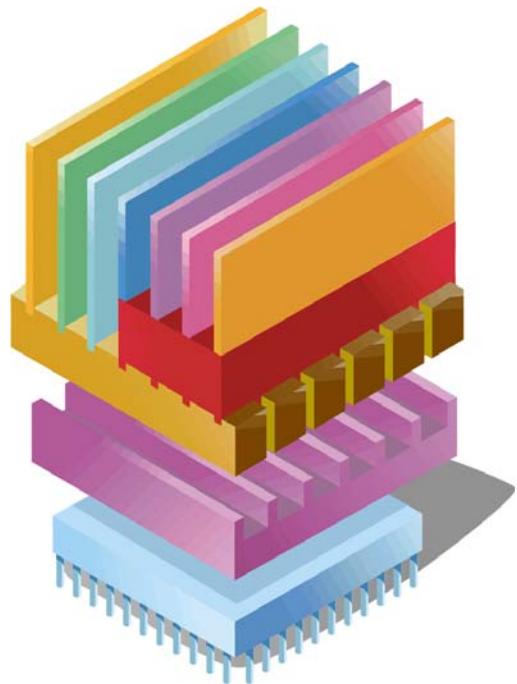




# Verix® V Operating System

*Programmers Manual*

For Vx Solutions



Verix® V Operating System Programmers Manual  
© 2009 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, Omni, VeriCentre, Verix, 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](http://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](http://www.verifone.com)

VeriFone Part Number 23230, Revision L



## CONTENTS

### CHAPTER 1 Programmers Quick Reference

PREFACE .....	21
Organization .....	21
Audience .....	22
Assumptions About the Reader .....	22
Conventions and Acronyms .....	23
Conventions .....	23
Acronyms .....	24
Related Documentation .....	26
Function Calls .....	29
Function Call Error Codes.....	47
DBMON Abort Codes .....	48
Event Codes Returned by wait_event() .....	49
Managing Application Data—Effective Use of Memory Space .....	50
Communications Buffer Space .....	50
Memory Management.....	50
Erase Flash.....	51
Keypad .....	51
Printer Control Codes .....	53

### CHAPTER 2 Application Programming Environment

Hardware .....	57
General Principles .....	57
Virtual Memory .....	58
First-in First-out (FIFO) Buffers.....	60
Memory Management Unit (MMU) .....	60
Virtual Memory Map .....	60
File System .....	61
File Locations .....	61
File Groups .....	61
Record-Structured Files .....	62
CONFIG.SYS File.....	62
Power-fail File Protection .....	62
Handles.....	62
Device APIs .....	62
Other Devices .....	64
Console.....	64
System Mode .....	64
Customizable Application Launcher .....	65
System Mode Battery Support .....	66
Battery Conditioner.....	68
Timers .....	69

<b>CHAPTER 3</b>		
<b>File Management</b>		
Verix V File Systems . . . . .		71
File Conventions . . . . .		71
File Storage . . . . .		71
Filenames . . . . .		72
File Groups . . . . .		72
Default System Files . . . . .		72
.out Files . . . . .		73
File Handles . . . . .		73
Generic Files . . . . .		73
Variable-Length Record (VLR) Files . . . . .		74
Compressed Variable-Length Record (CVLR) Files . . . . .		74
Keyed Files . . . . .		74
Verix V File Manager . . . . .		75
Support for File Groups . . . . .		76
Support for File Extension Areas . . . . .		76
Variable-Length Records . . . . .		81
Compressed Variable-Length Records . . . . .		81
File Access Function Calls . . . . .		82
Open Files . . . . .		82
open() . . . . .		84
Read Files . . . . .		86
read(), read_vlr(), and read_cvlr() . . . . .		87
Write Files . . . . .		88
write(), write_vlr(), and write_cvlr() . . . . .		89
File Positioning . . . . .		90
lseek(), seek_vlr(), and seek_cvlr() . . . . .		91
Insert and Delete Data . . . . .		93
insert(), insert_vlr(), and insert_cvlr() . . . . .		94
delete(), delete_vlr(), and delete_cvlr() . . . . .		95
delete_() . . . . .		96
Retrieve File Information . . . . .		97
get_file_size() . . . . .		98
get_file_date() . . . . .		99
SVC_CHECKFILE() . . . . .		100
Delete a File . . . . .		101
_remove() . . . . .		102
Lock and Unlock Files . . . . .		103
lock() . . . . .		104
unlock() . . . . .		105
Rename a File . . . . .		106
_rename() . . . . .		107
File Attribute Function Calls . . . . .		108
get_file_attributes() . . . . .		109
get_file_max() . . . . .		110
reset_file_attributes() . . . . .		111
set_file_attributes() . . . . .		112
set_file_max() . . . . .		113
Close Files . . . . .		114
close() . . . . .		115
Keyed File Reads and Writes . . . . .		116
Keyed File Function Calls . . . . .		117

## CHAPTER 4

### System Configuration File

getkey()	118
putkey()	119
File Directory Function Calls	120
dir_get_attributes()	121
dir_get_file_date()	122
dir_get_file_size()	123
dir_get_file_sz()	124
dir_get_first()	125
dir_get_next()	126
dir_get_sizes()	127
dir_put_file_date()	128
dir_reset_attributes()	129
dir_set_attributes()	130
file_copy()	131
SVC_RAM_SIZE()	132
unzip()	133
Flash File System	134
Restrictions on Flash Files	134
Open and Close	136
Flash Rebuild	136
Flash Erase	136
dir_flash_coalesce()	137
dir_flash_coalesce_size()	138
SVC_FLASH_SIZE()	139

Environment Variable Descriptions	142
Device Variables	146
Environment Variables Application Information	147
*ARG—Arguments	147
*B—Communication Device Buffers	147
*BCM	148
CHKSUM—Checksum Control	148
*COMBO	148
*COM2HW	148
*COM3HW	149
*COM2RB	149
*COM3RB	149
*DARK	149
*DBMON	149
*DEFRAG—Defragment Flash	149
*DIAG	150
*DOT0	150
*DOT1	150
*FA—File Authentication	150
*FILE	150
*GO—Startup Executable Code File	150
*GKE	151
*IPPMKI—Internal PIN Pad Communications Parameters	151
*MA	151
*MERR	152
*MN	152

*OFF . . . . .	152
*OFFD . . . . .	153
*PIPE . . . . .	153
*POW . . . . .	153
*PRTFNT . . . . .	153
*PRTLGO . . . . .	153
*PW—Password . . . . .	153
*SMDL—System Mode Download . . . . .	153
*SMPW—System Mode Password . . . . .	154
*SYSCHK . . . . .	154
*TIME—Set Timers . . . . .	154
*UNZIP—Decompress .ZIP . . . . .	154
*USBMEM . . . . .	154
UNZIP—Determine Decompress Results . . . . .	154
*VALID—List Groups to Search . . . . .	154
*Z Series—ZonTalk 2000 Control . . . . .	155
*ZB . . . . .	155
*ZINIT . . . . .	155
*ZRESP . . . . .	155
*ZSWESC . . . . .	155
*ZTCP . . . . .	155
*ZX . . . . .	155
Search/Update CONFIG.SYS . . . . .	155
get_env() . . . . .	157
put_env() . . . . .	158

## CHAPTER 5 Multitasking

Verix V Application Architecture . . . . .	159
Tasks . . . . .	159
Task Startup . . . . .	159
Task Termination . . . . .	160
Device Ownership . . . . .	160
Sharing the Console . . . . .	161
File Sharing and File Locking . . . . .	161
Task Function Calls . . . . .	162
_exit() . . . . .	163
get_group() . . . . .	164
get_task_id() . . . . .	165
get_task_info() . . . . .	166
run() . . . . .	167
run_thread() . . . . .	168
set_group() . . . . .	169
Application Threads and Semaphores . . . . .	170
Semaphore Example with Deterministic Scheduling . . . . .	171
Semaphore Example with User Events . . . . .	171
Semaphore Application Function Calls . . . . .	171
_exit() . . . . .	172
open() . . . . .	173
sem_init() . . . . .	174
sem_wait() . . . . .	175
sem_post() . . . . .	176
Pipes . . . . .	177

Pipe Interface . . . . .	178
Configure the Pipe . . . . .	178
Pipe Function Calls . . . . .	179
close() . . . . .	180
open() . . . . .	181
pipe_connect() . . . . .	182
pipe_init_char() . . . . .	184
pipe_init_msg() . . . . .	185
pipe_pending() . . . . .	186
read() . . . . .	187
write() . . . . .	188
Restart Capability . . . . .	189
SVC_RESTART . . . . .	190
<b>CHAPTER 6</b>	
<b>Event Handling</b>	
Event Function Calls . . . . .	193
clr_timer() . . . . .	194
peek_event() . . . . .	195
post_user_event() . . . . .	196
read_event() . . . . .	197
read_evt() . . . . .	198
read_user_event() . . . . .	199
set_timer() . . . . .	200
set_signal_events() . . . . .	201
SVC_WAIT() . . . . .	202
wait_event() . . . . .	203
wait_evt() . . . . .	204
Case Removal Latch on Vx700 PIN Pad . . . . .	205
get_latch_status() . . . . .	206
reset_latch() . . . . .	207
set_activation_hash() . . . . .	208
<b>CHAPTER 7</b>	
<b>Console Device</b>	
Display . . . . .	209
Default Font . . . . .	209
Font Files . . . . .	210
Big Font Files . . . . .	210
Backward Compatibility Mode . . . . .	211
set_bcm() . . . . .	212
Keypad . . . . .	213
Vx700 Keypad . . . . .	215
Keypress Scan Codes . . . . .	216
Alpha Key Support . . . . .	217
CELL_PHONE Mode on Vx810 . . . . .	221
setAlphaMode() . . . . .	223
getAlphaMode() . . . . .	224
alpha_multi_shift() . . . . .	225
Dual Keypress . . . . .	226
Hidden Function Support . . . . .	226
Enter System Mode . . . . .	226
Auto-Repeating Keys . . . . .	227
Console Ownership . . . . .	228

Management Information Block (MIB) . . . . .	228
Console Function Calls. . . . .	228
activate_task() . . . . .	229
alpha_shift() . . . . .	230
close() . . . . .	231
clreol() . . . . .	232
clrscr() . . . . .	233
contrast_down() . . . . .	234
contrast_up() . . . . .	235
delline() . . . . .	236
disable_hot_key() . . . . .	237
disable_key_beeps() . . . . .	238
enable_hot_key() . . . . .	239
enable_key_beeps() . . . . .	240
getcontrast() . . . . .	241
getfont() . . . . .	242
getgrid() . . . . .	243
getinverse() . . . . .	244
getscrollmode() . . . . .	245
get_console() . . . . .	246
get_font() . . . . .	247
get_font_mode() . . . . .	248
get_hot_key_sts() . . . . .	249
gotoxy() . . . . .	250
insline() . . . . .	251
inverse_toggle() . . . . .	252
kbd_pending_count() . . . . .	253
kbd_pending_test() . . . . .	254
key_beeps() . . . . .	255
open() . . . . .	256
putpixelcol() . . . . .	257
put_graphic() . . . . .	258
read() . . . . .	259
resetdisplay() . . . . .	260
screen_size() . . . . .	261
setcontrast() . . . . .	262
setfont() . . . . .	263
setinverse() . . . . .	264
setscrollmode() . . . . .	265
set_backlight() . . . . .	266
set_cursor() . . . . .	267
set_font() . . . . .	268
set_hot_key() . . . . .	269
SVC_INFO_DISPLAY() . . . . .	270
SVC_INFO_KBD() . . . . .	271
wherectr() . . . . .	272
wherewin() . . . . .	273
wherewincur() . . . . .	274
window() . . . . .	275
write() . . . . .	276
write_at() . . . . .	277

## CHAPTER 8

### Service Function Calls

get_component_vars()	280
set_combo_mode()	281
SVC_CHK_PASSWORD()	282
SVC_FLASH_SIZE()	283
SVC_INFO_COUNTRY()	284
SVC_INFO_CRASH()	285
SVC_INFO_DISPLAY()	286
SVC_INFO_EPROM()	287
SVC_INFO_HW_VERS()	288
SVC_INFO_KBD()	289
SVC_INFO_LIFETIME()	290
SVC_INFO_LOTNO()	291
SVC_INFO_MAG()	292
SVC_INFO_MFG_BLK()	293
SVC_INFO_MOD_ID()	294
SVC_INFO_MODULE_ID()	295
SVC_INFO_MODELNO()	297
SVC_INFO_PARTNO()	298
SVC_INFO_PIN_PAD()	299
SVC_INFO_PORT_IR()	300
SVC_INFO_PORT_MODEM()	301
SVC_INFO_PRNTR()	302
SVC_INFO_PTID()	303
SVC_INFO_RESET()	304
SVC_INFO_SERLNO()	305
SVC_SHUTDOWN()	306
SVC_LED()	307
SVC_RAM_SIZE()	308
SVC_VERSION_INFO()	309
SVC_INFO_OS_HASH()	310
FIFOs	311
SVC_CHK_FIFO()	312
SVC_CLR_FIFO()	313
SVC_GET_FIFO()	314
SVC_PUT_FIFO()	315
SVC_READ_FIFO()	316
SVC_WRITE_FIFO()	317
CRCs	318
CRC Function Calls	319
SVC_CRC_CALC()	320
SVC_CRC_CALC_L()	321
SVC_CRC_CCITT_L()	322
SVC_CRC_CCITT_M()	323
SVC_CRC_CRC16_L()	324
SVC_CRC_CRC16_M()	325
SVC_CRC_CRC32_L()	326
SVC_LRC_CALC()	327
SVC_MEMSUM()	328
SVC_MOD_CK()	329

<b>CHAPTER 9</b>		
<b>System Devices</b>		
Device Management Function Calls . . . . .	333	
get_name() . . . . .	334	
get_owner() . . . . .	335	
set_owner() . . . . .	336	
Magnetic Card Reader . . . . .	337	
No Data Characters on Track 3 MSR . . . . .	337	
Increased Buffer Size . . . . .	338	
Magnetic Card Reader Function Calls . . . . .	338	
card_pending() . . . . .	339	
close() . . . . .	340	
open() . . . . .	341	
read() . . . . .	342	
SVC_INFO_MAG() . . . . .	345	
VeriShield Protect (VSP) . . . . .	346	
VSP Encryption . . . . .	346	
Verix V Implementation . . . . .	347	
Data Formats . . . . .	349	
Internal OS Calls . . . . .	350	
System Mode Menu . . . . .	351	
Installation Sequence . . . . .	353	
VSP_Status() . . . . .	354	
VSP_Disable() . . . . .	355	
VSP_Enable() . . . . .	356	
VSP_Encrypt_MSR() . . . . .	357	
VSP_Encrypt_KBD() . . . . .	358	
VSP_Decrypt_PAN() . . . . .	359	
VSP_Decrypt_MSR() . . . . .	360	
VSP_Status_Ex() . . . . .	361	
Smart Card Reader . . . . .	362	
ICC Socket Locations . . . . .	362	
Real-Time Clock . . . . .	363	
Related Clock Function Calls . . . . .	363	
Real-Time Clock Function Calls . . . . .	365	
close() . . . . .	366	
open() . . . . .	367	
read() . . . . .	368	
read_clock() . . . . .	369	
write() . . . . .	370	
Timer Function Calls . . . . .	371	
read_ticks() . . . . .	372	
SVC_WAIT() . . . . .	373	
Beeper . . . . .	374	
Beeper Function Calls . . . . .	374	
play_RTTTL() . . . . .	375	
beeper_off() . . . . .	376	
close() . . . . .	377	
error_tone() . . . . .	378	
key_beeps() . . . . .	379	
normal_tone() . . . . .	380	
open() . . . . .	381	
sound() . . . . .	382	

Internal Printer . . . . .	384
Internal Printer Function Calls . . . . .	384
open() . . . . .	385
read() . . . . .	386
write() . . . . .	387
close() . . . . .	388
get_opn_blk() . . . . .	389
set_opn_blk() . . . . .	390
get_port_status() . . . . .	391
reset_port_error() . . . . .	393
set_serial_lines() . . . . .	394
set_fifo_config() . . . . .	395
get_fifo_config() . . . . .	396
set_fc_config() . . . . .	397
get_fc_config() . . . . .	398
Special Items . . . . .	399
<ESC>a<n>; . . . . .	404
<ESC>b<n>; . . . . .	405
<ESC>c . . . . .	406
<ESC>d . . . . .	407
<ESC>e<n>; . . . . .	408
<ESC>f<n>; . . . . .	409
<ESC>g . . . . .	410
<ESC>h<n>; . . . . .	411
<ESC>i . . . . .	412
<ESC>s . . . . .	413
<ESC>F<n>; . . . . .	414
<ESC>l<s><t>; . . . . .	415
<ESC>m<c><r1>...<rn>; . . . . .	417
<ESC>H<hh1>...<hhn>; . . . . .	422
<ESC>GL<f>,<t>,<w>,<h>,<b1>...<bn>; . . . . .	423
<ESC>GP<t>[,<m>]; . . . . .	425
<ESC>w<n>; . . . . .	426
<ESC>p<n0>,<n1>; . . . . .	427
<ESC>CS; . . . . .	428
Dot Graphics Mode . . . . .	429
Download Fonts and Logos . . . . .	432
Support for Paper-out LED . . . . .	434
SDIO . . . . .	434
SDIO Device Firmware Download . . . . .	435
SDIO API . . . . .	437
reset_ctls() . . . . .	440
set_event_bit() . . . . .	441
get_event_bit() . . . . .	443
get_sd_device_bits() . . . . .	444
USB Fingerprint Reader Device . . . . .	445
USB Low Layer Protocol . . . . .	445
MorphoSmart ILV Functions . . . . .	446
ILV Get_Descriptor . . . . .	451
Asynchronous Messages . . . . .	452
Types of Asynchronous Messages . . . . .	452
Image . . . . .	454

Image Header . . . . .	455
USB Barcode Scanner . . . . .	456
Configuring D130 as USB-COM . . . . .	456
Enabled Codes on Heron D130 Device . . . . .	458
Operating Test . . . . .	458
USB Keyboard . . . . .	459
USB to RS-232 Converter . . . . .	459
USB Device Bits . . . . .	460
Power . . . . .	460
USB Keyboard Scan Codes . . . . .	461
Support for Windows Keys . . . . .	464
Calculating Make and Break Scan Codes . . . . .	464
Metrologic Barcode Scanner . . . . .	465
Configuring the Barcode Scanner . . . . .	465
Keyboard Driver Data Output . . . . .	465
Metrologic USB Barcode Scanner API . . . . .	466
Processing Events . . . . .	467
Device Bits . . . . .	467
Power . . . . .	467
Operating Test . . . . .	467
USB Device Driver APIs . . . . .	469
open() . . . . .	470
read() . . . . .	471
write() . . . . .	472
close() . . . . .	473
USB Device Driver Events . . . . .	474
set_event_bit() . . . . .	475
get_event_bit() . . . . .	477
MC5727 USB Driver . . . . .	478
set_radio_ctl() . . . . .	479
get_radio_ctl() . . . . .	480
Data Interface (COM2) . . . . .	481
Control Interface (COM9) . . . . .	481
Debugging (COM10) . . . . .	481
Events . . . . .	481
set_event_bit () . . . . .	482
get_event_bit() . . . . .	483
System Mode . . . . .	484
 CHAPTER 10	
Smart Card API	
Smart Card API Function Calls . . . . .	487
PCI PED Requirement . . . . .	487
decrypt_session_data() . . . . .	488
gen_master_key() . . . . .	489
gen_session_key() . . . . .	491
test_master_key() . . . . .	492
Administrative Services . . . . .	493
Syntax . . . . .	493
Retrieve IFD Capability . . . . .	493
Set IFD Capability . . . . .	494
Protocol Information and Negotiation . . . . .	494
ICC Power Management . . . . .	495

## CHAPTER 11

### Communications

Mechanical Characteristics . . . . .	496
Communication Services . . . . .	497
ICC Insertion and Removal . . . . .	498
Enumeration of the Tags . . . . .	499
Enumeration of the Device Capabilities . . . . .	499
ICC Interface Management . . . . .	501
Protocol Support . . . . .	502
Specific Features for the Smart Card API . . . . .	504
Smart Card Code Examples . . . . .	509
Asynchronous Cards . . . . .	509
New Tags Added for Multi-Application Support . . . . .	510
Common Function Calls . . . . .	510
The Opn_Blk() Structure . . . . .	511
Character Mode . . . . .	511
Character Mode Initialization . . . . .	511
Communication Ports . . . . .	511
RS-232 Serial Port (COM1 and COM2) . . . . .	512
Communication Port Flow Control (COM1 and COM3) . . . . .	512
Selecting Non-Standard Bit Rates (COM1 and COM2) . . . . .	513
Determining Actual Bit Rate (COM1 and COM2) . . . . .	513
USB Dongles (COM3 and COM6) . . . . .	513
RS-232 Serial Port (COM1) . . . . .	515
External PIN Pad Serial Port (COM2) . . . . .	516
get_battery_value() . . . . .	520
Modem Port (COM3) . . . . .	521
Conexant Modems (Banshee and Eisenhower) . . . . .	521
Silicon Laboratories Modem (Si24xx) . . . . .	522
Asynchronous . . . . .	523
Synchronous . . . . .	524
SDLC Protocol . . . . .	524
Enhanced SDLC Protocol . . . . .	526
SDLC Status . . . . .	526
Country Profiles . . . . .	526
Combined Modem Profiles . . . . .	530
Modem Functions . . . . .	530
SVC_ZONTALK() . . . . .	531
Serial Printer Port (COM4) . . . . .	534
C Code Applet for COM4 Driver . . . . .	534
Printer Functions . . . . .	534
get_fc_config() . . . . .	535
set_fc_config() . . . . .	536
Internal PIN Pad Port (COM5) . . . . .	539
Files Used to Compile Driver . . . . .	539
Clear IPP Keys Upon Certificate Tree Removal . . . . .	540
IPP Function Calls . . . . .	540
open() . . . . .	541
read() . . . . .	542
write() . . . . .	543
close() . . . . .	544
set_opn_blk() . . . . .	545

get_opn_blk()	546
get_port_status()	547
reset_port_error()	548
set_serial_lines()	549
select_pipad()	550
IPP_power()	551
set_fifo_config()	552
get_fifo_config()	553
TerminatePinEntry()	554
PINentryStatus()	555
SVC_INFO_PIN_PAD()	556
USB External Serial (COM6)	557
USB Secure Mode	558
open()	559
close()	560
set_usb_multi_device()	561
get_usb_device_bits()	562
USB External Ethernet (ETH1)	563
USB Internal WiFi (WLN1)	563
9-Bit Serial Interface	563
MDB Physical Interface	563
MDB Dongle Version Number	563
9-Bit API	563
write_9bit()	565
read_9bit()	566
Bluetooth Modem Support	567
Modem Profile Loading	568
BT_Si2434_profile_load()	569
SDLC and V.80 Support	570
SDLC Packet Posting	571
Bluetooth Firmware Update	571
General Communication Device Functions	571
close()	572
download()	573
get_component_vars()	574
get_fifo_config()	575
get_opn_blk()	576
get_port_status()	577
open()	579
read()	580
reset_port_error()	581
set_fifo_config()	582
set_opn_blk()	583
set_serial_lines()	584
write()	586
Security Services Functions	588
File Encryption Support Functions	588
crypto_read()	590
crypto_write()	591
Crypto Functions	593

## CHAPTER 12

### Security/Crypto Library

AES()	594
DES()	595
GenerateRandom()	596
isAttacked()	597
rsa_calc()	598
SHA1()	599
VeriShield Security Script Functions	600
iPS_GetScriptStatus()	601
iPS_InstallScript()	602
iPS_ExecuteScript()	603
iPS_UninstallScript()	604
VSS PIN Entry Functions	605
iPS_CancelPIN()	606
iPS_GetPINResponse()	607
iPS_RequestPINEntry()	609
iPS_SelectPINAlgo()	610
iPS_SetPINParameter()	611
Key Loading Functions	612
iPS_CheckMasterKey()	613
iPS_DeleteKeys()	614
iPS_LoadMasterClearKey()	615
iPS_LoadMasterEncKey()	616
iPS_LoadSysClearKey()	617
iPS_LoadSysEncKey()	618

## APPENDIX A System Mode

When to Use System Mode	577
Local and Remote Operations	577
Verifying Terminal Status	578
Entering System Mode	578
File Integrity Check	579
System Error Handler	580
Developer Download	580
Resumable Download	580
IP/Dial Download	580
Start-up Screens	581
Default Certificate	581
Error Detection Screen	581
Interactive System Mode Entry at Start-Up	583
ZIP File Support	583
VeriShield File Authentication Support	583
Automatic Flash Coalesce	583
*GO and *ARG Processing	583
System Mode User Interface Conventions	585
Function Keys	585
Group ID	585
File Group Conventions	586
System Mode Menus	586
Eight-Line Standard Interface	587
16-Line, Four Function Keys Interface	591
Eight-Line Interface with Numeric Prompts	592
16-Line, Six Function Keys Interface	593

Four-Line Interface . . . . .	594
-------------------------------	-----

## **APPENDIX B VeriShield Security Scripts**

Verishield Security Script Implementation . . . . .	597
---	-----

## **APPENDIX C IPP Key Loading**

Data Passthrough . . . . .	599
User Interface . . . . .	599
Error Codes . . . . .	601
Master Key Protection . . . . .	601
PCI PED Enhancements . . . . .	602
Password Requirements . . . . .	602
Changing Passwords Manually . . . . .	603
Passwords Shorter than Required . . . . .	603
Passwords Longer than Required . . . . .	604
Download Password Change . . . . .	604
OS Upgrade . . . . .	605
Default Password . . . . .	605
IPP Key Load . . . . .	606
OS Expansion . . . . .	606
Upgrading to the Nine-Sector OS . . . . .	607
Hardware Specifics for Vx510 and Vx610 Units . . . . .	607

## **APPENDIX D IPP Communications Packets**

Advanced Programming in IPP . . . . .	611
Minor Differences by Packet . . . . .	611
Packets . . . . .	612
Packet Acknowledgement and Timing . . . . .	614
Encryption . . . . .	614
Constraints . . . . .	616
NAKs . . . . .	616
Time Outs . . . . .	616
Key Insertion . . . . .	616
Entering a PIN . . . . .	617
Restrict the Speed of the PIN Encryption Operation . . . . .	617
IPP7 . . . . .	617
GISKE . . . . .	617
Key Management Switching . . . . .	617
Using a Session Key . . . . .	619
Rules for Loading the Master Key (MS only) . . . . .	619
KLK . . . . .	621
3DES . . . . .	622
1DES . . . . .	623
Master Key Addressing . . . . .	623
Clear Text GISKE Key Block Loading Rule . . . . .	623
Common Packets . . . . .	624
Packet 01: Interactive Diagnostic Routine . . . . .	624
Packet 05: Transfer Serial Number . . . . .	624
Packet 06: Request PIN Pad Serial Number . . . . .	625
Packets 09 and 14: Response Packet to Packet 01 . . . . .	626
Packet 11: PIN Pad Connection Test . . . . .	631

Packets 7 and 12: Dummy Packets . . . . .	631
Packet 13: Select Baud Rate . . . . .	632
Packet 15: Set IPP Key Management Mode . . . . .	633
Packet 17: Set IPP7 Key Management Mode . . . . .	635
Packet 18: Check IPP7 Key Management Mode . . . . .	641
Packet Z60: Accept and Encrypt PIN (VISA Mode) . . . . .	645
Packet Z63: Accept and Encrypt PIN—Custom PIN Entry Requirements (VISA Mode) . . . . .	648
Packet M04: Read Permanent Unit Serial Number . . . . .	650
MS-Specific Packets . . . . .	652
Packet 02: Transfer Master Key . . . . .	652
Packet 04: Check Master Key . . . . .	655
MS Packet 08: Select a Master Key . . . . .	659
MS Packet 71: Transfer PIN Block . . . . .	660
Packet 07: Dummy Packet . . . . .	661
DUKPT-Specific Packets. . . . .	662
Packet 19: Select a DUKPT Engine . . . . .	662
Packet 25: Check the DUKPT Engine . . . . .	663
DUKPT Packet 73: Transfer PIN Block (for Packets Z60 or Z63). . . . .	664
DUKPT Packet 90: Load Initial Key Request . . . . .	666
DUKPT Packet 91: Load Initial Key Response. . . . .	667
DUKPT Packet 76: PIN Entry Test Request. . . . .	668
DUKPT Packet 71: Transfer PIN Block - (for Packet 76) . . . . .	668
DUKPT Packets 92 and 93. . . . .	670
DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request . . . . .	670
DUKPT Packet 75: DUKPT Accept and Encrypt PIN/Data Authentication Response . . . . .	671
Packet 78: DUKPT Accept and Encrypt PIN/Data Authentication Test Request . . . . .	673
MAC-Specific Packets . . . . .	674
MAC Packet Z66: Request MAC . . . . .	674
MAC Packet Z67: Return MAC. . . . .	677
Packet 72: Cancel MAC Session . . . . .	678
MAC Module Design. . . . .	678

## APPENDIX E Special Topics

Smart Card PIN Entry using an External PIN Pad . . . . .	681
Master-Session Key Management Function Calls and Smart Card PIN Entry. . . . .	682
Master/Session Functions . . . . .	683
decrypt_session_data() . . . . .	684
gen_master_key() . . . . .	685
gen_session_key() . . . . .	686
test_master_key() . . . . .	687
Support for APACS40 Cryptographic Functions . . . . .	688
Software Block Diagrams . . . . .	688
Calc_Auth_Parm() . . . . .	689
Calc_MAC() . . . . .	690
Create_MAC_Key() . . . . .	691
Init_MAC() . . . . .	692
New_Host_Key() . . . . .	693

Reset_Key()	694
Term_MAC()	695
String Utilities . . . . .	696
dtoa()	697
Itoa()	698
ultoa()	699
strnlwr()	700
strnupr()	701
SVC_HEX_2_DSP()	702
SVC_DSP_2_HEX()	703
SVC_PACK4()	704
SVC_UNPK4()	705
Counted Strings . . . . .	706
SVC_AZ2CS()	707
SVC_INT2()	708
SVC_2INT()	709
SVC_CS2AZ()	710

## APPENDIX F Download Operations

OS Download Precautions . . . . .	712
Preparing a Terminal to Accept Downloads . . . . .	713
Error Messages . . . . .	714
Back-to-Back Downloads . . . . .	716
Hardware Requirements . . . . .	716
Special Considerations . . . . .	716
Initiating a Back-To-Back Download . . . . .	718
USB Back-To-Back . . . . .	719
Logical File Transfers . . . . .	719
CONFIG.SYS Files . . . . .	720
Date, Time, and Passwords . . . . .	720
File Name Extensions and GIDs . . . . .	720
File Authentication . . . . .	720
File Authentication and Downloads . . . . .	721
Run-Time Authentication . . . . .	722
authenticate() . . . . .	723
file_copy_auth_bit() . . . . .	724
Support for Compressed Files . . . . .	725
Format for Compressed Files . . . . .	725
Application Interface to Decompression Service . . . . .	726
Verix V Support for File Groups . . . . .	726
Determine UNZIP Results . . . . .	727
User Interface . . . . .	727
Error Codes . . . . .	727
Performance . . . . .	728
Download Result Messages . . . . .	728
User Mode TCP/IP Download Support . . . . .	729
USB Flash Memory Download . . . . .	732
USB System Mode Download . . . . .	732
USB Flash Auto-Download . . . . .	734
Multiple ZIP file Downloads . . . . .	736
SD Memory Download . . . . .	738
SD Memory Read . . . . .	738

SD Memory Write . . . . .	738
Resumable Download . . . . .	738
Split File Naming Convention . . . . .	739
Combining Files . . . . .	739
CONFIG. SYS Variables . . . . .	740
Maximizing Free Flash Space . . . . .	741
Defragmenting Flash . . . . .	742
File Removal Specification Syntax . . . . .	744
Merging CONFIG.\$\$\$ into CONFIG.SYS . . . . .	745
Zip Files Inside Zip Files . . . . .	745
Automatic File Removal . . . . .	745
SVC_ZONTALK()/.download() . . . . .	746
IP Persistence . . . . .	746
Setting Up a Protected GID . . . . .	746
Deleting *GUARD variable . . . . .	746
Protecting The Application Files . . . . .	747
VeriCentre Downloads . . . . .	748
File Protection Application . . . . .	748

## APPENDIX G Power Management

Sleep Mode . . . . .	751
Hibernate . . . . .	751
Wakeup Events . . . . .	752
Power Management System . . . . .	752
Power Supply Control on Vx700 PIN Pad . . . . .	753
Printer, Battery, and Radio Interaction . . . . .	753
Application Interface . . . . .	754
Function Calls . . . . .	755

## APPENDIX H WWAN

Radio Modem Function Calls . . . . .	759
get_radio_sts() . . . . .	760
get_port_status() . . . . .	761
reset_port_error() . . . . .	762
set_radio_ctl() . . . . .	763
CDMA . . . . .	764
Hardware Interface . . . . .	764
SVC.H Symbols . . . . .	765
set_radio_ctl() . . . . .	766
get_radio_sts() . . . . .	767
Wireless Module ID EEPROM . . . . .	768
GPRS . . . . .	768
Vx670 CR GPRS . . . . .	769
Vx510 GPRS . . . . .	769
WiFi . . . . .	770

## APPENDIX I USB Support

USB Flash Drive . . . . .	771
USB Ethernet (ETH1) . . . . .	771
USB Ethernet Functions . . . . .	772
open() . . . . .	773
read() . . . . .	774

write() . . . . .	775
close() . . . . .	776
get_enet_status() . . . . .	777
get_enet_MAC() . . . . .	778
set_enet_rx_control() . . . . .	779
USB External Dial (COM3) . . . . .	780
Vx810 DUET Modem Device API . . . . .	780
USB External Serial (COM6) . . . . .	780
USB Internal WiFi (WLN1) . . . . .	781
Firmware Loading . . . . .	781
USB WiFi Functions . . . . .	783
open() . . . . .	784
read() . . . . .	785
write() . . . . .	786
close() . . . . .	787
USB_WIFI_POWER() . . . . .	788
WiFi Control and Status Management . . . . .	789
Sending PIMFOR Requests . . . . .	790
Receiving PIMFOR Responses and Traps . . . . .	790
Country/Region Configuration . . . . .	791
Management Information Block (MIB) . . . . .	791
USB Client . . . . .	792
HID Client . . . . .	792
RS-232 Client . . . . .	792
USB Client API . . . . .	792
get_usbd_status() . . . . .	794
usb_pending_out() . . . . .	795
USB Host . . . . .	796
USB Thermal Printer . . . . .	797
Font Memory . . . . .	798
Logo Memory . . . . .	798
Printer ID . . . . .	798
Firmware Version . . . . .	798
Print Buffer Management . . . . .	800
USB Device Drivers . . . . .	801
<b>APPENDIX J ASCII Table</b>	
Control Characters . . . . .	803
<b>GLOSSARY</b> . . . . .	805
<b>INDEX</b> . . . . .	807



This programmers manual supports the Verix V Development Toolkit (VVDTK) and development for the V<sup>x</sup>5xx/V<sup>x</sup>6xx transaction terminals (including V<sup>x</sup>670-based units). This manual:

- Describes the programming environment,
- Provides descriptions of the CONFIG.SYS variables,
- Provides API descriptions and code examples,
- Provides discussion on system and communication devices,
- Provides descriptions of the security features,
- Describes working with the IPP (internal PIN pad), and
- Provides information on downloading applications into a V<sup>x</sup>5xx/V<sup>x</sup>6xx/V<sup>x</sup>670-based terminal.

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

This manual also contains explicit information regarding the Application Programming Interface (API) with the Verix V 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, CONFIG.SYS variables, download procedures, and flash erasing instructions for the Verix V 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 V 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, file directory function calls, and the flash file system. |
| Chapter 4 | Describes the default system file, CONFIG.SYS.  |
| Chapter 5 | Describes the Verix V multiple application architecture.  |
| Chapter 6 | Presents an overview of the Verix V 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 V operating system and Verix V-based terminal device settings.
Chapter 9	Lists the APIs for Verix V-based terminal supported devices.
Chapter 10	Describes the function calls of the smart card API and discusses PIN entry.
Chapter 11	Lists the APIs for Verix V-based terminal supported communication devices.
Chapter 12	Lists the APIs related to security and the crypto libraries.
Appendix a	Describes the system mode operations, when to use system mode 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	Describes support for smart card PIN entry from an external PIN pad, Master/Session key management functions, APACS40 crypto functions and string utilities.
Appendix f	Provides procedures to download applications to the Verix V-based terminal.
Appendix g	Lists the power management states, function calls and variables.
Appendix h	Lists the radio modem function calls.
Appendix i	Describes support for USB host interface, function calls, application threads and semaphores.
Appendix j	Provides an ASCII table for Verix V-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 V-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 98, Windows 2000, or Windows XP.
- Has installed the VVDTK on this machine.
- Has access to V<sup>x</sup> Solutions development terminal.

## Conventions and Acronyms

### Conventions

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

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
ANSI	American National Standards Institute
APDU	Application Protocol Data Units
API	Application Program 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
BIOS	Basic Input Output System
BRK	Break
BWT	Block Waiting Time
CDC	Communications Device Class
CDMA	Code Division Multiple Access
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 <a href="#">Vvdtk</a> .
DTR	Data Terminal Ready
DUKPT	Derived Unique Key Per Transaction
EBS	European Banking Standard
EEPROM	Electrically erasable programmable read-only memory
EMV	Europay Mastercard and Visa
EOF	End-of-file
EPP	External Pin Pad
FIFO	First In, First Out
HID	Human Interface Device
ICC	Integrated Circuit Card; Smart Card

**Table 2** **Acronyms** (continued)

<b>Acronym</b>	<b>Definition</b>
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	Internal Pin Pad
ISR	Interrupt Service Routine
LAN	Local Area Network
LCD	Liquid Crystal Display
LRC	Longitudinal Redundancy Check
MAC	Message Authentication Code
MCU	Microcontroller
MDB	Multi Drop Bus
MIB	Management Information Block
MMU	Memory Management Unit
MPLA	Modem Profile Loading Application
MSAM	Micromodule-Size Security Access Module
MSO	MorpoSmart™
MSR	Magnetic Stripe Reader
MUX	Multiplexor
NMI	Nonmaskable Interrupt
OS	Operating System
OTP	One-Time Password
PCI PED	Payment Card Industry PIN Entry Devices
PED	PIN Entry Devices
PIN	Personal Identification Number
PKCS	Public Key Cryptography Standards
POS	Point-of-Sale
PSCR	Primary Smart Card Reader
PTID	Permanent Terminal Identification Number
PTS	Protocol Type Selection
RAM	Random Access Memory
RFID	Radio Frequency Identification
RFU	Reserved for Future Use
ROM	Read-only Memory
RTC	Real-Time Clock
RTTTL	Ring Tone Text Transfer Language
RTS	Request To Send
SAM	Security Access Module
SCC	Serial Communication Control

**Table 2 Acronyms** (continued)

Acronym	Definition
SCC buffer	Storage Connecting Circuit Buffer
SCR	Swipe Card Reader
SDIO	Secure Digital Input/Output
SDLC	Synchronous Data Link Control
SMS	Small Message Service
SRAM	Static Random-access Memory
TCB	Task Control Block
UART	Universal Asynchronous Receiver Transmitter
UPT	Unattended Payment Terminal
USB	Universal Serial Bus
VLR	Variable-length Record
VPN	VeriFone Part Number
VSS	VeriShield Secure Script
VVDTK	Verix V Development Toolkit
WTX	Workstation Technology Extended
WWAN	Wireless Wide Area Network

## Related Documentation

To learn more about the V<sup>x</sup> Solutions, refer to the following set of documents:

- *Omni 5xx/V<sup>x</sup>5xx/V<sup>x</sup>610 Installation Guide*, VPN - 23216.
- *V<sup>x</sup>5xx/V<sup>x</sup>6xx Certifications and Regulations*, VPN - 23218.
- *Verix V Tools Programming Reference Manual*, VPN - 23231.
- *Verix Operating System Programmers Manual*, VPN - 19733.
- *V<sup>x</sup>510 Quick Installation Guide*, VPN - 23219.
- *Verix Theory of Operation, Functional Specifications and Internal Design*, VPN - 22826.
- *Verix V Enhancements for V<sup>x</sup>570 (Predator P3/P4) MS300 USB Fingerprint Reader Device External Reference Specification*, VPN - 27282.
- *Verix V Enhancements for USB Phase 2 External Reference Specification*, VPN - 24294.
- *Verix V Enhancements for USB Auto-Download External Reference Specification*, VPN - 24052.
- *Verix V Enhancements for V<sup>x</sup>810 (Confidence) PIN Pad*, VPN - 24669.
- *Verix V Enhancements for V<sup>x</sup>810 DUEL External Reference Specification*, VPN - 27341.
- *Verix V Enhancements for Contactless Support in the V<sup>x</sup>810 PIN Pad*, VPN - 27342.

