa. The line always begins in normal print mode. The first DC2 code in one line allows the ensuing characters to print inversely.

For example, in the data string, abcd <DC2>123<DC2>efgh<LF>, 123 is highlighted by printing inversely.

Empty Print Buffer and Cancel Character Attributes

On receipt of CAN code (18h), the print buffer clears and the color (or inverse print) is also cancelled. However, the double-height and double-width attributes are not canceled.

Set Double-Width Attribute

RS code (1Eh) is received, ensuing characters are considered double width. This attribute remains active until the US code (cancel double width) is received. The double-width attributes do not change after line feeds or CAN codes are received.

NOTE

The double-width attribute has no effect on 48 × 48 and 64 × 64 fonts.

If the character is crossing the line boundary, that part of the character truncates and the following character wraps to the next line.

Cancel Double Width

The US code (1Fh) explicitly resets the double-width attributes. Characters received before the US code print double wide, but ensuing characters do not.

Select/Deselect Double Height

DC1 code (11h) controls double-height attributes for some characters in a line. After line feed is received, this attribute clears. This double-height control is similar to line attribute control <ESC>f<n>; differences are that <ESC>f<n>; applies to the *entire* line and the DC1 code applies only to characters. For 24 × 24 or 32 × 32 download fonts only, DC1 controls the double-height attribute. Refer to <ESC>f<n>; for more information.

NOTE

The double-height attribute has no effect in 48 × 48 and 64 × 64 fonts.

Escape Sequences

Escape sequences are multi-character control sequences. They are the ESC (1Bh) code, a single or multiple characters command, optionally followed by a numeric string that terminates with a semicolon (;). [Table 34](#) lists the printer device driver escape sequences.

Table 34 Printer Escape Sequences

<ESC> Code	Description
<ESC>a<n>;	Sets line height.
<ESC>b<n>, <m>;	Ejects lines. This also accepts an optional second parameter <m> — If <m> is missing or 0, the command ejects <n> lines (as in lines of text with inter-line spacing).
<ESC>c	Resets printer to power-up state.
<ESC>CS;	Retrieves firmware checksum and version.
<ESC>d	Requests printer status.
<ESC>DLRQ [*ZA=APPL_ID, *ZT=TERM_ID];	Ignored in Verix eVo-based terminals.
<ESC>e<n>;	Sets right margin.
<ESC>f<n>;	Selects line attributes.
<ESC>F<n>;	Selects characters per line mode.
<ESC>g	Enters graphics mode.
<ESC>GL<f>, <t>, <w>, <h>; <b1>...<bn>	Downloads graphic image into memory.
<ESC>GP<t>[, <m>];	Prints downloaded graphic image.
<ESC>H<hh1>...<hhn>;	Prints hex code character in downloaded font table.
<ESC>h<n>;	Selects country code.
<ESC>i	Requests printer ID. Terminal ID is "P".
<ESC>I	Ignored in Verix eVo-based terminals.
<ESC>K<n>;	Sets top of form and form length.
<ESC>l<s><t>;	Selects font table for printing and downloading.
<ESC>m<c><rl>...<rn>;	Downloads fonts into memory.
<ESC>S<n>;	Ignored in Verix eVo-based terminals.
<ESC>s	Prints a test pattern.
<ESC>w<n>;	Select fire pulse weight.

General Syntax

Escape sequences consist of ESC code (1Bh) and printable characters in the hex range 20h to 7Eh. Each sequence starts with the ESC code, followed by a unique single-letter code, and are optionally followed by one or more parameters, followed by a semicolon (;).

To ensure that characters within an escape sequence are always printable, parameters are represented as decimal integer strings. Generally, when multiple parameters are present, they must be separated by a comma (.). End the parameter list with a semicolon (;).

Examples

```
ESC a10;
```

```
ESC g
```

<ESC>a<n>;

Sets line height. The parameter <n> is the incremental unit. The unit is 0.1 4mm. The minimum value of <n> is 16; the maximum is 48; default is 22 (that is, 2.75 mm per line or 9.24 lines per inch).

If <n> = 0 or is out of range, it defaults to 22.

<ESC>b<n>,<m>;

Ejects <n> lines. Causes the printer to eject *n* lines of the currently specified height. The length of paper ejected is calculated as:

$$(\text{Number of Lines} \times \text{Line Height}) \div 8$$

8 sets the height in mm; use 203.2 for inches.

A zero value is ignored; the maximum value is 255.

The Eject <n> lines command accepts an optional second parameter <m> — If <m> is missing or 0, the command ejects <n> lines (as in lines of text with inter-line spacing; refer to **<ESC>a<n>;**). If <m> is 1, the command feeds paper forward by pixels for the thermal printer, and millimeters for the sprocket printer. If <m> is 2, the command feeds paper backwards in a similar manner.

<ESC>c

Resets printer to power-up state. This command is the software equivalent of toggling the power switch. All modes reset to default values. This command resets the printer device driver to the default state.

<ESC>d

Requests printer status. After receiving <ESC> d, the status byte in the Verix eVo-based terminals report to the host as follows:

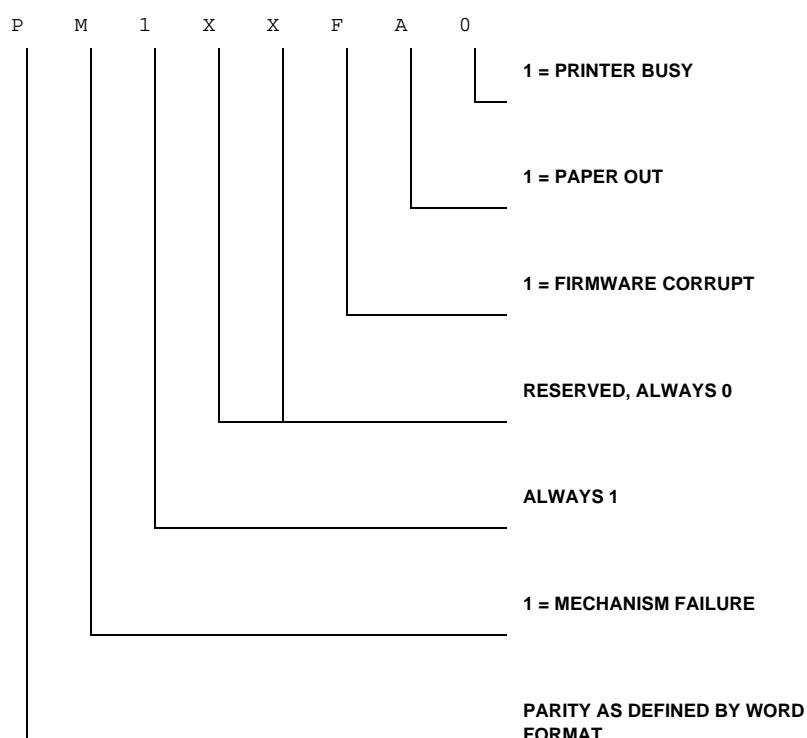


Figure 5 Verix eVo-based Terminals Status Byte Definition

For example: SP means all okay; 60h means the mechanism has failed. When the mechanism failed flag is set, other bits have no meaning.

<ESC>e<n>;

Sets right margin. The right margin setting controls if printing occurs when the buffer is full and at what position on the line. The buffer automatically prints when the maximum characters for a line is received. The printer automatically prints the line on receipt of the *N*th printable character. For values outside the valid range, lines wrap to next line.

In 42 character mode, the maximum characters per line is 42. In 32 character mode, the maximum characters per line is 32. If in 24 character mode, the maximum characters per line is 24. Default is 42. If <n> = 0 or is out of range, the printer device driver retains the last margin setting.

<ESC>f<n>;

Selects line attributes. These are attributes that must be applied to the *entire* line at the time. Character-by-character attributes (for example, boldface, double width, and so on) are set by the control codes discussed in [Control Codes](#).

Valid values and corresponding attributes are shown in [Table 35](#) and can be set only in the combinations shown.

Table 35 Valid Line Values and Attributes

Value	Description
0	Normal
1	Double height
2	Reserved for alternate font
3	Reserved for alternate font at double height

NOTE

For 24 × 24 and 32 × 32 download fonts, the double-height line attribute is not applied. Use the DC1 (11h) control code instead. Please refer to [Select/Deselect Double Height](#) for more information.

The double-height line attribute has no effect in 48 × 48 or 64 × 64 fonts.

<ESC>g

Enters graphics mode. When this escape sequence is received, the printer device driver enters a mode where printable characters are considered graphic images. Graphics mode is cancelled on receipt of any control code or escape command. When this occurs, the buffer clears and all mode flags are reset to default. Refer to [Dot Graphics Mode](#) for more information.

<ESC>h<n>;

The currently selected country. Country parameter affects printing certain character images. The country codes are listed in [Table 36](#).

Table 36 **Country Codes**

Code	Country
0	United States
1	France
2	Germany
3	United Kingdom
4	Denmark I
5	Sweden
6	Italy
7	Spain
8	Japan
9	Norway
10	Denmark II

<ESC>K<n>;

Parameter <n> is the form length in pixels for the thermal printer and millimeters for the sprocket printer. The top of form is set at the current vertical position of the print head.

This works in conjunction with the Vertical Tab (VT) code.

<ESC>i

Sends a request for the printer identity code. The ID code for Verix eVo-based terminals printer device driver is "P." Note that there is no emulation mode in the printer device driver.

<ESC>s

Prints a test pattern.

<ESC>F<n>;

Selects characters per line mode.

<ESC>F<n>;

selects <n> characters per line. <n> can be 42, 32, or 24.

<ESC>l<s><t>;

Selects font table for printing and downloading.

Prototype

<ESC>l<s><t>;

Parameters

- | | |
|-----|---|
| 1 | Lowercase L. |
| <s> | Font size, using the following values: <ul style="list-style-type: none"> • 1 selects 16 × 16 font. • 2 selects 5 × 8 font and 24 column mode. (The 5 × 8 fonts are printed in double width and double height.) • 3 selects 8 × 14 font and 32 column mode. • 4 selects 8 × 14 font and select 42 column mode. • 5 selects 24 × 24 font. • 6 selects 32 × 32 font. • 7 selects 48 × 48 font. • 8 selects 64 × 64 font. |
| <t> | Table ID, normally values from 0 to 64 (see *PRTFNT). <ul style="list-style-type: none"> • 0 selects the built-in font table. • 1 – 64 select download fonts table. <p>The 5 × 8 font uses table <i>t</i> to hold the font image.</p> <p>The 8 × 14 font uses tables <i>t</i> and <i>t</i>+1 to hold the font image.</p> <p>The 16 × 16 font uses tables <i>t</i>, <i>t</i>+1, <i>t</i>+2, <i>t</i>+3 to hold the font images.</p> <p>The 24 × 24 font uses tables <i>t</i>, <i>t</i>+1, <i>t</i>+2 through <i>t</i>+8 to hold the font images.</p> <p>The 32 × 32 font uses tables <i>t</i>, <i>t</i>+1, <i>t</i>+2 through <i>t</i>+15 to hold the font images.</p> <p>The 48 × 48 font uses tables <i>t</i>, <i>t</i>+1, <i>t</i>+2 through <i>t</i>+35 to hold the font images.</p> <p>The 64 × 64 font uses tables <i>t</i>, <i>t</i>+1, <i>t</i>+2 through <i>t</i>+63 to hold the font images.</p> |

The font table is selected for printing or downloading according to the value of <s> and <t>. Refer to <ESC>m<c><r1>...<rn>; for font download escape code information.

Example

<ESC>137 sets the current font table to font table 7, font size 8 × 14, and 32 column per line. From then on, the printer prints the 8 × 14 user-defined characters in table 7 in 32 column mode.

NOTE



The default for <t> is 0; the built-in font table is selected. An application *must* send this command before issuing any download commands.

Only the 5 × 8 and 8 × 14 fonts are built-in fonts. If the user selects a font ID other than these fonts *and* table 0 is selected, the printer device driver uses the 8 × 14, 42 column setting (<s> = 4).

The table below shows a partial list of the printable codes.

Table 37 <s> <t> Printable Codes Size Available Mode, Partial List

<s>	<t>	Printable Codes	Size	Column	Notes
2	0	20h–FFh	5 × 8	24	(built-in font)
2	1	00h–7Fh	5 × 8	24	
2	2	00h–7Fh	5 × 8	24	
.	
.	
2	9	.	.	.	(built-in font)
2	10	00h–7Fh	5 × 8	24	
.	
.	
3	0	20h–FFh	8 × 14	32	
3	1	00h–7Fh	8 × 14	32	(built-in font)
3	3	00h–7Fh	8 × 14	32	
.	
.	
3	9	00h–7Fh	8 × 14	32	
3	11	00h–7Fh	8 × 14	32	(built-in font)
.	
.	
4	0	20h–FFh	8 × 14	42	
4	1	00h–7Fh	8 × 14	42	
4	3	00h–7Fh	8 × 14	42	(built-in font)
.	
.	
4	9	00h–7Fh	8 × 14	42	
4	11	00h–7Fh	8 × 14	42	
.	
.	

<ESC>m<c><r1>...<rn>;

Downloads fonts into memory.

Prototype

<ESC>m<c><r1>...<rn>;

Parameters

<c>	The character code from 00h to 7Fh.
<r1>...<rn>	The horizontal dot-image bytes of the font. The data bytes that follow are the bytes extracted from the printer fonts files (.PFT) that are created by the Font Designer tool.

Examples

Selecting the download font size and table through <ESC>125 enables downloading of a user-defined character through code 41h. The following illustrates the dot pattern:

5 × 7 font	.ooo. <r1> = 0Eh
code 41h	.o.o. <r2> = 0Ah
in table 5	.o.o. <r3> = 0Ah
	oo.oo <r4> = 1Bh
	o...o <r5> = 11h
	o...o <r6> = 11h
	ooooo <r7> = 1Fh

The complete <ESC> m command sequence is:

<ESC> mA<r1><r2><r3><r4><r5><r6><r7>;

hex code: 1B 6D 41 0E 0A 0A 1B 11 11 1F 3B

If the <ESC>1413; escape code is already sent, download the user-defined character with code 62h into table 13 with the <ESC>m command. The following illustrates this dot pattern:

8 × 14 font	.oo...o... <r1> = 64h
code 62h	o...o... <r2> = 94h
	o...o... <r3> = 94h
	o...o... <r4> = 94h
	o...o...o <r5> = 97h
	o...o... <r6> = 94h
	o...o... <r7> = 94h
	.oo...o... <r8> = 64h
 <r9> = 00h
	oooooooo <r10>=FFh
o <r11>=01h
	oooooooo <r12>=FFh
	o..... <r13>=80h
	oooooooo <r14>=FFh

The complete <ESC>m command sequence is:

<ESC>mb<r1><r2>...<r13><r14>;

hex code: 1B 6D 62 64 94 ... 80 FF 3B

NOTE



Always remember the table and size selected for downloading or printing.

<ESC>m<c><r1>...<rn>;

If you've already sent <ESC>1161; you can download a user-defined character with code 30h into table 61 by the <ESC>m command. The following illustrates this dot pattern:

```

16 × 16 font      ..... <r1> =00h, <r9> =00h
code 30h          ..... <r2> =00h, <r10>=00h
                  .....o..... <r3> =04h, <r11>=00h
                  .....o..oo..o... <r4> =04h, <r12>=C8h
                  .oooooooo.....o.. <r5> =7Fh, <r13>=04h
                  .....o.....o. <r6> =04h, <r14>=02h
                  .....o..... <r7> =04h, <r15>=00h
                  .....oooooooo..... <r8> =07h, <r16>=F0h
                  ....oo.....o... <r17>=0Ch, <r25>=08h
                  ...o.o.....o... <r18>=14h, <r26>=04h
                  ..o..o.....o... <r19>=24h, <r27>=04h
                  .o...o.....o... <r20>=44h, <r28>=04h
                  .o...o.....o... <r21>=44h, <r29>=04h
                  .o...o.....o... <r22>=44h, <r30>=08h
                  ..ooo.....ooo.... <r23>=38h, <r31>=38h
                  ..... <r24>=00h, <r32>=00h

```

The complete <ESC>m sequence is:

<ESC>m0<r1><r2>...<r31><r32>;

hex code: 1B 6D 30 00 00 ... 38 00 3B

If the escape code <ESC>1520; is already sent, download the user-defined character with code 41h into table 20 with the <ESC>m command. The following illustrates this dot pattern:

```

24 × 24 font      ..... <r1> =00h <r9> =00h <r17>=00h
code 41h          .....oo.....oo.oo..... <r2> =03h <r10>=06h <r18>=C0h
                  .....o.o.....o.o..... <r3> =02h <r11>=44h <r19>=80h
                  .oooooooo.oooooooo.o..... <r4> =7Eh <r12>=7Eh <r20>=80h
                  .....o..o.....o..... <r5> =02h <r13>=40h <r21>=80h
                  .....o.....o.....o. <r6> =00h <r14>=00h <r22>=82h
                  ..o...o.o...o..oooooooooooo <r7> =22h <r15>=89h <r23>=FFh
                  .ooooooooooooooooo.o.....o. <r8> =3Fh <r16>=FDh <r24>=02h
                  ..o...o.o...o..o..oo.oo. <r25>=22h <r33>=89h <r41>=36h
                  ..o...o.o...o..oo..o.o... <r26>=22h <r34>=8Bh <r42>=24h
                  .oooooooo.ooooo.o...o.o... <r27>=3Eh <r35>=FAh <r43>=28h
                  ..o.oo.o.....o..... <r28>=2Dh <r36>=00h <r44>=20h
                  ....o...o...o.....o..... <r29>=08h <r37>=88h <r45>=20h
                  ...ooooooooooooo.....o..... <r30>=1Fh <r38>=FCh <r46>=20h
                  ...o.....o.....oo..... <r31>=10h <r39>=80h <r47>=60h
                  ..oo.....o..o.....oo..... <r32>=30h <r40>=90h <r48>=60h
                  .o.ooooooooooooo.....ooo.... <r49>=5Fh <r57>=F8h <r65>=70h
                  ...o.....o.....oo.o..... <r50>=10h <r58>=80h <r66>=D0h
                  ...ooooooooooooo...o..oo... <r51>=1Fh <r59>=F8h <r67>=98h
                  ...o.....o.....o..oo.. <r52>=10h <r60>=80h <r68>=8Ch
                  ...o.....o...o..oo..ooo. <r53>=10h <r61>=89h <r69>=8Eh
                  ...ooooooooooooo.o.....ooo <r54>=1Fh <r62>=FDh <r70>=07h
                  ...o.....o.....o.....o. <r55>=10h <r63>=02h <r71>=02h
                  .....o.....o..... <r56>=00h <r64>=04h <r72>=00h

```

The complete <ESC>m command sequence is:

<ESC>mA<r1><r2>...<r13><r72>;

hex code: 1B 6D 41 00 03... 07 02 00 3B

32 × 32 font	<r2>	<r9>	<r17>	<r25>
image map	:	:	:	:
	<r8>	:	:	:
		<r16>	<r24>	<r32>
	<r33>	<r41>	<r49>	<r57>
	:	:	:	:
	:	:	:	:
	<r40>	<r48>	<r56>	<r64>
	<r65>	<r73>	<r81>	<r89>
	:	:	:	:
	:	:	:	:
	<r72>	<r80>	<r88>	<r96>
	<r97>	<r105>	<r113>	<r121>
	:	:	:	:
	:	:	:	:
	<r104>	<r112>	<r120>	<r128>

48 × 48	<r1>	<r9>	<r17>	<r25>	<r33>	<r41>
font	:	:	:	:	:	:
image map	<r8>	<r16>	<r24>	<r32>	<r40>	<r48>
	<r49>	<r57>	<r65>	<r73>	<r81>	<r89>
	:	:	:	:	:	:
	<r56>	<r64>	<r72>	<r80>	<r88>	<r96>
	<r97>	<r105>	<r113>	<r121>	<r129>	<r137>
	:	:	:	:	:	:
	<r104>	<r112>	<r120>	<r128>	<r136>	<r144>
	<r145>	<r153>	<r161>	<r169>	<r177>	<r185>
	:	:	:	:	:	:
	<r152>	<r160>	<r168>	<r176>	<r184>	<r192>
	<r193>	<r201>	<r209>	<r217>	<r225>	<r233>
	:	:	:	:	:	:
	<r200>	<r208>	<r216>	<r224>	<r232>	<r240>
	<r241>	<r249>	<r257>	<r265>	<r273>	<r281>
	:	:	:	:	:	:
	<r248>	<r256>	<r264>	<r272>	<r280>	<r288>

<ESC>m<c><r1>...<rn>;

64 × 64 font image map	<r1>	<r9>	<r17>	<r25>	<r33>	<r41>	<r49>	<r57>
	:	:	:	:	:	:	:	:
	<r8>	<r16>	<r24>	<r32>	<r40>	<r48>	<r56>	<r64>
	<r65>	<r73>	<r81>	<r89>	<r97>	<r105>	<r113>	<r121>
	:	:	:	:	:	:	:	:
	<r72>	<r80>	<r88>	<r96>	<r104>	<r112>	<r120>	<r128>
	<r129>	<r137>	<r145>	<r153>	<r161>	<r169>	<r177>	<r185>
	:	:	:	:	:	:	:	:
	<r136>	<r144>	<r152>	<r160>	<r168>	<r176>	<r184>	<r192>
	<r193>	<r201>	<r209>	<r217>	<r225>	<r233>	<r241>	<r249>
	:	:	:	:	:	:	:	:
	<r200>	<r208>	<r216>	<r224>	<r232>	<r240>	<r248>	<r256>
	<r257>	<r265>	<r273>	<r281>	<r289>	<r297>	<r305>	<r313>
	:	:	:	:	:	:	:	:
	<r264>	<r272>	<r280>	<r288>	<r296>	<r304>	<r312>	<r320>
	<r321>	<r329>	<r337>	<r345>	<r353>	<r361>	<r369>	<r377>
	:	:	:	:	:	:	:	:
	<r328>	<r336>	<r344>	<r352>	<r360>	<r368>	<r376>	<r384>
	<r385>	<r393>	<r401>	<r409>	<r417>	<r425>	<r433>	<r441>
	:	:	:	:	:	:	:	:
	<r392>	<r400>	<r408>	<r416>	<r424>	<r432>	<r440>	<r448>
	<r449>	<r457>	<r465>	<r473>	<r481>	<r489>	<r497>	<r505>
	:	:	:	:	:	:	:	:
	<r456>	<r464>	<r472>	<r480>	<r488>	<r496>	<r504>	<r512>

<ESC>H<hh1>...<hhn>;

Prints hex code character in downloaded font table.

Prototype <ESC>H<hh1>...<hhn>;

Parameters

<hhx>

The hex number string from 00 to 7F:

- 00 prints character code 00h by using current font table and size.
- 0C prints character code 0Ch.

Several hex codes can be concatenated as a string of hex code.
; and 3Bh indicate the end of the hex code string.

Example

hex code: <ESC>153;ABC<ESC>H0C03;DEF<ESC>150;XYZ<LF>
1B 6C 35 33 3B 41 42 43 1B 48 30 43 30 33 3B 44 45 46 1B 6C 35 30 3B 58 59 5A 0A

where,	<ESC>153;	Selects font table 3 and 42 column mode.
	ABC	Prints characters of code 41h 42h 43h.
	<ESC>H0C03;	Prints characters of code 0Ch 03h.
	DEF	Prints characters of code 44h 45h 46h.
	<ESC>150;	Selects built-in font and 42 column mode.
	XYZ	Prints characters of code 58h 59h 5Ah.

Printable codes for the printer device driver are 20h–7Fh for normal operation. However, download fonts can be loaded to any 00h–7Fh code position. To access download fonts at the 00h–1Fh code position, use the <ESC>H; command.

<ESC>GL<f>,<t>,<w>,<h>;<b1>...<bn>;

<ESC>GL<f>,<t>,<w>,<h>;<b1>...<bn>;

Downloads a graphic image into memory.

Prototype

<ESC>GL<f>,<t>,<w>,<h>;<b1>...<bn>

A comma (,) is required to separate the parameters. For the image byte, the bits crossing the image width are truncated. For image widths not a multiple of 8 bits, the last image byte of a row should pad zeroes at right to form a byte. The printer device driver waits until the entire logo image byte count is received. There is no time-out during the wait for the logo image byte.

On Verix eVo-based terminals, the logo data storage area is immediately updated when a download logo command is received. This can lead to unpredictable results if a logo downloads while a previous logo is being rendered for printing. To avoid potential problems, the application must ensure that the printer is not busy by calling `get_port_status()` before downloading a logo. Similarly, downloaded fonts immediately update the font table in memory and results can be unpredictable if the font memory being downloaded is currently referenced in data being rendered for printing.

Parameters

<f>	Verix eVo-based terminals ignore this field. <ul style="list-style-type: none">• 2 is 8-bit format (see Figure 6).
<t>	Image ID (see *PRTLGO).
<w>	The image width; acceptable range 16 to 384.
<h>	The image height; acceptable range is 16 to 240.
<b1>...<bn>	The image data.

Figure 6 presents an 8-bit format.

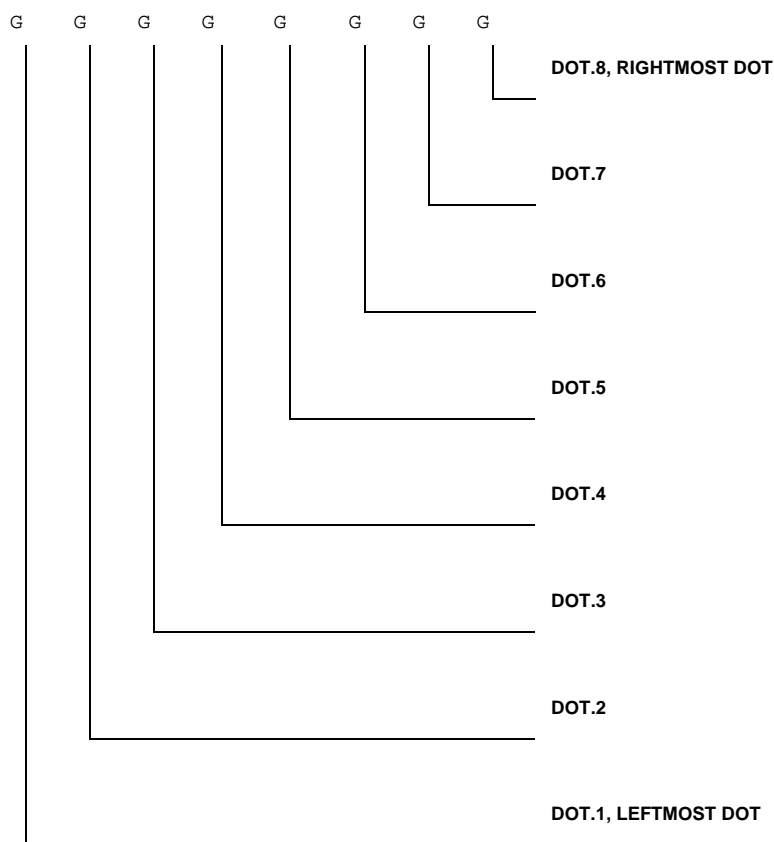


Figure 6 8-bit Format

Example

Image format = 0

Graphic image (logo) = 0

Image width = 128

Image height = 100

Byte count for a image dot line = $128 \div 8 = 16$ bytes

Total bytes for the image = $16 \times 100 = 1600$ bytes

<ESC>GL0,0,128,100;<b1><b2>...<b1600>

<ESC>GP<t>[,<m>;

Prints downloaded graphic image.

Prototype <ESC>GP<t>[,<m>] ;

Parameters

<t>	Image ID (see *PRTLGO).
<m>	Sets left margin for the image (optional); default is 0.

Example

<ESC>GP0,100;	Prints the downloaded image 0, and the left margin of the image aligns on the 101th dot.
---------------	--

NOTE

If there are characters not printed before the print logo command, the character line prints first, then the logo prints below it.

<ESC>w<n>;

Select factor to scale computed print strobe times. The formula is $(\langle n \rangle + 7) / 8$. Higher values extend the thermal head activation times to produce darker printouts.

See also [*DARK](#), which can be used to compensate for lower-sensitivity paper or 2-ply paper, which is considered as Unit Print Darkness.

Prototype

<ESC>w<n>;

Parameters

<n> Sets scale factor to $(\langle n \rangle + 7) / 8$. The allowed range for <n> is: 1-29 for Trident terminals.

Table 38 shows sample scale factor.

Table 38 Sample Scale Factor

<n>	Scale Factor
1	1.000
2	1.125
3	1.250
4	1.375
5	1.500
6	1.625
7	1.750
8	1.875
9	2.000
13	2.500
17	3.000
21	3.500
25	4.000
29	4.500

<ESC>CS;

Retrieves firmware checksum and version.

NOTE



Verix eVo-based terminals return a fixed string constant.

Example

Host	Printer Device Driver
<ESC>CS;	---
	<---- ID ACT CAL<CR><LF>
	-- ---- ----<CR><LF>
	01 3B1F 3B1F<CR><LF>
	<CR><LF>
	SW Version : 0PRED1A1<CR><LF>
	SW Build Date : August 7 2003<CR><LF>
	SW Build Time : 15:11:15<CR><LF>

Dot Graphics Mode

In dot graphics mode, the host has almost complete control over the mechanism and can print dots in any dot positions. For example, dot graphics mode can be used for landscape printing. The horizontal and vertical print density is 8 dots per mm.

In dot graphics mode, printable characters are subdivided into two groups of characters:

- Printable bit patterns
- Terminators

The wide variety of line terminators determine the way that received bit patterns print.

Printable Images

Graphic images are constructed one dot line-at-a-time in one pass. Paper feeds one dot-line after one line of image prints. The data for the image is presented sequentially in 6-bit increments. Bit 8 depends on parity; bit 7 is always 1; the remaining bits are the graphic-image bits. For graphic image bits, bit 6 is the leftmost bit and bit 0 is rightmost. The first code sent represents the leftmost carriage position, the last character the rightmost carriage position, and so on.

Due to mechanism configuration, the image data format is constructed as 384 dots per dot line. The host can send a maximum of 64 image code per dot-line, and one terminator code.

NOTE



Image code must *not* be less than hex number 40; the terminator *must not* be less than hex number 20.

Figure 7 presents the graphic image code.

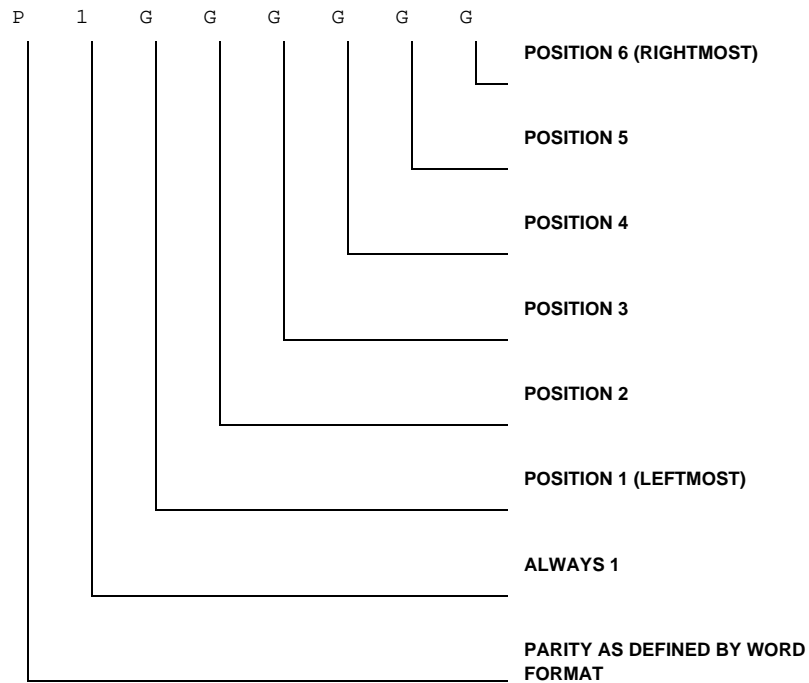


Figure 7 Graphic Image Code

Graphic Line Terminators

Once the graphic data are loaded, the dot line portion can print by sending a terminator character constructed as shown in [Figure 8](#). To exit dot graphics mode without printing, issue the CAN command. It is not a print requirement that the graphics buffer (which holds up to 384 bit image codes) be completely filled.

Figure 8 presents the graphic mode dot line terminators.

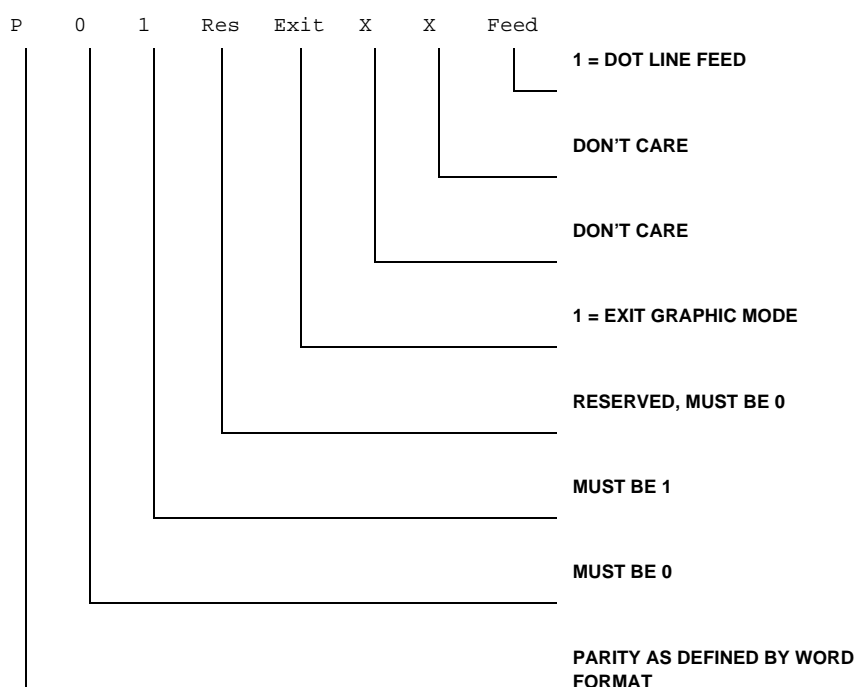


Figure 8 Graphic Mode Dot Line Terminators

Download Fonts and Logos

Download features must utilize all 8 bits of information. Set the communication link to the printer for 8-bit mode to enable download features. All code, from 00h to FFh, can be used as data bytes.

By default, the printer device driver allocates 64 KB of memory for storing downloaded fonts, and 12 KB for storing the logo image received from the host. Refer to *PRTFNT and *PRTLGO for more information.

NOTE



After a download font or logo command, if there is anything that has not printed it is cleared.

Built-In Fonts

The built-in fonts for the printer are referenced as font table 0 and three character sizes are available – 5 x 8, 8 x 14 in 32-column mode, and 8 x 14 in 42-column mode. The 5 x 8 character size is always printed double height and double width resulting in 24 columns. Unless otherwise selected, the default is 8 x 14 in 42-column mode.

In the built-in fonts, characters 0x20 – 0xFF are defined. The standard ASCII character set is defined in the printable range, 0x20 – 0x7E.

The built-in fonts are stored in the PRTFONT.PFT file. This file contains the 5x8 font followed by the 8 x 14 font and is organized as follows:

- Offset 0 – 7 : 5 x 8 font character 0

- Offset 8 – 15: 5 x 8 font character 1
- ...
- Offset 2040 – 2047 : 5 x 8 font character 255
- Offset 2048 – 2061 : 8 x 14 font character 0
- Offset 2062 – 2075 : 8 x 14 font character 1
- ...
- Offset 5618 – 5631 : 8 x 14 font character 255

Download Fonts (User-Defined Characters)

In the printer device driver there are the following two built-in fonts:

- 5 × 8 used in 24 column mode
- 8 × 14 used in 32 or 42 column mode

Every 1 KB of memory constitutes a table. Therefore, 64 KB contains 64 tables. For 5 × 8 fonts, each table can store *one* set of codes from 00h to 7Fh. For 8 × 14 fonts, one set of codes are stored in *two* tables.

There are three commands for the download fonts feature:

- <ESC>l<s><t>; selects a font table for printing and downloading

Character Size	Number of Font Tables
5 x 8	1
8 x 14	2
16 x 16	4
24 x 24	9
32 x 32	16
48 x 48	36
64 x 64	64

- <ESC>m<c><r1>...<rn>; downloads fonts into the memory
- <ESC>H<hh1>, ..., <hhn>; prints hex code characters

The printer does not prevent fonts from being partially overwritten. For example, a 16 x 16 font can be downloaded into table 1 and it will actually be stored in tables 1-4. Later, a 5 x 8 font can be downloaded to table 2 and a portion of the 16 x 16 font are overwritten.

On Verix eVo-based terminals, the thermal printer is not a separate processor. Using the current printer download font command, the memory used to store a downloaded font is part of the Verix eVo memory available to the OS and applications. If the application already has the font in a file, the font is actually duplicated on the system.

NOTE

Verix eVo-based terminals store the font in the memory and the font table is re-initialized on each terminal restart. Previous printers stored the fonts in flash memory and the fonts were retained through power cycles.

On Verix eVo-based terminals, as download font character commands are received, the corresponding font table is immediately updated. This may lead to unpredictable results if characters are downloaded while data is still being rendered. To avoid potential problems, application should ensure the printer is not busy ([get_port_status\(\)](#)) before downloading.

Download Logo Images

The printer device driver supports one command that allows downloading of logo images into the memory, and another command to print the downloaded image without resending the graphic image data.

Support for Paper-out LED

On some terminals, LED is used to indicate the printer's status. The LED blinks at a frequency of about 4Hz if there is a mechanism error. If there is no paper, the LED blinks at a frequency of about 1Hz. Otherwise, the LED is left ON and does not blink. This LED is not under application control.

On VX 680 terminal, the battery system controls a red/green LED, which indicates battery/charger and printer error status. [Table 39](#) lists the combination of printer and battery conditions, and the corresponding state of the bicolor LED.

Table 39 LED State on Different Printer and Battery Conditions

Condition	Red/Green LED
Sleeping	Green, blinks every 4 s.
Battery low	Red, 4 Hz flash.
Paper out	Red, 1 Hz flash.
Printer fault ^a	Red on solid.
Battery Charging	Orange ^b , 1 Hz flash.
-	Green ^c on solid.

a. Printer fault usually means that the printer mechanism has overheated.

b. When both red and green are turned on, the result is orange.

c. When the terminal is turned on, and none of the first five conditions is true, the green LED is solid on.

NOTE

When the terminal is turned off, both LEDs are off.

SDIO

The OS supports the single bit mode (SD-1) of SDIO communication. When an SDIO device is present in the SD slot, the OS automatically detects the condition, and verifies if the device in the SD slot is the SDiD 1050 device. The SDiD 1050 device returns a unique ID information in response to the OS initialization commands.

NOTE



The SDIO device in Vx810 CTLS must be inserted prior to powering up the terminal. Removal while powered on may result in damaging the SDIO card. The SDIO solution in Trident is completely different.

SDIO Device Firmware Download

The Contactless device contains firmware that manages the Contactless card protocol for the Vx810 CTLS only. The Trident CTLS solution is entirely different and is documented separately in the *Verix eVo CTLS Programmers Guide*, VPN DOC00314.

Automatic Firmware Download

The Contactless device firmware can be automatically updated by the Vx810 OS similar to how the modem profile and printer firmware are downloaded. The OS detects and download the Contactless device firmware file during the boot process.

NOTE



The contactless firmware download is not a frequent event as the contactless firmware must first be certified by the card associations before it can be used for payment transactions.

If the file `VIVOCTLS.FRM` is present and authentic, and the SDIO card is present, the VTM and the applications are not allowed to run until the firmware download is complete. There are several devices in the Vx810 system that can be upgraded through a firmware download. These are controlled by their individual drivers. If more than one firmware is loaded into Vx810 at the same time, each device driver will start a firmware download as soon as it loads at boot up. The OS status screen displays the status of the first driver to load and start firmware download. When it finishes ahead of the other downloads, the OS status screen switches to display the status of other downloads that are still running. Firmware files are downloaded at the same time, even if the status screen only displays a single download status.

When downloading firmware files, the following conditions must apply:

- 1 The file must be named `VIVOCTLS.FRM`.
- 2 The file must be downloaded into GID 1 RAM.
- 3 The file must be signed using the OS certificate.
- 4 If the file `VIVOCTLS.FRM` is present and authentic, and if the CTLS card is present, the OS loads it into the contactless device.

The process takes approximately four minutes and the following message is displayed:

```
CTLS Firmware
Download in
Progress
Blocks:XXXXX
```

“Blocks” is updated, which indicates that the firmware load is proceeding and that the terminal is not locked up.

- 5 When download is complete, the OS continues with the boot process and runs the application—no messages are displayed on the screen. However, if no application is present, “DOWNLOAD NEEDED” is displayed. The firmware file `VIVOCTLS.FRM` is retained until a successful download is completed; it is deleted the first time `DEV/CTLS` is opened after a successful download. Another firmware file will not be downloaded to the terminal until `DEV/CTLS` has been opened after the successful firmware download.

NOTE

`VIVOCTLS.FRM` is not a protected file and can be deleted using the VTM memory functions—`DEV/CTLS` must still be opened after a successful download before another firmware is accepted.

If the firmware download fails, the OS displays the following error message:

```
CTLS Firmware
Download Error
Any Key Continues
```

The firmware is retained and another download is attempted the next time the Vx810 reboots with the CTLS connected.

- 6 The file `VIVOCTLS.FRM` is downloaded into the Vx810 using any supported download method — USB Memory Card, VeriCentre, direct download, etc.
- 7 `VIVOCTLS.FRM` file versions are managed similar to how modem profiles are managed. Each version is kept in a .ZIP file named for the unique `VIVOCTLS.FRM` file version it contains. When the .ZIP file is downloaded into the terminal, the `VIVOCTLS.FRM` file is then extracted and downloaded.
- 8 If `VIVOCTLS.FRM` fails to load, the application cannot open the contactless device. This is similar to a missing or corrupt modem profile.

NOTE



User can re-load the same, or a different version of `VIVOCTLS.FRM` as many times as necessary to get a successful firmware download.

CAUTION



Users are cautioned not to design this API into their application architecture because ViVOpay firmware is not distributed in the format that can be downloaded using said API.

SDIO API

The OS presents the standard device API to the application. The application issues the normal device commands, `open()`, `close()`, `read()` and `write()`, to the SDIO device.

API commands unique to the contactless device are provided.

- `reset_ctls()`
- `set_event_bit()`
- `get_event_bit()`
- `get_sd_device_bits()`

CAUTION



Use of contactless APIs in platforms other than the Vx810 can cause the OS to crash with a System Error because there is no code linked to these function pointers.

Device Ownership

The OS provides the standard Verix eVo `set_owner()` and `get_owner()` API for the Contactless device.

Device Firmware Version

The OS can use the VivoPay command, `Get ViVOpay Firmware Version`, described in *ViVOpay Serial Port Interface Document*, to determine the current contactless firmware version.

open()

The application accesses the device using the name `DEV_CTLS` (`"/DEV/CTLS"`). The call `open(DEV/CTLS)` returns a valid device handle when the contactless device is present and operating. If no operating contactless device is detected, the call returns -1 with `errno` set to `EBADF`. This is a normal value for a terminal operating without the contactless device, and if the application attempts to open the contactless device in a platform that does not support it.

If an SD memory card or an MMC memory card is inserted in the SD slot and the application attempts to open it, the OS returns -1 with `errno` `EBADF`. If any SDIO card other than the Vx810 CTLS reader is inserted in the SD slot, and the application attempts to open it, the OS returns -1 with `errno` `EBADF`.

read() and write()

Data transfer to and from the SDIO contactless device does not use the common OS buffer pool defined by the `*B CONFIG.SYS` parameter. Unbuffered transmit data is immediately sent to the terminal. Receive data is held in a local 1024byte FIFO.



SD Communication is 1/2 duplex and the Vx810 CTLS device does not initiate communication with the Vx810. It only responds when the Vx810 requests for information.

If the application attempts to send data to the terminal at the same time it is attempting to send data to the application, either of the transmit or receive data will be corrupted.



`*B` communication device buffer maintains a set of memory buffers for communication device I/O operations. Limiting `*B` may cause unknown side effects—under certain conditions, a known side effect of setting this variable is the failure of modem profiles to load properly. This `CONFIG.SYS` variable is meant for use in applications where memory is limited and should not be set if memory is not limited.

set_opn_blk() and get_opn_blk()

A subset of the standard Verix eVo `set_opn_blk()` and `get_opn_blk()` API is supported for the CTLS SD UART (this does not affect the ViVo UART). The SD UART only supports baud rate settings—bit length, parity, and stop bit settings are ignored. However, the OS saves the entire open block passed through `set_opn_blk()` and returns the saved data in the `get_opn_blk()` call, similar to the printer device.

Open block data is stored in non-volatile memory. The open block settings are retained through power cycles. When the device is opened, the OS uses the most recent open block setting. The application retrieves this setting by calling `get_opn_blk()` after opening the device.

The OS uses the default factory settings until an application calls `set_opn_blk()` with a different setting. The OS retains the new setting until it is changed by another application call to `set_opn_blk()`.

NOTE



Both the SD UART and ViVO UART share the same factory default settings, {Rt_19200, Fmt_A8N1|Fmt_auto|Fmt_RTS, P_char_mode, 0}, which is returned by `get_opn_blk()` even if `set_opn_blk()` is not called.

the application requests rates other than 9600, 19200, 38400, 57600, and 115200, which the ViVO firmware supports, the OS returns -1 with `errno` set to `EINVAL`.

CAUTION



Any change to the SD UART settings can render the terminal inoperable. The `set_opn_blk()` API in Vx810 OS configures only the SD UART, while the ViVO firmware command, `Set Baudrate`, configures only the ViVO UART.

The Vx810 CTLS retains both the ViVO UART and the SD UART settings over power cycles. In order to ensure constant communication between the application and the Vx810 CTLS device, the application must verify the ViVO UART setting immediately after opening the Vx810 CTLS device by sending a PING command without changing the SD UART setting. If successful, the application can proceed to use the Vx810 CTLS device. Otherwise, the application should send a PING command using each supported baud rate until a successful PING response is received. This is done because multiple applications are allowed to use any device in a Verix eVo terminal, and the potential that any application can change the SD and ViVO UART settings (including applications invoked by VTM such as Debugger, Remote Diags, Contactless Diags, and application launcher functions) greatly exists.

get_port_status()

To provide compatibility with other communication devices, `get_port_status()` API is supported. Returned Byte 1 and Byte 3 contain the standard information for this API, while returned Byte 2 and Byte 4 is always 0 for the contactless device.

NOTE



The Vx810 does not buffer transmit data, but sends it immediately. Applications can use either or both the `get_port_status()` and `set_event_bit()` APIs to manage data receive functions.

Contactless Power

When the Vx810 PIN pad is powered up, the Contactless device is also powered up. The OS does not have the ability to control the Contactless device power independently from the Vx810 PIN pad power.

reset_ctls()

Reboots the contactless device micro-controller. This puts the contactless device in the same state as it is immediately after the `open()` call. The application is responsible for any necessary contactless device configuration after the contactless device is reset, refer to *ViVOPay Contactless EMV Serial Interface* for more information.

Prototype

```
int reset_ctls(void);
```

NOTE

It is required to `open()` the device before using this API.

The OS returns `-1` with `errno` set to `EBADF` if the application attempts to use this API before calling `open()`.

set_event_bit()

Allows the application to select one of the event bits that would otherwise be unused, and assigns it to the console device.

Prototype `int set_event_bit(int hdl, long flag);`

Parameter

`handle` The Contactless device handle returned from `open()`.
`flag` The bit value of the event bit to use.

For the contactless device, it does not generate events by default. When configured to do so, it only generates an event when data is available for the application to read. The `set_event_bit()` is used if the application wants to receive contactless events and specify which event bit should be used to report contactless events. This mechanism is provided because almost all the bits in the event mask have already been allocated, however, in a specific terminal installation, many of the predefined event bits are not used.

Example

```
set_event_bit(ctlsHdl, 0x80)
set_event_bit(ctlsHdl, 128)
set_event_bit(ctlsHdl, (1L<<7 ))
#define MY_CTLSEVENT 0x80
set_event_bit(ctlsHdl, MY_CTLSEVENT)
```

The above examples all cause bit 7 to be the contactless event. The `set_event_bit()` returns 0 if it is successful.

NOTE



Some event bits are reserved by the OS and are not allowed to be selected by `set_event_bit()`.

The reserved event bits are:

- EVT_USER
- EVT_SHUTDOWN
- EVT_SYSTEM

Return Values

Success 0, and places the device in the default mode - events turned off.

Failure -1 with `errno` set to `EINVAL`, when API is called with:

- One of the reserved event bits.
- A `flag` that defines more than 1 event bit (such as, `set_event_bit(ctlsHdl, 0x90)` returns error).
- A `handle` set to any non-contactless device handle (such as, `set_event_bit(ctlsHdl, 0x90)` returns error).

If the application uses this API to define an event and then calls `close()`, the OS clears the event setting. The application must call `set_event_bit()` each time it opens the contactless device.

CAUTION

The OS does not change the predefined event bits in `svc.h` when `set_event_bit()` is called. If `set_event_bit()` is used to set `EVT_MAG` for the contactless device, the application receives `EVT_MAG` for both the mag card and the contactless device.

The application should clear the event bits using `read_event()` before calling `set_event_bit()`. This prevents confusion if there is a pre-existing CTLS event bit set at the time `set_event_bit()` is called.

get_event_bit()

Allows the application to find out what event will be generated by the console device.

Prototype `long get_event_bit(int hdl);`

Parameter

`handle` The device handle returned from `open()`.

Return Values

Success 0, and places the device in the default mode - events turned off.

Failure -1 with `errno` set to `EINVAL`, when `handle` is set to any device that does not support this API.

NOTE



Verix eVo OS returns -1 with `errno` set to `EBADF` when any API is passed an invalid handle—set to any value other than what is returned by a successful `open()` call.

get_sd_device_bits()

Determines if an CTLS card is present. This does not actively check for card presence but returns the value of the OS SD status word. Each SD device driver sets or clears a device specific bit in the SD status word. The bit is set when the driver detects that an SD device is attached. The bit is cleared when the driver detects that the SD device is removed. The status word always has the current SD device connection status because the SD connector generates an interrupt when as SD Card is inserted or removed.

Prototype `unsigned long get_sd_device_bits(void);`

```
#define SD_FLASH 1<<0
```

```
#define SD_SDID1<<1
```

Return Values

A 32-bit word with the appropriate bit set to indicate which specific SDIO device is connected (if any).

If no device installed in the SDIO connector, returns `OUL`. At this time, there is only one SDIO device defined and supported `SD_CTL`.

NOTE



It is possible for an application to see the `SD_FLASH` bit set if an SD Memory card is inserted in Vx810. However, the OS does not provide any API for the application to interact with the SD memory cards.

Biometric Module (VX 520 GPRS)

The Biometric module is available on VX 520 GPRS terminals, this facilitates in the collection of Biometric data.

The fingerprint sensor uses a UPEK Capacitive (TCE1SM) display module with a 256 x 360-pixel resolution. It is connected to the unit via USB cable with a customized USB plug designed to be locked by the SAM cover.

Libraries

The reader has a defined protocol to communicate between the VX 520 GPRS and the biometric reader. Libraries are built to facilitate the applications' communication to the reader.

Simple Communications Library (SCL)

SCL (or the fingerprint module) — SCL is only an interface without any application logic. It is capable of producing fingerprint templates (enroll) and verifying them (match) in AuthenTec proprietary template format.

There are a number of functions for interfacing with the reader, for more information on user enrollment and verification APIs, refer to the document PTAPI Standard Functions from AuthenTec.

Biometric Data Conversion Library (BCLib)

BCLib is AuthenTec's library, providing access to biometric data conversion functions. This library provides applications with a tool to convert AuthenTec biometric data to ISO/ANSI standard formats and vice versa. The library can also be used for conversion among various data formats used in AuthenTec software, as well as for preparing third party data for internal testing.

For more information on image and template conversion APIs, refer to the document BCLib user Documentation from AuthenTec.

NOTE



The VX Bio Communication and Conversion Library are released as a downloadable zip file. Upon terminal download, it is extracted to the EOS space.

All APIs in SCL and BCL are provided via interface file (`biocc.o`) to applications. The shared library (`biocc.lib`) that this file interfaces will be in `N:15`. All API definitions will be in `"svc_bio.h"`.

NOTE



The libraries does not have any terminal dependent code. As long as `"/dev/bio"` is available as device and `SVC_INFO_BIO(void)` returns 1, these libraries will work on any terminal.

SVC_INFO_BIO()

This returns biometric device used.

Prototype `int SVC_INFO_BIO(void);`

Return Values

0	None
1	AuthenTec/UPEK reader

USB Barcode Scanner

The Heron™ D130 model is a multi-interface POS device that supports RS-232, USB, WEDGE and WAND, and is used on the Vx570 terminal via USB interface. There are different Interface selections under the D130 USB interface — the USB-KBD, USBIBM-Table-Top, and USB-COM. If the interface selection is not set, the default USB interface selection setting is USB-KBD. The Vx570 terminal, however, uses the USB-COM as the interface selection. This sets the USB class protocol to be defined for “communication devices.” USB-COM must be configured as the USB interface selection before attaching it to a Vx570 terminal.

The D130 scans the barcode by pulling the trigger or by correctly inserting the reader into the stand. Code scanning is performed along the center of the light bar emitted from the reading window. This bar must cover the entire code. Successful reading is signaled by an audible tone plus a good-read green spot.

NOTE



This device is supported in any terminal with USB Host capability.

Configuring D130 as USB-COM

Heron D130 is connected via USB cable. It is bus powered at 180mA @5VDC. To configure the D130 as a US-COM, scan the barcode below.



Once configured, the D130 works as a standard RS-232 device with the following configuration settings:

- 9600 baud, no parity
- 8 data bits
- 1 stop bit
- no handshaking
- delay disabled
- rx timeout 5 sec
- ack/nack disabled
- FIFO enabled
- serial trigger lock disabled
- 1 stop bit terminator = CR LF

USB Barcode Scanner API

The OS presents the standard API to the application such as `open`, `close`, `read`, and `write` functions. Since the D130 works as an RS-232 device, the OS driver enables the USB barcode scanner to perform like a regular Verix eVo UART device. This means that `set_opn_blk()`, `reset_port_error()`, `set_serial_lines()`, `set_fifo_config()`, `get_opn_blk()`, `get_fifo_config()`, and `get_port_status()` are available to the application under some limitations. The application is able to access the USB barcode device as `"/dev/bar."`

Below are the descriptions and limitations of the barcode scanner serial APIs.

<code>set_opn_blk()</code>	This returns success since there is no physical UART device.
<code>get_opn_blk()</code>	This returns success if <code>set_opn_blk()</code> is called first, if not, returns <code>-EINVAL</code> .
<code>reset_port_error()</code>	This returns success if <code>set_opn_blk()</code> is called first, if not, returns <code>-EINVAL</code> .
<code>set_serial_lines()</code>	This returns success if <code>set_opn_blk()</code> is called first, if not, returns <code>-EINVAL</code> .
<code>set_fifo_config()</code>	This does not apply to USB devices, returns <code>-EINVAL</code> .
<code>get_fifo_config()</code>	This does not apply to USB devices, returns <code>-EINVAL</code> .
<code>get_port_status()</code>	This only sends "success" or "EACCESS" only if the buffer parameter is an invalid pointer. <code>Set_opn_blk()</code> must be called first before calling <code>get_port_status()</code> . If <code>set_opn_blk()</code> is not called, <code>get_port_status()</code> returns <code>EINVAL</code> .
<code>barcode_raw_data()</code>	This is a Predator call only. The <code>barcode_raw_data</code> function copies the raw barcode buffer to buffer and returns the count.
<code>barcode_pending()</code>	This is a Predator call only. <code>barcode_pending()</code> returns the number of samples for the last scan attempt.

Enabled Codes on Heron D130 Device

The following are code selections enabled on the Heron D130 device. This also includes the data capacity of each barcode symbology.

Table 40 Heron D130 Enabled Codes

Symbology	Data Capacity
UPC - A	12 numeric digits - 11 user specified and 1 check digit.
UPC - E	7 numeric digits - 6 user specified and 1 check digit.
EAN - 8	8 numeric digits - 7 user specified and 1 check digit.
EAN - 13	13 numeric digits - 12 user specified and 1 check digit.

Table 40 Heron D130 Enabled Codes

Symbology	Data Capacity
Code 39	Variable length alphanumeric data - the practical upper limit is dependent on the scanner and is typically between 20 and 40 characters. Code 128 is more efficient at encoding data than Code 39.
Code 128	
	Code 128 is the best choice for most general bar code applications. Code 39 and Code 128 are both very widely used.
Interleave 2 of 5	Variable length numeric data - the practical upper limit is dependent on the scanner and is typically between 20 and 50 characters.

The caller does not know what kind of barcode is read since the Heron D130 device does not send the barcode information. Only the barcode ASCII data is sent.

Operating Test

Scanning the test barcode below returns a “test” ASCII followed by the CR-LF terminator.



USB Keyboard

The USB Keyboard driver uses the USB HID (Human Interface Device) Device Class Definition and works by sending HID commands/indexes. The driver converts these to standard IBM PC “make/break” scan codes. The scan codes are then passed to the application which decodes it to display the correct character.

NOTE



The application is responsible for detecting multiple key presses.

The key debounce is processed on the keyboard HW. The VTM does not support the USB keyboard, and the driver does not support key beeps. The keyboard is marked with Latvian characters, which are supported at application level and not with the OS.

The OS presents the standard API to the application such as `open()`, `set_opn_blk()`, `close()`, `read()`, and `write()` functions.

USB to RS-232 Converter

The USB to RS-232 Converter driver supports the existing USB modem dongle, USB to RS-232 converter cables from Teletec, and ViVo cables used for the Qx120 contactless device. The driver is implemented on COM6.

The following cables are supported:

- Teletec PN 24625-04-R and PN 24625-02-R—used to connect with Pinpad devices such as PP1000SE.
- Teletec P/N E-120-2327-00 REV A—used to connect to Qx120 contactless device.
- Teletec P/N 24440-02-R—this cable has a DB9 on one end and USB host on the other. The DB9 usually connects to PC and the USB port connects to the terminal. This is normally used for PC downloads.
- Teletec P/N 24805-2-R—this cable has an RJ45 on one end and a USB Host on the other. This usually connects to COM1 of a terminal and the USB port to another terminal. This can be used for terminal to terminal communication. This can also be used for PC download if the PC supports RJ45 serial communication.

NOTE



Teletec USB to RS-232 converter cables are not configured to work on Qx120 devices.