- *Verix V Enhancements for Contactless Support in the Vx810 PIN Pad External Reference Specification*, VPN - 27342.
- *Vx610 CDMA Operating System External Reference Specification*, VPN - 27405.
- *Verix V Operating System Enhancements for Vx670 Bluetooth Modem Support External Reference Specification*, VPN - 27328.
- *Verix V Enhancements for Vx700 Unattended Payment Terminals (UPT) External Reference Specification*, VPN - 27334.
- Verix V Enhancements for Vx570 Terminals, USB Keyboard Support for Latvia (IC9129), USB Support for USB to RS-232 Modules (IC9230) External Reference Specification, VPN - 06391.
- *Verix V OS Enhancements for Eisenhower Modem Support in the Vx510, External Reference Specification*, VPN - 27545.
- *Verix V OS Enhancements for the Vx510 GPRS Desktop Terminal Software Engineering Requirement Specification*, VPN - 28257.
- *V5 Operating System Addendum to the Verix V Operating System Programmers Manual*, VPN - 24305.
- *VMS-RAD Enhancements for Verix and Verix V Functional Reference Document*, VPN - 27199.
- *Verix V System Mode Functional Specification*, VPN - 27675.
- *Verix V Enhancements for Vx570 Terminals, USB Support for Metrologic Barcode Scanner*, VPN - 28593.

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.

---

## PREFACE

*Related Documentation*



## 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, keypad key return values, and printer control codes for the Verix V 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
<b>Update CONFIG.SYS Entries with Applications</b>		
int get_env(const char *key, char *buffer, int bufsize);	Access CONFIG.SYS entries.	<a href="#">157</a>
int put_env(const char *key, const char *val, int len);	Stores an environment variable in CONFIG.SYS.	<a href="#">158</a>
<b>Miscellaneous Service Calls</b>		
int get_component_vars(int handle, char *buffer, int len);	Returns information about an OS component (typically a driver).	<a href="#">280</a>
int set_combo_mode(int mode);	Sets the module specified by mode. (either in conventional telephone modem or as a TCP/IP adapter).	<a href="#">281</a>
int SVC_CHK_PASSWORD(char *buffer);	Compares the counted string in buffer to the password for the current group.	<a href="#">282</a>
int SVC_FLASH_SIZE(void);	Returns the amount of installed flash memory in kilobytes.	<a href="#">283</a>
int SVC_INFO_COUNTRY(char *buf_12);	Stores 12 bytes of factory-defined country variant data in the caller's buffer.	<a href="#">284</a>
int SVC_INFO_DISPLAY(char *buf_6);	Stores display type and size information in the caller's buffer.	<a href="#">286</a>
int SVC_INFO_HW_VERS(char *buf_2);	Stores a 2-byte factory-defined hardware version in the caller's buffer.	<a href="#">288</a>
int SVC_INFO_KBD(char *stuff_1x);	Stores the 1-byte keypad type in the caller's buffer.	<a href="#">289</a>

**Table 3** Function Calls (continued)

Function Call	Description	Page
int SVC_INFO_LOTNO(char *buf_6);	Stores a 6-byte factory-defined manufacturing lot number in the caller's buffer.	291
int SVC_INFO_MAG(char *buf_1);	Stores a 1-byte magnetic card reader type code in the caller's buffer.	292
int SVC_INFO_MFG_BLK(char *buf_30);	Stores 30 bytes of factory-defined manufacturing data in the caller's buffer.	293
int SVC_INFO_MOD_ID(void);	Returns a code indicating the type of modem installed.	294
int SVC_INFO_MODULE_ID(int port);	Detects the type of radio used such as the WiFi, GPRS, CDMA, and Ethernet.	295
int SVC_INFO_MODELNO(char *buf_12);	Stores a 12-byte factory-defined model number in the caller's buffer.	297
int SVC_INFO_PARTNO(char *buf_12);	Stores a 12-byte factory-defined part number in the caller's buffer.	298
int SVC_INFO_PIN_PAD(char *buf_1);	Stores a 1-byte PIN pad type code in the caller's buffer.	299
int SVC_INFO_PORT_IR(void);	Returns the serial port number for infrared communication (if supported).	300
int SVC_INFO_PORT_MODEM(void);	Returns the serial port number connected to the modem.	301
int SVC_INFO_PRNTR(char *buf_1);	Stores a 1-byte printer type code in the caller's buffer.	302
int SVC_INFO_RESET(char *buf_12);	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).	304
int SVC_INFO_SERLNO(char *buf_11);	Stores an 11-byte factory-set serial number in the caller's buffer.	305
int SVC_LED(int id, int mode);	Sets the light-emitting diode specified by ID on or off.	307
int SVC_RAM_SIZE(void);	Returns the amount of installed RAM in kilobytes.	308
long SVC_INFO_LIFETIME(void);	Returns the total number of seconds the terminal has been in operation.	290
void SVC_INFO_CRASH(struct info_crash_t *results);	Retrieves diagnostic information about the most recent fatal exception.	285

**Table 3**      **Function Calls** (continued)

Function Call	Description	Page
void SVC_INFO_EPROM(char *buf_9);	Stores a counted string that contains an 8-byte firmware version in the caller's buffer.	287
void SVC_INFO_PTID(char *buf);	Stores a counted string that contains an 8-byte terminal identification number in the caller's buffer.	303
void SVC_VERSION_INFO(char *buf);	Stores a counted string that contains the OS version information.	309
int SVC_SHUTDOWN(void);	Turns off the Vx610, Vx670, Vx810, and Vx700 terminals.	306
int SVC_INFO_OS_HASH (U8* hashout20, U8* keyin, int keysz);	Allows the application to compute a checksum for the entire OS.	310

## File Management

### File Access Function Calls

int close(int handle);	Closes file when access to the file is no longer needed.	115
int delete(int handle, unsigned int count); int delete_vlr(int handle, unsigned int count); int delete_cvlr(int handle, unsigned int count); int delete_(int handle, unsigned int count);	Deletes data from a file opened for write access at the location of the file position pointer.	96
<b>Note:</b> <i>delete_()</i> provides an alternate name for the <i>delete</i> function so as not to conflict with the <i>delete</i> keyword in C++.		
int get_file_date(int handle, char *yymmddhhmmss);	Returns information about the last update to the file.	99
int get_file_size(int handle, long *filesize);	Returns information about the file size.	98
int insert(int handle, const char *buffer,int size); int insert_vlr(int handle, const char *buffer,int size); int insert_cvlr(int handle, const char *buffer,int size);	Inserts data into a file opened for write access at the location of the file position pointer.	94
int lock(int handle, long reserved1, long reserved2);	Locks the open file associated with <i>handle</i> , preventing it from being accessed through any other handle.	104
int lseek(int handle, long offset, int origin); int seek_vlr(int handle, long offset, int origin); int seek_cvlr(int handle, long offset, int origin);	Sets the file position pointer of an open file to a specified location.	91
int open(const char *id, int opentype);	Allocates and returns an integer file handle used in all subsequent file operations.	84
int read(int handle, char *buffer, int count); int read_vlr(int handle, char *buffer, int count); int read_cvlr(int handle, char *buffer, int count);	Transfer data from a file opened for reading, to a buffer within the application's data area.	87

**Table 3 Function Calls** (continued)

Function Call	Description	Page
int _remove(const char *filename);	Deletes a specified file in the directory.	102
int _rename(const char *oldname, const char *newname);	Points the caller's pointer to a pair of pointers to ASCII pathnames.	107
int SVC_CHECKFILE(char *filename);	Calculates the checksum for the specified file.	100
int unlock(int handle, long reserved1, long reserved2);	Removes a lock set by lock() from the open file associated with handle.	105
int write(int handle, const char *buffer, int count); int write_vlr(int handle, const char *buffer, int count); int write_cvlr(int handle, const char *buffer, int count);	Transfer data from an application's buffer to a file that is open for writing.	89
<b>File Attribute Function Calls</b>		
int get_file_attributes(int handle);	Returns the attribute byte of an open file.	109
long get_file_max(int handle);	Returns the maximum file data size set in set_file_max().	110
int reset_file_attributes(int handle, int attributes);	Clears attribute flags for an opened file.	111
int set_file_attributes(int handle, int attributes);	Sets selected attribute flags for an open file.	112
int set_file_max(int handle, long maxsize);	Sets the maximum data size allowed for a file.	113
<b>Keyed File Reads and Writes</b>		
int getkey(const char *key, const char *buffer, int size, char *filename);	Retrieves data associated with a given key value.	118
int putkey(const char *key, const char *buffer, int size, char *filename);	Stores the data for a given key.	119
<b>File Directory Function Calls</b>		
int dir_get_attributes(const char *filename);	Provides access to the file attribute bits that the file system maintains.	121
int dir_get_file_date(const char *filename, char *yyyymmddhhmmss);	Retrieves the file date.	122
long dir_get_file_size(const char *filename);	Returns the size of the file.	123
long dir_get_file_sz(const char *filename);	Returns the number of data bytes in the file.	124
int dir_get_first(char *drive);	Returns a NULL-terminated string containing the name of the first file in the directory (usually CONFIG.SYS).	125
int dir_get_next(char *buffer);	Takes the current filename in the specified directory and retrieves the following filenames.	126

**Table 3**      **Function Calls** (continued)

Function Call	Description	Page
int dir_get_sizes(const char *drive, struct fs_size *fs);	Returns general information about the specified directory.	127
int dir_put_file_date (const char *filename, const char *yyyymmddhhss);	Attaches a date to the file.	128
int dir_set_attributes (const char *filename, int attributes);	Turns on one or more of the attribute bits for the specified file.	130
int dir_reset_attributes (const char *filename, int attributes);	Turns off the identified attributes.	129
int file_copy(const char *source, const char *target);	Copies the file named by source to target.	131
int SVC_RAM_SIZE(void);	Returns the amount of RAM memory, in kilobytes, installed in the terminal.	132
int unzip(const char *zipfile);	Decompresses a VeriFone zip format file.	133
<b>Flash File System</b>		
int dir_flash_coalesce(void);	Erases all flash file system files tagged for deletion and pushes the current files down, recovering memory space.	137
int dir_flash_coalesce_size(long *size);	Returns the number of bytes to reclaim with a coalesce.	138
int SVC_FLASH_SIZE(void);	Returns the amount of flash memory, in kilobytes, installed in the device.	139
<b>String Utilities</b>		
int dtoa (double d, char *buf, int buf_size, int format, int precision);	Converts a floating point value to a string.	697
char *ltoa(long value, char *buffer, int radix);	Converts a number to a string.	698
char *ultoa (unsigned long value, char *buffer, int radix);	Converts a number to a string.	699
void SVC_INT2(unsigned int value, char *buffer);	Converts a number to a string.	708
unsigned int SVC_2INT(const char *source);	Converts a counted ASCII string containing decimal digits to a binary value.	709
int SVC_CS2AZ(char *zstring, const char *cstring);	Converts counted string, cstring, to a standard C zero-terminated string, zstring.	710
int SVC_AZ2CS(char *cstring, const char *zstring);	Converts a zero-terminated string, zstring, to a counted string, cstring.	707
void strnlwr(char *dest, const char *source, int size);	Converts string to lowercase.	700
void strnupr(char *dest, const char *source, int size);	Converts string to uppercase.	701
void SVC_HEX_2_DSP(const char *hex, char *dsp, int n);	Converts binary data to ASCII text.	702

**Table 3** Function Calls (continued)

Function Call	Description	Page
void SVC_DSP_2_HEX(const char *dsp, char *hex, int n);	Converts ASCII hexadecimal data to binary.	703
int SVC_PACK4(char *dest, const char *source, int size);	Compresses ASCII data.	704
int SVC_UNPK4(char *dest, const char *source, int size);	Decompresses ASCII data compressed by SVC_PACK4().	705
<b>Event Handling</b>		
<b>Event Function Calls</b>		
int clr_timer(int timer_id);	Cancels a timer before it expires.	194
int set_timer(long msecs, long eventmask);	Schedules an event to occur after a specified delay.	200
int SVC_WAIT(unsigned int msecs);	Suspends the calling task for a specified amount of time.	202
long peek_event(void);	Examines pending events.	195
post_user_event(int user_task_id, int user_bits);	Allows to post an immediate event from a thread to another thread or task.	196
long read_event(void);	Reads and clears pending events.	197
long read_evt(int needed_events);	Reads and clears pending events. This functions is similar to read_event function except that only events listed in the bit mask needed_events will be reported to the caller.	198
int read_user_event(void);	Reads and clears the new “user events” field for the calling task. It also resets the new EVT_USER bit in the calling task’s main event.	199
long wait_event(void);	Waits for an event to occur. See <a href="#">Table 7</a> for event codes returned by wait_event( ).	203
int wait_evt(int needed_events);	Waits for an event to occur. The events that are listed in the bit mask needed events will only cause the task to awake.	204
int set_signal_events(int handle, char *signal);	Allows the radio to enable an event to occur when one or more input signal lines changes.	201

**Table 3** Function Calls (continued)

Function Call	Description	Page
<b>Console Device</b>		
int activate_task(int task_id);	Allows the current console owner to pass control of the console to the specified task.	229
int alpha_shift(int c);	Returns the character that follows char in the ALPHA key shift sequence.	230
int close(int handle);	Releases ownership of the console device.	231
int clreol(void);	Clears the display line from the current cursor position to the end of the line, relative to the current window.	232
int clrscr(void);	Clears the current display window and places the cursor in the upper-left corner of the window (column 1, line 1).	233
int contrast_down(void);	Decrementsthe current contrast setting.	234
int contrast_up(void);	Increments the current contrast setting.	235
int delline(void);	Deletes the display line that contains the cursor and moves all lines below it up one line.	236
int disable_hot_key(void);	Disables the hotkey feature.	237
void disable_key_beeps(void);	Disables beeps when keys are pressed.	238
void enable_hot_key(void);	Re-enables the hotkey feature after it is disabled by disable_hot_key().	239
int enable_key_beeps(void);	Enables audible key beeps on a keypress.	240
int get_console(int clear_keys);	Returns the handle for the console if the current task owns the console.	246
int get_font(char *font_name);	Returns the filename of the current font or the string DEFAULT if the default font is in use.	242
int get_font_mode(void);	Returns the mode of the current font setting.	248
void getfont(char *font);	Returns the display font.	242
int getcontrast(void);	Returns the current display contrast setting.	241
int getgrid(void);	Returns the current grid setting.	243
long get_hot_key_sts(void);	Retrieves hot key status.	249

**Table 3** Function Calls (continued)

Function Call	Description	Page
int getscrollmode(void);	Returns the current scroll mode setting.	265
int gotoxy(int x, int y);	Positions the cursor at the screen relative to the character position specified.	250
int insline(void);	Inserts a blank line following the line containing the cursor.	251
int kbd_pending_count(void);	Returns the number of keystrokes available for reading.	253
int kbd_pending_test(int t);	Tests if key <b>t</b> is in the keyboard buffer.	254
int key_beeps(int flag);	Turns on beeps when keys are pressed	255
int open("/dev/console", int unused);	Returns the handle for writing, reading, or closing the console device.	256
int put_graphic(const char *buf, int len, int x1, int y1, int x2, int y2);	Displays graphic images (for example, icons) within a specified area.	258
int putpixelcol(char *buffer, int len);	Displays graphic images on a byte-by-byte basis.	257
int read(int handle, char *buffer, int length);	Retrieves the keys in the key buffer.	259
int resetdisplay(const char *font, int grid_id);	Sets the font.	260
int screen_size(char *buf);	Stores the screen size in *buf.	261
int set_backlight(int mode);	Turns the backlight on/off .	266
int set_cursor(int flag);	Turns the cursor on and off.	267
int set_font(const char *font_name);	Sets the font to the specified font file.	263
int set_hot_key(int keycode);	Defines the hotkey and who owns it.	269
int setcontrast(int value);	Sets the display contrast level to the specified value.	262
int setfont(const char *font);	Sets the display font.	263
int setscrollmode(int mode);	Sets the scroll mode.	265
int SVC_INFO_DISPLAY(char *buf_6);	Stores display type and size information in the caller's buffer.	270
int set_bcm(int mode);	Overrides the CONFIG.SYS *BCM setting.	212
int SVC_INFO_KBD(char *stuff_lx);	Fills the caller's buffer with the one-byte keyboard type from the manufacturing block.	271
int wherecur(int *x, int *y);	Returns the current cursor position relative to the physical display not the current window.	272

**Table 3 Function Calls** (continued)

Function Call	Description	Page
int wherewin(int *x1, int *y1, int *x2, int *y2);	Returns the current display window coordinates into the four integer variables.	273
int wherewincur(int *x, int *y);	Returns the current cursor position relative to the current window <i>not</i> the physical display.	274
int window(int x1, int y1, int x2, int y2);	Defines a logical window within the physical display.	275
int write(int handle, const char *buffer, int count);	Writes buffer to display.	276
int write_at(char *buf, int len, int x, int y);	Similar to <code>write()</code> , except that the cursor is positioned prior to writing the data in the current font.	277
int getinverse(void);	Returns the current video setting.	244
int inverse_toggle(void);	Toggles the current inverse video setting. Equivalent to <code>setinverse(3)</code> .	252
int setinverse(int value);	Selects the inverse video setting based on the two LSBs of <i>value</i> .	264
int setAlphaMode(classic);	On Vx810, switches the keyboard operation to classic mode.	223
int setAlphaMode(cell_phone);	On Vx810, switches the keyboard operation to cell phone mode.	223
int getAlphaMode(void);	On Vx810, returns the current mode of the keyboard.	224
int alpha_multi_shift(int key, int *shift);	On Vx810, switches the keyboard operation to classic mode (default) or cell phone mode.	225

## System Devices

### Device Management Function Calls

int get_name(int handle, char *name_20);	Retrieves the device name associated with handle.	334
int get_owner(const char *id, int *task_id);	Retrieves owning task and handle for a device or pipe.	335
int set_owner(int handle, int task);	Transfers ownership of an open device to another task.	336

### Magnetic Card Reader

int card_pending(void);	Determines if there is unread data in the card reader buffer.	339
int close(int handle);	Disables the card reader input, preventing the terminal from recognizing card reads.	340
int open("/dev/mag", int unused);	Prepares the firmware to accept and store card reads.	341

**Table 3** Function Calls (continued)

Function Call	Description	Page
int read(int handle, char *buffer, int size);	Transfers data from a card reader scan into the buffer.	342
int SVC_INFO_MAG(char *buf_1);	Stores a 1-byte magnetic card reader type code in the caller's buffer.	345
<b>Clock and Timer Function Calls</b>		
int close(int handle);	Releases the resources associated with the clock handle.	340
int clr_timer(int timer_id);	Cancels a timer set by <code>set_timer()</code> .	194
void date2days(const char *yyyymmdd, long *days);	Sets <code>*days</code> to the number of days elapsed from January 1, 1980, to the specified date.	364
void datetime2seconds(const char *yyyymmddhhmmss, long *secs);	Converts a date/time string to the number of seconds since January 1, 1980 (that is, midnight Dec. 31, 1979).	364
int days2date(long *days, char *yyyymmdd);	Determines the date that is the given number of days past January 1, 1980.	364
unsigned long read_ticks(void);	Returns the number of clock ticks elapsed since the terminal was powered up or reset.	372
int open(const char *id, int);	Opens the clock/calendar device, returning its associated device handle.	367
int read(int hClock, char *buffer, int size);	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).	368
int read_clock(char *yyyymmddhhmmssw);	Stores the current time and date in the caller-provided buffer.	369
int seconds2datetime (const unsigned long *seconds, char *yyyymddhhmmss);	Converts the number of seconds since January 1, 1980 (that is, midnight Dec. 31, 1979) to a date/time string.	367
secs2time (const long *secs, char *hhmmss);	Converts the number of seconds to a time string.	364
int set_timer(long milliseconds, long eventmask);	Schedules an event to occur after the specified number of milliseconds (ms) elapsed.	200
int SVC_VALID_DATE(const char *yyyymddhhmmss);	Verifies that its argument represents a valid date and time.	364

**Table 3**      **Function Calls** (continued)

Function Call	Description	Page
int SVC_WAIT(unsigned int milliseconds);	Suspends the calling task for the specified number of milliseconds.	373
time2secs (const char *hhmmss, long *secs);	Converts the time into a seconds string.	364
int write(int handle, const char *buffer, int count);	Sets the system date and time.	370
<b>Beeper</b>		
void beeper_off(void);	Squelches the beeper.	376
int close(int handle);	Releases the handle associated with the beeper.	377
int enable_key_beeps(void);	Enables audible key beeps on a keypress.	238
void error_tone(void);	Produces a 100-ms tone at 880 Hz.	378
int key_beeps(int flag);	Turns on/off keypress beeps.	379
void normal_tone(void);	Produces a 50-ms tone at 1245 Hz.	380
int open (const char *id, int unused);	Explicitly opens the sound-generating device, returning its associated device handle.	381
void play_RTTTL(char *music);	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.	375
int sound(int note, int milliseconds);	Causes the beeper to generate one of the 96 standard tones at a specified time.	382

## Communication Devices

### FIFOs

void SVC_CLR_FIFO(fifo_t *fifo, int datasize);	Initializes the FIFO data structure pointed to by <code>fifo</code> with a capacity of <code>datasize</code> bytes.	313
int SVC_CHK_FIFO(const fifo_t *fifo);	Returns the number of bytes currently stored in the FIFO (that is, those written to it, but not yet read).	312
int SVC_GET_FIFO(fifo_t *fifo);	Retrieves byte from FIFO.	314
int SVC_PUT_FIFO(fifo_t *fifo, int val);	Add a byte to FIFO.	315
int SVC_READ_FIFO(fifo_t *fifo, char *buffer, int size);	Reads bytes from FIFO.	316
int SVC_WRITE_FIFO (fifo_t *fifo, const char *buffer, int size);	Writes bytes to FIFO.	317

### CRCs

unsigned int SVC_CRC_CALC(int type, const char *buffer, int size);	Calculates a CRC value for <code>size</code> bytes of data in <code>buffer</code> .	320
unsigned long SVC_CRC_CALC_L (int type, const char *buffer, int size);	Identical to <code>SVC_CRC_CALC()</code> , except that it returns a 32-bit result.	321

**Table 3** Function Calls (continued)

Function Call	Description	Page
unsigned char SVC_LRC_CALC(void const *buffer, int size, unsigned char seed);	Calculates the LRC (longitudinal redundancy check) value for size bytes of data in buffer.	327
unsigned int SVC_CRC_CRC16_L(void const *buffer, int sz, unsigned int seed);	Calculates a standard CRC16 CRC value for size bytes of data in buffer.	324
unsigned int SVC_CRC_CRC16_M(void const *buffer, int sz, unsigned int seed);	Calculates a standard CRC16 CRC value for size bytes of data in buffer.	325
unsigned int SVC_CRC_CCITT_L(void const *buffer, int sz, unsigned int seed);	Calculates a 16-bit CRC for size bytes of data in buffer using the CCITT polynomial	322
unsigned int SVC_CRC_CCITT_M(void const *buffer, int sz, unsigned int seed);	Calculates a 16-bit CRC for size bytes of data in buffer using the CCITT polynomial	323
unsigned long SVC_CRC_CRC32_L(void const *buffer, int sz, unsigned long seed);	Calculates a 32-bit CRC32 CRC value for size bytes of data in buffer.	326
unsigned int SVC_MEMSUM(const char *buffer, long size);	Computes the sum of size bytes from buffer, treating both the bytes and the sum as unsigned, and ignoring overflows.	328
unsigned int SVC_MOD_CK(const char *acct);	Generates a Luhn check digit for a sequence of digits or validates a sequence of digits containing a check digit.	329

**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.

int close(int handle);	Disables the device.	572
int download(int handle, void *parms);	Receives a download through the open serial port.	573
int get_component_vars(int handle, char *buffer, int len);	Returns the nonvolatile data for this communication port.	574
int get_fifo_config(int handle, char *buffer);	Gets the current FIFO configuration.	575
int get_opn_blk(int port, struct SetOpnBlk *ob);	Copies the current opn_blk structure into the caller's buffer.	576
int get_port_status(int port, char * bfr);	Copies current status information to caller's 4-byte buffer.	577

**Table 3**      **Function Calls** (continued)

Function Call	Description	Page
int open(const char *id, int opentype);	Prepares the asynchronous RS-232 interface for operation. Returns the device handle, for example, device_name = • "/dev/com1" • "/dev/com2" or • "/dev/com3"	579
unsigned long get_usb_device_bits (void)	Gets the status of all supported USB devices.	562
int read(int comm_handle, char *buffer, int size);	Transfers data from the RS-232 port into the buffer and returns the number of bytes actually read	580
int reset_port_error(int port);	Resets error conditions for parity, framing, and overrun.	581
int set_fifo_config(int handle, const char *buffer);	Sets the SCC transmit FIFO length.	582
int set_opn_blk(int port, const struct Opn_Blk *ob);	Initializes/reinitializes the communication parameters.	583
int set_serial_lines(int handle, char *buffer);	Uses the parameters in buffer to set/reset DTR, RTS, and BRK.	584
int write(int handle, const char *buffer, int count);	Transfers data from an application buffer into the device driver's buffer, only if the latter is empty.	586
<b>USB Ethernet (ETH1)</b>		
int close(int eth_handle);	Releases the Ethernet device.	776
int get_enet_status(int hdl, char *status4);	Checks whether Ethernet link is live or not.	777
int get_enet_MAC(int eth_hdl, char *MACbuf);	Returns the MAC (Media Access Control) address.	778
int open("/dev/eth1", int unused);	Returns the handle for reading or writing the Ethernet device.	773
int read(int eth_handle, char *buffer, int count);	Allows the user to read one packet from the Ethernet device.	774
int set_enet_rx_control(int hdl, int rx_control);	Used to suppress certain packets.	779
int write(int eth_handle, const char *buffer, int count);	Allows the user to write one packet for the Ethernet device.	775
<b>Modem Port (COM3)</b>		
int SVC_ZONTALK(unsigned char type);	Receives a download through the terminal modem.	531
<b>Serial Printer Port (COM4)</b>		
int get_fc_config(int fd, char *buffer);	Retrieves current hardware flow control configuration.	535
int set_fc_config(int fd, char *buffer);	Sets the hardware flow control configuration.	536

**Table 3** Function Calls (continued)

Function Call	Description	Page
<b>Internal PIN Pad Port (COM5)</b>		
int PINentryStatus(void);	Returns the PIN entry status and can be used to infer when the console belongs to the PIN-entry background task.	555
int SVC_INFO_PIN_PAD(char *buf_1x);	Fills the caller's buffer with the one-byte internal PIN pad availability information from the manufacturing block.	556
int TerminatePinEntry(void);	Ends the PIN entry session.	554
<b>USB Internal WiFi (WLN1) (on Vx670 only)</b>		
int USB_WIFI_POWER(int power);	Used to power the WiFi on or off.	788
int open("/dev/wln1", int unused);	Returns the handle for reading or writing.	784
int read(int handle, char *buffer, int count);	Allows the current owner tot read one packet from the WiFi device.	785
int write(int handle, const char *buffer, int count);	Allows the current owner to write one packet for the WiFi device.	785
int close(int handle);	Releases the ownership of the WiFi device.	787
<b>USB Client (on Vx810 only)</b>		
int get_usbd_status(int handle);	Checks whether the USB initialization is complete	794
int_usb_pending_out(int hdl);	Returns the amount of written but unsent data in the driver's buffers.	795

**Table 3** Function Calls (continued)

Function Call	Description	Page
<b>Multitasking</b>		
<b>Task Function Calls</b>		
int _exit(int status);	Terminates the calling task.	163
int get_group(void);	Returns the effective file group membership of the calling task.	164
int get_task_id(void);	Retrieves the task number.	165
int get_task_info(int id, struct task_info *info);	Stores information about a specified task in the info structure.	166
int run(const char *file, const char *args);	Executes the specified program file as a new task.	167
int set_group(int group);	Changes the effective file group membership of the calling task.	169
int run_thread(int routine, int parameter, int stacksize);	Executes the specified thread as a new task.	168
<b>Semaphore Application Function Calls</b>		
int sem_init(sem_t *sem, unsigned int value);	Initializes a semaphore to the value given.	174
int sem_wait(sem_t *sem);	Causes the calling thread to be suspended if the semaphore is unavailable.	175
int sem_post(sem_t *sem);	Frees the semaphore for general use and returns the handle immediately.	176
<b>Pipes</b>		
int open(const char *id, int opentype);	Allocates a pipe control block and assigns it the specified name.	181
int pipe_connect(int pipehandle, int targetpipehandle);	Allows data to be written using one handle, and subsequently to be read using the other handle.	182
int pipe_init_char(int handle, int max_input_chars);	Initializes the character mode pipe.	184
int pipe_init_msg(int handle, int max_input_msgs);	Initializes a message mode pipe.	185
int pipe_pending(int handle);	Tests pipe data availability.	186
int read(int handle, char *buffer, int count);	Transfers data from the serial port into the buffer.	187
int write(int handle, const char *buffer, int count);	Transfers data from an application buffer into the device driver's buffer.	188
int close(int handle);	Disables the serial port.	180
int SVC_RESTART(const char *filename);	Performs a complete terminal reset.	190

**Table 3 Function Calls** (continued)

Function Call	Description	Page
<b>Special Topics</b>		
<b>Support for APACS40 Cryptographic Functions</b>		
int Calc_Auth_Parm(const char *TranData, char *AuthParm);	Computes the authentication parameter based on the provided transaction data.	689
int Calc_MAC(const char *buffer, int len, char *mac8);	Computes the standard ANSI X9.19 message authentication code for the designated buffer.	690
int Create_MAC_Key(int hostkey, const char *A, const char *B);	Sets the current MAC key based upon use of the One Way function.	691
int Init_MAC(void);	Allows multiple tasks to use APACS40 features (one at a time).	692
int New_Host_Key(int hostkey, const char *rqst_residue, const char *resp_residue);	Updates the current host key in the APACS40.KEY Group 0 file for the designated host.	693
int Reset_Key(int hostkey);	Resets the current host key in the APACS40.KEY Group 0 file for the designated host.	694
int Term_MAC(void);	Clears the current owner variable of the APACS40 feature set so that another task can use the feature.	695

**Security/Crypto Functions**

int crypto_read(int handle, char *buffer, int count);	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.	590
int crypto_write(int handle, const char *buffer, int count);	Encrypts and writes count bytes of data from buffer to the open file associated with handle.	591
int DES(unsigned char ucDeaOption, unsigned char *pucDeaKey8N, unsigned char *pucInputData, unsigned char *pucOutputData);	Performs DES, DESX, and triple-DES computations.	595
int GenerateRandom(unsigned char * random8);	Returns an 8-byte random value.	596
int AES(unsigned char ucAesOption, unsigned char *pucAesKey8N, unsigned char *pucInputData, unsigned char *pucOutputData);	Performs AES computations on 128-bit data block.	594
int isAttacked(void);	Indicates if an attack occurred, causing the loss of the transaction keys or encrypted files.	597
int rsa_calc(unsigned short * msg, unsigned short *mod, int wds, int exp, unsigned short * result);	Performs a public key RSA computation.	598
int SHA1(unsigned char * unused, unsigned char *input_buffer, unsigned long nb, unsigned char * sha20);	Performs an SHA-1 computation as described in FIPS PUB 180-2.	599

**Table 3**      **Function Calls** (continued)

Function Call	Description	Page
int iPS_GetScriptStatus(unsigned char ucScriptNumber, unsigned char *pucINName);	Checks if a VeriShield security script file is installed in the Vx5xx/Vx6xx terminal and if so, returns the name of the script.	601
int iPS_InstallScript(char *pucINName);	Installs a VeriShield security script file in the unit.	602
int iPS_ExecuteScript(unsigned char ucScriptNumber, unsigned char ucMacroID, unsigned short usINDataSize, unsigned char *pucINData, unsigned short usMaximumOUTDataSize, unsigned short *pusOUTDataSize, unsigned char *pucOUTData);	Starts the execution of a given macro from a given loaded VeriShield security script.	603
int iPS_UninstallScript(unsigned char ucScriptNumber);	Uninstalls the specified VeriShield security script from the unit.	604
int iPS_CancelPIN(void);	Cancels the PIN processing.	606
int iPS_GetPINResponse(int *piStatus, PINRESULT *pOUTData);	Checks the status of the PIN session.	607
int iPS_RequestPINEntry(unsigned char ucPANDataSize, unsigned char *pucINPANData);	Initiates the PIN collection.	609
int iPS_SelectPINAlgo(unsigned char ucPinFormat);	Selects the PIN algorithm used during the next PIN session.	610
int iPS_SetPINParameter(PINPARAMETER *psKeypadSetup);	Configures several parameters for the upcoming PIN session.	611
int iPS_CheckMasterKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKVC);	Indicates if a key is present in the specified location.	613
int iPS_DeleteKeys(unsigned long ulKeyType);	Deletes the specified set of keys.	614
int iPS_LoadMasterClearKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKeyValue);	Loads the security script's master keys.	615
int iPS_LoadMasterEncKey(unsigned char ucKeySetID, unsigned char ucKeyID, unsigned char *pucINKeyValue);	Loads the security script's master keys without deleting the keys already loaded.	616
int iPS_LoadSysClearKey(unsigned char ucKeyID, unsigned char *pucINKeyValue);	Loads the KLK (system keys).	617
int iPS_LoadSysEncKey(unsigned char ucKeyID, unsigned char *pucINKeyValue);	Loads the system keys.	618
<b>WWAN Functions</b>		
int set_radio_ctl(int hdl, const char *sigs);	Sets and resets a control line to the radio module.	763
int get_radio_sts(int hdl, char *sigs);	Returns the status of RAD_INT and RAD_INT2.	760
<b>Power Management Functions</b>		
int get_battery_sts(void);	Indicates battery status.	755
int get_battery_value(int type);	Returns the requested battery values.	755
int get_dock_sts(void);	Indicates if the unit is docked or undocked.	755

**Table 3** Function Calls (continued)

Function Call	Description	Page
int get_powersw_sts(void);	Indicates if the power switch is being held down.	756
int set_backlight(int mode);	Allows the application to immediately control the state of the backlight.	756
int set_com1_pwr(const char *sigs);	Controls power on Vx610, Vx670, Vx810, and Vx700.	756
int BatteryRegs(char *buffer);	Returns the registers in the buffer.	756
int get_battery_initialization_status(void);	Returns the initialization status.	756
int SVC_SLEEP(void);	The terminal goes to sleep after 50 ms if all applications are idle.	757
int SVC_SHUTDOWN(void);	Commands the terminal to turn itself off.	757

Table 4 displays module IDs and device.

**Table 4** Module ID

Module ID	Logical Name	Device	37xx	Vx510, Vx610, Vx670	Vx570
2		Unknown, any modem other than listed below.	Y	Y	N
3	MID_TDK_ONLY	TDK Modem	Y	N	N
4	MID_BANSHEE_ONLY	Conexant Banshee modem	N	Y	Y
5	MID_CARLOS_ONLY	Carlos Modem	Y	N	N
6	MID_CO561_ONLY	Connect One Ethernet 10BaseT only	Y	N	N
7	MID_CARLOS_CO561	Connect One Ethernet 10BaseT and Carlos combo	Y	N	N
8	MID_MC56_ONLY	Siemens GSM/GPRS US only	Y	Y	N
9	MID_MC55_ONLY	Siemens GSM/GPRS International only	Y	Y	N
10	MID_EM3420_ONLY	Sierra CDMA 1xRTT only	Y	Y	N
11	MID_CO710_ONLY	Connect one WiFi 802.11b only	Y	Y	N
12	MID_CARLOS_MC56	Siemens GSM/GPRS US and Carlos combo	Y	N	N
13	MID_CARLOS_MC55	Siemens GSM/GPRS International and Carlos combo	Y	N	N
14	MID_CARLOS_EM3420	Sierra CDMA 1xRTT and Carlos combo	Y	N	N
15	MID_CARLOS_CO710	Connect One 802.11b WiFi and Carlos combo	Y	N	N

**Table 4** Module ID (continued)

Module ID	Logical Name	Device	37xx	Vx510, Vx610, Vx670	Vx570
16	MID_EISENHOWER_ONLY	Predator Conexant Eisenhower modem	N	Y	Y
17	MID_EISEN_USB_ETHERNET		N	N	Y
18	MID_EISEN_EM3420		N	N	Y
19	MID_EISEN_MC56		N	N	Y
21	MID_EISEN_USB_WIFI		N	N	Y
22	MID_BANSHEE_CO210	Conexant Banshee/CO210 combo	N	N	Y
23	MID_CO210_ONLY		N	N	Y
24	MID_ISDN_ONLY		N	N	Y

**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.

**NOTE**

On Vx610, the OS supports the Kyocera M200 CDMA Radio Module in place of the Sierra EM3420 CDMA radio module.

**Function Call Error Codes**

Error codes for the Verix V 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 V 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 WAIT).
EAGAIN	11	Resource temporarily unavailable.
ENOMEM	12	No memory available.

**Table 5** Error Codes Set by Function Calls (continued)

<b>Code</b>	<b>Value</b>	<b>Description</b>
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 MOUNT).
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	open( ) 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).
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

<b>Code</b>	<b>Description</b>
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()`.

**Table 7      Defined Events**

Name	Device	Description
EVT_ACTIVATE	Console	Console ownership returned to application. See <a href="#">set_hot_key()</a> in Chapter 7 for more information.
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.
EVT_COM6	COM6	Input available on COM6 on Vx670 terminal.  This is also available on Vx570 terminal that supports USB to RS-232 module and 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, /* MSO300 biometric device */
EVT_USER	CTLS	/* post_user_event summary bit */
EVT_CONSOLE	Console	Display output complete.
EVT_DEACTIVATE	Console	Console ownership lost. See <a href="#">set_hot_key()</a> in Chapter 7 for more information.
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.

**Table 7**      **Defined Events** (continued)

Name	Device	Description
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 V-based terminals is efficient use of memory, especially RAM 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 RAM. 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 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.

- Buffers are 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 8. By default, the maximum number is allocated.
- The environment variable `*B` in CONFIG.SYS contains the number of buffers allocated by the system.
- Buffers are automatically joined together as needed to store messages of up to 4 KB.

## Memory Management

USB devices require a significant amount of memory which is reserved as a buffer. This buffer is required to transfer a large amount of data in a short time. The maximum number of allocated buffers is 100KB, and the minimum number is 16KB. By default, the minimum number is allocated. The buffer size can be configured using the `*USBMEM` CONFIG.SYS variable.

## 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.

## Keypad

Figure 1 illustrates the V<sup>x</sup>5xx/V<sup>x</sup>610 terminal keypad layout.