- ViVoTech PN 220-2442-00—used to connect with Qx120 Contactless devices.

The VTM reflects a channel for USB to RS232 downloads. This channel is COM6 port similar to the VX 680 terminal.

NOTE



Only one USB to RS232 device (COM6) will be supported at anytime. It is not possible to support VX 680 USB UART and one of the new USB to serial cables at the same time.

The APIs supporting this driver are similar to the current Verix eVo USB UART (COM6) driver with added support to USB to RS-232 module and the converter module of the Qx120 Contactless device. The OS driver enables the USB UART to look like any other Verix eVo UART device but within the limitations of the USB UART hardware.

USB Device Bits

The USB Keyboard and the USB to RS-232 devices contain USB device bits. This allows the application to know what USB devices are connected to the terminal. The definition below are reflected in `svc.h`.

```
#define UDB_COM6      (1<< 4)
```

```
#define UDB_KYBD      (1<< 7)
```

Power The USB Keyboard and the USB to RS-232 devices need a 5V 100mA to power up, which is provided by the Verix V terminal. In this case, no power hub or external power source is needed to power up the USB devices.

USB Keyboard Scan Codes Scan codes are data from a keyboard created by keypresses. [Figure 9](#) illustrates a PC keyboard showing 102 keys.

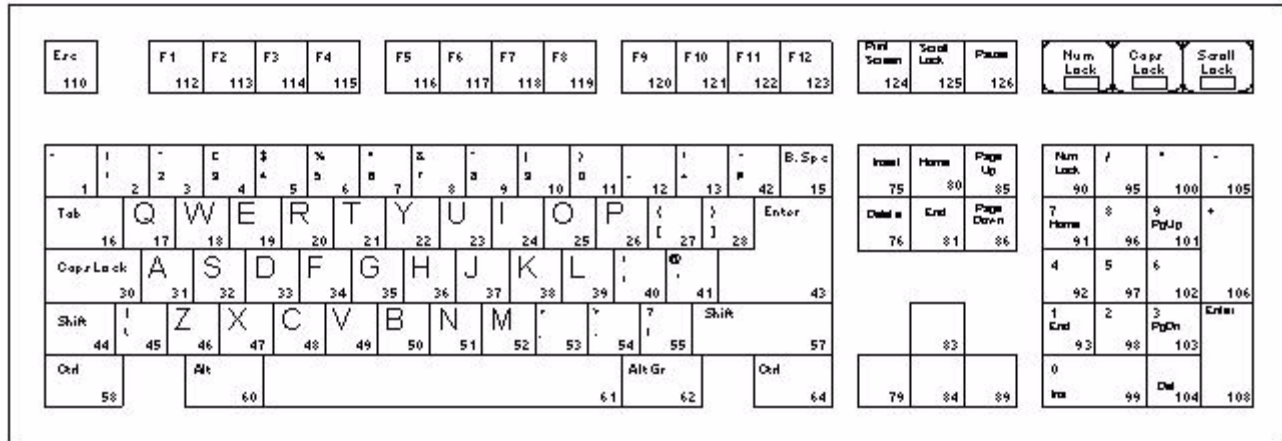


Figure 9 PC Keyboard

There are several scan codes for USB Keyboard such as XT, AT, and MF codes. XT codes are obsolete and MF codes are rarely used. In this OS, only the AT codes are supported. [Table 41](#) displays the scan codes.

Table 41 Scan Code Table

Key Number	AT Code
1	0E
2	16
3	1E
4	26
5	25
6	2E
7	36
8	3D
9	3E
10	46
11	45
12	4E
13	55
15	66
16	0D
17	15
18	1D

Table 41 **Scan Code Table** (continued)

Key Number	AT Code
19	24
20	2D
21	2C
22	35
23	3C
24	43
25	44
26	4D
27	54
28	5B
*29	5D
30	58
31	1C
32	1B
33	23
34	2B
35	34
36	33
37	3B
38	42
39	4B
40	4C
41	52
**42	5D
43	5A
44	12
**45	61
46	1A
47	22
48	21
49	2A
50	32
51	31
52	3A
53	41
54	49
55	4A
57	59
58	14
60	11

Table 41 **Scan Code Table** (continued)

Key Number	AT Code
61	29
62	E011
64	E014
75	E070
76	E071
79	E06B
80	E06C
81	E069
83	E075
84	E072
85	E07D
86	E07A
89	E074
90	77
91	6C
92	6B
93	69
95	E04A
96	75
97	73
98	72
99	70
100	7C
101	7D
102	74
103	7A
104	71
105	7B
106	79
108	E05A
110	76
112	5
113	6
114	4
115	0C
116	3
117	0B
118	83
119	0A
120	1

Table 41 Scan Code Table (continued)

Key Number	AT Code
121	9
122	78
123	7
124	E012E07C
125	7E
126	E11477E1F014 F077
NONE	None

* Only Keyboards with 101 keys - US Keyboard (and others).

** Only Keyboards with 102 keys - UK Keyboard (and others).

Support for Windows Keys

When the Windows 95 Operating System was introduced, three new keys were added to the PC keyboard. These keys have been retained for all subsequent Operating Systems and PCs. They are the two “Flying Windows” keys and the “Pop Up Menu” key. The appropriate “make” and “break” scan codes are shown below.

Table 42 Make and Break Scan Codes

Key	Make	Break
Left Flying Window	E0 1F	E0 F0 1F
Right Flying Window	E0 27	E0 F0 27
Pop Up Menu	E0 2F	E0 F0 2F

Calculating Make and Break Scan Codes

The PC keyboard interface is designed to provide the system software with maximum flexibility in defining certain keyboard operations. This is done by having the keyboard return scan codes rather than ASCII codes. Each key generates a “make” scan code when pressed and a “break” scan code when released.

To Calculate the “Make” and “Break” Scan Code

- 1 Use the keyboard sketches above to determine the “Key Number.”
- 2 Lookup the “Key number” in the table and read the make’s scan code. Note that some scan codes consist of more than 1 byte.
- 3 Calculate the break scan code.

Most PC’s made since 1989 use keyboards that generate AT scancodes. The break code for AT class scan codes is simply the make code preceded by hex F0 (e.g., the scan codes generated when the Escape key is pressed and released are 76 F0 76).

Example Below is an example output of USB Keyboard if the user type is "Hi\n". This example shows how to handle the SHIFT+KEY combination.

Table 43 Shift+Key Combination

AT Code	Key Number	ASCII Code
12	# Keynumber 44 down	SHIFT key down
33	# Keynumber 36 down	H key down
F033	# Keynumber 36 up	H key up
F012	# Keynumber 44 up	SHIFT key up
43	# Keynumber 24 down	i key down
F043	# Keynumber 24 up	I key up
5A	# Keynumber 43 down	ENTER key down
F05A	# Keynumber 43 up	ENTER key up

The keyboard also supports keyboard LED's. The driver manages the LED's such as caps lock, scroll lock, and num lock.

For more information about keyboard emulator, refer to <http://www.barcodeman.com/altek/mule/kbemulator/>.

Metrologic Barcode Scanner

The Verix V terminals supports the Metrologic barcode scanner, a USB Human Interface Device (HID) that uses the USB keyboard driver available in most Verix eVo terminals. This device converts USB HID raw data to ASCII if the Metrologic scanner is connected.

NOTE



The MS9590-106 VoyagerGS unit is used, which supports USB as its default communication protocol. The 10-pin RJ45 end of the cable is plugged into the MS9590-106 unit, while the other end containing the Type A USB is plugged into the host USB port.

Configuring the Barcode Scanner

The Metrologic barcode scanner should be set up before running any barcode application on a Verix V terminal. To set up the keyboard driver for Metrologic barcode scanner:

- 1 Set the Metrologic barcode scanner to USB HID configuration.
- 2 Set *KEYBOARD environment variable to either 2 or 3 depending on what the application requires.

Keyboard Driver Data Output

The keyboard driver can be set to support different data outputs. This is enabled by the environment variable, *KEYBOARD to be set at GID1.

Make/Break Scan Code Output

For keyboard devices that require a make/break scan code output data, which is the default configuration setting, set *KEYBOARD=1 to enable the driver to output make/break scan codes.

ASCII Code Output for Barcode Devices

To configure the keyboard driver support ASCII code output, set `*KEYBOARD=2`. This enables the driver to buffer the ASCII code output and place an event once CR or LF is received. The setting returns the complete single barcode string including the delimiter on a single read even if the length value passed is bigger than the barcode data.

ASCII Code Output for Keyboard/Barcode Devices

To configure the keyboard driver to support ASCII codes per character, set `*KEYBOARD=3`. This setting saves the ASCII codes to a FIFO and triggers the driver to send an event once data is received from the keyboard/barcode device.

Unlike the setting `*KEYBOARD=2`, this returns the barcode ASCII data per character in queue in the FIFO buffer. The barcode suffix set is treated as part of the data.

NOTE



Unlike in the setting `*KEYBOARD=3`, when multiple barcode data is passed, the `*KEYBOARD=2` returns the first barcode string on a first read even if the length of the data passed on `read()` is greater than the barcode data.

Example

When `*KEYBOARD=2` is set and the user fires two barcode data:

Barcode 1 = 1234567CR

Barcode 2 = abcdefgCR

First read.

`read(handle,buf,20)` returns 8 with a value of 1234567CR.

Second read.

`read(handle,buf,20)` returns 8 with value abcdefgCR.

Example

When `*KEYBOARD=3` is set, and the user fires two barcode data, the requested data is returned based on the length parameter passed on `read()`.

Barcode 1 = 1234567CR

Barcode 2 = abcdefgCR

First read.

`read(handle,buf,15)` returns 15 with a value of 1234567CRabcdefg.

Second read.

`read(handle,buf,20)` returns 1 with a value of CR.

Metrologic USB Barcode Scanner API

The OS presents the standard API to the application such as `open`, `close`, `read`, and `write` functions.

- If the device is not connected, `open("/DEV/usbkbd", " 0")` returns `-1` and `errno` is set to `EBADF`.

- If the device is removed after it is opened, API functions `read()` and `write()` return -1 and `errno` set to `ENXIO`.

If the application does not close the device, communication to the device may not be established when the Metrologic barcode scanner is subsequently connected. The application must close and reopen the device.

Events

Since the device uses the keyboard driver, the [MC5727 CDMA Radio](#) API is employed to set the event if there is an incoming data from the barcode device. This allows the application to select one of the event bits that would not otherwise be used and assign it to the Metrologic barcode scanner device.

Processing Events

Once event bit is set to Metrologic device, this signals the application that there is an incoming data.

- The driver receives an interrupt on every character data from the device.
- The driver queues the data to a FIFO buffer and send an event to the application informing that there is already barcode information to read.
- When multiple barcode is fired and data is not read, the driver will queue the data to its FIFO buffer. The application can still read all the barcode data that successfully stored to the FIFO buffer. The maximum character that can be stored to the FIFO buffer is 300 characters.

Example

```
hBarcode = open(DEV_KYBD, 0);
set_event_bit(hBarcode,EVT_BAR); // Setting EVT_BAR as the event
do {
    SVC_WAIT(5);
} while(!(read_event() & EVT_BAR)); // Wait for incoming data
retval = read(hBarcode, buf, 15); // Read the barcode data
```

Device Bits

The USB Keyboard device contains USB device bits. This allows the application to know what USB devices are connected to the terminal.

```
#define UDB_KYBD      (1<< 7)
```

This bit is reflected in `get_usb_device_bits()` API if the barcode scanner is connected and will be out if the barcode scanner is disconnected.

Power

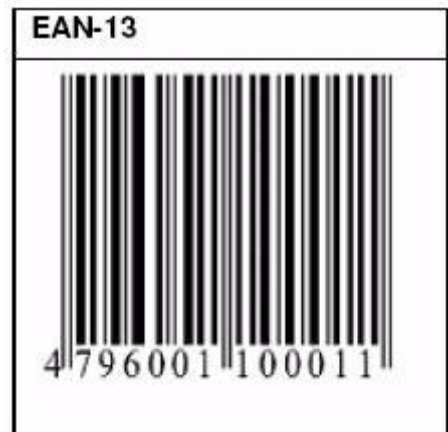
The Metrologic barcode scanner device has a maximum power of 450mA, and since the Verix V terminals provides the standard USB Host power of 5V 500mA, the device does not need an external power or a powered hub to work.

Operating Test

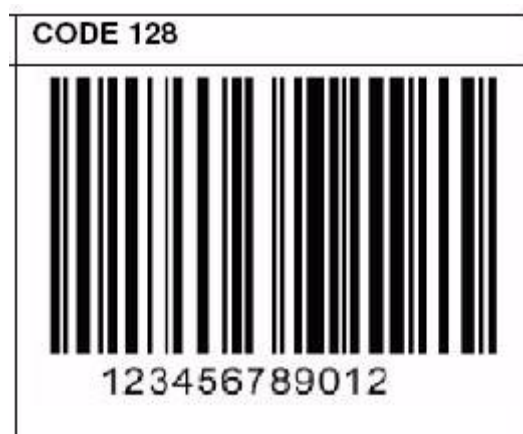
The Metrologic barcode scanner device driver in Verix eVo operating system is only a channel to get the data from the barcode device. The OS does not handle the configuration of Metrologic barcode scanner.

To configure the barcode scanner, the user can check the manual or download the configuration guide from the Metrologic website.

However, some of the popular barcode types are include in this document for initial testing (refer to the configuration manual for other barcode types supported on this device).



More barcode types are shown below:



Scanning the recall defaults barcode erases all the previous setting and return the scanner to its default communication protocol.

USB Device Driver APIs

The OS presents standard APIs to the application such as `open()`, `close()`, `read()`, and `write()` functions.

USB Device Driver Events

The Verix eVo operating system bit mask is completely allocated. However, not all events are wanted by an application so a mechanism is created to allow the user to overload a event bit. A new API is introduced to set an event `set_event_bit()`.

By default, the USB Keyboard does not generate events but it can do so if data is available for the application to read. If the application wants to receive keyboard events, it must use the API `set_event_bit()` to specify which event bit to use to report keyboard events.

The following are events that the OS issues to the application once the device is used:

Table 44 USB Driver Events

Event	Description
EVT_USB	Issued upon connecting/disconnecting them from the terminal.
EVT_BIO	Issued when there is incoming data from the fingerprint reader.
EVT_BAR	Issued when there is an incoming scanned barcode data.
EVT_COM6	The USB to RS-232 device generates EVT_COM6 if there is incoming data from the device.

If the application wants to receive keyboard events, it must use the new API `MC5727 CDMA Radio` to specify which event bit to use to report keyboard events. This is done because almost all the bits in the event mask have already been allocated—in some terminal installation, many of the predefined event bits are not used.

MC5727 CDMA Radio

The MC5727 driver provides CDMA EVDO radio modem connectivity. It is supported on COM2 and COM9.

- COM2 is the AT command and bulk data port.
- COM9 is used for CnS traffic.

Each provides an interface comprised of write, read, control, status, close, and open function for each device according to its functionality. Each COM port may be opened independently.

AT and Data Port (COM2)

Data and AT commands are exchanged with the MC5727 over USB Endpoint 2. The user opens and reads/writes to COM2 (USB COM2) to send and receive data on the CDMA radio modem link. The `read()`, `write()`, `open()`, `close()` functions are supported.

Status and Control Port (COM9)

Open and write to COM9 to send and receive CnS messages. Set the open block after opening. The Verix eVo libraries encapsulates/decapsulates the proprietary HIP protocol and CnS messaging.

The driver expects the user to set open block after opening. If `set_opn_blk()` is called for the second port opened, the user will receive a successful response. Functions `read()`, `write()`, `open()`, `close()`, `control()` and `status()` are supported for both ports.

NOTE



Refer to the corresponding Tools and Libraries FRD for more information.

Event Handling

Events are handled on COM9 by user-defined events function. COM2 uses `EVT_COM2`.

The following calls are in addition to the standard event API. The user will continue to use `read_event()` and other events for normal event handling. The following two calls are used to choose the event bit the OS will return to the user on a COM9 event. The device must be opened before the event bit is set.

- `int set_event_bit (int hdl, long flag);`
- `long get_event_bit (int hdl);`

Where:

`hdl` is the return from `open()` and `flag` is a long int where the event bit is set.

Returns:

- `set_event_bit` returns 0 on success.
- `get_event_bit` returns `EVENT`.

The reserved bits are as follows:

- `EVT_USER`
- `EVT_SHUTDOWN`
- `EVT_SYSTEM`

If the application calls `set_event_bit()` with one of these reserved event bits, `set_event_bit()` will return -1 with `errno` set to `EINVAL`.

WARNING



The OS will not change the predefined event bits in `svc.h` when the application calls `set_event_bit()`. For example, if the application uses `set_event_bit()` to set `EVT_MAG` for COM9 device, the application will receive `EVT_MAG` for both the mag card and the COM9 device.

NOTE



The MC5727 is powered internally from the terminal and not from the USB power budget so the device always returns 0 mA in the Power field.

APIs MC5727 supports the following commands used for serial ports:

- `int set_opn_blk(int hdl, const struct Opn_Blkc *ob);`

The following table lists the values that are defined for the MC5727. Other values may be accepted by the radio and ignored.

Table 45 Open Block Field Definitions

open_block_t field	Values
rate	All Verix eVo-defined rates, except Rt_user_defined.
format	Fmt_A7E1, Fmt_A7N1, Fmt_A7O1, Fmt_A8E1, Fmt_A8N1, Fmt_A8O1, Fmt_DTR, Fmt_RTS, Fmt_2stp
protocol	p_char_mode
parameter	Currently unused. No checking is done.

NOTE



Fmt_auto and Fmt_AFC cannot be set at the same time. Similarly, Fmt_RTS and Fmt_AFC also cannot be set at the same time.

The `set_opn_blk()` is unique for each of the two endpoints, endpoint 2 (COM2) and endpoint 4 (COM9). If `set_opn_blk()` is issued for COM9 after opening the port, it need not be set when opening the other port. If called after the second endpoint 4 `open()`, the OS will ignore the command and return success.

- `int set_serial_lines(int hdl, const char *mask);`

The input value is a bit mask. Bit 0 represents DTR, bit 1 represents RTS.

- `int reset_port_error(int hdl);` and `set_fifo_config();`

Both return 0.

- `int get_opn_blk(int hdl, struct Opn_Blkc *ob);`

Returns the currently defined values in the `open_block_t`.

- `int get_port_status(int hdl, char *buf4);`

Returns 4 bytes of data.

Table 46 get_port_status() Return Values

open_block_t field	Values
1	Number of messages in the read fifo.
2	0
3	0
4	Errors in a bit mask.

NOTE



A bit mask where b0=parity, b1=overrun, b2=framing, b3=DCD, b4=ring signal, b5=0 (not supported), b6=DSR, b7=break.

- `Int usb_pending_out(int hdl);`

Returns the number of messages in the write queue waiting for transmission.

- `Int get_fifo_config(int hdl, char *buf);` and `int get_fc_config(int hdl, char *buf);`

Both return 0.

- `Int set_radio_ctl(int hdl, char *sigs);`

Where bit 0= RAD_MOD, bit 2=RAD_OFF.

Table 47 **set_radio_ctl() Input Values**

BIT=1		BIT=0
RAD_MOD bit0	Hardware switch active	Normal state
RAD_OFF bit2	Normal / powered up	Powered off

The table below defines the resulting actions for MC5727 radio.

Table 48 **Resulting actions for MC5727 radio**

RAD_MOD bit 0	RAD_OFF bit 2	Meaning	Expected OS Action
1	1	Activate HW switch and power up radio	Activate HW switch
1	0	Activate HW switch and power off radio	Activate HW switch
0	1	USB normal connection and power up radio	Power up the terminal if the USB handle
0	0	USB normal connection and power off radio	Power off the radio

RAD_MOD and RAD_OFF are mutually exclusive. If RAD_MOD is set to 1, it will take precedence over power off (RAD_MOD = 0). It is recommended that the user call USB_COM2_RESET() to power up the radio when it is powered down. If the USB handle has been invalidated by the USB host when the set_radio_ctl() is made, the call will return an error.

- `int get_radio_sts (int hdl, char *sigs);`

Where sigs is a bit mask where bit 0= RAD_INT, zero (0) if powered, one (1) if not powered.

- `int USB_COM2_RESET (void);`

Toggles the power on the MC5727 off and on again. This function is only valid on MC5727. If other modules are installed on COM2, this call will return `EINVAL`.

NOTE

The user is expected to close all ports before calling `USB_COM2_RESET`.

The hardware will not power off until 10.5 seconds after system startup. The power up command is issued 600 milliseconds after the power off command. User commands to the radio may be issued after the power up is complete.

The call `set_radio_ctl()` can be used to power off the radio. However, after a device is closed, the Verix eVo USB architecture does not allow the user to use the device handle. To power up the radio, use `USB_COM2_RESET()`.

This call does not reset `RAD_MOD` to disable the hardware switch. The unit must be rebooted to reset `RAD_MOD`.

VX 520 color terminals use the same display and keypad drivers as VX 675, the same radio driver as VX 680 3G (HSPA +), and dual SIM.



Communications

This chapter introduces the communications devices and protocols used on VeriX eVo-based terminals and PIN pads, and discusses protocols used in most VeriX eVo applications. The communication ports on these terminals must be opened, configured, used, and then closed. Configuration of a port uses the `Opn_Blks()` structure.

The `Opn_Blks()` Structure

The data structure `Opn_Blks` (defined in the `svc.h` file) is used by an application to specify the baud rate, data format, and other protocol-specific parameters on VeriX eVo-based terminal communications devices. Settings are initialized or reset using a `set_opn_blk()` call. The structure is passed to the device through the `buf` parameter. This call must be made prior to any device read or write operations. The current structure can be obtained by calling `get_opn_blk()`.

Character Mode

In character mode, all inputs and outputs are treated as individual bytes; the data are not handled as a packet. From a system standpoint, character mode data is simply a stream of input and output bytes. No data validation or additional processing is performed. All intelligence must reside in the application.

Character Mode Initialization

The following linked example code illustrates a character mode `Opn_Blks` initialization call.

Example

Click the linked example to view the sample code.

Simultaneous Communication Channels

On VX 680 3G, the OS supports the ability to simultaneously use BT/Ethernet, BT/Dial-up, WiFi and 3G/GPRS network interfaces with TCP/IP usage on 1, 2, 3, or all 4 network interfaces. When TCP/IP is used on two or more network interfaces from the list of BT/Ethernet, BT/Dial-up, WiFi and 3G/GPRS, the OS provides the ability to force a socket connection onto a specific network.

The OS supports simultaneous usage of Ethernet dongle, Dial dongle, or Ethernet dongle with Dial dongle with one or more radio interfaces, including simultaneous usage of TCP/IP stack.

The 3G connection will not be accessed by the tablet or by any terminal application that is communicating with tablet.

Communication Ports

There are several communication ports in Verix eVo terminals. These ports generally use a common interface as described in [General Communication Device Functions](#). Functions specific to a device are described in their respective sections.

The following ports are used in Verix eVo-based terminals:

- [RS-232 Serial Port \(COM1\)](#)
- [Serial Port \(COM2\)](#) (Predator terminals)
- Radio devices (COM2)
- [Modem Port \(COM3\)](#)
- [Serial Printer Port \(COM4\)](#)
- [Internal PIN Pad \(IPP\) Port \(COM5\)](#)
- [USB External Serial \(COM6\)](#)
- 2 wire RS-232 Serial Port (COM7) (Trident)

RS-232 Serial Port (COM1 and COM2)

Verix eVo-based terminals implement RS-232 communications support with some model-specific variations. All terminals include at least one serial port for communication with another serial device. Device assignments are terminal-specific. Characteristics of this port are:

- Up to 115200 bps for most serial ports
- Support for character mode

RS-232 Serial Port Function Calls

All serial communication calls in the [General Communication Device Functions](#) section can be used on any serial port.

Communication Port Flow Control (COM1 and COM3)

The traditional flow control supported by Verix eVo-based terminals is set by the `Fmt_auto` parameter in [barcode_pending\(\)](#). This method of flow control allows the communication driver to react to the `CTS` signal by sending data if `CTS` is asserted by the other end of the communication session, or stop sending if `CTS` is deasserted. The application is responsible for controlling the `RTS` signal. If the application asserts `RTS`, the communication partner sends data and stops if `RTS` is deasserted.

Fully automated flow control is supported on all Verix V communication devices with hardware RTS/CTS signals or USB UART devices. `Fmt_AFC` is the recommended flow control for all devices supporting flow control. This is enabled through `Fmt_AFC`, and is mutually exclusive of the `Fmt_auto` feature. `Fmt_AFC` automatically controls CTS, RTS, and the flow control of data between communicating partners. The application cannot control RTS.

NOTE



When `Fmt_AFC` is used, any attempt to change RTS will result in an error.

All flow control is done by the driver. The application manages the available communication buffer space when performing writes by monitoring the status return, which indicates the number of bytes written. The number of bytes written should match the number of bytes in the buffer write.

Selecting Non-Standard Bit Rates (COM1 and COM2)

If necessary, these serial ports can be set to other rates besides the standard rates defined in `SVC.H`. To select a non-standard rate, the user must set two fields in the `opn_blk` structure prior to calling `set_opn_blk`:

Set the `rate` field to `Rt_user_defined`.

Set the `user_defined_rate` field to the actual bit rate desired.

NOTE



The `user_defined_rate` field must be within the range of 50 to 115200 bits per second.

Determining Actual Bit Rate (COM1 and COM2)

Due to the nature of the underlying hardware, the actual bit rate may not exactly match the selected rate. The actual bit rate can be determined by calling the function:

```
int get_bits_per_second (int hdl);
```

USB Dongles (COM3 and COM6)

There are times when conventional communication devices are needed for terminals such as the Vx670 and Vx 680. These terminals do not have conventional UART ports but support USB. This is accomplished by using the USB modem (COM3) and/or the USB UART (COM6). These dongles can be inserted into the base which has two USB connectors, or the Handy-Link cable which has a single USB connector. These devices have hard wired device names (COM6 for the UART and COM3 for the modem), and cannot support multiple devices of the same name.

USB devices have specific programming requirements because they can be inserted or removed any time. They are different from other communication devices, which are physically part of the terminal and cannot be removed. This affects the programming of the device from the `open()` to the `close()`.

When opening a USB device, it cannot be assumed that the device is present. Always check `open()` for success; if the result is -1, `errno` should be checked. If `errno` is `EBADF`, then the USB device has not been enumerated. Enumeration is the process where the USB driver recognizes the new USB device, and adds it to the USB bus. To assist in recognizing if a USB device is present, the USB driver posts an event if a USB device is enumerated or removed. The `EVT_USB` does not indicate which device caused the event. The application must check if the device of interest is present. The sample applications ([Asynchronous Example-SiLabs/](#)[Synchronous Example-SiLabs](#)) define how to wait on event and check the device using the `open` call.

NOTE



If a USB device is removed during a communication session, the device must be closed before it can be enumerated. The USB driver does not reconnect the USB device to the current communication session.

Once the USB modem or UART has opened the device, the normal `barcode_pending()` call is needed to configure the device for communication. The USB devices (COM3 and COM6) are configured just like other communication devices. The only consideration is that the result code and `errno` values should be checked as it is possible that the USB device was disconnected between the `reset_port_error()` and the `set_opn_blk` call.

Checking of the response code and `errno` is a good practice for all I/O calls. Waiting on a `read()` in a `while` loop without checking the response code and `errno` could lead to an infinite loop if the device is removed. Once the device is enumerated and removed, subsequent I/O calls return a -1 result code and `errno` set to `ENXIO`, except for the `close()` function.

NOTE



Polling communication devices is bad for power conservation and battery life. The OS has power saving features based on the application going into `wait_events` or other wait states rather than actively polling a device.

Removal of the USB device can also mean that the handheld terminal was removed from the base. This is the same as if the USB dongle was removed from the base socket or the Handy-Link cable.

The function `get_usb_device_bits` can be used to determine what USB devices are enumerated. The function returns an unsigned long bit map of devices. The devices are defined in `SVC.H`.

When waiting for events, the application must not assume (if the USB device is enumerated) that the next `EVT_USB` means the device is no longer enumerated or that the device is actually enumerating. With power management and power savings, the application may miss `EVT_USB` and may only be recognized on power up.

On COM6, the `get_fifo_config()` call returns an error because this does not apply to USB devices.

RS-232 Serial Port (COM1)

The RS-232 port on Verix eVo-based terminals are compliant with RS-232C. This includes hardware handshaking. Rates on this port are from 300 to 115200 baud.

Pin	Signal	Pin	Signal
1	N/C	6	RXD
2	N/C	7	TXD
3	N/C	8	CTS
4	N/C	9	RTS
5	GND	10	N/C

Signals and their corresponding function:

Signal	Function	Signal	Function
GND	Signal Ground	RTS	Request to Send
RXD	Receive Data	CTS	Clear to Send
TXD	Transmit Data		

If the `COM1_PWR_ON` is not supported, then the RS232 driver chip is always enabled. This means that the application can always use the serial port without turning the driver on. If the `COM1_PWR_PIN_ON` is not supported, then you cannot attach unpowered peripherals, such as barcode wands or PIN pads.

The API `set_com1_pwr()` always returns 0, unless called with the handle for some other COM port where it returns `-1` and `ERRNO` set to `EINVAL`.

NOTE



On VX 680/VX670, COM1 is an external serial available via the Handy-Link connector found on the bottom of the unit (also referred to as the cell phone-style connector).

UART devices connected to communication ports implemented via simulation on the USB support both types of flow control.

The USB Client mode defaults to HID if there is no defined value in the `*USBCLIENT` environment variable. If `*USBCLIENT` is defined as RS-232, the terminal assumes a PID value of 0x0216. The device supports connection to Windows XP PC using the updated `Vxuart.inf` included in the SDK.

Serial Port (COM2)

The Predator terminals COM2 port supports, transmits, and receives signals and provide +9V power for external devices such as, VX810, VX 820, VX805, the PINpad 1000SE or the SC5000. Most terminal's support the EPP on COM1.

Verix V terminals with radios such as VX610, VX510 GPRS, VX670, VX 680, and VX 520 GPRS support a full 8 wire UART interface for COM2. The serial port will supply the RTS, DTR, CTS, DCD, and DSR handshake signals to the radio, which are accessed via the standard `barcode_pending()` function. Additional signals specific to radio modems are `RAD_RST`, `RAD_OFF`, `RAD_MOD`, `RAD_INT` and `RAD_INT2`.

Use the standard serial I/O commands such as, `close()`, `open()`, `set_opn_blk()`, `read()`, and `write()` to control this port. Packet formats depend on the external device. Refer to the documentation of your external device.

- `RAD_MOD` switches the USB device lines. In the default position (0), the lines are connected to pins 2 and 3 on the RJ45 connector. If set using a hidden VTM function, the MC5727 is disconnected from the host pins of the ARM and connected to the device pins. This configuration is used to update modem firmware or PRI / PRL information using Sierra Wireless PC based tools.
- `RAD_OFF` pin is used by both M200 and MC5727. The MC5727 uses the same OS startup values as the M200. It can be controlled through `set_radio_ctl()` as it is in current Verix eVo products.
- `RAD_INT` allows the user to determine the status of the modem. This is returned in bit zero (0) of the return signals of `get_radio_ctl()`.

Predator devices on COM2 have a small electrically Erasable Programmable Read Only Memory (EEPROM) containing a module ID that allows the OS and applications to determine which radio module is installed.

The OS expects the EEPROM to be populated and the correct value, 73 (`MID_MC5727`) in manufacturing allowing the users to call `SVC_INFO_MODULE(2)` to determine that the attached radio modem is Sierra Wireless MC5727 CDMA EVDO radio modem. `MID_MC5727` will be defined in `svc.h` header file. The OS will use EEPROM identifier at boot up to identify which board is attached to the terminal. If it's `MID_MC5727`, the COM2 USB /MC5727 driver code will be loaded. Alternatively, the COM2 serial driver will be loaded to communicate with the M200 CDMA board. Reading the EEPROM may slow down the startup.

Predator GID1 `CONFIG.SYS` variables `COM2HW` and `COM3HW` allow the correct COM2 and COM3 module IDs to be overridden with user specified values.

CAUTION



This feature is intended for testing and should be used with caution. Using this feature can cause applications and the OS to incorrectly handle the module.

The variables are not protected (do not start with # or *), thus, are erased on full downloads and memory file clears. This is designed to reduce the risk of this feature being unintentionally enabled in the field.

NOTE



Trident uses the CIB to define the radios modules used and specific characteristics for the HW instead of the EEPROM used in Predator.

Example: Setting `COM2HW=11` in GID1 causes `SVC_INFO_MODULE_ID(2)` to return 11 regardless of which wireless module is installed on COM2.

Setting COM3HW=3 in GID1 causes SVC_INFO_MOD_ID() and SVC_INFO_MODULE_ID(3) to return 3 regardless of which modem is installed on COM3.

The GPRS cannot be powered without a SIM, which is inserted inside the slot right under the battery pack. If the terminal detects that there is no battery, and the radio is GPRS, the OS activates the /EMERGOFF pin on the radio module, which causes the radio to shut off. In the absence of the battery, the OS will not allow the application to turn the GPRS radio on.

get_battery_value()

Allows applications to obtain the battery's operating temperature, or the battery's output current.

Prototype

```
int temperature=get_battery_value(BATTERYTEMP);
int current=get_battery_value(BATTERYCURRENT);
int get_battery_value(FULLCHARGE);
int get_battery_value(REMAININGCHARGE);
int get_battery_value(BATTERYVOLTAGE);
int get_battery_value(CHARGERSTATUS);
int get_battery_value(LOWBATTLELEVEL);
int get_battery_value();
int get_battery_sts();
```

For more information, refer to the APIs in [Application Interface](#).

NOTE



get_battery_value(FULLCHARGE) always returns the "theoretical FC" provided by the manufacturer. There is no provision to uniquely identify each cell, thus, we cannot match an age or damage parameter to a specific cell. Aging or damage reduction of FC is not accounted for.

get_battery_value(LOWBATTLELEVEL) will still occur when the calculated Remaining Charge (RC) falls below the user specified value. The calculated values are shown in the Returned value column in [Table 49](#).

The VX 675 does not have a charge counter like other battery terminals, hence, we cannot provide direct RC values. Use a "modeled" approach to determine RC — read the voltage and use that value as an index into a table, which will provide a "best guess" as to the RC in various increments.

The return values chosen indicate that the battery has AT LEAST this amount of charge. The increments are chosen such that when the battery is "fuller", it stays full longer and that when the battery is low, there is some reserve. This allows the application to use a "Five Value battery icon" (4 bars + blank). Note that the application has control over the display and if they choose to display a numeric percentage, the result of that calculation will only result in 75, 50, 30, 10 and 0.

Table 49 Remaining Charge (RC) best guess range

Charge range	Returned value	Bars
75 – 100	2250 (75%)	4
50 – 74	1462 (50%)	3
30 – 49	785 (30%)	2
10 – 29	450 (10%)	1
0 – 9	0 (0%)	0

NOTE



Modeling the cell is a "best guess" at the RC. Applications that try to run the most transactions by using a LOW RC threshold i.e. numeric 5% will stop when the battery actually has ~10% remaining.

This method minimizes the API changes. An alternate method would be to deprecate this API and create a new API that returns numeric 0 - 4 corresponding to the battery "bars" in [Table 49](#).

Return Values

The battery temperature in degree Celsius, or the battery current in mA. A positive value means the battery is charging, while a negative value means that the battery is discharging.

VX 675 has no current measuring hardware.

get_battery_value(BATTERYCURRENT) always returns 0.

get_battery_value (CHARGEDSTATUS) returns the VX 675 battery status and external power supply availability.

Bit-mapped return values where:

Rrrr rrcp

p: 0 = External power supply not available

1 = External power supply available

c: 0 = Not charging

1 = Charging

r: Reserved for future use. This bit may be any value. It is up to the user to mask off unwanted bits.

This returns:

-1 = Error, not a battery operated terminal

0 = No external power, not charging

1 = External power available, not charging

2 = Illegal, not a valid combination, API will not return this value

3 = External power available, charging

This can also be used to determine when the battery is fully charged.

Example

Click the linked example to view the sample code.

Modem Port (COM3)

Verix eVo-based terminals use four different modems. The Conexant CX81802 (Banshee) modem, the Conexant CX86501 (Eisenhower) modem, the Conexant CX93040 (Harley) modem, and the Silicon Laboratory Modem Si24xx.

The Conexant modems (CX81802, CX86501, and CX93040) are very similar to each other, and use the same AT command set. Some features of the Conexant CX86501 that our industry does not generally use (such as, fax and voice), have been removed.

For more information on Conexant modems AT commands, refer to the *Conexant CX Modem AT Command Reference Manual*, Part Number E-102184. This manual details the AT commands for CX81801 SmartV.xx, CX06833 SMXXD, CX81300 SmartACF, and CX06827 SCXXD.

The Silicon Laboratories modem Si24xx is similar to the Conexant modems but has several differences in the AT command set.

NOTE



Three versions of the Silicon Laboratories Si24xx chip set are currently used—Si2434 (33.6kps), Si2415 (14.4kps), and the Si2493 (56kps).

For more information on the Silicon Laboratories Si2434 modem AT commands, refer to the *Silicon Labs AT Commands Manual*, Part Number - 24494. This manual details the AT commands for AN93 Si2493/Si2457/Si2434/Si2415/Si2404.

The sample applications below also provide the suggested AT initialization strings for the basic [Asynchronous](#) and [Synchronous](#) connections.

The modem port COM3 is also a USB device on V*670 or V*670. The V*670 terminal does not have an internal modem but uses a modem dongle with a USB UART chip and a Silicon Labs modem chip in a single unit. For more information, refer to [Silicon Laboratories Modem \(Si24xx\)](#).

The V*810 DUET base station contains a USB modem device, which is functionally the same as the USB modem dongle in the VX 680 terminal. The USB modem driver code is actually imported from the VX 680 OS without any changes except to recognize the new DUET USB modem PID. The VX 820 and V*805 DUET bases uses a Conexant Harley modem and is different from the V*810 DUET base or the USB modem dongle.

NOTE



The modem does not operate unless a model profile is loaded on the VX 820 PIN pad. Note that the VX 820 stand-alone PIN pad does not have a modem, thus, it is not normally loaded with a model profile. It always accepts a modem profile and saves it in flash memory, whether or not it is connected to a DUET.

A VX 820 PIN pad can be connected to a DUET, but the modem does not operate until a modem profile is present in the VX 820 PIN pad memory and loaded into the modem device. Modem profile naming rules and loading conventions are identical to that of the V*670 terminal.

The USB modem in the VX 820 DUET operates exactly the same as the USB modem in the VX670 terminal. The `UDB_COM3` is set in the USB device status. The API `get_component_vars()` returns a file name of “com3_usb.bin” for the USB modem device.

When an external USB modem dongle is also connected to the DUET USB Host port (USB type A), the OS does not allow the external USB modem dongle to be used (although the standard USB connect/disconnect chime sounds). In this case, the OS treats the external USB modem dongle as an unsupported device.

VX 675 Full Feature Base and VX 820 DUET Modem Device API

When the VX 820 PIN pad operates in a stand-alone mode, all modem related APIs return `-1` with `errno` set to `ENODEV`.

When the VX 820 PIN pad is connected to a DUET or VX 675 is connected to a VX 675 Full Feature base, the modem related APIs return normal response codes (for example, `open()` returns a valid handle). In this mode, the VX 820 PIN pad presents the same API and behavior as the VX 680 terminal when the external modem dongle is connected.

NOTE



The VX 675 Full Feature base and VX 820 DUET modem supports the Conexant Harley modem.

Conexant Modems (Banshee and Eisenhower)

The CX14.4 modem is an external modem contained within most terminals. Communication with the device is through COM3 and uses the functions described in [General Communication Device Functions](#). Modem-specific calls are described in the [Modem Functions](#) section.

Command and configuration of the modem is done using standard AT commands. These AT commands are described in the *Conexant CX Modem AT Command Reference Manual*.

Conexant Modems' Supported Data Protocols

- Bell 103
- Bell 212A
- V.21
- V.22
- V.22 SDLC FastConnect
- V.22bis
- V.32
- V.32bis
- V.44 data compression
- V.42bis and MNP 5 data compression
- V.42 LAPM and MNP 2-4 error correction

- V.90 (CX93040)
- V.92 (CX93040)
- RS-232 interface supports communication rates up to 115.2 Kbps

Asynchronous and synchronous (SDLC) modes are supported, although synchronous mode is implemented differently than other modems.

Silicon Laboratories Modem (Si24xx)

To use the modem dongle consisting of a USB UART chip and a Silicon Labs modem chip in a single unit, the unit must be plugged into the base. The modem dongle is Verix eVo device “/DEV/COM3”, with an the event bit of EVT_COM3. Only one USB modem dongle may be connected to the base at any time. Succeeding dongles will be ignored. Standard functions such as [open\(\)](#), [set_opn_blk\(\)](#), [read\(\)](#), [write\(\)](#), and [close\(\)](#) will remain available.

If the modem dongle is not plugged in, `open(“/DEV/COM3”, 0)` returns `-1`, with `errno` set to `EBADF`. If the modem dongle is removed after it is opened, API functions such as [read\(\)](#), [write\(\)](#), and [set_opn_blk\(\)](#) returns `-1` with `errno` set to `ENXIO`.

The Silicon Laboratory modem has some differences with how AT commands are concatenated. The total buffer length for an AT command is 64 bytes. No more than one `:Unn` command can be in an AT command but consecutive `:Unn` commands can be appended to a single `:Unn` command. The `:Unn` command must be the last command in the buffer. The semicolon (`;`) command delimiter can be used for the `+` commands, such as `+MS`. These commands have variable command parameters. When concatenating more AT commands after a `+` command, the modem should know the end of the command. See [Asynchronous Example-SiLabs](#) and [Synchronous Example-SiLabs](#) for examples.

The `ATZ` and `AT&F` commands are the same for the Silicon Laboratory modem. The `&F` command is a macro and is part of the modem profile. The `&F` command always return `OK` regardless of the state of the result code response for the modem. The response code for `ATZ` and `AT&F` will return immediately, but the modem goes into a reset and does not process further commands for 300ms. Any commands included in AT commands with the `Z` or `&F` commands are lost as the modem does a full reset to default condition including the command buffer. It is recommended that an `ATZ` or `AT&F` be issued to the modem when changing communication protocols (changing from asynchronous to synchronous or vice versa). When sharing the modem with other tasks at any time, the modem state cannot be guaranteed from a previous use.

NOTE



The Si2434 modem supports the same set of data protocols enumerated in [Conexant Modems' Supported Data Protocols](#).

Asynchronous

The modem is configured for speed buffering. This allows COM3 to operate at its maximum rate (115.2 kbps). It is highly recommended that communication with the modem be 115.2 kbps. The modem port rate is set using `set_opn_blk()`. The modem connection rate is set using the `+MS AT` command. FastConnect is defined using the `$F` command when using the Conexant modems, and U7A and &H13 when using the Silicon Laboratories modem (See the [Asynchronous Example-SiLabs/Synchronous Example-SiLabs](#), and the AT Command reference for details. The included example shows a typical 1200, V.22 FastConnect connection).

Asynchronous Example-Conexant

This sample program dials the HyperCom NAC with a host simulator board. The application sends the message defined below and receives a standard response.

Asynchronous Example-SiLabs

This sample program dials the HyperCom NAC with a host simulator board. The application sends the message defined below and receives a standard response. Note the difference in the AT Commands, use of events, and checks for the presence of USB device.

Synchronous

The Verix eVo terminals and modems always communicate asynchronously; no clock signals are present between the modem and terminal. The Conexant and Silicon Laboratory modems perform synchronous framing, while the Verix eVo operating system handles the SDLC protocol. The modem is opened and setup in asynchronous mode for initial configuration.

NOTE



The application is responsible for configuring (AT commands) the modem correctly for synchronous communication, the driver does not send any AT commands to the modem.

CAUTION



In synchronous communication, it is **REQUIRED** that the modem port speed be set to 115.2Kps. Failure to open the communication port at 115.2Kps can result in corrupted communication with the host.

The AT commands `+ES` and `+ESA` configure the modem for synchronous communication. The application then dials the host and once the connection is established and a connect message is received, a second `set_opn_blk()` function is called to change the modem state to synchronous. The CTS line is not monitored or used as a signal to switch modes, as is the case with OS previous to Verix V. See [Synchronous Example-Conexant](#) and [Synchronous Example-SiLabs](#) for differences between AT init strings.

SDLC Protocol

Most Verix eVo-based terminals support SDLC communications (refer to *Communications Programming Concepts*, October 1997, on the IBM Web site, at <http://www.ibm.com/us/>). This support is a subset of the frame formats defined for secondary stations in a half-duplex switched point-to-point configuration. IBM has defined three major groups of frame formats: supervisory, information, and unnumbered.

Supervisory Packet Format

Verix eVo-based terminals can receive all supervisory format frames. Only *ready-to-receive* (RR) frames are processed and only RR frames are generated.

Modulo 8 Information Packet Format

Verix eVo-based terminals can receive and generate Modulo 8 information frames. The terminals immediately acknowledge each Modulo 8 frame they receive; each Modulo 8 frame generated by the terminal should be immediately acknowledged.

NOTE



Modulo 128 packets are not supported.

Unnumbered Information Packet Format

Verix eVo-based terminals can receive all unnumbered information frame formats.

- The *set normal response mode* (SNRM) frame is processed and acknowledged with the *unnumbered acknowledgment* (UA) frame.
- The *disconnect* (DISC) message is acknowledged with the UA frame, but is not otherwise processed.
- All other unnumbered information frames are ignored.

Communication Sequence

Communication begins as soon as the modem handshake is complete; the supported handshakes include Bell 212A, CCITT V.22, and CCITT V.22bis. The HyperComm NAC fastconnect is supported for Bell 212 and CCITT V.22 modulations.

NOTE



The Verix eVo operating system supports SDLC/V.80 protocol on COM2 (Predator only, Vx670 BT) and COM3. See [Bluetooth Modem Application Flow](#) for more information on the Bluetooth modem transaction.

SDLC Initialization

The `set_opn_blk()` function call initializes an opened COM port.

Synchronous Example-Conexant

Conexant example.

Synchronous Example-SiLabs

SiLabs example.

The linked sample program dials the HyperCom NAC with a host simulator board. The application sends a message and receives a standard response.

When initializing a port for SDLC, the following additional settings are required in the `Opn_Blks` structure:

- To define SDLC mode, in the `Opn_Blk` structure use:

```
buffer.format = Fmt_SDLC
buffer.protocol = P_sdlc_mode;
```

- To define the SDLC secondary station address, in the `Opn_Blk` structure use:

```
buffer.trailer.sdlc_parms.address = 0x30;
```

- To connect to any SDLC host in the `Opn_Blk` structure use:

```
buffer.trailer.sdlc_parms.option= P_sdlc_sec;
```

NOTE



It is mandatory that the port speed be set to `Rt_115200`. Speeds slower than this will cause problems with the SDLC protocol.

Setting RS-232 Signals

To set RS-232 signals (DTR and RTS), use the command:

```
result = set_serial_lines(int hPort, char *signal);
```

Setting RTS/CTS Handshake

SDLC support includes support of RTS/CTS handshake through the RS-232 port. Use `set_opn_blk()` with the `format` parameter set to `Fmt_SDLC` and `Fmt_auto` or `Fmt_AFC`, the `protocol` parameter set to `P_sdlc_mode`, and the `protocol` parameter `trailer.option` set to `P_sdlc_sec`.

NOTE



`Fmt_AFC` is the preferred flow control for SDLC communications in Verix eVo environment.

Enhanced SDLC Protocol

The SDLC protocol automatically handles the ACK/NAK of the data packets and application. The host assumes that if a packet is received, the sending site receives the ACK for the packet. This is not necessarily true. If communication is lost after a data packet is received and before the ACK is sent, the sending site assumes that the packet is not received. However, the receiving site believes that the ACK is sent.

The SDLC protocol is enhanced so it does not post the received data packet that is read by the application until after the next poll or data packet indicates that the transmission is completed. This ensures that the sending site has received the acknowledgement to the data packet.

SDLC Status

Verification of the SDLC protocol status is supported. This modifies the `get_port_status()` to return a running count of polls from the host in the second byte of the 4 byte `get_port_status()` argument. The second byte is always 0 for Verix V and Verix eVo implementations of SDLC. In the old TXO environment, this byte contains the number of data packets in the reject queue.

Since Verix V and Verix eVo do not support the reject queue, this field is not used. The second byte is a modulo 256 of the actual count. After the count starts incrementing and the modulo 256 operation results in 0, 1 is added to the total count so the reported byte is never 0 after the polling starts.

NOTE



The VX 680 BT connection with the Dione Base returns the SDLC count in byte 3 instead of byte 2.

Country Profiles

The modem requires a country profile for the resident country. The profile file used for this modem is composed of two parts: the base patch file that contains relevant firmware updates and the country file that contains specific settings for the country.

Verifying the installation of a country profile on a terminal is simple:

- Check the version in the TERMINAL INFO screen selected in VTM menu 1 or menu 2 in some terminals.
- Select the TERMINAL INFO option to display system information.

NOTE



Checking the profile version requires the modem to be present as VTM issues AT commands to fetch the version. The Vx670 and VX 680 must be on the base and a USB modem dongle inserted to get the version. For the Vx810, VX 820 and Vx805 the Duet base must be connected.

Figure 10 illustrates the sample system information screen on VX 520.

```
VTM MGR TERMINAL INFO

PinPad                nn
Modem Type            40
Ver:                  F2000B03B5000104
Modem Model:          CX93001
Modem Ctry:           B5
```

Figure 10 Sample System Information Screen on VX 520

For more information on the System Information refer to the Reference Guide of your terminal.

The VER field, for Conexant modems, is composed of two parts—the first eight characters represent the patch version. The next eight characters compose the country code and version of this profile. If a valid country profile is not loaded, the first two bytes are not “B3.” The I3 response is the vendor part number, and the MODEM CTRY value is the country code. Note that the country code is also included in the second half of the VER field.

The country profile can be loaded using the modem configuration utility. If the country profile file is available, it can be downloaded using the following command:

Example Banshee example.

Example Eisenhower example.

Example Silicon Labs example.

NOTE



A new feature for Trident terminals with subdirectory OS or later is supported for both the Harley and SiLabs profiles in the same terminal. The VX 680 BT can support the USB modem dongle and the BT Base modem so both profile types are required.

To prevent loading the wrong modem profile into the terminal, the OS checks the filename of the modem profile contained in the `CONFIG.SYS` variable, `*MN`, against the type of modem installed in the terminal. To indicate whether the profile filename does not match the modem type installed in the terminal, an error return for `*MERR` is created. Table 50 lists the `*MERR` error code values.

Table 50 *MERR Values

*MERR Values	Display	Description
1	EXTENSION NOT .ZIP	File defined by *MN is not a zip file
2	NOT AUTHENTICATED	File defined by *MN is not authenticated
3	*MN FILE ZERO LEN	File defined by *MN has a length of zero
4	FILE COPYING ERROR	MODPROF.ZIP does not exist
5	FILE UNZIP ERROR	ZIP file fails unzip operation
6	NAME NOT MODPROF.S37	The file within the *MN zip file is not named MODPROF.S37
7	MODEM COMM ERROR	Modem Communication error such as: <ul style="list-style-type: none"> Modem fails to respond with OK when download is completed Modem does not respond with "." For each record written Modem does not respond as expected (AT** does not cause the "Download initiated" message)
8	MDM PROFILE MISMATCH	Modem profile does not match modem type
9	*MD UNZIP ERROR	Illegal profiles or other file types in the file pointed to by *MD

The downloaded modem profiles are composed of a zip file with a standard filename format, an authentication file, and a `CONFIG.SYS` variable, `*MN`, set to the .zip file name.

NOTE

The Banshee modem has a filename format of `w96_nnnn.zip`, Eisenhower has `ESN_nnnn.zip` and Harley has `HAR_nnnn.zip`.

Modem Recommendations

The general modem recommendations are listed below.

- It is recommended that the modem port, `COM3`, be opened at the fastest possible speed (115.2 Kbps). This minimizes latency between the application, modem, and host. This is especially important in synchronous SDLC mode.
- The Conexant modems do not support the 33.6 Kbps baud rate. Attempts to perform a `set_opn_blk()` on `COM3` at 33.6 Kbps results in a -1 status result and `errno` set to `-EINVAL`.
- If the Conexant modems are not commanded to use tone or pulse dialing, use `ATD` in the dial string to allow the modem to attempt to tone dial and test the line for acceptance of this mode. If the tone dial fails, the modem should use pulse dialing. The test for tone acceptance is performed after the first digit is dialed and results in a slight delay before the second digit is dialed.
 - If tone dialing is desired, use `ATDT` plus the number to dial.
 - If pulse mode is desired, use `ATDP` plus the phone number to dial.
 - When using `SVC_ZONTALK()`, either through the application or VTM, the default dialing mode is `ATD`.
 - The `CONFIG.SYS` variable `*ZP` stores the number to dial. If tone dialing is desired, add a `T` before the phone number to force tone dialing (for example, `*ZP=T1234567`). If pulse dialing is desired, add a `P` to the start of the phone number (for example, `*ZP=P1234567`).
- The modem supports line sense, but the modem must be configured before use. Line sense is automatically performed during dialing. If a line is not attached or the line is dead, the modem returns `NO LINE`.
 - To enable line sense, include `-STE=1` in the AT initialization string.
 - If a line sense test is required without dialing, the line voltage test Conexant modem command, `AT-TRV`, can be used. A returned result equal to ASCII 1.4 is equivalent of `NO LINE`.

The Silicon Laboratories modem has the line sense capability automatically enabled in the modem profile. When SiLabs modem command `ATD` is used, the modem examines the line to determine if the line voltage is between the voltages defined in `U83` and `U84`. `NO LINE` is reported if the test fails.

References

Refer to the *Modem AT Commands Reference Guide* and the *Silicon Laboratories AN93*, for more information.

Modem Profile Loading

The Modem Profile Loading application handles the combined profile types. The modem type matches the selection of profiles.

Modem Functions

The following functions are used in Verix eVo environment for the modem only.

Modem Sleep State Functions

The Conexant Harley modem CX93001-EIS modem defaults to normal sleep state operation, but it has the capability to go into three more different states with increasing power savings but limit the modem's operational capacity while in that state. The sleep state with the most power savings results in a non-operational modem until a character is sent to the modem UART interface. This means the modem must be in command state - the modem cannot be connected to another modem over the telephone network.

Sleep states are generally used for power savings, especially for terminals on battery power. Putting the modem in deep sleep also resolves the interference caused by the modem chip internal clocks with the GPRS radio. This interference affects the GPRS radio (either the 2G or 3G variants) and can cause connection issues. This is manifested by the long re-connect times in areas with marginal signal strength.

The modem driver's open/close state does not affect the modem's sleep state. It is true that the modem port must be open to send the AT commands to put the modem in sleep state, but the modem can be closed or remain open after the modem has been put in deep sleep mode, without any effect on the modem. The modem will wake with the first character (normally an AT command) sent to the modem UART port.

NOTE



The modem must be in command mode to be in deep sleep, but that command will fail as the "A" of the AT command would be lost. In consideration of the lost character, a function is created to wake the modem.

Modem sleep state functions are passed the handle of the modem port. If the handle is 0, it is assumed that modem port is closed and the function will open the port, perform the needed operation, and then close the port. If the handle is non-zero, the function will not open/close the port and assumes the port is already open. If an `open/read/write/set_opn_blk/close` function fails within the `modem_sleep()` and `modem_wake()` state functions, the function is aborted with the return set to `errno` value returned by the failing function. Otherwise, the successful operation of the function returns 0.

Refer to CX930xx-2x Modem AT Commands Reference Manual, VFI p/n E-REM-201692B, rev. A. on page 3.45, SLP command, for more information on modem sleep state details. Deep sleep mode 3 is used to completely turnoff the modem chip's clocks.

- `modem_sleep()`
- `modem_wake()`

modem_sleep()

Places the Conexant Harley modem chip into Deep Sleep mode.

Prototype `int modem_sleep(int hdl);`

Parameters

handle	Handle of modem. If handle is 0 then modem port is assumed to be closed and the function will open/close the port.
--------	--

Return Values

Success	0
Failure	Set to errno of the failing function inside <code>modem_sleep()</code> .

modem_wake()

Wakes modem from Deep Sleep mode.

Prototype `int modem_wake(int hdl);`

Parameters

handle	Handle of modem. If handle is 0 then modem port is assumed to be closed and the function will open/close the port.
--------	--

Return Values

Success	0
Failure	Set to errno of the failing function inside <code>modem_wake()</code> .

SVC_ZONTALK()

Receives a download through the terminal modem (ZonTalk is a reference to the protocol used by the VeriFone download servers.) See also, [download\(\)](#).

Prototype `int SVC_ZONTALK (unsigned char type);`

Parameters
 `type` Specifies the download type.

The basic type codes are:

- “F” for a full download
- “P” or “p” for partial downloads

Full and partial (uppercase “P”) downloads, when successful, force a restart of the application; partial downloads specified by a lowercase “p” allow the application to resume execution. This download-and-resume feature is transparent to the end user.

- “R” like F except that all application files in all groups in both I: and F: drives are deleted, and the flash is coalesced (Predator only, Trident ignores coalesce). Only Group 1 tasks are allowed to do this.
- “r” like R except that flash is not coalesced (Predator only, Trident ignores coalesce).

The basic type can be modified by setting the low- or the high-order bit of the type-code character, as follows:

- If the least-significant bit is set, the modem dial-out is bypassed. Use this option if the application has already opened the modem and established a connection.
- If the most-significant bit is set, an *alternate* modem is used. The interpretation of this depends on the terminal model. On terminals with only one modem, this has no effect.
- The NO_DIAL option is used if an application establishes the modem connection prior to calling [download\(\)](#). The NO_DIAL option is enabled if bit 0 of the SVC_ZONTALK() type variable is set to 1.

The SVC_ZONTALK() uses the following CONFIG.SYS variables to specify additional download parameters; however, their use depends on the modem hardware and the download server.

*COMBO	Sets the application group to use a modem or TCP/IP. <ul style="list-style-type: none">• 0 or not defined or invalid value = Modem is selected• 1 = TCP/IP is selected
*ZT	Terminal ID.
*ZA	Application ID.

*ZP	Host telephone number, or TCP/IP address including port number. SVC_ZONTALK() checks *ZP variable for the following if the TCP/IP feature is selected: <ul style="list-style-type: none"> • There must be at least one colon. • There must be at least one character before the first colon. • There must be at least one digit after the first colon.
*ZS	<ul style="list-style-type: none"> • If *ZS=1, the terminal serial number is added to the sign on packet. • If *ZS does not exist or if it exists but is not equal to '1', the serial number is not added to the sign on packet.
*ZINIT	Modem initialization string. Default = ATM0V0&D2.
*ZRESET	Modem reset string. Default = ATZ0.
*ZRESP	Modem connect response. Default = CONNECT.
*ZSWESC	Modem software escape flag. If set use '+++' to escape into command mode, rather than DTR transition.

If the download succeeds and the type is anything other than "p," the terminal restarts, thus SVC_ZONTALK() does not return to the caller.

ENOENT means there is no modem profile so the modem could not be used even if it is physically present.

EBUSY means the modem is not connected.

Return Values

For type "p" downloads, it returns 0 on success. For all types, it returns the following non-zero values on failure:

- 1 All errors (other than the following). Includes invalid arguments and CONFIG.SYS variable values.
- 2 Cannot open modem serial port.
- 3 I/O error reading or writing modem port.
- 4 Download host abort.
- 5 No phone number stored in *ZP.
- 6 Error writing file (possibly out of space).
- 7 Bad download data packet (host error).
- 8 Lost carrier.
- 13 Attempt to do type a 'R' or 'r' download from group other than 1.
- 90 Modem reported no line.
- 3 Modem reported no carrier.
- 7 Modem reported line busy.
- 6 Modem reported no dial tone.

8 Modem reported no answer.

External Modem only

-3 I/O error writing or reading modem port.

If the download fails, the user *must* reset the terminal, manually enter VTM, and reinitiate the download.

Serial Printer Port (COM4)

This section discusses serial printer port programming.

Special Items specifies the command set and operation of the microcomputer firmware that operates the printer device driver.

C Code Applet for COM4 Driver

The following linked example C code file uses the serial printer device driver. This example is *not* intended to be all inclusive, but demonstrates basic function calls and features.

Example

Click the linked example to view the sample code.

A DUET base station (V×810, VX 820, and V×805) contains a thermal or sprocket printer depending on the terminal type. Devices in the Duet are USB devices and set the UDB_COM4 bit in the USB device status to indicate if the device is present.

NOTE



Applications should always use the defined bit name for the device, in this case UDB_COM4, rather than the actual hex value.

The API `get_component_vars()` returns a file name of “com4_usb.bin” for the USB thermal printer device—the Micro-controller firmware. The MCU firmware provides the standard Verix eVo printer feature set and controls the DUET printer mechanism optimally.

NOTE



The *DARK parameter is modified for quality printing. Printing on Chinese Telcom 2-ply paper is not supported.

Internal PIN Pad (IPP) Port (COM5)

The Verix eVo internal PIN pad is a software-only emulation of the IPP7 chip used in Verix V platforms. IPP7 is also used in non-Verix V platforms. The VVIP is designed to work with existing applications written for the IPP7 chip. VVIP does not support Spain mode and Secure Messaging. See the Verix V ARM document for more details.

COM5 is the communication port assigned to the IPP. In Verix V terminals, the IPP7 chip and UART are emulated in software. Internal IPP is backwards compatible with IPP7. IPP can hold up to 10 triple length keys. The IPP appears as COM5 and the EPP appears on the port the device is connected, such as COM1, COM2, and others. The IPP is a device that supports state-of-the-art measures to provide security for PIN-based transactions.

Clear IPP Keys Upon Certificate Tree Removal

When Verix eVo file authentication detects a corrupt file authentication certificate tree, or tamper or any kind, it clears the IPP encryption keys and the exhaustive search algorithm counter.

IPP Function Calls

All of the functions that take a handle as a parameter will return –1 and set errno to EBADF if the handle is invalid.

NOTE



For information on how IPP communication parameters are initiated/reinitiated, please refer to [set_opn_blk\(\)](#).

APIs

This section presents the IPP function calls. You may also refer to the following generic APIs:

- [open\(\)](#)
- [read\(\)](#)
- [write\(\)](#)
- [close\(\)](#)
- [reset_port_error\(\)](#)

select_pinpad()

There is no port multiplexing hardware, thus, this always returns zero (0). Type is ignored.

Prototype `int select_pinpad(int type);`

Return Values Success: 0

IPP_power()

There is no IPP power on/off hardware.

Prototype `int IPP_power(int type);`

Return Values `IPP_power(0)` returns `-1` and set `errno` to `EINVAL`.

TerminatePinEntry()

Ends the PIN entry session.

Prototype `int TerminatePinEntry(void);`

Return Values

Success: 0

Failure: -1 with `errno` set to `EBADF`, device not open.

 -1 with `errno` set to `EINVAL`, `set_opn_cfg` not done.

PINentryStatus()

Returns the PIN entry status and can be used to infer when the console belongs to the PIN-entry background task.

Prototype `int PINentryStatus(void);`

Return Values

Success: 0, there is no background PIN entry task running.

 1, PIN entry in progress.

Failure: -1 with `errno` set to the following:

`EBADF` - device not open.

`EINVAL` - open block not set.

USB External Serial (COM6)

Apart from the regular UART port accessible via the Handy-Link connector, the V×670, VX 680, V×810, and VX 820 terminals (V×810 and VX 820 USB COM6 is internal to the Duet base but it can be treated just like the USB UART dongle) allow connection to other serial devices via the USB UART dongle.

The USB dongle has a USB connector on one end and an RJ-45 jack on the other. It has the same functions as COM1, hence, standard functions such as `open()`, `set_opn_blk()`, `read()`, `write()`, and `close()` remain the same.

Only one USB UART dongle may be connected at any time. Any dongles connected after the first are ignored. The USB UART is `" /DEV/COM6 "` with event bit `EVT_COM6`. If a USB UART is not connected, `open(" /DEV/COM6 " , 0)` returns `-1` and set `errno` to `EBADF`. If a USB UART is removed after it is opened, API functions such as `write()`, and `set_opn_blk()` return `-1` and set `errno` to `ENXIO`.

NOTE



If the dongle is removed during a communication session (the device is opened), the device must be closed as the OS will not re-establish the communication session if the device is enumerated. Due to this, the application must test for the USB presence. If the USB is not present, the application must close the device, then look for a USB event and try to open the device to restart the communication.

The USB UART can be removed at any time. When doing I/O operations, check the result code and `errno` for each call to determine if the device is present. The event `EVT_USB` is set if the USB UART is inserted or removed.

On the V×810 DUET, the OS USB Host driver detects the USB hub in the DUET the same way it detects the USB hub in the V×670 terminal base.

When the VX 820 PIN pad is connected to a DUET base station, a new USB device definition, `UDB_IBHUB`, is set in the USB device status. Applications can use the API `get_usb_device_bits()` and check for `UDB_IBHUB` to determine if they are running on a stand-alone VX 820 or on a VX 820 with DUET.

The presence of the VID/PID Hub indicates to the OS that it is running on a VX 820 PIN pad that is connected to a DUET base station. When detected, the OS allows application access to the DUET devices.

NOTE



There are no secure USB devices on the VX 820 DUET. the VX 820 DUET does not contain USB device power control circuitry so the USB Hub cannot control the device power.

The `EVT_USB` bit is set in all running tasks' event masks whenever the OS detects a USB device connection or removal (for example, if any USB device is currently connected and when single-USB-device mode is either enabled or disabled for any other USB device, the `EVT_USB` event will be generated for that device when the port is either powered off or on).

NOTE

When a Verix eVo terminal is in the **DOWNLOAD NEEDED** state, the VTM polls several devices in a round robin fashion looking for `ENQ` from a download host. COM6 for the Verix eVo terminal is added to the list of devices for this poll. Provided that downloading is enabled, this allows users to download files to the Verix eVo terminal at the **DOWNLOAD NEEDED** prompt without entering VTM.

The OS recognizes three different USB devices as COM6 serial port. These are:

- VFI dongle 24122-01-R, USB to RJ-45 serial.
- VFI cable E-120-2327-00 Rev A, USB to RJ-45 serial download cable.
- ViVO cable 220-2442-00 Rev A, USB to RJ-45 ViVO interface cable, blue light cable.

The VTM Download screen shows COM6 as a download choice if any of these devices are attached. The download function will only be successful if the cable and the download host are compatible. Some combinations of cables and hosts do not communicate and render the download function unsuccessful.

APIs

The following section presents the USB secure mode function calls. You may also refer to the following generic APIs:

- `open()`
- `close()`

Error Conditions and Error Codes

Errors are reported by returning a result of -1 with `errno` set to a specific standard error code. The conditions are described below:

<code>EBUSY</code>	Device already opened by another application.
<code>EBADF</code>	Device not available, device is physically not connected or the external USB ports are powered off. Call <code>get_usb_device_bits()</code> to investigate.
<code>EAGAIN</code>	If single-USB-device mode is requested but one or more external USB devices are already open (trying to open <code>"/dev/wln1"</code> in single-USB-device mode while an external USB device is already open).

set_usb_multi_device()

Bit `O_SINGLE_DEV` set means that the application is requesting single-USB device mode. The OS checks if any external USB ports (excluding the device specified by handle) are currently open. If so, this returns `errno=EAGAIN`. Otherwise, all other external USB ports are powered off. `O_MULTI_DEV` or other setting means the application can concurrently use other external USB devices. The OS powers on all external ports.

Prototype

```
int set_usb_multi_device(int hdl, int onOff);
```

Parameters

<code>hdl</code>	Device name for an already opened USB device.
<code>onOff</code>	Bitmask specifying whether other external USB devices may be powered on or not.

If used with a non-USB device handle, it returns -1 with `errno` set to `EINVAL`. Trident terminals will return - 1 with `errno` set to `EBADF` for this call. All Predator terminals will also return - 1 with `errno` set to `EBADF` except for Vx670.

get_usb_device_bits()

Retrieves the status of all supported USB devices.

Prototype

```
unsigned long get_usb_device_bits (void);
```

This function has no parameters. The returned value is a bit map with one bit per supported USB device. A “1” means the device is ready, that is, enumerated and initialized, while “0” means the device is not ready.

Symbol	Bit Position	Description
UDB_FLASH	0	USB Flash drive
UDB_ETHER	1	USB Ethernet
UDB_WIFI	2	USB WiFi
UDB_COM3	3	USB modem
UDB_COM6	4	USB UART (serial)
UDB_COM4	6	USB Printer
UDB_KYBD	7	USB keyboard
UDB_VX810	8	V×810 PIN pad
UDB_PP1000SE	9	PIN pad 1000SE
UDB_BIO	10	USB Biometric reader
UDB_BAR	11	USB Bar code scanner
UDB_COM2	12	USB Radio (CDMA)
UDB_COM9	13	USB Radio auxiliary port (CDMA)
UDB_COM10	14	USB Radio auxiliary port (CDMA)
UDB_VX820	19	VX 820 PIN pad
UDB_VX805	20	V×805 PIN pad
UDB_USBHUB	23	Internal USB hub
UDB_IBHUB	29	DUET USB hub
UDB_SINGLE	30	Single USB mode

If a bit is set, the corresponding device is available, enumerated, and initialized. Otherwise, the device is not available. The special bit `UDB_SINGLE` is defined to indicate if the terminal is currently in single-USB-device mode (set) or multi-USB-device mode (clear).

- If single-USB-device mode is on for WiFi, the WiFi and single-USB-device mode bits are set. All other device bits is off, regardless of whether anything is connected.
- If single-USB-device mode is on for an external USB device, the bits for all the other external USB devices will be cleared. The WiFi bit is set (assuming WiFi is populated in the terminal) and the single-USB-device mode bit is set.

- If single-USB-device mode is not set for any device, the bits are set for all devices that are enumerated and initialized. The single-USB-device mode bit is cleared.

USB External Ethernet (ETH1)

Predator only supports Ethernet as a USB device, either internally or by using a dongle. This device is ETH1. In Trident Ethernet has two types, one is part of the SOC and is always ETH1, the other is ETH2, which is the USB Ethernet and is found in terminals such as VX 820 DUET and V×805 DUET.

USB Internal WiFi (WLN1)

In Predator V×670 the Conexant CX3880 is used for WiFi support. Trident uses the Broadcom BCM4329 combination Bluetooth and WiFi chip. Application use is different for each chip as some stack elements are not included in the Conexant chip and are supported in a software library.

BCM4329 is connected internally so it cannot be physically connected and disconnected. However, when the VX 680 powers up and down, the driver will receive connect and disconnect events for this device. Also the `USB_WIFI_POWER()` function will power the device on and off, which results in connect and disconnect events, respectively.

The driver accepts the following vendor and product identification combinations. Productions units should report product ID 0x3005 because this indicates a newer version of the boot ROM code that optimizes firmware load speed. Most Conexant CX9213U evaluation boards have the old boot ROM code so report product ID 0x3889. The evaluation boards are still useful for debugging and troubleshooting because a USB data monitor can be inserted between the board and the terminal. Product ID 0x3000 is added to support a special manufacturing and test boot ROM but is no longer needed.

Table 51 Vendor and Product IDs

Vendor ID	Product ID	Description
0x0572	0x3889	Older boot ROM code
0x0572	0x3005	Newer boot ROM code
0x0572	0x3000	Manufacturing and test boor ROM code

If the product ID is 0x3889 or 0x3005 the driver sends the contents of group 1 `F: #PRISMA09.X2` to the device. Next, the driver waits for a PIMFOR trap packet containing the device Ethernet MAC address (a unique 48-bit hardware address). The driver saves the MAC address so the `get_enet_MAC()` function can return the MAC address to applications.

Firmware Loading

The major complication in firmware loading is that the firmware file may be moved by file manager Flash coalesce while the driver is accessing the file. The function `update_firmware_base()` is responsible for checking Flash coalesce and adjusting the firmware base address when it moves.

After sending the firmware, the loader waits for a packet from the chip to indicate it is up and running. The first packet from the chip is always a PIMFOR trap reporting the chip's MAC address. This address is stored so the status function `get_enet_MAC()` can return the MAC address. The entire packet is also stored in the Readahead buffer so that applications will also see this packet.

The firmware is stored as a protected Flash file in group 1. On every USB connection, the firmware is downloaded over the USB bus to the Partagas chip. The file name is `#PRISMA09.X2`. It must be authenticated using VeriShield so there must be an associated P7S file usually named `#PRISMA09.X2.P7S`. The '#' in the file name makes this a protected file so it cannot be easily removed. The Partagas chip is not usable without this file so it should be hard to remove. It can be removed using the existing Verix eVo API function `remove()` as in:

```
remove("F:#PRISMA09.X2")
remove("F:#PRISMA09.X2.P7S");
```

or by using DDL as in:

```
ddl -r F:#PRISMA09.X2 -r F:#PRISMA09.X2.P7S
```

Even though the device hardware supports 802.11g (speeds up to 54 Mbits/s), the Predator USB hardware cannot support such high speeds. Actual throughput is expected to be 3-5 Mbits/s. IP download requires loading Comm Server in the terminal. The USB WiFi hardware does not have a built-in IP stack (unlike the Connect One iChip) so Comm Server is required. Comm Server (or a stripped down version) can be loaded in the factory using the protected file feature (add the prefix '#' to the filename) so it cannot be easily removed.

The interface to the WiFi device is similar to the USB-based Ethernet device interface. However, WiFi requires extensive radio and security management functions not present in Ethernet. For instance, the Ethernet status and control functions `get_enet_status()` and both do not apply to WiFi. The status function `get_enet_MAC()` is supported due to many applications directly using this feature.

NOTE


The WiFi MAC address can be obtained via PIMFOR operations but this is excessively complicated for applications that only want to report to MAC address.

The `EVT_WLN` has been assigned to USB WiFi, thus, all incoming data and PIMFOR management packets set this event bit. The Conexant firmware for the Partagas chip is stored in the flash file system group 1. It is downloaded to the device on every USB connect. For offline applications, the USB WiFi device can be powered off using the `USB_WIFI_POWER()` function.

For more information on WiFi, refer to [WiFi Module](#) section on [WWAN](#).

This section lists the function calls that are used in Vx670 USB WiFi, you may also refer to the generic APIs listed below:

- `open()`

- `read()`
- `write()`
- `close()`

USB_WIFI_POWER()

This is used in powering the WiFi device on or off. The USB WiFi device is turned on during restart. The `USB_WIFI_POWER(0)` turns the device off and the `USB_WIFI_POWER(1)` turns the device on. Passing any other values returns -1

Prototype `int USB_WIFI_POWER (int power);`

Return Values

Success: 0

Failure: -1 and `errno` set to `EINVAL`.

If the WiFi device is open, calling this function to power off the device will return -1 and set `errno` to `EBUSY`. This prevents one task from interrupting another task using the device.

WiFi Control and Status Management

The chip on Vx670 implements status and control functions via specially formatted Ethernet packets. The Ethernet packets are read/written using the same read/write functions used in USB WiFi. The chip intercepts management packets so they are not transmitted over the radio.

WiFi data and management packets are received/sent to the Verix eVo `read()`/`write()` API functions after opening the WiFi device `"/DEV/WLN1"`. Verix eVo events are posted for both types of packets. Data packets are formatted as standard Ethernet packets and will not be discussed further. Management packets are formatted to look like Ethernet packets but with an EtherType value of `0x8828`. This value plus the PIMFOR magic number is used to distinguish data from management packets.

Table 52 PIMFOR Packet Format

Offset (bytes)	Length (bytes)	Data Type	Description
0-5	6	Unsigned 8 bit integer[6]	Destination MAC address
6-11	6	Unsigned 8 bit integer[6]	Source MAC address
12-13	2	Constant unsigned 16 bit integer = 0x8828	EtherType
14	1	Constant 8 bit unsigned int = 0x01	PIMFOR version
15	1	Unsigned 8 bit integer	PIMFOR operation (0..4)
16-19	4	Unsigned 32 bit integer	PIMFOR OID
20-21	2	Constant unsigned 8 bit integer[2]={0x00, 0x00}	PIMFOR reserved
22-25	4	Unsigned 32 bit integer	PIMFOR value length
26...	Varies	Varies	PIMFOR value

Table 53 defines PIMFOR values and their description.

Table 53 PIMFOR Operation Values

PIMFOR Operation Value	Description
0	PIMFOR get
1	PIMFOR set
2	PIMFOR response
3	PIMFOR error
4	PIMFOR trap

- The unsigned 16 and 32 bit integers are in big endian order, that is, the natural order for the Verix eVo environment.
- The Verix eVo low-level driver ignores the two MAC address fields. When receiving PIMFOR packets, the MAC address fields contain garbage values.

- The value field is variable length.

NOTE



Integers in this field are in little endian order. Refer to the *[Partagas] Conexant CX3889 Driver Programmers Manual, Document # DO-412064-TC Draft 1D, April 28, 2005*, for more information on the data type for each OID.

Sending PIMFOR Requests

The following example shows how to build a PIMFOR request packet to get the WiFi device MAC address. The following is the description of the MAC address OID from the Conexant documentation:

```
aMACAddress
Macro: GEN OID MACADDRESS
OID: 0x00000000
Type: uint8 t[6]
Access: Read, Write
The MAC address of the device.
```

Table 54 describes how the PIMFOR packet should look like in Verix eVo:

Table 54 PIMFOR Packet on Verix eVo

Offset (bytes)	Length (bytes)	Data Type	Description
0-5	6	Don't care	Destination MAC address
6-11	6	Don't care	Source MAC address
12-13	2	0x8828	EtherType
14	1	0x01	PIMFOR version
15	1	0x00	PIMFOR get operation
16-19	4	0x00000000	PIMFOR OID
20-21	2	0x00, 0x00,	PIMFOR reserved
22-25	4	0x00000006	PIMFOR value length
26-31	6	Don't care	PIMFOR value

At this point, this packet can be sent to the WiFi device by calling the standard `write()` function.

Receiving PIMFOR Responses and Traps

Some time after writing the PIMFOR get MAC address command, the PIMFOR response will be returned. The standard `read()` function will return PIMFOR and data packets. Applications/libraries can easily distinguish the two types of packet by examining the EtherType field.

Country/Region Configuration

The WiFi country configuration is set automatically from information sent by the access point.

Management Information Block (MIB)

The USB WiFi driver maintains its MIB in a nonvolatile area of RAM (as do all drivers). The contents of the MIB are listed below. This information can be retrieved by an application using the `get_component_vars()` function.

Table 55 describes the firmware loading process.

Table 55 **Firmware Loading Process**

Description	Offset	Length
Firmware load attempts	0	4
Firmware load failures	4	4
Last firmware load result	8	4

The firmware load attempts increments by 1 at the start of the firmware loading process. The firmware load failures increments by 1 when a firmware load fails. Attempts minus failures equals successful loads. Last firmware load result holds the result code of the last firmware load attempt.

Table 56 defines the load results and their descriptions. Conditions indicated by values 1 and 2 can be fixed by correctly loading the firmware file and the corresponding signature file. The rest of the conditions indicate serious OS or hardware problems.

Table 56 **Load Results**

Load Result	Description
0	Success
1	F:#PRISMA09.X2 missing
2	F:#PRISMA09.X2 not authentic
3	Other file error
5	ROM ABORT failed
6	Firmware USB write failed
7	ROM GO failed
8	USB input TD failed
9	LINKSTATE read failed
10	LINKSTATE trap failed

On VX 520 color, the OS sends the MIB MODEL ID field to VeriCentre to identify the terminal type for download. The MODEL ID is displayed on the copyright message screen during start up.

NOTE



The MIB MODEL ID field must be set and then added to the MIB at the factory before it is downloaded into the terminal.

9-Bit Serial Interface

MDb is a connection and communications protocol connecting various vending peripherals—Coin Acceptor/Changers, Bill Validators, etc., to the Vending Machine Controller (VMC).

MDB Physical Interface

The MDB is a current loop interface running at 9600 baud, 1 start bit, 8 data bits, 1 mode bit, and 1 stop bit—eleven bits in total. The Predator platform UARTs do not directly support 9 data bits, however, future platforms may directly support it. The OS generalizes the 9-bit API to make the applications as portable as possible.

The Predator-based `read_9bit()` and `write_9bit()` functions convert the 9-bit data into an internal 8-bit protocol to the dongle. The dongle then converts the 8-bit internal protocol into 9-bit serial data.

NOTE

The Trident SOC UART primcell supports 9-bit operation but at this time no devices have been defined for Trident terminals to support.

MDB Dongle Version Number

When RESET to the MDB Dongle is released, the MDB Dongle sends its version number to the host at 19200 9N1. It is up to the application to read and/or flush the buffer of this version number.

9-Bit API

The data structure `Opn_Blk` (defined in the `svc.h` file) is used by an application to specify the baud rate, data format, and other protocol-specific parameters on Verix eVo-based terminal communications devices. Settings are initialized or reset using a `set_opn_blk()` call. The structure is passed to the device through the buffer parameter. This call must be made prior to any device read or write operations. The current structure can be obtained by calling `get_opn_blk()`.

One more data format is added to the `opn_blk` data structure. The `svc.h` file now has `Fmt_A9N1`.

CAUTION

If you set the data format to `Fmt_A9N1`, the `read_9bit()` and `write_9bit()` API must be used.

write_9bit()

This non-blocking function call is similar to `write()` except that it takes a pointer to shorts. This procedure takes the words in the buffer and sends them in the internal 8-bit protocol to the MDB Dongle for conversion to 9-bit.

NOTE



Supported by Vx700 only.

Prototype

```
int write_9bit(int com, short *buf, int count);
```

Parameters

<code>com</code>	Handle of the desired COM port.
<code>buf</code>	Pointer to a short array.
<code>count</code>	Number of shorts to send.

Return Values

Success:	The number of shorts written.
Failure:	-1 and <code>errno</code> is set if there are any errors.

read_9bit()

This is similar to `read()` except that it takes a short pointer. This procedure accepts 8-bit internal protocol data from the MDB Dongle and returns the data to the application in the buffer.

NOTE



Supported by Vx700 only.

Prototype

```
int read_9bit(int com, short *buf, int count);
```

Parameters

<code>com</code>	Handle of the desired COM port.
<code>buf</code>	Pointer to a short array.
<code>count</code>	Maximum number of shorts to read.

Return Values

Success:	0 or positive, the number of shorts read.
Failure:	-1 and <code>errno</code> is set if there are any errors.

Cinterion PHS8-P Radio Support

3G Radio

The Cinterion PHS8-P radio on VX 680 3G supports HSPA+ and the OS device driver supports the radio.

HSPA+ has a maximum downlink PHY rate of 14.4 MBPS. Real world over-the-air transfer speeds is much lower than the maximum hardware speed.

GPS Receiver

The PHS8-P radio hardware supports GPS and A-GPS (assisted GPS). The OS device driver makes the GPS port available to EOS. The GPS radio is controlled via AT commands so EOS is responsible for turning the GPS radio ON and OFF when needed, loading A-GPS data, and collecting the location information.

Predator Bluetooth Modem Support

The Vx670 OS supports the EZURIO Bluetooth I/O module on COM2 UART and the Bluetooth base station, XPBS017, with Silicon Laboratories Si2434 modem chip set.

The OS driver makes the following assumptions:

- Only one Bluetooth base station is connected at any given time.
- The Bluetooth system is set up and configured by the application/library layer, and a connection has been established.
- The Si2434 modem is set up and configured by the application/library layer similar to that of the Vx670 USB modem.
- The RTS/CTS flow control between the OS driver and the Si2434 modem has been configured.
- The applications monitor the landline modem connection via DCD, disconnect the modem using DTR, and monitor the Bluetooth connection via DSR. The DCD reflects the Si2434 modem connection state, the DSR reflects the Bluetooth connection state, and the DTR controls the Si2434 modem as if the Bluetooth layer is not present.

NOTE



The generally released Verix eVo operating system includes the Bluetooth driver support for revisions QD0009A0 and later.

Consider the following risks and dependencies when communicating with the Bluetooth modem:

- The default state of the Bluetooth module enables the RTS/CTS flow control, and the DCD and DTR monitor and control the Si2434 modem. Control of the modem in SDLC may not be possible if the Bluetooth module configuration is changed with respect to these signals.
- The default communication rate between the Vx670 terminal and the Bluetooth module must be changed from 9600 bps to 115200 bps. Failure to change this rate results in SDLC protocol failure.

- The default communication rate between the Si2434 modem and the Bluetooth base must be changed from 57600 bps to 115200 bps. Failure to change this rate results in SDLC protocol failure.
- The Bluetooth module has a default timeout of 10 seconds for dropped connections. If connection is restored within 10 seconds, the Bluetooth transmits all data captured at either end of the Bluetooth link over that ten-second period. This may have detrimental effects on the SDLC protocol and may leave a transaction in an unknown state.

NOTE

It is highly recommended that the timeout be set to the allowable minimum value.

- The transaction time for SDLC type connections is longer than the typical landline timing since additional time is needed to establish a Bluetooth connection.
- The VTM does not support Bluetooth. The download menus are not modified to show download attempts using the Bluetooth communication layer (SVC_ZONTALK does not work with Bluetooth). The modem information displays do not show Bluetooth modem information on COM2.
- The function `download()` assumes that the communication layer is connected and ready to receive and send data. It provides the ZONTALK protocol for downloading but does not have knowledge of Bluetooth or the communication layer in general. The application monitors and recovers from communication errors.

Modem Profile Loading

The Bluetooth base station employs the same Silicon Laboratories Si2434 modem chipset found in the Vx670 USB modem, thus, it uses the same modem profile used by the USB modem solution. The OS does not recognize the Bluetooth modem until the Bluetooth base station connects with the terminal.

NOTE

The Verix eVo operating system has no control when these events occur. They are dependent on the application/library layer that establishes the connection.

The application/library layer controls the modem profile download and the function `BT_Si2434_profile_load()` is used to download the modem profile.

The Verix eVo terminal does not allow an application to open COM3 if the modem profile is not yet loaded. However, the OS cannot easily enforce this for COM2 Bluetooth. Bluetooth is one of the several communication device options for COM2, and the Bluetooth module can connect to several different base station types. Attempts to prevent the use of the Bluetooth Si2434 modem until a profile is

loaded is not practical since the application/library layer must perform this action. Failure for the application/library to call `BT_Si2434_profile_load()` may result in an unexpected modem configuration or connection issues, such as dropped calls or connection failure.

The function `BT_Si2434_profile_load()` is provided to test the COM2 device and verify that the Bluetooth module with Si2434 modem is used in the base station. If the device is correct, the profile state is tested in the modem and the profile is loaded, when needed.

BT_Si2434_profile_load()

This verifies the base station type (i.e, has Si2434 SiLabs modem chip set) and downloads the profile when it is not loaded yet or when the existing modem has a different profile version. This is called after the Bluetooth base station is connected and before communicating with the SiLabs modem.

Prototype `int BT_Si2434_profile_load(void);`

Return Values

- Success: 0, successful profile load or profile already loaded.
- Failure: -1, failed profile load with the following errno:
- `ENXIO` - unable to communicate with the modem, Bluetooth not connected.
 - `EBADF` - modem profile not present.
 - `EIO` - profile load started but communication with modem failed.
 - `ENODEV` - modem in base station is not Si2434, or the I/O module in the Vx670 does not return a Bluetooth module ID.

NOTE



If the Bluetooth connection is made to a base station that does not contain the Si2434 modem, the result codes may not exactly match the above list.

The device connected may not respond to SiLabs Si2434 AT commands, making it appear that it is not communicating or not connected. It could also send an unexpected return. At this point, an error is reported. Errors are set up with the assumption that the user is connecting to a base with a Si2434 modem.

SDLC and V.80 Support

The Vx670 OS supports SDLC/V.80 protocol on COM2 and COM3. The COM2 implementation is the same as the COM3 modem, and assumes the use of the Bluetooth module. The OS does not check the module ID during the `set_opn_blk()` switch from asynchronous to synchronous, but the module ID is tested when the modem profile load function `BT_Si2434_profile_load()` is called. Once the Bluetooth device is connected and the application is communicating with the Si2434 modem in the base station, the SDLC protocol behaves like the modem in COM3.

Bluetooth Modem Application Flow

Below is the expected application flow for a Bluetooth modem SDLC transaction:

- 1 Application opens Bluetooth device, COM2.
- 2 Application sends `set_opn_blk()` to COM2 for asynchronous mode.
- 3 Application configures Bluetooth module and changes its baud rate to 115200 bps.
- 4 Application sends `set_opn_blk()` to COM2 for new 115200 bps baud rate.
- 5 Application requests a connection to the Bluetooth base station.
- 6 Application puts the Bluetooth base station module in command mode and changes base station baud rate from 57600 bps to 115200 bps.
- 7 Returns the Bluetooth layer to data mode.
- 8 Application executes `BT_Si2434_profile_load()`. If 0 is returned, Si2434 modem profile is okay—this is a blocked function and does not return until completed. If a profile is loaded, this may take 1 to 2 seconds. The profile is loaded if no profile is present or the profile to be loaded does not match the profile present in the modem. The profile persists in the Si2434 as long as power is present.
- 9 If profile load is successful, the application sets up the Si2434 modem with AT commands similar to the USB modem.
- 10 If profile load fails, application may retry or abort modem operation.
- 11 Application sends dial command to the Si2434 modem and waits for connect message.
- 12 If expected connect message is received the application issues `set_opn_blk()` with SDLC parameters.
- 13 Application writes transaction data and waits for approval message from host.
- 14 Application processes approval message.
- 15 Application notifies DTR to hang-up the modem and the Bluetooth layer.
- 16 The application calls `set_opn_blk` to return to asynchronous mode.

17 Application closes the Bluetooth connection and device.

SDLC Packet Posting

The SDLC protocol is enhanced so it does not post the received data packet that is read by the application until after the next poll or data packet indicates that the transmission is completed. This ensures that the sending site has received the acknowledgement to the data packet. See [Enhanced SDLC Protocol](#) for more information.

Verification of the SDLC protocol status is supported. This modifies the `get_port_status()` to return a running count of polls from the host in the second byte of the 4 byte `get_port_status()` argument. See [SDLC Status](#) for more information.

Bluetooth Firmware Update

The core of the Bluetooth I/O module within the Vx670 is a third party module called the Ezurio Blu2i. Using the BlueCore Serial Protocol (BCSP), the module's firmware can be updated through the UART interface. However, the BCSP source is not appropriate to be implemented at the OS driver level because the software package is very large and the source is written in C++, which is not supported at the OS level.

The OS provides the low level support necessary to put the Bluetooth module in boot load mode. When the Bluetooth module is reset, the Bluetooth device comes up in boot mode and searches for a firmware upload attempt for about 250 ms, then the AT command processor starts.

NOTE



This amount of time is too quick for the OS to perform a firmware update. A mechanism to reset the Bluetooth module independent of the power cycle is thus provided. This is accomplished by the `set_radio_ctl()` function and the `RAD_RST` bit. These functions reset the Bluetooth module and allow the application to attempt a firmware load.

Ethernet Device

Ethernet is supported on various Verix V terminals. Check the terminal Reference Manual for the capability for each terminal type. While the hardware may be different between Predator and Trident, the interface is the same for internal Ethernet. The external USB Ethernet is similar but the USB has other considerations. Although not recommended, using both internal and external Ethernet at the same time is possible in Trident terminals. The complications occur in EOS and the TCP/IP stack.

Dial Device

The Trident terminals, except for the Vx670, use the Conexant Harley modem. The Vx670 uses the SiLabs 2434 USB Modem Dongle. The Conexant Harley modem uses the same AT command set as the Conexant Banshee (Vx510, Vx610) and Eisenhower (Vx570) modems. Note that the Vx670 BT supports both the old Dione Base (SiLabs modem) and the new Vx670 Base (Harley modem) as well as the USB modem dongle. Both modem profile types can be present in the terminal at the same time.

Trident Bluetooth Support

VeriFone ported the Bluetopia Bluetooth protocol stack from Stone Street One to Verix. This is compared to the Vx670 Bluetooth implementation where the stack was embedded in the EZURIO module. The BT stack provides “profiles” for the various devices in the VX 680 and Dione bases. The PAN profile is the VX 680 Base Ethernet, DUN is the VX 680 modem, SPP is the UART simulation via Ethernet in the VX 680 base. The Dione Base is only supported via the SPP profile and the user can switch between Ethernet or modem. The VX 680 BT model assumes the user is using EOS so only the basic APIs are described.

The Bluetooth stack is an OS downloadable module that is downloaded into GID 1 and then the OS moves the driver to GID 0.

Devices

The Bluetooth Driver provides 6 Verix devices; there is one main device and 5 profile devices. The main device `DEV_BLUETOOTH` is used for configuration and control functions. The other devices provide data channels for each of the four available Bluetooth profiles. The six Bluetooth devices also generate events to the tasks that own them as appropriate.

Table 57 Bluetooth Devices

Device Name	Device String Name	Role
DEV/BLUETOOTH	DEV_BLUETOOTH	Discovery, Control, Status, Pairing, Profile Connection Configuration
DEV/BT_SPP	DEV_SPP_BT	SPP Profile Slave Data COM device
DEV/BT_SPP1	DEV_SPP1_BT	SPP Profile Slave Data COM device, Modem support in Dione Base
DEV/BT_DUN	DEV_DUN_BT	DUN Profile Slave Data COM device
DEV/BT_OBEX	DEV_OBEX_BT	OBEX Control & Data
DEV/BT_PAN	DEV_PAN_BT	PAN Profile Slave Data ETH device

DEV_BLUETOOTH

Used for the main configuration and control interface to the Bluetooth stack including the discovery and pairing functions. If events for this device are configured with the `set_event_bit()` API, events are sent to the owner of this device for inquiry results, authentication, and encryption change events.

This device is also used to set up the connection parameters for each of the Bluetooth devices. It does not open profile connections — the connections are opened when an application calls the `open()` API for the associated profile.

DEV_SPP(1)_BT

This device allows communication to a Serial Port Profile connection. The SPP connection is established when the DEV_SPP_BT device is opened and closed when the device is closed. The connection parameters are configured by the DEV_BLUETOOTH [bt_spp\(1\)_configuration\(\)](#) API prior to opening the DEV_SPP_BT or DEV_SPP1_BT device. If events for this device are configured with the [set_event_bit\(\)](#) API, then events are generated to the owner of this device for the standard serial port event types.

DEV_SPP1_BT has been modified to support the Dione base and SDLC for the modem. The Dione base does not support RTS/CTS flow control as RTS is used to toggle the base between Ethernet and landline modem. If RTS is asserted the Dione base is switched to Ethernet. The user needs to add "Fmt_auto|Fmt_RTS" to the [set_opn_blk\(\)](#) format field of the open block.

NOTE



You cannot select Fmt_AFC and Fmt_auto as they are mutually exclusive.

For both the modem and Ethernet devices the SPP port is a pass through device. No Ethernet features are part of the OS driver or the BT stack. The same is true for the modem. The Ethernet chip responds to AT commands when first opened and has a command and data mode similar to a modem. The same Ethernet functions used for the VX 680 base will not work for the Dione base Ethernet.

The Dione base modem is the SiLabs 2434 modem chip. The modem uses the same modem profile as the USB modem dongle used in the VX 680, but the VX 680 base uses the Conexant modem and uses the modem profile used for the VX 520 and other Trident terminals. The VX 680 can support both profiles in the terminal at the same time, but they must be loaded separately. You may refer to these APIs for more information.

- [close\(\)](#)
- [write\(\)](#)
- [read\(\)](#)
- [set_event_bit\(\)](#)
- [get_event_bit\(\)](#)
- [set_opn_blk\(\)](#)
- [get_opn_blk\(\)](#)
- [get_port_status\(\)](#)
- [reset_port_error\(\)](#)
- [barcode_pending\(\)](#)
- [get_bits_per_second\(\)](#)

- `set_fifo_config()`
- `get_fifo_config()`
- `openaux()`
- `closeaux()`

The serial device generates events when serial characters are received. It is safe to attempt a read any time an event is received.

DEV_DUN_BT

This device allows communication to a Dial Up Modem profile connection. It is designed to look as much as possible like a standard Verix COM device. The DUN connection is established when the `DEV_DUN_BT` device is opened, and disconnected when the device is closed. The connection parameters are configured by the `DEV_BLUETOOTH` `bt_dun_configuration()` API prior to opening the `DEV_DUN_BT` device. If events for this device are configured with the `set_event_bit()` API, then events are generated to the owner of this device for the standard serial port event types.

You may refer to these APIs for more information.

- `open()`
- `close()`
- `write()`
- `read()`
- `set_event_bit()`
- `get_event_bit()`
- `set_opn_blk()`
- `get_opn_blk()`
- `get_port_status()`
- `reset_port_error()`
- `barcode_pending()`
- `get_bits_per_second()`
- `set_fifo_config()`
- `get_fifo_config()`
- `openaux()`
- `closeaux()`

The `DEV_DUN_BT` device generates events when serial characters are received. It is safe to attempt a read any time an event is received.

DEV_OBEX_BT

This device allows an application to transfer files from the terminal to an access point. The Bluetooth address and server port number are configured through the DEV_BLUETOOTH [bt_dun_configuration\(\)](#) API prior to opening DEV_OBEX_BT. To transfer a file to the access point, an application opens the DEV_OBEX_BT device with the [open\(\)](#) API, then writes the file name of the file to transfer with the [write\(\)](#) API. The [write\(\)](#) API returns 0 when the file transfer is complete or -1 if there was an error. You may refer to these APIs for more information.

- [open\(\)](#)
- [close\(\)](#)
- [bt_obex_push_file\(\)](#)

DEV_PAN_BT

This Device allows communication over a Personal Area Network (PAN) connection to an access point. It is designed to look as much as possible like a standard Verix ETH device. The connection parameters are configured by the DEV_BLUETOOTH [bt_dun_configuration\(\)](#) API prior to opening the DEV_PAN_BT device. The PAN connection is established when the DEV_PAN_BT device is opened and closed when the device is closed. Once a connection is open, this device looks like an Ethernet packet driver to a protocol stack running above it (DEV_ETH*). If events for this device are configured with the [set_event_bit\(\)](#) function, events are generated to the owner of this device for the usual DEV_ETH event types. This device also supports an aux device for monitoring the connection status. You may refer to these APIs for more information.

- [open\(\)](#)
- [close\(\)](#)
- [read\(\)](#)
- [write\(\)](#)
- [openaux\(\)](#)
- [closeaux\(\)](#)
- [get_enet_MAC\(\)](#)
- [get_enet_status\(\)](#)
- [set_enet_rx_control\(\)](#)
- [set_event_bit\(\)](#)
- [get_event_bit\(\)](#)

APIs Each device supports [open\(\)](#), [read\(\)](#), [write\(\)](#), [close\(\)](#), [openaux\(\)](#), [closeaux\(\)](#), [set_opn_blk\(\)](#), [get_opn_blk\(\)](#), [barcode_pending\(\)](#), and [get_port_status\(\)](#) as appropriate for the device. DEV_PAN does not need the UART specific calls such as [set_opn_blk\(\)](#). The following APIs are also supported.

Event Mechanism APIs

The Verix Bluetooth stack has a unique event mechanism. Each of the six Bluetooth devices is capable of generating events to the tasks that own them. By default, no events are generated. In order to enable event notification, events must be configured for each Bluetooth device after opening it. The `set_event_bit()` function allows the owner of each Bluetooth device to select which Verix event the device generates. By default the Bluetooth devices does not generate events.

In addition to the Verix event bits, the Bluetooth stack maintains a set of Bluetooth event bits. These Bluetooth event bits are handled in EOS and are not discussed here.

- `set_event_bit()`
- `get_event_bit()`
- `bt_peek_event()`
- `bt_clear_event()`
- `bt_enable_events()`
- `bt_disable_events()`

The events generated by `DEV_PAN_BT` follows the model of the `DEV_ETH1` Ethernet device.

Table 58 **DEV_PAN_BT Events**

Device	Event Causes	Application Response
DEV_PAN_BT	Rx frame complete or overflow	Read the packet
(AUX DEVICE)	Connect or disconnect	Read the link status

Master / Slave Handling

When connecting to any Bluetooth device, the Verix Bluetooth stack starts out as a master, but enables master/slave role switching using the Bluetooth `bt_L2CA_Set_Link_Connection_Configuration()` API. It is expected that an access point will switch over to being master after the initial connection. Any peripheral that is not a stack acts as a Slave to the access point and a master to the peripheral simultaneously, but at reduced performance.

Out of Range Detection

The Verix Bluetooth stack detects the access point going out of range by monitoring the link supervision timeout. The timeout defaults to 20 seconds. Once a disconnection occurs, the driver attempts to reconnect by re-opening the individual profile connections that are currently opened. While the link is in the disconnected state, any attempt to write to a Bluetooth device will return an `EPIPE` error.

Note that when using the landline modem in either the VX 680 base or the Dione base , the SDLC protocol reconnection after an out of range may still result in a dropped modem session. The SDLC is a polled protocol and the master is the host that has been dialed. If the host times out because of no response to the polling the host will disconnect. The polling timeout varies from host to host so please discuss with the host provider for details. This can range for a couple of seconds to maybe 30 seconds but the shorter time is probably closer to reality.

set_event_bit()

If an application wants to receive events from a Bluetooth device that it owns, it must use `set_event_bit()` to specify which event bit should be used. The example shown below will all cause bit 7 to be the Bluetooth device event.

```
set_event_bit(ctlsHdl, 0x80);
set_event_bit(ctlsHdl, 128) ;
set_event_bit(ctlsHdl, (1L<<7 ));
#define MY_CTLs_EVENT 0x80
set_event_bit(ctlsHdl, MY_CTLs_EVENT) ;
```

Some event bits are reserved by the OS. This means they cannot be selected by `set_event_bit()`. The reserved event bits are `EVT_USER`, `EVT_SHUTDOWN`, `EVT_SYSTEM`, `EVT_IFD_READY`.

It is the responsibility of the application to clear the event bits using `read_event()` before calling `set_event_bit()`. This prevents possible confusion if there is a pre-existing Bluetooth module event bit set at the time `set_event_bit()` is called. It is possible for the application to see two EVENT bits set the next time the Bluetooth device generates an event, the previous Bluetooth device event bit and the new Bluetooth device event bit. As a result, it would only see the first event after `set_event_bit()` is called. All subsequent Bluetooth device events would set only the new EVENT bit.

If the application uses `set_event_bit()` to define an EVENT and then calls `close()`, the OS will clear the EVENT setting. This means the application must call `set_event_bit()` each time it opens the Bluetooth module.



WARNING

The OS will not change the predefined event bits in `svc.h` when the application calls `set_event_bit()`. When the application uses `set_event_bit()` to set `EVT_MAG` for the Bluetooth module, the application will receive `EVT_MAG` for both the mag card and the Bluetooth module.

Prototype

```
int set_event_bit(int hdl, long flag);
```

Parameters

handle	The Bluetooth device handle returned by <code>open()</code>
flag	The bit map value with the event bit to be used.

Return values

0	Successful If the application calls <code>set_event_bit(ctlsHdl, 0)</code> the OS turns off events for the Bluetooth device and returns 0 to indicate a successful response.
-1	If the application calls <code>set_event_bit()</code> with one of the reserved event bits, <code>set_event_bit()</code> will return -1 with <code>errno</code> set to <code>EINVAL</code> .

If the application calls `set_event_bit()` with a flag that defines more than 1 event bit, the OS will return -1 with `errno` set to `EINVAL`. For example, `set_event_bit(ctlsHdl, 0x90)` will return an error.

If the application calls `set_event_bit()` with handle set to any device handle that does not support soft events, `set_event_bit()` will return -1 with `errno` set to `EINVAL`.

get_event_bit()

Allows applications to find out what event will be generated by a Bluetooth device. This function returns the event bit currently set in the Bluetooth device driver.

Prototype `long get_event_bit(int hdl);`

Parameters

:	handle	The Bluetooth device handle returned by <code>open()</code>
---	--------	---

Return Values

:	0	Bluetooth device is in default mode — no events will be generated.
:	-1	Verix V will return -1 with <code>errno</code> set to <code>EBADF</code> when any API has passed an invalid handle – meaning handle is set to any value other than what is returned by a successful <code>open()</code> call.

NOTE



See *Verix V Programmers Reference Manual*, Table 5 Error Codes Set by Function Calls, for an explanation of Verix V API `errno` handling.

bt_peek_event()

Allows applications to determine the cause or causes of Bluetooth events. Calling `bt_peek_event()` does not clear the event or have any other side effect. When an application handles an event cause, it must clear the corresponding bit using the `bt_clear_event()` API. There may be more than one bit set at any time. The event bit definitions are defined in `bt_lib.h`. The names of the bits are given in the event tables in the sections describing each Bluetooth device. The names begin with “BT_EVENT_BIT_”.

Prototype

```
int bt_peek_event(int hdl, DWord_t *event_mask);
```

Parameters

<code>handle</code>	The Bluetooth device handle returned by <code>open()</code>
<code>*event_mask</code>	Pointer to a 32-bit location where the current set of Bluetooth even bits are copied.

bt_clear_event()

Allows applications to clear the Bluetooth events. The application must set each bit in the `event_mask` parameter that corresponds to an event type to be cleared. Multiple event types can be cleared simultaneously. The event bit definitions are defined in `bt_lib.h`. The names of the bits are given in the event tables in the sections describing each Bluetooth device. The names begin with “BT_EVENT_BIT_”.

Prototype

```
int bt_clear_event(int hdl, DWord_t event_mask);
```

Parameters

<code>handle</code>	The Bluetooth device handle returned by <code>open()</code>
<code>event_mask</code>	32-bit variable containing a bit for each Bluetooth event type.

bt_enable_events()

Allows the owner of each Bluetooth device to choose which Bluetooth events it wishes to receive events notification from. For each bit that is set in the “events” parameter, the corresponding Bluetooth event is enabled. When enabled, the owner of the device will receive an event notification whenever the Bluetooth event bit gets set. When not enabled, the owner of the device will not receive an event when the Bluetooth event bit gets set. This function does not prevent Bluetooth event bits from getting set.

Prototype `int bt_enable_events(int hdl, long events);`

Parameters

handle	The Bluetooth device handle returned by open()
events	Bit map value with the event bits to enable set.

The sample code shown below enables enables the BT_EVENT_BIT_INQUIRY_RESULT and the BT_EVENT_BIT_AUTHENTICATION Bluetooth events to notify the owner of the Bluetooth device with the BTHdl handle. Any other Bluetooth event bit will not generate an event to that device when the Bluetooth event bit is set.

```
Bt_enable_events(BTHdl, BT_EVENT_BIT_INQUIRY_RESULT |
BT_EVENT_BIT_AUTHENTICATION);
```

bt_disable_events()

This function call is the inverse of `bt_enable_events()`. Any previously set bits can be cleared in the event mask. For each bit that is set in the “events” parameter, the corresponding Bluetooth event is disabled.

Prototype `int bt_disable_events(int hdl, long events);`

Parameters

handle	The Bluetooth device handle returned by open()
events	Bit map value with the event bits to enable set.

In the example below, events BT_EVENT_BIT_INQUIRY_RESULT and BT_EVENT_BIT_AUTHENTICATION will no longer be received.

```
Bt_disable_events(BTHdl, BT_EVENT_BIT_INQUIRY_RESULT |
BT_EVENT_BIT_AUTHENTICATION);
```

Bluetooth API Library

Each of the Bluetooth API commands in the `BT_API.LIB` utilizes the control/status functions to process a command. The first step initiates function execution, using a call to the control function of the associated device handle. When the control function returns, it waits for an event to indicate the initiated function has completed. Once the completion event is received, the status function is called to retrieve the results of the function. The `CONFIG.SYS` variable `*BTTOUT` indicates in timeout value in milliseconds. The default timeout value is 15000 or 15 seconds in time.

NOTE

Not all the APIs provided in `BT_API.LIB` are documented. Most of these are supported in EOS and should be of no interest to applications. `BT_API.LIB` is not provided in the standard SDK for Verix Applications. This is provided with the Bluetooth SDK.

DEV_BLUETOOTH APIs

The `DEV_BLUETOOTH` device is the main configuration and control interface to the Bluetooth stack including the discovery and pairing functions. This device is supported via EOS.

- `bt_perform_inquiry()`
- `bt_bluetooth_configuration()`
- `bt_spp(1)_configuration()`
- `bt_dun_configuration()`
- `bt_obex_configuration()`
- `bt_pan_configuration()`
- `bt_sdp_search_services()`
- `bt_version_string ()`
- `bt_mdm_profile_load()`

bt_perform_inquiry()

Performs a device inquiry and returns the number of Bluetooth devices found. The caller provides a pointer to a `device_table_t` structure. The driver populates the table with the information about the devices found.

Prototype `int bt_perform_inquiry(device_table_t *device_table);`

bt_bluetooth_configuration()

Configures the bluetooth connection character. Parameter `pairMode` receives the following choices:

- `pmNonPairableMode`
- `pmPairableMode`
- `pmPairableMode_EnableSecureSimplePairing`

Prototype `int bt_bluetooth_configuration(GAP_Pairability_Mode_t pairMode);`

bt_spp(1)_configuration()

Configures the connection established when an application opens the DEV_SPP_BT or DEV_SPP1_BT device. The device can act as either an Sequenced Packet Protocol (SPP) server or a client. The Boolean parameter “server” selects server mode when true and client mode when false.

Prototype

```
int bt_spp_configuration(BD_ADDR address, unsigned int port, char
*service_name, boolean server);
int bt_spp1_configuration(BD_ADDR address, unsigned int port, char
*service_name,boolean server);
```

Parameters

Server mode parameters:

address	Ignored for server mode.
port	The port on which the server should provide the SPP service.
service_name	A null-terminated ASCII string containing a name for the service as it will appear in the SDP entry.
server	Set to True.

Client mode parameters:

address	The address to connect to when opening a connection.
port	The port to connect to when opening a connection.
service_name	Ignored may be set to NULL.
server	Set to False.

bt_dun_configuration()

Configures the Bluetooth address and port number of the DUN port to which the stack should establish a connection when an application opens the DEV_DUN_BT device. The port number used is the value reported by the Sockets Direct Protocol (SDP) query.

Prototype `int bt_dun_configuration(BD_ADDR address, unsigned int port);`

bt_obex_configuration()

Configures the Bluetooth address and port number of the access point port to which the stack should transfer files when an application opens the DEV_OBEX_BT device and provides a file name. The port number used is the value reported by the SDP query.

Prototype `int bt_obex_configuration(BD_ADDR address, unsigned int port);`

bt_obex_push_file()

To transfer a file using the DEV_OBEX_BT device, an application must first configure the OBEX connection through the DEV_BLUETOOTH device with the [bt_obex_configuration\(\)](#) API. Then, an application opens the DEV_OBEX_BT device. The application then calls `bt_obex_push_file()` with the name of the Verix file to be transferred. This function returns when the file transfer is complete, or when an error occurred. If the file transfer was successful, this function returns the number of bytes transferred. If an error occurred a value less than zero is returned. If desired, another file may be transferred by calling `bt_obex_push_file()` again. After all files are transferred, the application closes the DEV_OBEX_BT device with the [close\(\)](#) API. The DEV_OBEX_BT device does not generate any events.

The second parameter passed to `bt_obex_push_file()` is a pointer to an application allocated structure where download progress is updated periodically.

```
typedef enum {
    OBX_IDLE=0,
    OBX_START,
    OBX_INPROGRESS,
    OBX_DONE,
    OBX_NUM_STATES
}OBEX_STATUS_TYPE;

typedef struct {
    OBEX_STATUS_TYPE status;
```

```
        unsigned long written;  
        unsigned long length;  
    }OBEX_STATUS_STRUCT_t;
```

The `OBEX_STATUS_STRUCT_t` is updated by the `bt_obex_push_file()` function during download. An independent task may be spawned to show progress of the file download. This is useful for firmware updates where the download takes several minutes to complete. The download status is set to `OBX_START` when entering `bt_obex_push_file()`. Once the download file has been found status is changed to `OBX_INPROGRESS`. When the download is completed the status is changed to `OBX_DONE`.

bt_pan_configuration()

Configures the Bluetooth address of the access point to which the stack should connect when an application opens the `DEV_PAN_BT` device. The port number use is the value reported by the SDP query. The local and remote service type is provided. See `PANAPI.H` for more information on service type selections.

Prototype

```
int bt_pan_configuration(BD_ADDR address, int portNumber,  
PAN_Service_Type_t localType, PAN_Service_Type_t remoteType);
```

bt_sdp_search_services()

Returns the services supported by a remote connection. The return value is a bit mask containing one bit for each supported service. Once this API is called, the `BT_EVENT_BIT_SEARCH_SERVICES_RESPONSE` event must be waited to indicate that the SDP has completed. Once completed, `bt_retrieve_sdp_search_list()` must be called to retrieve the Universally Unique Identifier (UUID) list from the Bluetooth Driver (BD).

The Bluetooth support service bits are:

- `BT_SPP_SERVICE`
- `BT_DUN_SERVICE`
- `BT_PAN_SERVICE`
- `BT_OBEX_SERVICE`

Prototype `int bt_sdp_search_services(BD_ADDR address);`

bt_version_string ()

Retrieves various version strings available from either the Verix Bluetooth Driver or the Bluetooth Stack. A string pointer is provided where the version information is stored. The `allocatedSize` parameter indicates the size of the string pointer passed in. The data returned is in ASCII format.

The version components strings are:

- `bt_driver_version`
- `bt_stack_version_number`
- `bt_stack_product_name`
- `bt_stack_company_name`
- `bt_stack_copyright`

Prototype `int bt_version_string (bt_component_t component, char *versionString, int allocatedSize);`

bt_mdm_profile_load()

Loads the modem profile to the modem. This function is called after the modem is opened and configured with `set_opn_blk()`. The function sends an AT command to the modem to determine the modem type, either the Conexant or SiLabs modem. Then selects the appropriate modem profile and loads. If the modem profile is not present the load fails.

Prototype `int bt_mdm_profile_load (int hdl);`

General Communication Device Functions

This section describes APIs that are used by all of the Verix eVo on-board communication devices.

These standard Verix eVo communication port APIs are

You may also refer to these APIs:

- `open()`
- `close()`
- `read()`
- `write()`
- `set_opn_blk()`
- `get_opn_blk()`
- `get_fifo_config()`
- `set_fifo_config()`
- `get_port_status()`
- `reset_port_error()`
- `set_serial_lines()`
- `get_component_vars()`

Error Conditions and Error Codes

Errors are reported by returning a result of -1 with `errno` set to a specific standard error code. The caller will receive error codes in the following situations:

<code>EINVAL</code>	No <code>set_opn_blk()</code> command issued or the caller's buffer is an invalid size. <code>set_opn_blk()</code> already called.
<code>ENXIO</code>	USB device not present.
<code>EBADF</code>	<code>comm_handle</code> or <code>fd</code> is not a valid open file descriptor.
<code>EACCESS</code>	Pointer is not valid. Access violation attempting to read from buffer.

download()

Receives a download through the open serial port specified by `handle`. It is similar to [SVC_ZONTALK\(\)](#) (which in fact calls it), but allows a download through any serial port.

Of the `CONFIG.SYS` variables described under [SVC_ZONTALK\(\)](#), only `*ZT` and `*ZA` are used by this function.

Prototype

```
int download (int hdl, void *parms);
```

Parameters

<code>handle</code>	Open serial port.
<code>parms</code>	Points to a one-character string which is the same as the <code>type</code> parameter passed to <code>SVC_ZONTALK()</code> , that is "F", "P", "p", "R", or "r". The "no dial" and "alternate modem" modifier bits are not recognized.

Return Values

Success:	99: Successful download. 100: Successful download, but no meaningful packets received.
Failure:	-1: User pressed CANCEL key. -3: Received too many NAKs. -4: Remote host sent "U" packet. -6: <code>write()</code> to file failed. -7: Timed out waiting for host. -8: Lost carrier from modem. -9: Sent too many NAKs. -10: <code>set_timer()</code> failed. -11: <code>write()</code> to COMx failed. -13 and <code>errno</code> set to <code>-EACCES</code> : Group access violation. -14: Timed out waiting for ENQ.

Other return values are the same as [SVC_ZONTALK\(\)](#) (although many modem-specific error codes do not apply).

If the download fails, the user *must* reset the terminal, manually enter VTM, and reinitiate the download.

set_serial_lines()

The standard Verix eVo communication port `set_serial_lines()` API is not applicable to the printer. COM1 and COM3 have automatic flow control (Fmt_AFC). When Fmt_AFC is set, RTS cannot be controlled by `set_serial_lines()`. RTS is controlled by the driver. If RTS is select and Fmt_AFC is set, this call is rejected and errno is set to EINVAL.

Attempting to drop carrier by using the following bit combinations when using the Huawei CDMA radio modem and calling `set_serial_lines()` will not clear DTR:

0x04 (BRK=1, RTS=0 & DTR=0)

0x00 (BRK=0, RTS=0 & DTR=0)

Alternatively, the radio responds to `set_serial_lines()` to drop DTR if the AT connect command (ATD#777) is issued over COM2—the endpoint used for PPP connections.

If the initial connection is made over COM9, the endpoint generally used for AT commands, the modem ignores the command and sends no response to the driver.