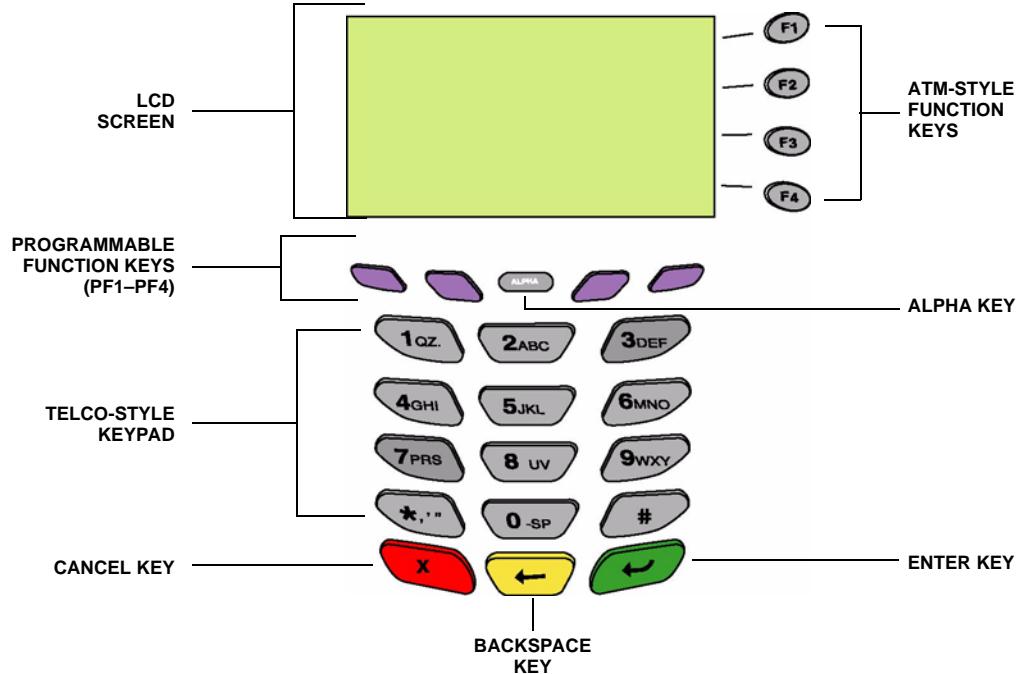
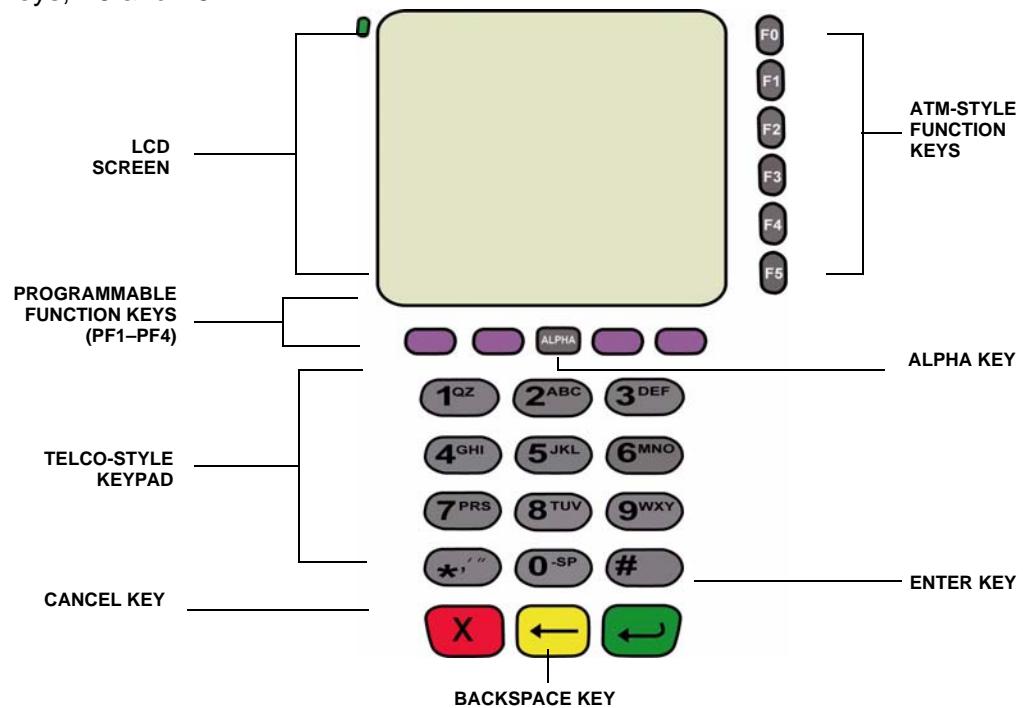


Figure 1 V<sup>x</sup>5xx/V<sup>x</sup>610 Terminal Keypad

Figure 2 illustrates the Vx670 terminal having two additional ATM-style function keys, F0 and F5.



**Figure 2** Vx670 Terminal Keypad

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.
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 <sup>a</sup>	0xEE	n with the high-order bit set.

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

<b>Keypress</b>	<b>Scan Code</b>	<b>Description</b>
F1	0xFA	z with the high-order bit set.
F2	0xFB	{ with the high-order bit set.
F3	0xFC	with the high-order bit set.
F4	0xFD	} with the high-order bit set.
F5 <sup>a</sup>	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 (Vx670).

## Printer Control Codes

**Table 9** lists the printer device driver control codes by name, hex code, and function.

**Table 9      Control Codes and Function**

<b>Name</b>	<b>Code</b>	<b>Operation</b>
NUL	0x00	Ignored
LF	0x0A	Print contents of buffer and advance to next line
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

Table 10 lists the printer device driver escape sequences.

**Table 10 Printer Escape Sequences**

<ESC> Code	Description
<ESC>a<n>;	Sets line height.
<ESC>b<n>;	Ejects <n> lines.
<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 V-based terminals.
<ESC>e<n>;	Sets right margin.
<ESC>f<n>;	Selects line attribute.
<ESC>F<n>;	Selects characters per line mode.
<ESC>g	Enters graphic mode.
<ESC>GL<f>, <t>, <w>, <h>; <b1>...<bn>	Downloads graphic image.
<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. Vx5xx/Vx6xx terminal ID is "P".
<ESC>I	Ignored in Verix V-based terminals.
<ESC>l<s><t>;	Selects font table for printing and downloading.
<ESC>m<c><r1>...<rn>;	Downloads fonts into printer.
<ESC>p<n0>, <n1>;	Sets the maximum dots on per pulse (portable units only).
<ESC>S<n>;	Ignored in Verix V-based terminals.
<ESC>s	Enters self-test mode; prints test message.
<ESC>w<n>;	Select fire pulse weight.

Table 11 presents a brief comparison of all devices available on different V<sup>X</sup> terminals.

**Table 11 Comparison of Available Devices Across V<sup>X</sup> Platforms**

Device	V <sup>X</sup> 510	V <sup>X</sup> 570	V <sup>X</sup> 610	V <sup>X</sup> 670	V <sup>X</sup> 700	V <sup>X</sup> 810 PIN Pad	V5	V <sup>X</sup> 800
COM1	External serial	External serial	External serial	External serial on Handy-Link	External serial	N/A	External serial	N/A
COM2	External PIN pad <sup>a</sup>	External PIN pad <sup>a</sup>	Internal GPRS/CDMA/WiFi	Internal GPRS/CDMA/Bluetooth	External serial	External serial	External PIN pad	External serial
COM3	Internal dial/ Ethernet <sup>b</sup>	Internal dial	Internal dial	USB external dial	USB external dial	N/A <sup>c</sup>	Internal dial	N/A
COM4	Integrated printer	Integrated printer	Integrated printer	Integrated printer	N/A	N/A	Integrated printer	N/A
COM5	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator	Internal PIN pad emulator
ETH1	N/A	USB internal Ethernet	N/A	USB external Ethernet	USB external Ethernet	USB external Ethernet	N/A	N/A
COM6	N/A	USB external serial <sup>d</sup>	N/A	USB external serial	USB external serial	N/A	N/A	N/A
COM8	N/A	N/A	N/A	N/A	External serial	N/A	N/A	N/A
WLN1	N/A	N/A	N/A	USB internal WiFi	N/A	N/A	N/A	N/A

- a. The external PINpad port can be used as a general serial port. It lacks CTS/RTS control lines but can supply power to the PINpad.
- b. On V<sup>X</sup>510 COMBO units, COM3 is hardware multiplexed to support dial and Ethernet.
- c. V<sup>X</sup>810 uses a modem only when connected to the integrated base, which has a built-in modem.
- d. Similar to the V<sup>X</sup>670 terminal.





## CHAPTER 2

# Application Programming Environment

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

Verix V-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 V OS manages tasks, memory, files, input/output devices, and other system resources. Applications request Verix V OS services through a trap mechanism encapsulated in a system call library.

## Hardware

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

- ARM 32-bit CPU
- 128 KB to 2 MB of *non-volatile* RAM (up to 36MB large memory is supported by the Vx510 terminal).
- 512 KB to 2 MB of flash 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 V 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 V-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)

Registers R0–R3 can be used within a routine; R0 is used to return results. All other registers must be preserved. Since Verix V 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.

## Virtual Memory

The ARM views memory as a 4 GB (4096 MB) flat, byte-addressable address space. Addresses are 32 bits. The Memory Management Unit (MMU) divides this into 4096 1-MB *sections*, and divides each section into *pages*: either 16 *large* pages (64 KB each), or 256 *small* pages (4 KB), or 1024 *tiny* pages (1 KB). Each section in use requires a *page table* that specifies the physical address corresponding to each page of virtual memory (or specifies that the virtual page is not valid). Small page tables occupy 1 KB (a 4-byte entry for each of the 256 pages in the section); tiny page tables require 4 KB (an entry for each of the 1024 pages). A 16-KB master table contains a pointer to the page table for each section or specifies that the section is not valid. Page table entries also specify the access allowed: read/write, read-only, or none.

Verix V uses the MMU to create a separate virtual address space for each running task. On a task swap, the OS updates the page tables appropriately for the new task.

User-space program code runs directly from its location in the flash or SRAM file system rather than being copied into RAM. A program is “loaded” by *mapping*, that is, setting up the page tables so that the physical memory where the file is stored appears at the desired virtual address. RAM for the task’s data and stack is allocated from wherever the OS can find it and similarly mapped to the virtual address where the program expects to find it.

Since the ARM MMU deals only with full pages, program files must be stored starting at a physical page boundary (that is, an address that is a multiple of the page size), and RAM must be allocated in whole pages. This normally results in some wasted space. 1 KB pages minimize this “round-up” waste, but require more space for page tables than 4 KB pages. Verix V normally uses 1 KB pages.

The major components of the Verix V OS are illustrated in Figure 3.

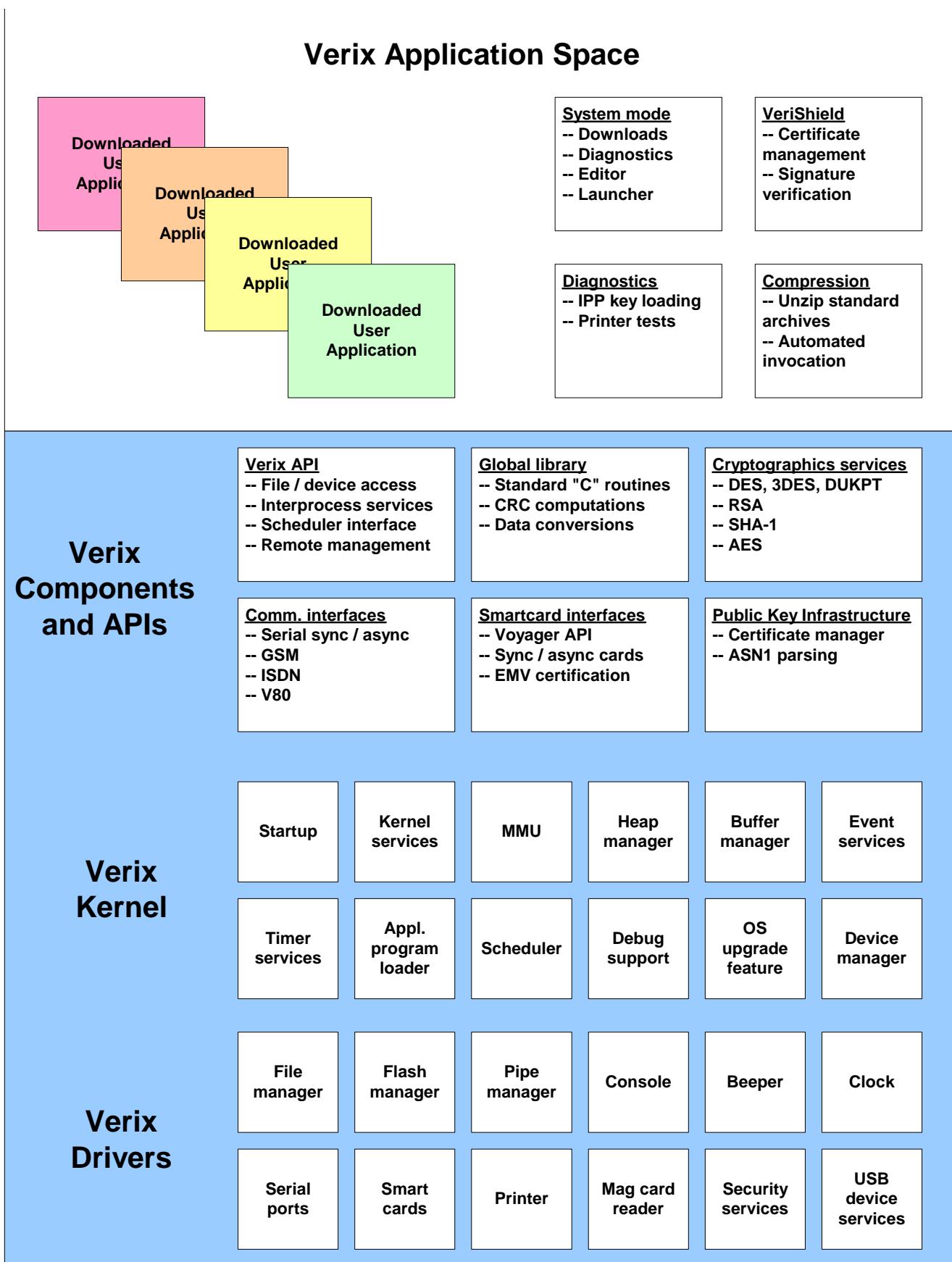


Figure 3 Verix V Block Diagram

## 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.

## Memory Management Unit (MMU)

All Verix V 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 1KB boundaries. The file manager performs this alignment automatically.

## Virtual Memory Map

User memory is allocated using the “7xx” sections, that is, all application memory lies in the virtual 256 megabytes with hexadecimal addresses in the range 0x70000000 through 0x7FFFFFFF.

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

- 0x70000000..0x700FFFFF: 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..0x703FFFFF: 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

0x701xxxx. 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..0x7046FFFF: 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..0x704FFFFF. This MB is mapped as 1024 *tiny* pages using one 4KB table.

## File System

The Verix V OS implements a file system in non-volatile RAM and 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, Verix V manages this memory region as a virtual device, much in the same manner as physical devices.

## File Locations

The physical location of a file can be designated by a drive prefix—either `I:` for the RAM file system, or `F:` for the flash system (flash files are subject to some restrictions on writing and updating, as detailed in [File Management](#)). If a filename does not include a drive prefix, it defaults to `I:`.

Note that there are multiple independent file systems in Verix. The most commonly used file system are those residing in SRAM. The RAM-based file system can be updated as well as read. There is, however, a flash-based file system that can be updated as well. Since it 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 those few cases where its behavior is unique.

There are additional similar file systems in the first sectors of the flash called boot-block sectors. The first such file system is called the boot-block file system, and the prefix “`B:`” is used to identify files in this file system. Additional files systems, labeled “`S:`”, “`T:`”, “`U:`”, “`V:`”, and “`W:`” for convenience, occupy succeeding sectors of FLASH. (The first 64KB of flash contains the kernel. The “`B:`” file system begins at physical address 0x00010000.) These Verix V file systems (B, S, T, U, V, and W) are injected during manufacturing and cannot be updated by the applications.

## File Groups

There is no hierarchical directory structure. However, files are assigned to 16 groups for access control. 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`).

## Record-Structured Files

In addition to simple stream-of-bytes files, Verix V also supports 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, 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 V 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 V currently 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 handle, char const *buffer, 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 handle, char const *buffer, 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 handle);`

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 handle, long offset, int origin);

This entry point supports the standard U\*ix 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.

## Other Devices

In addition to the console, Verix V-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.

### Console

The console comprises the display and keypad. The LCD display size is 8 lines by 21 characters (8 x 21) for 168 characters on Vx5xx/Vx610 terminals, and 21 x 16 for 336 characters on Vx670 terminal. 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.

The Verix V-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 [Figure 1](#) for a keypad illustration.

Verix V 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.

### System Mode

System mode is the first task run when a terminal is powered up or reset. System mode 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 system mode 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. Applications interrupted by entering system mode can be restarted only by a complete terminal reset. Refer to the reference manual for your terminal for a detailed description of system mode.

## Customizable Application Launcher

The application program can be customized to run on System Mode 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)

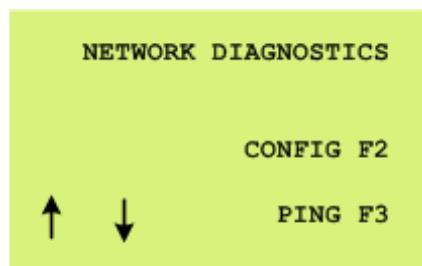
The system will search for variables in the order of \*MENU1,\*MENU2, etc. until the next variable is defined, and then wraps around to System Mode Menu 1. When the system finds the \*MENUx variable, it searches for variables in the order of \*MENUx1, \*MENUx2, etc. It displays the function label on 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

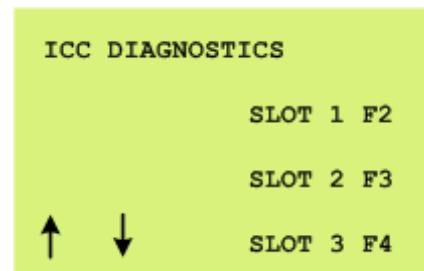
The following settings are used to configure new high-level screens that invoke specific user-written applications.

```
*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
```

When the user selects “MORE” from the System Mode Menu screen, the terminal displays the screen below.



- 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, System Mode Menu 7.
- 4 Press the **F2** key to go to the next menu specified by \*MENU(x+1); in this case, \*MENU2 which displays as follows:



- 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 System Mode Menu 1.

When the user selects one of the user programs identified, System Mode closes the console and invokes the program. When that program exits, System Mode 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.

## System Mode Battery Support

The battery system on Vx610 (supporting a battery capacity of 1800 mAH) keeps track of the remaining charge in the battery by communicating with the battery monitor chip and reading the charge register. The cells come out of the factory about 30% - 50% charged, and they are then assembled into the battery pack together with the monitor chip. The monitor chip does not read the status of the battery's charge. Battery conditioning also calibrates the monitor chip.

The battery conditioner calibrates the monitor chip by putting the battery pack into a known state, fully charged as determined by a charging current less than .1c with the battery voltage greater than 8.0 volts. The conditioner sets the current monitor register to the largest value and then it discharges the battery. When the

battery is fully discharged, determined by the battery voltage falling below 6.0 volts, 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 returns the battery status screen.

On V<sup>x</sup>670 terminal, the DS2438 Smart Battery Monitor automatically keeps track of the amount of current entering and leaving (charging and discharging) the battery cells. It is capable of measuring and keeping track of total charging and discharging current, as well as instantaneous current. These values are kept in EEPROM registers in the DS2438 chip and are available to the OS. The ISL6253 battery charger automatically charges the battery. The charger chip is enabled when the battery voltage falls below 8.0 volts and disabled when the charging current falls below 40 mA.

**NOTE**

The V<sup>x</sup>810 and the V<sup>x</sup>700 units do not operate from a battery and must be connected to an external power source to operate. The battery status is removed from the System Mode menus.

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 System Mode Battery Support menus are listed in [Table 12](#):

**Table 12      System Mode Battery Support**

Display	Action
<pre> VERIFONE OMNI QB00030 02/11/2004 VERIX COPYRIGHT 1997-2004 VERIFONE ALL RIGHTS RESERVED  BATTERY 73% </pre>	<p>At startup, the terminal displays a copyright notice screen that shows the terminal model number, the version of the V<sup>x</sup>5xx/V<sup>x</sup>6xx system firmware stored in the terminal's flash memory, the date the firmware was loaded into the terminal, the copyright notice, and the battery status.</p> <p>This screen appears for three seconds, during which time you can enter system mode by simultaneously pressing F2 and F4.</p> <p>To get battery status menu from the System Mode 7 screen, press F2.</p>

**Note:** For V<sup>x</sup>670 terminals, select BATTERY STATUS F4 on the startup screen, or BATTERY STATUS F0 on System Mode Menu 4. See [System Mode](#), for more details.

**Table 12      System Mode Battery Support** (continued)

Display	Action
<b>BATTERY CONDITIONER</b>  <b>FULL CHARGE 1800</b> <b>REMAINING 50% 900</b> <b>VOLTAGE 7245</b> <b>BATT CONDITION F4</b>	<p>This screen shows the status of the battery.</p> <p>To initiate the battery conditioner/cycler/caliber, press F4.</p> <p>The battery conditioner will check if the terminal is powered by the external power pack. If the power pack is not attached, System Mode will report an error. System Mode will charge, discharge, and then charge the battery again. During this process, the battery hardware charge counter will be calibrated; the battery's full capacity will be determined and saved.</p> <p><b>Note:</b> For Vx670 terminals, you will also see the information SN and INITIALIZED.</p>
<b>BATTERY CONDITIONER</b>  <b>/</b> <b>STEP 2/3 DISCHARGE</b>	<p>This screen is displayed during the battery conditioning.</p> <p>The line in column 1 of row 3 rotates to show that the system is still active and operating. Pressing the CANCEL key will abort the conditioning and return to the battery status screen.</p>

### Battery Conditioner

The cells and a monitor chip are then 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 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 less than 1700, the initializer sets the full charge register to 1700; otherwise, the initializer does not change the full charge register. Then the initializer writes the full charge register value into the remaining charge register.

## Timers

Timers are hardware (and “soft”) devices that can generate interrupts for driving the background processes of device drivers. In Verix V, timers are used to provide COM port baud clocks and schedule interval-timed background processing.





## CHAPTER 3

# File Management

The Verix V-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.

---

Verix V maintains all system environment information, user code, and data as memory files. Just as a PC application opens a file on disk, a Verix V 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 V manages this memory region as a *virtual* device, very much the same as the physical devices.

---

## Verix V File Systems

Verix V-based terminals have several independent file systems. The most commonly used file system exists in SRAM. The RAM-based file system can be updated as well as read. There is also an updateable flash-based file system. Since it 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. Files in the flash-based file system are identified with the prefix F:.

---

## File Conventions

### File Storage

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

Files are stored in non-volatile (battery-backed) RAM or flash memory. All files remain in memory even when power is removed from the terminal. 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.

**Filenames**

Filenames can be up to 32 characters long and *must* be terminated by a NULL character. Any non-NUL 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 V:

**NOTE**

The RealView® manuals use the .out suffix for what the Verix V calls the .odb file. The RealView naming convention is avoided on Verix V-based products as other VeriFone products and tools traditionally use .out to denote executable program files. See the *Verix, V 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 V-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 16 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.

For most users, only one user file group is required or useful. For details on all file groups, refer to [Support for File Groups](#).

**Default System Files**

One system file is always present in the RAM 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 V does not impose limits on the number of user files that can be created. However, expansion is limited by RAM 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\_RESET or SVC\_RESTART routine 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 system mode using the **EDIT** menu selection.

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()`

**Verix V File Manager**

The Verix V platform employs a file manager that attempts to recover unused memory space to provide as much data storage space as possible. This space recovery helps ensure maximum transaction storage capacity. Be aware that when writing, inserting, and deleting data, any files or data stored after the current file pointer are moved. While the time required to move this data is normally imperceptible, the time delay is multiplied when performing multiple operations, such as in a loop. Also, as the application stores additional transaction data, the delay becomes longer as each transaction is added to the same file.

The *file extension* feature allows you to specify a maximum size for a file, as described in [Support for File Extension Areas](#). If you use this feature, the required space is immediately claimed from the file system and is not available for other files. Future file operations on this file, however, do not disturb other files in the system.

To minimize potential delays resulting from file management routines to reclaim unused memory, observe the following guidelines:

- Place frequently updated files after any large data storage files; this minimizes the amount of data that must be moved during file management.

**NOTE**

Files are added to the directory chronologically. Ensure that large or static files are created/downloaded before small dynamic files.

- 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 instead of performing delete and write operations.

**NOTE**

While fixed-length records and padding can be used to prevent changing the size of file records during file management, be aware that these methods consume valuable transaction storage capacity.

## Support for File Groups

An application consists of one or more files, some of which can represent executable code files. Verix V supports a *Group ID* (GID) feature that allows these files to be separated from files in other applications. The system supports 16 groups. Not all groups are alike; the system enforces the following special rules:

- Group 0 is the *administrative* group, reserved for operating system files provided by VeriFone
- Group 1 is the *user* group, intended for the *primary* application, typically belonging to the *owner* of the terminal. Another term often used is *sponsor application*
- Groups 2 through 14 are intended for other applications
- Group 15 can be used as a *universal* group

The primary feature provided by the Group ID is a private name space for filenames. Within any particular file group, all filenames must be unique, but files from different groups are automatically distinct from one another. In practice, this gives each application a *private* file space, where all the tasks that are part of the application can see all of the application files, but tasks from outside the application cannot.

Sometimes it is useful or even necessary to share visibility. The system call `set_group()` allows a task to obtain visibility into the file space of another group. The policy pertaining to `set_group()` is summarized as:

- Tasks belonging to Group 1 (sponsoring application) can use `set_group` to access files in any group, except for Group 0
- Tasks belonging to any other group can use `set_group` to access files in Group 15 (universal application), as well as their own group

In addition, any file in Group 15 can be directly accessed if its name is preceded by the slash (/) character. To open the file `FOO` in Group 15, specify `/FOO` in the `open()` statement without using a `set_group` operation.

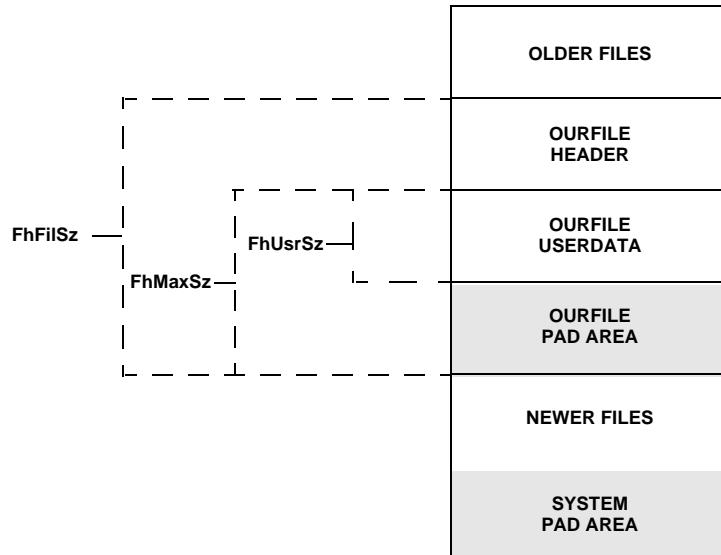
Analogous to the `set_group` call is `get_group`, which retrieves the current (temporary) group setting for the caller.

## Support for File Extension Areas

The Verix V file manager stores a file as a contiguous array of characters. When another file is created, it becomes another contiguous array, stored in memory immediately after the first. If the first array subsequently “grows” to accommodate new data, the newer file must be shifted. This means that updating one file can cause other files to *slosh* in memory. This design optimizes memory usage, but can significantly degrade system performance.

In Verix V, this problem is addressed with an optional file extension area that the user can specify to allow growth or shrinkage within a particular file without affecting other files in the system. The concept is simple: Provide a *pad area* at the end of the file and allow the system to adjust this pad area to accommodate changes in the amount of actual user data in the file.

The following discussion includes illustrations to clarify this concept. First, a typical file (*OURFILE*) is shown in [Figure 4](#) as it appears within the system, preceded by *older* files and followed by *newer* files.



**Figure 4      Typical File Space Usage**

Notice the pad area at the end of *OURFILE*. In Verix V, an expansion region is supported (shown as the greyed pad area) for each file. Within the file header, separate counts are maintained, *FhUsrSz* and *FhMaxSz*, to indicate the amount of user data with and without the pad.

Verix V file systems support this pad area for all files to ensure that each file begins on an integer boundary. This allows retention of all file headers on word-aligned boundaries.

Every file has a pad area defined, although it can be zero bytes long. For a *normal* file (that is, one without the expansion area feature enabled), the pad is no more than three bytes long—just enough to keep the next file word-aligned.

The main purpose of supporting the pad area is that it can be substantially larger than three bytes long. The file manager supports a call to define a *maximum space for user data* value. That is, you can update the *FhMaxSz* field maintained in the file header. Two calls provide access to this field:

- `long get_file_max(int handle);`
- `int set_file_max(int handle, long maxuserdatasize);`

The first of these calls returns the current value of *FhMaxSz*; the second assigns a new value to it. Note that there is a side-effect to setting *FhMaxSz*: An attribute bit, *ATTR\_NO\_GROW*, is defined for each file. *ATTR\_NO\_GROW* is closely tied to the functionality of the *FhMaxSz* field and its effect on inserting and deleting data within a file.

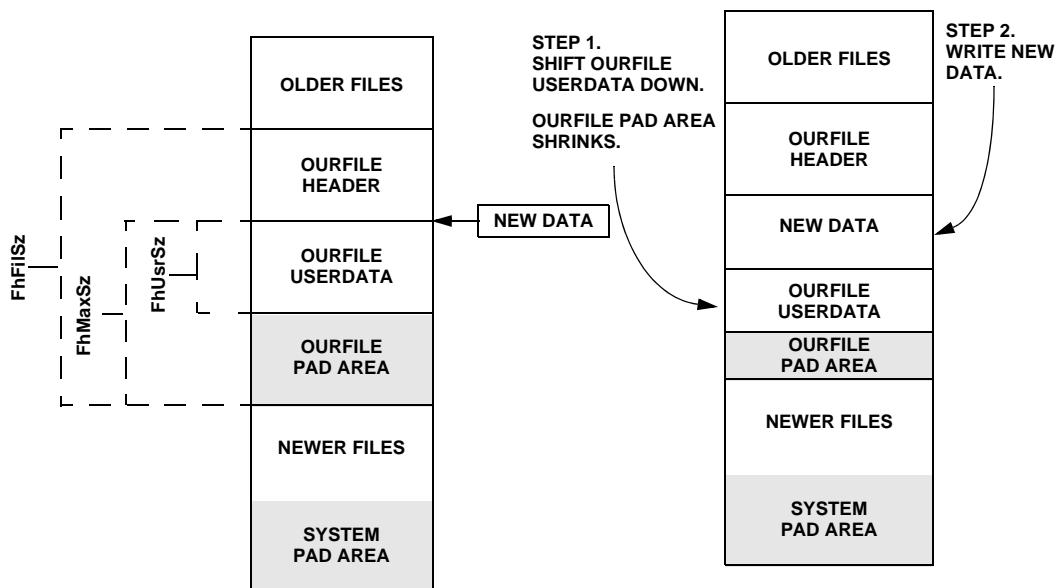
Several rules apply to the use of `set_file_max()`, since not every value for `maxuserdatasize` is valid:

- If `maxuserdatasize` is 0:
  - The pad area is minimized (to 3 bytes or less, just enough to ensure the file size is word-aligned), and
  - `ATTR_NO_GROW` is reset, allowing future writes to change the size of the file (and move newer files).
- If the specified `maxuserdatasize` is at least as big as the current number of user data bytes in the file but is less than the current setting of `FhMaxSz`:
  - The file shrinks to the new limit causing newer files to move, and
  - `ATTR_NO_GROW` is set, freezing the size of the file to this new limit.

Note that if you attempt to reduce the total file size to less than the size of the current data, the call is rejected, a result of `-1` returns with `errno` set to `EINVAL`.

- If the specified `maxuserdatasize` is the same as the current setting of `FhMaxSz` or larger:
  - The system attempts to expand the file, causing newer files to move, and
  - If sufficient space exists, `ATTR_NO_GROW` is set, freezing the size of the file to this new limit, and
  - If not enough space in the file system exists to accommodate the request, the call is rejected, a result of `-1` returned with `errno` set to `ENOSPC`.

Consider what happens to the file once `set_file_max()` is called to define a pad area for expansion. Assume that some data must be added to the file using one of the `write()` or `insert()` calls. As long as the amount of data added is less than the current size of the pad, the system reduces the pad by just the amount required to accommodate the new data without changing the total size of the file. [Figure 5](#) illustrates this process.



**Figure 5      New Data Added; Pad Area Shrinks**

In this example, some existing user data shifts down since the data is not being inserted at the end of the file; the new data is then written into the file. Note that no other files in the system are affected. In particular, newer files, which occur *after* this file in memory, do not get *sloshed* (that is, moved back and forth in memory) just because the size of the file is changed.

Furthermore, various `delete()` operations can be issued to analogously cause the pad area to grow without affecting the total size of the file and *without* causing newer files to move.

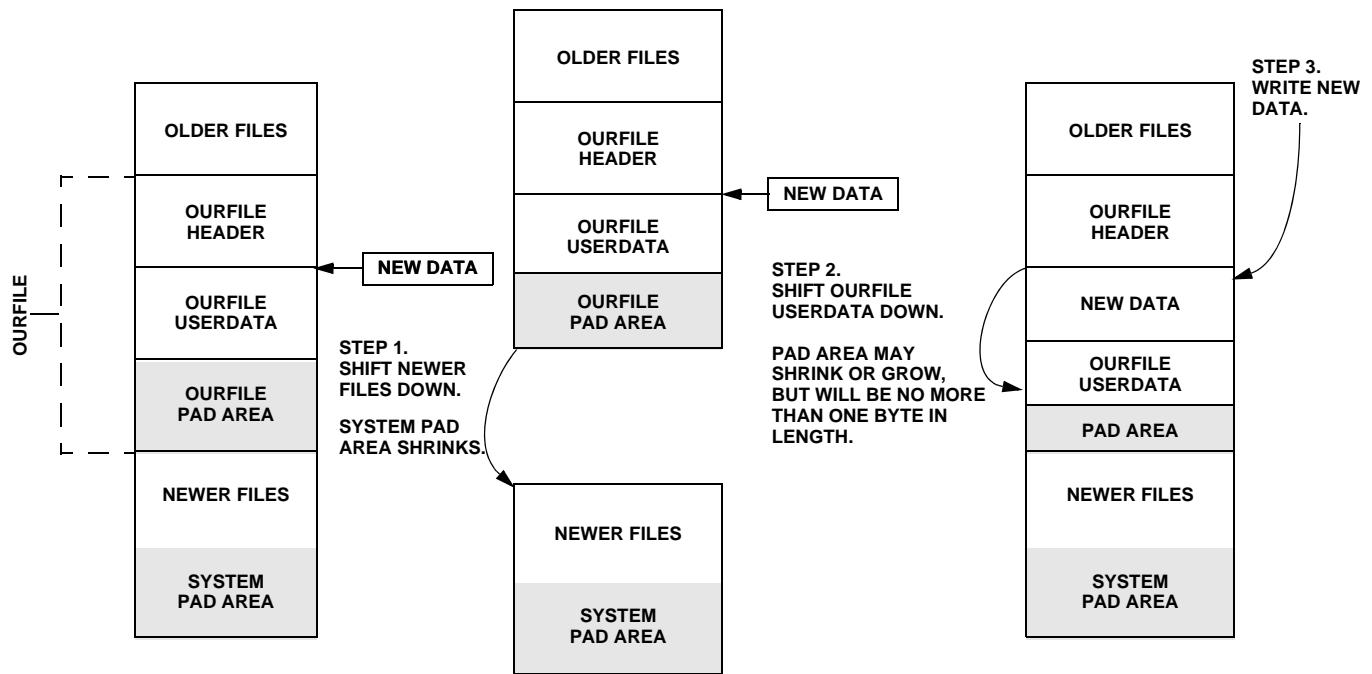
### Exceeding File Extension Area

An important issue to consider is what happens when the developer attempts to add a block of data larger than the designated pad.

If the `ATTR_NO_GROW` bit is set—which is normal after a call to `set_file_max()`—the `write()` or `insert()` call fails, a result of `-1` returns with `errno` set to `EFBIG`. Note that this error code indicates specifically that the file's expansion area would overflow; it does *not* imply that the system's expansion area is full.

If the `ATTR_NO_GROW` bit is not set, for example because it was explicitly reset with the `reset_file_attribute()` call, the system attempts to shrink the system pad area and move newer files to accommodate the requested expansion. If this process fails, a result of `-1` returns with `errno` set to `ENOSPC`. Otherwise, **OURFILE** expands and newer files *slosh*.

This process is illustrated in [Figure 6](#). Note that in the right-most section of the diagram, OURFILE USERDATA pad area is smaller. In fact, the OURFILE USERDATA pad is never greater than one byte after the system moves newer files. It is possible, however, that the pre-existing pad area was only 0-byte long, so in this “small” sense, it is possible for the pad to grow.



**Figure 6 Failed File Expansion**

Notice that the user does have the option of controlling the `ATTR_NO_GROW` bit directly. This is largely a by-product of the implementation.

Although there is some value to being able to interrogate the state of this feature using the `get_file_attributes()` call, there is no advantage to using either `set_file_attributes(..ATTR_NO_GROW)` or `reset_file_attributes(..ATTR_NO_GROW)`, and their use *not* recommended.

Finally, users of this feature should note the error code: `EFBIG`. The caller's `errno` is set to this value in only one situation: When an attempt to overflow the *maximum file size* previously set using `set_file_max()`. This error code informs the user that the previous estimate was incorrect; the user must either expand the file or turn off the fixed expansion area feature. In either case, the user should employ the `set_file_max()` call once more, either passing a larger size to expand the terminal's fixed-size area or passing a size of zero to disable the fixed-size limit (and allow newer files to slosh).

## Variable-Length Records

Basic file input/out in Verix V follows the Posix model in which files are treated as unstructured sequences of bytes. Verix 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\(\)](#), [write\\_vlr\(\)](#), and [write\\_cvlr\(\)](#)). 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. See [Figure 43](#) in [Appendix j](#) for an illustration of compressed character storage.

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  $\geq 0$  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 `EXIST`.

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.

#### 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.

## open()

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

### Prototype

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

### Parameters

filename	Pointer to a NULL-terminated string, up to 32 bytes long.
attributes	An integer that indicates the access attribute:
O_RDONLY	Opens the file for read-only access; the file cannot be written.
O_WRONLY	Opens the file for write-only access; the file cannot be read.
O_RDWR	Opens the file for read/write access; records can be read, written, inserted, or deleted.
O_APPEND	Opens the file with the file position pointer initially set to the end of the file.
O_CREAT	Opens a new file for write access.
O_TRUNC	Opens an existing file, truncates its length to zero, deletes previous contents.
O_EXCL	Returns error value if the file already exists; used with O_CREAT.
O_CODEFILE	Specifies that the file contains executable code. Normally this attribute is used by the download module and is not used by applications.

### 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.

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.

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.

---

**NOTE**



Since file handles are a limited resource, care should be taken in their allocation and use.

---

Note that there are several independent file systems in Verix V-based terminals. The most commonly used file system exists in SRAM. The RAM-based file system can be updated as well as read.

The flash-based file system can also be updated. Since the flash-based file system 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. Files in the flash-based file system are identified with the prefix F:.

**Read Files** The [read\(\)](#), [read\\_vlr\(\)](#), and [read\\_cvlr\(\)](#) functions transfer data from a file opened for reading to a buffer within the application's data area.

## read(), read\_vlr(), and read\_cvlr()

A successful call to these functions copies up to `count` bytes from the file to the address specified by `buffer`.

### Prototype

```
int read(int handle, char *buffer, int count);  
int read_vlr (int handle, char *buffer, int count);  
int read_cvlr (int handle, char *buffer, int count);
```

### Parameters

<code>handle</code>	Handle of the calling device.
<code>buffer</code>	Pointer to an array where the data is stored.
<code>count</code>	Determines the maximum value to read.
<code>read()</code>	Points to <code>bytes_read</code> bytes past its location before <code>read()</code> executed.
<code>read_vlr()</code> , <code>read_cvlr()</code>	Point to the next variable-length or compressed variable-length record in the file.

### Return Values

<code>Failure:</code>	–1 with <code>errno</code> set to EBADF: file not open (bad handle) or file currently locked by another user. –1 with <code>errno</code> set to EACCES: caller's buffer is not writable (for example, bad buffer address). –1 with <code>errno</code> set to EINVAL: the <code>count</code> parameter is too large.
<code>bytes_read</code>	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 Files** The [write\(\)](#), [write\\_vlr\(\)](#), and [write\\_cvlr\(\)](#) functions transfer data from an application's buffer to a file open for writing.

## write(), write\_vlr(), and write\_cvlr()

A successful call to these functions copies up to `count` 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(int handle, const char *buffer, int count);
int write_vlr (int handle, const char *buffer, int count);
int write_cvlr (int handle, const char *buffer, int count);
```

### Parameters

<code>handle</code>	Handle of the calling device.
<code>buffer</code>	Pointer to an array where the data is stored.
<code>count</code>	Determines the maximum value to write.
<code>write()</code>	Points to <code>bytes_written</code> bytes past its location before <code>write()</code> executed.
<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/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 VLR or CVLR record.
<code>write_cvlr()</code>	

### Return Values

<code>Failure:</code>	–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>ENOSPC</code> : Not enough memory to complete the request. –1 with <code>errno</code> set to <code>EACCES</code> : Caller's buffer is not readable (for example, bad buffer address). –1 with <code>errno</code> set to <code>EINVAL</code> : <code>count</code> parameter is too large. –1 with <code>errno</code> set to <code>EFBIG</code> : File expansion area is full. See <a href="#">set_file_max()</a> .
<code>bytes_written</code>	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.

## FILE MANAGEMENT

*write(), write\_vlr(), and write\_cvlr()*

**File Positioning** The [lseek\(\)](#), [seek\\_vlr\(\)](#), and [seek\\_cvlr\(\)](#) functions set the file position pointer of an open file to a specified location.

## **Iseek(), seek\_vlr(), and seek\_cvrlr()**

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_cvrlr()`, 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 handle, long offset, int origin);
int seek_vlr (int handle, long offset, int origin);
int seek_cvrlr (int handle, long offset, int origin);
```

### **Parameters**

`handle` Handle of the calling device.