It is assumed that buffer points to a single byte with bits set for each of these signals:

00	-BRK	-RTS	-DTR
01	-BRK	-RTS	+DTR
02	-BRK	+RTS	-DTR
03	-BRK	+RTS	+DTR
04	+BRK	-RTS	-DTR
05	+BRK	-RTS	+DTR
06	+BRK	+RTS	-DTR
07	+BRK	+RTS	+DTR

Note that asserting BRK does not imply any form of system-supplied time out for stopping the condition. The application must provide this mechanism. Calling `set_serial_lines()` while data are pending for transmission may result in loss of some data.

If hardware flow control is enabled, the RTS line is unaffected by `set_serial_lines()` since its state is dictated by the requirements of hardware flow control. The BRK state is still affected as indicated in the above table.

When doing `set_serial_lines` or `set_opn_blk()` to USB devices, there may be a brief glitch in the state of the DTR or RTS signals. This can cause the modem to disconnect even if the desired state of the signal is not changed.

If the state of DTR or RTS is not changed, do not include the option in the call. Setting the DTR and RTS in the USB device is caused by sending a low level USB command to the USB device. This command then sets the state as applicable. Setting the state of the signal will cause the state to be unknown for a brief time, and is a function of the HW used. Thus, when doing a synchronous connection and it is time to change from Async to Sync using the `set_opn_blk` call, do not include the `Fmt_DTR` or `Fmt_RTS` options as the modem will disconnect. The same goes for `set_opn_blk()`.

On BT, this function allows the application to control the virtual modem lines over the SPP connection and send break conditions. Specifically, this API allows the application to control the Bluetooth "RTR" signal which is mapped to the RTS Verix modem line output.

set_fifo_config()

The standard Verix eVo communication port `set_fifo_config()` API is not applicable to the printer. There is no UART hardware so this function always returns 0. The contents of settings are stored in transient memory. Calls to `get_fifo_config()` return the stored settings. `set_fifo_config()` sets the ARM UART and receive (Rx) FIFO length as follows:

buf, Byte 0, bit 0:	1 = FIFO enabled; 0 = FIFO disabled
buf, Byte 0, bits 1–5 unused	
buf, Byte 0, bits 6–7: Receive FIFO settings:	00 = 4 bytes; 10 = 12 bytes (default); 01 = 8 bytes; 11 = 15 bytes
buf, Byte 1: Transmit (Tx) FIFO settings:	00 = empty; 10 = 10 bytes; 01 = 4 bytes; 11 = 12 bytes (default)

The transmit and receive FIFOs are 16 bytes deep. The above settings determine when an interrupt is generated. The Rx FIFO default value is 12 bytes. When 12 bytes of data fill the Rx FIFO an interrupt is generated. If the FIFO is below 12 bytes and no data is received in 3 byte times (dependent on baud rate), an interrupt is generated. The ISR then moves data from the FIFO to a buffer.

The transmit FIFO default setting is 12 bytes. If the FIFO level falls below this value, an interrupt is generated. The ISR then adds more data to the FIFO from a buffer. This function is provided solely for performance tuning for high-speed communication. Under normal circumstances, the defaults should be more than adequate. Poor configuration settings can adversely affect communication performance.

To avoid loss of data, `set_fifo_config()` can only be called after calling `open()`, but before calling `set_opn_blk()`.

On BT, this function is for backwards compatibility, it does nothing but store the requested values.

get_fifo_config()

The standard Verix eVo communication port `get_fifo_config()` API is not applicable to the printer. There is no UART hardware so this function always returns 0. The values from the last call to `set_fifo_config()` are copied to settings.

On BT, this function is for backwards compatibility, it does nothing but return the requested values.



Security/Crypto Library

This chapter describes Verix eVo functions calls related to security and the crypto libraries.

- [Security Services Functions](#) detail the security functions related to the following:
 - [Crypto Functions](#) discuss functions related to AES, RSA, and SHA-1.
 - [File Encryption Support Functions](#) discuss functions related to PIN attack and tamper detection.
- [VeriShield Security Script Functions](#) detail the functions that support the key management schemes beyond the default DUKPT and PIN and MAC Master Session key schemes.
 - [VSS PIN Entry Functions](#) list and discuss PIN management.
 - [Key Loading Functions](#) list and discuss VSS key loading.

See [Support for APACS40 Cryptographic Functions](#) for more information on APACS40 support. See [IPP Key Loading](#) and [IPP Communications Packets](#) for more information on master session and DUKPT keys.

Security/Crypto Library Functions:

- | | | |
|---|---|---|
| • rsa_calc() | • crypto_write() | • crypto_read() |
| • AES() | • SHA1() | • DES() |
| • iPS_SetPINParameter() | • GenerateRandom() | • isAttacked() |
| • iPS_GetPINResponse() | • iPS_SelectPINAlgo() | • iPS_RequestPINEntry() |
| • iPS_InstallScript() | • iPS_CancelPIN() | • pcPS_GetVSSVersion() |
| • iPS_ExecuteScript() | • iPS_UninstallScript() | • get_tamper() |
| • CheckKeyAreaIntegrity() | • SHA256() | |

Security Services Functions

The security device (`/dev/crypto`) does not need to be open to use the following functions. All functions, data structures, and symbols are defined in the `<svc_sec.h>` header file.

File Encryption Support

- `crypto_read()`
- `crypto_write()`

Cryptographic Functions

- `SHA1()`
- `SHA256()`
- `rsa_calc()`
- `DES()`
- `AES()`

Random Number Generation

- `GenerateRandom()`

Attack Detection

- `isAttacked()`
- `get_tamper()`
- `CheckKeyAreaIntegrity()`

APACS40 Support (see Support for APACS40 Cryptographic Functions)

- `Init_MAC()`
- `Create_MAC_Key()`
- `Calc_Auth_Parm()`
- `Calc_MAC()`
- `New_Host_Key()`
- `Reset_Key()`
- `Term_MAC()`

File Encryption Support Functions

This feature can be used to guarantee that the file content will be lost if the unit is tampered with. The file is encrypted with a variant of a key that is erased from the terminal in case of attack, making it impossible to recover the content of the encrypted file. The key is unique per 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 the transaction batch file.

The Verix V terminals have incorporated into their design methods of determining terminal tampering. Tamper is reported for changes to the state of the terminal such as opening the case or temperature changes as examples. When a tamper occurs, the tamper event will be time stamped. The time stamp value is then stored in the tamper event time stamp registers. The function `AttackHandling()` will be changed to record the event and time stamp registers, clear the `Clear_Tamper_Registers` bit, and then re-enable the tamper detection circuitry.

This function is called when the tamper interrupt happens, or at power up the Tamper Input Status Registers show a Tamper State, or an invalid GEK is read from the KEY RAMs, or a valid random seed cannot be obtained from the hardware Random Number Generator. The function `TamperLog()` is added through `_security_services` to pass out the tamper event registers, and the time and date information of the last tamper event.

The `ERROR & TMPR LOGS` on VTM displays the details of the last tamper event. Substantial frequency deviations may not be logged as tamper events if the frequency deviations do not allow the chip to operate at all.

crypto_read()

Reads a maximum of count bytes of encrypted data from the open file associated with hdl, decrypts the data, and stores the result in the buffer. It returns the number of bytes actually read, which may be less than count if fewer bytes are available.

This allows applications to write encrypted files so the data cannot be read on another terminal or a PC. The crypto_read function accomplishes this by using a randomly generated key that is unique per terminal. The unique key prevents the data from being read on another terminal but applications must be aware of the consequences.

Back-to-back download copies all files, including the encrypted files, but the receiving terminal is not able to decrypt the encrypted files from the sending terminal because it has a different key. Applications can detect this by writing a signature value to encrypted files and checking for the signature value when reading. If the correct signature value is not present, the application should delete the encrypted files because they were copied from another terminal.

If a terminal is attacked/tampered, all keys are erased and new random keys are generated. This means encrypted files in the terminal cannot be read because the key has been erased. Encrypted data stored prior to the attack cannot be recovered. Applications can determine if the attack/tamper condition exists by calling the isAttacked() function.

Prototype

```
int crypto_read (int hdl, char *buf, int count);
```

Parameters

hdl	File handle.
buf	Pointer to the buffer holding the input data.
count	Number of bytes to read.

Return Values

Success:	0, executed
Failure	-1 with errno set to EBADF, invalid file handle.
	-1 with errno set to EACCES, invalid buffer pointer.
	-1 with errno set to EINVAL, invalid count value (negative).
	-1 with errno set to EIO, unit has been attacked, no key to decrypt the data.

crypto_write()

Encrypts and writes count bytes of data from buffer to the open file associated with `hdl`.

The `crypto_write()` returns the number of bytes actually written. Writes complete before the function returns. All writes must be done going forward in the file because data at one location affect the decryption of the data further in the file.

NOTE



The file must be opened for both reading and writing. That is, if the file was opened with the `O_WRONLY` flag set, the function returns -1 and `errno` set to `EBADF`.

This allows applications to write encrypted files so the data cannot be read on another terminal or a PC. The `crypto_write()` function accomplishes this by using a randomly generated key that is unique per terminal. The unique key prevents the data from being read on another terminal but applications must be aware of the consequences.

Back to back download copies all files including encrypted files but the receiving terminal is not able to decrypt the encrypted files from the sending terminal because it has a different key. Applications can detect this by writing a signature value to encrypted files and checking for the signature value when reading. If the correct signature value is not present, the application should delete the encrypted files because they were copied from another terminal.

If a terminal is attacked/tampered, all keys are erased and new random keys are generated. This means encrypted files in the terminal cannot be read because the key has been erased. Encrypted data stored prior to the attack cannot be recovered. Applications can determine if the attack/tamper condition exists by calling the `isAttacked()` function.

Prototype

```
int crypto_write (int hdl, const char *buf, int count);
```

Parameters

<code>hdl</code>	File handle.
<code>buf</code>	Pointer to the buffer holding the input data.
<code>count</code>	Number of bytes to write.

Return Values

Success:	0, executed
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> , invalid file handle.
	-1 with <code>errno</code> set to <code>EACCES</code> , invalid buffer pointer.
	-1 with <code>errno</code> set to <code>EINVAL</code> , invalid count value (negative).
	-1 with <code>errno</code> set to <code>EIO</code> , error on device or unit is attacked, cannot generate the encrypting key.

- 1 with `errno` set to `ENOSPC`, insufficient space left in file system.
- 1 with `errno` set to `EFBIG`, write would cause file to exceed maximum length.
- 1 with `errno` set to `EPIPE`, write to unconnected pipe.

Crypto Functions

The following security device algorithms are implemented in Verix eVo-based terminals:

- DES and 3DES
- AES
- RSA
- SHA-1

This section describes the functions used to perform general-purpose computations based on those algorithms.

AES()

Performs AES computations on 128-bit data block. The operation type and key length are specified using the `ucAesOption` parameter.

Prototype

```
int AES(unsigned char ucAesOption, unsigned char * pucAesKey8N, unsigned
char * pucInputData, unsigned char * pucOutputData);
```

Parameters

<code>ucAesOption</code>	Specifies the operation type and key length: <ul style="list-style-type: none"> • AES128E (04h) AES encryption using a 128-bit key • AES128D (05h) AES decryption using a 128-bit key • AES192E (06h) AES encryption using a 192-bit key • AES192D (07h) AES decryption using a 192-bit key • AES256E (08h) AES encryption using a 256-bit key • AES256D (09h) AES decryption using a 256-bit key
<code>pucAesKey8N</code>	Pointer to 8N-byte key block (N=2, 3 or 4).
<code>pucInputData</code>	Pointer to 16-byte input block.
<code>pucOutputData</code>	Pointer to 16-byte output block.

Return Values

Success: 0, executed.

Failure: -1 with `errno` set to `EACCES`, invalid buffer pointer.

DES()

Performs DES, DESX, and 3DES computations. The operation type and key length are specified using the `ucDeaOption` parameter.

Prototype

```
int DES(unsigned char ucDeaOption, unsigned char * pucDeaKey8N,  
        unsigned char * pucInputData, unsigned char * pucOutputData);
```

Parameters

<code>ucDeaOption</code>	Algorithm: <ul style="list-style-type: none">• DESX1KE (02h): DEAX encryption with single-length key• DESX1KD(03h): DEAX decryption with single-length key• DESX2KE (04h): DEAX encryption with double-length key• DESX2KD(05h): DEAX decryption with double-length key• DESX3KE (06h): DEAX encryption with triple-length key• DESX3KD(07h): DEAX decryption with triple-length key• DESE (08h): DEA encryption with single-length key• DESD (09h): DEA decryption with single-length key• TDES2KE (0Ch): TDEA encryption with double-length key• TDES2KD (0Dh): TDEA decryption with double-length key• TDES3KE (0Eh): TDEA encryption with triple-length key• TDES3KD (0Fh): TDEA decryption with triple-length key
<code>pucDeaKey8N</code>	Pointer to 8N-byte key block (N=1, 2 or 3).
<code>pucInputData</code>	Pointer to 8-byte input block.
<code>pucOutputData</code>	Pointer to 8-byte output block.

Return Values

Success: 0, executed

Failure: -1 with `errno` set to `EACCES`, invalid buffer pointer.

GenerateRandom()

Returns an 8-byte random value.

Prototype `int GenerateRandom(unsigned char * random8);`

Parameters

<code>random8</code>	Pointer to the 8-byte buffer where the random value transfers.
----------------------	--

Return Values

Success:	0, executed
Failure:	-1 with <code>errno</code> set to <code>EACCES</code> , invalid buffer pointer.

isAttacked()

Indicates if an attack occurred, causing the loss of the transaction keys or encrypted files. It returns 0 if no attack occurred since the last key loading or file encryption, 1 if otherwise. It also returns 1 if the unit has never been injected with a key and no encrypted file has been written.

Prototype `int isAttacked(void);`

Return Values

Success:	0, no attack occurred since keys were last loaded.
Failure:	1, an attack occurred and encrypted files are lost.

get_tamper()

Returns tamper information for the last 6 tampers.

Prototype `int get_tamper (int index, unsigned char *tamperInfo);`

Parameters

index	The tamper to get. Value is from 0 - 5, where 1 is the tamper before the last tamper, and so on.
tamperInfo	The pointer to a 10 byte buffer. <ul style="list-style-type: none">• tamperInfo[0] security barrier flag.• tamperInfo[1,2,3] tamper bit mask.• tamperInfo[4..9] 6 byte BCD time: yymmddhhmmss.

Return Values Always 0.

CheckKeyAreaIntegrity()

Checks the integrity of the encrypted key files. If any corruption is detected, all keys are deleted, the unit is restarted and a new GEK is generated.

Prototype `int CheckKeyAreaIntegrity (void);`

Return Values

Success:	0, keys OK.
Failure:	No return, unit restarts.

rsa_calc()

Performs a public key RSA computation. It supports keys up to 2048 bits and exponent values of 2, 3, and 65537.

Prototype

```
int rsa_calc(unsigned short * msg, unsigned short * mod, int wds, int exp,
             unsigned short * result);
```

Parameters

<code>msg</code>	Array of unsigned 16-bit integers holding the input.
<code>mod</code>	Array of unsigned 16-bit integers holding the modulus. High bit <i>must</i> be set. Typically an RSA modulus always has the most-significant bit set. This implementation verifies and enforces that, because if the bit is not set, it might cause unexpected behaviors. The high bit of the modulus is the MSB of the first 16-bit integer of the array.
<code>wds</code>	Number of 16-bit unsigned shorts in <code>msg</code> and <code>mod</code> .
<code>exp</code>	Code for exponent: actual exponent is $2^{\text{exp}+1}$. Acceptable values are 0, 1, 16 that correspond to exponents of 2, 3, and 65537, respectively.
<code>result</code>	Array of 16-bit integers holding the result on exit.

NOTE



If you manipulate arrays of unsigned chars for `msg`, `mod`, and `result`, you can typecast them into `(unsigned short *)` as long as you ensure that they are aligned on a 2-byte boundary. The `__align()` function can be used to enforce the alignment. For instance, you can define the following:

- `__align(2) unsigned char msg[255];`
- `__align(2) unsigned char mod[255];`
- `__align(2) unsigned char result[255];`

Return Values

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EACCES</code> , invalid buffer pointer. -1 with <code>errno</code> set to <code>EINVAL</code> , invalid parameter.

SHA1()

Performs an SHA-1 computation as described in FIPS PUB 180-2. It returns a 20-byte message digest.

Prototype

```
int SHA1(unsigned char * unused, unsigned char * input_buffer,
unsigned long nb, unsigned char * sha20);
```

Parameters

unused	This parameter is not used. It can be set to any value.
input_buffer	Pointer to the input buffer holding the message to process.
nb	Number of bytes in the buffer.
sha20	Pointer to the 20-byte buffer where the message digest is transferred.

Return Values

Success: 0, executed
Failure: -1 with errno set to EACCES, invalid buffer pointer.

SHA256()

Performs a SHA-256 computation and returns a 32-byte message digest.

Prototype

```
int SHA256 (unsigned char *unused, unsigned char *input_buffer,
            unsigned long nb, unsigned char *sha32);
```

Parameters

unused	This parameter is not used. It can be set to any value.
input_buffer	Pointer to the input buffer holding the message to process.
nb	Number of bytes in the buffer.
sha32	Pointer to the 32-byte buffer where the message digest is transferred.

Return Values

Success: 0, executed.
Failure: -1 with errno set to EACCES, invalid buffer pointer.

VeriShield Security Script Functions

The Verix eVo IPP supports the standard DUKPT and Master/Session management schemes. These schemes should meet the needs of the customers, and because they are hard coded into the Verix eVo unit, no customization of the security module is required.

For customers who need more flexibility, the VeriShield Security Script feature provides support for different:

- Key management schemes
- PIN block formats such as PVV, CVV, and IBM3624
- Encryption algorithms such as 3DES, AES, RSA

The security device (`/dev/crypto`) must be opened prior to using the VSS functions (including the PIN entry and key loading functions). All functions, data structures, and symbols are defined in the `<svc_sec.h>` header file.

All information is written in a script file (ASCII) using a `.vss` extension. This script is processed by the VeriShield Security Scripts (VSS) Converter and is converted into a downloadable file (`*.vso`) for the Verix eVo. The download is protected by the VeriShield File Authentication (FA) module. Therefore, the VeriShield Security Script file must be downloaded along with its signature file generated with the VeriShield File Signature tool.

NOTE



The VSS Converter (PN S6743-VSSCON40) is a tool required for developing VSS scripts.

The certificate used for FA must have VeriShield Security Scripts enabled for the GID in which the script is loaded. VeriShield Security Scripts have a special flag in the certificate for each GID. So, if a sponsor certificate is loaded and a secure script is to be loaded, it can only go into a GID permitting scripts.

Up to eight VeriShield Security Scripts can coexist in the Verix eVo-based unit at the same time. Each script defines its independent key space and can be loaded using the generic key loading security scripts.

The functions used to access the VeriShield security scripts are:

- `pcPS_GetVSSVersion()`
- `iPS_ExecuteScript()`
- `iPS_InstallScript()`
- `iPS_UninstallScript()`

See [VeriShield Security Scripts](#) for more information.

The functions used to manage PIN entry are:

- `iPS_CancelPIN()`
- `iPS_GetPINResponse()`
- `iPS_RequestPINEntry()`
- `iPS_SelectPINAlgo()`
- `iPS_SetPINParameter()`
- `iPS_SetPINBypassKey()`

The Key Loading Functions are:

- iPS_DeleteKeys()
- iPS_LoadSysClearKey()
- get_rkl_krd_cert()
- iPS_LoadSysEncKey()
- iPS_LoadMasterClearKey()
- iPS_LoadMasterEncKey()
- iPS_CheckMasterKey()

pcPS_GetVSSVersion()

Retrieves the VSS Version string.

Prototype `char* pcPS_GetVSSVersion (void);`

Return Values Pointer to a 10-byte (including the null) ASCII null terminated version string.

iPS_GetScriptStatus()

Checks if a VeriShield security script file is installed in the Verix eVo terminal and, if so, returns the name of the script.

Prototype `int iPS_GetScriptStatus(unsigned char ucScriptNumber,
unsigned char *pucINName);`

Parameters

<code>ucScriptNumber</code>	Script number. Range [0..7]
<code>pucINName</code>	Pointer to the application buffer where the 8-character name of the VeriShield security script (the string defined using the 'SCRIPT' keyword in the script file) is transferred.

Return Values

Success: 0, executed.

Failure: EBADF: Caller does not own crypto device.
 EACCES: Invalid buffer pointer.
 E_VS_SCRIPT_NOT_LOADED: Script is not loaded or is not accessible from the current GID.
 E_VS_SYSTEM_ERROR: General system error.

iPS_InstallScript()

Installs a VeriShield security script file in the unit. The script file name must have a .VSO extension and must be authenticated. The function performs several verifications on the script file during the install process, such as the compatibility between the version of the tool that generated the file and the version of the internal script interpreter.

Prototype `int iPS_InstallScript (char * pucINName);`

Parameters

<code>pucINName</code>	Pointer to the null-terminated filename.
------------------------	--

Return Values

Success: 0, executed.

Failure: EBADF, caller does not own crypto device.

 EACCES, invalid buffer pointer.

 E_VS_LOADING, file is not loaded, not authenticated, not accessible from the current application group, is not a VeriShield script file, or is not compatible with the script interpreter engine.

 E_VS_SYSTEM_ERROR, general system error.

iPS_ExecuteScript()

Starts the execution of a given macro from a given 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 execute.
usINDataSize	Size of the input data (in bytes).
pucINData	Pointer to the buffer containing the input data
usMaximumOUTDataSize	Maximum size of the output data. This is typically the size of the output buffer.
pusOUTDataSize	Pointer to the number of bytes returned by the macro in the output buffer.
pucOUTData	Pointer to the output buffer. The number of bytes returned in the output buffer is specified by pusOUTDataSize. If the macro 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

Success: 0, executed.

Failure: EBADF, caller does not own crypto device.

EACCES, invalid buffer pointer.

E_VS_SCRIPT_NOT_LOADED, script is not loaded, not authenticated, or not accessible from the current GID.

E_VS_MACRO_NOT_EXIST, macro does not exist in this script.

E_VS_BAD_LENGTH, usINDataSize is less than of the value expected by the macro or usOUTDataSize is less than the number of bytes the macro is attempting to return.

E_VS_BAD_CHAINING, bad sequence of macro (see Chaining Mechanism).

E_VS_SYSTEM_ERROR, general system error.

> 0 and < 256, macro execution error. The returned value is the value of the opcode that caused the execution error.

iPS_UninstallScript()

Uninstalls the specified VeriShield security script from the unit. The associated keys are deleted. The script file remains in the file system and can be reinstalled later.

Prototype `int iPS_UninstallScript(unsigned char ucScriptNumber);`

Parameters

<code>ucScriptNumber</code>	Script number. Range [0..7]
-----------------------------	-----------------------------

Return Values

Success: 0, executed

Failure: EBADF, caller does not own crypto device.

`E_VS_SCRIPT_NOT_LOADED`, this script is not installed, not authenticated, or is not accessible from the current application group.

`E_VS_SYSTEM_ERROR`, general system error.

VSS PIN Entry Functions

This section discusses the PIN entry functions.

NOTE



The PIN entry functions described in this section are only to be used with VeriShield Secure Scripts.

When the security device (`/dev/crypto`) is opened, the following parameters are set as defaults:

The following can be changed with function `iPS_SetPINParameter()`:

```
ucMin           = 4
ucMax           = 12
ucEchoChar      = '*'
ucDefChar       = ' '
ucOption        = 0x00
```

The following can be changed with function `iPS_SetPINBypassKey()`:

```
ucPinBypassKey  = 0x0D {ENTER}
```

The VSS PIN entry Functions should be called in the following order:

- `iPS_SelectPINAlgo()`
- `iPS_SetPINParameter()` - Optional, not needed if defaults are used.
- `iPS_SetPINBypassKey()` - Optional, not needed if default is used.
- `iPS_RequestPINEntry()`
- `iPS_GetPINResponse()`

iPS_CancelPIN()

Cancels the PIN processing.

Prototype `int iPS_CancelPIN(void);`

Return Values

Success: 0, executed.

Failure: EBADF: Caller does not own crypto device.



 E_KM_SYSTEM_ERROR: General system error.





iPS_GetPINResponse()

Checks the status of the PIN session. It will typically be 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 time-out as required in certain countries. The function returns the number of PIN digits entered and the last non-numeric key pressed.

Prototype `int iPS_GetPINResponse(int * piStatus, PINRESULT * pOUTData);`

Parameters

<code>piStatus</code>	<p>OK(0x00): Done. The PIN is stored internally and is ready to be processed by a script command.</p> <ul style="list-style-type: none"> • 0x01: Unit is idle. • 0x02: Collecting PIN. • 0x05: Aborted by user (the <CANCEL> key () was pressed). • 0x06: No PIN entered (only if this option is turned on). • 0x0A: Aborted by user. The <CLEAR> key () was pressed with no PIN digit in the buffer (the user had not entered any PIN digit, or had already cleared out all PIN digits). This value can be obtained only if <code>ucOption.bit4</code> has been set using the iPS_SetPINParameter() function.
<code>pOUTData</code>	<p>Pointer to the <code>PINRESULT</code> context structure giving the parameters used with the iPS_GetPINResponse() function. This struct is defined in the <code>svc_sec.h</code> file.</p> <p>This structure will return different information depending on the status of the PIN session. If <code>*piStatus</code> is equal to:</p> <ul style="list-style-type: none"> • OK(0x00): done. <ul style="list-style-type: none"> - <code>pOUTData->nbPinDigits</code> Number of PIN digits entered (PIN length). - <code>pOUTData->encPinBlock</code> 8-byte buffer contains no relevant information.

- 0x01: PIN pad is idle
 - pOUTData contains no relevant information.
- 0x02: Collecting PIN.
 - pOUTData->nbPinDigits: Number of PIN digits entered so far.
 - pOUTData->encPinBlock: The first byte of the buffer contains the value of the last non-numeric keypress. Values can be:
 - 0x00: Last key pressed was a numeric key (PIN digit).
 - 0x0D: Last key pressed was the <ENTER> () key.
 - 0x08: Last key pressed was the <CLEAR> () key.
- 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 keypress. Values can be:
 - 0x1B: <CANCEL> () key
 - (0x08: <CLEAR> () key
 - 0xFE: Timeout expired (only if ucOption.bit7 is turned on).
- 0x06: No PIN entered (Only if this ucOption.bit1 is turned on)
 - pOUTData contains no relevant information.

Return Values

Success: 0, executed.

Failure: EBADF: Caller does not own crypto device.

EACCESS: Invalid buffer pointer.

E_KM_SYSTEM_ERROR: General system error.

iPS_RequestPINEntry()

Initiates the PIN collection. Once the PIN entry is complete, the PIN is processed according to the algorithm specified by the previous [iPS_SelectPINAlgo\(\)](#) function. The PIN is then placed in a buffer and made available to the VeriShield Security Scripts. This function is non-blocking but allows the PIN pad to perform other tasks while the customer is entering the PIN.

Prototype

```
int iPS_RequestPINEntry(unsigned char ucPANDataSize,  
    unsigned char *pucINPANData);
```

Parameters

ucPANDataSize	This parameter is ignored and is retained only for <i>compatibility</i> .
pucINPANData	This parameter is ignored and is retained only for <i>compatibility</i> .

Return Values

Success: 0, executed.

Failure:

- EBADF: Caller does not own crypto device.
- EACCESS: Invalid buffer pointer.
- E_KM_NO_ALGO_SELECTED: A PIN algorithm must be selected first.
- E_KM_BAD_SEQUENCE: A PIN session is already started.
- E_KM_SYSTEM_ERROR: General system error.
- E_KM_ACCESS_DENIED: General system error.

Notes on PIN Exhaustion Protection

The device meets the logical security requirements of the PED specification, Paragraph B9. This paragraph requires that the device limit PIN encryption commands to an average rate of no more than one every 30 seconds.

The implementation is based on the “leaky bucket” algorithm that allows a “burst” of PIN encryptions that have less than 30 seconds between them. Every 33 seconds, Verix eVo adds one token until the maximum of 10 is reached. The content of the bucket is retained over the terminal’s power cycling. Every time a PIN session is requested a token is removed from the bucket. If there is no token available in the bucket the `iPS_RequestPINEntry()` function returns `E_KM_ACCESS_DENIED` error. The application displays an appropriate message and tries again few seconds later.

iPS_SelectPINAlgo()

Selects the PIN algorithm used during the next PIN session. The PIN algorithm cannot be changed during a PIN session.

In the Verix eVo environment, the only supported mode is 0Bh for use with VeriShield security scripts. In this mode, the PIN is saved internally and is retrieved by a security script command for post-processing.

Prototype

```
int iPS_SelectPINAlgo(unsigned char ucPinFormat);
```

Parameters

ucPinFormat • 0Bh = Store the PIN internally for post-processing by a VSS command.

Return Values

Success: 0, executed.

Failure: EBADF, caller does not own crypto keys.

 EACCESS, invalid buffer pointer.

 E_KM_OUT_OF_RANGE, ucPinFormat is out of range.

 E_KM_BAD_SEQUENCE, PIN algorithm cannot be changed during a PIN session.

 E_KM_SYSTEM_ERROR, general system error.

iPS_SetPINBypassKey()

Replaces the default value (ENTER) with whatever is specified. When the new key is pressed without any PIN digits having been entered, it behaves exactly like the null PIN key of the past (ENTER)—PIN processing state changes to 0x06 (PIN_NULL_PIN) and no error beep is generated.

Prototype `int iPS_SetPINBypassKey(unsigned char ucPinBypassKey);`

Parameters

<code>ucPinBypassKey</code>	Any value is accepted.
-----------------------------	------------------------

NOTE



The bypass key must not be a digit key. Error checking is done when the PIN entry routine is started, thus, [iPS_RequestPINEntry\(\)](#) can return a value of EINVALID, even though `iPS_SetPINBypassKey()` did not report an error.

Return Values

Success:	0, executed.
Failure:	EBADF, caller does not own crypto device.



iPS_SetPINParameter()

Configures several parameters for the upcoming PIN session.

Prototype

```
int iPS_SetPINParameter(PINPARAMETER * psKeypadSetup);
```

Parameters

psKeypadSetup	Pointer to the PINPARAMETER context structure giving the parameters used with the iPS_SetPINParameter() function. This struct is defined in the <code>svc_sec.h</code> file.
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	Character echoing PIN digit on the PIN pad display.
ucDefChar	Default field fill character. This field should be set to 0x20 (ASCII space character) when no specific fill character is needed. Setting ucDefChar to 0 may have undesired side effects.
ucOption.bit0	1 turns Auto Enter feature on. Ends PIN entry when maximum number of allowable PIN digits are entered.
ucOption.bit1	1 accepts No PIN entry (pressing ENTER before any digit).
ucOption.bit2	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 PIN digits). When the PIN session is cancelled this way, the *piStatus value returned by the iPS_GetPINResponse() function is 0x0A.
ucOption.bit5	May be 0 or 1. The OS modifies this bit as needed. The value set by the application is ignored.
ucOption.bit6	Modifies the behavior of iPS_GetPINResponse() for last key pressed. <ul style="list-style-type: none"> - 0x00 for no key press or digit since last poll - 0xF9 for too many digits - Key code for key press See iPS_GetPINResponse() for more details.
ucOption.bit7	Modifies the behavior of iPS_GetPINResponse() for the last key pressed, if 5 minute PIN entry timeout expired: <ul style="list-style-type: none"> - 0 = 0x1B <CANCEL> - 1 = 0xFE <timeout expired>

Return Values

Success: 0, executed.

Failure: EBADF, caller does not own crypto device.

EACCESS, invalid buffer pointer.

E_KM_OUT_OF_RANGE, at least one of the parameters is out of range.

E_KM_SYSTEM_ERROR, general system error.

Key Loading Functions

The functions described in this section load security script keys if the script allows their use.

The `VSS_KLK` (VeriShield Security Script key loading key) 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`, it must be loaded in the clear the first time. It must also be loaded before all other keys, otherwise, other keys in the unit will be erased. This must be done in a secured 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 an encrypted form, therefore, knowledge of `VSS_KLK` is required.

Each script defines its own set of keys and if they can be loaded with the generic key loading functions. Some scripts may disallow their use and may implement custom macro commands for key loading.

iPS_CheckMasterKey()

Indicates if a key is present in the specified location. The key verification code (KVC) argument is irrelevant in the VeriX eVo environment because this function is used only for security script keys. The key can be part of a double- or triple-length DES key. For security reasons the KVC portion of the key cannot be returned.

Prototype

```
int iPS_CheckMasterKey
(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *
pucINKVC);
```

Parameters

ucKeySetID	Key set identifier. <ul style="list-style-type: none">• 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	Key identifier. This is the master key number / Key index in the selected set.
pucINKVC	Not used

Return Values

Success:	0, executed
Failure:	EBADF, caller does not own crypto device. EACCES, invalid buffer pointer. E_KM_NO_KEY_LOADED, VSS_KLK is absent. No encrypted loading possible. E_KM_KEY_INTEGRITY_ERROR, the key is corrupt. E_KM_OUT_OF_RANGE, ucKeySetID or ucKeyID is out of range or script is not loaded. E_KM_SYSTEM_ERROR, general system error.

iPS_DeleteKeys()

Deletes the specified set of keys.

Prototype `int iPS_DeleteKeys (unsigned long ulKeyType);`

Parameters

`ulKeyType` Indicates which sets of keys are to be erased. Each bit corresponds to a set of keys, meaning that several sets can be erased in one function call.

- `DEL_SYSTEM`: System key (`VSS_KLK`)
- `DEL_VSS0`: Keys associated to VSS loaded in slot #0
- `DEL_VSS1`: Keys associated to VSS loaded in slot #1
- `DEL_VSS2`: Keys associated to VSS loaded in slot #2
- `DEL_VSS3`: Keys associated to VSS loaded in slot #3
- `DEL_VSS4`: Keys associated to VSS loaded in slot #4
- `DEL_VSS5`: Keys associated to VSS loaded in slot #5
- `DEL_VSS6`: Keys associated to VSS loaded in slot #6
- `DEL_VSS7`: Keys associated to VSS loaded in slot #7
- `DEL_ALL`: Delete all keys in the unit.

For example, `iPS_DeleteKeys(DEL_VSS2 | DEL_VSS3)` deletes only keys belonging to the Security Scripts loaded in slot #2 and #3.

Return Values

Success: 0, executed

Failure: `EBADF`, caller does not own crypto device.

`E_KM_SYSTEM_ERROR`, general system error.

iPS_LoadMasterClearKey()

Loads the security script's master keys. The values are sent in the clear, but must all be 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 in encrypted form. This function loads the keys defined by VSS if the option has not been disabled in the script.

NOTE



This function should be used exclusively in a secured environment.

Prototype

```
int iPS_LoadMasterClearKey
(unsigned char ucKeySetID, unsigned char ucKeyID,
unsigned char *pucINKeyValue);
```

Parameters

ucKeySetID	Key set identifier. <ul style="list-style-type: none"> • 00: Key set defined in VSS #0 • 01: Key set defined in VSS #1 • ... • 07: Key set defined in VSS #7
ucKeyID	Key identifier. This is the master key number/key index in the selected set.
pucINKeyValue	Pointer to the 8-byte buffer containing the cleartext value master key.

Return Values

Success:	0, executed
Failure:	EBADF, caller does not own crypto device. EACCES, invalid buffer pointer. E_KM_OUT_OF_RANGE, ucKeySetID or ucKeyID is out of range or script is not loaded. E_KM_FEATURE_DISABLED, key loading support disabled by a script. E_KM_SYSTEM_ERROR, general system error.

iPS_LoadMasterEncKey()

Loads the security script's master keys without deleting the keys already loaded. The new values must be encrypted under the current value of VSS_KLK. This function loads the keys defined by VSS if the option has not been disabled in the script. 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

ucKeySetID	Key set identifier. <ul style="list-style-type: none">• 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	Key identifier. This is the master key number/key index in the selected set.
pucINKeyValue	Pointer to the 8-byte buffer containing the encrypted value master key.

Return Values

Success:	0, executed.
Failure:	EBADF, caller does not own crypto device. EACCES, invalid buffer pointer. E_KM_NO_KEY_LOADED, VSS_KLK is absent. No encrypted loading possible. E_KM_KEY_INTEGRITY_ERROR, VSS_KLK is corrupt. E_KM_OUT_OF_RANGE, ucKeySetID or ucKeyID is out of range or script is not loaded. E_KM_FEATURE_DISABLED, key loading support disabled by a script. E_KM_SYSTEM_ERROR, general system error.

iPS_LoadSysClearKey()

Loads the VSS_KLK (system keys). The values are presented in the clear. Before writing the new value of the key, all other keys in the terminal are erased.

NOTE

This function should be used exclusively in a secure environment.

Prototype

```
int iPS_LoadSysClearKey
(unsigned char ucKeyID, unsigned char *pucINKeyValue);
```

Parameters

ucKeyID	Key identifier. 0x00 = VSS_KLK (16 bytes)
pucINKeyValue	16-byte buffer containing the clear-text key

Return Values

Success:	0, executed.
Failure:	EBADF, caller does not own crypto device. EACCES, invalid buffer pointer. E_KM_SYSTEM_ERROR, general system failure.

iPS_LoadSysEncKey()

Loads the system keys. The new values must be encrypted under the current value of VSS_KLK. Contrary to clear-text 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	Key identifier. <ul style="list-style-type: none">0x00 = VSS_KLK (16 bytes)
pucINKeyValue	16-byte buffer containing the encrypted keys.

Return Values

Success:	0, executed.
Failure:	EBADF, caller does not own crypto device. EACCES, invalid buffer pointer. E_KM_NO_KEY_LOADED, VSS_KLK is absent. No encrypted loading possible. E_KM_KEY_INTEGRITY_ERROR, VSS_KLK is corrupt. E_KM_SYSTEM_ERROR, general system error.

get_rkl_krd_cert()

Retrieves the Key RecFailureng Device certificate and writes the data to filename.

Prototype `int get_rkl_krd_cert(char *filename);`

Parameters

filename	Filename that will retrieve the certificate data.
----------	---

Return Values

Success:	0
Failure:-1	EEXIST - File already exists. ENOENT - Key Receiving Device certificate not found. EINVAL - Unable to open filename.



Verix Terminal Manager

The VeriFone Terminal Manager (VTM) is Trident's version of Predator's System Mode. The same functions and features of System mode can be found in VTM.

VTM has a flatter menu system compared to System Mode, with most features accessible on top menu.

When to Use VTM

Use the VTM functions to perform different subsets of related tasks:

- **Application programmers** configure a development terminal, download development versions of the Trident application program, then test and debug the application until it is validated and ready to be downloaded to other terminals.
- **Deployers of Trident terminals to end-user sites** perform the specific tasks required to deploy a new Trident terminal on-site, including configuring the terminal, downloading application software, and testing the terminal prior to deployment.
- **Terminal administrators or site managers** change passwords, perform routine tests and terminal maintenance, and configure terminals for remote diagnostics and downloads by telephone.

To perform the subset of tasks that corresponds to a job, select the appropriate VTM menus and execute the corresponding procedures.

Local and Remote Operations

The VTM operations available on a Trident terminal can be divided into the following two categories or types:

- **Local operations** address a stand-alone terminal and do not require communication or data transfers between the terminal and another terminal or computer. Perform local VTM operations to configure, test, and display information about the terminal.
- **Remote operations** require communication between the terminal and a host computer (or another terminal) over a telephone line or a cable connection. Perform remote VTM operations to download application software to the terminal, upload software from one terminal to another, or perform diagnostics over a telephone line.

This chapter contains descriptions on how to perform local VTM operations.

Verifying Terminal Status

The Trident terminal you are using may or may not have an application program running on it. After you have set up the terminal and the terminal is turned on, use the following guidelines to verify terminal status regarding software and current operating mode:

- If no application program is loaded into terminal RAM or flash, the message **DOWNLOAD NEEDED** appears on the display screen.

NOTE



Enter VTM by simultaneously pressing 7 and ENTER keys to perform the necessary download.

- If an application program is loaded and *GO is set in the configuration file in Group 1 to the application's name into the terminal RAM or flash, an application-specific prompt appears. The application is running and the terminal is in normal mode. If all installation steps are complete, the terminal can process transactions.

Entering Verix Terminal Manager

To prevent unauthorized use of the VTM menus, the terminal OS requires a system password each time you enter VTM. To access the VTM password entry screen, simultaneously press 7 and ENTER keys. The default factory-set system password is "166831" for terminals without the ALPHA key. If an ALPHA key is present the default password is "Z66831" or "1 ALPHA ALPHA 66831". Use the following key sequence to enter this password:

1 6 6 8 3 1 ENTER.

"166831" is the default password for models that do not have an ALPHA key. Default factory-set system password for other models is "Z66861".

Passwords

The VTM password must be between 5 and 10 numeric characters long. Otherwise, VTM will present a series of screens requiring the user to choose a new 5 – 10 character password. After entering the correct password, the terminal enters the VTM and displays the first VTM main menu. You can now cycle through all VTM main menus.

CAUTION



If you change a password but forgot it later on, no password recovery method is available. Without the password, you are unable to access VTM operations and may be prevented from requesting a download, performing remote diagnostics, or changing any of the information already stored in memory. The terminal can, however, continue to process transactions in normal mode.

If you forget or lose the system password of your terminal, please contact your local VeriFone representative for assistance.

System Password

When you key in the system password to enter the terminal manager, an asterisk (*) appears for each character you type. This prevents your password from being seen by unauthorized persons.

NOTE



Some application program downloads automatically reset the system password. If your system password no longer works, check if a download has changed your password.

File Group Passwords

From manufacture, each file group uses the default password “166831” (or “Z66831” for other models), which is entered as follows:

1 6 6 8 3 1 ENTER or 1 ALPHA ALPHA 6 6 8 3 1

VTM Menu

After supplying the correct password, the main VTM screen appears below:

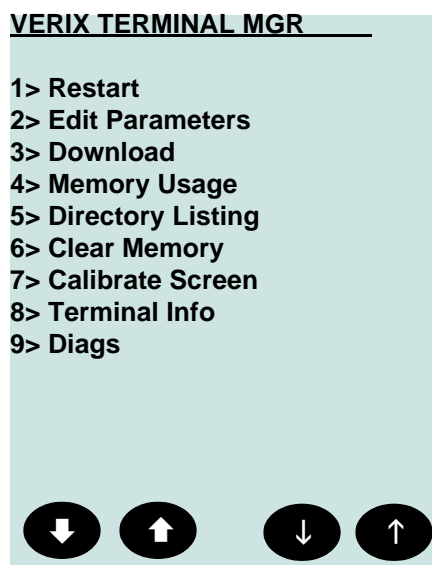


Figure 11 VTM display on VX 680

NOTE



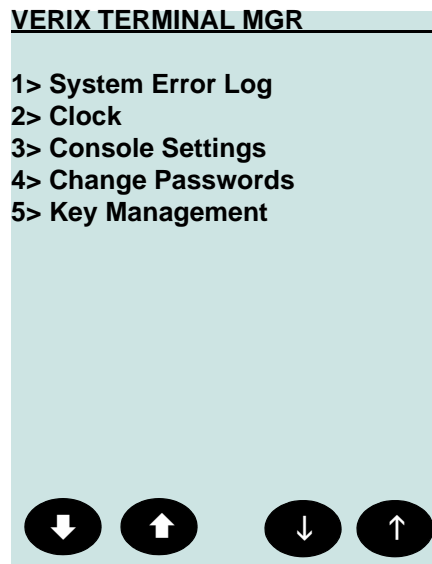
Edit Parameters option in the VTM menu may not display properly when the NAND file system is full.

NOTE



VX 680 VTM menus are shown. The VX 520 menus are arranged differently but all functions/features are available.

When the leftmost downward pointing arrow is touched, the following screen appears:



NOTE



The VX 680 VTM menus are selected using the touchscreen icons as no screen or function keys are available. The VX 520 uses the screen and function keys to navigate VTM. Some terminals have a navigation key, a 4-way rocker switch similar to some cell phones. When present, VTM can be navigated using this key.

VTM runs on all currently planned screens, displaying in the space available. For example, VX 680 displays 9 choices per page, where the VX 520 displays 3 per page. VX805 uses 6 line menu displays.

There are two methods for selecting a prompt/function:

- On the keypad, the user can press the number that corresponds to that displayed next to the prompt.

Example: 4> Memory Usage

The numbers vary on different platforms.

- The user can move an inverted video line up/down the screen and then press the ENTER key to choose the prompt/function.

To move between screens, choose arrows at the bottom of the screen. On the VX 520 the arrows are displayed at the bottom of the screen. A horizontal key is used to choose an arrow. On other models (touch screen), the buttons are displayed at the bottom of the screen and is touched to activate.

The left large down arrow increments the page, the left large up arrow decrements the page. The right small down arrow moves the invert line down one line and the left small up arrow moves the invert line up one line.

Text is in upper and lower case. The larger screens of the VX 680, and VX 820 use 8x16 font file by default. It is loaded by the OS on startup. Input on

terminals without an alpha key supports multi-key press mode (also called cell phone mode). It is advantageous to make passwords numeric and non-repeating. To enter the current default password “166831” (“Z66831” for other models), the user must wait between each key press. Terminals with the alpha key use the traditional alpha key selection of alpha characters and selected characters such as . , ? or /. See [alpha_shift\(\)](#) for more information.

Features

The Download screen has the same look and feel as the Predator terminals. The prompts are chosen using the same methods as described above. Edit offers New, Find, Edit, and Clear options on the main EDIT screen. Memory Usage displays the same information as Predator terminals.

MEMORY USAGE		
Drive I: Files		1
Inuse		2 KB
Drive F: Files		0
Inuse		0
RAM Avail	29412	KB
Flash Avail	125776	KB

Figure 12 Memory Usage Display for VX 680

The I: and F: drive numbers are the same as in Predator, except the Inuse number is always a multiple of 2 KB. Files are written on 2 KB boundaries. Each file’s size is reported as the next highest 2 KB number of bytes to the actual size of the file (a file of 2KB+x bytes where $0 < x < 2KB$ is reported as 4 KB in size).

RAM Avail represents the KB available to the user for heap and stack. Flash Avail represents the KB available for application download.

NOTE



I: and F: drives are both Flash file system drives. There is no RAM file system. RAM is used for execution of binaries, stack, and heap.

RAM and Flash directories look the same. PRINT needs to be a touch prompt on all but the VX 520. Terminal Info displays the same information as Predator terminals.

NOTE



Refer to your terminal’s specific Reference Guide for a detailed information on VTM menus and sub-menus.



VeriShield Security Scripts

This chapter discusses the VeriShield Security Script (VSS) concept that allows creation and customization of security modules to support different key management schemes such as 3DES for master and session keys, offline PIN verifications, APACS40, and so on.

Verishield Security Script Implementation

This chapter focuses on the VeriShield security script implementation in the Verix eVo terminals. In its default configuration, the Verix eVo-based unit supports two key management schemes:

- DUKPT
- Master/Session

Those two schemes meet the needs of most of the customers and since they are part of the Verix eVo OS, no customization of the security module is required.

For flexibility, the VeriShield security script mechanism provides support for:

- Additional key management schemes
- Different PIN block formats such as PVV, CCV, IBM 3624
- Additional key space
- Different encryption algorithms such as, 3DES, AES, and RSA

See [VeriShield Security Script Functions](#) or refer *VeriShield Security Scripts*, 21883 for more information.



IPP Key Loading

This appendix describes IPP key loading and memory area of the IPP.

The role of system mode in key insertion is limited to providing a pass-through connection from COM1 (RS-232 port) to COM5 (IPP port) for use by an external key loading tool, such as SecureKit. Except for the packet buffering described below, it does no interpretation of the data, other than to indicate the number of bytes transmitted and received (user feedback). It has no knowledge of key loading protocols and does not do any setup or monitoring. COM1 settings are independent of COM5 settings.

Since VTM runs at the application level, IPP driver command filtering applies. Otherwise, there are no restrictions on the commands that can be sent.

Data Passthrough

The IPP driver requires that complete commands be sent by a single call to `write()`. Therefore, incoming data from COM1 is buffered until a full command packet is assembled before written to COM5.

Normal packet structure is:

STX	Data	ETX	LRC
-----	------	-----	-----

or

SI	Data	SO	LRC
----	------	----	-----

The passthrough code treats any <STX> or <SI> character as the start of a packet and the next <ETX> or <SO> character as its penultimate byte. Any characters received between the end of one packet and the start of the next are passed through unbuffered. These would normally be <ACK>, <NAK>, and <EOT> control characters. The parity bit is ignored for the purpose of recognizing special characters, but it is unchanged in the data sent to the IPP. VTM never deletes or alters any data.

There is no requirement to write whole packets in the other direction, so data read from COM5 is written to COM1 uninspected and unbuffered.

User Interface

When the user selects `IPP KEY LOAD` they are normally prompted to enter the group 1 password, using the standard prompt dialog.

By default, COM1 is set to 19200 baud, A8N1 format. These can be overridden by the `CONFIG.SYS` variable `*IPPMKI` (see [*IPPMKI—Internal PIN Pad Communications Parameters](#)). If used, set `*IPPMKI` to a string containing the desired baud rate or the following flags:

- E Even parity (A7E1)
- O Odd parity (A7O1)
- D Assert DTR
- R Assert RTS

NOTE

*IPPMKI refers only to settings used for the external COM1 (RS-232) port where the key loading system (usually a PC running SecureKIT) is physically connected. It does not affect the IPP itself, which is accessed by applications as /DEV/COM5.

E, O, D, and R also set Fmt_A7E1, Fmt_A7O1, Fmt_DTR, and Fmt_RTS, respectively.

The flags and rate can be intermixed in any order. Unrecognized characters are ignored. For example:

- *IPPMKI = 1200E sets the port to 1200 baud, even parity
- *IPPMKI = ER sets the port to 19200 baud (default), even parity, assert RTS
- *IPPMKI = R,9600,E sets the port to 9600 baud, even parity, assert RTS (commas are ignored)

*IPPMKI is intended to support key loading software with fixed communication requirements; the baud rate probably has no significant effect on performance given the small amount of data involved. Note that COM1 settings are independent of COM5 (IPP) settings.

When pass-through mode starts, the following screen displays:

```
INTERNAL PIN PAD
KEY LOADING MODE

BYTES SENT  2862
BYTES RCVD   894

Press 1 to END
```

Figure 13 Sample IPP Key Load Screen

The figure below shows a sample IPP Key Load screen.

```
INTERNAL PIN PAD
KEY LOADING MODE

BYTES SENT  0
BYTES RCVD  0

Press 1 to END
```

Figure 14 Sample IPP Key Load screen for VX 680 Terminal

The byte counts are updated to provide feedback to the user. Pressing CLEAR ends passthrough mode, closes COM5, and restarts the terminal.

If an error occurs, the messages, “** IPP ERROR *n* **” or “** COM1 ERROR *n* **” display on the bottom line. Pressing any key other than F4 or CLEAR erases the message and resumes data passthrough. Pressing F4 or CLEAR restarts the terminal.

Error Codes

Error messages for the IPP DIAG and IPP KEY LOAD screens are reported on the last line of the screen as “** *dev* ERROR *n* **,” where, *dev* is the device, either IPP or COM1, and *n* is the error code. Positive numbers for the error code map to values in `errno.h`. Negative numbers map to values generated by the VTM application, as follows:

- -1 Cannot open COM5 (errno set)
- -2 Cannot communicate with IPP
- -3 COM5 read/write error (errno set)
- -4 Time out waiting for start of packet
- -5 Time out waiting for next byte, <ACK>, and so on
- -6 Received too many <NAK>s
- -7 Received <EOT> before end of transaction

Master Key Protection

The secure memory area used for both MS and DUKPT keys is protected by CRC checksums. When the terminal powers on, the IPP checks its storage integrity using the stored CRC values. If the result is a mismatch, the entire memory area and all stored keys are erased.

Tampering with the terminal (for example, opening the case) also erases all stored keys.

PCI PED Enhancements

On Verix V, two timeouts are enforced during IPP keyloading. If no data is received from the host within 60 seconds, the program will terminate and return to VTM. If the operation is not completed within 15 minutes, the program will stop transferring data and notify the user; once the notification is acknowledged, the program will force a system restart.

A `system_integrity_check` has also been added to the VTM. This typically runs at startup at least once in every 24-hour period. By default it is scheduled to run in the early morning hours. Setting the `*SYSCHK=hhmm` allows the user to select a new time to run the system check.

The system integrity check validates all stored secret keys and, by default, checks all file systems and groups including all authenticated files. On startup, the system integrity check ignores any errors found and reported through the interactive file check to give the user up to 24 hours to correct the problem. Otherwise, if system errors are found, the system will hang displaying the message “CALL CUSTOMER SERVICE.”

NOTE



If errors are found during the periodic daily check, part of error handling is restarting the system so that the startup logic can notify the user.

The application loading process then verifies checksums for all code files, including associated library files. Files that fail the test will not be run. If “FATAL ERROR” occurs, the “CALL CUSTOMER SERVICE” message appears. User can still select the CANCEL key to continue, however, the `*GO` variable will be cleared so that no application can run.

Password Requirements

The terminal’s operating system conforms to the PCI PED requirements for initiating the IPP key loading operation.

The VTM IPP key loading process requires users to enter the VTM password followed by the GID 1 password. In compliance to the PCI PED, the requisite VTM and GID 1 passwords are at least five characters long each. This is to maintain a consistent user interface for password entry.

NOTE



Currently, the OS allocates up to 10 characters per password.

PIN entry is limited to a maximum of 120 entries per hour. The `REFILL_RATE`, which defines the token creation rate, is changed from one every 30 seconds to one every 33 seconds. The `MAX_TOKENS`, the most entries that can be accumulated, is changed from 127 to 10. These revised values allow up to 119 PIN entries on the first hour, and up to 109 PIN entries per hour thereafter.

Changing Passwords Manually

Change the VTM password or any GID password from the VTM Passwords Menu.

To change the password manually

- 1 Navigate to the VTM Password Menu.
- 2 Enter the new system password. The new password must be at least five characters long but not exceeding ten characters.
- 3 Press the ENTER key.

Passwords Shorter than Required

Attempting to enter a new password with less than five characters results in an error message accompanied by a beeping sound.

Figure 15 shows the error message on a four-line display.

```
Change Passwords      G1
                        |
                        |
Please Try Again
```

Figure 15 Error Message on a Four-Line Display

Figure 16 shows the error message on an eight-line display.

```
Change Passwords      G1
                        |
                        |
Please Try Again
```

Figure 16 Error Message on an Eight-Line Display

Figure 17 shows the error message on a 16-line display.

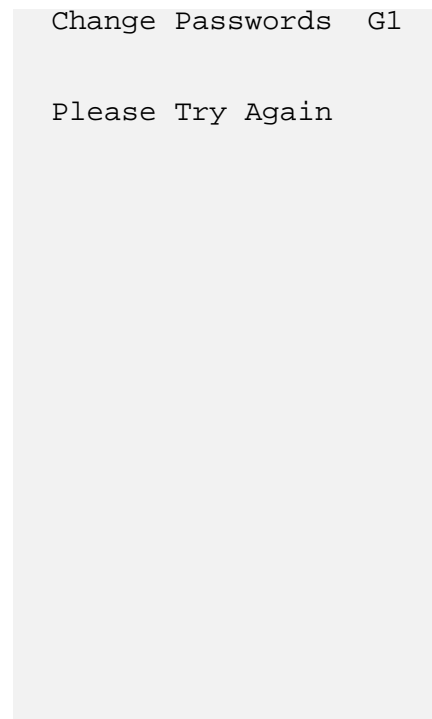


Figure 17 Error Message on a 16-Line Display

When this happens, press ENTER key to return to the Passwords Menu. Continue by reentering a valid password of five characters minimum length. Pressing the CANCEL key at this point lets you exit the Change Password screen but terminates the manual password change.

Passwords Longer than Required

When user attempts to enter a password exceeding ten characters, a beep sounds in each succeeding key presses after the tenth character. However, this has no critical effect on the password since the OS sets the first ten characters entered as the new password.

Download Password Change

A user can also set the VTM password or any GID password by downloading a Password Change parameter. This can be done from any download server — VeriCentre, DDL, or customer-developed custom server.

- If the downloaded password is at least five characters and no more than ten characters long, the OS accepts the new password, which must then be used for all future operations where password is required.
- If the downloaded password is more than ten characters long, the OS truncates the downloaded password to the first ten characters. The new truncated password is accepted and must then be used for all future operations where password is required.
- If the downloaded password is less than five characters long, the OS still accepts the new “short” password until the next time the user attempts to perform an operation where password is required. Before another operation

can be performed, the OS requires the user to change the short password to a valid password that complies with the required five-character minimum length. To change the password, follow the password entry logic described in the [Changing Passwords Manually](#).

In this case, the OS continues to operate normally for all activities which do not require a password, hence, any applications loaded in the terminal are unaffected and will run normally.

CAUTION

It is possible for a download server to change a terminal password to one which can not be entered on the terminal keyboard. In this case, the terminal could be rendered unusable, depending on which password is changed and the specific terminal configuration.

The CONFIG.SYS variables *SMPW and *PW should only be set using characters that are supported on the terminal keypad, otherwise, access to the VTM or individual GIDs are blocked. When this happens, the terminal must be sent to an authorized VFI Repair Center to clear the terminal's memory and reset the default password. If the unusable password is set during a partial download, any data still resident in the terminal will be lost when the terminal's memory is cleared.

Not all devices' keypads are the same. Caution should be exercised in creating VTM and GID passwords. The only valid characters for the passwords are those that can only be entered from the device keypad. The allowable characters are (in uppercase only):

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

. * , ' " - + # ! : ; @ = & / \ % \$ _

OS Upgrade

When upgrading from earlier Verix eVo operating system, where passwords as short as one character are allowed, to the PCI PED-compliant OS, the procedures enumerated in [Download Password Change](#) are similarly applied.

This means that the OS accepts the short password and does not enforce the PCI PED-compliant password until the user attempts to perform an operation requiring the use of password. Only then will the user be required to change the password to comply with the PCI PED standards before he can continue with the transaction.

Any application(s) running in the terminal are unaffected by the new OS.

NOTE

Passwords used in Trident terminals has to meet the described standards. The scenario discussed above applies only to Predator terminals.

Default Password

The OS sets a default password of Z66831 for VTM and for GID 1. The GID 2 to GID 15 passwords are empty by default. The established manufacturing process, which uses a script to set GID 2 to GID 15 passwords to Z66831, is maintained.

IPP Key Load

The user is required to enter the GID 1 password each time IPP KEY LOAD is selected. This standard is imposed even if the user previously entered the GID 1 password in the current VTM session.

The GID 1 password entry is also required when IPP KEY LOAD operation is restarted (that is, Key Load session is terminated for any reason such as session timeout or CANCEL key is accidentally pressed).



IPP Communications Packets

This appendix describes the required packet commands of the IPP for MS (Master Session) or DUKPT operations supported by the Verix eVo OS.

Advanced Programming in IPP

For programmers familiar with Verix eVo IPP6/IPP7 MS and DUKPT features, the Verix V IPP (VVIPP) has almost all of the same features. The differences are summarized in Table 59.

Table 59 Differences in Verix V IPP

IPP	IPP6	IPP7	VVIPP	IPP8	VVIPP8
Secure Message Mode	No	Yes	No	Yes	No
Spain SEMP/4B	Yes	Yes	No	Yes	No
Key tagging	Yes	No	No	No	No
DUKPT Engines	1	1	1	3	3

VVIPP supports IPP7 GISKE 3DES key features with one enhancement: All 10 master keys can be triple-length keys.

Minor Differences by Packet

<SI>0103<SO> PROM Checksum

The value of the checksum does not match IPP7 because Verix eVo OS does not use the same code.

<SI>0108<SO> IPP ROM Version Number

The return packet is

```
<SI>14IPP7 PREDvvv 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

There is no IPP UART, so setting the baud rate does nothing. However, the baud rate is stored in non-volatile memory so it can be returned 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 Verix eVo-based 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. This speeds 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

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.

Packets

The packet set is similar to that used for external PIN pads, such as the PINpad 1000, however, unlike previous IPPs, the Verix V 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 Verix V IPP the COM5 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

Verix eVo IPP does 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.

Table 60 lists packets used in both MS and DUKPT sessions.

Table 60 Common Packets

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

Table 60 Common Packets (continued)

Packet	Description
12	Dummy packet
13	Select baud rate
14	Response to Packet 01
15	Set IPP key management mode
17	Set IPP7 key management mode
18	Check IPP7 key management mode
72	Cancel PIN session
M04	Read Permanent Unit Serial Number (IPP8 Emulation)
Z7	Cancel PIN session (Do not use. Only implemented for backwards compatibility).

Table 61 lists packets supported by IPP for MS.

Table 61 IPP Supported Packets 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

Table 62 lists packets supported by IPP for DUKPT.

Table 62 IPP Supported Packets 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

Table 62 IPP Supported Packets for DUKPT (continued)

Packet	Description
19	Select a DUKPT Engine (IPP8 Emulation)
25	Check the DUKPT Engine (IPP8 Emulation)

Packet Acknowledgement 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, that is, a packet includes with the wrong header, wrong trailer, wrong field separator, that has out of range indexing (for example, packet 02, master key address = 15), or with incorrect packet length, and so on.

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.

Figure 18 illustrates an MS encryption session.

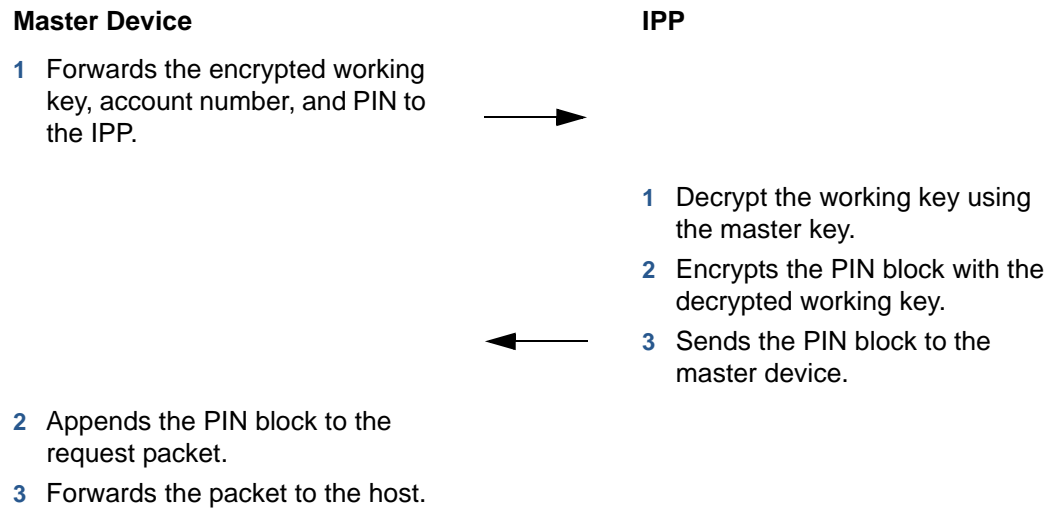


Figure 18 Master Session Encryption Example

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 key serial number (KSN) and a unique initial PIN Encryption Key (PEK). 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.

Figure 19 illustrates the DUKPT method of encryption.

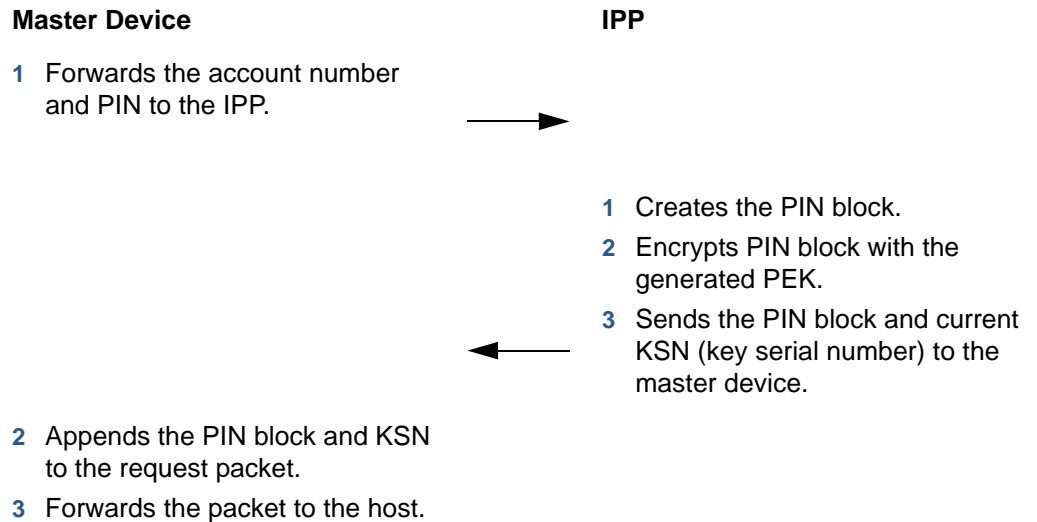


Figure 19 DUKPT Session Encryption Example

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.

PCI requirements limits the PIN entry session to five (5) minutes. It will be cancelled if this time is exceeded. The design is backwards compatible to have no effect on current applications. The expired timeout behaves like the CANCEL key was pressed. The following packets will cancel a PIN session:

- <STX>72<ETX>{LRC} This packet should be used to cancel the PIN session.
- <SI>72<SO>{LRC} **Do not use.** Only implemented for backwards compatibility.
- <SI>Z7<SO>{LRC} **Do not use.** Only implemented for backwards compatibility.

Any other packets sent during the PIN session will cause PIN entry errors. See [Packet 72: Cancel MAC Session](#) for packet details.

Restrict Speed of the PIN Encryption Operation

This restriction only applies to Master Session. DUKPT has no restriction. 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 maintained across power cycles.) 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 Verix eVo IPP. Verix eVo IPP is backward compatible with IPP6 and IPP5. Exceptions to this rule are noted.

GISKE Global Interoperable Secure Key Exchange (GISKE) is an industry standard key block format used 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 Key Loading Key (KLK) 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 more details on GISKE refer *GISKE Key Block Spec, VPN 22986*.

Key Management Switching The rules for key management switching (see [Packet 17: Set IPP Key Management Mode](#)) are shown in [Table 63](#).

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

Table 63 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	<ul style="list-style-type: none"> • 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 				

a. Spain and SM modes not supported in Verix eVo. 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 IPP 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

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 IPP 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 [Table 64](#) applies to 1DES master key, 3DES master key (GISKE), and KLK (GISKE). The original GISKE (ASCII-hex) key usage attribute value is saved in memory (2 bytes).

Table 64 Key Attributes

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 is 1DES 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.

Table 65 lists the key length register values for 1DES, 3DES, and three-key 3DES.

Table 65 Key Length Register Values

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 Verix V, all master key locations 0–9 can hold single-, double-, or triple-length DES keys. Verix eVo IPP can hold at most 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 block 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

Clear text 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.

Table 66 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.

Table 67 Packet 05 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 05
[vvv]	3AN	PIN pad version number
[ddddd]	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}

Table 68 Packet 05 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>05[vvv][dddddd][p][bb][nnnn]<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
	←	<SI>05[vvv][dddddd][p][bb][nnnn]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

Table 69 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}

Table 70 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

Table 70 Packet 06 Format (continued)

Data Element	Characteristic	Comments
[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}

Table 71 Packet 06 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>06<SO>{LRC}	➡	
	⬅	<ul style="list-style-type: none">• ACK if LRC• NAK if LRC incorrect• EOT after 3 NAKs
	⬅	
		<SI>06[vvv][dddddd][p][bb][nnnn]<SO>{LRC}
<ul style="list-style-type: none">• 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 [Table 72](#).

Table 72 **Packets 09 and 14 Communication Protocol**

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: <ul style="list-style-type: none"> • 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}
	←	

Table 72 **Packets 09 and 14 Communication Protocol** (continued)
















Master Device	Transmit Direction	IPP
ACK/NAK/EOT		
		EOT to terminate process.
02 RAM Test/Continuous		
<SI>0102<SO>{LRC}		IPP
		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}		IPP
		ACK/NAK/EOT
		<SI>14xx<SO>{LRC} where, xx is the one-byte PROM internal checksum. There are two checksums inside the IPP: <ul style="list-style-type: none"> • 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.

Table 72 **Packets 09 and 14 Communication Protocol** (continued)

Master Device	Transmit Direction	IPP
06 Serial Number Check		
<SI>0106<SO>{LRC}	→	IPP6 and earlier
	←	ACK/NAK/EOT
	←	<SI>14xxxxxxxxxxxxxxxx<SO>{LRC} where, xxxxxxxxxxxxxxxxxxxx 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}	→	IPP
	←	ACK/NAK/EOT
	←	<SI>09<SO>{LRC}
	→	ACK/NAK/EOT
	→	<SI>09<SUB>PROCESSING<SO>{LRC}
<SI>09<SUB>PROCESSING<SO>{LRC}	←	ACK/NAK/EOT
	←	<SI>09<SUB>PROCESSING<SO>{LRC}
	→	ACK/NAK/EOT
ACK/NAK/EOT	→	
	←	EOT to terminate process.

Table 72 **Packets 09 and 14 Communication Protocol** (continued)

Master Device	Transmit Direction	IPP
08 IPP PROM Version Number		
<SI>0108<SO>{LRC}	→	IPP
	←	ACK/NAK/EOT
	←	<SI>14IPPx vvvvxxx MM/YY<SO>{LRC} where: <ul style="list-style-type: none"> • 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}	→	IPP
	←	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.)

Table 72 **Packets 09 and 14 Communication Protocol** (continued)

Master Device	Transmit Direction	IPP
10 Clear IPP		
<SI>0110<SO>{LRC}	→	IPP
	←	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 1 second. Else, it assumes that the IPP is not connected.

Table 73 **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}

Table 74 **Packet 11 Communication Protocol**

Master Device	Transmit Direction	IPP
<SI>11<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • 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 1 second.

**Packet 13: Select
Baud Rate**

Verix V supports this packet but it has no effect. Verix eVo-based 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 memory. After a power cycling memory test or loss of backup battery power, the baud rate setting is reset to the default.

Table 75 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 <ul style="list-style-type: none"> • 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

Table 76 **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,
		<div> <div>x = baud rate code</div> <div>yyyyy = string for selected baud rate</div> </div> <ul style="list-style-type: none"> • 1 • 1200 (default) • 2 • 2400 • 3 • 4800 • 4 • 9600 • 5 • 19200
		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:

- VISA MASTER SESSION+DUKPT mode

VISA MASTER SESSION+DUKPT mode covers MS and DUKPT and MAC process of standard ANSI MAC. The Verix eVo IPP does not include SEMP/4B mode, and erases keys when this mode is selected.



In the Verix eVo 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}

Table 77 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 <ul style="list-style-type: none"> • "SPAIN" – Spain SEMP/4B mode (Not supported) • "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}

Table 78 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: <ul style="list-style-type: none"> • "SPAIN" – Spain SEMP/4B mode (Not supported) • "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>15VISA<SO>{LRC} (set MS / DUKPT mode)



In IPP6, the following packet 15 variation is included for compatibility purposes only, and does not result in the key information being erased.

Table 79 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

Table 80 Packet 15 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>15IPP2<SO>{LRC}	➔	
	➔	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect
	➔	<SI>15IPP2<SO>{LRC}
ACK/NAK	➔	
	➔	EOT to terminate process.

Packet 17: Set IPP Key Management Mode

This packet sets or clears a number of control switches in the key management options register for IPP key management configuration. IPP supports the following additional functions:

- 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 IPP is set to 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 IPP to erase all existing keys or non-volatile security values stored for secure messaging.

Incoming Packet Format:

<SI>17[KMM][PINER]<SO>{LRC}

Table 81 Packet 17 Format

Data Elements	Characteristics	Comments																								
<SI>	1H	Shift In, value: 0Fh																								
Packet Type	2AN	Value: 17																								
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 IPP is as follows: <div><div>Bit</div><table><tr><th>2</th><th>1</th><th>0</th><th>Description</th></tr><tr><td>0</td><td>0</td><td>0</td><td>1DES MS (default)</td></tr><tr><td>0</td><td>0</td><td>1</td><td>Mixed mode (1DES and 3DES GISKE)</td></tr><tr><td>0</td><td>1</td><td>0</td><td>3DES GISKE MS</td></tr><tr><td>0</td><td>1</td><td>1</td><td>Not supported</td></tr></table><div><div>Bit 3</div><div>Description</div><table><tr><td>0</td><td>1DES DUKPT (default)</td></tr><tr><td>1</td><td>3DES DUKPT</td></tr></table></div></div>	2	1	0	Description	0	0	0	1DES MS (default)	0	0	1	Mixed mode (1DES and 3DES GISKE)	0	1	0	3DES GISKE MS	0	1	1	Not supported	0	1DES DUKPT (default)	1	3DES DUKPT
2	1	0	Description																							
0	0	0	1DES MS (default)																							
0	0	1	Mixed mode (1DES and 3DES GISKE)																							
0	1	0	3DES GISKE MS																							
0	1	1	Not supported																							
0	1DES DUKPT (default)																									
1	3DES DUKPT																									

Table 81 **Packet 17 Format** (continued)

Data Elements	Characteristics	Comments
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

Table 81 **Packet 17 Format** (continued)

Data Elements	Characteristics	Comments																			
DUKPT Engine 1/2 Mode Flag [DEMF]	1AH	The one ASCII-Hex digit is used produce half of a control byte.																			
Note: This field was added for IPP8 emulation.		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																			
	<table><tr><td>Engine=</td><td><u> </u></td><td>1</td><td>2</td></tr><tr><td>DEMF = 0x30</td><td>0</td><td>1DES</td><td>1DES</td></tr><tr><td>0x31</td><td>1</td><td>3DES</td><td>1DES</td></tr><tr><td>0x32</td><td>2</td><td>1DES</td><td>3DES</td></tr><tr><td>0x33</td><td>3</td><td>3DES</td><td>3DES</td></tr></table>	Engine=	<u> </u>	1	2	DEMF = 0x30	0	1DES	1DES	0x31	1	3DES	1DES	0x32	2	1DES	3DES	0x33	3	3DES	3DES
Engine=	<u> </u>	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

Table 82 Packet 17 Set IPP Key Management Mode

Master Device	Transmit Direction	IPP
<SI>17[KMM][PINER]<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKS
	←	
		<SI>17[KMM][PINER]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

- 1 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.
- 2 When the IPP receives packet 17 to change KM modes (for example, to 3DES). 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.
- 3 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.
- 4 Switching from SM to IPP mode causes a factory reset. The IPP clears the contents of memory and communication to the IPP is reset to the default, 1200 baud, 7 data bits, even parity, and 1 stop bit (7E1).
- 5 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.

- 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}

Some valid IPP KMM are shown above. 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 IPP Key Management Mode

Checks the setting in the IPP key management options register.

Request Packet Format

<SI>18<SO>{LRC}

Table 83 **Packet 18 Format**

Data Elements	Characteristics	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	2AN	Value: 18

Table 83 **Packet 18 Format** (continued)

Data Elements	Characteristics	Comments																																																						
Key Management Mode [KMM]	2AH	<p>The two digits are concatenated big-endian, to produce a single control byte. The key management mode register (8 bits) in IPP is as follows:</p> <table><thead><tr><th colspan="3">Bit</th><th></th></tr><tr><th>2</th><th>1</th><th>0</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td><td>Old single DE</td></tr><tr><td>0</td><td>0</td><td>1</td><td>Mixed mode (1DES and 3DES GISKE).</td></tr><tr><td>0</td><td>1</td><td>0</td><td>3DES GISKE MS</td></tr><tr><td>0</td><td>1</td><td>1</td><td>Not supported</td></tr></tbody></table> <table><thead><tr><th>Bit 3</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>1DES DUKPT</td></tr><tr><td>1</td><td>3DES DUKPT</td></tr></tbody></table> <table><thead><tr><th>Bit 4</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>Zero key support off.</td></tr><tr><td>1</td><td>Zero key support on.</td></tr></tbody></table> <table><thead><tr><th>Bit 5</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>Zero GISKE session key support off.</td></tr><tr><td>1</td><td>Zero GISKE session key support on.</td></tr></tbody></table> <table><thead><tr><th>Bit 6</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>At least one MS key or KLK key has been loaded.</td></tr><tr><td>1</td><td>All MS master keys and the KLK are clear (no keys loaded).</td></tr></tbody></table> <table><thead><tr><th>Bit 7</th><th>Description</th></tr></thead><tbody><tr><td>0</td><td>MAC empty working key support off.</td></tr><tr><td>1</td><td>MAC empty working key support on.</td></tr></tbody></table>	Bit				2	1	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	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.
Bit																																																								
2	1	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	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.																																																							

Table 83 **Packet 18 Format** (continued)

Data Elements	Characteristics	Comments														
DUKPT Engine 1/2 Mode Flag [DEMF]	1AH	The one ASCII-Hex digit is used produce half of a control byte.														
Note: This field was added for IPP8 emulation.		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:														
	<table><tr><td>Engine=</td><td>1</td><td>2</td></tr><tr><td>DEMF = 0x30</td><td>0 1DES</td><td>1DES</td></tr><tr><td>0x31</td><td>1 3DES</td><td>1DES</td></tr><tr><td>0x32</td><td>2 1DES</td><td>3DES</td></tr><tr><td>0x33</td><td>3 3DES</td><td>3DES</td></tr></table>	Engine=	1	2	DEMF = 0x30	0 1DES	1DES	0x31	1 3DES	1DES	0x32	2 1DES	3DES	0x33	3 3DES	3DES
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

Table 84 Packet 18 Check IPP Key Management Mode

Master Device	Transmit Direction	IPP
<SI>18<SO>{LRC}	➡	
	⬅	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
	⬅	
		<SI>18[KMM][PINER]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

- 1DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18000<SO>{LRC}

- Mixed MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18010<SO>{LRC}

- 3DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18020<SO>{LRC}

- 1DES MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18080<SO>{LRC}

- Mixed MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18090<SO>{LRC}

- 3DES MS mode, zero key support off, zero GISKE session support off, and 3DES DUKPT mode:

<SI>180A0<SO>{LRC}

- 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, Verix eVo OS 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. 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}
```

Table 85 **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
[www...www] or DUKPT ENCRYPTION	16AH or 120AH	<p>[www...www] – encrypted working key (encrypted session key)</p> <p>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.</p> <p>Size of [www...www] indicates which packet format is used:</p> <ul style="list-style-type: none"> • 16AH – 1DES, key-only format • 120AH – GISKE key block format. For more details on GISKE refer <i>GISKE Key Block Spec</i>, VPN 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).

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:

$$04 \leq [\text{min len}] \leq [\text{max len}] \leq 12$$

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).

Table 86 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	<p>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.</p> <p>Size of [www...www] indicates which packet format is used:</p> <ul style="list-style-type: none"> • 16AH: 1DES, key-only format • 120AH: GISKE key block format. For more details on GISKE refer <i>GISKE Key Block Spec</i>, VPN 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).

Table 86 **Packet Z63 Format** (continued)

Data Elements	Characteristics	Comments
		<ul style="list-style-type: none"> 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.	EACCESS
See Restrict Speed of the PIN Encryption Operation .	

Packet M04: Read Permanent Unit Serial Number

This command is used to check the permanent unit serial number.

NOTE



This packet is added for IPP8 emulation.

Request Packet Format

<SI>M04<SO>{LRC}

Table 87 **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}

Table 88 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

Table 89 Packet M04 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>M04<SO>{LRC}	➔	
	➔	<ul style="list-style-type: none"> • ACK if LRC okay. • NAK if LRC incorrect (EOT after 3 NAKs).
	➔	<SI>M04[PUSN]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

Table 90 MS Packet 02 Format

Data Element	Characteristic	Comments
<STX>	1H	Shift In, Value: 0Fh
Packet Type	3AN	Value: 02

Table 90 MS Packet 02 Format (continued)

Data Element	Characteristic	Comments
[n]	1N	Address or key usage identifier. 1DES: <ul style="list-style-type: none"> Master key address is 0-9 3DES: <ul style="list-style-type: none"> Master key address for double- or triple-length keys is 0–9, 'Fa
[hhh...hh]	16H	Master key in ASCII. <ul style="list-style-type: none"> 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 <i>GISKE Key Block Spec</i>, VPN 22986.
<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 [Table 91](#).

Table 91 Default Key Attributes

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

Sample Packet 02 in Key-only Format

```
<SI>0200123456789ABCDEF<SO>{LRC}
```

Master Device	Transmit Direction	IPP
<SI>02[n][hhhhhhhhhhhhhhhhhh]<SO>{LRC}	➡	
	⬅	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
	⬅	<SI>02[n][hhhhhhhhhhhhhhhhhh]<SO>{LRC}
<ul style="list-style-type: none"> • ACK if LRC and key echo are okay • NAK if LRC incorrect • EOT after 3 NAKs • EOT if LRC correct, but key echo incorrect 	➡	
	⬅	<ul style="list-style-type: none"> • IPP saves the new master key only on receipt of ACK • EOT terminates session

Sample Packet 02 in GISKE Key Block Format:

<SI>

```
02000123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234
56789ABCDEF0123456789ABCDEF012345678901234567890123<SO>{LRC}
```

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 02
[n]	1N	Response code (0–7): <ul style="list-style-type: none"> • 0 = No error • 1 = Error: IPP in incorrect KM mode

Table 93 Packet 02 Response Format (continued)

Data Element	Characteristic	Comments
		<ul style="list-style-type: none"> • 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 range 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}

Table 94 Packet 02 GISKE Key Block Format Communication Protocol

Master Device	Transmit Direction	IPP
<SI>02[r][hhh.hhh]<SO>{LRC}	➡	
	⬅	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs
	⬅	<SI>02[n]<SO>{LRC}
<ul style="list-style-type: none"> • 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 	➡	
	⬅	<ul style="list-style-type: none"> • 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.

Table 95 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
	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 IPP (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}

Table 96 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: <ul style="list-style-type: none"> • 0 = No master key at address [a] • F = Master key present at address [a]

Table 96 Response Packet 04 Key-only Format (continued)

Data Element	Characteristic	Comments
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

Table 97 Response Packet 04 Key-only Format Communication Protocol

Master Device	Transmit Direction	IPP
<SI>040<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs • PIN pad checks requested address [a].
	←	
	→	<SI>04[r]<SO>{LRC}
<ul style="list-style-type: none"> • 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 memory (2 bytes).

<SI>042<SO>{LRC}

Table 98 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: <ul style="list-style-type: none"> • 0 = No master key at address [a] • F = Master key present at address [a]

Table 98 **Response Packet 04 GISKE Key Block Format** (continued)

Data Element	Characteristic	Comments
Key Usage Attribute (KUA)	2AH	<p>Only when master key is present at address [a]:</p> <ul style="list-style-type: none"> • 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)
Algorithm	1AH	<p>(optional) Only if the master key is present at address [a]. The value is stored in the Key Attributes register.</p> <ul style="list-style-type: none"> • 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 <p>Note: To save storage space in memory, the algorithm attribute is converted to [x], a hex number ranging from 0–F (4 bits). In the response packet (to packet 04), the IPP converts the number back to characters used in GISKE specification.</p>

Table 98 Response Packet 04 GISKE Key Block Format (continued)

Data Element	Characteristic	Comments
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. <ul style="list-style-type: none"> • 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 <p>Note: To save storage space in memory, 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.</p>
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 reinject old keys. If not used, this field is filled with ASCII 0 (0x30). <p>Note: The IPP allocates 1 byte per key for each key version register.</p>
Key Length (KL)	1AH	(optional) Only if the master key present at address [a]. <ul style="list-style-type: none"> • 1: Single-length key • 2: Double-length key • 3: Triple-length key <p>Note: The IPP allocates 1 byte per key for each key version register.</p>
<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}

Table 99 Response Packet 04 GISKE Block Format Communication Protocol

Master Device	Transmit Direction	IPP
<SI>04[a]<SO>{LRC}	➔	
	➔	<ul style="list-style-type: none"> • ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKS • PIN pad checks requested address [a].
	➔	<SI>04[r][KUA][MOUA][MACMA][KV][KL]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

Table 100 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}

Table 101 MS Packet 08 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>08[a]<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs • PIN pad makes master key [a] active.
	←	EOT

- 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.

Table 102 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:

Table 103 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
Error Code	1N	<ul style="list-style-type: none"> • 1 = No master key • 2 = Account or working key error • 3 = PIN too long. • 4 = PIN too short / non-decimal digits 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 (although unnecessary), 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 IPP: 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 (eg. packet Z60, Z63, 76, Z69 and 90).

NOTE



This packet was added for IPP8 emulation.

Table 104 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}

Table 105 DUKPT Packet 19 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>19[a]<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs).
	←	Echo packet 19 setting <SI>19[a]<SO>{LRC}
<ul style="list-style-type: none"> • ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs) 	→	

Table 105 **DUKPT Packet 19 Communication Protocol** (continued)

Master Device	Transmit Direction	IPP
		...
		IPP changes DUKPT engine
		...
	←	EOT to terminate process.

NOTE



- 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").

Request Packet Format

<SI>25<SO>{LRC}

Table 106 **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}

Table 107 **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}

Table 108 Packet 25 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>25<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> • ACK if LRC okay. • NAK if LRC incorrect (EOT after 3 NAKs).
	←	Response current active DUKPT engine. <SI>25[a]<SO>{LRC}
<ul style="list-style-type: none"> • 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.

Table 109 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.

Table 109 **DUKPT Packet 73 Format** (continued)

Data Element	Characteristic	Comments
<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:

Table 110 **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	<ul style="list-style-type: none"> • 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 do 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.

Table 111 DUKPT Packet 90 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	2AN	Value: 90
[IPEK]	16H	Initial PIN Encryption Key, hexadecimal
[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

- MAX: 57 characters
- MIN: 41 characters



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.

Table 112 DUKPT Packet 90 Communication Protocol

Master Device	Transmit Direction	IPP
90 Packet	→	
	←	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs • Initialization of 21 Future Keys
	←	
		Packet 91 with confirmed status
<ul style="list-style-type: none"> • 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.



Table 113 DUKPT Packet 91 Format

Data Element	Characteristic	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	2AN	Value: 91
[CS]	1N	Confirmation status: <ul style="list-style-type: none"> • 0 = Confirmed • 1 = Not confirmed • 2 = (IPP only) Error; incorrect key length. Confirmation status 2 only applies to IPP. 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
<ETX>	1H	End of Text, Value: 03h
{ LRC }	1H	Error Check Character

DUKPT Packet 91 Length

- MAX: 6 characters
- MIN: 6 characters

Table 114 DUKPT Packet 91 Communication Protocol

Master Device	Transmit Direction	IPP
		Packet 91
<ul style="list-style-type: none"> • 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.

Table 115 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.

Table 116 DUKPT Packet 76 Communication Protocol

Master Device	Transmit Direction	IPP
76 Packet	➡	
	⬅	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect
	⬅	Packet 71 with PIN = 1234
<ul style="list-style-type: none"> • 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.

Table 117 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:

Table 118 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	<ul style="list-style-type: none"> • 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 in the Verix eVo environment.

DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request

On receipt of the Z69 packet, Verix eVo OS 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.




Table 119 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

Table 120 DUKPT Packet Z69 Communication Protocol

Master Device	Transmit Direction	IPP
Z69 Packet		
		<ul style="list-style-type: none"> • ACK of LRC okay • NAK if LRC incorrect
		<ul style="list-style-type: none"> • EOT after 3 NAKS
		Packet 75 with confirmed status
		<ul style="list-style-type: none"> • 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.

Table 121 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
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>750000000D04396624CF6892B04A000002468000048D5D7AF0333800FD<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:

Table 122 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
Error Code	1N	<ul style="list-style-type: none"> • 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.

Table 123 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.
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error Check

DUKPT Packet 78 Length:

- MAX: 33 characters
- MIN: 18 characters

NOTE



As 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}

Table 124 DUKPT Packet 78 Communication Protocol

Master Device	Transmit Direction	IPP
78 Packet	→	
	←	<ul style="list-style-type: none"> • ACK if LRC • NAK if LRC incorrect
	←	Packet 75 with PIN = 1234
<ul style="list-style-type: none"> • 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. The following are the packets in this module:

- Z66: Request MAC
- Z67: Return MAC
- 72: Cancel MAC Session

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.

Table 125 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 <ul style="list-style-type: none"> • 4 = The last packet • 5 = The first/middle packet Binary Data; Range: 6–7 <ul style="list-style-type: none"> • 6 = The last packet • 7 = The first/middle packet

Table 125 MAC Packet Z66 Format (continued)

Data Element	Characteristic	Comments
[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 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:

Table 126 Rules for Request MAC

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.

Table 127 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: <ul style="list-style-type: none"> • 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
<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.

Table 128 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 }

Table 129 Packet 72 Communication Protocol

Master Device	Transmit Direction	IPP
<STX>72<ETX>{ LRC }		
		<ul style="list-style-type: none"> • 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 Verix eVo 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 [Table 130](#).

Table 130 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.



Account Data Encryption

Account Data Encryption (ADE) provides options for account data protection.

ADE Key Loading

The OS manages ten 2TDEA DUKPT engines for ADE. Key generation applies the “Data Encryption, request or both ways” variant as defined in ANSI X9.24-1:2009.

Unique Key Enforcement

The OS ensures that all keys are unique by rejecting duplicate key loads. Before a new DUKPT key is loaded, its Key Serial Number (KSN) is compared to the KSNs of previously loaded DUKPT keys to ensure uniqueness across IPP DUKPT, VSS DUKPT, and ADE DUKPT.

The target key slot during key loading is not checked for uniqueness, thus allowing a key to be replaced by an identical key. Key clearing caused by a key load is always performed before the uniqueness check.

Cleartext Key Loading via VTM

A new VTM menu option allows the user to load ADE keys. This function operates similarly to existing IPP key loading.

- Two passwords are required to access key loading—the VTM password and the Group 1 password. These are the same passwords required to access IPP key loading.
- Keys are sent in cleartext.

Cleartext key loading should only be performed in a secure environment.

Existing ADE keys are cleared at the start of a new cleartext key load session. IPP, VSS, VCL, and VRK keys are not affected. ADE cleartext key load session begins upon receiving the first valid cleartext ADE key load packet, and ends upon exiting key loading.

- Keys are loaded using a packet-based protocol on COM1 (COM2 on terminals that lack COM1).
- The default port setting is 19200 8N1.
- Two timeouts are used (same as IPP key loading):
 - 15 minute session timeout – time since session started
 - 1 minute communications timeout – time since terminal last received data from key loader

- Key loading session ends and exits by using these methods:
 - User presses an exit key
 - Timeout timer expires
- One of the following occurs upon exit:
 - If data was sent or received, or if either timeout expired, the terminal reboots.
 - If no data was transferred and the key load session did not time out, the terminal returns to the previous VTM menu.

Packet A90

Loads a 2TDEA ADE DUKPT key.

Table 131 ADE DUKPT Packet A90 Communication Protocol

Key Loader	Transmit Direction	Terminal
<STX>A90[Slot][IK][KSN]<ETX><LRC>	→	
	←	<ACK>
	←	<STX>A90[Result]<ETX><LRC>
<ACK>	→	
	←	<EOT>

The table below details the ADE DUKPT packet A90 format.

Table 132 ADE DUKPT Packet A90 Format

Data Element	Comments
[Slot]	2-byte ADE key slot, valid values: 00-09
[IK]	Initial DUKPT Key, 16-byte (convert to 32 ASCII Hex characters to put in packet)
[KSN]	Key Serial Number; 10-byte (convert to 20 ASCII Hex characters to put in packet)
Result	2-byte result code: <ul style="list-style-type: none"> • 00 = Success • 01 = Internal error • 02 = Invalid format • 03 = Unsupported key slot • 04 = Duplicate key

Encrypted Key Loading via VRK

ADE keys can be loaded using VRK. The TR-31 header must use the following values:

- Algorithm: “T” (TDEA)
- Mode Of Use: “E” (Encryption)
- Optional Block 04 (Key Engine Type): “a” (ADE)
- Optional Block 05 (Key Slot of Engine): Single digit between 0 and 9 inclusive

ADE APIs

Below are the ADE APIs defined 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 Return Codes

```
#define ADE_SUCCESS 0
```

```
#define ADE_ERR_PM_PTR -1
```

```
#define ADE_ERR_PM_LEN -2
```

```
#define ADE_ERR_PM_KEY -3
```

```
#define ADE_ERR_PM_ALG -4
```

```
#define ADE_ERR_PM_MODE -5
```

```
#define ADE_ERR_PM_IV -6
```

```
#define ADE_ERR_PM_PAD -7
```

```
#define ADE_ERR_NO_KEY -11
```

```
#define ADE_ERR_OFF -12
```

```
#define ADE_ERR_GENERIC -99
```

// 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 setting and ADE key.

Prototype `int ade_encrypt(ade_encrypt_in* in, ade_encrypt_out* out);`

Parameters

Inputs

<code>in->ptext</code>	Plaintext data to encrypt
<code>in->ptextLen</code>	Number of bytes to encrypt Valid values: <ul style="list-style-type: none">• If <code>in->pad</code> does not equal <code>ADE_PAD_NONE</code>, between 1 and 383, inclusive.• If <code>in->pad</code> equals <code>ADE_PAD_NONE</code>, between 1 and 384, inclusive, and must be divisible by the block size of the encryption algorithm (<code>in->encAlg</code>).
<code>in->keyIndex</code>	ADE key index Valid values: <ul style="list-style-type: none">• Between 0 and 9, inclusive = ADE DUKPT engines 0 through 9
<code>in->encAlg</code>	Encryption algorithm Valid values: <ul style="list-style-type: none">• <code>ADE_ALG_TDEA</code> = TDEA (Block size: 8 bytes)
<code>in->encMode</code>	Encryption mode of operation Valid values: <ul style="list-style-type: none">• <code>ADE_MODE_ECB</code> = ECB mode• <code>ADE_MODE_CBC</code> = CBC mode
<code>in->iv</code>	IV Setting Valid values: <ul style="list-style-type: none">• <code>ADE_IV_NONE</code> = No IV (not allowed for CBC mode)• <code>ADE_IV_ZERO</code> = Zero IV• <code>ADE_IV_RAND</code> = Random IV
<code>in->pad</code>	Padding scheme Valid values: <ul style="list-style-type: none">• <code>ADE_PAD_NONE</code> = No padding added• <code>ADE_PAD_PKCS7</code> = PKCS#7 padding<ul style="list-style-type: none">• Each padding byte equals the padding length. Example: XX XX XX XX 04 04 04 04• <code>ADE_PAD_X923</code> = ANSI X.923 padding

- Final padding byte equals the padding length and all other padding bytes equal 0x00.
Example: XX XX XX XX 00 00 00 04
- ADE_PAD_ISO7816 = ISO/IEC 7816-4 padding
 - First padding byte equals 0x80 and all other padding bytes equal 0x00.
Example: XX XX XX XX 80 00 00 00

Outputs

out->ctxt	Ciphertext
	<ul style="list-style-type: none"> • This output buffer must be able to hold the entire ciphertext, which does not exceed 384 bytes.
out->ctxtLen	Number of bytes of ciphertext
out->ivData	IV data
	<ul style="list-style-type: none"> • This output buffer must be able to hold the entire IV, which never exceeds the encryption algorithm's block size. • If in->iv equals ADE_IV_NONE, then out->ivData is not used.
out->supData	Supplementary Data
	<ul style="list-style-type: none"> • This output buffer must be able to hold any supplementary output data. • If a DUKPT engine was used, then out->supData will contain the 10-byte KSN.

Return Values

Success:	ADE_SUCCESS	Encryption succeeded without error.
Failure:	ADE_ERR_PM_PTR	A pointer parameter references memory that is unusable (for example, null pointer, buffer too small, and so on.).
	ADE_ERR_PM_LEN	Parameter in->ptextLen is an invalid value.
	ADE_ERR_PM_KEY	Parameter in->keyIndex is an invalid value.
	ADE_ERR_PM_ALG	Parameter in->encAlg is an invalid value.
	ADE_ERR_PM_MODE	Parameter in->encMode is an invalid value.
	ADE_ERR_PM_IV	Parameter in->iv is an invalid value.
	ADE_ERR_PM_PAD	Parameter in->pad is an invalid value.
	ADE_ERR_NO_KEY	A key cannot be retrieved from the specified slot.
	ADE_ERR_OFF	ADE is turned off.
	ADE_ERR_GENERIC	An undefined or unexpected error has occurred.

Example

Click on the linked example to view the sample code for ADE encryption. Refer to [Parameters](#) for more information.

ade_status()

Returns the status of the ADE module.

Prototype `int ade_status();`

Return Values

This function returns a 32-bit map of the ADE status.

- ADE_STS_ENG0 - ADE_STS_ENG9: Status of each ADE DUKPT engine
 - 0 = Engine is not usable (that is, no key loaded or engine exhausted)
 - 1 = Engine is usable (that is, key is loaded)
- ADE_STS_ON: ADE on/off status
 - 0 = ADE is turned off
 - 1 = ADE is turned on
- ADE_STS_ENABLED: ADE feature status
 - 0 = ADE is disabled
 - 1 = enabled

ade_active()

A Boolean check indicating if the ADE system is fully functional.

Prototype `int ade_active();`

Return Values

Success: Non-0 = ADE is on and has at least one usable key loaded.

Failure: 0 = The ADE system is currently not usable.

ADE State Controls

The ADE module is controlled by the following three states:

- ADE keys loaded or not (default: no keys loaded)
- ADE turned on or off (default: off)
- Feature enabled or disabled (default: disabled)

The ADE must be turned on and at least one usable ADE key must be loaded in order for the ADE module to be in an encrypting state (that is, capable of performing ADE encryptions).

Turning ADE On or Off

ADE can only be turned on or off using a new VTM menu. Two passwords are required to access this menu—the VTM password and the Group one password. These are the same passwords required to access IPP key loading.

ADE can only be turned on if both of the following are true:

- At least one ADE key is loaded and usable.
- A current valid FE license for ADE is installed.

Turning ADE off does not involve additional checks.

Feature Enablement

A valid and current ADE license must be installed in order to turn on ADE. Turning on ADE enables an on-demand license and/or decrement a COUNT license.

The tag value for the VX eVo ADE feature is [0106,0101].

ADE Status Displays

This section details the startup display, non-transient visual indication, and the VTM ADE status menu.

Startup Display

The OS briefly displays a splash screen during OS startup (boot). This splash screen changes depending on the ADE module's state.

- If the ADE module is in an encrypting state (that is, `ade_active() != 0`), the OS firmware version is displayed with “.1” appended to it. (for example, QT680101.1).
- If the ADE module is NOT in an encrypting state (that is, `ade_active() == 0`), the OS firmware version is displayed with “.0” appended to it. (for example, QT680101.0).

Non-transient Visual Indication

The device must display a non-transient visual indication if ADE is active. Since the application owns the console, it is responsible for displaying the ADE indicator. The API `ade_active()` may be used for this purpose.

VTM ADE Status Menu

VTM ADE menu allows the user to view the current status of the ADE module. Refer to [ADE VTM Menus](#) for more information.

ADE VTM Menus

ADE Status Menu

New menus are added to the VTM for ADE. Two passwords are required to access any ADE secure operation menu—the VTM password and the Group one password. These are the same passwords required to access IPP key loading.

This menu allows the user to view the current ADE status. The following status displays are available:

- Key status for each engine
- ADE on/off status
- ADE feature status

This menu displays the same information that is available through the `ade_status()` API.

ADE Key Load Menu

This menu starts the ADE cleartext key loading service. Refer to [Cleartext Key Loading via VTM](#) for more information.

ADE On/Off Menu

This menu allows the user to turn ADE on or off. Refer to [Turning ADE On or Off](#) for more information.

ADE Menu Flowchart

Figure 20 shows the process flow of the ADE menu.

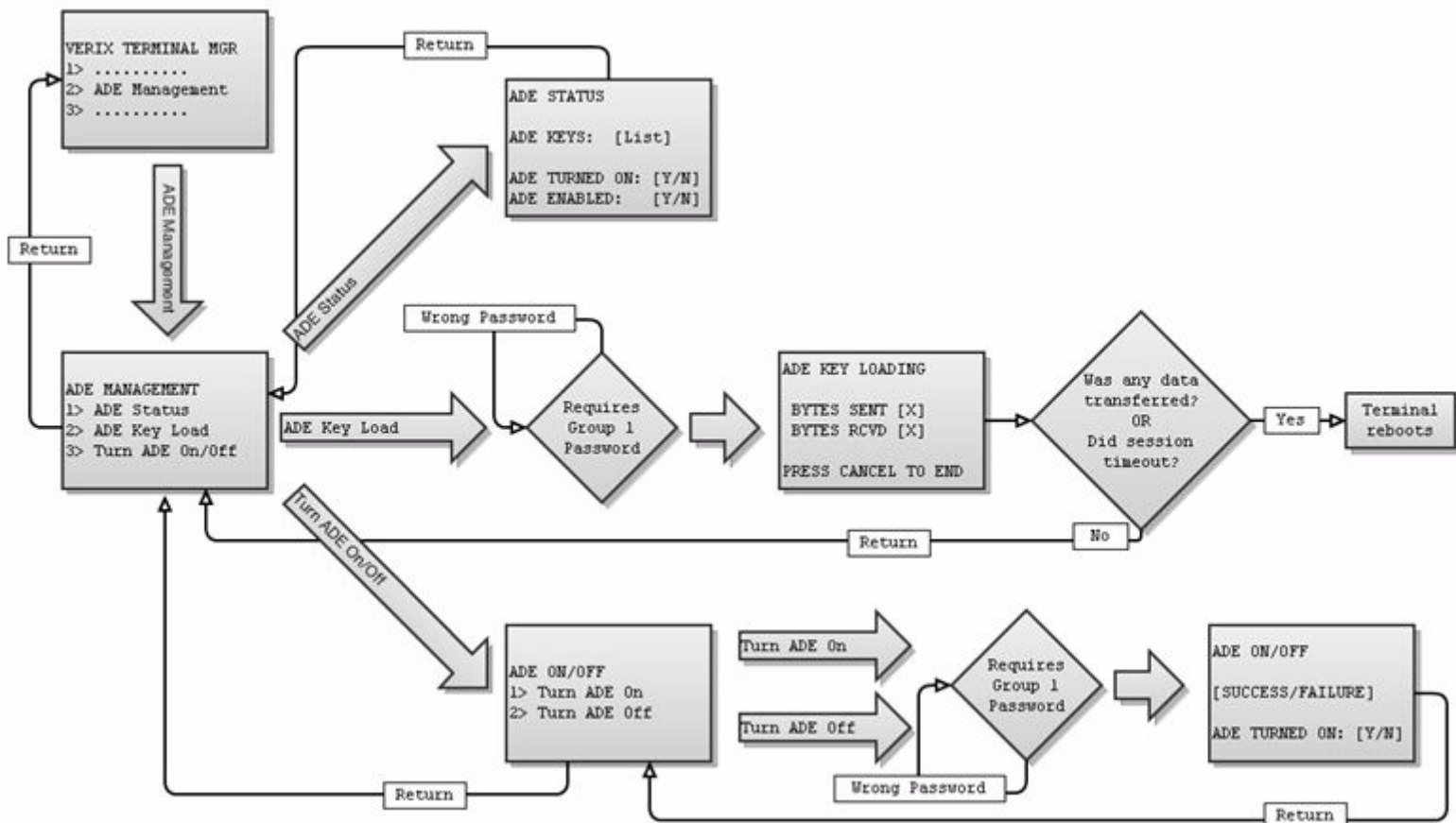


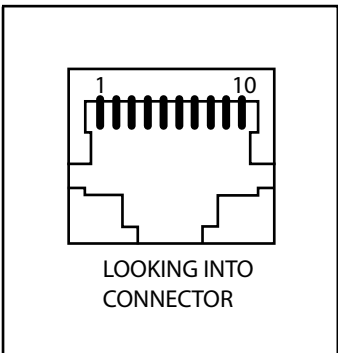
Figure 20 ADE Menu Flowchart

Special Topics

Smart Card PIN Entry using an External PIN Pad

This appendix discusses topics that cover more than one subject area.

Trident terminals do not support a 2 wire UART with power pin. Instead, COM1 has +9v power available and can be used for PINPad 1000 and SC5000. The port can be used for asynchronous communications, at baud rates from 300–115200 and *only* supports character mode.

	Pin	Signal
	1	n/c
	2	VPinPAD (+9V DC regulated power) ^a
	3	n/c
	4	n/c
	5	GND
	6	RXD
	7	TXD
	8	n/c
	9	n/c
	10	n/c

a. Maximum 450 mA.

Signal	Function	Signal	Function
GND	Signal Ground	TXD	Transmit data
RXD	Receive Data	VPinPAD	PIN pad device power

Some smart card applications (for example, EMV) require users to enter a PIN using an external PIN pad. A conflict arises in that PIN pads encrypt PINs before transmitting them over unprotected data links, but the smart card reader will accept only unencrypted PINs. The solution is having the terminal and the external PIN pad share an encryption key. The application and Verix eVo OS implement the host side master/session key management protocol.

This encryption method assumes the master key can be loaded into individual PIN pads. Some PIN pads however, require all keys to be loaded in a single operation. For these devices, the session key must be loaded at the same time as the host master keys, and the same key must then be stored on the terminal. This particular usage is not supported by the current interface.

NOTE

The application is responsible for all communication with the PIN pad and the smart card, and all knowledge of their protocols, data formats, and so on. The Verix eVo environment provides only core cryptographic services.

Security Considerations

Verix eVo-based terminals store the master and session keys encrypted using the same process used to store IPP master keys. The master key used for smart card PIN entry is not an IPP master key. The IPP master keys are completely separate.

Unencrypted PINs and keys are vulnerable when they are passed back to application code. Applications that handle them must be written carefully to avoid further weakening security.

For security considerations, please observe the following guidelines:

- 1 Avoid storing PIN or key values longer than absolutely necessary. Once a PIN is sent to the smart card, erase the data in memory. *Never* write this data to files.
- 2 For sensitive applications, disable debugging. A debuggable task can be spied on by the debugger or any other program that knows about the debugging interface. Use the VRXHDR utility to clear the `.out` file debug flag.
- 3 Use file authentication properly. Change the default keys and ensure that new keys remain secret.
- 4 Limit the visibility of PINs by encapsulating critical functions in libraries. For example, the entire process of entering a PIN and transferring it to the smart card can be handled by a single library function, with the caller never seeing the sensitive data. More protection can be obtained by running this library function in a separate server task so that the PIN never appears in the application task's data space.

**Master-Session
Key Management
Function Calls and
Smart Card PIN
Entry**

The master-session key management function calls support EMV smart card PIN input from an external PIN pad. The basic problem is that PIN pads encrypt PINs before returning them to applications. EMV PINs however, must be presented to the smart card as clear text. The solution is for the terminal and PIN pad to share an encryption key.

The application and Verix eVo operating system implement the host-side of a master-session key management protocol, supplying an encryption key to the PIN pad that can then be used to decrypt the PIN block. The following outlines this process:

- 1 The application requests the Verix eVo operating system to generate a *master* key using `gen_master_key()`. The Verix eVo operating system returns it to the caller and stores it for its own use.
- 2 The application downloads this master key to the PIN pad.

- 3 The application requests the Verix eVo operating system to generate a *session* key and encrypt it with the master key, using `gen_session_key()`. Verix eVo OS returns it to the caller and stores it for its own use.
- 4 The application sends the encrypted session key to the PIN pad. The PIN pad decrypts it using the master key.
- 5 The application sends a “get PIN” request to the PIN pad using a dummy account number.
- 6 The user enters their PIN. The PIN pad forms a PIN block, encrypts it with the session key, and returns the result.
- 7 The application requests that Verix eVo OS decrypt the PIN block using the stored session key, using the `decrypt_session_data()` function call.
- 8 The application extracts the PIN from the decrypted PIN block and sends it to the smart card.

Note that all communication with the PIN pad and smart card and all knowledge of their protocols, data formats, and so on is the responsibility of the application code (which may include VeriFone-supplied libraries).

Master/Session Functions

This section describes Master/Session function calls used for smart card PIN entry.

decrypt_session_data()

DES decrypts 8 bytes of data with the current session key. The key must have been set by a prior call to [gen_session_key\(\)](#) in the same task. Decryption is done in place that is, the result replaces data. See also [gen_session_key\(\)](#) and [gen_master_key\(\)](#).

Prototype `int decrypt_session_data (char *data8);`

Return Values

Success: 0

Failure: -1 errno set to `EACCES`: Invalid key pointer.

 -1 errno set to `ENOENT`: No master key loaded.

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

gen_master_key()

Generates and stores a master key. The 8-byte binary key value is returned in `key8` and stored internally in non-volatile memory for future use. Only tasks running in Group 1 are allowed to set or erase the master key.

Prototype

```
int gen_master_key (char *key8, int options);
```

Parameters

`options` The options listed below control if the master key is random or derived from a seed, how its parity bits are handled, and a way to clear it.

Symbolic Name	Value	Description
KEY_PARITY_ZERO	1	Set key parity bits to zero.
KEY_PARITY_EVEN	2	Set key parity even.
KEY_PARITY_ODD	3	Set key parity odd.
KEY_SEEDED	4	Use input key value as a seed to generate the key.
KEY_ERASE	8	Clear the stored master key.

KEY_PARITY Options

Selected options are added (ORed) together. Only one of the three `KEY_PARITY` options can be used. The parity bits are the LSB of each byte of the key. Even parity means that the number of “1” bits in the byte, including the parity bit, is even. If no parity option is specified, the parity bits are essentially random.

KEY_SEEDED Option

If the `KEY_SEEDED` option is used, the input value of `key8` is the seed to generate a random-appearing-but-reproducible key. This makes it possible to set the same key on multiple terminals so that PIN pads can be used interchangeably among them. It does not allow a particular key value to be set because you cannot determine the seed required to generate a specific key. If the `KEY_SEEDED` option is not present, a seed is constructed from the internal time-varying data that generates an essentially random key.

KEY_ERASE Option

If `KEY_ERASE` is specified, the stored master key is deleted from memory. All other options are ignored. A valid `key8` pointer is required, even if it is not used.

NOTE



There is no application-callable function for retrieving the stored key.

Return Values

Success: 0

Failure: -1 errno set to `EACCES`: Invalid key pointer.

 -1 errno set to `EPERM`: Calling task is not in Group 1.

gen_session_key()

Generates a (reasonably) random session key and DES encrypts it with the current master key. The 8-byte result is returned in `key8` and stored for use by [decrypt_session_data\(\)](#).

Session keys can only be used by the task that created them and are not preserved when the terminal is reset. See also [gen_master_key\(\)](#) and [decrypt_session_data\(\)](#).

Prototype `int gen_session_key (char *key8, int options);`

Parameters

`options` The same options as used for [gen_master_key\(\)](#), except that `KEY_SEEDED` is ignored if present.

Return Values

Success: 0

Failure: -1 `errno` set to `EACCES`: Invalid key pointer.

 -1 `errno` set to `ENOENT`: No master key loaded.

test_master_key()

Tests if a master key has been stored by [gen_master_key\(\)](#).

Prototype `int test_master_key (void);`

Return Values

0 No master key present.

1 Master key present.

Support for APACS40 Cryptographic Functions

The Verix eVo operating system includes a suite of cryptographic functions to support the APACS40 standard of the Association for Payment Clearing Services in the United Kingdom and elsewhere. For detailed information on APACS40, refer to the document, *Standard 40: Acquirers Interface Requirements for Electronic Data Capture Terminals EFT Interface* (version 14, 30 September 1998).

Software Block Diagrams

Figure 21 shows the relationship between the various function calls, in which one requires the result of another. For example, neither `Calc_MAC` nor `Create_Auth_Parm` can be called until `Create_MAC_Key` has been called, and once `New_Host_Key` has been called, `Create_MAC_Key` must be called again.

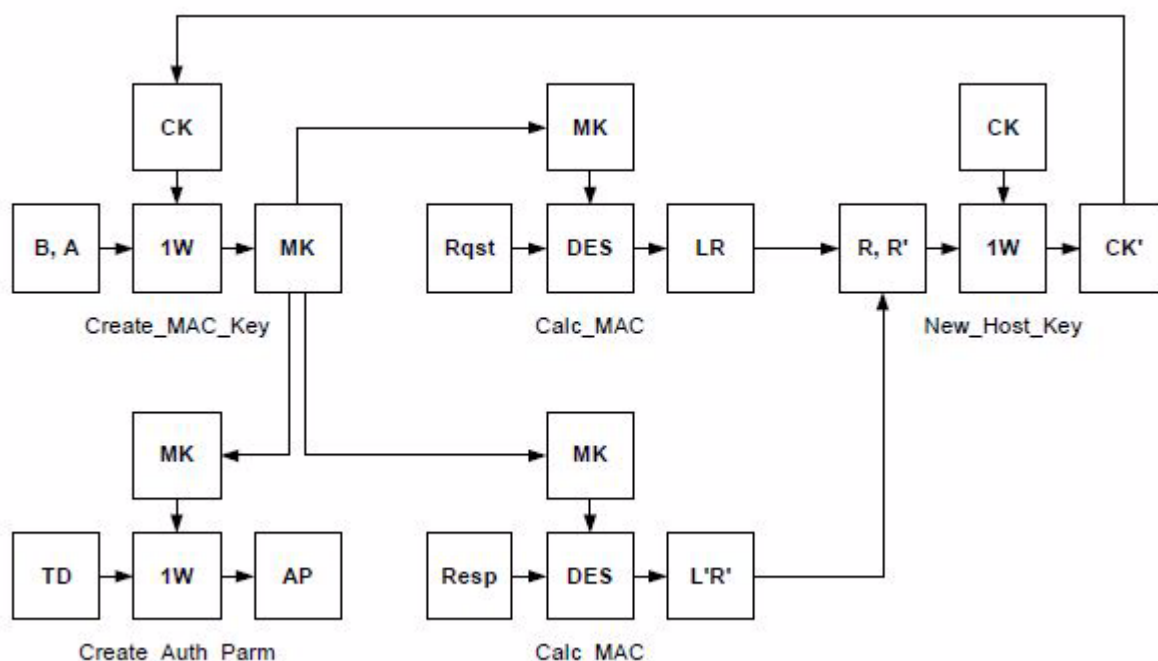


Figure 21 APACS40 Function Call Relationships

The keys are stored in a file in Group 0. One hundred pairs of keys are stored; one pair for each supported financial institution. The two keys in a pair are:

- The *seed* key, which is initially loaded, and
- The *current* key, which evolves with each transaction

The current key of the pair is set to the seed key when the seed key is initially loaded and whenever the `Reset_Key` function is called due to loss of synchronization. For convenience, all current keys are stored first, then all seed keys.

Keys for APACS40 operation must be loaded with a special VeriFone-developed key-loading application in a properly secured area. Arrangements for this service should be made in conjunction with the purchase of the terminals.

Calc_Auth_Parm()

Computes the authentication parameter based on the provided transaction data. The key used is the current MAC key that was set by a prior call to the `Create_MAC_Key` routine, which uses the current host key. Note that the MAC key is also used to compute the conventional MAC within the `Calc_MAC` function.

NOTE



Before using this function, you must call `Create_MAC_Key`.

Prototype `int Calc_Auth_Parm(const char *TranData, char *AuthParm);`

Return Values

Success: 0: Completes normally.

Failure: -1 and `errno` set to `EACCES`: One of the caller's buffers cannot be read (invalid pointer).

-1 and `errno` set to `EBUSY`: Another caller owns the feature.

-1 and `errno` set to `EINVAL`: Current MAC key is invalid.

Calc_MAC()

Computes the standard ANSI X9.19 message authentication code for the designated buffer. The entire MAC is returned to the application. The key used is the one previously set by calling the `Create_MAC_Key` function, which is based on the current host key.

NOTE



Before using this function, you must call `Create_MAC_Key`.

Prototype `int Calc_MAC(const char *buf, int len, char *mac8);`

Return Values

Success: 0: Completes normally.

Failure: -1 and `errno` set to `EACCES`: One of the caller's buffers cannot be read (invalid pointer).

-1 and `errno` set to `EBUSY`: Another caller owns the feature.

-1 and `errno` set to `EINVAL`: Current MAC key is invalid.

Create_MAC_Key()

Sets the current MAC key based upon use of the One Way function. Call this function at the beginning of a transaction. It must be called prior to calling `Calc_Auth_Parm` or `Calc_MAC`. (For information on the One Way function, refer to the APACS40 specification.)

The `hostkey` parameter specifies which host is involved. The MAC key is generated from the current value of the key for this host.

Prototype

```
int Create_MAC_Key(int hostkey, const char *A, const char *B);
```

Return Values

Success: 0: Completes normally.

Failure: -1 and `errno` set to `EACCES`: One of the caller's buffers cannot be read (invalid pointer).

-1 and `errno` set to `EBADF`: The called file is missing, not properly initialized, or corrupt.

-1 and `errno` set to `EBUSY`: Another caller owns the feature.

-1 and `errno` set to `EINVAL`: The host key is out of range.

Init_MAC()

Allows multiple tasks to use the APACS40 features (one at a time). In a multi-application environment, call this function at the beginning of a transaction. Call the function `Term_MAC` at the end of the transaction.

Prototype

```
int Init_MAC(void);
```

Return Values

Success: 0: Completes normally.

Failure: -1 and `errno` set to `EBUSY`: Another caller owns the feature.

NOTE



If any APACS40 call returns a result of `EBUSY`, the APACS40 feature is in use by another task. That task must use the `Term_MAC` function to release the APACS feature.

New_Host_Key()

Updates the current host key for the designated host. After computing the new value of current host key, the firmware encrypts it and writes it back to the file in Group 0.