`offset` Specifies the number of bytes to move the seek pointer from the specified starting point; used with `lseek()`; can be positive or negative.

`lseek()` can be positive or negative, and it specifies the number of bytes to seek forward or backward from the specified origin.

For the functions `seek_vlr()` and `seek_cvrlr()` the `offset` value *must* be positive, and it specifies the number of records to seek into the file.

### **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_cvrlr()`) 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.

## FILE MANAGEMENT

*lseek(), seek\_vlr(), and seek\_cvlr()*

Failure -1 with errno set to EBADF: file not open (bad handle), or file currently locked by another user

-1 with errno set to EINVAL: The origin is not SEEK\_SET, SEEK\_CUR, or SEEK\_END, offset is too large, or (for seek\_vlr or seek\_cvlr) offset is negative.

**Example**    bytes=lseek(handle,4L,SEEK\_SET);

## Insert and Delete Data

The [insert\(\)](#), [insert\\_vlr\(\)](#), and [insert\\_cvlr\(\)](#) functions insert data into a file opened for write access at the location of the file position pointer. The [delete\(\)](#), [delete\\_vlr\(\)](#), and [delete\\_cvlr\(\)](#) functions delete data from a file opened for write access at the location of the file position pointer.



- 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.
- The file extension feature (see [Support for File Extension Areas, page 76](#)) can be used to modify or eliminate the recommended restrictions.

## 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 handle, const char *buffer, int size);  
int insert_vlr (int handle, const char *buffer, int size);  
int insert_cvlr (int handle, const char *buffer, int size);
```

### Parameters

<code>handle</code>	Handle of the calling device.
<code>buffer</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:	<ul style="list-style-type: none"><li>–1 with <code>errno</code> set to <code>EBADF</code>: File not open (bad handle) or file currently locked by another user.</li><li>–1 with <code>errno</code> set to <code>ENOSPC</code>: Not enough memory to complete the request.</li><li>–1 with <code>errno</code> set to <code>EACCES</code>: Caller's buffer is not readable (for example, bad buffer address).</li><li>–1 with <code>errno</code> set to <code>EINVAL</code>: <code>count</code> parameter is too large.</li><li>–1 with <code>errno</code> set to <code>EFBIG</code>: File expansion area is full. See <a href="#">set_file_max()</a>.</li></ul>

## **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 handle, unsigned int count);  
int delete_vlr (int handle, unsigned int count);  
int delete_cvlr (int handle, unsigned int count);
```

### **Parameters**

`handle` Handle of the calling device.

`count` For `delete()`: The number of bytes to delete.

For `delete_vlr()` and `delete_cvlr()`: 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 `errno` set to EBADF: file not open (bad handle), or file currently locked by another user.

-1 and `errno` set to EINVAL: Invalid `count` 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 handle, unsigned int count);`

**Parameters**

`handle`    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).

## Retrieve File Information

The [get\\_file\\_size\(\)](#) function returns information about the file size. [get\\_file\\_date\(\)](#) returns information about the last update to the file. See also [dir\\_get\\_file\\_size\(\)](#) and [dir\\_get\\_file\\_date\(\)](#).

## get\_file\_size()

This call retrieves the file size as a 4-byte long integer returned in the indicated buffer. The file must be open *before* calling this function.

### **Prototype**

```
int get_file_size(int handle, long *filesize);
```

### **Parameters**

handle Handle of the calling device.  
filesize 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. See [Figure 5](#).

### **Return Values**

Success: 0  
Failure: -1 with errno set to EBADF: File not open.  
-1 with errno set to EACCES: The buffer specified in filesize 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 handle, char *yymmddhhmmss);`

**Return Values**    Returns a 12-byte timestamp in `buffer`. 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

## **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 errno set to EPERM: Checksums do not match.<br>-1 and errno set to EBADF: Invalid handle or file currently locked by another user.<br>-1 and errno set to ENOENT: File does not exist.<br>-1 and errno set to EACCES: 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
```

**Delete a File** The `_remove()` function deletes a specified file in the directory.

## [\\_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 and Unlock Files

Use `lock()` and `unlock` with open files associated with handle. Locking files prevents them 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.

Attempts to open or access a locked file fail with an EACCESS 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()

---

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 handle, long reserved1, long reserved2);
```

### Parameters

reservedn Reserved for future use.

### Return Values

Success: 0

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

-1 and errno set to EACCESS: The file is already locked.

## **unlock()**

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

### **Prototype**

```
int unlock (int handle, 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 a File** The [\\_rename\(\)](#) function renames a specified file in the directory.

## [\\_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.

### **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");`).

## **File Attribute Function Calls**

The function calls listed in this section set and return file attributes. See also, [File Directory Function Calls](#).

## get\_file\_attributes()

Returns the attribute byte for an open file. It is similar to [dir\\_get\\_attributes\(\)](#), except that the file is specified by `handle` rather than a name. See [Table 13](#) for a list of attribute bits.

See also, [dir\\_get\\_attributes\(\)](#), [set\\_file\\_attributes](#), [reset\\_file\\_attributes\(\)](#).

**Prototype**    `int get_file_attributes (int handle);`

### **Return Values**

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



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. See also, [set\\_file\\_max\(\)](#).

**Prototype**    `long get_file_max (int handle);`

**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 handle, 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 handle, 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. See [Support for File Extension Areas](#),

### NOTE



The file must be in RAM and open with write access.

The purpose of setting a maximum file size is not usually to restrict its size, but to improve performance. By default, files occupy only the amount of memory required for their current data, growing and shrinking as data is added or deleted. Since files are stored contiguously in memory changing the size of a file causes all files which follow it to be shifted to either make room for a growing file or to fill the gap left by a shrinking file. This *sloshing* occurs on every call to write, insert, and so on, and can have significant performance impact.

Setting a maximum file size freezes (not necessarily permanently) the memory space reserved for a file. The file can grow or shrink within this space without affecting other files. If a write would cause the file length to exceed the maximum set size, it fails and sets errno to EFBIG. At this point the application can increase the maximum size and retry the write.

If the file designated by handle does not have a maximum size set, then its physical size is increased to accommodate maxsize data bytes and the ATTR\_NO\_GROW attribute is set (see [Table 13](#)). This does not affect the logical length of the file; the extra space serves as an expansion area. If the file already has a maximum size, its value changes to maxsize and the file size adjusts accordingly. In either case, if maxsize is less than the current data length, set\_max\_size fails with errno set to EINVAL, and no change made. If there is not enough memory available to allocate maxsize bytes, set\_max\_size fails with errno set to ENOSPC.

Setting maxsize to zero shrinks the file size to the actual current length and clears the ATTR\_NO\_GROW attribute bit, restoring default sloshing behavior.

The current maximum size of a file is returned with [get\\_file\\_max\(\)](#). The ATTR\_NO\_GROW flag can be queried with [get\\_file\\_attributes\(\)](#), and can be changed by [set\\_file\\_attributes\(\)](#) or [reset\\_file\\_attributes\(\)](#), but this is not recommended. See also, [get\\_file\\_max\(\)](#).

### Prototype

```
int set_file_max (int handle, long maxsize);
```

### Return Values

Success: 0

Failure: -1 and errno set to EBADF: Invalid file handle or no write access.  
-1 and errno set to EINVAL: Size shorter than current data length or file in flash.  
-1 and errno set to ENOSPC: Not enough memory to satisfy request.

### Example

The linked code example creates a data file with a maximum size 8 KB file.

**Close Files** The [close\(\)](#) function closes the file specified in the handle parameter.

## **close()**

---

Each file opened by an application must also be closed when access to the file is no longer needed.

**Prototype**    `int close(int handle);`

**Parameters**

`handle`    Once the file is closed, the `handle` is no longer valid and all internal resources used by the handle are released.

**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 V 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 ZonTalk 2000 is used to download a keyed file, keys *must* be 7 bytes or less. ZonTalk 2000 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.

**Keyed File Function Calls** The `getkey()` and `putkey()` function pair retrieves data from and stores data to keyed files.

## getkey()

Retrieves data associated with a given key value.

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

**Parameters**

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

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 V 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, `buffer` 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 *buffer, 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 ZonTalk downloads).
<code>buffer</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



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, `buffer` 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.

## **File Directory Function Calls**

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.

## **dir\_get\_attributes()**

---

Provides access to the file attribute bits that the file system maintains. See [Table 13](#) 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 13](#) is returned.

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\_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**

`filename`    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` 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` 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.



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 *buffer);`

**Parameters**

`buffer`    Contains the name of the file returned from a prior call to `dir_get_first()`.

**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 `buffer` parameter, such as `I:` for internal memory or `F:` for flash memory.

---

## 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`.

### NOTE



The structure `fs_size` is defined in the header file `svc.h`.

### Prototype

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

### Parameters

`drive` File system location.

### Return Values

On return, `*fs` is filled with information about the file system.

`short count` Indicates the number of files in the directory.

`long InUse` Contains the amount of memory used by the file system, including memory manager overhead (in bytes).

`long Avail` Contains the amount of remaining space in the file system.

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

-1 with `errno` set to `EACCES`: Either `drive` or `fs` is an invalid pointer.

## **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 13](#) descriptions of the attribute bits, and [dir\\_set\\_attributes\(\)](#) for an example.

See also, [reset\\_file\\_attributes\(\)](#), [dir\\_get\\_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 13](#) for a list of attribute bits.

### **Prototype**

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

**Table 13 Attribute Bits and Function**

Bit	Name	Function
0	ATTR_READONLY	Intended to mark files that cannot be written.  <b>Note:</b> This function is not currently supported.
1	ATTR_NO_CKSUM	Verix V 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 V 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.

## SVC\_RAM\_SIZE()

Returns the amount of RAM memory, in kilobytes, installed in the terminal. See also, [dir\\_get\\_sizes\(\)](#) and [SVC\\_FLASH\\_SIZE\(\)](#).

**Prototype**    int SVC\_RAM\_SIZE (void);

**Example**

## 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

All Verix V-based terminals include support for a separate user file system stored in flash memory. Any file with F: preceding the filename is assigned to flash. In many respects, it behaves similar to the default RAM file system previously discussed. the following is an overview of features unique to the flash file system.

The most significant difference between flash and RAM is that the space used by a flash file is not automatically recovered when the file is deleted. That space is not immediately available for use by other files. Many applications use flash files to hold only codefiles and parameter files downloaded from a host PC. For this kind of application, all flash files are read-only, and are handled exactly like files in the RAM file system.

### Restrictions on Flash Files

There are several restrictions on flash files:

- Only one flash file can be open with write access at any time. If a second flash file is open with write access, the open returns EACCES. Multiple flash files with read-only access can be open at the same time.
- If a file that exists is open with write access, the existing file is deleted and a new file created. That is, the file is truncated; it is important to include the O\_TRUNC access option in the `open()` statement.
- When a flash file is removed, the space used by the file is not immediately recovered, so it is not available for use by other flash files. To recover this space and make it available, the flash file system must be *coalesced* to bring all valid flash files together at the beginning of the flash file system, with all the remaining space erased and available for new flash files. This process is invoked using `dir_flash_coalesce()`.
- Any file function that only requires a read access to a file returns EBUSY if the file requested is not closed by its creator (that is, the write-access handle is *not closed*). These functions include `open()` with O\_RDONLY access, `read()`, all `status()` functions, and some directory functions (`dir_xxx()`).
- If a flash file is opened as read-only and a new `open()` with write access to that file is issued, that file is tagged as deleted by the write handle, and a new file is created. The handle with read-only access is still accessing the deleted file until a `close()` is done on that handle. Any `open()` after the `open()` with write access gets the new file.

### File Manager Function Calls Supported in the Flash File System

- `open()`— multiple read handles, one write handle
- `close()`
- `read()`
- `write()`
- `lseek()`
- `status()` functions (on open file):

- `get_file_size()`
- `get_file_date()`
- `get_file_attributes()`
- `get_file_max()`
- Control functions (on open file):
  - `lock()`/`unlock()`
  - `set_file_attributes()`
  - `reset_file_attributes()`
  - `put_file_date(int handle, char *yyyymmddhhmmss)` — same as `dir_put_file_date()`, except file is specified by handle instead of filename
  - `dir_get_first() *drive parameter`
  - `dir_get_next()`—returns drive as part of filename
  - `_remove()`
  - `dir_get_file_sz()`
  - `dir_get_sizes() *drive parameter`, count includes deleted files
  - `SVC_CHECKFILE()`
  - `dir_get_file_date()`
  - `dir_get_attributes()`
  - `get_dir_file_hdr()`
  - `get_dir_file_size()`



\*`drive` parameter—the above functions require the `char *drive` parameter to distinguish between the RAM and flash file systems.

### File Manager Function Calls Not Supported in the Flash File System

The following functions return EINVAL if a file handle for a flash file is passed as a parameter:

- Control functions (on open file):
  - `delete()`
  - `insert()`
  - all VLR and CVLR functions

The following functions return EBADF if a flash filename is passed as a parameter:

- Directory functions (on closed file):
  - `rename()`

- `dir_put_file_date()`
- `file_append()`
- `file_copy()`
- `dir_set_attributes()`
- `dir_reset_attributes()`

## Open and Close

When a new flash file is opened, only the filename is written to the flash. There is only support for one open flash file with write access at any time.

To open a flash file with write access, the file must have the `O_APPEND` bit set, and either `O_RDWR` or `O_WRONLY` set. If the file to open already exists, `O_TRUNC` is also required; if it is a new file, `O_CREAT` is required. An attempt to open a second file with write access at the same time returns `errno` set to `EACCES`.

When opening an existing closed flash file for write access, the existing file is deleted. A new version of the file is created at the next available address in flash with a size of zero. The contents of the previous file are lost. The file is not actually removed from the file system until a `dir_flash_coalesce` call is performed; the filename is changed to tag the file as deleted.

Multiple flash files can be opened with read-only access (with the `O_RDONLY` attribute bit set). To open a file with read-only access, the file must exist and must be closed after creation. If it was not closed, the open with read-only access returns `EBUSY`.

## Flash Rebuild

If a flash file was written to and was not closed on a memory clear, the file is tagged deleted. This means that if the terminal loses power or `SVC_RESTART()` or `dir_flash_coalesce()` is called before a flash file with write access was closed, that file is tagged as deleted at restart.



The file count in the flash file system header includes all the deleted but *not erased* files. This is because to get to the current files, the files tagged deleted must be skipped. All space must be accounted for.

## Flash Erase

The only way to erase flash is to call `dir_flash_coalesce()`. During a file operation to a flash file if there is not enough room to write the header on `open()` or to write the data on `write()`, the function fails with `ENOSPC`. When this occurs, one of the two following operations *must* be performed to allow further writes to the flash file system. Also,

- The coalesce process deletes any file with write access not closed prior to the coalesce. The `dir_flash_coalesce_size()` function returns `EBUSY` if there is an open file, so call this function prior to performing the coalesce.
- This is a time-consuming process; use accordingly.

## 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 that one sector (64 KB) of flash memory is reserved for use as a swap sector while performing this operation. During the coalesce process, all interrupts are disabled. The terminal restarts when the coalesce is complete because code files may have moved in the process. `dir_flash_coalesce()` cleans up the flash file system by temporarily copying active files to a work area, erasing flash, and rewriting the files so that they are packed contiguously at the start of memory. Deleted files are eliminated by this process and the space they occupied becomes available for new files. Any files open for write are deleted. See also [dir\\_flash\\_coalesce\\_size\(\)](#).

Flash memory cannot be reused simply by overwriting it with new data. It must first be erased, a whole sector at a time. Therefore when a flash file is deleted it is not physically removed, but simply marked as deleted in the directory. The space it occupies is not available for new files. New files are always written at the end of the used portion of flash memory. When an existing file is rewritten (opened with `O_TRUNC`), it is marked as deleted and a new file with the same name is created. (Note that this is what happens when a new version of a file downloads.)

The amount of space available to be reclaimed by a coalesce can be determined by calling [dir\\_flash\\_coalesce\\_size\(\)](#).

The coalesce operation can take several seconds. Furthermore, the terminal restarts after the operation is complete, so the function never returns.

`dir_flash_coalesce()` would typically be called only after a download or at application startup as shown in the following linked example.

**Prototype**      `int dir_flash_coalesce (long *size);`

**Return Values**      This function returns EBUSY if there is a file with write access that has not yet been closed.

**Notes**      Defragmentation moves closed and bounded files, as well as a final unbounded file. This means that coalesce can be called while there are open output files and that the writes continue to the new location of the file when defragmentation is finished. Based on the size of the file system and its fragmentation, this could be a relatively time consuming process; use accordingly.

**Example**      An application could use the linked code example to clean up flash at terminal startup. If the coalesce function is called, the application restarts and the code runs again, but this time `space` is zero.

## **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**

`size` 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**

Success: 0; no files open for writing.

Failure: -1 with errno set to EBUSY: No files are open for writing.

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

### **Example**

See the linked code example in `dir_flash_coalesce()`.

## SVC\_FLASH\_SIZE()

Returns the amount of flash memory, in kilobytes, installed in the device. See [SVC\\_RAM\\_SIZE\(\)](#) for an example. See also [SVC\\_RAM\\_SIZE\(\)](#), [dir\\_get\\_sizes\(\)](#), [dir\\_flash\\_coalesce\\_size\(\)](#).

**Prototype**

```
int SVC_FLASH_SIZE (void);
```





## CHAPTER 4

### System Configuration File

Verix V-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 system mode 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 V 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

Table 14 lists the currently available Group 1 CONFIG.SYS environment variables and their descriptions. [Environment Variables Application Information](#) further discusses variable usage. CONFIG.SYS files residing in other groups are for application use only.

**Table 14      CONFIG.SYS Environment Variables**

Variable	Description
*APNAME	In the ENHANCED UI (new system mode), this is the name of the application residing in a GID.  <b>Note:</b> During FULL download, *APNAME is displayed in the confirmation process. If *APNAME is not defined, APPLICATION is displayed, instead.
*ARG	Arguments for the startup application specified in *GO.
*B	Sets the number of communication buffers. Minimum is 1; maximum is 256 (default).
*BCM	Sets the Backward Compatibility Mode used on Vx670 terminals. This allows the legacy 8-line display applications to use the larger 16-line display terminal without modification.  The values are:  0 = Backward compatibility is set to OFF (default). 1 = Sets the application in lines 5-12, without arrow extensions. 2 = Like 1, but with arrow extensions at the bottom of the screen.
CHKSUM	Disables checksum verification on startup. CHKSUM = 2 checksum disabled; default is checksum enabled.
*COMBO	Sets the application group to use a modem or TCP/IP. 0=modem,1=TCP/IP
COM2HW	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.  <b>Note:</b> 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	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.  <b>Note:</b> 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.

**Table 14      CONFIG.SYS Environment Variables** (continued)

Variable	Description
*COM2RB	Indicates the size of the asynchronous receive buffer on COM2 only. If this variable doesn't exist, the default size (1024 bytes) is taken.  <b>Note:</b> The buffer size ranges from 1024 to 8192 bytes. If any value is specified outside this range, the default value is used.
*COM3RB	Indicates the size of the asynchronous receive buffer on COM3 only. If this variable doesn't exist, the default size (1024 bytes) is taken.  <b>Note:</b> The buffer size ranges from 1024 to 8192 bytes. If any value is specified outside this range, the default value is used.
*DARK	Sets the scale factor for the print strobe activation time to compensate for different types of thermal papers.
*DBMON	Configures the baud rate for the debug monitor (in the form <i>p</i> <i>b</i> ). Valid values for <i>p</i> are 0–2, setting the COM port; default is 1 (COM1). Valid values for <i>b</i> are 0–13; default is 9 (115,200 kbps). Baud rate setting is ignored when configured for USB ( <i>p</i> = 0).
*DEFFRAG	Sets automatic flash defragmentation on terminal power up.
*DIAG	Executes a diagnostic or key-loading program <i>once</i> in terminal system mode; allows a diagnostic program to run <i>once</i> when security is preventing access.
*DOT0 <sup>a</sup>	Indicates the maximum number of dots to print per strobe when the nRAD_HPWR signal line is low.  <b>Note:</b> This variable does not apply to Vx670.
*DOT1 <sup>a</sup>	Indicates the maximum number of dots to print per strobe when the nRAD_HPWR signal line is high.  <b>Note:</b> This variable does not apply to Vx670.
*FA	When this variable is set to 1, all signature files are retained in terminal memory; default is 1.
*FILE	Maximum number of files that can be open simultaneously. Minimum is 10; maximum is 224; default is 30.
*GO	File Group 1 application filename to execute.
*GKE	Indicates the type of event that has to be generated when the user has pressed green key.
*IPPMKI	Sets communication parameters for internal PIN pad key loading from system mode.
*MA	This is used in ENHANCED mode. It's set when MULTI-APP download is chosen.
*MERR	Modem profile load error code.
*MN	Modem configuration file name.
*OFF	An automatic transition from idle (sleep) to OFF occurs if the unit is continuously idle for five minutes, OFF variable is set to adjust this time period.

**Table 14      CONFIG.SYS Environment Variables** (continued)

<b>Variable</b>	<b>Description</b>
*OFFD	Indicates the amount of time that OS will delay 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.
*PIPE	Controls the number of pipe handles available. Minimum is 0; maximum is 256 (default).
*POW	Specifies the number of milliseconds needed to achieve SLOW MODE (sleeping) state which involves a sustained period of idleness for applications and drivers.
*PRNT	If set to zero in group 1, will prevent download errors from printing.
*PRTFNT	Specifies the amount of SRAM allocated for printer font tables (in 1-KB increments). The range is 0 to 256, and the default is 64.
*PRTLGO	Specifies the amount of SRAM allocated for printer logos (in 12-KB increments). The range is 0 to 10, and the default is 1.
*PW	Group access system mode password; password for the current group.
*SMDL	Enables poll-for-direct download during the start-up sequence before displaying the copyright screen.
*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	System mode entry password.
*SMUI	Indicates which system mode is in use. This can be set by selecting either ORIGINAL UI or ENHANCED UI from the System Mode Menu 8 (or System Mode Menu 4 on Vx670). This can also be edited directly.  The values are:  0 = ORIGINAL (default). 1 = ENHANCED.
*TIME	Number of system timers; default 10: used with <code>set_timer()</code> .
*UNZIP	Group 1 CONFIG.SYS file. Automatic decompression of only one archive.  If *UNZIP is set, the zip archive automatically decompresses at startup.  UNZIP
UNZIP	Limited results about the file decompression process 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.

**Table 14      CONFIG.SYS Environment Variables** (continued)

Variable	Description
*USBMEM	Specifies the amount of memory reserved for buffer space. By default the memory reserved is 16KB.
*VALID	List additional groups to search during file authentication for certificate and signature files using VeriShield.
*ZA	ZonTalk application ID.
*ZP	ZonTalk host telephone number; can use embedded dialing control characters.  <b>Note:</b> This variable is used to hold phone numbers for modem downloads, whereas for TCP/IP downloads it holds the IP address and port number of the VeriCentre host. The *ZP variable should be assigned appropriately prior to selecting either modem or TCP/IP download.
*ZT	ZonTalk terminal ID.
*ZINIT	External modem initialization string.  Default is ATM0V0&D2. <ul style="list-style-type: none"><li>• V0 - sets terse mode (numeric responses),</li><li>• &amp;D2 - drop DTR to hang up.</li></ul>
*ZRESET	External modem reset string.  Default is ATZ0.
*ZRESP	External modem connect response.  Default is CONNECT 2400.
*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.
*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.
*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 Vx terminals, and RS-232 for Vx PIN pads.
*AKM	Sets the alpha character entry mode on Vx810 PIN pad. The OS operates in cell phone alpha character mode if *AKM=CP. If it has any other value or is not present, the OS operates in classic mode.

**Table 14** CONFIG.SYS Environment Variables (continued)

Variable	Description
*CPAD	When the Vx810 operates in cell phone alpha mode, the OS translates the repeated key presses that take place within 1.5 seconds of each other (default) to the next alpha character in the alpha shift sequence for that key. If a key is pressed more than 1.5 seconds after the last, the OS returns the key as input.
Additional variables	Terminals can include several other CONFIG.SYS variables. Also, the application can create several application-specific variables. Consult your application specification for additional CONFIG.SYS variable definitions. Remember that the *name convention is used by system variables. Use the # character to preserve a variable between downloads.

- a. This variable does not apply to Vx670.

On Vx810, the application may set the battery and power CONFIG.SYS variables but they do not affect the Vx810 platform.

## 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 15](#).

**Table 15** Verix V Device Handles

Device	/dev Name	<svc.h> Variable
Magnetic card reader	/dev/mag	DEV_CARD
Real-time clock	/dev/clock	DEV_CLOCK
Beeper	/dev/stderr	DEV_BEEPER
Console (keypad and display)	/dev/console	DEV_CONSOLE
COM port 1	/dev/com1	DEV_COM1
COM port 2	/dev/com2	DEV_COM2
COM port 3	/dev/com3	DEV_COM3
COM port 4/Integrated thermal printer	/dev/com4	DEV_COM4
COM port 5	/dev/com5	DEV_COM5
COM port 6	/* com 6 */	DEV_COM6
COM port 8	/* com 8 */	DEV_COM8
COM port 9	/* com 9 */	DEV_COM9
COM port 10	/* com 10 */	DEV_COM10
Mag card	/* mag card */	DEV_CARD
Barcode reader	/* bar code reader */	DEV_BAR
MSO300 Biometric	/* MSO300 Biometric device*/	DEV_BIO
CTLS	/* Contactless device */	DEV_CTLS
USB keyboard	/* USB Keyboard HID converted to make and break code*/	DEV_KYBD[

**Table 15      Verix V Device Handles (continued)**

Device	/dev Name	<svc.h> Variable
USB host	/* PP1000SE and Vx810 device */	DEV_USBSER
Semtek device driver	/* Semtek device driver */	DEV_SEMTEK
Customer smart card	/dev/icc1	DEV_ICC1
Merchant SAM	/dev/icc2	DEV_ICC2
Merchant SAM	/dev/icc3	DEV_ICC3
Merchant SAM	/dev/icc4	DEV_ICC4
Merchant SAM	/dev/icc5	DEV_ICC5
Merchant SAM	/dev/icc6	DEV_ICC6
USB External Ethernet	/dev/eth1	DEV_ETH1
USB Internal WiFi	/dev/wln1	DEV_WLN1
USB Client	/dev/usbd	DEV_USBD

## Environment Variables Application Information



**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 '\*'.

### \*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). 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 RAM is not limited.

**\*BCM**

By designating the group-wide `*BCM` variable, user can modify the application code to include a call to `set_bcm(char Mode)` at initialization. The parameter mode can be set to 0, 1, or 2.

<code>set_bcm(0)</code>	Set BCM off.
<code>set_bcm(1)</code>	Set BCM on, no arrow extensions.
<code>set_bcm(2)</code>	Set BCM on, with arrow extensions.

**NOTE**

The `set_bcm()` must be called after the application opens the console.

**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 system mode). 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.

**\*COMBO**

Sets the application group to use a modem or TCP/IP. 0=modem,1=TCP/IP

**\*COM2HW**

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** 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.

**\*COM2RB** Indicates the size of the asynchronous receive buffer on COM2 only. If this variable doesn't exist, the default size (4096 bytes) is taken. The buffer size ranges from 4096 to 8192 bytes. If any value is specified outside this range, the default value is used.

**\*COM3RB** Indicates the size of the asynchronous receive buffer on COM3 only. If this variable doesn't exist, the default size (1024 bytes) is taken. The buffer size ranges from 1024 to 8192 bytes. If any value is specified outside this range, the default value is used.

**\*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).

**\*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 system mode flash defragment function is only available manually.

**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 system mode; allows a diagnostic program to run *once* when security is preventing access.

**\*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.

**\*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).

**\*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. System mode 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.

**\*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.

**\*IPPMKI—Internal PIN Pad Communications Parameters**

This variable sets communications parameters for the internal PIN pad key loading from system mode (see [Appendix c, IPP Key Loading](#)). The value can specify a baud rate or the following flags:

- |                 |             |
|-----------------|-------------|
| • E A7E1 format | • D Set DTR |
| • O A7O1 format | • 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.

**\*MA**

This is set when MULTI-APP download is chosen. \*ZA is set to \*MA when MULTI-APP is chosen in the ENHANCED UI.

**\*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 16](#) lists the \*MERR error code values.

**Table 16 \*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:  •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

**\*MN** Verix V 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 RAM 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.

<b>*OFFD</b>	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.
<b>*PIPE</b>	Controls the number of pipe handles available. Minimum is 0; maximum is 256 (default).
<b>*POW</b>	Indicates the amount of time (milliseconds) that Verix V 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).
<b>*PRTFNT</b>	Specifies the amount of SRAM 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.
<b>*PRTLGO</b>	Specifies the amount of SRAM 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.
<b>*PW—Password</b>	File group access password. System mode 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.

---

<b>*SMDL—System Mode Download</b>	This flag enables polling for direct download during the start-up sequence before displaying the copyright screen. Supported values are: <ul style="list-style-type: none"> <li>• *SMDL=0 do not poll (default)</li> <li>• *SMDL=1 (for developers only) poll for download</li> </ul> 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.

**\*SMPW—System Mode Password**

Setting this variable sets the system mode entry password. As with \*PW, \*SMPW is not stored in CONFIG.SYS.

**\*SYSCHK**

Allows the user to select a new time to run the system check.

**\*TIME—Set Timers**

Sets the number of system timers. By default, 30 timers are available. This number can be increased to 200 (maximum) by setting \*TIME=200. Timers are shared by the OS and all user tasks, so the number available to a particular application is <\*TIME. See [set\\_timer\(\)](#).

**\*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.

**\*USBMEM**

Sets the memory size of the USB device. Since USB devices can transfer a large amount of data in a short time, a significant amount of memory must be reserved for buffers and buffer management structures. By default the buffer size is 16KB, but it may be changed using the new CONFIG.SYS variable \*USBMEM, which allows the user to specify the amount to reserve in kilobytes.

**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.

**\*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 V OS 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`

**\*Z Series—  
ZonTalk 2000  
Control**

Sets the COM port rate for downloads (see [Appendix 11, Conexant Modems \(Banshee and Eisenhower\)](#)). The following four entries in the CONFIG.SYS file control baud rates for application downloads (*only*) with VeriCentre download application:

- \*ZA=xxxxxx      VeriCentre application ID (name).
- \*ZP=xxxxxx      VeriCentre download telephone number. Can use embedded dialing control characters. Must be a valid phone number. For an IP download, a valid IP address, including port number, is required.

\*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 Verix V OS 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 block size for download.

**\*ZINIT**      External modem initialization string.

Default is ATM0V0&D2.

- V0 - sets terse mode (numeric responses),
- &D2 - drop DTR to hang up.

**\*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.

**\*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.

**\*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.

**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:
- ZonTalk 2000 only: Keys must be  $\leq 7$  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 *buffer, int bufsize);
```

### **Parameters**

*key	Zero-terminated string that contains the name of the environment variable to retrieve. Maximum characters allowed is 32.
*buffer	Array where the current setting of the environment variable is stored.
bufsize	Maximum number of characters to store in buffer. Values longer than defined are truncated. The maximum length of an environment variable is 128 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 V 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 (...)
```



## CHAPTER 5

# Multitasking

The Verix V OS allows multiple application tasks to share the CPU, using simple “round-robin” scheduling. See [Figure 3](#) for a block diagram of the Verix V OS.

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
- Pipes

## Verix V Application Architecture

### Tasks

In the Verix V 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.

### Task Startup

In Verix V-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.

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 V-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 V 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 V-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.

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.

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.

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 handle, 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.

## 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.

## 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 V-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()` 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.

## **\_exit()**

Terminates the calling task. Task memory is reclaimed by the system; any open files or devices are closed.



**NOTE**

The functionality of this call differs from Verix 68K functionality in that `_exit()` supports arguments in Verix V.

**Prototype**    `int _exit(int unused);`

**Example**

## 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**

## **get\_task\_id()**

Retrieves the task number. Task numbers are assigned sequentially as new tasks are created. These numbers are reused when a task exits. Task 1 is the system mode 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**

## 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 <a href="#">SVC_WAIT()</a> .
TASK_EXIT	-1	Task has exited.
TASK_DEBUG	-128 (0x80)	Task is under control of the debugger (see <a href="#">TASK_DEBUG</a> ).
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 V OS 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 RAM 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**

routine	Name of the routine.
parameter	Parameter to the routine which you want to run.
stacksize	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\_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***

## Application Threads and Semaphores

The Vx570 terminal supports 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

## Semaphore Example with User Events

### Example

## Semaphore Application Function Calls

The following function calls listed in this section are semaphore application function calls.

## \_exit()

Terminates the process. 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 exitcode);`

**Parameters**    `exitcode`    Return code.

**Return Values**    Success:    0  
                    Failure:    Any other value.

## open()

---

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

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.

**Prototype**      `int open(char *path, int parms);`

**Parameters**

`path`      Path of the file or device.

`parms`      The parameter used to open the file or device.

**Return Values**

Success:      Positive value.

Failure:      Negative value.

## 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**

sem\_t \*sem      Pointer to a semaphore structure.

value      The initial value for semaphore.

### **Return Values**

Success:      0

Failure:      -1 with `errno` 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\_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 `errno` set to:  
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.



Use `errno` with caution as it is a global variable shared by other threads.

## **sem\_post()**

Frees the semaphore for general use and returns the handle immediately.

**Prototype**    `int sem_post(sem_t *sem);`

**Parameters**    `sem_t *sem`      Pointer to a semaphore structure.

**Return Values**

Success:      0

Failure:       -1 with `errno` 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.

---

## 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.

In practice, most tasks do not need more than one pipe, and some tasks do not need any. To conserve RAM space, determine on a system basis how many pipes are required and ensure that the `*PIPE` variable is set accordingly.

## 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

### Prototype

```
int status = pipe_init_msg (int pipehandle, int maxmessages);  
int status = pipe_init_char (int pipehandle, int maxcharacters);
```

### Parameters

maxmessages	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).
maxcharacters	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.

## **close()**

---

Closes open pipe handle.

**Prototype**    `int close(handle);`

**Parameters**    `handle`              Handle of pipe to close.

**Return Values**    Always returns 0.

**Action**    If `handle` is not found in the list of open pipes, only return 0; if found, mark this pipe closed. Any data in the FIFOs remains in the FIFOs after `close()`; it is *not* purged.

## open()

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.

### Prototype

```
handle = open(const char * pipename, int opentype);
```

### Parameters

- `pipename` There are two ways to identify this parameter:  
    Statically defined: In the form "`P:name`," where `name` is the name (an ASCII string of up to 7 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 eight characters of the name (not counting `P:`) are significant.
- `opentype` Currently ignored.



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.

### Return Values

- Success: The handle of the created pipe.
- Failure: –1 with `errno` set to `ENOMEM`: Not enough memory to allocate needed data structures.  
–1 with `errno` set to `EACCES`: The `pipename` parameter is not readable.

## pipe\_connect()

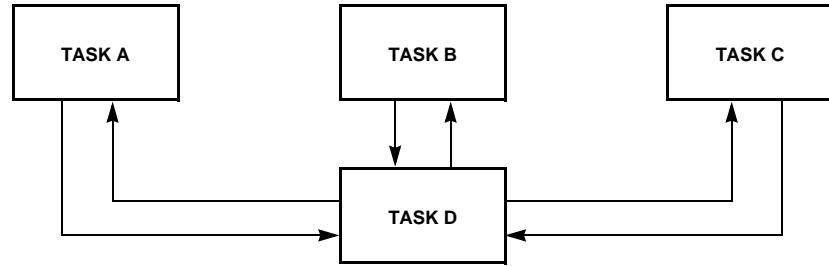
This is the *plumbing* operation. It allows data to be written using one handle, and subsequently to be read using the other handle.

### Prototype

```
int status = pipe_connect(int pipehandle, int targetpipehandle);
```

The second handle must be obtained from a separate pipe open( ) command.

As an example, consider the configuration in [Figure 7](#). 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 7 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 <code>open()</code> .
targetpipehandle	Destination (the handle of a pipe opened by another task).



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 <code>errno</code> set to <code>EINVAL</code> : Target is invalid or the pipe is not configured. -1 with <code>errno</code> set to <code>EBADF</code> : Caller does not own the <code>pipehandle</code> , <code>targetpipe</code> is not open or is not configured.

#### Example

## [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 handle, 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 handle, 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\_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 handle);`

### **Return Values**

Failure: -1 and errno set to EBADF: The handle is not valid, if is not owned by the caller, or is not configured.

### **Example**

## read()

Reads data from the open pipe handle.

### Prototype

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

### Parameters

handle	Handle of pipe to read from.
buffer	Storage area for bytes or message to be read.
count	Size of the buffer.

### Return Values

If `count` is set to 0, 0 returns. If `count` is greater than 0 and the pipe is a *character* pipe, read from this handle's read FIFO either until `count` 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.

### NOTE



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.

Success: The number of bytes actually read from the pipe.

Failure: -1 and `errno` set to `EBADF`: Pipe handle is invalid, not owned by caller, or is not configured.

-1 and `errno` set to `EACCES`: Buffer is not writable.

## write()

Writes data to the open pipe.

**Prototype**    `int write(int handle, const char *buffer, int count);`

**Parameters**

`handle`   Handle of pipe to write to.

`buffer`   Storage area for bytes or message to be written.

`count`   Number of bytes to write to the pipe.

For a *character* pipe, write `count` 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. Signal `EVT_PIPE` to the other normal completion trap to the other handle associated with this pipe. That is, the pipe specified in the previously issued `pipe_connect()` command. The return value is the number of bytes written.

**Return Values**

Success:   Number of bytes actually written to the pipe.

Failure:   –1 and `errno` set to `ENOMEM`: Not enough memory to allocate for the FIFOs.

–1 and `errno` set to `EBADF`: Pipe not owned by caller or is not configured.

–1 and `errno` set to `EBADF` (message pipes only): Not enough memory to allocate for the message.

–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.

## **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.



## CHAPTER 6

### Event Handling

Verix V 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 Verix V event word for the application is set to 0 as part of the process of waking up the application that was blocked. In this way, the application can manage new input, issue another `wait_event()` call, then sleep until more input arrives.

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 17 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 17** Defined Events

Name	Device	Description
EVT_ACTIVATE	Console	Console ownership returned to application. See <a href="#">set_hot_key()</a> in Chapter 7 for more information.
EVT_BAR	Bar Code Reader	Input available <a href="#">set_hot_key()</a> .
EVT_CLK	Clock	1-second event (only available to the task that opened the clock).
EVT_COM1	COM1	Input available on COM 1.
EVT_COM2	COM2	Input available on COM 2.
EVT_COM3	COM3	Input available on COM 3.
EVT_COM4	COM4	Input available on COM 4.
EVT_COM5	COM5	Input available on COM 5.
EVT_COM6	COM6	Input available on COM6 on Vx670 terminal only.  This is also available on Vx570 terminal that supports 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. See <a href="#">set_hot_key()</a> in Chapter 7 for more information.
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.

**Table 17      Defined Events** (continued)

Name	Device	Description
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.

## Event Function Calls

The following functions implement the exception-handling system.

## **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 `set_timer()`.

**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, and
- it does not clear the event mask.

## **post\_user\_event()**

Call that allows posting of an immediate event from one thread to another thread or task. The OS maintains a new 32 bits user events field for each task. This call combines the currently set bits with those passed as user bits, using a binary-OR operator.

**Prototype**    `post_user_event(int user_task_id, int user_bits);`

### **Parameters**

user_task_id	User task ID.
user_bits	A bit which is sent by one task to another task.

### **Return Values**

Success	0
Failure	-1 with caller's errno to <code>EINVAL</code> , indicates either 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.

**Notes**    • `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

<code>needed_events</code>	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 new “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\_timer()

Schedules an event to occur after a specified delay.

**Prototype**    int set\_timer(long msecs, long eventmask);

**Parameters**

long msecs	Delay before the event occurs. Maximum value is one day (86,400,000 ms).
long eventmask	Event to post when the timer expires. Normally, this is EVT_TIMER, 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.

**Notes**

By default, 30 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 (30) 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.

## **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.
	1 = CTS
	2 = DCD

**Return Values**

Success:	0
Failure:	-1
<ul style="list-style-type: none"><li>• If invalid pointer, <code>errno</code> is set to <code>EACCES</code>.</li><li>• If executed on a COM port that does not support it, <code>errno</code> is set to <code>EINVAL</code>.</li><li>• If open block is not set, <code>errno</code> is set to <code>EINVAL</code>.</li></ul>	



This function works only for Vx610 COM2. If executed on Vx510, the function will return `SUCCESSFUL` but no event is generated as Verix V do not have the required handshake lines.

---

## 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 msecs);
```

### Parameters

msecs	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 <a href="#">set_timer()</a> .) -EINVAL: The caller specified a value greater than 65535.
----------	--

### Notes

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 17 lists event bits).