Prototype

```
int New_Host_Key(int hostkey, const char *rqst_residue,  
const char *resp_residue);
```

Parameters

hostkey	A number from 0 through 99.
rqst_residue resp_residue	Represent the low-order four bytes of the eight-byte MAC computed for the request message and the response message.

Return Values

Success:	0: Completes normally.
Failure:	–1 and errno set to <code>EACCES</code> : One of the caller's buffers cannot be read (invalid pointer).
	–1 and errno set to <code>EBADF</code> : Called file is missing, not properly initialized, or corrupt.
	–1 and errno set to <code>EBUSY</code> : Another caller owns the feature.
	–1 and errno set to <code>EINVAL</code> : Host key is out of range.

Reset_Key()

Resets the current host key for the designated host. The seed key initially loaded for this host again becomes the current key. Typically this reset operation must be performed when the terminal and host lose synchronization.

Prototype

```
int Reset_Key(int hostkey);
```

Return Values

Success:	0: Completes normally.
Failure:	–1 and errno set to <code>EBUSY</code> : Another caller owns the feature.
	–1 and errno set to <code>EBADF</code> : Called file is missing, not properly initialized, or corrupt.
	–1 and errno set to <code>EINVAL</code> : Host key is out of range.

Term_MAC()

Clears the current owner variable of the APACS40 feature set so that another task can use the feature. Verix eVo OS maintains a variable (initially 0) that holds the task ID of the current *owner* of the APACS40 feature. Whenever a call is made on any feature, the caller's ID is compared with the saved value. If a mismatch is detected, the caller assumes ownership (if there was no previous owner) or the caller is rejected with `errno` set to `EBUSY`.

Prototype

```
int Term_MAC(void);
```

Return Values

Success: 0: Completes normally.

Failure: -1 and `errno` set to `EBUSY`: Another caller owns the feature.

NOTE

If any APACS40 call returns a result of `EBUSY`, the APACS40 feature is in use by another task. That task must use the `Term_MAC` function to release the APACS feature.

Example

The linked C code example is not intended to be all inclusive, but demonstrates basic functions and features.

String Utilities

The functions listed in this section perform various conversions on strings:

- `dtoa()`
- `SVC_2INT()`
- `strnupr()`
- `SVC_UNPK4()`
- `ltoa()`
- `DLLs`
- `SVC_HEX_2_DSP()`
- `ultoa()`
- `SVC_AZ2CS()`
- `SVC_DSP_2_HEX()`
- `SVC_INT2()`
- `strnlwr()`
- `SVC_PACK4()`

dtoa()

Converts the double precision floating point value `val` to a string in `buf`. The result is similar to that of

```
sprintf(buf, "%.<precision><format>", val);
```

Thus `format` is 'f', 'e', 'E', 'g', or 'G'. The function returns the length of the string. If `bufsize` is too small to accommodate the full result, the required length returns. If an argument is not valid the result is -1.

The standard C method for converting numeric values to strings is `sprintf`. However, on Verix eVo-based platforms, `sprintf` does not support floating point arguments. For example the code

```
sprintf(buf, "%8.3f", val);
```

can be replaced by

```
dtoa(val, buf, sizeof(buf), 'f', 3);  
sprintf(buf, "%8s", buf);
```

Prototype

```
int dtoa (double val, char *buf, int bufsize, int format, int precision);
```

Parameters

<code>val</code>	Floating point value to convert.
<code>buf</code>	Pointer to buffer to store converted value.
<code>bufsize</code>	Size of the buffer.
<code>format</code>	One of 'f', 'e', 'E', 'g', or 'G'
<code>precision</code>	Floating point value.

Return Values

Success: ≥ 0 : Number of bytes stored in `buf`.

The result is equivalent to:

```
sprintf(buf, "%.<precision><format>", d);
```

ltoa()

Converts a number to a string. Alternatives to `ltoa` include [SVC_INT2\(\)](#) and the standard C library function `sprintf`. See also [ultoa\(\)](#) and [SVC_INT2\(\)](#).

Prototype

```
char *ltoa (long value, char *buf, int radix);
```

Parameters

<code>value</code>	The number to convert to an ASCII string. The result is stored in <code>buf</code> as a zero-terminated string.
<code>buf</code>	Where the converted number will be stored.
<code>radix</code>	Specifies the conversion radix. For example, 10 converts to decimal; 16 to hex. Valid radix values are 2–16. Negative numbers are formatted with a minus sign for radix 10 only. For other radices the value is treated as unsigned and the function is equivalent to ultoa() .

Return Values

The return value is the address of `buffer`. The effect of passing invalid arguments is undefined.

Example

ultoa()

Converts a number to a string. It is identical to [ltoa\(\)](#), except that `value` is interpreted as unsigned in all cases. See also [SVC_INT2\(\)](#).

Prototype

```
char *ultoa (unsigned long value, char *buf, int radix);
```

Parameters

<code>value</code>	The number to convert to an ASCII string.
<code>buf</code>	address of buffer to store string
<code>radix</code>	Specifies the conversion radix. For example, 10 converts to decimal; 16 to hex. Valid radix values are 2–16. Negative numbers are formatted with a minus sign for radix 10 only. For other radices the value is treated as unsigned and the function is equivalent to ltoa() .

Return Values

The return value is the address of `buffer`.

strnlwr()

Converts string to lowercase. Copies a string from `source` to `dest`, changing uppercase letters to lowercase. The conversion is done in place if `source` and `dest` are the same; otherwise, they should not overlap. Copying continues until a zero terminator is found or `size` characters have been copied. In either case, a zero terminator is appended to the result. Note that `dest` should contain `size+1` bytes to allow for the terminator. See also [strnupr\(\)](#).

Prototype `void strnlwr (char *dest, char *source, int size);`

Parameters

<code>source</code>	Input string to convert.
<code>dest</code>	Address to stored converted string.
<code>size</code>	Size of buffer at <code>dest</code> .

Example

strnupr()

Converts string to uppercase. Copies a string from `source` to `dest`, changing lowercase letters to uppercase. Except for the direction of the case conversion, the function is the same as [strnlwr\(\)](#).

Prototype `void strnupr (char *dest, char *source, int size);`

Parameters

<code>source</code>	Input string to convert.
<code>dest</code>	Address to stored converted string.
<code>size</code>	Size of buffer at <code>dest</code> .

SVC_HEX_2_DSP()

Converts binary data to ASCII text. [ultoa\(\)](#) and the standard C library function `sprintf` provide alternate ways to format binary data as hexadecimal. See also [SVC_DSP_2_HEX\(\)](#), and [ultoa\(\)](#).

Prototype `void SVC_HEX_2_DSP (char *hex, char *dsp, int n);`

Parameters

`hex` The input in hex consists of n bytes of binary data.

`dsp` The result stored in `dsp` consists of $2 \times n$ ASCII hexadecimal digits, 0–9 or A–F. It is not zero-terminated.

`n` Number of bytes to convert.

`hex` and `dsp` should not overlap.

For example, if $n = 2$ and `hex` points to the two bytes { 0x4A, 0x2E }, then the result stored in `dsp` is { 0x34, 0x41, 0x32, 0x45 }, or the equivalent { '4', 'A', '2', 'E' }.

Note that if `SVC_HEX_2_DSP` is used to convert an integer value to hex, the result reflects the internal byte order. On little-endian processors, such as the Z80, the least-significant byte is first, which is probably not what was intended.

Example The linked code example creates a data packet for sending a master key to a PIN pad. PIN pad data is sent in ASCII format.

SVC_DSP_2_HEX()

Converts ASCII hexadecimal data to binary. See also [SVC_HEX_2_DSP\(\)](#).

Prototype `void SVC_DSP_2_HEX (char *dsp, char *hex, int n);`

Parameters

`dsp` The input in `dsp` consists of $2 \times n$ characters, one per hex digit. If `dsp` contains invalid hex digits, the result is undefined.

`hex` The output in `hex` is n bytes.

`dsp` and `hex` should not overlap.

`n` Number of bytes to convert.

Hex digits in the input can be either upper or lowercase.

For example, if `dsp` points to the four bytes { 0x34, 0x41, 0x32, 0x65 } or the equivalent { '4', 'A', '2', 'e' }, and $n = 2$, then the two bytes { 0x4A, 0x2E } are stored in `hex`.

Note that if `SVC_DSP_2_HEX` is used to convert a string to an integer value, the byte order will be wrong on little-endian processors, such as the Z80.

Results If `dsp` contains invalid hex digits, the result is undefined.

Example Use the linked code example file to convert ASCII-hex characters to hexadecimal.

SVC_PACK4()

Compresses ASCII data. The compression algorithm is the same one used for CVLR. It is effective on data that consists primarily of numeric digits. Non-numeric data is not compressed, and byte values greater than 0x5F are changed on decompression. See [write\(\)](#) for a detailed description. See also [SVC_UNPK4\(\)](#).

`SVC_PACK4()` compresses `size` bytes of data from `source` into `dest`.

Prototype

```
int SVC_PACK4 (char *dest, char *source, int size);
```

Parameters

`dest` Pointer to buffer to store compressed data.

`source` Pointer to ASCII data to compress.

`size` Must be ≤ 255 .

If `source` and `dest` point to the same buffer, the compression is done in place. Otherwise the buffers should not overlap.

Return Values

The function returns the number of compressed bytes, or -1 if `size` is out of range. The result can be decompressed by [SVC_UNPK4\(\)](#).

Example

Click the linked example to view the sample code.

SVC_UNPK4()

Decompresses ASCII data compressed by [SVC_PACK4\(\)](#). See [write\(\)](#) for details of the compression algorithm, and [SVC_PACK4\(\)](#) for a linked code example. See also [read\(\)](#).

`size` bytes of compressed data from `source` are expanded into `dest`, and the number of decompressed bytes returned.

Prototype

```
int SVC_UNPK4 (char *dest, char *source, int size);
```

Parameters

`dest` Pointer to buffer to store compressed data.

`source` Pointer to ASCII data to compress.

`size` Must be ≤ 255 .

If `source` and `dest` point to the same buffer, the compression is done in place. Otherwise the buffers should not overlap.

Return Values

The function returns -1 if `size` is out of range. `dest` must have room for $2 \times \text{size}$ bytes to handle the maximum compression case.

Counted Strings

Counted strings are used by a number of Verix eVo calls, mostly for historical reasons. The first byte of a counted string contains its length, including the count byte. For example, “hello” is represented as an array of 6 bytes, the first having the value 6 and the next 5 containing the ASCII data bytes. There is no terminator. The maximum length is 254 data bytes (count byte = 255). The following APIs are discussed in this section.

- `SVC_CS2AZ()`
- `SVC_AZ2CS()`
- `SVC_INT2()`
- `SVC_2INT()`

SVC_CS2AZ()

Converts counted string, `cstring`, to a standard C zero-terminated string, `zstring`. If `zstring` and `cstring` point to the same buffer, the conversion is done in place. Otherwise they should not overlap. See also [SVC_AZ2CS\(\)](#).

Prototype `int SVC_CS2AZ (char *zstring, const char *cstring);`

Parameters

`zstring` Counted string to convert.
`cstring` Pointer to buffer to store output string.

Example The linked code example contains three different techniques for converting a counted string to a zero-terminated string.

SVC_AZ2CS()

Converts a zero-terminated string, `zstring`, to a counted string, `cstring`. `cstring` and `zstring` should not overlap. See [DLLs](#) for a description of counted strings.

Prototype `int SVC_AZ2CS (char *cstring, const char *zstring);`

Parameters

`cstring` Pointer to buffer to store output string.
`zstring` Counted string to convert.

Example The linked file contains a function to convert a string to an integer using [SVC_2INT\(\)](#), which requires a counted string argument.

SVC_INT2()

Converts a number to a string. See [DLLs](#) for a description of counted strings. See also [ltoa\(\)](#), [ultoa\(\)](#), and [SVC_2INT\(\)](#).

Prototype `void SVC_INT2 (unsigned int value, char *buf);`

Parameters

value	The number to convert.
buf	The address of buffer to store the string.

Example Click the linked example to view the sample code.

SVC_2INT()

Converts a null-terminated ASCII string containing decimal digits to a binary value. Non-numeric characters are ignored; this includes the minus sign, if present. An empty string or string containing no digits is converted to 0. The standard C library functions `strtol`, `strtoul`, `atoi`, and `atol` are alternatives to this function. See also [SVC_INT2\(\)](#).

Prototype `unsigned int SVC_2INT (const char *source);`

Parameters

source	Null-terminated string to convert.
--------	------------------------------------

Return Values If the value is too large to fit in an unsigned int, the result is undefined. See [DLLs](#) for a description of counted strings.

Source	Result	Notes
"123"	123	
"-42"	42	Non-numeric "-" is ignored.
"Win98 ver 1.3"	9813	All non-digits characters are ignored.
""	0	Zero length string.

DLLs

Support for shared libraries has improved significantly and further enhancements are in the works. Some of these features are available using newer versions of the ARM compiler and linker—at least RVDS 2.2 or preferably RVDS 4.0.

Removing DLLs

To remove the library and recover the space used by it, the function `unload_DLL()` is used. It allows the OpSys to recover memory allocated to the program by the previously-issued `load_named_DLL` call. Refer to *Verix V Operating System Tools Reference Manual*, VPN - 23231.

For more information on shared libraries, refer to the *Verix eVo Volume III: Operating System Programming Tools Reference Manual*, VPN - DOC00303.

unload_DLL()

Allows the OpSys to recover memory allocated to the program by the previously-issued `load_named_DLL` call.

Prototype `int unload_DLL(int load_address);`

Parameters

<code>value</code>	The number to convert.
<code>load_address</code>	The address of buffer to store the string.

Once the `unload_DLL` call has been invoked, all virtual memory allocated for this DLL will be returned to the OpSys and available for further use. Similarly, all physical memory allocated for data storage will be returned. Any subsequent memory access to code or data will result in a program abort.

Return Values Successful operation is indicated by a return code of 0. If an error is detected in attempting to process this call, a result of `-EINVAL` is returned.

Diagnostic Logging

The OS needs to support a method for recording significant events detected by modules within the OS and by user-level applications and library routines.

dbprintf Method

All Trident platforms to date have supported a form of logging based on the `*DEBUG` environment variable. Within OpSys components, the system call `Uart_Printf` is used to record activity, especially when drivers or applications are invoked or when exceptions occur. For most production terminals these calls produce no visible effect, but if the variable `*DEBUG` has been set, then all of these messages will be serialized over the selected communications port. In a similar manner, applications can invoke the call `dbprintf` to record whatever activity is significant or useful for debugging. For more information, refer to *Verix V Operating System Tools Reference Manual*, VPN - 23231.

Memory-Based Logging

The `CONFIG.SYS` variable `*LOG` may be used to reserve an area in the file system for a circular, memory-based log specified in kilobytes. The allowable range for `*LOG` is any number between 50 and 32 x 1024. While it is not prohibited, in general you should not set both `*DEBUG` and `*LOG` at the same time.

When `SYSLOG` is active, any call to `Uart_Printf` (by supervisor-level modules) or to `dbprintf` (by user-level modules) is diverted to the system log. The diagnostic data for the call is copied into the ring buffer in the memory instead of being serialized over an external port. The same record-size limit (nominally eighty bytes maximum) apply. Each record in the log is time-stamped with a five-byte header that includes the day of month and time of day in milliseconds. Leading and trailing spaces are not stored.

System Call to Display Memory

A variation of the `dbprintf` system call provides a formatted display of memory. See `dbdump()` or refer to the *Verix eVo Volume III: Operating System Programming Tools Reference Manual*, VPN - DOC00303, for more information.

Log Reset

The `CONFIG.SYS` variable `*LOG` is read during startup. If it has changed since the previous startup, then the existing log (if any) is deleted, and a new log (if enabled) is created.

Configurable OpSys Logging

The `CONFIG.SYS` variable `*DEBUGO` is read during startup. If present, it may be used to enable additional logging. These options should not be used in production, since they will rapidly fill the log.

Format `*DEBUGO=abcd`, where the characters `a,b,c,d` each represent multiple bits.

First Postition	Activity
a=1	Scheduler: task resumption
a=2	Scheduler: <code>SVC_WAIT</code>
a=4	<code>wait_event</code>
a=8	Semaphore wait
Second Position	
b=1	Pipe manager: write
b=2	Pipe manager: read
Third Position	
c=1	Ethernet: write
c=2	Ethernet: input event
c=4	Ethernet: read
Fourth Position	
d=1	Keyboard: input event
d=2	Keyboard: read
d=4	Security: significant calls

For example, set `*DEBUGO=7003` to enable logging of common scheduler and keyboard activity.

dbdump()

This causes the memory to be “dumped” into the system log (if *LOG has been set) or to be serialized on the selected uart port (if *DEBUG has been set). Otherwise the call is useless.

Prototype `int dbdump(void *buf, int bufsize);`

The indicated memory is displayed sixteen bytes per line. Each byte is displayed in hexadecimal format (using a standard “%02X “ option). After displaying all sixteen bytes in this manner, each character is displayed “raw” with the assumption that it is Ascii data; non-Ascii bytes will be translated to ‘.’ characters.

Return Values The normal result is 0, indicating success. Failure occurs if the caller does not have access to the indicated block of memory.

Network Device Drivers

Network device drivers such as modem, radio, and Ethernet support interface for network administration. By default, the application opens and owns the interface. Using the same event bit for multiple related devices (such as, BCM WiFi) is not a limitation if each device is opened by a different task or thread. In the case, where the same task/thread must open multiple devices using the same event bit, the drivers will support the “soft event bit” mechanism where the event bit can be changed by applications per device.

These modems are connected to COM2 or COM3, which varies by terminal. Some devices are connected to the same COM port by a hardware multiplexor.

- POTS Banshee (Predator only)
- POTS Eisenhower (Predator only)
- POTS USB modem SiLabs (Predator and Trident)
- POTS Harley (Trident only)
- CDMA SierraWireless 3420 — V
- GPRS Siemens MC55i — VX 680
- GPRS Siemens MC55i — VX 680, VX 520 GPRS
- ISDN Stollman
- BLUT Bluetooth radio — VX 680

COM ports and communication devices which varies by terminal.

- COM1 UART external RS-232 port
- COM2 UART external PIN Pad (Predator only) port or internal radio connected
- COM3 internal modem connected or external USB modem
- COM6 USB UART

GPRS modems GSM 7.10

The GSM 7.10 multiplexor (mux) protocol supports multiple virtual channels over a single physical channel. When this protocol is implemented inside the OS device driver, the IP stack is allowed to own a virtual channel for PPP packets, another task to own a virtual channel for connection establishment and control, and another task to own a virtual channel for network link monitoring.

For backwards compatibility, the GSM mux defaults OFF. The application that opens the GPRS COM port must call `set_opn_blk` to enable the GSM mux protocol. Once this is done, two additional devices may be opened—`/dev/gmx1` and `/dev/gmx2`.

The `/dev/com2` now includes an additional option to struct `Opn_Blks` protocol, `P_gsm_mux`. When the application specifies this new option, the device driver sends commands to the GSM modem to enable the GSM mux protocol.

The `/dev/gmx1` and `/dev/gmx2` are opened after the application has enabled the mux protocol. These devices support the usual `open()`, `close()`, `read()`, `write()`, `status()`, and control functions for serial ports.

Example Task 1 opens `/dev/com2` then enables the mux protocol. It can hand off COM2 to the dialer/IP stack to get this port connected and in PPP mode. Task 1 can run task 2 which opens `/dev/gmx1` to get signal quality and connection status. Task 1 can run task 3 to monitor incoming Unsolicited Response Codes (URCs).

USB Ethernet USB Ethernet device driver presents two devices to application programmers. Each device supports `open()`, `close()`, `read()`, `write()`, and the event bit `EVT_ETH1`.

- `/DEV/ETH1` is for Ethernet data packets formatted as Ethernet 802.3 packets. This device is normally owned by the IP stack.
- `/DEV/ETHTRP1` is for Ethernet link status packets.

NOTE



All devices report events on the same event bit but every task/thread has its own event mask. When an incoming Ethernet packet arrives on `/dev/wln1`, the event bit is set for the task that opened the device. Another task with `/dev/trp1` open will not have its event bit set so it will not be activated.

Below is a format of network status packets:

```
struct eth_link_status {
    uint32_t status; // 0 link down, 1 link up
};
```

WiFi Device dependent code can determine which USB WiFi chip is installed by calling function `get_usb_device_bit()`. It currently returns a 1 in bit position 2 when the Partagas chip is connected. When the BCM4315 chip is present, it returns a 1 in bit position 10.

```
unsigned long get_usb_device_bits (void);
#define UDB_WIFI (1<<2)
#define UDB_BCMWIFI (1<<10)
```

USB WiFi Partagas

The Verix V Partagas device driver presents four devices to application programmers. Each device supports `open()`, `close()`, `read()`, and `write()`.

- `/DEV/WLN1` is for WiFi data packets formatted as Ethernet 802.3 packets. This device is normally owned by the IP stack.
- `/DEV/WLNCTL1` is for Conexant PIM4 command/response packets to manage the WiFi link layer. These packets are Ethernet packets with Ether Type 0xTBD.

- `/DEV/WLNTRP1` is for Conexant network status packets. These are formatted as Ethernet 802.3 packets with Ether Type = 0xTBD. If this device is not opened, the network status packets are delivered on `/DEV/WLN1`.
- `/DEV/WLNEAP1` is for Ethernet packets for EAPOL (Ether Type 0x888e) and RSN_PREAUTH (Ether Type 0x88c7). This is normally owned by the WPA supplicant. If this device is not opened, the EAPOL and RSN_PREAUTH packets are delivered on `/DEV/WLN1` (currently, only WPA-PSK is supported).

NOTE

All devices report events on the same event bit but every task/thread has its own event mask. When an incoming Ethernet packet arrives on `/dev/wln1`, the event bit is set for the task that opened the device. Another task with `/dev/wlnctl1` open will not have its event bit set so it will not return from `wait_event()`.

BCM4315 Device

The Verix V BCM4315 device driver presents four devices to application programmers. Each device supports `open()`, `close()`, `read()`, `write()`, and the event bit `EVT_WLN1` (default event but can be changed to any other unused event bit).

- `/DEV/WLN1` is for WiFi data packets formatted as Ethernet 802.3 packets. This device is normally owned by the IP stack.
 - `int read(int hdl, char *buf, int len)`
Returns 1 Ethernet packet, if any.
 - `int write(int hdl, char *buf, int len)`
Sends 1 Ethernet packet.
- `/DEV/WLNCTL1` is for Broadcom IOCTL command/response packets to manage the WiFi link layer. These packets are not Ethernet packets. They are formatted according in Broadcom's IOCTL format.
 - `int read(int hdl, char *buf, int len)`
Receives 1 Broadcom IOCTL response packet, if any.
 - `int write(int hdl, char *buf, int len)`
Sends 1 Broadcom IOCTL command packet. See Broadcom documentation for the available commands.
- `/DEV/WLNTRP1` is for Broadcom network status packets. These are formatted as Ethernet 802.3 packets with Ether Type = 0x886c. If this device is not opened, the network status packets are delivered on `/DEV/WLN1`.
 - `int read(int hdl, char *buf, int len)`
Returns 1 Broadcom network status packet, if any.
 - `int write(int hdl, char *buf, int len)`
Returns `EINVAL` because this is a read-only device.

- `/DEV/WLNEAP1` is for Ethernet packets for EAPOL (Ether Type 0x888e) and RSN_PREAUTH (Ether Type 0x88c7). This normally owned by the WPA/WAP2 supplicant. If this device is not opened, the EAPOL and RSN_PREAUTH packets are delivered on `/DEV/WLN1`.

NOTE

All devices report events on the same event bit but every task/thread has its own event mask. When an incoming Ethernet packet arrives on `/dev/wln1`, the event bit is set for the task that opened the device. Another task with `/dev/cmr1` open will not have its event bit set so it will not return from `wait_event()`.

Example of using these devices:

- Task 1 opens `/dev/wlnctl1`. Sends IOCTL commands to connect to an AP.
- Task 2 opens `/dev/wln1`. Pass packets to/from IP stack. Or assign control of this device to an IP stack.
- Task 3 opens `/dev/wlneap1`. Waits for EAP authentication requests and responds to them.
- Task 4 opens `/dev/wlntrp1`. Waits for network status changes and sends info other tasks about them. For example, it could trigger user events to other tasks or send messages via pipe to other tasks.

NOTE

VX 680 uses Broadcom BCM4329 chip.

3G

On VX 680 3G the Cinterion PHS8-P 3G (GSM/GPRS/UMTS/HSPA+) cellular radio is connected via USB 1.1 host port (12 Mbit/s maximum) to the Trident chip. The radio must be configured for USB composite mode (`AT^SDPORT=3`) for normal use. There is no hardware pass-through like the one used with the CDMA radio.

Device Ports

In composite mode, the radio has five virtual serial interfaces. One is reserved and another is dedicated to a proprietary protocol. This leaves three virtual serial interfaces that Cinterion names the Modem, Application, and NMEA interfaces. The serial interfaces are detailed below.

Verix Device Name	Cinterion name	Description
<code>/DEV/COM2</code>	Modem	Primary data interface for use by EOS and IP stack. Has AT command interface.
<code>/DEV/COM9</code>	Application	Control and status interface for use by EOS. Has AT command interface.
<code>/DEV/COM10</code>	NMEA	GPS data output for use by EOS. Does not have AT command interface. When GPS is turned on via AT commands on

the application interface, NMEA data (NMEA format) comes out of this interface.

Each interface appears to applications and libraries as Verix COM serial ports, it has the usual Verix COM port functions. The interfaces are virtual serial ports so some options such as baud rate do not change anything. The COM port functions are listed below:

```
int open (const char *id, int flags);
int write (int hdl, const char *buf, int len);
int read (int hdl, char *buf, int len);
int close (int hdl);
int set_opn_blk (int hdl, const struct Opn_Blkc *ob);
int set_serial_lines (int hdl, const char *mask);
int get_opn_blk (int hdl, struct Opn_Blkc *ob);
int get_port_status (int hdl, char *buf);
int set_radio_ctl (int hdl, const char *sigs);
int get_radio_sts (int hdl, char *sigs);
int set_event_bit (int hdl, long flag);
long get_event_bit (int hdl);
int reset_port_error (int hdl);
int set_signal_events (int hdl, char *evts);
int openaux(int devhdl);
int closeaux(int devhdl);
```

The following functions are not implemented for this device because they based on the GSM multiplexer protocol which is not used with this radio. The GSM multiplexer is not used because the USB protocol creates the three required virtual serial interfaces.

```
int set_gsm_powersave(int hdl, unsigned int power);
int set_gsm_break(int hdl);
```

Power save on this device is controlled by the OS device driver, not EOS. The AT+CFUN command does not control power save when the radio is connected via USB. The driver is responsible for putting the radio in power save mode using USB suspend when the system goes into deep sleep.

Device Presence

The following items are added to `SVC.H` to define a new module identifier values for the Cinterion PHS8-P radio.

```
#define MID_PHS8P (89) /* Cinterion PHS8-P radio with GPS */
#define MID_PHS8P_NOGPS (90) /* Cinterion PHS8-P radio without GPS */
```

When the Cinterion PHS8-P radio and GPS hardware are present, the following existing function returns `MID_PHS8P`. When the Cinterion PHS8-P radio is present but GPS hardware is not, the following function returns `MID_PHS8P_NOGPS`.

```
SVC_INFO_DEV_TYPE(INFO_PRES_GPRS);
```

The NOGPS option is included in case a hardware variant is built without GPS antenna or other hardware required for GPS.

On VX 520 3G, the GPS data port (COM10) cannot be used without a license — A GPS FE license is required in testing the GPS hardware. However, the GPS data port can still be used (even without a license), provided that the terminal serial number has not been set — it is assumed that the serial number is injected after the GPS hardware is tested.

Network Device Status

Below are the device status variables used:

- `openaux()`
- `closeaux()`

openaux()

Returns a Verix device handle for a limited device to be used for link monitoring.

On BT, this function opens an auxiliary device that only has access to the status functions. This allows a second program or task to monitor the connection.

Prototype `int openaux(int devhdl);`

Parameters

`devhdl` Verix device handle returned from calling `open()`.
The handle of the primary network device. The caller must own `devhdl`.

Return Values

Success: `> 0`, the device handle.
Failure: `-1` with `errno` set to `EBUSY` if the caller does not own the device—possibly owned by another task.

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

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

closeaux()

Closes the device status specified by the device handle. The caller must own the handle.

On BT, this function closes the auxiliary device opened by `openaux()`.

Prototype `int closeaux(int devhdl);`

Parameters

`devhdl` Verix device handle.

Return Values

Success: `0`
Failure: `-1` with `errno` set to `EBUSY` if the caller does not own the device.

TCP/IP Network Support

Predator terminal supports an OS-downloadable driver for TCP/IP support. The apps/libs have a direct socket API access similar to the system lib (VFSDK). The BSD socket interface plus the OS sends `EVT_SOKT` when incoming data arrives on a socket. This allows tasks to use a single event loop for devices, pipes, and sockets.

The TCP/IP data interface is called by traditional OS driver. It resides in the Verix eVo file system and can support more than one link layer (e.g. Ethernet & PPP). For more information on API please see *Verix eVo Network ERS*, VDN - 28783.

TCP/IP support in Trident is provided by EOS and is described in the EOS library documentation. See *Verix eVo Volume II Operating System and Communication Programmers Guide*, VPN - DOC00302 for more information.

TCP/IP Module

The TCP/IP stack on VX 680 3G supports simultaneous socket connections over multiple networks. For example, the TCP/IP stack supports socket connection A over the BT network, while simultaneously supporting a socket connection B over the Wi-Fi network.

The OS provides additional APIs to force a socket connection onto a specific network interface. The default number of socket buffers can be increased to accommodate simultaneous network operation.

TCP Window can be set to a larger value to accommodate increased latencies due to running simultaneous network interfaces. The OS supports simultaneous TCP connections over separate network interfaces both of which require routing.

The TCP/IP stack increases the default number of sockets and the maximum number of sockets that can be specified using `*SOCKET`.

Socket Events

`EVT_SOKT` is an event that can be waited upon using the function `wait_event()`. This is generated on all socket events.

Treck Socket Events

Following are the available socket events:

- `TM_CB_CONNECT_COMPLT` - 0x0001
- `TM_CB_ACCEPT` - 0x0002
- `TM_CB_RECV` - 0x0004
- `TM_CB_RECV_OOB` - 0x0008
- `TM_CB_SEND_COMPLT` - 0x0010
- `TM_CB_REMOTE_CLOSE` - 0x0020
- `TM_CB_SOCKET_ERROR` - 0x0040
- `TM_CB_RESET` - 0x0080
- `TM_CB_CLOSE_COMPLT` - 0x0100

- TM_CB_WRITE_READY - 0x0200

NOTE

If your applications are required to be ported to Linux with standard sockets interface, it is best to use the standard socket interface.

Invoking the OS Socket Interface

The OS socket interface must be initialized by the Verix eVo Communication Engine (CommEngine) "N:VXCE.OUT" so that appropriate network connection(s) can be made. It is likely, however, that at least some portion of the socket manager will need to be initialized while the OS itself is being brought up.

The CommEngine assigns network capable devices to the IP stack. The IP stack does not automatically grab devices because some devices are capable of operating in network and non-network configurations. For example, dial modems can be used for PPP network connections and can be used at other times for non-network connections such as VeriCentre download.

Bypassing the OS Socket Interface

For compatibility, Verix eVo OS must be disabled and allow user applications to open network communication ports directly.

IP Stack Control and Configuration

API functions are provided by the IP stack to add/delete/configure/control network interfaces at run-time. Network devices cannot be dedicated to the IP stack.

NOTE

Adding multiple network interfaces to the TCP/IP stack is possible. There is no architectural limitation, except for mux port or radio interference issue.

For example, a modem may be used by the stack sometimes and used directly by an application for legacy protocols at other times.

- `int net_if_add(net_if_t *net_if, long event_mask);`

Example

Adds network interface to the IP stack and returns a handle for the network interface device. The caller opens the device as usual then passes the device handle to the stack in `net_if` structure. From this point on the stack reads/writes on the device. The caller may still call control and status function using the device handle. The `event_mask` parameter specifies the event bit the stack uses to report network device status changes.

- `int net_if_del(int hdl);`

Deletes network interface from IP stack. Further `net_if_*` calls using this handle value are not valid.

- `int net_if_up(int hdl);`

Changes network interface to the up state.

- `int net_if_down(int hdl);`

Changes network interface to the down state.

- `int net_if_get_status(int hdl);`

Returns the current network interface up/down status.

- `int net_if_info(int hdl, net_if_t *net_if);`

Returns the current network interface device parameters in `net_if`. Handle is the value returned by `net_if_add`.

- `int net_if_open_sockets(int hdl);`

Example

Returns the number of open sockets in the network interface associated with handle.

Dual SIM Design

A variant of VX 520 GPRS supports dual SIM consisting of a fixed primary SIM socket and a removable secondary SIM socket. A SIM select pin (GPB12) is an output from SoC to the dual SIM select hardware.

PIN	Description
0/low	Select SIM1 (primary SIM, power up default)
1/high	Select SIM2 (backup SIM)

SIM protection is handled by the radio for all removable SIMs. HW MUX is necessary to switch two SIM slots that connects to the radio.

VX 680 3G supports dual SIM that are both removable. `set_SIM_slot()` and `get_SIM_slot()` are not backwards compatible with `SIM_SELECT()` API.

SIM_SELECT()

Selects which SIM card to use. When CIB detects the presence of dual SIM on OS boot, GPB12 is initialized to an output pin and set to zero (0) to select the primary SIM by default.

The application may call `SIM_SELECT(int simindex)` while the terminal is up and running, to dynamically select between the two SIM cards through hardware MUX depending on the value of `simindex`.

NOTE



Call proper shutdown of the radio before using this function. Radio power on and off must be handled properly.

Prototype `int SIM_SELECT(int simindex);`

Parameters

`simindex` The index of SIM slot to be selected.

Return Values

Success: 0

Failure: -1 with `errno` set to `EACCESS` (radio is powered on), `ENODEV` (dual SIM hardware is not present), or `EINVAL` (invalid parameter).

Example

```
// Switch from SIM1 to SIM2 then back to SIM1
int com2;

SIM_SELECT(0);
com2 = open(DEV_COM2, 0);
...
write(com2, "AT^SMSO\r", 8); // Graceful radio shutdown
// wait for ^SHUTDOWN
close(com2);

SIM_SELECT(1); // Switch to backup SIM

com2 = open(DEV_COM2, 0);
...
write(com2, "AT^SMSO\r", 8); // Graceful radio shutdown
// wait for ^SHUTDOWN
close(com2);

SIM_SELECT(0); // Switch to primary SIM
```

set_SIM_slot()

On VX 680 3G, this API Selects the specified SIM card slot. In order to do this, the API powers off the radio and powers it back on before returning. It is up to the caller to re-initialize the 3G modem again. The application ensures that the radio and network stack are in a valid state when powering off of the radio. This API does not detect problems like “no SIM card” in the selected slot. The SIM slot selected will not be preserved across power cycle. SIM0 is always selected on power up.

Prototype `int set_SIM_slot(int hdl, int slotNumber);`

Parameters

hdl	Device Handle
slotNumber	0 to select primary SIM slot. 1 to select secondary SIM slot.

Return Values

0	Success
1	OS Error. Errno is EBADF if caller does not own the GPRS device. Errno is EINVAL if slotNumber is not 0 or 1.

get_SIM_slot()

On VX 680 3G terminals, this API Returns the index of the currently selected SIM slot.

Prototype `int get_SIM_slot(int hdl);`

Parameters

hdl	Device Handle
-----	---------------

Return Values

0	Primary SIM slot is selected.
1	Secondary SIM slot is currently selected.

Feature Enablement (FE)

Some terminal features require a license key to unlock the feature.

License Database

The OS maintains a license database to keep track of which features should be enabled. The OS performs no direct enforcement, but rather provides the following APIs for applications or device drivers to determine when a feature is enabled.

- `feature_license_total()`
- `feature_license_expiration()`
- `feature_license_is_enabled()`
- `feature_license_get_tag()`
- `feature_license_get_detail()`

GPS

The Cinterion PH8S-P 3G radio GPS feature is controlled entirely by AT commands so the OS device driver cannot block this feature. EOS can block this feature unless applications can bypass EOS and directly access the radio with AT commands. There is no easy way to add FE for this feature if applications do not use EOS and directly control the radio.

Bypassing EOS is possible if the hardware includes a switch between the GPS antenna input that can be controlled via software, the OS device driver can connect/disconnect the GPS antenna based on FE.

HSPA+

The Cinterion PHS8-P 3G radio HSPA+ feature is controlled entirely by AT commands so the OS device driver cannot block this feature. EOS can block this feature unless applications can bypass EOS and directly access the radio with AT commands. There is no easy way to add FE for this feature if applications do not use EOS and directly control the radio.

The Cinterion radio provides a method to set the maximum downlink PHY rate to either 7.2 MBPS or 14.4 MBPS and to set the maximum uplink PHY rate to either 2.0 MBPS or 5.76 MBPS. The PHY rates of 14.4 and 5.76 are not exclusive to HSPA+ however the Cinterion radio does provide these throughput controls. Whatever software module implements this, it should provide a configuration option of high throughput or low throughput, and should not separately control the throughput for uplink and for downlink.

There is a negative impact on power save when running transactions at 7.2 MBPS downlink and 2.0 MBPS uplink PHY rates as these transactions may take twice as long to complete.

feature_license_total()

Returns the number of feature licenses installed.

Prototype `int feature_license_total(void);`

Return Values

<code>>= 0</code>	Number of licenses installed.
<code>< 0</code>	Error

feature_license_expiration()

Returns license expiration status details.

Prototype `int feature_license_expiration(featureLicenseTag* pTag, int* pUnit);`

Parameters

[in] pTag	A pointer to a featureLicenseTag structure that contains the major and minor tag values for which status is returned.
[out] pUnit	Indicates whether the return value indicates the number of days or uses remaining. If the return value is non-positive, this variable is not used. For example : 1 Day remaining, 2 Uses remaining.

Return Values

<code>> 0</code>	Number of days or uses until the license expires.
<code>= 0</code>	License never expires
<code>= -1</code>	Bad parameter
<code>= -2</code>	License is not installed.
<code>= -3</code>	License is installed, but is expired.
<code>= -4</code>	License is installed, but not yet active.
<code>= -5</code>	License is installed and will be enabled on demand.

feature_license_is_enabled()

Returns boolean indicating if license is currently enabled. Applications can use this function for enforcement. This is the only function that triggers an On Demand start, and the only function that decrements COUNT.

Prototype `int feature_license_is_enabled(featureLicenseTag* pTag);`

Parameters

[in] pTag	A pointer to a <code>featureLicenseTag</code> structure that contains the major and minor tag values for which status is returned.
-----------	--

Return Values

= 0	Feature is disabled, license is missing, or there is an error.
= 1	Feature is enabled.

feature_license_get_tag()

Returns installed license tags based on index.

Prototype `int feature_license_get_tag(int index, featureLicenseTag* pTag);`

Parameters

[in] index	A value from 0 to license total - 1. <code>feature_license_total()</code> returns the number of licenses installed.
[out] pTag	A pointer to a <code>featureLicenseTag</code> structure that will be populated with the major/minor values for the feature license index.

Return Values

= 0	License found
= -1	Bad parameter
= -2	License is not installed.

feature_license_get_detail()

Returns the details of a specific feature license Name=Value pair.

Prototype

```
int feature_license_get_detail(featureLicenseTag* pTag, char*
licenseFieldName, char* buff, int buffLength);
```

Parameters

[in] pTag	A pointer to a <code>featureLicenseTag</code> structure that contains the major and minor tag values for which detail is returned.
[in] licenseFieldName	A pointer to a null terminated string that specifies the feature license field to return. The following fields are supported: ISSUER, CUSTOMER, DESCRIPTION, START, END, and COUNT. Note that a single license supports either START/END or COUNT, but not both.
[out] buff	A pointer to a null terminated string that contains the value associated with the specified name.
[in] buffLength	The size (in characters) of <code>buff</code> . The string copied to <code>buff</code> is truncated if it is longer than <code>buffLength</code> .

NOTE



If an on demand license has not yet been enabled, `START` returns "On demand" and `END` returns the activation period in days. If a license never expires, `END` returns "Never expires". In all other cases, `START` and `END` return dates formatted as YYYYMMDD.

Return Values

= 0	License found
= -1	Bad parameter
= -2	License is not installed.



Download Operations

The three download methods for Verix eVo-based terminals are *direct*, *remote*, and *back-to-back*. This chapter provides download precautions and describes procedures for downloading applications and files to Verix eVo-based terminals. Some operations are not explained in detail; there may be references to pertinent documentation.

To successfully transfer files into a Verix eVo-based terminal, users can:

- Perform direct downloads using the Direct Download utility (DDL): DDL is part of the VVDTK. Direct downloads use a cable between the development PC and the terminal. See DDL- Direct Download Utility in the *Verix eVo Volume III Operating System Programming Tools Reference Manual*, VPN - DOC00303 for more information.
- Perform remote downloads using VeriCentre, a VeriFone remote download tool. Remote downloads use the public telecommunications network (phone line), or the TCP/IP (available on some terminals). VeriCentre is described in the online manuals on the VeriCentre CD-ROM.

To support ECSecure, the OS includes terminal serial number from the MIB in the download sign-on packet. See *ECSecure*, VDN - 23773, for details.



CAUTION

Not all devices' keypads are the same. Caution should be exercised in creating VTM and GID passwords. Valid characters for the passwords are those that can only be entered from the device keypad. The allowable characters include:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

. * , ' " - + # ! . : ; @ = & / \ % \$ _

The `CONFIG.SYS` variables `*SMPW` and `*PW` should only be set using characters that are supported on the terminal keypad, otherwise, access to the VTM or individual GIDs are blocked. When this happens, the terminal must be sent to an authorized VFI Repair Center to clear the terminal's memory and reset the default password. If the password is set during a partial download, any data still resident in the terminal will be lost.

NOTE



On the VX 820 PIN pad, VeriCentre-initiated downloads are not supported since they require the use of a modem. However, modem tests can be performed on the VX 820 DUET base station because it has a built-in modem. Modem tests can be performed if the VX 820 unit is connected to the base.

- Perform back-to-back downloads between two terminals: Back-to-back downloads initiate a transfer of the contents of one terminal to a similar terminal. The *Gold* (sending) terminal and the *Target* (receiving) terminal are connected by a null modem RS-232 cable, and applications and files download between the terminals. Note that both terminals must have the same model number in the MIB. The MIB model number is checked and if not matched the download fails.

A downloadable OS can be downloaded into the I: file system through ways enumerated above. During authentication, the newly-downloaded OS burned in F: drive, replacing the old OS.

OS Download

OSDL is made available as a shared library supporting all VeriCentre download primitives. The shared system library SYS.LIB implements two additional functions to support downloading over a network.

- `SVC_ZONTALK_NET()` is analogous to the older call `SVC_ZONTALK()`. It allows the caller to invoke a TCP/IP-based download, with or without SSL. The routine establishes the network connection based on parameters defined in the caller's `CONFIG.SYS` environment file.
- `download_net()` is analogous to the older call `download()`. The caller is responsible for establishing the network connection prior to calling `download_net`.

The use of SSL requires considerable program memory. Network-based downloads without SSL can occur with no heap at all and with a stack size of 8 KB or less. You can increase the download speed by letting the download code use the caller's heap for its download buffer—set `*ZH` to the maximum packet size (up to 4KB) and ensure that your heap has a single block of at least 1000 bytes larger than your `*ZH` setting. If SSL is selected, however, you may need a heap size of 100 KB or more, as well as a stack size of about 20 KB.

Optional Functions

In addition to the above functions, below are several other functions that can be used for customizing the download. However, these optional functions are usually not needed, and the two functions discussed above would suffice.

- `fpNotify()`
- `shRegisterNotifyCB()`
- `shSetParam()`
- `shRegisterExtendedProcessing()`
- `fpshPostProcess()`
- `fpshPreProcess()`

SVC_ZONTALK_NET()

Establishes TCP/IP or SSL connection to perform the VeriCentre download.

NOTE



The SSL connection established on this API does not include SSL server certificate validation and provides no support for SSL client certificate verification.

Prototype

```
int SVC_ZONTALK_NET(unsigned char type);
```

Parameters

type Download type (full/partial). Valid values are F, P, p, R, and r.

Return Value

DLE_MEANINGFUL (99)	Download is successful and meaningful. This is received when a part or full data is downloaded into the GID selected for the download.
DLE_MEANINGLESS (100)	Download failed—terminal ID not available, application is not configured, or download type does not match with the setting on the VC.
DLE_USER_CANCEL (-1)	User cancelled the download.
DLE_TCPIP_SSL_ERROR (-2)	Error in TCP/IP or SSL connection.
DLE_BAD_TX_COM (-3)	Received too many NAKs.
DLE_HOST_CANCEL (-4)	Remote host sent 'U' packet or aborted the download.
DLE_NO_SSL_SUPPORT (-5)	Cannot link SSL library.
DLE_BAD_WRITE (-6)	Write to file failed.
DLE_HOST_TO (-7)	Timed out waiting for host.
DLE_LOST_CARRIER (-8)	Lost carrier from modem.
DLE_BAD_RX_COM (-9)	Sent too many NAKs.
DLE_BAD_TIMER (-10)	OS <code>set_timer()</code> failed.
DLE_BAD_COMPORT (-11)	OS <code>write()</code> to COMx failed.
DLE_ENQ_TO (-14)	Timed out waiting for ENQ.

SVC_ZONTALK_NET() looks for the following environment variables.

*ZA	Application name
*ZB	Maximum data-packet size (range 240 to 1014, default 1000)
*ZH	Maximum data-packet size if heap space is available (range 1024 to 4096)
*ZT	Terminal ID

*ZN

IP address and port number of host (VeriCentre or SSL Server)

Format: <IP dotted-quad address>:<Port Number>

Example: 10.64.30.228:5001

Alternate formats are also accepted.

- First, the port number (and the colon that precedes it) are optional. If not specified, a default value will be used based on the type of download—if *ZSSL=1, the default port is 443. Otherwise, the default port is 8013.
- Secondly, the network address itself may be specified by using its domain name instead of its “dotted quad” address. Again, the port may be specified (with colon followed by the numeric port number) or may be omitted if the above default value is appropriate.

*ZSSL

Sets the type of connection. 1 for SSL connection and 0 for plain TCP/IP connection

download_net()

Requires application to be connected to a TCP/IP or SSL server prior to calling `download_net()` enabling developers to create/develop applications, which require more robust TCP or SSL connections and/or SSL options to fit their software needs.

Prototype

```
int download_net(int sock, int *SSLstruc, const void *type);
```

Parameters

sock	Socket handle returned by <code>socket()</code> API in the application. This is used for plain TCP/IP download and is not needed for SSL download.
SSLstruc	SSL object return by <code>SSL_new()</code> API in the application. If value is not NULL, sock handle value is ignored and OSDL will automatically conduct an SSL download. If value is NULL, OSDL will use sock handle and automatically conduct a plain TCP/IP.
type	Download type (full/partial). Valid values are F, P, p, R, and r.

Return Value

DLE_MEANINGFUL (99)	Download is successful and meaningful. This return value is received when a part or full data is downloaded into the GID selected for the download.
DLE_MEANINGLESS (100)	Download failed—terminal ID not available, application is not configured, or download type does not match with the setting on the VC.
DLE_USER_CANCEL (-1)	User cancelled the download.
DLE_TCPIP_SSL_ERROR (-2)	Error in TCP/IP or SSL connection.
DLE_BAD_TX_COM (-3)	Received too many NAKs.
DLE_HOST_CANCEL (-4)	Remote host sent 'U' packet or aborted the download.
DLE_NO_SSL_SUPPORT (-5)	Cannot link SSL library.
DLE_BAD_WRITE (-6)	Write to file failed.
DLE_HOST_TO (-7)	Timed out waiting for host.
DLE_LOST_CARRIER (-8)	Lost carrier from modem.
DLE_BAD_RX_COM (-9)	Sent too many NAKs.
DLE_BAD_TIMER (-10)	OS <code>set_timer()</code> failed.
DLE_BAD_COMPORT (-11)	OS <code>write()</code> to COMx failed.
DLE_ENQ_TO (-14)	Timed out waiting for ENQ.

`SVC_ZONTALK_NET()` looks for the following environment variables.

*ZA	Application name
-----	------------------

*ZB	Maximum data-packet size (range 240 to 1014, default 1000)
*ZH	Maximum data-packet size if heap space is available (range 1024 to 4096)
*ZT	Terminal ID

fpNotify()

A user-provided call-back function that may be invoked as needed during processing of SVC_ZONTALK_NET or download_net. To enable these callbacks to occur, the user must register this function using `shRegisterNotifyCB()` prior to calling SVC_ZONTALK_NET or download_net.

Prototype `short (*fpNotify) (short shActionId, void *vActionValue);`

To Perform	shActionId	vActionValue	Suggested Usage of vActionValue in application
To display messages sent from the download server	UPDATE_DISPLAY	Null-terminated string contains the progress of the download. For example, "*** _____"	Application displays the vActionValue
To process CANCEL key during download	VERIFY_CANCEL	NULL	The call-back function should return less than zero if download needs to be cancelled. Download will continue for any other value
To display a filename that is being downloaded	NEW_FILE	Null-terminated string contains the filename which is to be downloaded	Application displays the filename
To display the drive where the following files are being downloaded	SWITCH_DRIVE	Null-terminated string contains the new drive. Example: "F"	Application displays the new drive
To display the Group ID where the following files are being downloaded	SWITCH_GROUP	Null-terminated string contains the new group ID. Example: "15"	Application displays the new group ID.
To update the terminal clock with the time received from the download server	TIME_CHANGE	Null-terminated string contains the date and time. Format: "YYYYMMDDhhmmss"	Application can update the clock
To handle erroneous packet from the download server	INVALID_PACKET	Null-terminated string contains the packet ID received	Based on the packet ID, the application can decide to abort or to continue the download. To abort, return a value less than zero. Download will continue for any other value.

shRegisterNotifyCB()

An application interface call that can register a call-back function for getting the download progress. The first parameter 'Notify' function is called by the library as the download progresses. Refer to [fpNotify\(\)](#) for more details.

[shRegisterNotifyCB\(\)](#) function is not called if the application does not require download progress notification. In such a case, the download continues without notifying the progress. The user can turn off any or all of the notifications by calling this function again with appropriate values in *shFilter*.

Prototype

```
short shRegisterNotifyCb (fpNotify Notify, short shFilter);
```

Parameters

<i>Notify</i>	A function pointer with two parameters, <i>shActionId</i> and <i>vActionValue</i> .
<i>shFilter</i>	Can be used to filter the library notifications. The application can bit wise OR any of the actions listed in <i>fpNotify()</i> . Based on the filter, the library calls the <i>fpNotify()</i> function.

shSetParam()

An application interface that allows the user to set several timeout values that apply during processing of SVC_ZONTALK_NET or download_net.

Prototype

```
short shSetParam (unsigned short shParamID, void *ParamValue);
```

shParamID	Data Type of the ParamValue	Default Value	Usage of ParamValue
IPDL_TCPTIMEOUT	unsigned long	2950 ms	<p>The library waits for the time specified in (time-out value) the parameter before retrying packets other than ENQ.</p> <p>Note: If specified value is less than 1000 milliseconds, the default will be used.</p>
IPDL_ENQTIMEOUT	unsigned long	30000 ms	<p>This is the time to wait from the call of</p> <p>SVC_ZONTALK_NET() or download_net() till IPDL receives the first ENQ packet from the download host.</p> <p>Note: If specified value is less than 1000 milliseconds, the default will be used.</p>

Return Values

Success: 0, the parameter was successfully set.

Failure: -1, and if ParamValue is NULL, caller's errno is set to DLE_INVALID_PARAM. If ParamID is not one of the above supported parameters, caller's errno is set to DLE_NOT_SUPPORTED.

shRegisterExtendedProcessing()

An application interface call that can register two call-back functions that will be called during the download operation. If the user does not want to perform extended processing, the user does not need to provide the two extended operations, nor does the user need to register them with this call.

Refer to [fpshPreProcess\(\)](#) and [fpshPostProcess\(\)](#) for more details

Prototype

```
short shRegisterExtendedProcessing  
(fpshPreProcess PreProc, fpshPostProcess PostProc);
```

Parameters

PreProcess	A function pointer with four parameters— vDataReceived, shReceivedLen, vDataToProcess, and pshDataLength.
PostProcess	A function pointer with four parameters—vData, shDataLen, vPreProcessedData, and pshProcessedDataLength.

fpshPostProcess()

A user-provided function invoked by the download library after receiving the packet from the download server. The library internally creates a buffer of maximum allowable size (based on *ZB or *ZH settings) and passes it as vDataToProcess. The application has to ensure that this buffer size is not exceeded while copying post processed data. This function should return DLE_CONTINUE_PROCESS to proceed with the download. If the application returns an unexpected return value, download continues.

Prototype

```
short (*fpshPostProcess)
(void *vDataReceived, short shReceivedLen, void *vDataToProcess,short *pshDataLength);
```

Parameters

vDataReceived	Pointer to the buffer containing the data received from VeriCentre.
shReceivedLen	Size of vDataReceived in bytes.
vDataToProcess	Pointer to the buffer containing the post processed data.
shDataLength	Pointer containing the size of the data.

Return Values

DLE_ABORT_DOWNLOAD (103)	User aborts the download.
DLE_CONTINUE_PROCESS (98)	Continues the download.
DLE_CONTINUE_RECV_RESET_TIMER (96)	Resets the InterCharTimeout.

fpshPreProcess()

A user-provided function invoked by the download library before sending the packet to the download server. The library internally creates a buffer of maximum allowable size (based on *ZB or *ZH settings) and passes this buffer as vPreProcessedData. The application has to ensure that this buffer size is not exceeded while copying preprocessed data. This function should return DLE_CONTINUE_PROCESS to proceed with the download. The library aborts download for other values.

Prototype

```
short (*fpshPreProcess)
(void *vData, short shDataLen, void *vPreProcessedData, short *pshProcessedDataLength);
```

Parameters

vData	Pointer to the buffer containing the data to be sent to VeriCentre.
shDataLen	Size of vData in bytes.
vPreProcessedData	Pointer to the buffer containing the pre-processed data.
pshProcessedDataLength	Pointer containing the size of the data.

Return Values

DLE_ABORT_DOWNLOAD (103)	User aborts the download.
DLE_CONTINUE_PROCESS (98)	Continues the download.

OS Download Precautions

Downloading an operating system into a Verix eVo-based terminal should be done with caution. Because the very files that control the download are being replaced, there are risks associated with the process.

Failure to correctly execute the download process or allow it to complete can leave the terminal non-functional and only an authorized VeriFone repair facility can reset the terminal. Use the following recommendations to reduce these risks and facilitate a successful operating system download.

- Download all operating system files into the terminal's memory file space.

Note: Loading into memory is the default.

- Do not interrupt terminal power during the entire operating system download process. This includes blackouts, switching off power, or unplugging the terminal.

- Although the terminal display may appear to be frozen for several minutes after the file download and authentication phase, do not interrupt the process.

Downloading operating system files into the flash file system is possible and does not necessarily produce error messages. However, this process may lead to an incomplete upgrade and possibly, a disabled terminal.

Interrupting the power during any part of the download process can lead to a disabled terminal. With this in mind, if the terminal is being used in an area affected by frequent power blackouts or brownouts, consider using an uninterruptible power supply (UPS).

This pause is normal and occurs while performing significant memory management tasks. Note too, that the terminal reboots several times during the download. The operating system download process is not complete and the terminal should not be powered down until "DOWNLOAD NEEDED INVALID *GO PARAMETER" displays.

For Trident terminals, OS download interruption will not harm the terminal. After the file download, the OS is burned into F: drive and is authenticated. An application can complete a partial download of an OS update then shutdown the network and perform an SVC_RESTART to allow the OS to authenticate and burn automatically.

- Download operating system files separately from other applications.
- Always download the entire operating system.

While the file structure of the terminal does accommodate multiple application downloads, the unique memory management features performed during an OS download could interfere with other files.

The segmented design of OS files does allow modular or partial replacement.

This should be avoided as a partially replaced module is not compatible with existing components.

The terminal will malfunction and may become disabled requiring service. A download of the entire operating system prevents any potential module compatibility problems.

Note: This only affects Predator OS.

These preventative measures ensure a properly functioning terminal at the completion of the operating system download process.

Preparing a Terminal to Accept Downloads

Unless the download is being performed under application control, you must set up the terminal to accept an incoming download.

To Prepare the Terminal to Accept Downloads (VTM):

- 1 Power up (or power cycle) the terminal. If the terminal displays **DOWNLOAD NEEDED**, it is ready to accept a download. If an application starts or an error message appears, restart the terminal either by entering VTM and selecting **RESTART** or by power cycling.
 - a Press 7 and Enter keys during the first screen (copyright notice) display.

NOTE



You can enter VTM anytime by pressing 7 and Enter keys but the terminal does not accept any downloads if an application is currently running. If an application is loaded, you must enter VTM before the application launches.

- 2 Select **DOWNLOAD** from the menu. Enter the file group where the files should be stored (see [Verix eVo Support for File Groups](#)).

- 3 For Predator, Select a `FULL` or `PARTIAL` download. When using the `ORIGINAL` UI, a full download erases all existing RAM files; when using the `ENHANCED` UI, a full download erases all existing RAM and flash files. A partial download preserves existing files except those with the same name as the downloaded file. These files are overwritten instead. Trident terminals delete all files for full download.
- 4 Select the download source: `COM1`, `COM2`, `MODEM`, or `WIRELESS`. Direct and back-to-back downloads normally use `COM1`.
- 5 The terminal displays `DOWNLOADING NOW`, (or `UNIT RECEIVE MODE` in the `ENHANCED` UI) indicating that it is ready to receive the download.

In most cases, either the terminal or the downloading device (DDL on a PC, download host, or Gold terminal) starts first. For example, if DDL is already running, the download immediately begins.

During the download, asterisks appear on the terminal display. Each asterisk indicates that approximately 10% of the download is complete.

Software developers must usually perform many frequent downloads. An automatic download feature is provided for this purpose. If the `CONFIG.SYS` variable `*SMDL` (page 180) is set to 1, the terminal checks for incoming downloads on restart.

NOTE



`*SMDL` works for direct *full* downloads through `COM1` at baud rates of 19200 or 115200, and through USB.

While the terminal can stay on the `DOWNLOADING` screen for an indefinite amount of time, you can start DDL then restart the terminal. This procedure detects and performs the download with no further intervention.

CAUTION



When this feature is enabled anyone can download files *without* using passwords. Therefore, *never* set `*SMDL` in deployed terminals.

NOTE



You can interrupt direct-download transfers at any time by using the cancel key on the terminal or `Ctrl-C` from the host PC.

The `DOWNLOAD FAILED` error screen returns a status code of `-1` to indicate the download failed. Direct downloads always return the same error codes (and success codes) as modem downloads.

File Name Extensions and GIDs

By default, downloaded files are stored in Group 1 of the memory file system. Different groups can be specified on terminal download setup, however, there is another method that is more convenient and flexible.

If a file named `SETGROUP.n` (where *n* is a number from 1 through 15) is downloaded, the terminal considers *n* an instruction to store subsequent files in Group *n*. This remains in effect until the end of the download or another `SETGROUP.n` file is received. `SETGROUP` files must have a length of zero to be interpreted in this special way (if it contains data, it is stored as an ordinary file). The normal group access rules apply.

This means that if the initial group set in the terminal download menu is Group 1, you can change to any other group during the download, but from any other initial group, you can change only to Group 15 (the *public* group).

In most cases, the terminal should remain set to Group 1 (default) and `SETGROUP` files should only be used to control the download. For this to work, you must know the Group 1 password.

Use a similar mechanism to download flash files. Download a zero-length file named `SETDRIVE.F` to direct the terminal to store the following files in drive *F*: (the flash file system). Use `SETDRIVE.I` to revert to the memory file system.

The DDL tool recognizes `SETGROUP.n` and `SETDRIVE.x` as special cases, and does not require that actual files with these names be stored on the PC. Other download tools may require you to create actual files.

NOTE



These files *must* not contain any data (0K in size).

Error Messages

Table 133 provides explanations of error messages that appear on the host screen.

Table 133 Error Messages

Error Message	Description
Out of memory	Attempt to allocate dynamic storage failed.
Unknown option '<option>'	The given option is not recognized.
Unmatched quote (<file>, <line number>)	Unbalanced quotation marks were found while reading arguments. If this occurred in an argument file, the location is shown.
Too many arguments on line (<file>, <line number>)	A maximum of 32 arguments can be specified on each line of an argument file. The location is returned.
Line too long(<file>, <line number>)	A line in an argument file exceeds the maximum length, which is approximately 300 characters. (Note that configuration settings are limited by the terminal to 256 characters.)
Missing argument for <name> option	The indicated option requires an argument, but none is present.

Table 133 **Error Messages** (continued)

Error Message	Description
Argument files nested too deeply	Argument files specified in an <code>-f</code> option cannot be nested more than two deep.
Insufficient FLASH	Coalescing flash may free enough memory to clear the problem. If not, files must be deleted. Files can be deleted in VTM or an application can provide a way to selectively delete files.
Insufficient RAM	Some files must be deleted. Files can be deleted in VTM or an application can provide a way to selectively delete files.
Invalid port number '<number>'	The given COM port number is either not numeric or out of range.
Invalid baud rate '<rate>'	The given baud rate is not supported (or not a valid number).
Invalid timeout '<value>'	The given time-out value is either not numeric or negative.
Invalid clock offset '<value>'	The given clock setting offset is not in the acceptable range of <code>-23</code> to <code>+23</code> hr.
File <name> not found	The given file does not exist.
Can't open <file>	The given file could not be opened.
file <name> not readable	The given file could not be accessed. (It might be a directory instead of a file.)
Download of <file> failed - <reason>	Attempt to download indicated file unsuccessful. Communications-related problems are usually the reason for this error message (for example, time-out).
Setting <variable> failed - <reason>	Attempt to set indicated configuration variable unsuccessful. Communications-related problems are usually the reason for this error message (for example, time-out).
Communication open failed - <reason>	Either the communication port could not be opened or an error in the initial sign-on message exchange occurred.
Clock set failed - <reason>	Attempt to set terminal clock unsuccessful.
Password set failed - <reason>	Attempt to set terminal password unsuccessful.
User Break	Download interrupted by Ctrl+C or Ctrl+Break from keyboard.
NO *ZTCP VARIABLE	The *ZTCP variable is not defined.
NOT FOUND	The *ZTCP variable does not exist
NOT AUTHENTIC	The *ZTCP variable is not authenticated
RUN FAILURE	The *ZTCP variable exists and is authenticated but fails to run

File Authentication

File authentication is based on public key cryptography, but knowledge of this is not necessary to use file authentication on VeriFone terminals. The basic process involves using a secret *key* to *sign* files before initiating a download to the terminal. This produces a *signature* file downloaded along with the file.

The VeriShield File Signing Tool (FST) (Predator) or the VeriFone FST (Trident) included in the DTK are used to sign files. See the help files included with the tool for specifics on using VeriShield FST or VeriFone FST.

NOTE



VeriFone FST can be used on Predator provided that the user has VeriFone FST compatible file signing cards.

The Verix eVo OS checks the signature using the *key certificate* stored in the terminal. If the certificate passes the authentication process, Verix eVo concludes that the file is signed by someone who knows the secret key and that the file is valid.

Unlike Predator terminals, Verix eVo terminals are shipped from manufacture without a default certificate—a default certificate is not available for download.

NOTE



Certificate and key files included in the VVDTK are not valid on Verix eVo terminals.

For development, like for deployment, customers must obtain VeriShield signer cards. Development and production signer cards must be generated under distinct sponsor certificates, so that development cards could be distributed, without any security concern to personnel non-authorized to sign production software. The process of obtaining and installing certificates are not described in this manual. It is assumed that the developer has the proper signer cards and certificate files.

File Authentication and Downloads

Download the signature (.p7s) file and the file it signs. The signature file must always be loaded into I: drive, even if the file it signs is in F: drive. Both files *must* be loaded to the same file group. See DDL- Download Utility in the *Verix eVo Volume III Operating System Programming Tools Reference Manual* VPN DOC00303 for information on using the DDL utility.

NOTE



If a file fails authentication, a message is displayed. A key must be pressed to clear the message.

If you use a zip file to download, include the signature files in it. It is not necessary to sign the zip file.

Examples

- Download to I: drive:

```
ddl hello.out hello.p7s *go=hello.out
```

- Download to F: drive:

```
ddl SETDRIVE.F hello.out SETDRIVE.I hello.p7s *go=f:hello.out
```

or, more briefly, since I: drive is the default:

```
ddl hello.p7s SETDRIVE.F hello.out *go=f:hello.out
```

- Download to a different file group:

```
ddl SETGROUP.3 hello.p7s SETDRIVE.F hello.out
```

```
** VERIFYING FILES**
```

```
Check certificate  
OWNER.CRT
```

```
**AUTHENTIC**
```

Actual authentication is done on terminal restart (note that messages appear on the screen during the authentication process).

Initial loading of system certificates occurs. This screen appears for approximately one second after authentication completes.

All other certificates are processed with this message scheme. If the authentication does not succeed, the bottom line of the screen reads — FAILED —.

```
** VERIFYING FILES**
```

```
Compare Signature  
FOO.P7S  
FOO.OUT
```

```
**AUTHENTIC**
```

When signature files are processed, the name of the target file also appears. As with certificates, upon successful verification, this text line appears for approximately one second. Failure is indicated by displaying — FAILED — on the screen.

NOTE



If the file signature verification fails, a message indicating the failure is displayed. The file will not be executed. Press any key to clear the message.

Flash Full Error

During downloads to flash if the flash is full, an error screen appears to prompt the user to continue the download. Space must be cleared by deleting unnecessary files and defragmenting flash. Use the ***DEFRAG—Defragment Flash** environment variable to set automatic defragmentation of flash. *Defrag is a Predator only function. It is not used on Trident.

Run-Time Authentication

A more dynamic authentication allows for a continuous upgrade without restarting the terminal. For other types of updates, such as replacing a shared library, a restart is still required.

authenticate()

Marks the target file as authentic on conditions that the signature file is found, its format is correct, it identifies a target file that is also present, and the needed cryptographic signatures are correct. If the targeted file is a new shared library, it can be dynamically loaded.

NOTE



Before calling `authenticate()`, the signature and the file to be authenticated must be in the same subdirectory and the application must change to that subdirectory.

Prototype

```
int authenticate(const char *signature_file_name);
```

Parameters

`signature_file_name` Name of the signature file.

Return Values

Success: 1
Failure: -EINVAL or -22

Example

```
int h;
char curdir[256];
chdir("I:/");
mkdir("abc");
//OT36457 authenticate a file in a subdir
file_copy("tt.out", "abc/tt.out");
file_copy("tt_p7s", "abc/tt_p7s");
chdir("abc");
authenticate("tt_p7s");
//now check to see if the file is authenticated
```

file_copy_auth_bit()

Allows applications to copy the authenticated bit of a source file to a target file. This is useful when the application copies an authenticated file using `file_copy(const char *source, const char *target)` and wants to retain the authenticated bit.

Prototype `int file_copy_auth_bit(const char *source, const char *target);`

Parameter

<code>source</code>	Authenticated file
<code>target</code>	Target file

Return Values

Success	0, target file is marked as authenticated
Failure	-1 with <code>errno</code> set to: ENOENT - if file source or target does not exist. EINVAL - if file source and target are not in the same group, or if file source is not authenticated. EACCESS - if the cryptographic signatures of file source and target are not equal.

NOTE



The authenticated bit is lost after the file “target” is transferred to a different unit through back-to-back download, because it does not have a corresponding `.p7s` file. This function can only be used to set the target file authenticated if the source file is authenticated. It cannot set the target file un-authenticated due to the nature of flash memory. An un-authenticated source file returns an error. Lengthy processor-intensive operations such as file authentication or SSL encryption may be extensively slow on the older architecture.

Support for Compressed Files

The Verix eVo OS supports use of compressed files to reduce download time and memory use within the terminal. The OS provides the capability to decompress archived files. No support is provided to compress files within the terminal.

Format for Compressed Files

Files *must* be compressed using special features of the VeriCentre download server. Multiple files can be compressed within a single archive. The amount of compression varies with the type of file being compressed. Automatic Decompression

During terminal startup, the terminal checks for the environment variable `*UNZIP` in the Group 1 `CONFIG.SYS` file. If `*UNZIP` is set, the zip archive file decompresses at startup.

For example, for compressed files to be properly authenticated, the `.out` file must be signed *before* creating the archive file. Use the following procedure as an example:

- 1 Sign the `MYFILE.OUT` file to create the `MYFILE.OUT.p7s` file.
- 2 Compress both files to create the `MYFILE.ZIP` archive file.
- 3 Download the `MYFILE.ZIP` file.
- 4 Set `*UNZIP=MYFILE.ZIP`.
- 5 Set `*GO=MYFILE.OUT MYFILE.OUT.P7S`.

The terminal performs a software reset and processes the `*UNZIP` variable. `MYFILE.ZIP` is decompressed and removed from memory. `MYFILE.OUT` is authenticated using `MYFILE.OUT.P7S`. `MYFILE.OUT` then executes.

See also [*GO—Startup Executable Code File](#) and [*UNZIP—Decompress.ZIP](#).



Signing the zip file does not authenticate the `.out` file. The `.out` file must also be signed.

Use the previous procedure to ensure the file is authenticated.

Normal startup is delayed while automatic decompression occurs. In particular, the `*GO` parameter (specifies which application to execute) is not checked until after automatic decompression of the archive. Therefore, it is possible to download a compressed application, restart the terminal, decompress the application, and then execute it—all automatically. Note that this is the only automatic decompression service provided. In particular, downloading more than one zip file is possible, but only one automatically decompresses and it must be downloaded into Group 1.

Application Interface to Decompression Service

The following system call is provided for decompression:

```
int unzip(char *unzipfilename);
```

This call can be used by any application, not just by applications in Group 1. The name of the zip archive *must* be completely specified (for example, `F: FONTS . ZIP`, if the archive is in flash).

Successful invocation of the unzip operation results in a return code of 0. The original application can then continue operating during decompression. That decompression operation may ultimately fail (for example, if there is not enough memory space for all files to decompress).

Besides checking the result code immediately returned, the calling program can also use the `wait_event()` mechanism to synchronize with the `unzip()` process. On completion of the requested decompress operation, the calling program receives the `EVT_SYSTEM` event.

Only one program at a time can request this service. If one program issues an unzip call, no other unzip request is honored until the first one completes. Any attempt to schedule another unzip request results in a return code of `-ENOMEM`. In fact, no other program can start (with the `run()` system call) until the unzip operation is complete.

Verix eVo Support for File Groups

Whenever a file is unzipped, either automatically or as a result of a call to the `unzip()` function, the default location for the unzipped file is the memory file system for the current group. This destination can be overridden by creating the archive with specially named subdirectories. For example, suppose the PC has the following files in the current directory and several subdirectories:

Name on PC	Name in Terminal	File System	Group
X.OUT	X.OUT	I:drive	Current
3\Y.OUT	Y.OUT	I:drive	3
3\Y.DAT	Y.DAT	I:drive	3
F3\Z.OUT	F:Z.OUT	F:drive	3
F\MAIN.OUT	F:MAIN.OUT	F:drive	Current
15\COURIER.FON	COURIER.FON	I:drive	15
F15\ARIAL.FON	F:ARIAL.FON	F:drive	15
15F\ARIAL.FON	15F/ARIAL.FON	I:drive	Current

The redirection to flash or to other groups occurs because the files are identified in subdirectories with very specific names. In general, a file targeted for a subdirectory with a top-level name that is both entirely numeric and represents a valid group is eligible for special processing when the subdirectory prefix is stripped from the name and the file is placed in the specified group.

Similarly, if the top-level name is just *F* or if it begins with *F* but is otherwise entirely numeric and specifies a valid group, that file is placed in the default F: drive (or in the specifically identified flash group) and the triggering prefix removed.

Notice in the last example that the prefix *15F* does not satisfy any of the special rules, so that file is directed to the current group in I: drive. The backslash (\), used to identify subdirectories on the PC, is converted to a forward slash (/) as part of the archive creation process, and is retained in the terminal during the unzip operation.

NOTE



During back-to-back download, code file attributes of executable files with unconventional file extensions (not `.out` or `.lib` files) are preserved.

Determine UNZIP Results

Limited results can be obtained using the variable `UNZIP` in `CONFIG.SYS`, which is set to 0 when `UNZIP.OUT` starts, and to 1 on successful conclusion. Note that `*UNZIP AND UNZIP` are two different variables.

If a failure occurs during decompression (typically due to insufficient space in the file system), recovery procedures must account for the partial extraction (if some files were decompressed before the failure condition occurred, they remain in the file system).

If a file being decompressed already exists in the terminal, the file in the archive will replace the existing file. No warning or other notification is displayed, nor is any attempt made to verify that the surviving file is newer than the file it replaces.

User Interface

When `UNZIP.OUT` begins, it attempts to open the console to provide the user with details about its operation. Typically, these consist of the name of the zip file being read and the name of each file being written. For example:

```
** UNZIP stuff.zip  
  
somefile.out  
  
lotsadat.txt  
  
controls.dat
```

Error Codes

For fatal exceptions, numeric error codes are provided ([Table 134](#)). You must press any key to conclude the aborted operation.

```
** UNZIP Error n  
  
xxxxxx  
  
yyyyyy
```

Record these values to help determine the cause of failure (typically lack of available memory). The diagnostic screen displays until you press a key. Note that `xxxxxx` and `yyyyyy` often include the name of the file being extracted when the error occurred. The most common error is 19, `E_WRITE`, which indicates the target volume is full.

Table 134 displays a list of error codes.

Table 134 Unzip Error Codes

Error	Code	Description
E_NOMEM	1	Out of memory
E_OPEN_ZIP	2	Can't open zip file (name, errno).
E_MULTIDISK	3	Part of multi-disk archive.
E_ECREC_SIG	4	End central directory signature missing or bad.
E_ECREC_LOC	5	End central directory receive location wrong.
E_EMPTY_ZIP	6	Empty zip file.
E_EOF	7	Unexpected EOF.
E_SEEK	8	Seek-related error.
E_CFH_SIG	9	Central file header signature not found.
E_VERSION	10	Need later version (file, version).
E_METHOD	11	Unsupported compression method (file, method).
E_ENCRYPTED	12	Encrypted file (file, method).
E_OFFSET	13	Bad zip file offset (file, offset).
E_CDATA	14	Bad compressed data (file).
E_CRC	15	Bad CRC (file).
E_UNLINK	16	Can't delete old file (file, errno).
E_CREATE	17	Can't create new file (file, errno).
E_READ	18	Error reading zip file.
E_WRITE	19	Error writing output file.
E_SIG	20	Signal caught (-, signal).

Performance

As a rough estimate, the unzip operation writes the decompressed file at approximately 20,000 bps. This speed can vary significantly depending on the type of file.

Download Result Messages

Various messages can appear at the end of a download, including messages from the download host. Terminal-generated messages appear in [Table 135](#).

For messages generated by the download host, see the documentation for the VeriCentre host software.

At the end of a VTM-initiated download, the last error message remains on screen until a key is pressed.

Table 135 Download Result Error Messages

Error Message	Description
NO LINE	Phone line in use.
NO DIAL TONE	No dial tone.
NO CARRIER	Unable to establish communications.
BUSY	Busy signal.
NO ENQ FROM HOST	Host did not send ENQ.

Table 135 Download Result Error Messages (continued)

Error Message	Description
BAD RX COMM	Terminal received too many bad packets.
BAD TX COMM	Host received too many bad packets.
LOST CARRIER	Lost carrier during communications.
NO RESPONSE FRM HOST	Timed out waiting for packet from host.

Back-to-Back Downloads

Back-to-back downloads transfer all data from one terminal to another by transferring each user file from the Gold (sending) terminal to the Target (receiving) terminal.

All files initially download into I: drive, and are then redirected based on the directory and subdirectory names of the sending terminal's file system. Signature files must always be authenticated in I: drive. If the target file that the signature file authenticates is stored in flash, the signature file is moved to flash only after the target file successfully authenticates.

To successfully perform a back-to-back download, all signature files that are required to authenticate application executables must reside in the memory of the sending terminal. If the *FA variable is present in the Group 1 CONFIG.SYS file of the sending terminal, it must be set to 1 to retain all previously downloaded signature files.

If a signature file is missing on the sending terminal, the target application file that it authenticates is not authenticated on the receiving terminal and, if the target file is an executable, it is not allowed to run on the receiving terminal.

Hardware Requirements

Connect Verix eVo-based terminals using an RS-232 serial cable through the COM1 ports. The RS-232 serial cable must be designed with proper wiring and connectors for the COM1 port on each end (VeriFone P/N 05651-xx).

Special Considerations

There is special handling for signature and certificate files to comply with and preserve file authentication mechanisms.

Restrictions and Limitations

File authentication imposes restrictions on when a back-to-back download can execute. The following must be true for file transfers to work:

- The certificate trees in both terminals *must* be synchronized. See [Synchronized Certificate Trees](#).
- No required certificates replaced or removed from the Gold terminal.
- No required signature files removed from the Gold terminal. The default is to retain all signature files, but the *FA variable in CONFIG.SYS forces removal of these files.

NOTE



When using the `*FA` variable, back-to-back downloads will not work. For information on the `*FA` variable, refer to [*FA—File Authentication](#).

If the certificates in the terminals do not match, files do not authenticate properly once transferred to the Target terminal. If the certificate trees do match but a certificate has been replaced or removed, files on the sending terminal previously authenticated by the removed certificate cannot be authenticated on the receiving terminal.

If the customer does replace/remove a certificate, files previously authenticated by the removed certificate must be reauthenticated on the sending terminal prior to performing a successful back-to-back download.

When the back-to-back download completes and all certificates and signature files authenticate, the receiving terminal restarts. If the name of the `*GO` application is specified in the Group 1 `CONFIG.SYS` file of the receiving terminal, the application executes and the application prompt or logo is displayed on the terminal.

Synchronized Certificate Trees

The certificate trees of two terminals are synchronized if:

- They match *exactly*. Any and all certificate updates to one terminal are also applied to the other terminal in the same sequence.
- The sending terminal is exactly one revision newer than the receiving terminal.

The factory certificate tree consists of two certificates: *root and partition*. No file can be authenticated in this configuration until a sponsor and a signer certificates are installed. See [Figure 22](#).

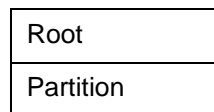


Figure 22 **Factory Certificate Tree**

The customer installs a sponsor certificate and a signing certificate (see [Figure 23](#)). At this point, all new files must be authenticated with the signing certificate.

Initial Customer Certificate Tree: Install sponsor certificate (authenticated by the partition certificate) and add new Signer1 (authenticated by sponsor).

Root
Partition
Sponsor
Signer1

Figure 23 Initial Customer Certificate Tree

At this point, the terminal is still synchronized with a factory unit and can initiate back-to-back downloads to a factory unit because it is—at most—one revision newer.

If the customer now replaces the signing certificate with a different certificate, the terminal is no longer synchronized with the factory terminal and cannot initiate back-to-back downloads to the factory unit. It can, however, be downloaded to a unit with the initial customer certificate tree, as shown in [Figure 22](#).

Second Customer Certificate Tree: Replacement sponsor certificate (authenticated by the partition certificate) and add new Signer2 (authenticated by sponsor).

Root
Partition
Sponsor
Signer2

Figure 24 Second Customer Certificate Tree

Valid Back-to-Back Downloads

The following shows back-to-back downloads in the *valid* direction:

- Factory terminal —→ Factory terminal
- Initial customer —→ Factory terminal
- Second customer —→ Initial customer

Invalid Back-to-Back Downloads

The following shows back-to-back downloads in the *invalid* direction:

- Second customer —→ Factory terminal
- Factory terminal —→ Initial customer

Units that are exactly one revision newer can initiate back-to-back downloads because the revision certificates, when sent to the Target terminal, can be authenticated then applied. Units with greater than one revision cannot initiate back-to-back downloads because the revision certificate was authenticated by an intermediate certificate not present in the Target terminal, so the Target terminal cannot authenticate the revision certificate.

In the above examples, a download from initial customer-to-factory state (one revision) works because there is no sponsor certificate installed in the target unit that would prevent another sponsor certificate from being installed.”

NOTE



You cannot initiate a back-to-back download from a second customer unit to a factory unit (two revisions) because the sponsor certificate in the second customer certificate tree is a replacement for the initial customer sponsor, which is not present in the factory unit.

Initiating a Back-To-Back Download

Back-to-back downloads are initiated on the sending terminal.

To Initiate a Back-To-Back Download:

- 1 Configure a Gold terminal for an application download operation to a deployment terminal:

If the *FA variable (if present in the application) is set to 0, you can reset it to 1. Ensure that the download is exactly what you want your Target terminals to receive. Ensure that previously authenticated files are not changed prior to the file transfer operation.
- 2 Configure the Target terminal to receive an application download from the Gold terminal. From VERIX TERMINAL MGR MENU 1, set Group 1 and COM1 as the port to receive the file transfer.
- 3 Establish a physical communication link between the Gold and the Target terminals.
- 4 From any Verix Terminal Manager menu on the Gold terminal, press [*] and enter the GID1 password to initiate the file transfer.
- 5 From VERIX TERMINAL MGR MENU 1 on the deployment terminal, select either a full or a partial download. The Gold terminal begins to transfer files to the Target terminal.

The terminal screen displays “UPLOADING NOW.”

See [Preparing a Terminal to Accept Downloads](#) if the Target terminal is not already at the DOWNLOAD NEEDED prompt.

NOTE



The user can interrupt the transfer at any time by pressing the cancel key on either terminal.

USB Flash Memory Download

The OS provides the USB Flash Memory Download functionality in the Verix eVo terminals. It allows the user to download a single file, `VERIFONE.ZIP`, into the current group from the USB flash memory. This download can be through VTM or an automatic download.

USB VTM Download

To begin downloading from a USB flash drive:

- 1 Insert the flash drive into the USB port of the terminal.
- 2 On VTM, Enter the target file group for the download. `FILE GROUP _1` (Group 1) is displayed as the default selection

To select a file group other than Group 1, press the number of the desired file group (2 – 15) for the download.

- 3 Enter the password of the selected file group. Verify your password and re-enter it.
- 4 Select 1> Single app to download a single application, or Select 2> Multi-app to download multiple applications.
- 5 Indicate whether to run a full or partial download.

Selecting FULL DNLD on a single application download will warn you that all existing files in the selected group will be deleted. Selecting FULL DNLD on a multiple application download, will prompt you to clear the existing application on the currently selected group. While downloading, in addition to copying the file `VERIFONE.ZIP` into the RAM of the current group, the VTM also sets the `CONFIG.SYS` variable `*UNZIP=VERIFONE.ZIP` in the current group. On restart, the archive is unzipped.

- 6 Select the appropriate download option - USB DEV in this example, and select CONTINUE to proceed with the download.

NOTE



USB FLASH MEMORY is displayed only on platforms that support USB and where the memory stick is plugged in.

- 7 The terminal automatically download the file `VeriFone.zip` from the USB flash drive. `USB DOWNLOAD COMPLETE` appears on the terminal screen after a successful download. If you performed a full download, the terminal restarts automatically. Otherwise, you must restart the terminal manually by selecting RESTART. If an application resides on the terminal following the download, it executes on restart.

On startup, the file authentication module authenticates any new signature files downloaded with the OS files. When the signature file authentication routine starts, the status display informs you of the progress of the authentication process.

If file authentication succeeds for a specific signature file, the ***AUTHENTIC*** message is displayed directly below the filename of the signature file. If file authentication fails for a specific signature file, the ***FAILED*** message is displayed for five seconds below the filename and the terminal beeps three times, allowing you to note which signature file failed to authenticate. The authentication process then proceeds to the next signature file until all signature files are validated.

When all new signature files are authenticated, the terminal restarts, and the application specified in the ***GO** variable or the default application in Group 1 executes and starts running on the terminal.

- 8** If the downloaded application successfully authenticates, the corresponding application prompt/logo is displayed upon restart. The terminal can now process transactions.

Logical File Transfers

Logical file transfers transfer all files in the user file system. No system files are transferred. RAM used for stack, heap, or other memory requirements (buffers, pipes, and so on) are not transferred.

RAM is automatically cleared in the receiving terminal at the start of the download (the same as a regular download). Flash is not cleared in the Target terminal. The download can fail if there is not enough room in file system to accommodate all files being transferred. Clear unwanted files on the Target terminal prior to initiating a download. The download can also fail if the file system is smaller on the receiving terminal than that of the Gold terminal.

File group information transfers with each file. File attributes are not transferred because the only way to set attributes is from an application call. Any application that sets the Gold's file attributes transfers to the Target terminal and sets the Target's file attributes when executed.

Authenticated files must be reauthenticated on the Target terminal. See [File Authentication](#).

Insufficient File System Space

Some files must be deleted. Files can be deleted in VTM or an application can provide a way to selectively delete files.

CONFIG.SYS Files

Each group has its own **CONFIG.SYS** file. The **CONFIG.SYS** files are transferred as files and any existing **CONFIG.SYS** files in the receiving terminal are replaced. This means that any settings in the Target terminal's **CONFIG.SYS** files are overwritten at transfer. See [Chapter 4](#) for information on the **CONFIG.SYS** file.

Date, Time, and Passwords

The date and time of the Gold terminal transfer to the Target terminal. The passwords of the Gold do not transfer.

USB Flash Auto-Download

On units that support USB Flash, auto-download provides a faster way to download `VERIFONE.ZIP` from the USB flash drive compared to the USB VTM download. Auto-download is initiated when the terminal is in idle state. When no applications are running, the Verix eVo operating system monitors the USB flash drive for the file `VERIFONE.ZIP` and then copies it to the I: drive `GID1` if exists.

How Auto-Download Works

- `Config.$$$` - The file `config.$$$` is for setting config variables for the applications. This file must use the Compressed Variable Length Records format. You can create this file using the `vlr` utility in the SDK. First, create a text file with the KEY, VALUE pair. Each KEY and each VALUE must be in a new line. Each GID can be loaded with a `config.$$$` file.

For example, you create a text file `config.txt` first with the following lines:

```
*GO  
  
app.out
```

Then you create the `config.$$$` file by running:

```
Vlr -c config.txt config.$$$
```

- `VERIFONE.zip` - Just like any other zip file, you can download this to the Verix eVo terminal except that the OS automatically sets the `*UNZIP` config variable to `VERIFONE.zip`. You can also include multiple `config.$$$` files in `VERIFONE.zip` to set config variables for different GIDs. The variables will be added to the existing `CONFIG.SYS` file if there is one.
- Full Download – The USB auto-download clears the I: drive files in `GID1` before copying `VERIFONE.zip` to I: drive, `GID1`.
- Start and End – The Verix eVo operating system starts copying `verifone.zip` from the flash drive to RAM when no application is running and when the USB flash drive with the file `VERIFONE.zip` is plugged in. This is an added feature to the auto-download through `COM1`.

The OS displays the message:

```
Sysmode Download G1  
  
Downloading Now
```

After completing the download, the OS displays either of the following messages, depending on whether or not the download is a success:

```
USB Download Complete  
  
USB Download Failed
```

NOTE



At this point, the user can unplug the USB flash drive and restart the terminal. If the flash drive is not removed and the config variable `*GO` is not set (or is invalid), the OS again downloads `VERIFONE.zip` after restarting.

- ***SMDL** - As an added benefit to the OS and application developers, when ***SMDL** is set, the Verix eVo operating system monitors the USB flash drive for a maximum of two seconds and downloads `verifone.zip`, if it exists, during OS startup. This is an added feature to the auto-download through `COM1`.

Adding Variables to VERIFONE.zip

When adding variables to the `VERIFONE.zip` file, include the file `CONFIG. $$$` instead of `CONFIG.SYS`. This prevents the OS from losing the unzipped result stored in the variable `UNZIP` and averting failure messages.

NOTE



When used, the `CONFIG.SYS` overwrites the existing variables. This causes the OS to lose the unzipped result stored in the `UNZIP` variable and breaks the communication between the application and the OS, which causes error messages. Whereas, `Config. $$$` is merged with the existing file so the unzip result in variable `UNZIP` can be easily read by the OS.

Automatic Download on VX 820 PIN Pad

The OS supports automatic download feature. Automatic download is initiated automatically in two ways:

- 1 Insert the external memory device (USB Memory Stick or SD Memory Card), or start the download at the download host (VeriCentre or PC running DDL), when the terminal is at the `DOWNLOAD NEEDED` screen.
- 2 Set `CONFIG.SYS *SMDL=1`, insert the external memory device or start the download at the download host, and restart or power cycle the VX 820 PIN pad.

The OS automatically performs a download from any one of the following sources:

- Default Serial Port
- USB device connection to external USB host
- SD memory card
- USB Memory stick

NOTE



Automatic downloads do not delete any files from the RAM or FLASH. This is different from downloads through the VTM Download menu.

When the OS is ready to perform an automatic download from an external memory device, it displays a screen that prompts the user if he wants to download all files from the external memory device.

- If the user selects **YES**, the OS performs automatic download. The OS informs the user when download is complete, and the external memory device should be removed.

Because the OS is performing a download, the normal download screens are displayed (the `WAITING FOR DOWNLOAD` screen, and the `DOWNLOAD STATUS` screen with a line of underscore characters replaced with asterisk characters as the download proceeds). However, because external memory device download speeds are very high, these screens appear to flash very briefly on the screen that it may not even be noticed by the user.

- If the user chooses `NO`, the OS displays a screen informing the user that the download is cancelled and the external memory device should be removed.

When prompted, remove the external memory device and press any key to restart the system. The OS cannot automatically restart when the download (or cancellation) is complete because, upon restarting, it immediately detects the presence of the download device and then triggers another automatic download. The OS then reboots after the automatic download is complete and after the user has pressed any key.

NOTE



By default, the OS uses the USB Memory Stick for automatic download when both the USB Memory Stick and SD Memory Card are inserted into a terminal at the same time.

Multiple ZIP file Downloads

VTM allows downloading of one or more ZIP files in various ways and the OS is able to search for subdirectories on the memory stick. Long file names on the flash drive are supported, unlike in the initial implementations. File names in the Verix eVo “I:” and “F:” file systems may be up to 32 characters long. The archive is directly unzipped from the `FLASH` drive, eliminating the need to copy the ZIP file to the Verix eVo file system and then unzipping it after the terminal restarts. This way, less space is required in the terminal (since it is not necessary to store the ZIP file at all) and the operation occurs immediately, rather than waiting for a restart.

Subdirectory Structure

The subdirectory `VERIFONE` must exist in the root directory of the memory stick. If not, then only the existing support for `VERIFONE.ZIP` applies. Within the subdirectory `VERIFONE`, additional ZIP files can be specified — for groups between 1 and 15, the designation `<n>` indicates the corresponding decimal number in ASCII.

Additional subdirectories may be specified in a way that clearly indicates their group association. Subdirectory `<n>` represents files destined for group `<n>`. Within the subdirectory `VERIFONE` a filename ending with `-<n>.ZIP` is intended for loading into the group `<n>` in RAM while filenames that do not end with `-<n>.ZIP` are intended for loading into the currently selected group.

Examples:

- The contents of the zip file `\verifone\abc.zip` are intended for the current group.

- The contents of the zip file `\verifone\8\abc.zip` are intended for group 8.
- The contents of the zip file `\verifone\sofpay-2.zip` are intended for group 2.

Downloading Into an Empty Terminal

When downloading into an empty terminal, if the `DOWNLOAD NEEDED` condition exists, then VTM automatically searches for an attached memory stick for eligible files. The following steps are then taken:

- 1 Searches for the file `VERIFONE.ZIP` in the root directory. If it exists, then it is downloaded.
- 2 Searches for the subdirectory `VERIFONE` in the root directory. If it does not exist, then no files are downloaded. In the initial implementation, the terminal simply ignores the drive.
- 3 Checks for the file `VERIFONE.ZIP` within the `VERIFONE` directory. If it exists, then the ZIP file is downloaded into the current directory.
- 4 Beginning with group 1, and proceeding similarly for each group `<n>`:
 - a Searches the subdirectory `VERIFONE` for filenames ending with `-<n>.zip`.
 - b Searches the subdirectory `verifone\<n>` for ZIP files. If such file is found, copies it to group `<n>` RAM or FLASH as appropriate, sets `*UNZIP` for group `<n>` as appropriate, and proceeds to the next group.

If more than one file is found, the first file is presented to the user for downloading. If the user selects `YES`, its contents are unzipped immediately to group `<n>`. The user is asked about each file in this group and then proceeds to the next group.

NOTE



It is possible for more than one ZIP file to be downloaded and unzipped in each group.

In cases where multiple ZIP files are detected (step 4 only), the system initially asks the user if he wants to download `ALL FILES`; if the user selects `YES`, then no further question is asked.

SD Memory Download

Trident terminals supports SD cards but SD card receiver is not populated on most terminals. The OS does not support application access to an optional SD memory card. The OS does not generate an audible tone when an SD memory device is inserted or removed.

The OS automatically detects the event when an SD memory card is inserted in the SD slot. It also determines that the particular device inserted is an SD memory card.

SD Memory Read

The SD memory card if the SD receiver is present is solely used for application downloads. When present, the VTM Download menu is enhanced to include SD as one of the download options every time the user enters the Download menu. If both the SD card and the USB flash memory are inserted, the OS only displays the USB flash download option and not the SD option.

The OS also supports automatic download from an SD memory card if no application is present or if `CONFIG.SYS` contains `*SMDL=1`. This is the same as automatic download for COM1 and USB Memory Stick.

SD Memory Write

Applications and libraries cannot use the SD memory card as an extension of internal RAM or FLASH. The OS does not provide an API for applications to access an SD device.

User Mode TCP/IP Download Support

The OS provides this functionality in the Verix eVo terminals by allowing the user to add menus to the VTM system.

VTM allows the port to be selected using menus like the following two screens:

VTM DOWNLOAD MGR Gn

```
1> Modem
2> COM1
3> COM7
4> SD Card
5> USB Flash Memory
6> TCPIP
7> USB Dev
8> COM6
9> COM2
```

If the user selects TCP/IP then the system considers several possibilities, based on a new variable in `CONFIG.SYS`.

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

If the user selects TCP/IP when the option is unavailable, an error tone is heard and a message is displayed.

NOTE



To go back to VTM menu, press `CANCEL` key.

If `*ZTCP` is present, then control passes to the specified download application for SSL downloads. The program needs to be in Group 1 (a Group 15 application may also be designated using the conventional forward-slash prefix). The program name is specified via a new `CONFIG.SYS` variable, `*ZTCP`.

For example, `*ZTCP=DNLDTCP.OUT` specifies a user-written download application.

VTM calls `run()` to run the specified program passing several parameters:

- `argv[1]` = download type: either “F” for full, or “P” for partial.
- `argv[2]` = download group: for example, “01” for group 1. The group will always be two digits to simplify parsing the user application.

NOTE



The same parameters are also communicated by VxEOS to NCP. NCP seeing arguments will run in download mode. Similar to Download, NCP will display the screens to confirm or enter the download parameters required by `SVC_ZONTALK_NET()`, `*ZA`, `*ZT`, `*ZN` and `*ZSSL`.

Prior to invoking `run()`, the VTM will close the console and allows the user application. It is up to the user application to configure the radio and perform the remaining download by itself.

If `*ZTCP` is not present and the module has an IP stack (ConnectOne Ethernet and WiFi), the terminal should “dial” for an IP download.

If `*ZTCP` is not present and the radio is either a CDMA or GPRS, an error message is displayed when TCP/IP button is pressed.

**No *ZTCP Variable
and no VxEOS**

If `*ZTCP` is present and is pointing it to a non-existent program, the terminal message displays `<Program Name> Not Found`. If the specified program is not authenticated, the message displays `<Program Name> Not Authenticated`. If the program fails to run, the message displays `<Program Name> Run Failure`.

When the user selects COM2, if `*ZTCP` is not present and the radio module is not a ConnectOne WiFi radio, the message is displayed as shown below:

NOTE



When an application runs another application, the first application must have access to the second. The second application is then assigned the “current” group ID of the caller.

For example, a group 1 application can issue `run("/foo.out" , "")` and launch application "foo", which resides in group 15, but with access now to group 1 files (and all others, via `set_group`). The same application "foo.out" can simultaneously call `run()` from a group 3 program. In this case, it will have access to group 3 (and group 15) files only. This inheritance property is because utility programs are often placed in the universal group 15.

Resumable Download

A resumable download (RDL) refers to a sequence of one or more files, usually ZIP files, which are divided into smaller files and then separately downloaded as individual files and reassembled in the terminal. This eliminates downloading the same data that has been partially downloaded from a previous attempt. VeriCentre keeps track of the parts that have been successfully downloaded and resumes download at the point of failure.

NOTE



ZonTalk file transfer protocol between Verix eVo and VeriCentre remains unchanged. Split files use the same download protocol as other files. RDL is only supported in VeriCentre 3.0 Enterprise and LE versions. Older versions of VeriCentre does not support RDL.

In a resumable full download, VTM does not remove files until all split files are received and checked for a correct checksum, unlike in a regular full download where VTM removes files before the download starts, which means losing the application when the download fails.

Split File Naming Convention

Split files have a unique naming convention to identify them. A single ZIP file can be split into parts that assume the format `#<ZIP filename>~n`, where `n` is the number suffix of the broken part; the last part of the split file takes the suffix `~LAST`.

Example

If a large file is split into ten small parts, the parts will assume the following format:

#Sample.ZIP~1

#Sample.ZIP~2

...

#Sample.ZIP~9

#Sample.ZIP~LAST

#Sample.ZIP~CRC

NOTE



CRC is computed using the same algorithm used to compute the CRC on Zontalk packets. The ZIP file may include a `CONFIG. $$$` file containing application environment variables to be merged into `CONFIG. SYS`. However, this is not required in the Verix eVo environment.

*ZA Length

The maximum size of the application ID in a download request is 10 characters, thus, in a resumable download, the application name stored in *ZA must not exceed 9 characters so that the “*” suffix required by VeriCentre is accommodated.

Combining Files

The VTM checks for the split files in all groups every time a VTM startup is performed:

- 1 If split files exists:
 - a Reassembles the split files.
 - b Checks the CRC. If CRC is bad:
 - Displays and prints message.
 - Removes split files in the group.
 - Sets flag to retry resumable download from the start.
- 2 If CONFIG. \$\$\$ file is found:
 - a merges contents into CONFIG. SYS.
- 3 If *UNZIP2 exists or *UNZIP exists,
 - a Processes zip file and removes files.
 - b Merges CONFIG. \$\$\$, if found in zip file.
- 4 If the previous step creates another *UNZIP2 or *UNZIP,
 - a Does not advance to the next group.
 - b Otherwise, advances to the next group.

NOTE



When needed, Flash is defragmented between every step to minimize exhaustion of free Flash space. If Flash space is low, this may take a long time due to many defragmentation cycles.

CONFIG. SYS Variables

Apart from split files, the resumable download searches for the following variables in each group:

Table 136 RDL CONFIG.SYS Variables

Name	Description	Minimum	Default	Maximum
*ZTRY	RDL retry count	0	5	30
*ZDLY	Delay in seconds between RDL attempts	0	5	60
*ZRESUME	RDL retry control	0	0	2
*ZRDL	RDL VTM UI Control	N/A	N/A	N/A

- If *ZRDL exists, the VTM download screens prompt the user whether to do a regular or resumable download. The value of *ZRDL is ignored.

- *ZDLY is the delay in seconds between download attempts value ranges between 5 - 60 seconds.
- *ZRESUME=0 (does not exist) means that download is not attempted even if it is incomplete or download failed.
- *ZRESUME=1 retries the download and reduces the retry count.
- *ZRESUME=2 retries the download but does not reduce the retry count.
- *ZRESUME is managed by the terminal and VeriCentre. There is no need for the user to define this.
- *ZTRY is the maximum number of retry's. A counter equal to *ZTRY is set when resumable download is started from VTM. The counter decrements for each download retry.

NOTE



*ZTRY is not decremented on each retry.

VTM Download

If *ZRDL exists, the VTM download screen displays option to perform a resumable download. The screen displays the appropriate message if download fails to complete (try count reaches maximum specified by *ZTRY). If the CRC in the ~CRC file does not match the computed CRC, a message is prompted for 5 seconds before the terminal reboots. If the retry count has not been exhausted, *ZA is reset to the application ID suffixed with '*'.

If a printer is available and paper is loaded, the download error messages send from the download host will be displayed and printed. VTM will not wait for a keypress to continue, unlike in a regular download.

Clear Split Files

In the VTM MEMORY FUNCTIONS screen, selecting the option CLEAR SPLIT FILES removes all split files in F: and I: in all groups. Flash is also defragmented after removing split files.

Maximizing Free Flash Space

Flash space can easily be exhausted if the split files and the resulting large ZIP file are all maintained in the Flash. To ensure that maximum free space in Flash is maintained:

- 1 Create output ZIP file #mapp.zip.
- 2 Open input split file #mapp.zip~1.
- 3 Append contents of the split file to ZIP file.
- 4 Delete #mapp.zip~1 split file.
- 5 Repeat steps for #mapp.zip~2 until #mapp.zip~LAST.

If there is enough free file space, the entire file can be reassembled with only one defragmentation cycle at the end to recover space taken by the deleted split files.

`*UNZIP2`

On startup, VTM searches for the environment variable, `*UNZIP2`, which is similar to `*UNZIP` with extra directives for removing files. On terminals running on OS that supports RDL, if `*UNZIP2` exists, then `*UNZIP` is ignored—consequently, if `*UNZIP2` is not present, then `*UNZIP` is processed as usual. On terminals running on OS that does support RDL, `*UNZIP2` is ignored.

To verify the integrity of the ZIP files, an additional check is done prior to processing `*UNZIP2`. If any ZIP file fails the integrity check, `*UNZIP2` is ignored. This is intended to prevent removal of old files followed by a failed unzip.

NOTE



These features can be used independently of resumable downloads.

***UNZIP2 File Removal Directive \$R**

The `*UNZIP2` remove file directive is embedded in the `*UNZIP2` field along with the names of ZIP files to be unzipped. The directive starts with the characters “\$R” followed by the file removal specification, which can be the name of a file to remove or a wildcard to remove groups of files.

Example If `*UNZIP2` is set to the following:

```
$R*:./, #A.ZIP
```

This specifies that all files in F: and I: in the current group be deleted. Flash is defragmented before `#A.ZIP` is unzipped.

Multiple \$R commands maybe used.

Example `*UNZIP2=$R*:2/, $R*:3/, g2app.zip, g3app.zip`

This removes the files from GID 2 and GID 3, defragments Flash and then unzips the files `g2app.zip` and `g3app.zip`.

***UNZIP2 File Removal List #REMOVE\$.\$\$\$**

If a long list of files must be removed by specifying each name, the file removal directive is limited by the maximum length of a `CONFIG.SYS` variable value. This can happen if two applications are sharing a directory but only one set of application files must be removed.

`#REMOVE$. $$$` is a compressed variable length record (CVLR) created using the same `VLR.EXE` application used to create `CONFIG. $$$`. Each record in the file contains a file removal specification similar the one used for the download ‘R’ packet.

Example If `#REMOVE$.TXT` contains the single line.

```
*: ./
```

This specifies that all files in F: and I: in the current group be deleted. VLR.EXE converts the text file to CVLR format.

```
vlr -c #REMOVE$.TXT #REMOVE$.$$$
```

The #REMOVE\$.\$\$\$ file is downloaded with a ZIP and is included in the *UNZIP2 list of files.

Example

In the following example, #REMOVE\$.\$\$\$ and *UNZIP2 are downloaded to GID2. The ZIP file is downloaded to F: GID15. The '#' prefix on the ZIP file prevents it from being deleted by the normal file removal process.

```
ddl setgroup.2 #REMOVE$.$$$ *defrag=0 \
*unzip2=#REMOVE$.$$$,F:/#bigfat.zip \
setdrive.F setgroup.15 #bigfat.zip
```

When VTM entry processes *UNZIP2, it checks each zip file before removing any files. When the file #REMOVE\$.\$\$\$ is found, it removes files as specified by each record in the file. The file may contain as many records as needed. In this example, *:./ results in removing all files in GID2 F: and I:. The ZIP file is not removed because it is in GID15. After reaching the end of #REMOVE\$.\$\$\$, the file is removed.

VTM unzips the file F:/#bigfat.zip, after unzipping, the file is removed. Finally, after processing *UNZIP2 in all groups, VTM entry proceeds on as before.

File Removal Specification Syntax

The syntax follow the format: [drive] [group] [filename]

A drive is specified by the prefix of "F:" or "I:" or "*" for both. If no drive is specified, I: is used. After the optional drive prefix, an optional group may be specified. The group is indicated as an optional number followed by a slash ("/"), such as "3/" for group 3. A slash by itself represents group 15, as usual. The symbol "*" indicates all groups, while "." represents the currently selected group. If no group is specified, the currently selected group is used.

Following are the optional Verix eVo filenames that can be used:

File	Removes "file" from I: in the current group
F:file	Removes "file" from F: in the current group
I:file	Removes "file" from I: in the current group
*:file	Removes "file" from both I: and F: in the current group
/file	Removes "file" from I: in group 15
F:/file	Removes "file" from F: in group 15
n/file	Removes "file" from I: in group n
F:n/file	Removes "file" from F: in group n

<code>./file</code>	Removes “file” from I: in the current group
<code>./</code>	Removes ALL files from I: in the current group
<code>I:./</code>	Removes ALL files from I: in the current group
<code>F:2/</code>	Removes ALL files from F: in group 2
<code>*:./</code>	Removes ALL files from both I: and F: in the current group
<code>*/</code>	Removes ALL files from I: in ALL groups
<code>F:*/</code>	Removes ALL files from F: in ALL groups
<code>I:*/</code>	Removes ALL files from I: in ALL groups
<code>*:*/</code>	Removes ALL files from F: and I: in ALL groups

If the download is started in groups 2 - 15, group file access restrictions apply as usual. For instance, if the download is started in group 2, `RF:1/` is not allowed because this would remove all files in group 1 Flash which a group 2 application should not be able to do.

If all files in RAM in all groups are removed (for example, `R*: */`, `RI: */`, or `R*/`), `CONFIG.SYS` in groups 2 - 15 are removed and the unprotected entries in group 1 `CONFIG.SYS` are removed. In all other cases, `CONFIG.SYS` is not affected. In particular, `RI:n/` removes all files except `CONFIG.SYS` in RAM in group n and no changes are made to `CONFIG.SYS`.

Merging CONFIG.\$\$\$ into CONFIG.SYS

`CONFIG.$$$` is merged into `CONFIG.SYS` before the unzip phase and after each ZIP file is processed. This allows a single ZIP file to contain `CONFIG.$$$` and ZIP files for multiple GIDs.

Breaking multiple application downloads into smaller ZIP files helps reduce the chance of running out of free space in Flash. In addition, a variable in `CONFIG.SYS` can be removed by including a record in `CONFIG.$$$` with a zero-length value record.

Zip Files Inside Zip Files

If a ZIP file contains one or more zip files:

- The `CONFIG.$$$` file is unzipped to its destination group.
- The child ZIP files are unzipped to their destination groups.
- `Config.$$$` is merged to `CONFIG.SYS`.

If there are multiple `Config.$$$` files, group number takes precedence. That is, `Config.$$$` file unzipped in group 1 is first checked for `*UNZIP2` variable. The child ZIP file associated with `*UNZIP2` is then unzipped. If the child ZIP file contains the file `Config.$$$`, it is merged into `CONFIG.SYS`.

Using the `*UNZIP/*UNZIP2` feature to unzip a files leave the unzip result code in the `CONFIG.SYS` variable `UNZIP`. When unzipping multiple ZIP files, `UNZIP` contains the result code from the last unzip operation. The unzip result code is then saved for each zip file.

Example After processing *UNZIP=A.ZIP,B.ZIP, the following are left in the CONFIG.SYS assuming the files are successfully unzipped:

```
UNZIP_A=1      (new) Unzip result code for A.ZIP
UNZIP_B=1      (new) Unzip result code for B.ZIP
UNZIP=1        (old) Unzip result code for B.ZIP—the last ZIP file.
```

Automatic File Removal

Automatic file removal eliminates old application files in the terminal while downloading new files. This function can be set to optional file removal when *ZRDL is defined—this allows the terminal to run old applications in case the new application download fails.

NOTE



Ensure that there is enough space in the terminal to hold the old and new applications (usually compressed in a ZIP file) at the same time.

VTM always clears RAM and Flash files before the download proceeds. If the download fails, the terminal is left without an application.

```
VTM DOWNLOAD MGR Gn
****WARNING****
```

```
All Files Will Be
Cleared From Group
```

```
1>Cancel Download
2>Continue
```



SVC_ZONTALK(/download())

I: drive and F: drive files are removed when a “meaningful packet” is received. This feature removes files when the terminal established contact with the download host before receiving new files. This keeps the old application files if, for example, the download host phone line was busy.

NOTE



File removal triggered by “meaningful packets” is disabled during resumable downloads.

IP Persistence

To ensure that IP download or dial download can be performed anytime, Comm Server and VMAC application files as well as the data files associated with them, need to be protected when the terminal is cleared of memory. GIDs 1 and 15 can be configured as protected GIDs to ensure that these files are not deleted in the event of memory clearing.

NOTE



This is supported in all hardware releases that support communications other than Dial only.

Setting Up a Protected GID

The `CONFIG.SYS` variable, `*GUARD`, solely supported for GID 1 and GID 15, indicates to the OS that the GIDs are protected. The variable `*GUARD` is placed in GID 1, and the OS only searches for it in GID 1. If this is located in other GIDs, the variable will be ignored.

The `*GUARD` variable will have the following values:

<code>*GUARD =1</code>	Protection in GID 1 is enabled
<code>*GUARD =15</code>	Protection in GID 15 is enabled
<code>*GUARD =1 , 15</code>	Protection in GID 1 and 15 is enabled
No <code>*GUARD</code>	Default value. Protection is disabled for backward compatibility.

Deleting `*GUARD` variable

The `*GUARD` variable can be cleared from the terminal. Once this `CONFIG.SYS` variable is cleared, the protected GID can also be deleted. The following mechanisms apply when deleting `*GUARD`:

- The variable can be edited from the VTM—it can be disabled or deleted at VTM.
- The variable can be cleared through a VeriCentre download if the `*GUARD` variable is set to empty.
- The variable can be cleared by downloading an empty `CONFIG.SYS` file.
- The `CONFIG.SYS` file can be deleted through the remove command in VeriCentre.

Protecting the Application Files

The OS does not allow files or parameters to be cleared from the GID if it is configured to be protected—all requests to clear a protected GID will be ignored. This protection applies to both RAM and Flash files. These files will remain in the terminal even if the user selects “CLEAR MEM” option.

When a download is performed from VTM, the application prompts the user to clear the application from the group. The OS walks the user through the prompts but does not, in fact, delete a protected group.

NOTE



In order to delete the GID, the user first needs to delete the variable `*GUARD` from the `CONFIG.SYS` file in GID 1.

VeriCentre Downloads

During download process, VeriCentre can instruct the OS to either remove all files through a wildcard (`*.*`) or to remove a specific filename. However, if GID1 contains the `CONFIG.SYS` parameter that instructs the OS to protect the files, then the remove all files through a wildcard command will be ignored and the files will not be cleared from the GID. The VeriCentre request to remove a specific file name will still be performed.

NOTE



This GID protection is similar to the file protection feature currently implemented in the OS.

The Comm Server application enables the IP download. Removing this application from the terminal causes an error. Thus, downloads need to be configured in such a way that GID 1 is not deleted before Comm Server is downloaded. Otherwise, the terminal ends up doing dial downloads only.

During VeriCentre download, when a remove command is sent, it will be ignored at the terminal level in GIDs 1 and 15 if both are marked as protected—the OS only removes files from GIDs 2 - 14 instead. Continuous update of Comm Server and VMAC is supported because VeriCentre has the ability to overwrite existing files in GID 1, and add new files.

NOTE



Users cannot delete these files from the terminal through a VeriCentre remove or a Clear GID command.

File Protection Application

The `*GUARD` protection mechanism protects the guarded GIDs during mass deletions (i.e. memory clearing, wildcard deletions, etc.), but files and variables can always be replaced or removed individually. This prevents users from accidentally deleting protected data while in VTM, but still allows individual files and variables to be updated or removed via downloads or applications.



WWAN

This appendix describes the application interface to the Wireless Wide Area Network (WWAN).

The goal of using a WWAN is portability. The terminals can be used in non-traditional settings—mobile and transportable.

WWAN support consists of the radio modem module connected to a COM2 port. There is also a software-controlled power switch to turn the on and off power to the radio modem. VeriX eVo OS consists of OS function calls that control the additional hardware.

Radio Modem Function Calls

The device handle for the radio modem is `DEV_COM2`. The application can then turn the radio modem on and off with function calls. The terminal must be awake to receive data from either modem. To receive an incoming call, the terminal must have done the following:

- 1 Wake up the terminal.
- 2 Place the modem in a mode where an incoming ring is signaled over the data lines.
- 3 Set events to trigger the application.

NOTE



COM2 and COM3 ports cannot be opened simultaneously in Vx610.

The following function calls are specific to the radio modem, you may also refer to this APIs:

- `get_port_status()`
- `reset_port_error()`

set_gsm_break()

Sends a break signal to the GSM radio.

Prototype `int set_gsm_break(int hdl);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

NOTE



This function call is not supported in VX 680 3G.

set_gsm_powersave()

Sends the GSM radio power save command setting. The power setting is a range from 0 to 9.

Prototype `int set_gsm_powersave(int hdl, unsigned int power);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1: EINVAL

NOTE



This function call is not supported in VX 680 3G.

set_radio_ctl()

Controls the settings of RAD_MOD, RAD_RST, and RAD_OFF.

Prototype

```
int set_radio_ctl(int hdl, const char *sigs);
```

Parameters

hdl Handle for COM port.
*sigs Pointer to the data.

Return Values

Success: 0

Failure: -1

- If invalid pointer, or cannot read buffer, `errno` is set to `EACCES`.
- If executed on a COM port that does not support `set_radio_ctl()`, `errno` is set to `EINVAL`.
- If open block is not set, `errno` is set to `EINVAL`.
- If the radio is GPRS, and the application is trying to set `RAD_OFF`, and `nBATT_INT` is 0, return `EACCES`.

NOTE



Unused bits 3-7 should be set to 0.

Example

The following example sets `RAD_OFF`, `RAD_RST`, and `RAD_MOD` to 1.

```
char sigs = 0x07;
set_radio_ctl(hdl, &sigs);
```

NOTE



`set_radio_ctl()` function works perfectly only on `/Dev/COM2` and should not be called with any other devices.

get_radio_sts()

Returns the status of the signal RAD_INT and RAD_INT2.

Prototype `int get_radio_sts(int hdl, char *sigs);`

Parameters

<code>hdl</code>	Handle of the COM port.
<code>*sigs</code>	Pointer to the data.

Return Values

Success: 0

Failure: -1

- If invalid pointer, or cannot read buffer, `errno` set to `EACCES`.
- If executed on a COM port that does not support it, `errno` is set to `EINVAL`.
- If open block not set, `errno` is set to `EINVAL`.

NOTE



Unused bits 2-7 should be set to 0 or 1. Never assume the unused bits will be 0.

CDMA

The CDMA radio module consists of a Sierra Wireless EM3420 CDMA radio module. The CDMA radio SERIAL port speed is 115,200 bits/s. For more details refer to *EMXXX Development Kit AT Command Reference Manual*, E-2130394 Revision A from Sierra.

NOTE



The VX 680 CDMA uses Huawei EM660 EVDO radio module.

Communication to the wireless modules is over the COM2 serial port. This supplies the necessary handshake signal lines to the radio—RTS, DTR, CTS, DCD, DSR, which are accessed via the standard `set_serial_lines()` function. Additional signals specific to radio modems are RAD_RST, RAD_OFF, RAD_MOD, RAD_INT and RAD_INT2.

Hardware Interface

There are seven RS-232 lines available on the wireless radio serial port. The following table shows the connections between the COM2 port and the new Kyocera and current radio modules.

Table 137 COM 2 Port Connections

COM2 Lines	Direction	Kyocera CDMA Lines	Siemens GPRS Lines	CO WiFi Lines
RX	<---	RXD Mux	RXD	TXDH
TX	--->	TXD Mux	TXD	RXDH
RTS	--->	RTS Mux	RTS	CTSH
CTS	<---	CTS Mux	CTS	RTSH
DTR	--->	DTR	DTR	DSRH
DSR	<---	RI	DSR	DTRH
DCD	<----	DCD	DCD	RIH

There are five radio control lines in addition to the RS-232 interface lines. The first column of the table shows the lines available to the applications to control the radio modem. The first three signals are outputs from the CPU to the radio modules. The other two are inputs to the CPU from the radio modules.

Table 138 Radio Control Lines Connections

COM2 Lines	Direction	Kyocera CDMA Lines	Siemens GPRS Lines	CO WiFi Lines
RAD_MOD	--->	Mux Control	UNUSED	MSEL
RAD_RST	--->	/VEXT	/IGN	/RST
RAD_OFF	--->	/XCRV_EN	/EOFF	ON/OFF
RAD_INT	<---	XVCR_DET	VDD	RIH ^ DCDH
RAD_INT2	<---		/RING0	DCDH

SVC.H Symbols

The following symbols are available in `SVC.H` for use with the `set_radio_ctl()` and `get_radio_sts()` functions.