**Notes** 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

`needed_events`      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.

## Case Removal Latch on Vx700 PIN Pad

The Vx700 PIN pad case has a removal switch that triggers the EVT\_REMOVED OS event, delivered to the application. The OS is notified that the keypad unit has been removed from its host system by monitoring the case removal switch.

The application initially uses set\_activation\_hash to store the value in the OS. Since the old hash value must be given to set it, a default hash value is provided by the OS. It is good practice to change it during application startup.

---

### NOTE



The default hash value is the same as the OS default password.

---

During normal operation the application reads the EVT\_REMOVED event. If set, it stops the normal processing and awaits input of an activation password/hash. It compares the input to that stored in the OS and resets the latch using reset\_latch(). The OS toggles to case latch reset (nCASE\_OPEN\_LATCH\_RESET). If successful, normal operations resume. The application retrieves the latch status at any time using get\_latch\_status().

## **get\_latch\_status()**

Retrieves the latch status.

**Prototype**    int get\_latch\_status(void);

### **Return Values**

- |   |                             |
|---|-----------------------------|
| 1 | If latched.                 |
| 0 | If open or unit is removed. |

## **reset\_latch()**

Resets the latch.

**Prototype**    `int reset_latch(char *hash);`

### **Return Values**

- 1     If hash is a match and latch is reset.
- 0     If hash is not a match.
- 1    If latch reset is unsuccessful.

## **set\_activation\_hash()**

---

Sets/stores Activation Code hash.

**Prototype**    int set\_activation\_hash(char \*oldHash, char \*hash);

### **Return Values**

- 1     Success.
- 1    If \*hash either contains characters not on the keypad or it does not meet the length requirement of five to ten characters.
- 0     If otherwise.



## CHAPTER 7

# Console Device

The console controls the display and keypad.

## Display

The LCD display size of V<sup>x</sup>5xx/V<sup>x</sup>610 terminals is 21 x 8 (21 columns by eight lines) for 168 characters in the default font, which is an ASCII font built in to the system; V<sup>x</sup>670 has 21 x 16 display size for 336 characters. Other fonts can be downloaded as files.

The display is configured based on the font. For example, on an 8-line display with an 8 x 16 font, the display is 4 lines by 16 characters; for a 16 x 16 font, the display is 4 lines by 8 characters.

Graphic images can be displayed by creating custom font files or through direct pixel writes using [putpixelcol\(\)](#).

When writing to the display, the default behavior is for the display to wrap output within the current window and truncate anything outside of the window. The APIs that modify this behavior are [put\\_graphic\(\)](#) and [setscrollmode\(\)](#).

When selecting a position for text or graphics, invalid coordinates are forced to valid values based on the current font size and window. No error returns.

The V<sup>x</sup>670 terminal features inverse video settings. These can be defined using the functions [getinverse\(\)](#), [inverse\\_toggle\(\)](#), and [setinverse\(\)](#).

## Default Font

There is one built-in font (the default font) that is in effect until an application sets another font. This font is the 6 x 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 system mode).
- Characters 0x16–0x17 combine to form a down arrow (similar to the arrow used in System Mode).
- 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  $8 \times 10$  is *not* supported.

**Table 18** lists the standard Verix V-based font files supported by the console driver, and their data and file sizes.

**Table 18      Verix V Font Files Supported**

Font Character Size (pixel col x row)	Characters per Screen (row x col)		Bytes per Character
	8-Line LCD	16-Line LCD	
6 x 8	8 x 21	16 x 21	6
8 x 16	4 x 16	8 x 16	16
16 x 16	4 x 8	8 x 8	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.

## Backward Compatibility Mode

On Vx670, Backward Compatibility Mode (BCM) provides a method for existing applications to use the larger display without modification. All current API functions supported for applications by the OS will be supported in the BCM. Applications that use the 8-line mode will be displayed on lines 5-12. The user can choose to display vertical lines that represent arrow extensions in line 14-16. This helps the user identify where the arrows displayed on line 12 are pointing.

The user can designate BCM for an application without updating the code by setting \*BCM in CONFIG.SYS. \*BCM can be set in each GID. There are two settings. If set to one (1), the application is displayed in lines 5-12 without the four arrow extensions shown at the bottom of the screen. If set to two (2), the arrow extensions will be displayed for all application screens.

If the user needs a more fine grained setting, the application code can be modified to include a call to the following functions:

## **set\_bcm()**

Setting the BCM using this call will override the CONFIG.SYS\*BCM setting.

**Prototype**    int set\_bcm(int mode);

**Return Values**

set_bcm(0)	Set BCM off.
set_bcm(1)	Set BCM on, no arrow extensions.
set_bcm(2)	Set BCM on, with arrow extensions.

## Keypad

The keypad contains a 12-key Telco-style keypad, four function keys, the ALPHA, Enter, Backspace, and Cancel keys, and four screen-addressable keys (in this documentation they are referred to as PF1–PF4, with PF1 being the leftmost key and PF4 rightmost).

Figure 8 shows the typical V<sup>x</sup>5xx/V<sup>x</sup>610 keyboard layout with corresponding labels.

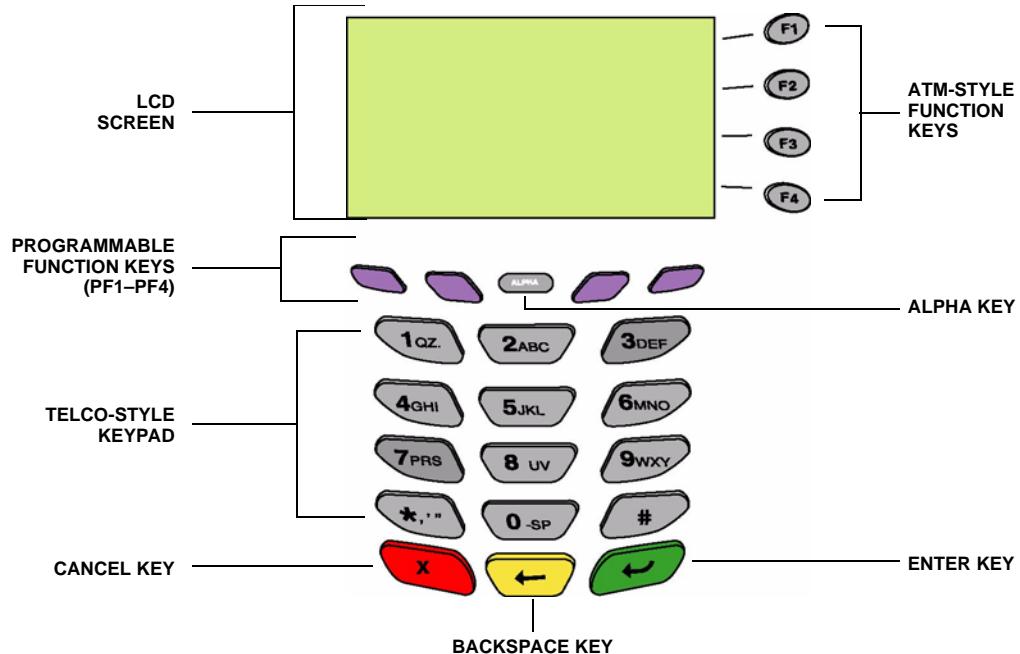


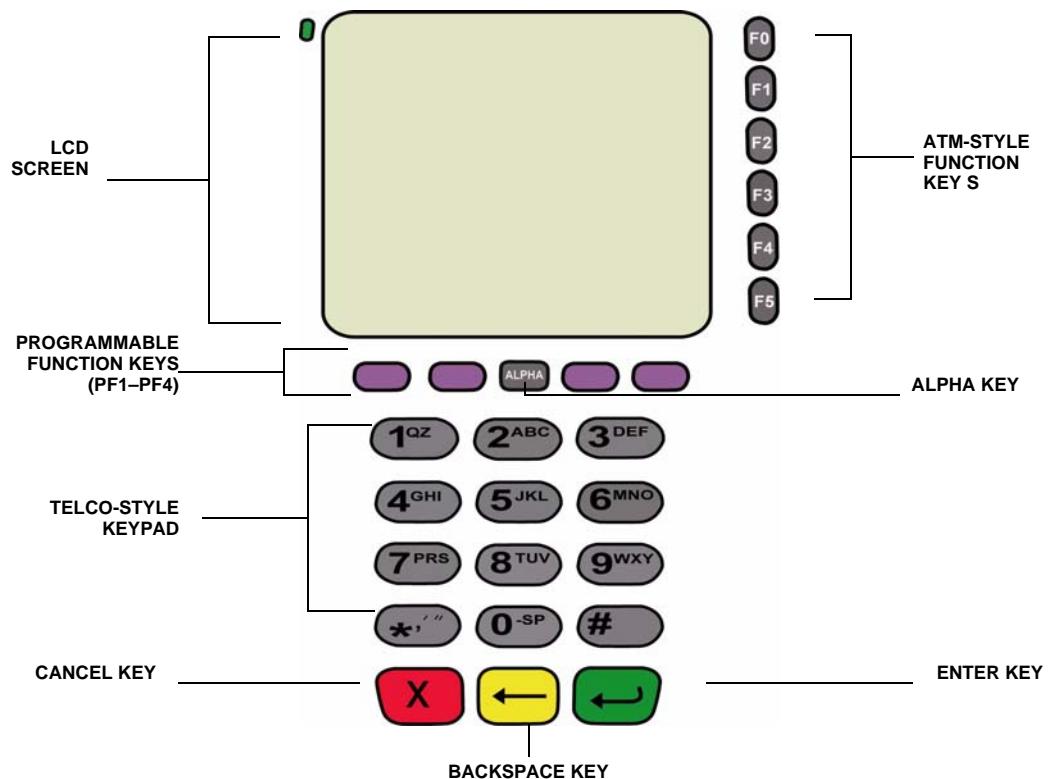
Figure 8 V<sup>x</sup>5xx/V<sup>x</sup>6xx Terminal Keypad

**NOTE**



The V<sup>x</sup>810 PIN pad also supports the above keypad layout.

Figure 9 shows the V<sup>x</sup>670 terminal having two additional ATM-style function keys, F0 and F5.



**Figure 9      V<sup>x</sup>670 Terminal Keypad**

The application controls the function of the screen-addressable keys. The following are system mode 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 system mode, 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.

**Vx700 Keypad**

On the Vx700 PIN pad, two changes were made in the standard Verix V (Telco) keypad. First, the star key ('\*') on the lower left of the keypad is replaced by a DOWN arrow. This key returns key scan code 0x6C. The pound symbol key ('#') on the lower right of the keypad is replaced by an UP arrow. This key returns key scan code 0x6D. The character set key mapping changes to a variant of the EBS 100.

**Table 19      Keypad Changes**

New Key	Scan Code
DOWN arrow	0x6D
UP arrow	0x6C

Second, the V5 PIN pad keypad does not support the eight screen keys traditionally seen on Verix V terminals. For more information, see the [Character Mapping](#) table.

**Keypad Configuration**

Eight screen keys traditionally used in Verix V user interface and Verix V applications are removed — the four ATM-style function keys arranged on the right side of the display (F1 – F4) and the four programmable function keys arranged across the bottom of the display (PF1 – PF4).

The System Mode screens are updated to use numeric keys 1 – 4 in place of the ATM-style F1 – F4 function keys. The DOWN/UP arrow keys now replace the STAR and POUND keys used in place of the two leftmost programmable functions keys (PF1 and PF2).

**NOTE**

The left and right arrows used in EDIT mode are not available to the user. Screens that offer navigation choices and request input (keys 1-4) are changed (split) to two screens.

Figure 10 displays the type 4 keypad with up and down arrows.



**Figure 10      Vx700 Keypad: User-Defined Keys Removed**

### Keypress Scan Codes

Table 20 lists the keypress scan codes.

**Table 20      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.

**Table 20      Keypress Scan Codes** (continued)

Keypress	Scan Code	Description
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 <sup>a</sup>	0xEE	n with the high-order bit set.
F1	0xFA	z with the high-order bit set.
F2	0xFB	{ with the high-order bit set.
F3	0xFC	with the high-order bit set.
F4	0xFD	} with the high-order bit set.
F5 <sup>a</sup>	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.

a. Applicable for 16 x 21 LCD (VX670).

## Alpha Key Support

Normal system usage, as well as some system mode operations, requires entering alphanumeric information. The ALPHA key is provided to support alphanumeric entries. This section describes system mode ALPHA key functionality. The application must implement ALPHA key functionality. [alpha\\_shift\(\)](#) helps applications implement ALPHA key support.

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 21 illustrates how to enter the string: 2 A E S ! space.

**Table 21 Alphanumeric Key Entry Example**

Desired Character	Keypress	Notes
2	Press 	
A	Press 	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  once	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	
!	Press 	
	Press  once	
space	Press 	
	Press  twice	

Table 22 shows different characters and how to access them from the keypad:

**Table 22 Alphanumeric Characters and Shift Entries**

Key Press	No Key Press	Press ALPHA One Time	Press ALPHA Two Times	Press ALPHA Three Times	Press ALPHA Four Times
	1	Q	Z	.	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	R	S	7
	8	T	U	V	8
	9	W	X	Y	9
	0	-	space	+	0
	*	,	,	"	*
	#	!	:	;	@

The # key also supports the additional characters: =, &, /, \, %, \$, and [space]. To enter =, press # once and then press ALPHA **five** times. To enter &, press # once and then press ALPHA **six** times. To enter /, press # once and then press ALPHA **seven** times.

To enter \, press # once and then press ALPHA **eight** times. To enter %, press # once and then press ALPHA **nine** times. To enter \$, press # once, then press ALPHA **ten** times. To enter [space], press # once, then press ALPHA **eleven** times.

### Vx700 Alpha Character Support

The blue HELP key on the Vx700 unit is also used as the ALPHA key. The alpha character set is distributed among the keys.

The Vx700 PIN pad does not support the cell phone mode (multi-key press mode). A new Verix V keypad type, based on EBS 100, is added. It returns key scan codes for the UP and DOWN arrows that are not typically used by the Verix V keypad. The DOWN arrow returns the key code 0x6C and the UP arrow returns the key code 0x6D. The high order bit is set when returned to the user. The manufacturing block (MIB) contains the new keypad type 4.

The alpha character set is not printed on the keys (Table 23 shows the key/character mapping). The characters that are currently available on the PLUS and MINUS keys on EBS 100 keypads are distributed to other keys. The \* and # on the list is mapped to the one (1) key. The remaining characters are available on the zero (0) key. The high bit is set on all key scan codes returned to the user.

**Table 23 Character Mapping**

Key	Key Scan Code	Character Set
1	0xB1	* #
2	0xB2	A B C
3	0xB3	D E F
4	0xB4	G H I
5	0xB5	J K L
6	0xB6	M N O
7	0xB7	P Q R S
8	0xB8	T U V
9	0xB9	W X Y Z
0	0xB0	- , ' " . <space character> + ! : ; @ = & / \ % \$ _
DOWN Arrow	0xEC (new code)	
UP Arrow	0xED (new code)	
CANCEL (red)	0x9B	
BKSP (yellow)	0x88	
CLEAR (yellow)	0x8E	
ENTER (green)	0x8D	
HELP (blue)	0x8F	Also the ALPHA key and key scan code

Configuring the Vx700 PIN pad from the keypad is not commonly done. For the most part, defining variables/parameters in the CONFIG.SYS file is done over the network either from VeriCentre, other servers, or over COM1. If a user is required to input variables via the keypad, it is recommended that a key mapping or overlay be used to show the character mapping set.

**NOTE**



The standard password is keyed-in similar to that of the EBS 100 keypad. This is done by using the following key sequence: 9 ALPHA ALPHA ALPHA ALPHA 66831. The F2/F4 keys are unavailable. Press ENTER+7 key sequence to enter System Mode.

## CELL\_PHONE Mode on Vx810

Apart from the classic method where the user enters alpha characters the same way it is done on all current Verix V based terminals, the Vx810 PIN pad also supports the CELL\_PHONE mode.

CELL\_PHONE Mode allows the user to select cell phone style (multiple key press mode) alpha character entry. When the Vx810 operates in cell phone mode, the alpha characters are obtained by pressing specific keys repeatedly until the desired character is displayed (i.e, 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 V 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 the Vx810 operates in cell phone alpha mode, the OS translates the repeated key presses that take place within 1.5 seconds of each other (default) to the next alpha character in the alpha shift sequence for that key. If a key is pressed more than 1.5 seconds after the last, the OS returns the key as input.

**NOTE**

Pressing the “2” key twice in a row with a 1.5 second (or more) pause 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 1500 or 1.5 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).

**NOTE**

900A00 results in the value 900. 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 System Mode 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 <code>alpha_multi_shift()</code> .
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:

- One of the pair of keys must be from column three of the physical keypad shown in [Figure 8](#) (the control characters are: d, Cancel, Backspace, Enter), otherwise the first key scanned of the pair returns as a single key.
- The second key must be a numeric key (that is, 0–9). Scan codes for control characters and any other keys are undefined.

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 24](#).

**Table 24      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



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 system mode. These are the only keypresses that do not follow the restrictions defined above.

## Hidden Function Support

There are System Mode 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 System Mode

To enter system mode, simultaneously press the F2 and F4 keys. The dual keypress Enter+7 also enters system mode. The console driver detects the keys for system mode entry when they are pressed simultaneously and debounced for the normal duration.

When this key sequence is detected and debounced, the console driver deactivates the current task (if it is not already the system mode task) and activates the system mode task.

The console driver does not save and restore the display on entry and exit of system mode. It is the application's responsibility to restore the display (if desired).

## 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 100 ms while the key is being held.

## Console Ownership

Verix V 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 four ways to transfer ownership of the console from one task to another:

- Press the hot key (pair)
- Press the system mode 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 key FIFO 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.



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.

## Management 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 25](#). This information can be retrieved by an application (usually the terminal management agent) using the [get\\_component\\_vars\(\)](#) function.

**Table 25 Management 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	

## Console Function Calls

This section presents descriptions of the function calls used to control the console.

## activate\_task()

Allows the current console owner to pass control of the console to the specified task. Since this is a device call, 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.

**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 system mode 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 21](#)). 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 22](#).

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 <a href="#">Table 22</a> .   |
| <code>alpha_shift(A)</code> | Returns “B”, which is the character that follows “A” in the ALPHA key shift sequence in <a href="#">Table 22</a> .   |
| <code>alpha_shift(B)</code> | Returns “C”, which is the character that follows “B” in the ALPHA key shift sequence in <a href="#">Table 22</a> .   |
| <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 <a href="#">Table 22</a> . |

## **close()**

Releases ownership of the console device.

**Prototype**    `int close(int handle);`

**Parameters**    `handle`    Handle of the console device

**Return Values**

Success:    0

Failure:    -1 and errno set to EBADF: Caller does not own console device.

## **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);`

**Parameters**    None

**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);`

**Parameters**    None

### **Return Values**

Success:    0

Failure:    -1 and errno set to EBADF: Caller does not own console device.

## 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).

**Prototype**      `int contrast_down(void);`

**Parameters**      None

**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)`.

**Prototype** `int contrast_up(void);`

**Parameters** None

**Return Values**

Success: 0

Failure: -1 and `errno` set to `EBADF`: Calling task does not own the console.

## **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);`

**Parameters**    None

**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.

**Prototype**    `int disable_hot_key(void);`

**Parameters**    None

**Return Values**

Success:    0

Failure:    -1 and errno set to EBADF: Caller does not own console device.

## **disable\_key\_beeps()**

---

Disables audible key beep on key press.

**Prototype**    void disable\_key\_beeps(void);

**Parameters**    None

**Return Values**    None

## **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);

**Parameters** None

**Return Values** None

## **enable\_key\_beeps()**

---

Enables audible key beeps on a keypress.

**Prototype**    void enable\_key\_beeps(void);

**Parameters**    None

**Return Values**

Success:    0

Failure:    -1 and errno set to EBADF: Caller does not own console device.

## **getcontrast()**

Returns the current contrast setting for the display. See also, [setcontrast\(\)](#).

**Prototype**    `int getcontrast(void);`

**Parameters**    None

***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.

**Return Values**    None

## **getgrid()**

Returns the current grid setting. The font file character size determines the grid.

**Prototype**    `int getgrid(void);`

**Parameters**    None

**Return Values**

Success Code:    Font size

2                 $6 \times 8$

0                 $8 \times 16$  and  $16 \times 16$

Failure:        -1 and errno set to `EBADF`: Caller does not own console device.

## **getinverse()**

---

Returns the current inverse video setting.

**Prototype**    `int getinverse(void);`

**Parameters**    None

**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> . |

## 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);`

**Parameters** None

**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 EBADF: Caller does not own console device.  |

## **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:    $\geq 0$ : Console handle

Failure:      -1 and errno set to `EBADF`: Caller does not own console device.

## 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);`

**Parameters**      None

### ***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\_hot\_key\_sts()

Returns the current status of the hot key status. 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**

Similar to the Verix 68K terminals, 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. For screen keys F1–F4, the secured key codes are 0x01–0x04, respectively.

See also [set\\_hot\\_key\(\)](#), [disable\\_hot\\_key\(\)](#), [enable\\_hot\\_key\(\)](#), and [Table 20](#).

**Prototype**

```
long get_hot_key_sts(void);
```

**Parameters**

None

**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.

## **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.

## insline()

Inserts a blank line following the line containing the cursor. All lines following it move down one line. See also [delline\(\)](#).

**Prototype**    `int insline(void);`

**Parameters**    None

**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);`

**Parameters**    None

### **Return Values**

0       Success.

-1       If calling task does not own the console, `errno` is set to `EBADF`.

## **kbd\_pending\_count()**

Returns the number of keystrokes available for reading. The maximum number of keystrokes is 20.

**Prototype**    `int kbd_pending_count(void);`

**Parameters**    None

**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.

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.

## open()

Returns the handle for writing (or reading or closing) the console device.

### NOTE



This function *must* be called *before* any other console functions can be called.

The following operations are performed on calls to the `open()` function:

- Set the font to the default
- Clear the display
- Clear the key buffer
- Turn off the cursor
- Set 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

### Prototype

```
int open(char *device_name, int unused);
```

### Parameters

`device_name` The console device name. Use `DEV_CONSOLE` in `svc.h`.

### Notes

The application that opens the console may receive the following events:

- `EVT_KBD` when a key is available to be read
- `EVT_DEACTIVATE` when this task loses ownership of the console

### Return Values

Success:  $\geq 0$ : The console handle.

Failure:  $-1$  with `errno` set to `ENODEV`: Invalid `device_name` parameter.

## 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 `buffer` in columns of 8 pixels horizontally. If the current character size is 8x16 or 16x16, 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 *buffer, int len);
```

### Parameters

<code>buffer</code>	Pointer to data to write.
<code>len</code>	Number of bytes in <code>buffer</code> to write.

### Return Values

Success:	>0: Number of characters written to display. This should be the same as <code>len</code> .
Failure:	-1 with <code>errno</code> is set to <code>EACCES</code> : <code>buffer</code> is an invalid pointer. -1 with <code>errno</code> is set to <code>EINVAL</code> : Negative length parameters. -1 and <code>errno</code> set to <code>EBADF</code> : Caller does not own the console device.

## **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 `length` regardless of number of characters written to the display (for example, if `buf` is truncated, `length` is still returned).

**Failure:** -1 with `errno` set to `EACCES`: Invalid `buf` pointer.  
-1 with `errno` set to `EINVAL`: `length` is negative.  
-1 and `errno` set to `EBADF`: Caller does not own the 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 system mode entry keys and the hot key. The size of the key buffer is 20 characters.

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

**Parameters**

handle	Console handle
buffer	Pointer to store read keys
length	Maximum number of keys to store in the buffer

**Return Values**

>0	Number of keys stored in the buffer. Maximum is length.
-1	If <i>handle</i> is invalid, <i>errno</i> is set to <code>EBADE</code> . For invalid <i>buffer</i> pointer, <i>errno</i> is set to <code>EACCESS</code> . For negative <i>length</i> parameters, <i>errno</i> is set to <code>EINVAL</code> .

## 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**

<code>font</code>	Font definition file used for the font. To select the default font, use a null string.
<code>grid_id</code>	Not used.

**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> : <code>font</code> is an invalid pointer. Other <code>errno</code> s may be set by the file manager.

## **screen\_size()**

Retrieves the screen size.

**Prototype**    `int screen_size(char *buffer);`

**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.

## **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.

### **Prototype**

```
int setcontrast(int value);
```

### **Parameters**

`value`      Contrast setting:

- 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 `errno` set to `EBADF`: 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 & 3	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 backlight on/off. The state of the backlight is preserved in the MIB data area for the console.

**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\_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\_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\\_hot\\_key\(\)](#)

Defines which key is the hot key and who owns it. The calling task receives control when this key is pressed. The currently active task is notified of its imminent loss of control through an `EVT_DEACTIVATE` event, and the owning task receives an `EVT_ACTIVATE` event. The pressed key does not appear in the key buffer and is not returned to the application.

### **Prototype**

```
int set_hot_key(int keycode);
```

### **Parameters**

`keycode` Keyboard scan code to assign to hot key. See [Table 20](#) and [Table 24](#) for keyboard scan codes. No checks are made to ensure that `keycode` is a valid keycode scan code.

### **Return Values**

Success: 0

Failure: -1 with `errno` set to `EBADF`: Caller does not own the console device.

---

### **NOTE**



There can only be one hot key owner for each system restart/power cycle.

---

## 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**

`stuff_6x`    Pointer to the buffer.

**Return Values**

0x30 0x30 0x30 0x30 0x30 0x30	= 1 line (7 segment)
0x30 0x30 0x30 0x30 0x30 0x31	= 2 line (7 segment)
0xFF	no data, since manufacturing block has not been loaded.

## SVC\_INFO\_KBD()

Fills the caller's buffer with the one-byte keyboard type from the manufacturing block.

**Prototype** int SVC\_INFO\_KBD(char \*stuff\_1x);

**Parameters**

stuff\_1x Pointer to the 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#)
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

- x      Location to return the horizontal (or column) coordinates of the cursor.
- y      Location to return the vertical (or row) coordinates of the cursor.

### Return Values

Success: 0

Failure: -1 with errno set to EBADF: Caller does not own console.  
-1 with errno set to EACCES: Either x or y 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

- |                 |   |
|-----------------|---|
| <code>x1</code> | Pointer to location to store window top-left x (horizontal or column) coordinate.     |
| <code>y1</code> | Pointer to location to store window top-left y (vertical or row) coordinate.          |
| <code>x2</code> | Pointer to location to store window bottom-right x (horizontal or column) coordinate. |
| <code>y2</code> | 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()

Writes buffer to display. The characters stored in `buffer` 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.

### Prototype

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

### Parameters

`handle` Console handle  
`buffer` Pointer to data to write  
`length` Number of bytes in `buffer` to write

### Return Values

Success: Value will be `length` regardless of number of characters written to the display (that is, if truncated, `length` is still returned).  
Failure: -1 with `errno` set to `EBADF`: handle is invalid.  
-1 with `errno` set to `EACCES`: Invalid `buffer` pointer.  
-1 with `errno` set to `EINVAL`: A negative `length` parameter was passed.

## **write\_at()**

Similar to [write\(\)](#), 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.

---

## CONSOLE DEVICE

*write\_at()*



## CHAPTER 8

### Service Function Calls

The function calls listed in this chapter retrieve information about the Verix V-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\\_VERSION\\_INFO\(\)](#).

For drivers associated with a device, `handle` 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 `buffer`, with length `len`. The format is as follows:

**Table 26**

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– <code>len</code>	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.

**Prototype**    `int get_component_vars (int handle, char *buffer, int len);`

**Return Values**

Success: Information about OS component

Failure: –1 and `errno` set to `EACCESS`: Invalid buffer pointer.

–1 and `errno` set to `EINVAL`: Invalid component number or `len` is less than 18.

**Example**    The linked example program displays the file name and build time of each OS component.



On Vx810 DUET, this returns a file name of “com4\_usb.bin” for the USB thermal printer device — the Micro-controller firmware.

## **set\_combo\_mode()**

Sets the module specified by mode either in conventional telephone modem or as a TCP/IP adapter.

**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 [SVC\\_CS2AZ\(\)](#) for a description of counted strings.

**Prototype**

```
int SVC_CHK_PASSWORD (const char *buffer);
```

**Return Values**

- 1: Contents of buffer match password.
- 0: Contents of buffer do not match password.
- EACCES: Caller's buffer is not readable.

## SVC\_FLASH\_SIZE()

Returns the amount of installed flash memory in kilobytes. See SVC\_RAM\_SIZE() for examples.

**Prototype**    `int SVC_FLASH_SIZE (void);`

## 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\_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 V traps it and saves some diagnostic information in a *crash* log (also known as an *error* log). The information can be viewed through a system mode 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, yyymmddhhmmss */  
};
```

## 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**  
stuff\_6x    Pointer to the buffer.

**Return Values**

0x30 0x30 0x30 0x30 0x30 0x30	= 1 line (7 segment)
0x30 0x30 0x30 0x30 0x30 0x31	= 2 line (7 segment)
0xFF	no data, since manufacturing block has not been loaded.

## 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\\_VERSION\\_INFO\(\)](#).

Verix V version numbers are in the form *QAhhvvmm*, where:

- QA Designates the Verix V OS
- hh Hardware definition, currently 00
- vv Released version, beginning with 00
- mm Minor release, usually A0

**Prototype** void SVC\_INFO\_EPROM (char \*buf\_9);

**Example** See the linked example in [SVC\\_VERSION\\_INFO\(\)](#).

## **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\_KBD()

Fills the caller's buffer with the one-byte keyboard type from the manufacturing block.

**Prototype**    `int SVC_INFO_KBD(char *stuff_1x);`

**Parameters**  
`stuff_1x`    Pointer to the 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#)
- 0xFF No data; manufacturing block not loaded.

## **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 V 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\_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\_MAG()

Stores a 1-byte magnetic card reader type code in the caller's buffer. The types are defined as follows:

- 0: No card reader present.
- 1: Dual-track, tracks 1 and 2.
- 2: Dual-track, tracks 2 and 3.
- 3: Triple track.
- 4: Japan format (triple track, tracks 1 and 2, plus JIS Type II front track).
- 5: Triple track, Vx670.
- 6: Triple track, MagnePrint.

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 27](#) 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 27 Manufacturing Block Contents**

Field	Size	Related Function
Manufacturing Data	30	<a href="#">SVC_INFO_MFG_BLK()</a>
Model Number	12	<a href="#">SVC_INFO_MODELNO()</a>
Country	12	<a href="#">SVC_INFO_COUNTRY()</a>
Part Number	12	<a href="#">SVC_INFO_PARTNO()</a>
Hardware version	2	<a href="#">SVC_INFO_HW_VERS()</a>
Lot Number	5	<a href="#">SVC_INFO_LOTNO()</a>
Serial Number	11	<a href="#">SVC_INFO_SERLNO()</a>
Permanent Terminal ID	8	<a href="#">SVC_INFO_PTID()</a>
Keypad Type	1	<a href="#">SVC_INFO_KBD()</a>
Mag Card Reader Type	1	<a href="#">SVC_INFO_MAG()</a>
Display Type	6	<a href="#">SVC_INFO_DISPLAY()</a>
Printer Type	1	<a href="#">SVC_INFO_PRNTR()</a>
PIN Pad Type	1	<a href="#">SVC_INFO_PIN_PAD()</a>

**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\_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.



On Vx670 and Vx700, this always returns 50. This is also true for Vx810 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. This is supported in V<sup>x</sup>510, V<sup>x</sup>610, V<sup>x</sup>670, and V<sup>x</sup>810 units.

---

**NOTE**

The V<sup>x</sup>610 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 V<sup>x</sup>810, this always returns a value 50. This is true whether the V<sup>x</sup>810 PIN pad is operating as a stand-alone device or is connected to a DUET. This is similar to how the V<sup>x</sup>670 terminal operates.

---

**Prototype**

```
int SVC_INFO_MODULE_ID(int port);
```

**Return Values**

Success: Module ID of COM port.

- 2 - unknown, any modem other than from the list below
- 3 - TDK Modem
- 4 - Conexant Banshee modem
- 5 - Carlos Modem
- 6 - Connect One Ethernet 10BaseT only
- 7 - Connect One Ethernet 10BaseT and Carlos combo
- 8 - Siemens GSM/GPRS US only
- 9 - Siemens GSM/GPRS International only
- 10 - Sierra CDMA 1xRTT only
- 11 - Connect one WiFi 802.11b only
- 12 - Siemens GSM/GPRS US and Carlos combo
- 13 - Siemens GSM/GPRS International and Carlos combo
- 14 - Sierra CDMA 1xRTT and Carlos combo
- 15 - Connect One 802.11b WiFi and Carlos combo
- 16 - Predator Conexant Eisenhower modem
- 17 - Predator Conexant Eisenhower modem and USB Ethernet
- 18 - Predator Conexant Eisenhower modem and Sierra CDMA 1xRTT
- 19 - Predator Conexant Eisenhower modem and Siemens GSM/GPRS US and Carlos combo
- 21 - Predator Conexant Eisenhower modem and USB WiFi
- 22 - Conexant Banshee/CO210 combo

- 23 - MID\_CO210\_ONLY.
- 24 - MID\_ISDN\_ONLY

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.

## 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\_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\_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 Vx810 DUET, these values are true even if the Vx810 is operating as stand-alone device or connected to the DUET.

## 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\_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\\_SERLNO\(\)](#).

**Prototype**    `void SVC_INFO_PTID (char *buf);`

## 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

## 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\\_PTID\(\)](#).

### Prototype

```
int SVC_INFO_SERLNO (char *buf_11);
```

### Return Values

Success: 0

Failure: -1 with errno set to EACCES: Invalid buffer pointer provided.

## SVC\_SHUTDOWN()

Commands the V<sup>x</sup>610/V<sup>x</sup>670 terminals 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 a V<sup>x</sup>610/V<sup>x</sup>670.

## 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 V-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 Vx810 PIN pad does not support a printer and a battery. No LEDs are supported to show their status.

## SVC\_RAM\_SIZE()

---

Returns the amount of installed RAM in kilobytes. See also [SVC\\_FLASH\\_SIZE\(\)](#) and [dir\\_get\\_sizes\(\)](#).

**Prototype**    int SVC\_RAM\_SIZE (void);

**Example**

## SVC\_VERSION\_INFO()

Stores a counted string that contains the OS version information, which supplements the version number returned by [SVC\\_INFO\\_EPROM\(\)](#). 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\\_EPROM\(\)](#) 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\_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);`

## 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 V linkable library contains a number of routines to support FIFOs. This section describes the routines available in the library. In Verix V, 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.

---

## **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**

```
void SVC_CLR_FIFO (fifo_t *fifo, int datasize);
```

## 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 `buffer`. 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 *buffer, 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 *buffer, int size);`

**Return Values** Returns the number of bytes added.

**Example**

## CRCs

CRCs (cyclic redundancy checks) 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 28](#) summarizes the CRC algorithms supported by Verix. 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 28** CRC Algorithms Supported by Verix

Function	Algorithm	Size	Bit Order	Seed	Result	Default Seed	Type
<a href="#">SVC_CRC_CALC()</a>	LRC	8	—	—	—	0	0
<a href="#">SVC_CRC_CRC16_L()</a>	CRC16	16	LSB first	1,2	2,1	0	1
<a href="#">SVC_CRC_CRC16_M()</a>	CRC16	16	MSB first	2,1	2,1	0	2
<a href="#">SVC_CRC_CCITT_L()</a>	CCITT	16	LSB first	1,2	2,1	-1	3
<a href="#">SVC_CRC_CCITT_M()</a>	CCITT	16	MSB first	2,1	2,1	-1	4
<a href="#">SVC_CRC_CRC32_L()</a>	CCITT	32	LSB first	4,3,2,1	4,3,2,1	-1	5

**CRC Function Calls** This section presents the CRC function calls.

## **SVC\_CRC\_CALC()**

Calculates a CRC value for `size` bytes of data in `buffer`. 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 28](#) lists the specific function for each type.

See also [SVC\\_LRC\\_CALC\(\)](#), [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 *buffer, 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 *buffer, int size);`

## SVC\_CRC\_CCITT\_L()

Calculates a 16-bit CRC for `size` bytes of data in `buffer` using the CCITT polynomial:

$$x^{16} + x^{12} + x^5 + 1.$$

`SVC_CRC_CALC(3, buffer, size)` is equivalent to  
`SVC_CRC_CCITT16_L(buffer, size, -1)`

**Prototype**

```
unsigned int SVC_CRC_CCITT_L (const void *buffer, int sz, unsigned int seed);
```

## SVC\_CRC\_CCITT\_M()

Calculates a 16-bit CRC for `size` bytes of data in `buffer` using the CCITT polynomial

$$x^{16} + x^{12} + x^5 + 1$$

`SVC_CRC_CALC(4, buffer, size)` is equivalent to  
`SVC_CRC_CCITT_M(buffer, size, -1)`

**Prototype**

```
unsigned int SVC_CRC_CCITT_M (const void *buffer, int sz,  
                             unsigned int seed);
```

## SVC\_CRC\_CRC16\_L()

Calculates a standard CRC16 CRC value for `size` bytes of data in `buffer`.  
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 reverse the byte order when appending the CRC to a message (see example below).

`SVC_CRC_CALC(1, buffer, size)` is equivalent to  
`SVC_CRC_CRC16_L(buffer, size, 0)`.

See also [SVC\\_CRC\\_CRC16\\_M\(\)](#) and [SVC\\_CRC\\_CALC\(\)](#).