//Radio Control Outputs

```
#define RAD_MOD      (1<<0)
#define RAD_RST      (1<<1)
#define RAD_OFF      (1<<2)
```

// Radio Control Inputs

```
#define RAD_INT      (1<<0)
#define RAD_INT2     (1<<1)
```

// Kyocera M200 CDMA

```
#define M200_SEL_UART1      (0)
#define M200_SEL_UART2      (RAD_MOD)
#define M200_ON_ASSERT      (0)
#define M200_ON_DEASSERT    (RAD_OFF)
#define M200_VEXT_ASSERT    (0)
#define M200_VEXT_DEASSERT  (RAD_RST)
#define M200_XCVR_ON        (RAD_INT)
```

Wireless Module ID EEPROM

The radio modules have a small EEPROM (electrically erasable programmable read only memory) containing a module ID that allows the OS and applications to determine which radio module is installed. The radio module is connected to COM2.

The landline modem has a resistor, which identifies the modem. The OS reads this resistor value on startup to identify the modem connected to COM3.

The return value for `SVC_INFO_MODULE_ID(2)` is `MID_M200`, which has the value of 70.

Module ID Override

The `GID1 CONFIG.SYS` variables `COM2 HW` and `COM3 HW` allow the correct COM2 and COM3 module IDs to be overridden with user specified values.

CAUTION



This feature is intended for testing and should be used with caution. Using this feature can cause applications and the OS to incorrectly handle the module.

The variables are not protected (do not start with # or *), thus, are erased on full downloads and RAM file clears. This is designed to reduce the risk of this feature being unintentionally enabled in the field.

Example: Setting `COM2HW=11` in `GID1` causes `SVC_INFO_MODULE_ID(2)` to return 11 regardless of which wireless module is installed on `COM2`.

Setting `COM3HW=3` in `GID1` causes `SVC_INFO_MOD_ID()` and `SVC_INFO_MODULE_ID(3)` to return 3 regardless of which modem is installed on `COM3`.

GPRS

The GPRS radio modem consists of the Siemens MC55/56 GPRS radio module. The radio can be turned on using the `set_radio_ctl()` function or by powering the terminal off and back on. For more details refer *MC55/MC56 AT Command Set*, E-MC5555ATC0011 from Siemens.

NOTE



Do not turn off the radio module (GPRS or CDMA) using AT commands. When switched off using AT commands, the radio module is not capable of receiving any command to turn it back on, except by physically resetting the terminal. Instead, use the function `set_radio_ctl()` to turn the module off and then back on.

Do not change its baud rate using AT commands.

On terminals where there is a requirement that the GPRS radio should only be powered when the SIM is present. The SIM is located under the battery. The hardware has been added such that the signal `nBATT_INT` is generated to indicate that battery present.

If the terminal detects that there is no battery and the radio is GPRS, the OS will activate the `/EMERG OFF` pin on the radio module, thus shutting the radio off.

VX 680 GPRS

The VX 680 GPRS is an enhanced release version the Vx670 GPRS terminal. It uses the Siemens MC55i GPRS radio and the MRD531 MSR decoder chip.

The function `SVC_INFO_MODULE_INFO(2)` returns the symbol `MID_MC55i_ONLY`, which is 72 for the MC55i module. The function `SVC_INFO_MODULE_INFO(3)` returns the value 50.

NOTE



The VX 680 GPRS OS also runs on VX 680 Classic—the OS reads the hardware configuration information from the Manufacturing Information Block (MIB) to determine which hardware version is present.

The VX 680 GPRS does not support internal USB Hub chip and control of the USB Host power. It is not possible to use USB dongles on the USB Host port on the multi-I/O connector while the terminal is in the base.

VX 520 GPRS

The VX 520 GPRS terminal has variants with and without battery configuration. Units that do not have battery configuration are externally powered without a battery, thus, battery APIs return `EINVAL`. The environment variables relating to battery functions are not used. The `COM2` serial port is used to communicate to the Siemens MC55i GPRS radio module, similar to the VX 680 GPRS. There is no external connection to the `COM2` port.

NOTE



The landline modem greatly reduces the radio modem's receiver sensitivity, thus, only a single COM port can be opened at any given time (either `COM2` or `COM3`, but not both at the same time).

The serial port supplies the following handshake signal lines to the radio:

- `RTS`
- `DTR`
- `CTS`
- `DCD`
- `DSR`

These signals are accessed via the standard `set_serial_lines()` and `get_serial_lines()` functions.

Additional signals specific to radio modems are:

- `RAD_RST`
- `RAD_OFF`
- `RAD_MOD`
- `RAD_IINT`
- `RAD_INT2`

These radio control lines are in addition to the RS-232 interface lines. The function `set_radio_ctl()` is used control signals from the CPU to the radio modem. The function `get_radio_sts()` is used to read inputs from the radio modules to the CPU.

Symbols for the Above Functions

The existing symbols for the Siemens MC5x radio family may be used.

```
// Radio control outputs
#define RAD_MOD (1<<0)
#define RAD_RST (1<<1)
#define RAD_OFF (1<<2)
// Radio control inputs
#define RAD_INT (1<<0)
#define RAD_INT2 (1<<1)
```



```
// Siemens MC55/56 GSM/GPRS
#define MC5X_IGT_ASSERT (0)
#define MC5X_IGT_DEASSERT (RAD_RST)
#define MC5X_EOFF_ASSERT (0)
#define MC5X_EOFF_DEASSERT (RAD_OFF)
#define MC5X_VDD_ASSERTED (RAD_INT)
#define MC5X_RI_ASSERTED (RAD_INT2)
```

GPRS modems GSM 7.10

The GSM 7.10 multiplexor (mux) protocol supports multiple virtual channels over a single physical channel. When this protocol is implemented inside the OS device driver, the IP stack is allowed to own a virtual channel for PPP packets, another task to own a virtual channel for connection establishment and control, and another task to own a virtual channel for network link monitoring.

For backwards compatibility, the GSM mux defaults OFF. The application that opens the GPRS COM port must call `set_opn_blk` to enable the GSM mux protocol. Once this is done, two additional devices may be opened—`/dev/gmx1` and `/dev/gmx2`.

The `/dev/com2` now includes an additional option to struct `Opn_Blkc.protocol`, `P_gsm_mux`. When the application specifies this new option, the device driver sends commands to the GSM modem to enable the GSM mux protocol.

The `/dev/gmx1` and `/dev/gmx2` are opened after the application has enabled the mux protocol. These devices support the usual open, close, read, write, status, and control functions for serial ports.

USB EM660 Radio Modem Power Management

Please refer to [USB_COM2_RESET\(\)](#) for USB EM660 Radio Modem power management API.

3G

Please see [3G](#) for more information.

WiFi Module

On VX 680, the WiFi/BT device is Broadcom BCM4329 chip.

The WiFi module on VX 680 3G terminals uses the same Broadcom BCM4329 WiFi chip that is used for VX 520. It is distributed with the EOS software package.

The WiFi module shall undergo WiFi certification procedures for 802.11b/g WPA(2) Personal/Enterprise with EAP TLS. This must be done as WiFi Certification must be done for each terminal/device.

The number of receive buffers in the driver can be increased to support increased latencies due to multiple networks running at the same time or due to CPU efficiency improvements by batch processing of receive packets.

The number of transmit buffers in the driver can be increased to maintain the required throughput while latencies increase due to multiple networks running at the same time or due to CPU efficiency improvements by batch processing of transmit packets.

Bluetooth Radio Module

Point-to-Point links (like those that are not using an access point) with WiFi can be done with IBSS. WiFi Direct is not required or desired.

The Bluetooth Module on VX 680 3G uses Broadcom BCM4329. This is the same chip used on VX 520 terminals. The Bluetooth Driver is loaded separately from the OS distributed with the EOS software package.

The number of receive buffers in the driver can be increased to support increased latencies due to multiple networks running at the same time or due to CPU efficiency improvements by batch processing of receive packets.

The number of transmit buffers in the driver can be increased to maintain the required throughput while latencies are increased due to multiple networks running at the same time or due to CPU efficiency improvements by batch processing of transmit packets.



USB Support

USB Flash Drive

This section details the USB features supported in the Verix eVo environment.

On terminals that support it, this enables the VTM to copy the file `VERIFONE.ZIP` from the root directory of a Flash drive to `I:VERIFONE.ZIP` in GID 1, and sets `*UNZIP=VERIFONE.ZIP`. For more information, refer to [USB Flash Memory Download](#).

USB Ethernet (ETH1)

In some terminals, some of the devices are permanently connected (ASIX AX88772 USB Ethernet) while others are plugged during operations. The USB Ethernet is independent of the modem but both can operate simultaneously.

On others, the USB Ethernet is an external device, which is a permanently attached internal device. The USB Ethernet device requests 150mA and can be directly powered from the USB host port on the terminal or any of the host ports on the cradle. No more than one USB Ethernet device may be connected at any given time; only the first device connected to the terminal is recognized.

Still others support the existing USB to Ethernet hardware, packaged in a “dongle” cable instead of mounted internally. When this cable is used, applications can access the Ethernet using `DEV_ETH1` (“/DEV/ETH1”). Applications cannot receive Ethernet disconnect events because the Ethernet dongle cable is also the power cable. The only way to disconnect the Ethernet is to disconnect the power.

The Vx810 DUET base station may contain an optional USB Ethernet device. If the Ethernet device is present on the DUET, `UDB_ETHER` is set in the USB device status word. The API `get_component_vars()` then returns a file name of “usbax772.bin” for the USB Ethernet device. The VX 820 and Vx805 DUET use the USB Ethernet dongle `DEV_ETH1`.

When at the same time, the USB Ethernet device is present on the DUET and another external USB Ethernet dongle is connected to the DUET USB Host port (USB Type A), the OS renders the external USB Ethernet dongle useless (the standard USB connect/disconnect chime sounds) and treats the additional external USB Ethernet dongle as unsupported device.

The Verix V AX88772 device driver presents two devices to application programmers. Each device supports `open()`, `close()`, `read()`, `write()`, and the event bit `EVT_ETH1`.

- `/DEV/ETH1` is for Ethernet data packets formatted as Ethernet 802.3 packets. This device is normally owned by the IP stack.
- `/DEV/ETHTRP1` is for Ethernet link status packets.

Below is a format of network status packets:

```
struct eth_link_status {  
    uint32_t status; // 0 link down, 1 link up  
};
```

USB Ethernet Functions

This section details the APIs used in USB-based Ethernet.

- `get_enet_status()`
- `get_enet_MAC()`
- `set_enet_rx_control()`

You may also refer to these APIs:

- `open()`
- `read()`
- `write()`
- `close()`

get_enet_status()

This API checks whether Ethernet link is live or not. The status is returned in the status4 array.

On BT, this function returns the connection status of the PAN network.

Prototype int get_enet_status(int hdl, char *status4);

Parameters

hdl	The handle is the value returned by open().
*status4	*status4 Pointer to a 4 byte array. Bit 0 of status4[0] is 1 if the Ethernet link is up. Bit 0 is 0 if the link is down. All other bits in the array are reserved.

Return Values

Success:	0: The status array contains the link status.
Failure;	-1 and errno set to EBADF.

NOTE



The following code demonstrates the correct way to test for link status.

```
if ((status4[0] & 1))
// link up
else
// link down
```

get_enet_MAC()

This API returns the Media Access Control (MAC) address.

On BT, this function returns the Bluetooth BD_ADDR.

Prototype int get_enet_MAC(int hdl, char *MACbuf);

Parameters

hdl	The handle is the value returned from open(DEV_ETH1, 0).
*MACbuf	6 byte Ethernet MAC address.

Return Values

Success	0: MAC address is filled in the MAC buffer.
Failure	-1 and errno set to EBADF: hdl is not a valid open file descriptor. -1 and errno set to EACCES: MACbuf is not writable.

set_enet_rx_control()

This function call is used with the USB Ethernet dongle. This function enables/disables multicast and broadcast. Multicast is bit 0x08 and Broadcast is bit 0x02.

On BT, this function is provided for compatibility with ETH drivers. It does nothing but return 0.

Prototype `int set_enet_rx_control(int hdl, int rx_control);`

Parameters

hdl	Device handle
-----	---------------

Return Values

Success:	0
Failure:	-1 : EINVAL

USB Client

The OS determines the type of USB client device it presents to a USB host at boot time. After booting, the USB client device cannot be changed.

The device supports connection to Windows XP PC using the updated `Vxuart.inf` included in the SDK.

NOTE



When creating a PC/Windows application using Trident's USBCLIENT driver, the application should do the following:

- 1 Enable a 2ms delay in between read and write operation.
- 2 On reading data, the application should read until zero length value is returned (zero length means that all the data has already been received).
- 3 Always check the error code on read or write operation.

HID Client

The OS allows the USB client port to be a Human Interface Device (HID). When the OS boots in this mode, it supports USB-enhanced DDL downloads from the host to the VX 820 PIN pad. The VTM Download menu presents USB as one of the options that users can select. When the terminal is configured as a USB HID, the USB device API is disabled. HID mode only supports downloads using the enhanced DDL application.

RS-232 Client

The OS allows the USB client port as an RS-232 device. This client is compatible with the Windows driver, `usbser.sys`, which is included in all Windows XP and Windows 2000 installations. Some terminals provide a `VXUART.inf` file for the customer. The `.inf` file is installed to Windows and it identifies the connected device. When the OS boots on this mode, the standard DDL can be used on the host. The VTM Download menu contains USB as one of the options to choose.

NOTE



`CONFIG.SYS *ZB` is not supported for USB downloads.

USB Client API

The OS presents the standard device API to the application. This means the application is able to issue `open()`, `close()`, `read()`, and `write()` commands to the USB Client device.

NOTE



The USB device API is disabled if the unit is configured as HID.

The application can access the USB device as `/DEV/USBD`. USB Client events are reported on event `EVT_USB_CLIENT` (bit 15 in the event mask). The RS-232 mode allows the application to use the USB interface the same way it would use a standard UART-based COM port. This means that the OS returns `success` for API calls like `set_opn_blk()` even though it has no actual effect on the USB interface. This allows existing applications expecting a standard COM port

interface to use the USB port in RS-232 mode with minimal code changes.

However, since RS-232 mode is emulated the same way as a standard UART, `set_opn_blk()` must be called first before calling the `read()/write()` operations. If `set_opn_blk()` is not called, `read()/write()` operations return `EINVAL`. The functions `set_opn_blk()`, `reset_port_error()`, `set_serial_lines()`, `set_fifo_config()`, `get_open_blk()` and `get_fifo_config()` return success since there is no physical UART.

The API `get_port_status()` only returns success, or `EACCESS` if the buffer parameter is an invalid pointer. Since RS-232 mode is emulated as standard UART, `set_opn_blk()` must be called first before calling `get_port_status()`, if `set_opn_blk()` is not called, `get_port_status()` returns `EINVAL`.

USB Client also introduces the following APIs:

get_usbd_status()

Checks whether USB initialization is complete.

Prototype `int get_usbd_status(int hdl);`

Return Values

- 1: USB is initialized.
- 0: USB not yet initialized.

The USB device HW does not have a disconnect interrupt line. The event `EVT_USB` only generates upon connection. Use the API `get_usbd_status()` to know if the USB device is disconnected.

usb_pending_out()

Returns the amount of written but unsent data in the driver's buffers.

Prototype `int usb_pending_out(int hdl);`

Return Values

- Success: The amount of written, unsent data in the driver's buffer.

USB Host

Applications are not expected to receive USB connect and disconnect events in the VX 820 PIN pad because its single connector contains both power and signal lines. It is impossible to disconnect the USB device without also disconnecting the power. Connect and disconnect tones are unlikely to occur in an operational configuration because the power and the USB Host signals are in the same cable. The external USB device cannot be connected and disconnected without simultaneously disconnecting the power to the VX 820 PIN pad.

A developer cable with USB device and USB Host connector is available. When used, the VX 820 PIN pad inherits the USB connect/disconnect tones in the VX 680 terminal. The tones may sound different on the VX 680 terminal and the VX 820 PIN pad depending on the buzzer hardware and case design. The difference in tones is outside the control of the OS.

USB Thermal Printer

The DUET contains a micro-controller (MCU) connected to a thermal print mechanism. The OS communicates with the printer MCU through the DUET USB driver.

NOTE



Applications should always use the defined bit name for the device, in this case UDB_COM4, rather than the actual hex value.

The API `get_component_vars()` returns a file name of "com4_usb.bin" for the USB thermal printer device — the Micro-controller firmware. The MCU firmware provides the standard Verix eVo printer feature set and controls the DUET printer mechanism optimally.

Font Memory

The MCU font memory can be configured from 0 to 256 Kb of fonts with a default of 64 Kb. At start-up, the Verix eVo printer driver sends the value of `*PRTFNT`, if present, to the MCU. On the VX 820 DUET, `*PRTFNT` is not used, thus, it has no effect on the printer operation.

Logo Memory

The MCU logo memory can be configured from 0 to 10 logos with a default of 1 logo. Each logo requires 12 Kb of memory. At start-up, the Verix eVo printer driver sends the value of `*PRTLGO`, if present, to the MCU. On VX 820 DUET, `*PRTLGO` is not used, thus, it has no effect on the printer operation.

Printer ID

Because the VX 820 DUET printer driver provides a completely different architecture, the command "Request Printer ID" (`<ESC> "i"`) returns a value of "4" which is unique to the Duet's USB printer device.

Firmware Version

The "Request Firmware Checksum" command (`<ESC> "CS;"`) returns a string of text containing each module's checksum and the firmware version and build date/time:

```
"ID ACT CAL\r\n"
```

```
"01 4353 4353\r\n"
```

"\r\n"

"SW Version : *0P8IB1A1*\r\n"

"SW Build Date : *Sep 25, 2007*\r\n"

"SW Build Time : *16:48:39*\r\n"

The information allows the user and/or the OS to uniquely identify what version of the printer MCU firmware is currently loaded. The exact value used in the data fields that are italicized may change from one firmware version to the next.

MCU Firmware download

The VX 820 OS downloads the MCU firmware when necessary, which resides in a remote flash memory device in the DUET base station. The MCU firmware is not affected even if the OS is updated.

The MCU firmware load is similar to modem profiles load. At boot up, the OS checks for the presence of a special printer MCU firmware file named *IBUSBPRN.MCU* in the memory. If present, the OS downloads it to the printer MCU in the DUET base station. When the download is complete, the OS deletes the file upon first use of the printer device. The printer MCU file does not need to be retained in the VX 820 memory because it is burned into the flash memory for the DUET's printer MCU.

CAUTION



Power loss during a firmware download may render the printer permanently unusable. Ensure that an Uninterruptible Power Supply is used during firmware updates. A firmware download lasts approximately 30 seconds after the OS has fully booted.

Downloading Modem Profile and Printer Firmware at the same download session might cause unusual behavior. The message indicating that printer firmware download is in progress may be absent during the download and a blank screen is shown instead. Do not remove power during this time as printer firmware downloads take up to 30 seconds. This happens when the modem driver initiates an OS restart after loading the modem profile. The time to load a modem profile is much shorter than the time to load the printer firmware – thus, if the printer driver also began a firmware download, it holds off the OS restart until the printer firmware download is complete. This protects the printer from flash corruption. The OS architecture does not allow screen messages to be displayed during the OS restart state – the screen is blank. Once the printer firmware has finished loading, the OS restart completes with both the new modem profile and the new printer firmware loaded.

MCU Firmware File

The firmware file is a binary image file built for download to the printer MCU. Download file names start with the letter “D” (i.e., a file name used for a downloadable module might be `DIBT02Q6.BIN`). The loader begins a Zontalk download protocol beginning at offset zero incrementing until the end of the file is reached.

Firmware Distribution Files

The firmware file must be signed and authenticated by the OS upon download. FW files are signed using the same VeriFone Operating System signing keys as operating system upgrades. Two files are distributed for each firmware release:

- 1 `IBUSBPRN.ZIP` – Contains all files needed to download firmware files using the `DDL.EXE` serial port downloader.
- 2 `VERIFONE.ZIP` – This file may be copied to a USB memory device. When plugged into the Integrated Base, USB port files may be downloaded into the file system.

Additional USB Printer Firmware Download Rules

Once the printer firmware has been downloaded to the printer MCU, a power cycle should be performed before any subsequent printer firmware downloads are attempted. The firmware file is deleted only after a successful download when the printer device is opened by an application or VTM. Printer firmware files loaded will overwrite any existing Printer firmware file. A Printer firmware file overwriting a successfully loaded Printer firmware file will not be downloaded to the Printer MCU. Any existing printer firmware files must be deleted before starting subsequent downloads. Performing printer diagnostics from VTM opens the device, causing the firmware file to be deleted.

Once the Firmware file is loaded, the OS is restarted to authenticate the file. Upon the first restart, the file is not yet authenticated so the download does not occur. After authentication, the terminal restarts and the authenticated firmware downloads.

Printer Device API

When operating in stand-alone mode and the application attempts to open the printer device, the OS returns `-1` with `errno` set to `ENODEV`. If the application opens the printer device when the VX 820 PIN pad is connected to a DUET base station, the OS returns a valid handle.

Print Buffer Management

The VX 820 DUET printer MCU maintains a print buffer that holds printable characters received from the application's `write()` command. The contents of the buffer is not sent to the printer mechanism until the MCU receives an LF character or the contents of the print buffer exceed the right margin. The printer MCU is located in the DUET base, which is not reset when the application restarts. Similarly, the VX 820 DUET printer is not reset when the OS undergoes a "soft" reboot or when the application is restarted (by `SVC_RESTART`).

NOTE



A soft reboot happens when the OS enters and exits VTM. The hardware is not power cycled in a soft reboot as opposed to a hard reboot where power is cycled.

The VX 820 DUET printer behaves like an external printer, and is not aware that the main CPU in the VX 820 has restarted. The MCU retains any pending print data in its print buffer before the VX 820 PIN pad is restarted. This data needs to be flushed so that the next time the VX 820 opens COM4 and sends new print data, the old data is not printed.

At start-up, the application initializes the printer to a known state by sending a series of printer commands that include the CAN character, as well as setting the font, character attributes, and inter-line spacing, among others. The CAN character flushes out the print buffer and resets the character attributes to the default state (disable inverse and disable double height/width).



ASCII Table

An ASCII table for the Verix eVo operating system display is presented as [Figure 25](#) below. The table is formatted for quick reference, as follows:

- The letters and numbers in the column to the left of the table and in the row above the table are when combined, the hexadecimal value of an ASCII character located in the corresponding row and column coordinate.
- The numbers shown in white on a navy blue background within the table itself are the decimal value of the ASCII characters in that table cell.
- The large character located in the middle of each cell is the ASCII character.

For example, to determine the hexadecimal value of the plus (+) sign:

- 1 Locate the plus sign ASCII character in the table (decimal 43).
- 2 From this position, follow the row to the left and view the hexadecimal value in the column outside the table. This value (2) is the first digit of the ASCII character's hexadecimal value.
- 3 Now, from the plus sign, follow the column to the top of the table and view the hexadecimal value in the row above the table. This value (B) is the second digit of the hexadecimal value.
- 4 The hexadecimal value for the ASCII plus sign (+) is therefore 2Bh.

Control Characters

When the default 6 x 8 font is selected, control characters (values <32 or 20h) do not display. Some control characters cause specific actions to occur, such as clear the screen (FF), move the cursor to the start of the next line (LF), move to the previous column (BS), and so on.

If a different built-in font is selected, for example 8 x 16, the control characters may appear to be similar to the corresponding character 4 rows below it and have a preceding carat (^). For example, NUL is "Control+@", but may appear as "^@", ESC is "Control+[\" or "^[\", and US is "Control+_\" or "^_\".

Default and built-in fonts display the DEL character as a checkerboard pattern that fills the entire character cell.

		Least-Significant Nibble															
		_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
Most-Significant Nibble	0_	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
		16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
		DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	1_	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
		48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	2_	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
		80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	3_	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
		112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	4_	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
	5_		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
	6_		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>
	7_		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N
			P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^
			`	a	b	c	d	e	f	g	h	i	j	k	l	m	n

Figure 25 ASCII Table For Verix eVo-based Terminal Display



GLOSSARY

asynchronous smart card A card that follows the ISO 7816 International standard.

ATR [an] Answer to reset.

BCD Binary coded decimal

BWT Block waiting time

CBC Cipher Block Chaining'

CFB-64 64-bit Cipher Feedback

checksum A simple error-detection scheme where each transmitted message is accompanied by a numerical value based on the number of set bits in the message. Each time the system starts the application, it validates the checksum of all the files in the file system.

checksum A value used to ensure data is stored or transmitted without error. It is created by calculating the binary values in a block of data using some algorithm and storing the results with the data.

coalesce To unite into a whole. The "garbage" collection process for the Verix eVo Flash file system (F:).

code files Usually stored in the Flash file system.

communication device buffer 64 bytes in length and can contain up to 63 bytes of data. The maximum number of allocated buffers is 255, and the minimum number is 1. By default, the maximum number is allocated.

CONFIG.SYS A compressed ASCII format file maintained as a keyed file.

count byte The first byte in a counted-string record, representing the size of the data area that follows, plus one.

CRC Cyclic redundancy checks are a form of checksum used to detect data communication errors.

CVLR Compressed variable-length record are identical to VLR files with the addition of a compression algorithm applied to the data on writing and reading.

data files All files used by the application, including CONFIG.SYS, files for batches, negative files, reports, and so on.

direct download Transfers data from the development PC to the terminal through a cabled direct connection.

DTK Developer's toolkit. A package available from VeriFone containing tools to support application development in the Verix eVo environment.

EPP external PIN pad. A peripheral device that connects to the terminal to allow a customer to enter a PIN.

FIFO A first-in-first-out queue of bytes, typically used as a buffer.

file extension feature Allows you to specify a maximum size for a file.

file handle A number the operating system assigns to a file when opened. The operating system uses the file handle internally when accessing the file.

generic files Can contain any type of data.

gold terminal The *Gold* (sending) terminal and the *Target* (receiving) terminal are connected by a cable, and applications and files download between the terminals.

ICC Integrated circuit card; smart card.

IFD Smart card interface device.

IFSD Information field size reader.

IPP internal PIN pad. An internal device that allows a customer to enter a PIN.

ISR Interrupt service routine.

key The first record in a CVLR file; gives the data record an alphanumeric name or identifier, providing random access to the records. Can also represent a secret value used in cryptographic algorithms, such as VeriShield file authentication or APACS40 message authentication.

keyed files Allows records to be accessed by unique character-based strings. In a keyed file, each record consists of two elements: a key value and its associated data. See CVLR files.

LRC Longitudinal redundancy check; the XOR of the bytes. An error checking method that generates a parity bit from a specified string of bits on a longitudinal track.

LSB Least-significant bit.

Luhn check A standard scheme for detecting data entry and transmission errors in account numbers. Letting the least-significant (right-most) digit be digit 1, each odd-numbered digit is added to the sum, and each even-numbered digit is “double-added.” Double-adding

memory management Maximizes file space available for application and data files and ensures adequate space is available for stack and heap use.

MIB Management information block.

MMU Memory management unit.

MSB Most-significant bit.

NMI Non-maskable interrupt is hardware interrupt used to protect file systems during power failures. The interrupt bypasses and takes priority over interrupt requests generated by software and keyboard and other devices.

non-volatile RAM Storage area in RAM for data files.

PIN Personal identification number. A security feature usually used in conjunction with ATM card transactions.

pipes A temporary software connection between two programs or commands.

PTT Postal, Telegraph, and Telephone: The governmental agency responsible for combined postal, telegraph, and telephone services in many countries.

rasterization The conversion of vector graphics (images described in terms of mathematical elements, such as points and lines) to equivalent images composed of pixel patterns that can be stored and manipulated as sets of bits.

real-time clock Device that maintains the current date and time and provides a source of interrupts for system timing.

remote download Transfers data over the public telecommunications network (phone line).

SCC Serial communication control.

SCC buffer Storage connecting circuit buffer.

SDLC Synchronous data link control is a data transmission protocol used by networks and conforming to IBM's Systems Network Architecture (SNA).

slosh File movement back and forth in memory.

SPI Serial peripheral interface.

synchronous smart card A card with its own unique timing sequence for communication and control rather than following the ISO 7816 international standard. Also referred to as a memory card.

VVDTK Verix V Development Toolkit

VLR Variable-length record files are suited to ASCII data and can contain arbitrary data, including embedded NULL characters. Data is accessed by record number rather than byte address.

VTM Verix Terminal Manager.

volatile RAM Storage area in RAM for all global and local data and the system stack and heap.

WWAN Wireless Wide Area Network



Symbols

(257, 266, 271, 272

() 124, 129, 130

*get_errno_ptr() 195

_remove() 137

_rename() 139

Numerics

1DES master key 624

2TDEA DUKPT 683

3G 793

A

abort 59

Account Data Encryption 683, 684, 691

 ADE APIs 685

 ADE Display Status 691

 ADE Feature Enablement 691

 ADE Key Load Menu 692

 ADE Menu Flowchart 692

 ADE On/Off Menu 692

 ADE State Controls 691

 ADE Status Menu 692

 ade_active() 690

 ade_encrypt() 687

 ade_status() 689

 Encrypted Key Loading 684

 Turn ADE On or Off 691

 Visual Indication 691

 VTM Menus 691

acronym list 28

activate_task() 255

AES() 570

ALPHA key 244

alpha_shift() 256

ANSI X9.19 message authentication code 700

applications

 event word 221

 sleep mode 221

ASCII table 807

asynchronous cards 409

automatic file decompression 758

auto-repeating keys 251

B

backlight 296

back-to-back downloads 176

battery_conditioner_status() 82

baud rate 493

baud rates 497

Bluetooth Radio Module 794

buffers

 application 122

BWT 391, 404, 407

C

card_magprint_count() 369

card_magprint_data() 369

card_magprint_stat() 370

card_mode() 370

card_pending() 370

card_raw_data() 371

CCITT algorithm 348

certificate trees 762

 default 763

 factory 763

 initial customer 764

 second customer 764

 synchronized 763

certificates for file authentication 754

character mode initialization 493

character set 437

checksum 337

checksum validation 173

Cinterion PHS8-P 3G 723

Cleartext Key Loading 683

clock 413

clock functions See *functions*.

clock ticks 417

close file 143

close() 143

clr_timer() 226

clreol() 257

clrscr() 257

coalesce flash 160

Communication Channels 493

communication sequence 506

compress ASCII data 710

compressed files 758

compressed variable-length record (CVLR) 74, 111,
125, 130, 133, 710

 files 110

CONFIG.SYS 74, 171, 767

CONFIG.SYS environment variables

 *ARG 172

 *B 172

 *FILE 109

- *GO 74
- *SMDL 180
- *SMPW 181
- *TIME 181
- *UNZIP 182
- *VALID 182
- *ZA 514
- *ZM 515
- *ZT 514
- CHKSUM 172
- UNZIP 182
- console
 - determine the handle for 276
 - ownership 253
 - sharing 192
 - transfer ownership 253
- console functions. *See functions, console*
- conventions
 - filenames 107
 - measurement 27
- counted strings 310
- country setting for terminal 311
- crash log 334
- CRC 345
- CRC functions. *See functions, CRC.*
- CRC, cyclic redundancy check 345
- CRC16 algorithm 349
- CRC32 algorithm 351
- crypto libraries 563
- crypto_read() 566
- crypto_write() 567
- cryptography 754
- cryptography functions 569
- current key 699
- current owner variable 703
- cursor
 - positioning 281
 - turn on/off 297
- CVLR. *See also, compressed variable-length record.* 74
- CWT 404
- D**
- data
 - format 493
- date setting 122
 - time setting 767
- date/time 411
- date2days() 411
- datetime2seconds() 411
- day-of-week setting 122
- days2date() 411
- decompress ASCII data 711
- decompress files 758
- decompression
 - application interface 759
 - implementation 761
 - performance 761
 - user interface 760
- decrypt_session_data() 696
- decryption 696
- default font. *See font*
- delete
 - key 146
 - record pair 146
- delete(), delete_vlr(), and delete_cvlr() 133
- delete_() 133
- DES encryption 698
- DES() 571
- device
 - handles 362
 - names 362
- device drivers
 - console 253
 - serial printer 517
- device management functions. *See functions, device management*
- device ownership 364, 366
- Device Ports 723
- devices
 - beeper 418
 - capability enumeration 400
 - magnetic card reader 367
 - real-time clock 387
- dir_flash_coalesce() 159
- dir_flash_coalesce_size() 160
- dir_get_all_attributes() 147
- dir_get_attributes() 147
- dir_get_file_date() 147
- dir_get_file_size() 148
- dir_get_first() 149
- dir_get_next() 150
- dir_get_sizes() 151
- dir_put_file_date() 152
- dir_reset_attributes() 153
- dir_set_attributes() 154
- Direct Download Utility (DDL) 737
- disable_kbd() 266
- display
 - ASCII table 807
- display contrast 271, 272
- download
 - result messages 761
- download fonts 462
- download function call 558

download() 558
 downloads 558
 and file authentication 754
 back-to-back 738, 762
 certificate revisions 765
 interrupting 751, 765
 invalid direction 764
 valid direction 764
 download-and-resume 514
 from system mode 750
 full 514
 partial 514
 drivers 308
 console 253
 mag driver return codes
 MAG_BADJS 121
 MAG_BADLRC 121
 MAG_BADTRK 121
 MAG_NODATA 121
 MAG_NOERR 121
 MAG_NOETX 121
 MAG_NOSTX 121
 MAG_PARITY 121
 MAG_REVETX 121
 See also, *device drivers* 517
 Dual SIM Design 729
 DUKPT
 IPP7 665
E
 _exit() 195
 EBC 404
 EMV PINs 694
 EMV smart card 694
 enable_hot_key() 270
 enable_kbd() 265
 enable_key_beeeps() 270
 encrypted session key 695
 encryption key 694
 environment functions. See *functions*.
 Environment Variables
 *AKM 172
 *APNAME 172
 *ARG—Arguments 172
 *B 172
 *BCM 172
 *COM1RB 173
 *COM2RB 173
 *COM3RB 173
 *COM8RB 173
 *COMBO 173
 *CPAD 173
 *DARK 173

*DBMON 174
 *DEBUG 174
 *DEBUGO 174
 *DEBUGT 174
 *DEFRAG 175
 *DIAG 175
 *DIRCOM 175
 *DOT 175
 *DOT1 175
 *ESDMON 175
 *ETHSPD 175
 *FA 176
 *FILE 176
 *FK_FKEY 176
 *FK_HKEY 176
 *FK_TITLE 176
 *FKEY 176
 *GKE 177
 *GO 176
 *GUARD 177
 *HEAP 177
 *IPPMKI 177
 *KEYBOARD 178
 *LOG 178
 *LOGP 178
 *MA 178
 *MAXSH 178
 *MENUx 178
 *MENUxy 178
 *MERR 178
 *MN 179
 *OFFD 179
 *PIPE 179
 *POW 179
 *PRNT 179
 *PROT 179
 *PRTFNT 180
 *PRTLGO 180
 *PW 180
 *RKLMKI 180
 *SCTOUT 180
 *SMDEF 180
 *SMDL 180
 *SMGIDS 180
 *SMPW 181
 *SMU2U 181
 *SMUI 181
 *SOFT_PWR_OFF 181
 *SYSCHK 181
 *TIME 181
 *TMA 181
 *TURNOFF 181

- *TURNON 181
- *TZ 181
- *TZRULE 181
- *UNZIP 182
- *UNZIP2 181
- *USBCLIENT 182
- *USBGDEV 182
- *USBGPID 182
- *USBGVID 182
- *USBRESET 182
- *VALID 182
- *VSOPATH 183
- *WEAR 183
- *ZB 183
- *ZDLTYPE 183
- *ZDLY 183
- *ZH 183
- *ZINIT 183
- *ZN 184
- *ZR 184
- *ZRDL 184
- *ZRESP 184
- *ZRESULT 184
- *ZRESUME 184
- *ZS 184
- *ZSSL 184
- *ZSWESC 184
- *ZTCP 184
- *ZTRY 184
- *ZX 184
- CHKSUM 172
- COM2HW 173
- COM3HW 173
- UNZIP 182
- error beep 418
- error codes
 - API interface
 - EACCES 700, 701, 702
 - EBADF 371, 700, 701, 702, 703
 - ENOSPC 700, 701, 702
 - EBADF 148, 214
 - EINVAL 229
 - ENODEV 119
 - ENOSPC 229
- files
 - EEXIST 112
 - ENOENT 137
- flash
 - EBUSY 160
- keyed files
 - EBADF 146
- error log 334
- error message
 - flash insufficient 753
- error messages
 - insufficient flash 753
 - insufficient RAM 753, 767
- error_tone() 420
- event 221
- event functions. *See functions, event*
- event mask 233
- events
 - EVT_KBD 253
- exception handling 225
- exceptions 225
- external PIN pad 693
- F**
 - fatal exceptions 334
 - Feature Enablement 732
 - feature_license_expiration() 733
 - feature_license_get_detail() 735
 - feature_license_get_tag() 734
 - feature_license_is_enabled() 734
 - feature_license_total() 733
 - FIFO functions. *See functions, FIFO.*
 - FIFOs 340
 - file attribute functions. *See functions, file attribute*
 - file authentication 754, 762
 - certificates
 - default 763
 - initial customer certificate tree 764
 - partition 763
 - root 763
 - Signer2 764
 - downloads and 754
 - key certificate and 754
 - signature file 754
 - file decompression, automatic 758
 - file directory functions. *See functions, file directory*
 - file functions. *See functions.*
 - file groups 107, 759
 - Group 0 107
 - Group 1 107
 - file groups, and decompressing files 759
 - file handle 113, 143
 - termination 144
 - file information functions. *See functions.*
 - file lock 194
 - file naming conventions 107
 - file pointer 113
 - file position pointer 133
 - file positioning functions. *See functions, file position-*
ing.
 - file signing tool 754

- file updates 135
- files
 - .p7s
 - automatic decompression in terminal memory 758
 - close file 143
 - code file 190
 - compressed and downloads 758
 - conventions for 106
 - create a new file 112
 - decompression, and groups 759
 - file directory
 - first file determination 149
 - subsequent file determination 150
 - file directory information 151
 - file handle 109, 113, 143
 - file position pointer 133
 - filenames 107
 - font 299
 - groups 107
 - insert data into 132
 - keyed 110, 144
 - new file 112
 - paired 110
 - position pointer in 113
 - See Also *seek pointer*, *seek*
 - primary user group 107
 - SETDRIVE 752
 - SETGROUP 752
 - signature 176, 754
 - signing 754
 - store files 106
 - update information 135
 - variable-length (VL) 109
 - write access 113
 - writes
 - seek to the end 113
- fixed-length records 131
- flash
 - amount installed 160
 - coalesce 160, 175
 - defragmenting 175
 - file system 190
 - full 755
 - insufficient error message 753
 - insufficient FLASH error 753, 767
 - memory 106
- flash directory functions. See *functions*.
- font
 - font file 299
- font data organization 437
- font definition file 291
- font files
 - .fon 238
- Font Generation tool 299
- fonts
 - 5 × 8 462
 - 8 × 14 462
 - default 237, 253
 - default 237
 - download fonts 462
 - font files 237
 - Font Generation tool 299
 - tables 462
- functions
 - application
 - sponsoring application 176
 - beeper
 - error_tone() 420
 - normal_tone() 421
 - clock
 - date2days 411
 - datetime2seconds 411
 - days2date 411
 - seconds2datetime 411
 - secs2time() 411
 - SVC_VALID_DATE 411
 - time2secs() 411
 - console
 - activate_task() 255
 - alpha_shift() 256
 - clreol() 257
 - clrscr() 257
 - contrast_down() 258
 - contrast_up() 258
 - delline() 265
 - disable_hot_key() 265
 - disable_key_beeps() 266
 - enable_hot_key() 270
 - enable_key_beeps() 270
 - get_console() 276
 - get_font() 278
 - get_font_mode() 279
 - getcontrast() 270, 271, 272
 - getfont() 272
 - getgrid() 273
 - getinverse() 273
 - gotoxy() 281
 - inverse_toggle() 283
 - put_graphic() 290
 - screen_size() 292
 - set_backlight() 276, 277, 278, 296, 297, 298
 - set_cursor() 297
 - set_font() 299, 300

- set_hot_key() 301
- setinverse() 295
- SVC_INFO_DISPLAY() 302
- SVC_INFO_KBD() 303
- wherecur() 303
- wherewin() 304
- wherewincur() 304
- window() 305
- write_at() 305
- CRC
 - SVC_CRC_CALC() 347
 - SVC_CRC_CALC_L() 347
 - SVC_CRC_CCITT_L() 348
 - SVC_CRC_CITT_M() 348
 - SVC_CRC_CRC16_L() 349
 - SVC_CRC_CRC16_M() 350
 - SVC_CRC_CRC32_L() 351
 - SVC_LRC_CALC() 337
 - SVC_MEMSUM() 337
 - SVC_MOD_CHK() 338
- device management
 - get_name() 364
 - get_owner() 364
 - set_owner() 366
- event
 - clr_timer () 226
 - peek_event () 226
 - read_event() 227
 - set_timer () 229
 - SVC_WAIT() 232
 - wait_event () 233
- FIFO
 - SVC_CHK_FIFO() 342
 - SVC_CLR_FIFO() 342
 - SVC_GET_FIFO() 343
 - SVC_PUT_FIFO() 343
 - SVC_READ_FIFO() 344
 - SVC_WRITE_FIFO() 344
- file access
 - close() 143
 - open() 116
 - read() 120
 - read_cvlr() 125
 - read_vlr() 124
 - write() 126
 - write_cvlr() 126, 130
 - write_vlr() 126, 129
- file attribute
 - get_file_attributes() 140
 - get_file_max() 140
 - reset_file_attributes() 141
 - set_file_attributes() 142
 - set_file_max() 142
- file directory
 - dir_get_attributes() 147
 - dir_get_file_date() 147
 - dir_get_file_size() 148
 - dir_get_first() 149
 - dir_get_next() 150
 - dir_get_sizes() 151
 - dir_put_file_date() 152
 - dir_reset_attributes() 153
 - dir_set_attributes() 154
- file information
 - get_file_date() 135
 - get_file_size() 134
 - SVC_CHECKFILE() 136, 173
- file positioning
 - lseek() 131
 - seek_cvlr() 131
 - seek_vlr() 131
- files
 - close() 143
 - delete() 133
 - delete_cvlr() 133
 - delete_vlr() 133
 - insert() 132
 - insert_cvlr() 132
 - insert_vlr() 132
 - remove() 137
- flash
 - dir_flash_coalesce() 159
 - dir_flash_coalesce_size 160
 - SVC_FLASH_SIZE() 160
- get_env() 187
- IPP
 - get_port_status() 433
- keyed files
 - getkey() 145
 - putkey() 146
- magnetic card reader
 - card_pending() 370
- Master/Session 695
 - decrypt_session_data() 696
 - gen_master_key() 697
 - gen_session_key() 698
 - test_master_key() 698
- Modem Sleep State
 - modem_sleep() 513
 - modem_wake() 513
- One Way 701
- pipe interface
 - pipe_connect() 216
 - pipe_init_char() 214, 218

pipe_init_msg() 52, 200, 205, 206, 207, 208,
 209, 210, 211, 212, 214
 pipe_init_msg() 52, 218, 219
 pipe_pending() 219
 put_env() 188
 Reset_Key 699
 RS-232
 download() 558
 get_port_status() 81, 82
 SVC_ZONTALK() 514
 security/crypto library
 AES() 570
 crypto_read() 566
 crypto_write() 567
 DES() 571
 GenerateRandom() 572
 isAttacked() 572
 rsa_calc() 574
 SHA1() 575
 service
 SVC_VALID_DATE() 411
 string conversions
 ltoa() 706
 strnlwr() 707
 strnupr() 707
 SVC_2INT() 714
 SVC_AZ2CS() 713
 SVC_DSP_2_HEX() 709
 SVC_HEX_2_DSP() 708
 SVC_INT2() 714
 SVC_PACK4() 710
 SVC_UNPK4() 711
 ultoa() 706
 SVC_RESTART 220
 system information
 get_component_vars() 308
 SVC_CHK_PASSWORD() 310
 SVC_INFO_COUNTRY() 311
 SVC_INFO_CRASH() 334
 SVC_INFO_DISPLAY() 302
 SVC_INFO_EPROM() 335
 SVC_INFO_HW_VERS() 312
 SVC_INFO_LIFETIME() 333
 SVC_INFO_LOTNO() 313, 314
 SVC_INFO_MFG_BLK() 315
 SVC_INFO_MOD_ID() 316
 SVC_INFO_MODELNO() 320
 SVC_INFO_OS_HASH() 321
 SVC_INFO_PARTNO() 323
 SVC_INFO_PIN_PAD() 324
 SVC_INFO_PORT_MODEM() 325
 SVC_INFO_PRNTR() 326

SVC_INFO_PTID() 336
 SVC_INFO_RESET() 327
 SVC_INFO_SERLNO() 328
 SVC_PORT_IR() 324, 325
 SVC_VERSION_INFO() 336
 tasks
 _exit() 195
 get_task_id() 197
 get_task_info() 198
 run 199
 set_group() 201
 time
 date2days() 411
 datetime2seconds() 411
 days2date() 411
 read_clock() 413
 SVC_VALID_DATE() 411
 TXO. See *TXO functions*.
 VSS
 iPS_CancelPIN() 583
 iPS_CheckMasterKey() 591
 iPS_DeleteKeys() 592
 iPS_ExecuteScript() 580
 iPS_GetPINResponse() 583
 iPS_GetScriptStatus() 578
 iPS_InstallScript() 579
 iPS_LoadMasterClearKey() 593
 iPS_LoadMasterEncKey() 594
 iPS_LoadSysClearKey() 595
 iPS_LoadSysEncKey() 596
 iPS_RequestPINEntry() 585
 iPS_SelectPINAlgo() 586
 iPS_SetPINParameter() 587, 588
 iPS_UninstallScript() 581

G

gen_master_key() 697
 gen_session_key() 698
 GenerateRandom() 572
 get PIN request 695
 get_battery_sts() 56
 get_battery_value() 56
 get_bits_per_second() 414
 get_component_vars() 308
 get_console() 276
 get_dock_sts() 56
 get_env() 171, 187
 get_file_attributes() 140
 get_file_date() 135
 get_file_max() 140
 get_file_size() 134, 135
 get_font() 278
 get_font_mode() 279

get_kb_backlight() 274
 get_key() 145
 get_name() 364
 get_native_group() 197
 get_owner() 364
 get_performance_counter() 99
 get_performance_frequency() 99
 get_port_status() 81, 82, 433
 get_rkl_krd_cert() 597
 get_SIM_slot() 729
 get_task_id() 197
 get_task_info() 198
 getcontrast() 270, 271, 272
 getfont() 272
 getgrid() 273
 getinverse() 273
 getkey() 143, 145

GID

password 310
 see also, *groups* 752

GISKE

key attribute 624
 KLK key loads 625

global variables

device names 362

gotoxy() 281

GPS 732

Group 1 107

group password compare 310

groups

Group 1 752
 Group 15 752
 public 752

H

host key reset 702

host key update 702

hot key 265

hot key status 280

HSPA+ 732

I

infrared device 324, 325

insert(), insert_vlr(), and insert_cvlr() 132

insufficient buffers 213

int 40

interrupts 411

inverse_toggle() 283

IPP

key attributes 624
 key length 624
 key version 624

IPP7 624

DUKPT modes 665

iPS_CancelPIN() 583

iPS_CheckMasterKey() 591

iPS_DeleteKeys() 592

iPS_ExecuteScript() 580

iPS_GetPINResponse() 583

iPS_GetScriptStatus() 578

iPS_InstallScript() 579

iPS_LoadMasterClearKey() 593

iPS_LoadMasterEncKey() 594

iPS_LoadSysClearKey() 595

iPS_LoadSysEncKey() 596

iPS_RequestPINEntry() 585

iPS_SelectPINAlgo() 586

iPS_SetPINParameter() 587, 588

iPS_UninstallScript() 581

isAttacked() 572

K

key 131, 625

beep 418

delete 146

maximum length 144

key attributes, IPP 624

key certificate 754

key FIFO 253

key loading 590

keyed file functions. See *functions*, *keyed files*

keyed files 110, 144

rules 144

key-loading program 699

keypad

ALPHA key use 244

color coded function keys 241

keys 244

APACS40 699

auto-repeating 251

color coded function 241

current 699

current, reset 702

host 700, 701, 702

hot key 192

MAC 700

programmable function keys 78

seed 699, 702

session 698

keys and file authentication 754

key-value pairs 74

KLK (GISKE) 625

L

load security script keys 590

lock files 194

lock() 138

lock_kbd() 287

LRC, longitudinal redundancy check 337

lseek() 113

lseek(), seek_vlr(), and seek_cvlr() 131

ltoa() 706

Luhn check 338

M

MAC 700

MAC value 626

MAG_BADJS 121

MAG_BADLRC 121

MAG_BADTRK 121

MAG_NODATA 121

MAG_NOERR 121

MAG_NOETX 121

MAG_NOSTX 121

MAG_PARITY 121

MAG_REVETX 121

magnetic card reader 367

magnetic card reader functions. *See functions, magnetic card reader*

magprt_mode_control() 372

main task 189

management information block (MIB) 253

manufacturing block 315

master key 694, 697

Master/Session 605

master-session key management 694

measurement conventions 27

memory

flash

insufficient FLASH error 753

RAM

insufficient RAM error 753, 767

memory management unit (MMU) 66

memory_access() 72

MIB 253

model number 320

modem

determine port number 325

escape to command mode 184

handshake 506

initialization string setting variable 183

Modem Sleep State Functions 511

modes

character 494

dot graphic mode 437

double width mode 437

SDLC 507

system mode 78

system mode entry 250, 750

Modulo 8 information packets 506

multitasking 189

N

new file 112

normal beep 418

normal_tone() 421

O

On 795

One Way function 701

operating system

version information 336

Opn_Blk 493

P

.p7s file *See file authentication*

Packet A90 684

packet format

Modulo-128 506

Modulo-8 506

supervisory 506

unnumbered information 506

packet formats 506

part number 323

password 310, 767

passwords 600

peek_event () 226

PIN block 695

PIN entry functions 582

PIN pad 324

external 693

PINs 694

pipe 189

pipe_connect() 216

pipe_init_char() 214, 218

pipe_init_msg () 52, 200, 205, 206, 207, 208, 209, 210, 211, 212, 214

pipe_init_msg() 218, 219

pipe_pending() 219

pipes 213

character 213

function calls for configuring 214

handles 216

maximum number 213

message 213

play 419

play_RTTTL() 419

port number 325

ports

COM for SDLC 506

COM2 497

external PIN pad (COM2) 497

RS-232 497

serial 494

post_user_event () 227

printer

- determine type installed 326
- protocol-specific parameters 493
- put_BMP_at() 290
- put_env() 171, 188
- put_file_date() 156
- put_graphic() 290
- putkey() 146

R**RAM**

- error messages
 - insufficient RAM 753, 767
- non-volatile 106
- read_clock() 413
- read_event() 227
- read_evt 228
- read_RTC() 415
- read_user_event () 228
- ready-to-receive packets 506
- real-time clock 387
- reset_file_attributes() 141
- Returns 273
- RS-232
 - signals 507
- RS-232C compliance 497
- rsa_calc() 574
- run() 199
- run_thread() 200

S

- screen_size() 292
- SDLC initialization 506
- SDLC protocol 506
 - IBM packet formats 506
 - Modulo 8 information packets 506
 - unnumbered information packet formats 506
- secret key 754
- secs2time() 411
- security 754
- security libraries 563
 - PIN entry functions
 - CancelPIN 583
 - GetPINResponse 583
 - RequestPINEntry 585
 - SelectPINAlgo 586
 - SetPINParameter 588
 - VeriShield security scripts functions
 - ExecuteScript 580
 - GetScriptStatus 578
- security scripts
 - key management 605
 - DUKPT 605
- seed key 699
- seek 113

- seek pointer 113, 116, 129, 130, 133
- seek_cvlr() 113
- seek_vlr() 113
- Selects 295
- sem_init() 167, 168, 169, 170, 205
- sem_post() 207
- sem_prop() 209
- sem_wait() 207
- Semaphores 202
- serial number 328
- session key 695
- session keys 698
- set_backlight() 276, 277, 278, 296, 297, 298
- set_battery_value() 100
- set_cursor() 297
- set_enet_rx_control() 798
- set_errno_ptr() 201
- set_file_attributes() 142
- set_file_max() 142
- set_font() 299, 300
- set_group() 201
- set_gsm_break() 786
- set_gsm_powersave() 786
- set_hot_key() 292, 301
- set_kb_backlight() 293
- set_owner() 366
- set_owner_all() 366
- set_SIM_slot() 729
- set_timer () 229
- setinverse() 295
- shm_close() 71
- shm_open() 69
- signature file 754
- smart card functions. *See functions, smart card*
- smart card reader
 - ICC socket locations 387
- smart card support 387
- start_battery_conditioner() 81
- Startup Display 691
- string conversion functions. *See functions, string conversions.*
- string conversions 704
- strnlwr() 707
- strnupr() 707
- sts_kbd_lock() 301
- supervisory packet format 506
- SVC_2INT() 714
- SVC_AZ2CS 713
- SVC_CHECKFILE() 136, 173
- SVC_CHK_FIFO() 342
- SVC_CHK_PASSWORD() 310
- SVC_CLR_FIFO 342

SVC_CRC_CALC() 347
 SVC_CRC_CALC_L() 347
 SVC_CRC_CCITT_L() 348
 SVC_CRC_CCITT_M() 348
 SVC_CRC_CRC16_L() 349
 SVC_CRC_CRC16_M() 350
 SVC_CRC_CRC32_L() 351
 SVC_DSP_2_HEX() 709
 SVC_FLASH_SIZE() 160
 SVC_GET_FIFO() 343
 SVC_HEX_2_DSP() 708
 SVC_INFO_CIB_ID() 353
 SVC_INFO_CIB_VER() 353
 SVC_INFO_COUNTRY() 311
 SVC_INFO_COUNTRY_EXT() 311
 SVC_INFO_CRASH() 334
 SVC_INFO_DISPLAY 302
 SVC_INFO_DISPLAY() 302
 SVC_INFO_EPROM() 335
 SVC_INFO_HW_VERS() 312
 SVC_INFO_HW_VERS_EXT() 312
 SVC_INFO_IR() 324, 325
 SVC_INFO_KBD() 303
 SVC_INFO_KEYBRD_TYPE() 313
 SVC_INFO_LIFETIME() 333
 SVC_INFO_LOTNO 313, 314
 SVC_INFO_LOTNO_EXT() 314
 SVC_INFO_MFG_BLK() 315
 SVC_INFO_MFG_BLK_EXT() 316
 SVC_INFO_MOD_ID() 316
 SVC_INFO_MODELNO() 320
 SVC_INFO_MODELNO_EXT() 320
 SVC_INFO_MODEM_TYPE() 321
 SVC_INFO_OS_HASH() 321
 SVC_INFO_OS_HMAC_SHA1() 322
 SVC_INFO_OS_HMAC_SHA256() 322
 SVC_INFO_PARTNO 323
 SVC_INFO_PARTNO_EXT() 323
 SVC_INFO_PIN_PAD() 324
 SVC_INFO_PORT_MODEM() 325
 SVC_INFO_PRNTR() 326
 SVC_INFO_PTID() 336
 SVC_INFO_READ_MIB() 326
 SVC_INFO_RELEASED_OS() 327
 SVC_INFO_RESET() 327
 SVC_INFO_SBI_VER() 328
 SVC_INFO_SERLNO() 328
 SVC_INFO_SERLNO_EXT() 329
 SVC_INFO_URT0_TYPE() 329
 SVC_INFO_URT1_TYPE() 330
 SVC_INFO_URT2_TYPE() 331
 SVC_INFO_URT3_TYPE() 331

SVC_INT2() 714
 SVC_LEDS() 332
 SVC_LRC_CALC() 337
 SVC_MEMSUM() 337
 SVC_MOD_CK() 338
 SVC_PACK4() 710
 SVC_PUT_FIFO() 343
 SVC_READ_FIFO() 344
 SVC_RESTART 109, 220
 SVC_UNPK4() 711
 SVC_VALID_DATE 411
 SVC_VALID_DATE() 411
 SVC_VERSION_INFO() 336
 SVC_WAIT() 232
 SVC_WRITE_FIFO() 344
 SVC_ZONTALK() 514
 system
 date setting 122
 day setting 122
 library 362
 time setting 122
 timing 411
 system devices 361
 access 361
 system information functions. *See functions, system information.*
 system mode 78
 downloads from 750
 entry 250, 750
 local and remote operations 599

T

task ID 703
 tasks 189, 194
 end a task 190
 hot key 192
 locking files during use 194
 one starts another 220
 TCP/IP Module 727
 terminal
 ASCII table for display 807
 identification number 336
 last reset date 327
 life span determination 333
 password 600
 serial number 328
 verify status 600
 terminal management 253
 terminal reset 109
 terminal security 754
 test_master_key() 698
 Threads 202
 time setting 122

time2secs() 411
 timers, cancelling 226
 timestamp 135
 Toggles 283
 traps 65

TXO functions

getscrollmode 274
 inline 283
 kbd_pending_count() 285
 kbd_pending_test() 286
 putpixelcol() 287
 resetdisplay() 291
 SVC_INFO_KBD() 303
 wherecur() 303
 wherewin() 304
 wherewincur() 304
 window() 305

U

ultoa() 706
 Unique Key 683
 unlock() 138
 unnumbered information packet formats 506
 USB 795

device functions

get_usb_device_bits() 524
 External Serial (COM6) 521
 Flash Drive 795
 Internal WiFi (WLN1) 525, 526

USB_COM2_RESET() 103

utilities

strings 704

V

variable-length record (VLR) 74, 109, 111, 124, 129, 132

variables

*GO 107
 global 362

VeriCentre 183, 758

VeriShield 182, 754

VeriShield security script functions 576

version, OS 336

VLR. See *variable-length record*.

VSP_Crypto() 380

VSP_Init() 384

VSP_Passthru() 384

VSP_Reset() 385

VSP_Result() 385

VSP_Xeq() 386

VTM ADE Status Menu 691

VX 680 3G 729

W

wait_event () 233

wait_evt() 234

wherecur() 303

wherewin() 304

wherewincur() 304

WiFi Module 793

window() 305

write_at() 305

Z

ZonTalk

CONFIG.SYS *Z series variables 183



VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA.

1-800-VeriFone
www.verifone.com

Verix eVo Volume I: Operating System

Programmers Manual