### Prototype

```
unsigned int SVC_CRC_CRC16_L (const void *buffer, 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 `buffer`. 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, buffer, size)` is equivalent to  
`SVC_CRC_CRC16_M(buffer, size, 0)`

### Prototype

```
unsigned int SVC_CRC_CRC16_M (const void *buffer, int sz,  
                             unsigned int seed);
```

## SVC\_CRC\_CRC32\_L()

Calculates a 32-bit CRC32 CRC value for `size` bytes of data in `buffer`. 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(buffer, size, -1) is equivalent to  
SVC\_CRC\_CALC\_L(5, buffer, size)

### Example

## SVC\_LRC\_CALC()

Calculates the LRC (longitudinal redundancy check) value for `size` bytes of data in `buffer`. 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 *buffer, 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 `buffer`, 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 *buffer, long size);`

## SVC\_MOD\_CK()

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 29](#) considers the 16-digit account number 4168641234567890:

**Table 29      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_CK (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.



## CHAPTER 9

### System Devices

This chapter describes the API (application programming interface) for the following system devices:

- Magnetic Card Reader
- Smart Card Reader
- Real-Time Clock/Calendar
- Beeper
- Internal Printer
- SDIO
- USB Fingerprint Reader Device
- USB Barcode Scanner
- USB Keyboard
- USB to RS-232 Converter
- Metrologic Barcode Scanner

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.

Verix V-based terminal standard device names defined in the `<svc.h>` file are listed in [Table 30](#).

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 30](#).

**Table 30**      **Verix V Device Handles**

Device	/dev Name	<svc.h> Variable
Magnetic card reader	/dev/mag	DEV_CARD
Bar code reader	/dev/bar	DEV_BAR
Real-time clock	/dev/clock	DEV_CLOCK
Beeper	/dev/stderr	DEV_BEEPER
Console (keypad and display)	/dev/console	DEV_CONSOLE
USB fingerprint reader device	/dev/bio	DEV_BIO
COM port 1	/dev/com1	DEV_COM1
COM port 2	/dev/com2	DEV_COM2
COM port 3	/dev/com3	DEV_COM3
COM port 4	/dev/com4	DEV_COM4
COM port 5	/dev/com5	DEV_COM5
COM port 6	/* com 6 */	DEV_COM6
COM port 8	/* com 8 */	DEV_COM8
COM port 9	/* com 9 */	DEV_COM9
COM port 10	/* com 10 */	DEV_COM10
Mag card	/* mag card */	DEV_CARD
Barcode reader	/* bar code reader */	DEV_BAR
MSO300 Biometric	/* MSO300 Biometric device */	DEV_BIO
CTLS	/* Contactless device */	DEV_CTL5
USB keyboard	/* USB Keyboard HID converted to make and break code */	DEV_KYBD
USB host	/* PP1000SE and Vx810 device */	DEV_USBSER
Semtek device driver	/* Semtek device driver */	DEV_SEMTEK
Customer smart card	/dev/iccl	DEV_ICC1
Merchant SAM	/dev/icc2	DEV_ICC2
Merchant SAM	/dev/icc3	DEV_ICC3
Merchant SAM	/dev/icc4	DEV_ICC4
Merchant SAM	/dev/icc5	DEV_ICC5
Merchant SAM	/dev/icc6	DEV_ICC6

## Device Management Function Calls

Each device is associated with a fixed handle ([Table 30](#)). For example the console handle is always 1. However Verix 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.

## 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 0–31) 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 (0–31).

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. See [Pipes, page 177](#).

**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**

<code>device</code>	The name used in a call to <code>open()</code> for example, <code>/DEV/COM1</code> or <code>P:MYPIPE</code> . Names are not case-sensitive.
<code>*task</code>	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 <code>set_owner()</code> . If the device or pipe is not open, <code>*task</code> 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_component_vars()`.

- 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 handle, 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.

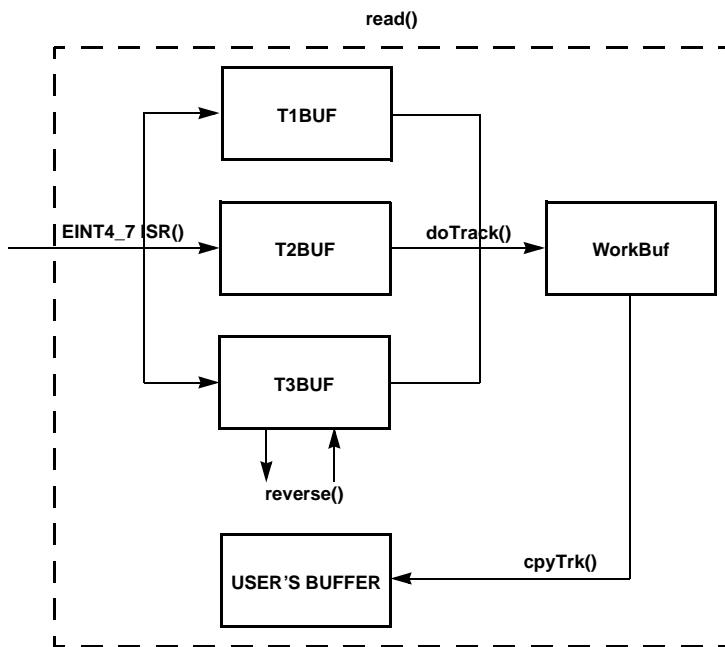
EINVAL: Invalid task number.

### **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.

## Magnetic Card Reader

The Verix V-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*. **Figure 11** illustrates buffer allocation.



**Figure 11 Software Block Diagram: Buffer Use**

The Vx810 PIN pad supports a triple track magnetic card reader, but has a variant without the Mag Card reader and circuitry. This variant is identified by a value of “0” in the Mag Card Type byte in the MIB (offset 93) and checked by the Verix V OS. If the Mag Card Type byte in the MIB is “0”, and an application calls `open()`, the Mag Card device, the OS returns -1 with `errno` set to `ENODEV`.

**NOTE**



The Mag Card device information is also directly available to the application through the `SVC_INFO_MAG()` system call.

### Hybrid Card Reader

On Vx700 PIN pad, a hybrid card reader (MX710) 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 from 92/32/92 bytes, respectively. This matches the buffer allocation in the Vx670 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.

#### **Magnetic Card Reader Function Calls**

This section presents general functions for the magnetic card driver API.

## **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.

## close()

---

Disables the card reader input, preventing the terminal from recognizing card reads.

**Prototype**    `int close(int handle);`

**Parameters**

`handle`    Value returned from `open()`.

**Return Values**

Success:    0

Failure:    -1 and `errno` set to `EBADF`: `handle` is not a valid, open file descriptor.

Any card data queued on device `close()` are discarded. Any card swipes that occur before device `open()` or after device `close()` are discarded.

## 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.

**Prototype**    int open(const char \*device\_name, 0);

**Parameters**

\*device\_name        = DEV\_CARD

**Return Values**

Success: File descriptor to access the magnetic card device.

Failure: -1 and errno set to ENODEV: device\_name is invalid.

-1 and errno set to ENOSPC: open( ) cannot obtain an OS buffer.

Reopening the magnetic card device without an intervening close( ) succeeds and does *not* clear any pending card data.

## read()

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.

**Prototype**    `int read(int handle, char *buffer, int size);`

### Parameters

`handle`    Value returned from `open()`.

`size`    Maximum number of bytes to read.

`buffer`    Pointer to the data area.

The data returned in `buffer` 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)

### Return Values

Success:    Number of bytes read or 0 if no card swipe available.

Failure:    -1 and `errno` set to `EINVAL`: `size < 6`. Six is the minimum acceptable buffer size. Depending on the application, a buffer as large as 239 bytes can be required.

              -1 and `errno` set to `EACCES`: Access violation attempting to write to `buffer`.

              -1 and `errno` set to `EBADF`: `handle` is not a valid open file descriptor.

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,  $c_1$ ,  $c_2$ , and  $c_3$  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

**NOTE**

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

**NOTE**

JIS B 9561 Type I track 1 is the same as ISO 7811 track 1.

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

ISO 7811 track 2



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.

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.

## SVC\_INFO\_MAG()

Stores a 1-byte magnetic card reader type code in the caller's buffer. The types are defined as follows:

- 0: No card reader present.
- 1: Dual-track, tracks 1 and 2.
- 2: Dual-track, tracks 2 and 3.
- 3: Triple track.
- 4: Japan format (triple track, tracks 1 and 2, plus JIS Type II front track).
- 5: Triple track, Vx670.
- 6: Triple track, MagnePrint.

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.

## 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 V for the initial phase of the VSP program. [Table 31](#) 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 31** MSR Card Data

Cleartext MSR Card Data			
Track	Count	Stat	Data
1	68	0	"%B4150000000000000^TESTCARD/VISA^_1121010000456700000012300000?K"
2	42	0	";4150000000000000=_1121011234567000000?:"
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.

$$\text{Cleartext: "4150000000000000"} \quad (\underline{0}+\underline{0}+\underline{0}+\underline{0}+\underline{0}+\underline{0}) \bmod 10 = 0$$

$$\text{Encrypted: "4150001882680000"} \quad (\underline{2}+\underline{8}+\underline{7}+\underline{2}+\underline{3}+\underline{8}) \bmod 10 = 0$$

- 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 V Implementation

The Verix V 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).

## 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\\_Disable\(\)](#)
- [VSP\\_Enable\(\)](#)
- [VSP\\_Encrypt\\_MSR\(\)](#)
- [VSP\\_Encrypt\\_KBD\(\)](#)
- [VSP\\_Decrypt\\_PAN\(\)](#)
- [VSP\\_Decrypt\\_MSR\(\)](#)
- [VSP\\_Status\\_Ex\(\)](#)

## **Data Formats**

There are five data types passed via the parameter lists of the APIs:

- `char st[8 + 9+1];`
- `char MSRc[248], char 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.

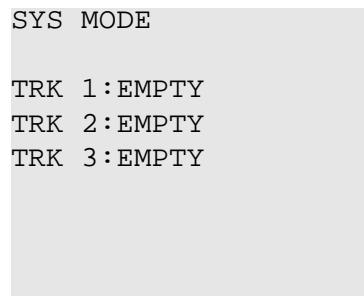
- When the MSR device driver collects data from a card swipe, the cleartext MSR image is encrypted via `VSP_Encrypt_MSR()`. Only the encrypted results are passed to the application via `read()`. Thus, the application unknowingly is dealing with encrypted MSR data.
- When the VSS or IPP code performs 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.

**System Mode Menu** The System Mode MAG CARD DIAG test screens list the decode status message for each of the three tracks. The three status messages are five characters long:

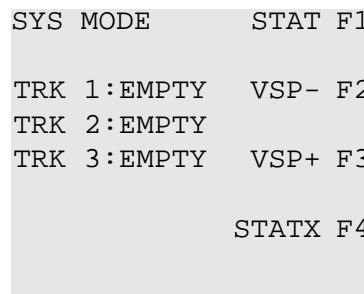
- “VALID”
- “EMPTY”
- “ERROR”

Below are sample screens from an eight-line display:



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, as shown below.



The function keys are mapped as follows:

KEY	FUNCTION
F1	int VSP_Status (char *st);
F2	int VSP_Disable (char *st);
F3	int VSP_Enable (char *st);
F4	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:

SYS MODE	STAT F1
TRK 1:EMPTY	VSP- F2
TRK 2:ERROR	
TRK 3:VALID	VSP- F3
2.2.0020	
0004 0200	STATX F4
0000 0001	

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 (V<sup>x</sup>800 and V<sup>x</sup>810) and COM1 for all others. The serial format is async, 8N1, 115.2 KBPS.

The V<sup>x</sup>700 has an eight-line display, but uses number keys in place of the function keys. The general screen appears as follows:

SYS MODE
TRK 1:EMPTY
TRK 2:EMPTY
TRK 3:EMPTY
1> STAT 2.2.0020
2> VSP- 0004 0200
3> VSP+ 0000 0001
4> STATX

For the four-line display, the general screen appears as follows:

SYS MODE	STAT F1
TRK 1:EMPTY	VSP- F2
TRK 2:EMPTY	
TRK 3:EMPTY	VSP+ F3

There is no room to display the VSP status in its entirety, thus, the text for "SYS MODE" is replaced with the status messages displayed below, before returning to "SYS MODE":

- "2.2.0020"
- "0004 0200"
- "0000 0001"

The messages switch to the next one in the sequence every few seconds, or when a key is pressed.

The general screens for 16-line displays are much like that for an 8-line display, only with more spacing between the lines. The positions of the function key labels differ between V<sup>x</sup>670 and V<sup>x</sup>810 due to the physical alignment of the function keys.

## Installation Sequence

To properly install and activate VSP inside a Verix V terminal:

- 1 Update the OS to a VSP-capable OS. For example, a V<sup>x</sup>570 will require a download of QC0011A1 or higher. Refer to the FSN or Release Notes for the proper OS version.
- 2 Enable the upper system GID by loading the Certificate Tree files and the Certificate files. Refer to the section for the VSP Driver.
- 3 Load the VSP driver by downloading the VSP driver files. Refer to the section for the VSP Driver.
- 4 In the secure room where the IPP keys are injected into the terminal, enter the System Mode MAG CARD DIAG test, and swipe command cards MASTER COMP 1 and MASTER COMP 2. The data on these cards contain the master key components (split knowledge). The VSP driver combines the key parts together and decrypts its initial data/configuration file.



### NOTE

Do this before running the System Mode IPP KEY LOAD routine because the key loading process reboots the terminal.

- 5 Download the terminal application, and connect the terminal to the host system. The set-up should be equivalent to a field installation.
- 6 With the terminal application running, swipe the ACTIVATE command card. The data on this card instructs the VSP driver to generate a new set of encryption keys, and to activate the VSP encryption code. The VSP driver returns an MSR image, which appears like normal card data. This data contains key synchronization information and must be passed to the host system, which in turn, forwards the data to the host's DA (Decryption Appliance).
- 7 At this point, VSP is fully installed and activated in the terminal. If there is any customer-specific command card issued for the terminal application, then follow the instructions that accompany the card(s). Each card may encode a command to change a VSP configuration setting from its current state, or to generate a new VSP key.

## 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.

EACCES - Argument access error.

EINVAL - Invalid argument.

EBUSY - Device is in use (cannot encrypt new data if MSR has pending input data).

## 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.

EACCES - 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.

EACCES - 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.

EACCES - 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.

EACCES - 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.

EACCES - 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.

EACCES - Argument access error.

EINVAL - Invalid argument.

EBUSY - Device is in use (cannot encrypt new data if MSR has pending input data).

## **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.

EACCES - Argument access error.

EINVAL - Invalid argument.

EBUSY - Device is in use (cannot encrypt new data if MSR has pending input data).

## 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.



Developers can write applications for supported synchronous cards. Contact your VeriFone representative for specific synch card support.

Vx810 PIN pad complies 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 System Mode 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.

## ICC Socket Locations

ICC socket locations for Vx5xx/Vx610 terminals are shown in [Figure 13](#).

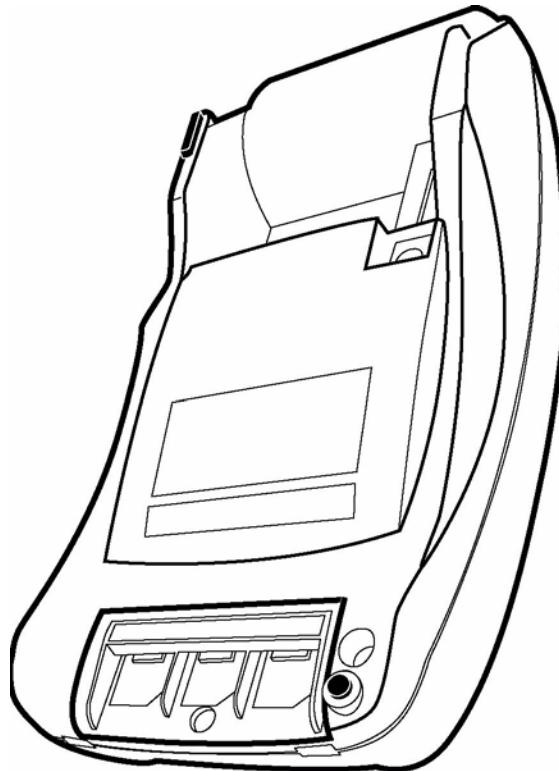
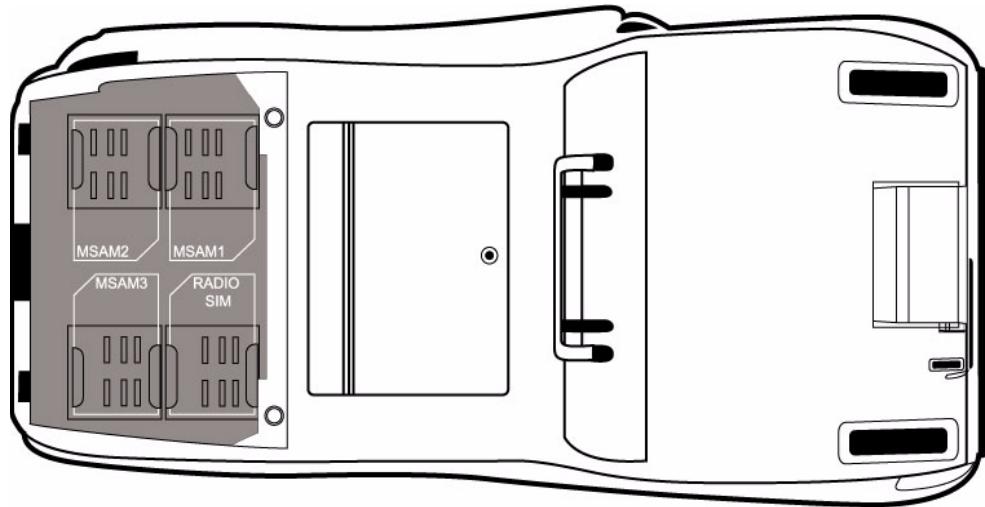


Figure 12 Vx5xx/Vx610 ICC Socket Locations

ICC socket locations for Vx670 terminal are shown in Figure 13.



**Figure 13     Vx670 Socket Locations**

## Real-Time Clock

Verix V-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.

On Vx670, the DS3610 chip requires RTC driver changes to read/update the clock register bytes. There are no registers for the 100 year bit and 1000 year bit on this chip, hence, 21st century is assumed.

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.

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.

- `read_clock()` can be used regardless of current ownership of this device. It is available to every application and can be used without opening the clock device.
- If a valid buffer is specified, the function returns -1 after copying fifteen bytes of ASCII date and time information into the caller's buffer. If the caller's buffer pointer is not valid, the function returns 0.
- If an application *must* write to the clock (that is, to change the time setting), it must first open the real-time clock device.

## Related Clock Function Calls

The following are clock functions. Unlike some other operating systems, Verix V 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 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:
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:
yyyy	Indicates year.
mm	Indicates month.
dd	Indicates day.
hh	Indicates hour.
mm	Indicates minutes.
ss	Indicates seconds.
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.

**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. The following calls are related to the date and time information supported by the real-time clock:

## **close()**

---

Releases the resources associated with the clock handle.

**Prototype**

```
int status, hClock;  
status = close (hClock);
```

## open()

Explicitly opens the clock/calendar device, returning its associated device handle.

**NOTE**



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.

**Prototype**

```
int open(const char *device_name, 0);
```

The application that opens the clock receives an event (`EVT_CLOCK`) once per second.

**Parameters**

device_name	= DEV_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).

### Prototype

```
int bytes_read = read(int hClock, char *buffer, int size);
```

### Parameters

*size*      Determines how many bytes actually read (maximum 15).

The 15-byte ASCII character array is returned in *\*buffer* in the format:  
*yyyymmddhhmmssd*.

where,    *yyyy* = year

*mm* = month

*dd* = day

*hh* = hour

*mm* = minutes

*ss* = seconds

*d* = day number (0 = Sunday ... 6 = Saturday)

### Return Values

Success:    Positive number indicating the number of bytes returned.

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

              -1 with errno set to EINVAL: size not 15.

              -1 with errno set to EACCES: buffer invalid.

## 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.

## write()

Sets the system date and time.

**Prototype**    `int bytes_written = write(int hClock, char *buffer, int size);`

**Parameters**

`buffer`    Returns the contents of the buffer in the same format as `read()`, except that only 14 bytes are actually used—the day of the week (15th byte) is determined by the system.

`size`    *Must* be set to 14; the year must be in the range 1990 to 2089.

**Return Values**

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`.

## Timer Function Calls

Timers as system features are related to the clock. See the following calls for more timer-related functionality:

- [clr\\_timer\(\)](#)
- [set\\_timer\(\)](#)
- [SVC\\_WAIT\(\)](#)

## **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);
```

**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.

## 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.

### Parameters

msecs      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.

### Notes

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 code example displays a message for three seconds. Note that it does not check the SVC\_WAIT return value. This is typical, although as noted above it is possible for the call to fail.

## Beepers

The Verix V-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 Vx810 PIN pad, the OS does not contain any pre-defined tunes. It inherits the tune playing feature of the Vx670 terminal, although the notes may sound differently depending on the buzzer hardware and case design. The application causes the Vx810 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`).

## **play\_RTTTL()**

This library 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 return value 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      Name of the music.

### ***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.

## **close()**

Releases the handle associated with the beeper.

**Prototype**    `int status, hBeeper;`

**Example**    The linked C code file is a beeper example.

## **error\_tone()**

Produces a 100-ms tone at 880 Hz. Control immediately returns to the caller.

**Prototype**    `void error_tone (void);`

## **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(short flag);`

**Parameters**

<code>flag</code>	0 = Keybeeps OFF
	1 = Keybeeps ON

**Return Values**

Success:	0
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : Caller does not own the console device.

## **normal\_tone()**

Produces a 50-ms tone at 1245 Hz. Control immediately returns to the caller.

**Prototype**    `void normal_tone (void);`

## open()

Explicitly opens the sound-generating device, returning its associated device handle.

### Prototype

```
int open(const char *device_name, 0);
```

### NOTE



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.

### Parameters

device_name	= DEV_BEEPER
-------------	--------------

## 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 V.

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
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

**Table 32** Beeper Tones (continued)

Note	N#	Nominal	Note	N#	Nominal
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

## **Internal Printer**

The internal printer communications are affected by the components described below.

### **Internal Printer Function Calls**

The functions in this section control the internal printer communications.

## open()

Opens the printer device. On success, the printer handle is returned, and this handle can be used for `read()`, `write()`, `close()`, and other APIs.

**NOTE**

If an application opens the printer device when the Vx810 PIN pad is connected to a DUET base station, the OS returns a valid handle.

**Prototype**

```
int open ("/dev/com4", int unused);
```

**Return Values**

Success: 0

Failure: -1 with errno set to:

- EBUSY, printer is already owned by another task.

On Vx810 unit,

- EBADF means the device is not connected. This is a normal value for a Vx810 operating as a stand-alone device.
- EBUSY means the OS is loading the printer firmware into the printer microcontroller.

**NOTE**

On the Vx810 PIN pad, the OS returns ENODEV if an application attempts to open the printer device. The Printer Diagnostics screens are removed from the System Mode menus. The application may operate an external RS-232 printer through the COM port but this is entirely managed in an application space with no additional printer-specific functions provided by the OS. Connection to an external printer depends on the COM port configuration used in the specific system installation.

## 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.

### Prototype

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

### Parameters

handle	Printer handle.
buffer	Pointer to store the data read.
length	Maximum number of bytes to store in buffer.

### Return Values

Success:	$\geq 0$ : Number of bytes stored in <code>buffer</code> . Maximum is <code>length</code> .
Failure:	-1 with <code>errno</code> set to <code>EBADF</code> : <code>handle</code> is invalid. -1 with <code>errno</code> set to <code>EACCES</code> : <code>buffer</code> is an invalid pointer. -1 with <code>errno</code> set to <code>EINVAL</code> : <code>length</code> is negative.

## write()

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

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

**Parameters**

<code>handle</code>	Printer handle.
<code>buffer</code>	Pointer to the data to write.
<code>length</code>	Number of bytes in buffer.

**Return Values**

Success:	$\geq 0$ : Number of bytes in buffer processed.
Failure:	-1 with <code>errno</code> is set to <code>EBADF</code> : <code>handle</code> is invalid. -1 with <code>errno</code> is set to <code>EACCES</code> : <code>buffer</code> is an invalid pointer. -1 with <code>errno</code> is set to <code>EINVAL</code> : <code>length</code> is negative.

## **close()**

---

Releases ownership of the printer. If anything is queued for printing, it is lost.

**Prototype**    `int close (int handle);`

**Parameters**    `handle`        Printer handle.

**Return Values**    Success:        0: Successful close.

Failure:        -1 with `errno` set to `EBADF`: with `If handle` is invalid.

## get\_opn\_blk()

Copies the current Opn\_Blk structure into the caller's ob structure. See [set\\_opn\\_blk\(\)](#) for additional details on the opn\_blk() for the printer.

**Prototype**    `int get_opn_blk (int handle, Opn_Blk *ob);`

**Parameters**

<code>handle</code>	Printer handle.
<code>ob</code>	Pointer to open block structure containing configuration.

**Return Values**

Success:	0: Successful close.
Failure:	-1 with errno set to EBADF: with If handle is invalid.

## **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_Blk` information (for `get_opn_blk()`), but not validate the parameters.

**Prototype**    `int set_opn_blk (int handle, Opn_Blk *ob);`

**Parameters**

<code>handle</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>If handle</code> is invalid.

## get\_port\_status()

Copies current port status information to caller's 4-byte buffer as follows and returns a result code indicating whether or not any output is currently queued for the printer.

The Vx810 PIN pad communicates with the printer MCU through a USB UART. This USB UART creates a delay in receiving the response characters from the printer MCU because the USB UART collects serial Rx data from the MCU until its 64-byte buffer is filled or until a timeout occurs. At that point, it sends the buffered data to the Vx810 PIN pad. If an application sends a sequential series of commands to the printer, which causes a total of 64 or more bytes of response data from the MCU, the application sees the `get_port_status()` Rx data count go from 0 to 64 in one increment; this reflects the expected behavior of the USB UART—that it sends the contents of its 64-byte buffer when it is full.

**Prototype** `int get_port_status(int handle, char *buffer);`

### 4-byte Buffer Contents

- |          |  |
|----------|--|
| 1st byte | The amount of input pending.<br>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.   |
| 3rd byte | Number of output slots available. This is computed as maximum slots less slots in use.<br><br>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. |
| 4th byte | Constant. CTS detected and DCD present.  |

### 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)       |

**Parameters**

handle      Printer handle.  
buffer      Pointer to buffer to store printer status

**Return Values**

Success:    0: No output pending.  
              >0: Output pending.  
Failure:    -1 with errno set to EBADF: handle is invalid.  
              -1 with errno set to EACCES: buffer is an invalid pointer.

## **reset\_port\_error()**

Has no effect and the corresponding error indicators are always 0. In general for Verix V communication ports, `reset_port_error()` resets the error indicators for parity, framing, and overrun errors, and the break indicator.

**Prototype**    `int reset_port_error(int handle);`

### ***Return Values***

Success: 0

Failure: -1 with `errno` set to `EBADF`: handle is invalid.

## **set\_serial\_lines()**

The standard Verix V communication port `set_serial_lines()` API is not applicable to the printer. `set_serial_lines()` normally sets or resets DTR, RTS, and BRK based on `buffer`.

### **Prototype**

```
int set_serial_lines(int handle, char *buffer);
```

### **Parameters**

`handle`      Printer handle.

`buffer`      Not used.

### **Return Values**

Success:    0

Failure:    -1 with `errno` set to `EBADF`: `handle` is invalid.

## **set\_fifo\_config()**

The standard Verix V communication port `set_fifo_config()` API is not applicable to the printer. `set_fifo_config()` normally sets transmit FIFO length based on `buffer`.

### **Prototype**

```
int set_fifo_config (int handle, char *buffer);
```

### **Parameters**

<code>handle</code>	Printer handle.
<code>buffer</code>	Not used.

### **Return Values**

Success: 0

Failure: -1 with `errno` set to `EBADF`: handle is invalid.

## **get\_fifo\_config()**

The standard Verix V communication port `get_fifo_config()` API is not applicable to the printer. `get_fifo_config()` normally gets the current FIFO configuration. It normally stores constant values (0) in the buffer.

### **Prototype**

```
int get_fifo_config (int handle, char *buffer);
```

### **Parameters**

`handle`      Printer handle.

`buffer`      `*buffer` and `*(buffer+1)` are set to 0.

### **Return Values**

Success: 0

Failure: -1 with `errno` set to `EBADF`: `handle` is invalid.,

-1 with `errno` set to `EACCES`: `buffer` is an invalid address.

## **set\_fc\_config()**

The standard Verix V communication port `set_fc_config()` API is not applicable to the printer. `set_fc_config()` normally sets the hardware flow control configuration based on `buffer`.

### **Prototype**

```
int set_fc_config (int handle, char *buffer);
```

### **Parameters**

`handle`      Printer handle.

`buffer`      Not used.

### **Return Values**

Success: 0

-1      If `handle` is invalid, `errno` is set to `EBADF`.

## **get\_fc\_config()**

The standard Verix V communication port `get_fc_config()` API is not applicable to the printer. `get_fc_config()` normally gets the current hardware flow control configuration. It stores the constant values (0) in the buffer.

### **Prototype**

```
int get_fc_config (int handle, char *buffer);
```

### **Parameters**

handle      Printer handle.  
buffer      \*buffer and \*(buffer+1) are set to 0.

### **Return Values**

0	Success
-1	If handle is invalid, errno is set to EBADF. If buffer is an invalid address, errno is set to EACCES.

## 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 V<sup>x</sup>5xx/V<sup>x</sup>6xx may be different from other printers. When a parsing error is detected, V<sup>x</sup>5xx/V<sup>x</sup>6xx, like existing printers, throws the current command away. On Verix V-based terminals, the character that caused the parsing error is also 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
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>;	Ejects <n> lines.
<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 V-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 SRAM.
<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. V <sup>X</sup> 5xx/V <sup>X</sup> 6xx terminal ID is "P".
<ESC>I	Ignored in Verix V-based terminals.
<ESC>l<s><t>;	Selects font table for printing and downloading.
<ESC>m<c><r1>...<rn>;	Downloads fonts into SRAM.
<ESC>p<n0>,<n1>;	Sets the maximum dots on per pulse (portable units only).
<ESC>S<n>;	Ignored in Verix V-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 0.14mm. 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>;**

---

Ejects *<n>* lines. Causes the printer to eject *n* lines of the currently specified height. The length of paper ejected is calculated as:

$$(Number\ of\ Lines \times Line\ Height) \div 8$$

8 sets the height in mm; use 203.2 for inches.

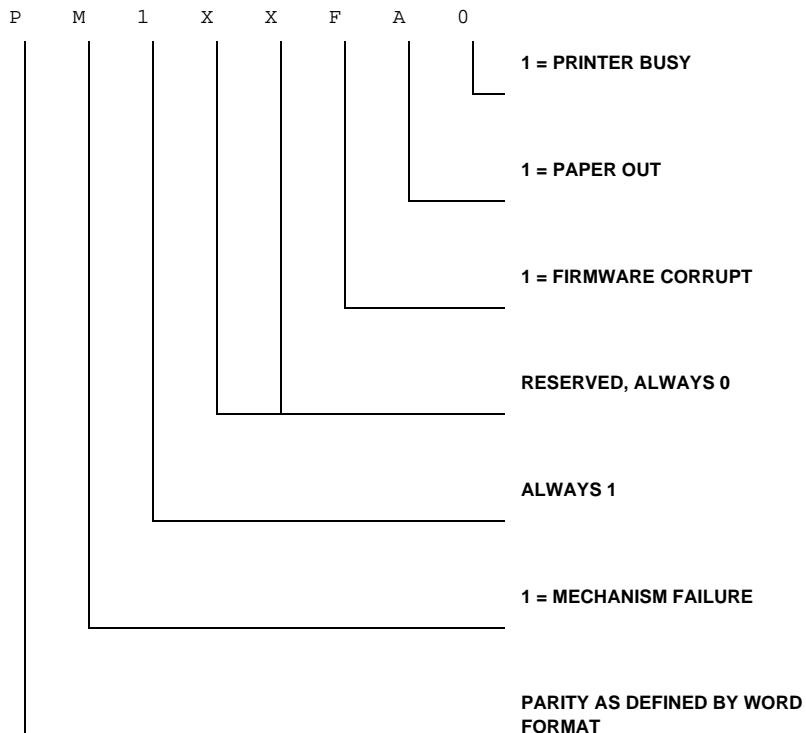
A zero value is ignored; the maximum value is 255.

## <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 V-based terminals report to the host as follows:



**Figure 14      Verix V-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



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>;**

Selects country parameter that affects printing certain character images in countries other than the U.S.A. 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>**i**

---

Sends a request for the printer identity code. The ID code for Verix V-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:

- 1 selects  $16 \times 16$  font.
- 2 selects  $5 \times 8$  font and 24 column mode. (The  $5 \times 7$  fonts are printed in double width and double height.)
- 3 selects  $8 \times 14$  font and 32 column mode.
- 4 selects  $8 \times 14$  font and select 42 column mode.
- 5 selects  $24 \times 24$  font.
- 6 selects  $32 \times 32$  font.
- 7 selects  $48 \times 48$  font.
- 8 selects  $64 \times 64$  font.

<t> Table ID, normally values from 0 to 64 (see \*PRTFNT).

- 0 selects the built-in font table.
- 1–64 select download fonts table.

The  $5 \times 8$  font uses table  $t$  to hold the font image.

The  $8 \times 14$  font uses tables  $t$  and  $t+1$  to hold the font image.

The  $16 \times 16$  font uses tables  $t$ ,  $t+1$ ,  $t+2$ ,  $t+3$  to hold the font images.

The  $24 \times 24$  font uses tables  $t$ ,  $t+1$ ,  $t+2$  through  $t+8$  to hold the font images.

The  $32 \times 32$  font uses tables  $t$ ,  $t+1$ ,  $t+2$  through  $t+15$  to hold the font images.

The  $48 \times 48$  font uses tables  $t$ ,  $t+1$ ,  $t+2$  through  $t+35$  to hold the font images.

The  $64 \times 64$  font uses tables  $t$ ,  $t+1$ ,  $t+2$  through  $t+63$  to hold the font images.

The font table is selected for printing or downloading according the value of  $<s>$  and  $<t>$ . Refer to <ESC>m<c><r1>...<rn>; for font download escape code information.

**Example** <ESC>l37 sets the current font table to font table 7, font size  $8 \times 14$ , and 32 column per line. From then on, the printer prints the  $8 \times 14$  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  $5 \times 8$  and  $8 \times 14$  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  $8 \times 14$ , 42 column setting ( $<s> = 4$ ).

**Table 37 <s> <t> Printable Codes Size Available Mode, Partial List**

<b>&lt;s&gt;</b>	<b>&lt;t&gt;</b>	<b>Printable Codes</b>	<b>Size</b>	<b>Column</b>	<b>Notes</b>
2	0	20h–FFh	5 × 7	24	(built-in font)
2	1	00h–7Fh	5 × 7	24	
2	2	00h–7Fh	5 × 7	24	
.	.	.	.	.	
.	.	.	.	.	
2	9	.	.	.	
2	10	00h–7Fh	5 × 7	24	
.	.	.	.	.	
.	.	.	.	.	
3	0	20h–FFh	8 × 14	32	(built-in font)
3	1	00h–7Fh	8 × 14	32	
3	3	00h–7Fh	8 × 14	32	
.	.	.	.	.	
.	.	.	.	.	
3	9	00h–7Fh	8 × 14	32	
3	11	00h–7Fh	8 × 14	32	
.	.	.	.	.	
.	.	.	.	.	
4	0	20h–FFh	8 × 14	42	(built-in font)
4	1	00h–7Fh	8 × 14	42	
4	3	00h–7Fh	8 × 14	42	
.	.	.	.	.	
.	.	.	.	.	
4	9	00h–7Fh	8 × 14	42	
4	11	00h–7Fh	8 × 14	42	
.	.	.	.	.	
.	.	.	.	.	

## **<ESC>m<c><r1>...<rn>;**

Downloads fonts into SRAM.

**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 code 41h in table 5	.oooo. <r1> = 0Eh .o.o. <r2> = 0Ah .o.o. <r3> = 0Ah oo..oo <r4> = 1Bh o...o <r5> = 11h o...o <r6> = 11h oooooo <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:

&lt;ESC&gt;m&lt;c&gt;&lt;r1&gt;...&lt;rn&gt;;

**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...o <r6> =94h  
              o...o...o <r7> =94h  
              .o...o... <r8> =64h  
              ..... <r9> =00h  
              ooooooo <r10>=FFh  
              .....o <r11>=01h  
              ooooooo <r12>=FFh  
              o..... <r13>=80h  
              ooooooo <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.

If you've already sent <ESC>l161 ; 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..o..o... <r4> =04h, <r12>=C8h  
              .ooooooo.....o.. <r5> =7Fh, <r13>=04h  
              .....o.....o. <r6> =04h, <r14>=02h  
              .....o..... <r7> =04h, <r15>=00h  
              .....ooooooo.... <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  
 .ooooooo..ooooooo.o..... <r4> =7Eh <r12>=7Eh <r20>=80h  
 .....o....o....o....o..... <r5> =02h <r13>=40h <r21>=80h  
 .....o....o....o....o..... <r6> =00h <r14>=00h <r22>=82h  
 ..o....o....o....ooooooo <r7> =22h <r15>=89h <r23>=FFh  
 ..ooooooooooooooo.o.....o. <r8> =3Fh <r16>=FDh <r24>=02h  
 ..o....o....o....oo..oo. <r25>=22h <r33>=89h <r41>=36h  
 ..o....o....o....oo....o. <r26>=22h <r34>=8Bh <r42>=24h  
 ..oooooo..oooooo.o....o.... <r27>=3Eh <r35>=FAh <r43>=28h  
 ..o....o....o....o..... <r28>=2Dh <r36>=00h <r44>=20h  
 ..o....o....o....o..... <r29>=08h <r37>=88h <r45>=20h  
 ..ooooooooooooooo....o.... <r30>=1Fh <r38>=FCh <r46>=20h  
 ..o....o....o....oo..... <r31>=10h <r39>=80h <r47>=60h  
 ..oo....o....o....oo..... <r32>=30h <r40>=90h <r48>=60h  
 .o....oooooooooooo....ooo.... <r49>=5Fh <r57>=F8h <r65>=70h  
 ..o....o....o....oo....o.... <r50>=10h <r58>=80h <r66>=D0h  
 ..oooooooooooo....o....oo.... <r51>=1Fh <r59>=F8h <r67>=98h  
 ..o....o....o....o....oo.... <r52>=10h <r60>=80h <r68>=8Ch  
 ..o....o....o....oo....ooo.... <r53>=10h <r61>=89h <r69>=8Eh  
 ..oooooooooooo....o....ooo.... <r54>=1Fh <r62>=FDh <r70>=07h  
 ..o....o....o....o....o.... <r55>=10h <r63>=02h <r71>=02h  
 .....o....o....o....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

## SYSTEM DEVICES

<ESC>m<c><r1>...<rn>;

32 × 32 font	<r2>	<r9>	<r17>	<r25>
image map	:	:	:	:
	<r8>	:	:	:
		<r16>	<r24>	<r32>
		<r33>	<r41>	<r49>
		:	:	:
		:	:	:
		<r40>	<r48>	<r56>
				<r64>
		<r65>	<r73>	<r81>
				<r89>
		:	:	:
		:	:	:
		<r72>	<r80>	<r88>
				<r96>
		<r97>	<r105>	<r113>
				<r121>
		:	:	:
		:	:	:
		<r104>	<r112>	<r120>
				<r128>

48 × 48 font	<r1>	<r9>	<r17>	<r25>	<r33>	<r41>
image map	:	:	:	:	:	:
	<r8>	<r16>	<r24>	<r32>	<r40>	<r48>
		<r49>	<r57>	<r65>	<r73>	<r81>
		:	:	:	:	:
		<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>

64 × 64    <r1>    <r9>    <r17>    <r25>    <r33>    <r41>    <r49>    <r57>  
 font            :            :            :            :            :            :            :  
 image map    <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**

&lt;hhx&gt;

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>;

Downloads a graphic image into SRAM.

### 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 V-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 V-based terminals ignore this field.
<t>	<ul style="list-style-type: none"> <li>• 2 is 8-bit format (see <a href="#">Figure 15</a>).</li> </ul>
<w>	Image ID (see <a href="#">*PRTLGO</a> ).
<h>	The image width; acceptable range 16 to 384.
<b1>...<bn>	The image height; acceptable range is 16 to 240.
	The image data.

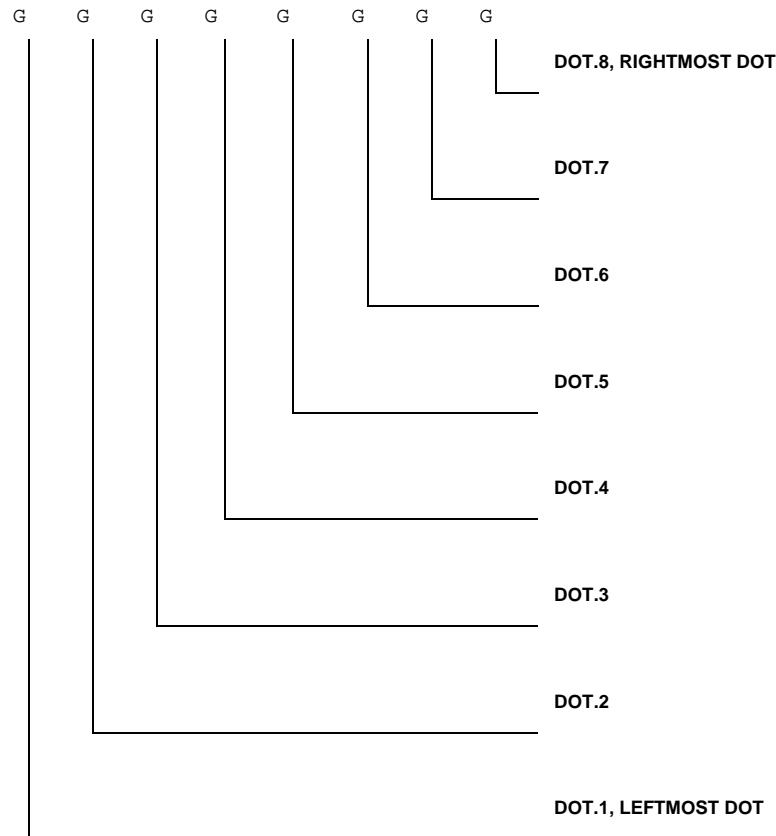
**Figure 15     8-bit Format*****Example***

Image format = 0

Graphic image (logo) = 0

Image width = 128

Image height = 100

Byte count for a image dot line =  $128 \div 8 = 16$  bytesTotal bytes for the image =  $16 \times 100 = 1600$  bytes

&lt;ESC&gt;GL0,0,128,100;&lt;b1&gt;&lt;b2&gt;...&lt;b1600&gt;

## <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 ( $< n > + 7$ ) / 8 . Higher values extend the thermal head activation times to produce darker printouts.

See also \*DARK This command can be used to compensate for lower-sensitivity paper or 2-ply paper.

**Prototype**    <ESC>w<n>;

**Parameters**

<n>              Sets scale factor to ( $< n > + 7$ ) / 8.  
 The allowed range for <n> is:  
 1-25 for Vx510, Vx570, Vx610, and Vx510G.  
 1-9 for Vx670.  
 1-29 for V5 and Vx810 integrated base.

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>p<n0>,<n1>;**

---

Portable terminals only. Sets maximum number of dots on per fire pulse.

## <ESC>CS;

Retrieves firmware checksum and version.

**NOTE**

Verix V-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.

---

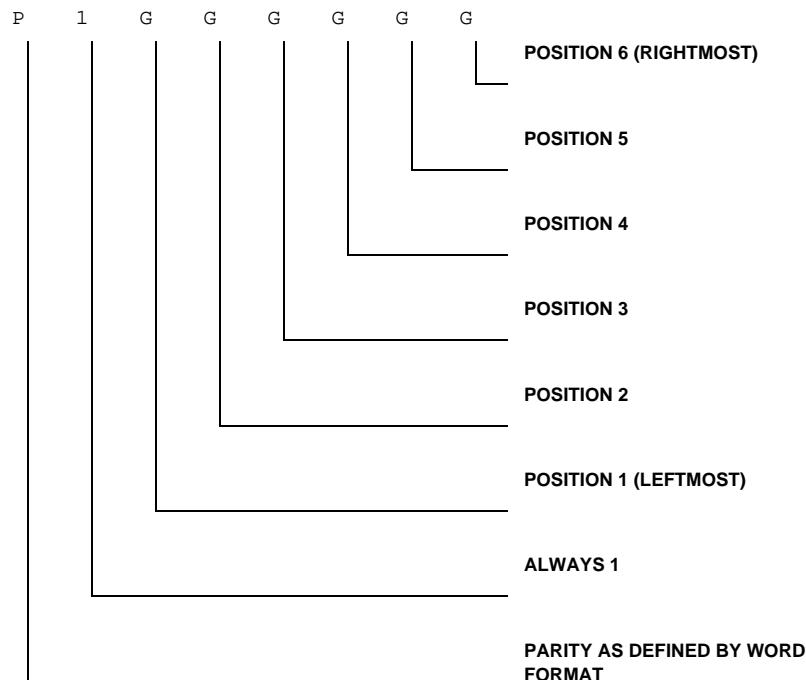
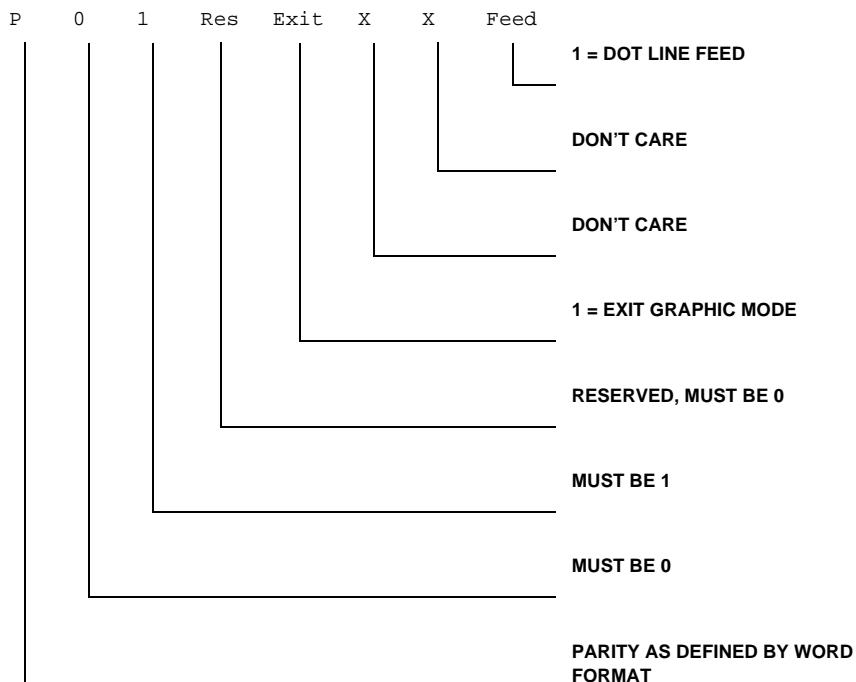


Figure 16      Graphic Image Code

## 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 17](#). 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 17     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 SRAM 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 – 5x8, 8x14 in 32-column mode, and 8x14 in 42-column mode. The 5x8 character size is always printed double height and double width resulting in 24 columns. Unless otherwise selected, the default is 8x14 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 8x14 font and is organized as follows:

- Offset 0 – 7 : 5x8 font character 0
- Offset 8 – 15: 5x8 font character 1
- ...
- Offset 2040 – 2047 : 5x8 font character 255
- Offset 2048 – 2061 : 8x14 font character 0
- Offset 2062 – 2075 : 8x14 font character 1
- ...
- Offset 5618 – 5631 : 8x14 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
5x8	1
8x14	2
16x16	4
24x24	9
32x32	16
48x48	36
64x64	64

- <ESC>m<c><r1>...<rn>; downloads fonts into the SRAM
- <ESC>H<hh1>, . . . , <hhn>; prints hex code characters

The printer does not prevent fonts from being partially overwritten. For example, a 16x16 font can be downloaded into table 1 and it will actually be stored in tables 1-4. Later, a 5x8 font can be downloaded to table 2 and a portion of the 16x16 font are overwritten.

On Verix V-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 V 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 V-based terminals store the font in RAM 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 V-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 SRAM, and another command to print the downloaded image without resending the graphic image data.

**Support for Paper-out LED**

The Vx570 terminal's 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 Vx670 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 <sup>a</sup>	Red on solid.
Battery Charging	Orange <sup>b</sup> , 1 Hz flash.
-	Green <sup>c</sup> 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 Vx810 CTLS is installed in the SD card slot on the Vx810 PIN pad. The OS operates the Vx810 CTLS in SDIO version 1.0 mode.

**NOTE**

The Vx810 CTLS must not be inserted or removed while the Vx810 is operating.

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 Vx810 CTLS device. The Vx810 CTLS returns a unique ID information in response to the OS initialization commands.

The OS includes a driver for the Vx810 CTLS module identified in `get_component_vars()` as `SDIOCTLSS.BIN`.

**NOTE**

The OS does not support removing or connecting the Vx810 CTLS device while an application is running.

**SDIO Device  
Firmware  
Download**

The Contactless device contains firmware that manages the Contactless card protocol.

**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**

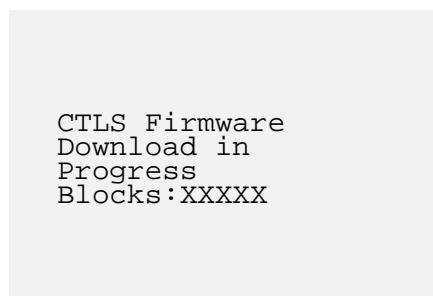
The contactless firmware download is not a frequent event as the contactless firmware must first be certified by the card associations before it can be used for payment transactions.

If the file VIVOCTL.S.FRM is present and authentic, and the Vx810 CTLS card is present, the System Mode 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 the 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 VIVOCTL.S.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 VIVOCTL.S.FRM is present and authentic, and if the Vx810 CTLS card is present, the OS loads it into the contactless device.

The process takes approximately four minutes and the following message is displayed:



“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 VIVOCTL.S.FR.M 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 Vx810 CTLS until DEV/CTLS has been opened after the successful firmware download.

**NOTE**



VIVOCTL.S.FR.M is not a protected file and can be deleted using the System Mode 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:



The firmware is retained and another download is attempted the next time the Vx810 reboots with the Vx810 CTLS connected.

- 6 The file VIVOCTL.S.FR.M is downloaded into the Vx810 using any supported download method — USB Memory Card, VeriCentre, direct download, etc.
- 7 VIVOCTL.S.FR.M file versions are managed similar to how modem profiles are managed. Each version is kept in a .ZIP file named for the unique VIVOCTL.S.FR.M file version it contains. When the .ZIP file is downloaded into the terminal, the VIVOCTL.S.FR.M file is then extracted and downloaded.
- 8 If VIVOCTL.S.FR.M 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 VIVOCTL.S.FR.M 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 V `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 the Vx810 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 Vx810 CTLS. Receive data is held in a local 1024byte FIFO.

**CAUTION**

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 Vx810 CTLS at the same time it is attempting to send data to the application, either of the transmit or receive data will be corrupted.

**CAUTION**

\*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 RAM is not limited.

**set\_opn\_blk() and get\_opn\_blk()**

A subset of the standard Verix V set\_opn\_blk() and get\_opn\_blk() API is supported for the Vx810 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.

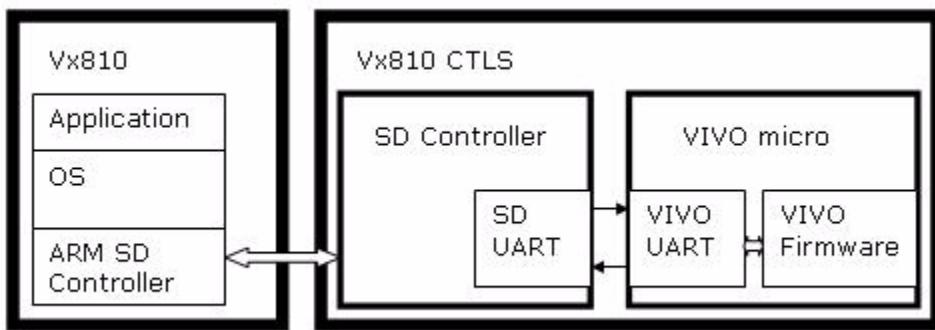
The OS uses the default factory settings until an application calls set\_opn\_blk() with a different setting. The OS retains the new setting until it is changed by another application call to set\_opn\_blk().

**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.

The figure below illustrates the Vx810 CTLS communication architecture.



**Figure 18 Vx570 CTLS Communication Architecture**



**CAUTION** Any change to the SD UART settings can render the Vx810 CTLS inoperable. The `set_opn_blk()` API in the 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 V terminal, and the potential that any application can change the SD and ViVO UART settings (including applications invoked by System Mode 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 contactless device. By default, the contactless device does not generate events. When configured to generate events, 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.

**Prototype** `int set_event_bit(int handle, long flag);`

### Parameter

- |        |   |
|--------|---|
| handle | The Contactless device handle returned from <code>open()</code> . |
| flag   | The bit value of the event bit to use.                            |

**Example**

```
set_event_bit(ctlsHdl, 0x80)
set_event_bit(ctlsHdl, 128)
set_event_bit(ctlsHdl, (1L<<7 ))
#define MY_CTLSS_EVENT 0x80
set_event_bit(ctlsHdl, MY_CTLSS_EVENT)
```

The above examples all cause bit 7 to be the contactless event. The `set_event_bit()` returns 0 if it is successful.

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 contactless device in the default mode - events turned off.  |
| Failure | -1 with <code>errno</code> set to <code>EINVAL</code> , when API is called with: <ul style="list-style-type: none"><li>• one of the reserved event bits,</li><li>• a flag that defines more than 1 event bit (i.e., <code>set_event_bit(ctlsHdl, 0x90)</code> returns error), and</li><li>• a handle set to any non-contactless device handle (i.e., <code>set_event_bit(ctlsHdl, 0x90)</code> returns error).</li></ul> |

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()

Determines the event generated by the contactless device.

**Prototype** long get\_event\_bit(int handle);

**Parameter**

handle The Contactless device handle returned from `open()`.

**Return Values**

Success 0, and places the contactless device in the default mode - events turned off.  
Failure -1 with `errno` set to `EINVAL`, when API is called with handle set to any non-contactless device handle.

**NOTE**



Verix V 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 Vx810 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 an 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\_CTLs.

**NOTE**

It is possible for an application to see the SD\_FLASH bit set if an SD Memory card is inserted in the Vx810. However, the OS does not provide any API for the application to interact with the SD memory cards.

## USB Fingerprint Reader Device

The MorphoSmart™ Optic (MSO) is a precision optical biometric fingerprint terminal that captures fingerprint images for enrollment, authentication, and identification. Depending on the model, the MorphoSmart™ Optic offers remote and/or local matching. Remote matching captures an image and sends it to a host PC running MorphoSoft™ for template comparison. Local matching captures, encodes, and discards the original image, and compares the retained template to the local database on the terminal.

The MSO300 model with USB interface communicates with VeriFone's Vx570 USB Host. To create a communication channel for the MSO300 fingerprint reader device, a USB device driver is created on the Vx570 OS. This enables the MSO300 fingerprint reader device to perform the biometric function of acquiring and encoding images, discarding images, and sending templates to the Vx570 terminal using the USB Interface.

---

**NOTE**



Images are generally discarded unless otherwise specifically requested by the host.

---

### USB Low Layer Protocol

The MorphoSmart™ terminal is processed as a Communication Device Class (CDC), in accordance with the USB Device Class Specifications, Version 1.1. The device can also be used with a USB2.0 controller, but still uses the USB 1.1 transfer rate.

Below are the communication rules governing the channel and the device hardware and software.

#### Data Transmission

The data transmitted is either a command (from Host to MorphoSmart™) or a reply to a command (MorphoSmart™ to Host), in accordance with the ILV protocol.

#### Frame Format

The data to be transmitted is included in a frame according to the format described below:

**Table 40 Data Frame Format**

Part	Description
Start of frame	A set of 4 ASCII characters : 'S'Y'N'C
Length of frame	4 bytes
1 complement of the length of the frame	4 octets
Data	Command or answer (ILV format as described in MorphoSmart Host System Interface Specification)
End of frame	A set of 2 ASCII characters : 'E'N

## Data Structure

The application data has three fields:

**Table 41 ILV Structure**

Initials	Description
I	Identifier - the identifier of the command.
L	Length - the length of the Value field in byte.
V	Value - the data or parameters.

This data structure is a variable. The Value field can contain optional ILV formatted data and its length is a variable. The Identifier field is 8 bits long and the L field is 16 bits long

Identifier	Length	Value
1 byte	2 bytes	<Length> bytes

For a length value of >64k, the ILV is formatted as follows:

Identifier	Length	Length Value	Value
1 byte	0xFFFF	4 bytes	<Length> bytes



All parameters are in bytes. Only ASCII or corresponding table parameters are specified. The data are in Little Endian format.

## MorphoSmart ILV Functions

Below are the ILV Function of MorphoSmart™.

**Table 42 MorphoSmart ILV Functions**

ID	Description	Function
<b>Initialization Functions</b>		
0x05	GET_DESCRIPTOR	Retrieves information from the biometric module.
<b>Biometric Functions</b>		
0x20	VERIFY	Captures and verifies against a reference template.
0x21	ENROLL	Captures and adds to database and/or export templates.
0x22	IDENTIFY	Captures and identifies against the local database.
0x23	VERIFY MATCH	Verifies a list of reference templates against a search template.
0x24	IDENTIFY MATCH	Identifies a search template against the local database.

**Table 42 MorphoSmart ILV Functions (continued)**

ID	Description	Function
<b>Database Management Functions</b>		
0x30	CREATE BASE	Creates a local database.
0x32	ERASE BASE	Erases the local database (destroys all records, not database structure).
0x34	ERASE ALL BASE	Erases all local database (destroys all records, not database structure).
0x3B	DESTROY BASE	Destroys the local database.
0x33	DESTROY ALL BASE	Destroys all the local databases.
0x35	ADD RECORD	Adds a record to the local database.
0x36	REMOVE RECORD	Removes a record from the local database.
0x38	FIND USER BASE	Searches for a record that matches a specified pattern.
0x3C	UPDATE PUBLIC DATA	Updates a database public field.
0x3D	UPDATE PRIVATE DATA	Updates a database field (public or private).
0x3E	GET PUBLIC FIELDS	Retrieves a database public field list.
0x3F	GET DATA	Retrieves a database public field.
0x07	GET BASE CONFIG	Retrieves a database configuration.
<b>Security Management Functions</b>		
0x80	SECU GET CONFIG	Retrieves the MorphoSmart™ security configuration.
0x81	SECU READ CERTIFICATE ID	Retrieves the X.509 certificate from MSO certification path.
0x82	SECU STORE CERTIFICATE	Loads a host X.509 certificate.
0x83	SECU STORE PKCS12	Loads a PKCS#12 token.
0x84	SECU MUTUAL AUTH INIT 1	The first step to establish a secure tunnel.
0x85	SECU MUTUAL AUTH INIT 2	The second step to establish a secure tunnel.
0x86	SECU PROTOCOLE	The security envelope.
<b>OTP Management Functions</b>		
0xB0	OTP ENROLL USER	Enrolls the user in the OTP token in the OTP database.
0xB1	OTP GENERATE	Generates the OTP.
0xB2	OTP GET STATUS	Gets the status of the OTP token.
0xB3	OTP SET PARAMETERS	Sets the parameters of the OTP token.
<b>Configuration</b>		
0x90	GET_MSO_CONFIG	Retrieves the value of one configuration parameter.

**Table 42 MorphoSmart ILV Functions** (continued)

ID	Description	Function
<b>Database Management Functions</b>		
0x30	CREATE BASE	Creates a local database.
0x32	ERASE BASE	Erases the local database (destroys all records, not database structure).
0x34	ERASE ALL BASE	Erases all local database (destroys all records, not database structure).
0x3B	DESTROY BASE	Destroys the local database.
0x33	DESTROY ALL BASE	Destroys all the local databases.
0x35	ADD RECORD	Adds a record to the local database.
0x36	REMOVE RECORD	Removes a record from the local database.
0x38	FIND USER BASE	Searches for a record that matches a specified pattern.
0x3C	UPDATE PUBLIC DATA	Updates a database public field.
0x3D	UPDATE PRIVATE DATA	Updates a database field (public or private).
0x3E	GET PUBLIC FIELDS	Retrieves a database public field list.
0x3F	GET DATA	Retrieves a database public field.
0x07	GET BASE CONFIG	Retrieves a database configuration.
<b>Security Management Functions</b>		
0x80	SECU GET CONFIG	Retrieves the MorphoSmart™ security configuration.
0x81	SECU READ CERTIFICATE ID	Retrieves the X.509 certificate from MSO certification path.
0x82	SECU STORE CERTIFICATE	Loads a host X.509 certificate.
0x83	SECU STORE PKCS12	Loads a PKCS#12 token.
0x84	SECU MUTUAL AUTH INIT 1	The first step to establish a secure tunnel.
0x85	SECU MUTUAL AUTH INIT 2	The second step to establish a secure tunnel.
0x86	SECU PROTOCOLE	The security envelope.
<b>OTP Management Functions</b>		
0xB0	OTP ENROLL USER	Enrolls the user in the OTP token in the OTP database.
0xB1	OTP GENERATE	Generates the OTP.
0xB2	OTP GET STATUS	Gets the status of the OTP token.
0xB3	OTP SET PARAMETERS	Sets the parameters of the OTP token.
<b>Configuration</b>		
0x90	GET_MSO_CONFIG	Retrieves the value of one configuration parameter.

**Table 42 MorphoSmart ILV Functions (continued)**

ID	Description	Function
<b>Database Management Functions</b>		
0x30	CREATE BASE	Creates a local database.
0x32	ERASE BASE	Erases the local database (destroys all records, not database structure).
0x34	ERASE ALL BASE	Erases all local database (destroys all records, not database structure).
0x3B	DESTROY BASE	Destroys the local database.
0x33	DESTROY ALL BASE	Destroys all the local databases.
0x35	ADD RECORD	Adds a record to the local database.
0x36	REMOVE RECORD	Removes a record from the local database.
0x38	FIND USER BASE	Searches for a record that matches a specified pattern.
0x3C	UPDATE PUBLIC DATA	Updates a database public field.
0x3D	UPDATE PRIVATE DATA	Updates a database field (public or private).
0x3E	GET PUBLIC FIELDS	Retrieves a database public field list.
0x3F	GET DATA	Retrieves a database public field.
0x07	GET BASE CONFIG	Retrieves a database configuration.
<b>Security Management Functions</b>		
0x80	SECU GET CONFIG	Retrieves the MorphoSmart™ security configuration.
0x81	SECU READ CERTIFICATE ID	Retrieves the X.509 certificate from MSO certification path.
0x82	SECU STORE CERTIFICATE	Loads a host X.509 certificate.
0x83	SECU STORE PKCS12	Loads a PKCS#12 token.
0x84	SECU MUTUAL AUTH INIT 1	The first step to establish a secure tunnel.
0x85	SECU MUTUAL AUTH INIT 2	The second step to establish a secure tunnel.
0x86	SECU PROTOCOLE	The security envelope.
<b>OTP Management Functions</b>		
0xB0	OTP ENROLL USER	Enrolls the user in the OTP token in the OTP database.
0xB1	OTP GENERATE	Generates the OTP.
0xB2	OTP GET STATUS	Gets the status of the OTP token.
0xB3	OTP SET PARAMETERS	Sets the parameters of the OTP token.
<b>Configuration</b>		
0x90	GET_MSO_CONFIG	Retrieves the value of one configuration parameter.

**Table 42 MorphoSmart ILV Functions** (continued)

ID	Description	Function
0x91	MODIFY_MSO_CONFIG	Modifies the value of one configuration parameter.
<b>Unlocking</b>		
0x8B	GET_UNLOCK_SEED	Retrieves a seed to initialize the unlock process.
0x8C	UNLOCK	Unlocks the functions using the seed.
<b>Miscellaneous</b>		
0xEE	CONFIG UART	Changes the UART configuration.
0x70	ASYNC MESSAGE	The asynchronous message.
0x71	CANCEL	Cancels a live finger acquisition.
<b>Invalid LV</b>		
0x50	ILV_INVALID	Invalid ILV.

### ILV Function Description

Below is a sample implementation of the ILV Get\_Descriptor function.

## ILV Get\_Descriptor

This function fetches the MorphoSmart™ descriptors.

Identifier Value	0x05	
Length Value	0x0001	
Value (Parameters)	Format	1 byte

The format specifies the required descriptors among the values listed below:

ID_FORMAT_TEXT	0x2F	Text format descriptor - all main descriptors formatted in separate character strings.
ID_FORMAT_BIN_VERSION	0x74	Version as returned by the descriptor - the software version descriptor.
ID_FORMAT_BIN_MAX_USER	0x75	Max number of persons enrolled in the base for the descriptor - the maximum number of records that can be stored into the internal database, according to hardware and software configuration of the MorphoSmart™ device.

**Example** Sample code for the `ILV GET_DESCRIPTOR` function.



Other descriptions of the `ILV` function can be found on the MorphoSmart™ Host System Interface Specification.

**Asynchronous Messages** The MorphoSmart™ manages asynchronous messages that indicate the status of a current live acquisition. The asynchronous messages are managed for all live-finger acquisition functions—enroll, verify, and identify. Reception of these messages is fully customizable. Asynchronous information statuses include:

- User directions (press harder, move left, remove finger, etc.,).
- The finger number and acquisition number during the enrollment process.
- A low-resolution image of the finger is received.

A live finger acquisition ends when one of the following occurs:

- Timeout expiration (timeout could be infinite if required).
- A finger is detected.
- Cancel command is executed.

**Types of Asynchronous Messages**

The following are types of asynchronous messages:

**MESSAGE\_COMMAND\_CMD**

This asynchronous message is used to signal an action that the user needs to perform.

Identifier value	MESSAGE_COMMANDS_CMD	
Length value	0X0004	
Value	Commands	4 bytes

**Commands**

0	MORPHO_MOVE_NO_FINGER	No finger is detected.
1	MORPHO_MOVE_FINGER_UP	User must move his finger up.
2	MORPHO_MOVE_FINGER_DOWN	User must move his finger down.
3	MORPHO_MOVE_FINGER_LEFT	User must move his finger to the left.
4	MORPHO_MOVE_FINGER_RIGHT	User must move his finger to the right.
5	MORPHO_PRESS_FINGER_HARDER	User must press his finger harder.
6	MORPHO_LATENT	Finger is in the same place as a previous acquisition. User must move his finger.
7	MORPHO_REMOVE_FINGER	User must remove his finger.
8	MORPHO_FINGER_OK	The finger acquisition is correctly completed.

## MESSAGE\_ENROLLMENT\_CMD

This asynchronous message is used to signal the enrollment status action.

Identifier value	MESSAGE_ENROLLMENT_CMD	
Length value	0X0004	
Value	Finger Number	1 byte
	Finger Total	1 byte
	Capture Number	1 byte
	Capture Total	1 byte

- Finger number - the current number of enrolled fingers (starts from 1).
- Finger total - the total number of fingers to enroll.
- Capture number - the current number of acquisitions of the currently enrolled finger (starts from 1).
- Capture total - the total number of acquisitions of the enrolled finger.

## MESSAGE\_IMAGE\_CMD

This asynchronous message is used to transmit an image for the host GUI

Identifier value	MESSAGE_IMAGE_CMD	
Length value	0X000C + <L>	
Value	Image Header	0x0C bytes
Raw Image	L byte	

- Image header - specifies the image format.
- Raw image - the image data.

## MESSAGE\_CODE\_QUALITY\_CMD

This asynchronous message is used to recover the quality note of the image to be coded. The ILV is encapsulated in the Value field of an asynchronous ILV.

Identifier value	MESSAGE_CODE_QUALITY_CMD	
Length value	0X0001	
Value	Code quality (value between 0 and 255)	1 byte

Code quality is the quality note score of the image to be coded. The value of 255 corresponds to the best quality note score. The code quality depends on the type of finger — a correct finger starts at 40 while a good finger is over 120.

This ILV is sent when the quality score of the finger's picture is greater than the minimum quality score required for storage or comparison.

## MESSAGE\_DETECT\_QUALITY\_CMD

This asynchronous message is used to recover the quality note calculated by the “presence detection” function.

Identifier value	MESSAGE_DETECT_QUALITY_CMD
Length value	0X0001
Value	Detect quality (value between 0 and 255) 1 byte

Detect quality is the quality note score calculated by the “presence detection” function. The value of 255 corresponds to the best quality note score. The detect quality depends on the type of finger — a correct finger starts at 40 while a good finger is over 120.

To avoid a system overload, this ILV is sent only if the communication is free, as the messages are relative to the finger actions (finger move left, remove finger, etc.,) that the user needs to perform. The Host system receives ILV by means of the callbacks system of the asynchronous ILV (encapsulated in the Asynchronous Message ILV 0x71).

**Image** Below are the image ILV functions.

### Export Image

ILV formatted data used with ILV\_ENROLL to export the image that has been captured to extract the minutiae.

Identifier value	ID_EXPORT_IMAGE
Length value	0X0006
Value	Compression Type 1 byte
	RFU 1 byte

### Image Type

For the MorphoSmart™ terminal, this defines the ID\_DEFAULT\_IMAGE 416 x 416 pixels, 500 x 500dpi.

### Compression

Below is the ILV format for compression.

Identifier value	ID_COMPRESSION
Length value	0X0002
Value	Compression Type 1 byte
	RFU 1 byte

## Compression Type

The compression types are:

ID\_COMPRESSION\_NULL

ID\_COMPRESSION\_V1

RFU: Not used. Set to zero (0).

**Image Header** Formatted data containing the image parameters information.

Header Revision	Version number. Current version number is zero (0).	
Header Size	10 (0x0A)	
Header	Row number. Column number Res. Y Res. X Compression Compression Parameter	2 bytes 2 bytes 2 bytes 2 bytes 1 byte 1 byte

- Row number - raw number of the image in pixels.
- Column number - column number of the image in pixels.
- Compression:

ID\_COMPRESSION\_NULL no compression.

ID\_COMPRESSION\_V1 default compression algorithm.

## USB MSO300 Fingerprint Reader API

The OS presents standard APIs to the application such as open, close, read, and write functions. The application is able to access the USB fingerprint reader device as "/dev/bio."

## 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.

### 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.

USB-COM \*



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 UART device. This means that `set_opn_blk()`, `reset_port_error()`,

`set_serial_lines()`, `set_fifo_config()`, `get_open_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_open_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> .

## 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 43      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.
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	Code 128 is more efficient at encoding data than Code 39.  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 System Mode 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 Vx670 converter module, 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**

This cable is not configured to work on Qx120 devices.

- ViVoTech PN 220-2442-00—used to connect with Qx120 Contactless devices.

The System Mode reflects a channel for USB to RS232 downloads. This channel is COM6 port similar to the Vx670 terminal.

**NOTE**

Only one USB to RS232 device (COM6) will be supported at anytime. It is not possible to support Vx670 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 V 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 UART device but within the limitations of the USB UART hardware.

### Events

The Verix V OS has no available event bits. A new API is introduced to set an event for the USB Keyboard device similar to that of the Vx570 Contactless device.

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 new API [set\\_event\\_bit\(\)](#) 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.

### 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.

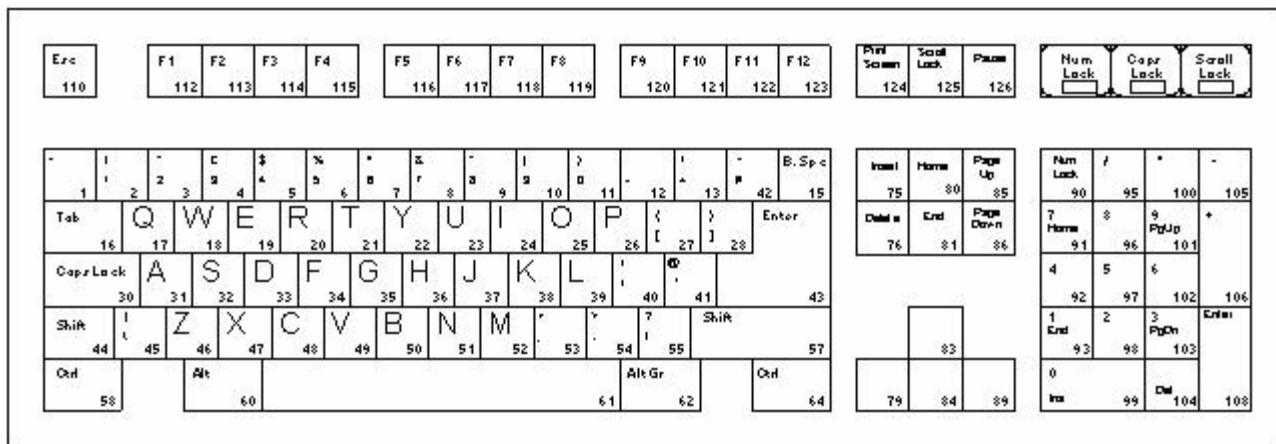
The definition below are reflected in `svc.h`.

```
#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 Vx570 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 19 illustrates a PC keyboard showing 102 keys.



**Table 44 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
19	24
20	2D
21	2C
22	35

**Table 44 Scan Code Table (continued)**

<b>Key Number</b>	<b>AT Code</b>
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
61	29
62	E011
64	E014
75	E070

**Table 44 Scan Code Table (continued)**

Key Number	AT Code
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
121	9
122	78
123	7
124	E012E07C

**Table 44 Scan Code Table (continued)**

Key Number	AT Code
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 45 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 “H\n”. This example shows how to handle the SHIFT+KEY combination.

**Table 46 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

**Table 46 Shift+Key Combination** (continued)

AT Code	Key Number	ASCII Code
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 Vx570 terminal supports the Metrologic barcode scanner, a USB HID (Human Interface Device) that uses the USB keyboard driver available in most Verix V 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 the Vx570 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 `set_event_bit()` 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

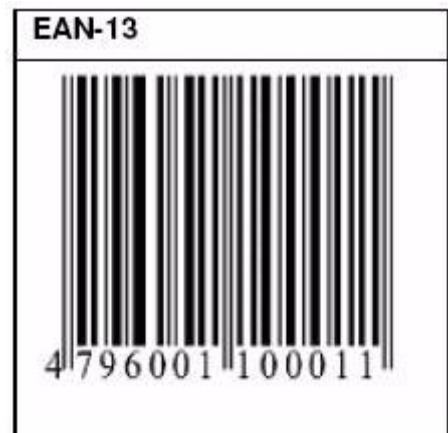
The Metrologic barcode scanner device has a maximum power of 450mA, and since the Vx570 terminal provides the standard USB Host power of 5V 500mA, the device does not need an external power or a powered hub to work.

## Operating Test

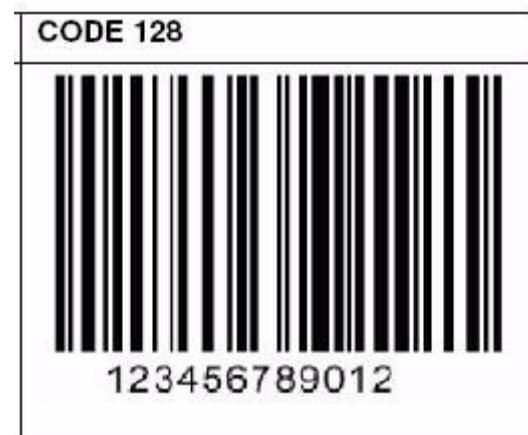
The Metrologic barcode scanner device driver in Verix V OS is only a channel to get the data from the barcode device. The OS does not handle the configuration of Metrologic barcode scanner.

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 included 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.

## **open()**

---

Acquires ownership of the device.

### **Prototype**

```
int open ( "/dev/bio" , int unused );
```

- The application accesses the USB barcode device as “/dev/bar.”
- The application accesses the USB keyboard device or the Metrologic barcode device as “/dev/usbkbd.”
- The application accesses the USB to RS-232 device as “/dev/com6.”

### **Return Values**

Success: Handle.

Failure: -1 with `errno` set to `EINVAL`. A negative count results in an error.

-1 with `errno` set to `EBUSY`. This is returned by the OS if another task currently owns the device.

- If the USB Keyboard is not connected, this returns -1 and set `errno` to `EBADF`.
- If the USB UART is not connected, `open( "/DEV/COM6" , 0 )` returns -1 and `errno` set to `EBADF`.

## read()

Allows the current owner to read data from the device.

**Prototype**    `int read (int handle, char *buffer, int count);`

**Parameters**

handle	Handle to read from.
buffer	Storage area for bytes or message to be read.
count	Size of the buffer

**Return Values**

Success:    Number of bytes transferred.

Failure:    -1 with `errno` is set to `EBUSY`. A negative count results in an error.

              -1 with `errno` is set to `EINVAL`.

- If the USB keyboard is removed after it is opened, this returns -1 and `errno` is set to `ENXIO`.

The Heron D130 barcode scanner device is configured to work as an RS-232 device, thus `set_opn_blk()` must be called first before calling the `read()` function. The data read is a string of characters terminated with CR LF.

In reading data from the Metrologic barcode device, it is recommended to indicate the maximum length of the barcode data to the count parameter so that the complete barcode information can be derived.

## write()

Allows the current owner to write data to the device.

### NOTE



Writing data to the Heron D130 device is not permitted.

### Prototype

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

### Return Values

Success: Number of bytes transferred.

Failure: -1 with `errno` is set to `EINVAL`. A negative count results in an error.

- If the USB UART is removed after it is opened, this returns -1 and `errno` is set to `ENXIO`.
- This function is not supported on Metrologic Barcode device. Calling this function always returns `-EINVAL`.

## close()

Releases ownership of the device so that another application can use it.

**Prototype**    `int close (int handle);`

**Parameters**    `handle`      Handle of the console device.

**Return Values**

Success:    0

Failure:    -1 and errno set to EBADF, if the caller is not the current owner.

**NOTE**



If the application does not close the device, it is not possible to communicate with the device when the USB Keyboard is subsequently connected. The application must close and reopen the device.

For more information, refer to *Verix V Enhancements for Verix V Terminals External Reference Specification*, VPN - 23959.

## USB Device Driver Events

The Verix V OS has no available event bits. A new API is introduced to set an event for the USB Keyboard device similar to that of the V<sup>x</sup>570 Contactless device.

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. 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.

The following are V<sup>x</sup>570 events that the OS issues to the application once the device is used:

**Table 47      USB Driver Events**

Event	Description
EVT_USB	Issued to the MSO300 fingerprint reader, USB keyboard, and USB to RS-232 devices upon connecting/disconnecting them from the V <sup>x</sup> 570 terminal.
EVT_BIO	Issued when there is incoming data from the MSO300 fingerprint reader.
EVT_USB	Issued when the barcode scanner device is connected from the V <sup>x</sup> 570.
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 similar to the COM6 driver on the Verix V Terminal.

## **set\_event\_bit()**

Allows the application to select one of the event bits that are not used and then assigns it to the USB keyboard device.

**Prototype**    `int set_event_bit(int handle, long flag);`

**Example:** The following examples cause bit 7 to be the keyboard event:

```
set_event_bit(kbdHdl, 0x80)
set_event_bit(kbdHdl, 128)
set_event_bit(kbdHdl, (1L<<7 ))
#define MY_KBD_EVENT 0x80
set_event_bit(kbdHdl, MY_KBD_EVENT)
```

### **Parameters**

**handle**       The keyboard device handle returned from `open()`.  
**flag**           The bit value of the event bit to use.

### **Return Values**

Success: 0

The OS reserves some event bits, which are not allowed to be selected by `set_event_bit()`. The reserved event bits include:

- `EVT_USER`
- `EVT_SHUTDOWN`
- `EVT_SYSTEM`

If the application calls `set_event_bit()` with one of these reserved event bits, the call returns -1 with `errno` set to `EINVAL`.

If the application calls `set_event_bit(kbdHdl, 0)`, the OS turns off events for the USB Keyboard device and returns 0 to indicate a successful response. This places the USB keyboard device in the default mode—does not generate events.

If the application calls `set_event_bit()` with a flag that defines more than 1 event bit, the OS returns -1 with `errno` set to `EINVAL` (e.g., `set_event_bit(kbdHdl, 0x90)` returns an error).

If the application calls `set_event_bit()` with the handle set to a non-USB keyboard device, it returns -1 and the event is not set.

**WARNING**

When the application calls `set_event_bit()`, the OS does not change the predefined event bits in `svc.h`; thus, if the application uses `set_event_bit()` to set `EVT_MAG` for the USB Keyboard device, the application receives `EVT_MAG` for both the mag card and the Metrologic barcode scanner device.

## **get\_event\_bit()**

Allows the application to find out what event will be generated by the USB keyboard device.



This is for use of optional future devices, and is not expected to be deployed in all, or majority of the terminals.

**Prototype**    `long get_event_bit(int handle);`

### **Parameters**

`handle`       The keyboard device handle returned from `open()`.

### **Return Values**

**Success:**   The event bit currently set in the Metrologic barcode device driver.

If 0 is returned, the keyboard device is in the default mode - no events are generated.

**Failure:**   -1 when handle is set to any device that does not support this API.

## **MC5727 USB Driver**

On Vx610 terminal, the MC5727 driver is a USB device driver similar to other Verix USB device drivers. Three USB devices are supported—COM2, COM9, and COM10.

- COM2 is the AT command and bulk data port.
- COM9 is used for CnS traffic.
- COM10 is used to deliver DM logs for debug or certification.

Each provides an interface comprised of write, read, control, status, close, and open function for each device according to its functionality.

The Vx610 terminal supports the following commands used for serial ports:

- `set_opn_blk()`
- `set_serial_lines()`
- `get_opn_blk()`
- `get_port_status()`
- `set_radio_ctl()`
- `get_radio_ctl()`

## **set\_radio\_ctl()**

Controls the settings of RAD\_MOD, RAD\_RST, and RAD\_OFF.

**Prototype**    `set_radio_ctl(char *sigs);`

Where:

- Bit 0 = RADMODE
- Bit 2 = RAD\_OFF

## get\_radio\_ctl()

Returns the status or RAD\_INT and RAD\_INT2.

**Prototype**    `set_radio_ctl(char *sigs);`

Where, Bit 0 = RAD\_INT.

<b>Data Interface (COM2)</b>	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 <code>read()</code> , <code>write()</code> , <code>open()</code> , <code>close()</code> functions are supported; <code>control()</code> and <code>status()</code> are handled over COM9.
<b>Control Interface (COM9)</b>	<p>The control interface is over USB Endpoint 4 in/out bulk endpoint. The USB messages on this endpoint are proprietary HIP messages carrying CnS (control and status messages) and DM logging messages—Verix V libraries implement the CnS messaging, while DM messages are primarily used for certification and debugging.</p> <p>Open and write to COM9 to send and receive CnS messages. A CnS message over COM9 turns DM logging on and off. A USB Pass-through application opens COM10 and delivers the log messages to a waiting application. The <code>read()</code>, <code>write()</code>, <code>open()</code>, <code>close()</code> functions are supported; <code>control()</code> and <code>status()</code> are not supported.</p>
<b>Debugging (COM10)</b>	<p>The MC5727 provides a logging facility that outputs activity logs carried in DM messages through USB Endpoint 4 (COM10). For Vx610, this is turned on and off through COM9. When activated, the logs wait on COM9 message stream and returned through COM10 to a USB Pass-through application, which opens two devices, COM10 and a USB UART.</p> <p>The MC5727 driver may need to parse messages received on Endpoint 4 from the modem outside ISR processing. This requires passing packets to a FIFO (1024 bytes) and processing them in a callout. Subsequently, CnS messages are queued to COM9 FIFO, DM logs queued to the COM10 FIFO.</p> <p>Since this delivers traffic poached from COM10 and that traffic does not require an acknowledgement or response, only <code>read()</code>, <code>open()</code>, <code>close()</code> will be supported on COM10. COM9 and COM10 are only valid devices if the MC5727 is detected. Other platforms return -1 with errno set to ENODEV.</p>
<b>Events</b>	Events are handled on COM9 and COM10 by user-defined events function. COM2 uses EVT_COM2. The following calls are in addition to the standard event API used in normal event handling.

## **set\_event\_bit ()**

---

Selects the event bit returned to the user on COM9 or COM10.

**Prototype**    int set\_event\_bit (int hdl, long flag);

**Parameters**    hdl      The return value from open().

                  flag     A long int where the event bit is set.

**Return Values**    Success: 0

## **get\_event\_bit()**

Selects the event bit returned to the user on COM9 or COM10.

**Prototype** long get\_event\_bit ( int hdl );

**Parameters** hdl The return value from open().

**Return Values** Success: Returns EVENT.

If the application calls `set_event_bit()` with one of the reserved event bits, `set_event_bit()` will return -1 with `errno = EINVAL`. The reserved bits are as follows:

- EVT\_USER
- EVT\_SHUTDOWN
- EVT\_SYSTEM

If an event bit is not chosen for COM9 or COM10 after opening, the OS uses EVT\_SYSTEM. This event is also used by other system functions like unzip and docking. It is recommended that the user choose an event bit when opening either COM9 or COM10.

**NOTE**

COM 10 is normally opened by the USB pass-through application.

The event must be defined with `set_event_bit()` each time the device is opened.

**WARNING**

The OS does 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.

**System Mode**

If available, standard information on each USB device currently attached to the terminal will be displayed, such as:

- Serial Number
- Vendor ID
- Release Number
- Product ID
- HUB
- Port
- Class
- Subclass
- Power
- Speed

**USBD Mode Selection**

This function facilitates modem firmware upgrade and update of PRI/PRL by manufacturing using tools running on the PC. System Mode provides a hidden key function to control the USBD Mode Selection through P56\_RAD\_MOD (default value OFF).

Pressing PF4 toggles the P56\_RAD\_MOD value. The resulting value is displayed on the screen for 30 seconds, then returns to the main DIAGS AND LOGS menu.

SYS MODE MENU 2> TERMINAL INFO F3> DIAG AND LOGS F3> PF4.

### USB Pass-through Application

This application is part of the System Mode Application Launcher. When installing and running the USB Pass-through application, the user must install `vxuart.inf` on the PC to connect through USB UART to the PC. The batch file used for download uses `ddl` from the PC and set all `CONFIG.SYS` variables.

**Example** \*USBCLIENT=RS232, \*MENU1=USB PASS THROUGH, \*MENU11=PASSTHRU.

The file is MAC-ready and can be run through the library interface. When the application is started, it ensures that the USB device lines are available on the COM1 interface. It opens the USB UART and COM10, and subsequently pass all messages directly between the end points.





## CHAPTER 10

### Smart Card API

This chapter describes the function calls of the smart card API and discusses PIN entry.

Smart card support for Verix® V 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® V 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® V Operating System application layer.

### Smart Card API Function Calls

#### **PCI PED Requirement**

The following EPP functions have been deleted from Verix V terminals in conformity to the PCI PED certification. If any of these functions are called, an error will be returned, with result set to -1 and `errno` set to `ENOSYS`.

- `decrypt_session_data()`
- `gen_master_key()`
- `gen_session_key()`
- `test_master_key()`

## **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\_SEDED 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.    |

## Administrative Services

This section describes smart card administrative services and their syntax. The following services are discussed:

- Retrieve IFD Capability
- Set IFD Capability
- Protocol Information and Negotiation
- ICC Power Management
- Mechanical Characteristics
- Communication Services
- ICC Insertion and Removal

### 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)

### Retrieve IFD Capability

```
RESPONSECODE           IFD_Get_Capabilities
(
    IN  DWORD          Tag
    OUT BYTE[ ]        Value
)
```

### Expected Behavior and Results

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 [Table 3.1 - Set/Get Capabilities Example, page 510](#))
- ICC interface state (see [Table 3.2 - Get Capabilities Example, page 509](#))
- Protocol parameters (see [Table 3.3 - Get Capabilities Example, page 510](#))

- Specific smart card API features (see page 504)

RESPONSECODE can be one of the following:

- IFD\_Success: Value successfully retrieved.
- IFD\_Error\_Tag: Tag does not exist
- IFD\_Error\_Not\_Supported: Tag not supported

## Set IFD Capability

```
RESPONSECODE           IFD_Set_Capabilities(
    IN DWORD          Tag
    IN BYTE[ ]        Value
)
```

### Expected Behavior and Results

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.

RESPONSECODE can be one of the following:

- IFD\_Success: Parameter successfully set.
- IFD\_Error\_Set\_Failure: Operation failed.
- IFD\_Error\_Tag: Tag does not exist.
- IFD\_Error\_Value\_Read\_Only: The value cannot be modified.
- IFD\_Error\_Not\_Supported: Tag not supported.

## Protocol Information and Negotiation

```
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
)
```

### Expected Behavior and Results

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.

RESPONSECODE can be one of the following:

- IFD\_Success: PTS succeeded.
- IFD\_Error\_PTS\_Failure: PTS failed.
- IFD\_Error\_Not\_supported: PTS not supported.
- IFD\_Protocol\_Not\_supported: Protocol not supported.

## ICC Power Management

```
RESPONSECODE      IFD_Power_ICC (
    IN WORD      ActionRequested
)
```

### Expected Behavior and Results

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).

RESPONSECODE can be one of the following:

- IFD\_Success
- IFD\_Error\_Power\_Action: The requested action could not be carried out.
- IFD\_Error\_Not\_supported: One of the requested actions is not supported.

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 [Table 3.2 - Get Capabilities Example](#) and [Table 3.3 - Get Capabilities Example](#) 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.

<b>Mechanical Characteristics</b>	Support for the following three function calls is optional.
<b>Expected Behavior and Results</b>	<p><b>Swallow the ICC</b></p> <p>RESPONSECODE                    IFD_Swallow_ICC ( )</p> <p>This function causes a mechanical swallow of the ICC, if the IFD supports this feature.</p> <p>RESPONSECODE can be one of the following:</p> <ul style="list-style-type: none"><li>• IFD_Success: Card successfully swallowed.</li><li>• IFD_Error_Swallow: Card not swallowed.</li><li>• IFD_Error_Not_supported: Function not supported.</li></ul>
<b>Expected Behavior and Results</b>	<p><b>Eject the ICC</b></p> <p>RESPONSECODE                    IFD_Eject_ICC( )</p> <p>This function causes a mechanical ejection of the ICC, if the IFD supports this feature.</p> <p>RESPONSECODE can be one of the following:</p> <ul style="list-style-type: none"><li>• IFD_Success: Card successfully ejected.</li><li>• IFD_Error_Eject: Card not ejected.</li><li>• IFD_Error_Not_supported: Function not supported.</li></ul>
<b>Expected Behavior and Results</b>	<p><b>Confiscate the ICC</b></p> <p>RESPONSECODE                    IFD_Confiscate_ICC()</p> <p>This function causes the IFD to confiscate the ICC, if the IFD supports this feature.</p>
<b>NOTE</b> 	Currently no Verix-based terminals support this feature. The function always returns IFD_Error_Not_supported.
	<p>RESPONSECODE can be one of the following:</p> <ul style="list-style-type: none"><li>• IFD_Success: Card successfully confiscated.</li><li>• IFD_Error_Confiscate: Card not confiscated.</li><li>• IFD_Error_Not_supported: Function not supported.</li></ul>

## 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.

### Data Exchange with the ICC

```
RESPONSECODE           IFD_Transmit_to_ICC (
    IN BYTE [ ]        CommandData
    OUT BYTE [ ]       ResponseData)
```

#### Expected Behavior and Results

This function instructs the smart card API to send to the ICC the command specified in the CommandData parameter and return the response of the ICC in the ResponseData parameter.

This function 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;
```

RESPONSECODE can be one of the following:

- IFD\_Success: The request was successfully sent to the ICC.

- `IFD_Communication_Error`: The request could not be sent to the ICC.
- `IFD_Response_TimeOut`: The IFD timed out waiting for the response from the ICC.
- `IFD_Error_BadFormat`: 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()`.

### ICC Present

RESPONSECODE

`IFD_Is_ICC_Present()`

**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.

### Expected Behavior and Results

Asynchronously signals insertion of an ICC into the interface device.

RESPONSECODE can be one of the following:

- `IFD_Success`: ICC present.
- `IFD_Failure`: ICC not present.

### ICC Removed

RESPONSECODE

`IFD_Is_ICC_Absent()`

### Expected Behavior and Results

Asynchronously signals removal of the ICC from the interface device.

RESPONSECODE can be one of the following:

- `IFD_Success`: ICC not present.
- `IFD_Failure`: ICC present.

**NOTE**



For information on Synchronous Card Communication Refer to *SC5000 CardSlot Library Programmers Guide*, VPN - 22564.

## Enumeration of the Tags

This section lists the tags.

### 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 [Table 3.1 - 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 48](#) can be set and read.

**Table 48** Codes for Enumerating Interface Device Capabilities

Data Element	Tag	MAX Length	Data Encoding
<b>Communications</b>			
Channel ID	0x0110	4 bytes	<p>Dword encoded as 0xDDDDCCCC, where:</p> <ul style="list-style-type: none"> <li>• <i>DDDD</i> = data channel type</li> <li>• <i>CCCC</i> = channel number</li> </ul> <p>The following encodings are defined for <i>DDDD</i>:</p> <ul style="list-style-type: none"> <li>• 0x01 serial I/O; <i>CCCC</i> is port number.</li> <li>• 0x02 parallel I/O; <i>CCCC</i> is port number.</li> <li>• 0x04 PS/2 keyboard port; <i>CCCC</i> is zero.</li> <li>• 0x08 SCSI; <i>CCCC</i> is SCSI ID number</li> <li>• 0x10 IDE; <i>CCCC</i> is device number.</li> <li>• 0x20 USB; <i>CCCC</i> is device number.</li> <li>• 0xFy vendor-defined interface, with <i>y</i> in the range 0-15; <i>CCCC</i> is vendor defined.</li> </ul>
<b>Mechanical Characteristics</b>			
Mechanical characteristics supported	0x0150	4 bytes	<p>Dword result of a bitwise OR operation performed on the following values:</p> <ul style="list-style-type: none"> <li>• 0x00000000 No special characteristics.</li> <li>• 0x00000001 Card swallow mechanism.</li> <li>• 0x00000002 Card eject mechanism.</li> <li>• 0x00000004 Card capture mechanism.</li> </ul> <p>All other values are RFU.</p>
<b>Protocol (see PC/SC Part 2 of this specification)</b>			
Asynchronous protocol types supported	0x0120	4 bytes	<p>Dword encoded as 0x0RRRPPPPP, where:</p> <ul style="list-style-type: none"> <li>• <i>RRR</i> is RFU and should be 0x000.</li> <li>• <i>PPPP</i> encodes the supported protocol types. A '1' in a given bit position indicates support for the associated ISO protocol.</li> </ul> <p>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.</p>

**Table 48 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 date 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 0x4RRRPPPPP where: <ul style="list-style-type: none"><li>• RRR is RFU and should be 0x000.</li><li>• PPPP encodes the supported protocol types. A '1' in a given bit position indicates support for the associated protocol.</li><li>• 0x0001 indicates support for 2-wire protocol.</li><li>• 0x0002 indicates support for 3-wire protocol.</li><li>• 0x0004 indicates support for I<sup>2</sup>C-Bus protocol.</li></ul> 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"><li>• 0x00000000: No devices.</li><li>• 0x00000001: RFU.</li><li>• 0x00000002: Numeric (that is, PIN) pad.</li><li>• 0x00000004: Keyboard.</li><li>• 0x00000008: Fingerprint scanner.</li><li>• 0x00000010: Retinal scanner.</li><li>• 0x00000020: Image scanner.</li><li>• 0x00000040: Voice print scanner.</li><li>• 0x00000080: Display device.</li><li>• 0x0000dd00: dd is vendor selected for a vendor-defined device.</li></ul>
-------------------------------------	--------	---------	---

**Table 48 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"> <li>• 0x00000000: No devices.</li> <li>• 0x00000001: RFU.</li> <li>• 0x00000002: Numeric (that is, PIN) pad.</li> <li>• 0x00000004: Keyboard.</li> <li>• 0x00000008: Fingerprint scanner.</li> <li>• 0x00000010: Retinal scanner.</li> <li>• 0x00000020: Image scanner.</li> <li>• 0x00000040: Voice print scanner.</li> <li>• 0x00000080: Display device.</li> <li>• 0x0000dd00: dd in the range 0x01-0x40 is vendor selected for a vendor-defined device.</li> <li>• 0x00008000: Indicates encrypted input supported.</li> </ul>
<b>Power Management</b>			
Power mgmt. supported	0x0131	4 bytes	<ul style="list-style-type: none"> <li>• If 0, device does not support power down while ICC inserted.</li> <li>• If non-zero, device supports power down while ICC inserted.</li> </ul>
<b>Vendor</b>			
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"> <li>• MM = major version.</li> <li>• mm = minor version.</li> <li>• bbbb = build number.</li> </ul>
IFD serial number	0x0103	32 bytes	ASCII string.
<b>Vendor Defined Features</b>			
Vendor defined features	May use values in range 0x0180-0x01F0	--	Refer to <a href="#">Specific Features for the Smart Card API</a> .

- ICC Interface Management** Tags listed in [Table 49](#) are set by the smart card API. These tags should not be set by the application. Tags listed in [Table 49](#) 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 49](#). The smart card API returns the state of the selected ICC.

**Table 49** Codes for Enumerating ICC State

Information	Tag	MAX Length	Responses (return as integer)
ICC present	0x0300	1 byte	<ul style="list-style-type: none"> <li>• 0 = not present.</li> <li>• 1 = card present but not swallowed (applies only if the IFD supports ICC swallow).</li> <li>• 2 = card present (and swallowed if the IFD supports ICC swallow).</li> <li>• 4 = card confiscated.</li> </ul>
ICC interface status	0x0301	1 byte	Boolean: <ul style="list-style-type: none"> <li>• 0 = contact inactive.</li> <li>• 1 = contact active.</li> </ul>
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"> <li>• 0 = unknown ICC type.</li> <li>• 1 = 7816 asynchronous.</li> <li>• 2 = 7816 synchronous.</li> <li>• Other values RFU.</li> </ul>
<b>New Features:</b>			
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 50](#) 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 `select-a-particular ICC` and issue the `IFD_Power_ICC()` command before attempting to read any value mentioned in this [Table 50](#). The smart card API returns the ICC state of the selected ICC.

Tags listed in [Table 50](#) are read-only, except tag 0x208. Only tag 0x208 can be set by the application.

**Table 50 Codes for Enumerating Interface Device Protocol Options**

Data Element	Tag	MAX Length	Read-Only	Comments
Current protocol type	0x0201	4 bytes	✓	Dword encoded in the same manner as available protocol types. It is illegal to specify more than one protocol in this value.
Current CLK	0x0202	4 bytes	✓	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	✓	F encoded as a little-endian integer (can be modified through PTS).
Current D (bit rate conversion factor)	0x0204	4 bytes	✓	D encoded as a little-endian integer (can be modified through PTS).
Current N (guard time factor)	0x0205	4 bytes	✓	N encoded as a little-endian integer (can be modified through PTS).
Current W (work waiting time)	0x0206	4 bytes	✓	W encoded as a little-endian integer. Only valid if current protocol is T=0.
Current IFSC (information field size card)	0x0207	4 bytes	✓	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	✓	BWT encoded as a little-endian integer. Only valid if current protocol is T=1.
Current CWT (character waiting time)	0x020A	4 bytes	✓	CWT encoded as a little-endian integer. Only valid if current protocol is T=1.
Current EBC encoding	0x020B	4 bytes	✓	EBC encoded as: <ul style="list-style-type: none"><li>• 0 = LRC</li><li>• 1 = CRC</li></ul> Only valid if current protocol is T=1.

## Specific Features for the Smart Card API

Table 51 describes specific features for the smart card API.

**Table 51 Codes for Enumerating Specific Features for the Smart Card API**

Data Element	Tag	MAX Length	Data Encoding
Open the smart card reader	0x0180	4 bytes	<p><b>Note:</b> These new features are not a part of the PC/SC standard, but are required for the smart card API.</p> <p>This tag opens the smart card reader. Before reading any tag value mentioned in <a href="#">Table 48</a>, 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"> <li>• 0x00000001 reader 1 is selected.</li> </ul> <p>This tag can only be set.</p> <p><b>Note:</b> 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><b>Note:</b> 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"> <li>• The least-significant word defines the selected ICC.</li> <li>• The most-significant word defines the standard.</li> </ul> <p>Valid Values:</p> <ul style="list-style-type: none"> <li>• CUSTOMER_SLOT</li> <li>• MERCHANT_SLOT_1</li> <li>• MERCHANT_SLOT_2</li> <li>• MERCHANT_SLOT_3</li> <li>• MERCHANT_SLOT_4</li> </ul> <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><b>Note:</b> 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 51 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"> <li>• CUSTOMER_SLOT</li> <li>• MERCHANT_SLOT_1</li> <li>• MERCHANT_SLOT_2</li> <li>• MERCHANT_SLOT_3</li> <li>• MERCHANT_SLOT_4</li> </ul> <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 <a href="#">Table 49</a> or <a href="#">Table 50</a>.</p> <p>Valid values:</p> <ul style="list-style-type: none"> <li>• CUSTOMER_SLOT</li> <li>• MERCHANT_SLOT_1</li> <li>• MERCHANT_SLOT_2</li> <li>• MERCHANT_SLOT_3</li> <li>• MERCHANT_SLOT_4</li> </ul> <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"> <li>• 0x00000000: Requests not to manage the NAD.</li> <li>• 0x0000XX01: Requests to manage the NAD. XX contains the SAD and DAD as described in ISO-7816.</li> </ul> <p>This tag can be set and read.</p> <p><b>Note:</b> 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"> <li>• If read value is zero, then the convention is direct.</li> <li>• If read value is non zero, then the convention is inverse.</li> </ul> <p>This tag is read-only.</p> <p><b>Note:</b> 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><b>Note:</b> Only valid for T=1 protocol.</p> <ul style="list-style-type: none"> <li>• 0x00000000: Requests not to manage the WTX.</li> <li>• 0x00000001: Requests to manage the WTX.</li> </ul> <p>This tag can only be set.</p>

**Table 51 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><b>IMPORTANT:</b> 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"> <li>• 0x00000001: ICC is a class A card (5V).</li> <li>• 0x00000002: ICC is a class B card (3V).</li> <li>• 0x00000003: ICC is a class AB card.</li> <li>• 0x00000004: ICC is a 1.8V card.</li> </ul> <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 <a href="#">Table 52</a> for ESC code values). Error code = 0x000000XX: where XX error code defined in errno.h.</p> <ul style="list-style-type: none"> <li>• 0x00000001: W time-out (T0).</li> <li>• 0x00000002: CWT time-out (T1).</li> <li>• 0x00000004: BWT time-out (T1).</li> </ul> <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"> <li>• 0x00000000: EMV 3.1.1.</li> <li>• 0x00000001: ISO7816-3.</li> </ul> <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"> <li>• 0x00000000: An asynchronous card is indicated. Default value.</li> <li>• 0x000000XX</li> </ul> <p>This tag can be read and set. See <a href="#">SC5000 CardSlot Library Programmers Guide</a>, 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 <a href="#">Test if ICC Present or Absent Example</a>.</p>
ICC structure size	0x01B2	4 bytes	This tag is related to the 0x01B0 tag. The application can request the size of the ICC structure before issuing the 0x01B0 tag.
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 51 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 52 displays the ESC code values.

**Table 52 ESC Code Values**

Escape Code	Value
NO_ERROR	0
<b><i>Generic Error Definitions</i></b>	
CARD_DEACTIVATED	0x01
CARD_MOVED	0x02
CARD_NOT_PRESENT	0x03
<b><i>Error Definitions for ATR</i></b>	
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 52 ESC Code Values**

<b>Escape Code</b>	<b>Value</b>
ATR_NOT_SUPPORTED	0x26
<b><i>Error Definitions for T=0 Protocol</i></b>	
TOO_SHORT_APDU	0x30
WRONG_APDU	0x31
WWT_EXCEEDED	0x32
INS_ERROR	0x33
T0_PARITY_ERROR	0x34
<b><i>Error Definitions for T=1 Protocol</i></b>	
CARD_MUTE_NOW	0x50
RESYNCHRONISED	0x51
CHAIN_ABORT	0x52
BAD_NAD	0x53
IFSD_NOT_ACCEPTED	0x54
<b><i>Error Definitions for PPS Negotiation</i></b>	
PARAM_ERROR	0x70
PPS_NOT_ACCEPTED	0x71
RESPONSE_ERROR	0x72
PCK_ERROR	0x73
PPS_PARITY_ERROR	0x74
<b><i>Hardware Errors</i></b>	
CARD_ERROR	0xE0
BAD_CLOCK_CARD	0xE1
UART_OVERFLOW	0xE2
SUPERVISOR_ACTIVATED	0xE3
TEMPERATURE_ALARM	0xE4
FRAMING_ERROR	0xE9
<b><i>Additional Errors not from EMV Library</i></b>	
TOO_MANY_CARDS_POWERED	0x109 (PIN pad only)

## Smart Card Code Examples

### Asynchronous Cards

This section provides code examples for both asynchronous and synchronous smart 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.

#### **Table 3.2 - Get Capabilities Example**

The linked example code file returns general information about the card or the reader.

**Table 3.3 - Get Capabilities Example**

The linked example code file returns general information about the card.

**New Tags Added  
for Multi-  
Application  
Support**

[Example Application 1](#)

[Example Application 2](#)

**Table 3.1 - Set/Get Capabilities Example****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.



## Communications

This chapter introduces the communications devices and protocols used on Verix V-based terminals and PIN pads, and discusses protocols used in most Verix V applications. The communication ports on these terminals must be opened, configured, used, and then closed. Configuration of a port uses the `Opn_Blk()` structure.

### The `Opn_Blk()` Structure

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 V-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()`.

### 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

##### Example

### Communication Ports

There are five communication ports in Verix V terminals. A sixth communication port is added in the Vx670 ([USB External Serial \(COM6\)](#)). 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 V-based terminals:

#### NOTE



On the Vx670 terminal, the [Modem Port \(COM3\)](#) is a USB device like the [USB External Serial \(COM6\)](#).

The Vx810 DUET and the Vx700 PIN pad both support [Modem Port \(COM3\)](#), while the Vx570 terminal also supports [USB External Serial \(COM6\)](#).

- RS-232 Serial Port (COM1)
- External PIN Pad Serial Port (COM2)
- Modem Port (COM3)

- Serial Printer Port (COM4)
- Internal PIN Pad Port (COM5)
- USB External Serial (COM6)

## RS-232 Serial Port (COM1 and COM2)

Verix V-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

### NOTE



On Vx670, COM1 is accessible via the Handy-Link connector and power barrel connector. To connect to other serial devices, COM6 is added on Vx670 and Vx810, which is a USB UART dongle but has the same functions as COM1. For more information, see [USB External Serial \(COM6\)](#).

## 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 V-based terminals is set by the `Fmt_auto` parameter in [`set\_opn\_blk\(\)`](#). 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 a feature of Verix V COM1, COM3, and COM6 drivers. 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.

**NOTE**

On V<sup>x</sup>670, V<sup>x</sup>810 DUET, and V<sup>x</sup>700 units that support USB to RS-232 devices, COM3 and COM6 are external USB devices (see [USB Dongles \(COM3 and COM6\)](#)). To use the modem or the USB UART, it must be plugged to the base of the terminal, or use the Handy-Link connector and power barrel connector.

## 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)

Devices that require cables are not the primary method of communication in the V<sup>x</sup>670 terminal. However, there are times when conventional communication devices are needed. 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. The V<sup>x</sup>670 only supports a single USB UART and a single USB modem. 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.

**NOTE**

The V<sup>x</sup>810 DUET and the V<sup>x</sup>700 PIN pad both support [Modem Port \(COM3\)](#), while the V<sup>x</sup>570 terminal also supports [USB External Serial \(COM6\)](#).

**NOTE**

On V<sup>x</sup>670 CR GPRS, the base does not support the simultaneous use of the multi-I/O and base connectors. The USB Host connector cannot connect to any USB devices when using a modified base with an exposed multi-I/O connector. While the unit is in the base, any USB devices must be connected through the base. The OS is unable to control the USB Host 5V power on the USB Host port on the multi-I/O connector, and it is not notified if a USB draws too much power.

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 `set_opn_blk()` 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 `open()` 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**

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 V<sup>x</sup>670 terminal sits in the base and is secured by gravity. There are no locks keeping the terminal in the base. This means that the terminal can be removed from the base without application control. This removal can be intentional or unintentional as a bump may unseat the terminal from the base enough to lose the USB connection.

**NOTE**

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`.

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 V-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		

The Vx610 COM1 port has two power related features. The RS-232 level driver chip can be powered on and off under software control, this affects TXD, RXD, CTS, RTS, and other signals. There is also a power pin for peripherals connected to COM1, such as smart RS-232 bar code readers. This can also be turned on and off under software control. The Vx610 does not support barcode readers.

The table below shows the available hardware support on various platforms.

**Table 53      Hardware Support on Various Platforms**

	Vx510	Vx570	Vx610	Vx670	Vx700	Vx810
COM1_PWR_ON	Supported	Not Supported	Supported	Supported	Supported	Supported
COM1_PWR_PIN_ON	Not Supported	Not Supported	Supported	Not Supported	Not Supported	Not Supported

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 `int set_com1_pwr(const char *sigs);` always returns 0, unless called with the handle for some other COM port where it returns -1 and `ERRNO` set to `EINVAL`.

**NOTE**

On V<sup>x</sup>670, 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). A power barrel connector is required to access this port.

COM6 supports RTS and CTS signals, but there are no physical wires for these signals between the dongle and the terminal. These signals are handled virtually via commands between the terminal communication driver and the USB UART bridge in the dongle.

The V<sup>x</sup>700 PIN pad does not support a power source on COM1, therefore, `set_com1_pwr()` always returns -1 with `errno` set to `EINVAL`. COM1 on the V<sup>x</sup>700 PIN pad requires an RJ48 10-pin cable (P/N 27716) uniquely configured for the V<sup>x</sup>700 PIN pad. The COM1 cables used in other Verix V units can not be used with the V<sup>x</sup>700 unit.

On V<sup>x</sup>510 GPRS, COM1 power pin is permanently on and is not controllable, while the RS-232 level translator chip is controllable similar to the V<sup>x</sup>610 terminal. The API `set_com1_pwr()`, which always returns zero, controls the RS-232 level translator chip but does not support power pin control.

**NOTE**

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.

## **External PIN Pad Serial Port (COM2)**

The External PIN Pad port supports, transmits, and receives signals and provide +9V power for external devices such as, the PINpad 1000SE or the SC5000.

The V<sup>x</sup>5xx and V<sup>x</sup>700 units support the External PIN Pad (EPP) on COM2, while the V<sup>x</sup>6xx terminal supports a radio modem (WAN) on COM2.

On V<sup>x</sup>510 GPRS, COM2 is used to communicate to the Siemens MC55i GPRS radio module. There is no external connection to the COM2 port.

**NOTE**

The V<sup>x</sup>810 and V<sup>x</sup>810 DUET are powered via COM2 but do not supply power.

The communication for wireless modules on V<sup>x</sup>610 and V<sup>x</sup>670 terminals is over the COM2 port. The serial port will supply the RTS, DTR, CTS, DCD, and DSR handshake signals to the radio, 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`.

The Vx670 OS also supports COM2 Bluetooth as well as 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. See [Bluetooth Modem Support](#) for more information on the Vx670 Bluetooth module.

Table 54 lists the COM2 devices on different Vx terminals.

**Table 54 COM2 Device on Vx Platforms**

Vx Platform	COM2 Device
Vx510	External Pinpad
Vx510 GRPS	Internal GPRS
Vx570	External Pinpad
Vx610	Internal GPRS/CDMA/WiFi
Vx670	Internal GPRS/CDMA/Bluetooth
Vx810	External Pinpad
Vx810 DUET	External Pinpad
Vx700	External Pinpad

**NOTE**



The external PIN pad port can be used as a general serial port. It supplies power to the PIN pad although it lacks CTS/RTS control lines.

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.

On Vx610, COM2 supports the MC5727 Sierra Wireless MiniCard radio modem in place of the Kyocera M200 CDMA radio solution. 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. `Read()`, `write()`, `open()`, `close()` are supported while control and status are handled over COM9.

The MC5727 board is connected through USB using the USB host port. Table 55 compares the modem connections of Sierra MC5727 modem and the previous Kyocera M200 modem.

**Table 55 Vx610 Modem Connections**

ARM9	Vx610 J6 PIN	I/O	Kyocera M200	Sierra MC5727	
			PIN Name	PIN Name	Note
GPE 4	3	I	nP56_DSR1	nP56_DSR1	No Use
GPE 5	4	I	RXD1	RXD1	No Use
GPE 4	5	O	TXD1	TXD1	No Use
GPE 3	6	I	nP56_DCD1	nP56_DCD1	No Use
GPE 2	9	I	nP56_CTS1	nP56_CTS1	No Use

**Table 55 Vx610 Modem Connections (continued)**

ARM9	Vx610 J6 PIN	I/O	Kyocera M200	Sierra MC5727
GPE 1	10	O	nP56_DTR1	nP56_DTR1
GPE 0	12	I	nP56_RTS1	nP56_RTS1
<b>GPB 0</b>	<b>14</b>	<b>O</b>	<b>MUX_CDMA</b>	<b>nP56_RAD_MOD</b>
<b>GPB 1</b>	<b>16</b>	<b>O</b>	<b>VEXT#</b>	<b>nP56_RAD_RST</b>
<b>GPB 2</b>	<b>17</b>	<b>O</b>	<b>XCVR_EN#</b>	<b>P56_RAD_OFF</b>
GPB 9	21		NO USE	NO USE
PGP 10	22	I	nP56_RAD_HPWR	nP56_RAD_HPWR
GPF 1	25	I	XCVT_DET	nP56_RAD_INT
DP 1	29		pUSB_DEVICE	pUSB_DEVICE
DP 0	30		pUSB_HOST	pUSB_HOST
DN 1	31		nUSB_DEVICE	nUSB_DEVICE
DN 0	32		nUSB_DEVICE	nUSB_DEVICE
IIC_SCL	39		IIC_SCL	IIC_SCL
IIC_SDA	41		IIC_SDA	IIC_SDA
GPF2	45	I	P56_RAD_INT2	P56_RAD_INT2
				No Use

- GPB0 / P56\_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 System Mode 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.
- GPB2 / P56\_RAD\_OFF pin is used by both M200 and MC5727. The MC5727 uses the same OS startup values as the M200 and previous devices on the Vx610 have used. It can be controlled through `set_radio_ctl()` as it is in current Verix V products.
- GPF1 / nP56\_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()`.

Devices on COM2, such as the Vx610 radio module and the Vx670 Bluetooth, 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 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 Vx610. 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.

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.

On Vx670, COM2 supports Siemens GPRS radio and uses the Conexant CX3889 USB WiFi chip. Calling the function [SVC\\_INFO\\_MODULE\\_ID\(\)](#) will return the following values:

**Table 56      SVC\_INFO\_MODULE\_ID() Values**

SVC_INFO_MODULE_ID(2)	Radio Device on COM2
8	Siemens MC56 US
9	Siemens MC55 International
10	Sierra Wireless EM3420

**NOTE**

On Vx700 units, COM2 and COM8 require an 8-pin RJ-45 connector. COM8 is used for the Port 3 serial device and COM3 is used for the USB modem device.

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.

The Vx610 battery conforms to the maximum charging requirements. If the battery charges continuously for more than 6.5 hours, the terminal stops charging.

## **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);
```

**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.

## Modem Port (COM3)

Verix V-based terminals use three different modems. The Conexant CX81802 (Banshee) modem is employed in V<sup>x</sup>510/V<sup>x</sup>610 terminals; the Conexant CX86501 (Eisenhower) modem is used in V<sup>x</sup>570 and V5 units; and the Silicon Laboratory Modem Si24xx is used in V<sup>x</sup>670, V<sup>x</sup>810 DUET, and the V<sup>x</sup>700 units.

---

**NOTE**

The Eisenhower modem is supported in the V<sup>x</sup>510 and V<sup>x</sup>610 terminals in addition to the currently used Conexant CX81802-V33 (Banshee) modem.

The V<sup>x</sup>810 PIN pad does not support either an internal or external modem. All modem related APIs in the V<sup>x</sup>810 PIN pad return ENODEV. Modem related CONFIG.SYS parameters may be set but with no effect. Application programs invoking a download must use the `download()` API since the `SVC_ZONTALK()` API requires a modem to function. The application may invoke `download()` to run over any available port, depending on cable configuration, including the USB client port.

---

**NOTE**

On V<sup>x</sup>510 GPRS, the Banshee 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 Conexant modems (CX81802 and CX86501) 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 (i.e. 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 *Silicone 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.

## Conexant Modems (Banshee and Eisenhower)

The CX14.4 modem is an external modem contained within the V<sup>x</sup>5xx/ V<sup>x</sup>610 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
- 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)

V<sup>x</sup>670 and V<sup>x</sup>700 units do not have an internal modem. Instead, a modem dongle consisting of a USB UART chip and a Silicon Labs modem chip in a single unit is available. To use the modem, the unit must be plugged into the base. On the V<sup>x</sup>810 DUET, the modem is located internally in the DUET base where it cannot be removed, but it still uses the same USB modem design and acts just like the V<sup>x</sup>670 modem.



Another option to access the modem is by using the Handy-Link with the power barrel connector.

The modem dongle will be Verix device “/DEV/COM3”, with an the event bit of EVT\_COM3. Only one USB modem dongle may be connected to the base of V<sup>x</sup>670 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 )` will return -1, and set `errno` to EBADF. If the modem dongle is removed after it is opened, API functions such as [read\(\)](#), [write\(\)](#), and [set\\_opn\\_blk\(\)](#) return -1 and set `errno` 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 V 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 V OS 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 Verix. See [Synchronous Example-Conexant](#) and [Synchronous Example-SiLabs](#) for differences between AT init strings.

**SDLC Protocol**

Most Verix V-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.

<b>Supervisory Packet Format</b>	Verix V-based terminals can receive all supervisory format frames. Only <i>ready-to-receive</i> (RR) frames are processed and only RR frames are generated.
<b>Modulo 8 Information Packet Format</b>	Verix V-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.

<b>Unnumbered Information Packet Format</b>  <b>Communication Sequence</b>	<p>Verix V-based terminals can receive all unnumbered information frame formats.</p> <ul style="list-style-type: none"> <li>The <i>set normal response mode</i> (SNRM) frame is processed and acknowledged with the <i>unnumbered acknowledgment</i> (UA) frame.</li> <li>The <i>disconnect</i> (DISC) message is acknowledged with the UA frame, but is not otherwise processed.</li> <li>All other unnumbered information frames are ignored.</li> </ul> <p>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.</p>
<b>NOTE</b>	 The Verix V OS supports SDLC/V.80 protocol on COM2 and COM3. See <a href="#">Bluetooth Modem Application Flow</a> 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

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_Blk` 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, RTS, and so on), 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 V.

### 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 and Verix V implementations of SDLC. In the old TXO environment, this byte contains the number of data packets in the reject queue. Since Verix and Verix V 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.

### 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.

**NOTE**



On Vx670, the OS loads the country profile whenever the USB modem is plugged in.

Verifying the installation of a country profile on a terminal is simple:

- Check the version in the SYSTEM INFORMATION screen selected in SYSTEM MODE MENU 3 (SYSTEM MODE MENU 2 on V<sup>x</sup>670).
- Select the SYSTEM INFO option and use the down arrow at the bottom of the display to page down. The following screen displays:

Figure 20 illustrates the sample system information screen on V<sup>x</sup>5xx/V<sup>x</sup>6xx.

SYSTEM INFORMATION	
MDM TYPE	4
VER	B3050500B5000100
MODEM CTRY	B5
I1	042
I3	CS81802-V32

Figure 20 Sample System Information Screen on V<sup>x</sup>5xx/V<sup>x</sup>610



The V<sup>x</sup>570 terminal uses the Eisenhower modem. The VER and I<sub>3</sub> fields on the SYSTEM INFORMATION screen will have a different information.

To view the modem information for the V<sup>x</sup>670 and V<sup>x</sup>700 units, the USB modem must be connected to the terminal.

On V<sup>x</sup>670 terminal, the System Information screen appears as illustrated below:

SYSTEM INFORMATION	
MDM TYPE	50
MODEM CTRY	+GCI:B5
VER	US
	B5V002-02
PATCH VER	C:bcd8
MDM FW VERSION	71
MODEM MODEL	2434
	↑ ↓

Figure 21 System Information Screen on V<sup>x</sup>670

For more information on the System Information screen on V<sup>x</sup>670, refer to [System Mode](#).

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 Vx670 VER is composed of a six-character country mnemonic, the country code, and the version with major and minor parts.

The country profile can be loaded using the modem configuration utility (P/N S6100-VVCPIAIMG). 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.

### CAUTION



In older OS versions, loading the wrong profiles in a terminal causes the modem not to function. When this happens, the terminal must be loaded on site with the correct profiles via a direct download; otherwise, the terminal must be brought to a repair facility.

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 57](#) lists the \*MERR error code values.

**Table 57 \*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

**Table 57 \*MERR Values** (continued)

*MERR Values	Display	Description
7	MODEM COMM ERROR	Modem Communication error such as: • 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, and the Eisenhower has ESN\_nnnn.zip. The V<sup>X</sup>670, V<sup>X</sup>810, and V<sup>X</sup>700 units use the Silicon Laboratories Si24xx chip set with a filename format of SL\_nnnn.zip, which is also added.

**Notes**

- 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 system mode, 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 CX14.4 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 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 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 *CX14.4 High Speed Command Reference Guide* and the *Silicon Laboratories AN93*, for more information.

## Combined Modem Profiles

On V<sup>x</sup>510, both Banshee and Eisenhower modems can be downloaded to the same terminal, allowing the customer to manage a single file containing profiles for both modems.

A CONFIG.SYS variable, \*MD, is added containing both the Banshee and Eisenhower profiles and authentication files. When the OS recognizes the \*MD variable, the System Mode determines the modem type, unzips the combined files, and then deletes the unused profile. The appropriate profile is retained in the GID 1 file system. The CONFIG.SYS variable, \*MN, is set to the profile name and an SVC\_RESTART is performed. The restart allows the OS to process the new profile with the existing system.

## 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 V for the modem only.

## 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\(\)](#).

**NOTE**

On V<sup>x</sup>810, this always returns a value of 50. This is true whether the it is operating as stand-alone device or is connected to a DUET. This is similar to how the V<sup>x</sup>670 terminal operates.

**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 RAM (I:) and flash (F:) are deleted, and the flash is coalesced. Only Group 1 tasks are allowed to do this.
- “r” like R except that flash is not coalesced.

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 SVC\_ZONTALK. 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. 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"><li>• There must be at least one colon.</li><li>• There must be at least one character before the first colon.</li><li>• There must be at least one digit after the first colon.</li></ul>
*ZS	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 2400.
*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 Can't 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 System Mode, and reinitiate the download.

## Serial Printer Port (COM4)

### C Code Applet for COM4 Driver

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.

The [Printer Functions](#) section presents function call descriptions for calls only used in the Verix V printer device 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

On the Vx810 DUET, a USB thermal printer is used. The DUET base station contains a micro-controller (MCU) connected to a thermal print mechanism. The OS communicates with the printer MCU through the USB UART, which is functionally the same as the USB UART dongle on the Vx670 terminal. The Vx670 USB UART driver attaches the USB UART PID to the printer device (COM4) for the Vx810 PIN pad. A new USB device definition, UDB\_COM4, is set in the USB device status.

---

#### 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 V printer feature set and controls the DUET printer mechanism optimally.

On Vx670 CR GPRS, the printer speed is not reduced while the Vx670 is in the base. The USB power management always subtracts 200 mA from the available printer power. The OS takes this safe option because it is unable to determine if a plugged device supplies the appropriate power.

On Vx510 GPRS, the thermal printer in the Vx510 GPRS functions the same as the existing printer, but uses the SII printer mechanism HPML20803-05-R (MPN:LTPA245S-384-E).

---

#### NOTE



The \*DARK parameter is modified for quality printing. Printing on Chinese Telcom 2-ply paper is not supported.

---

## Printer Functions

This section presents descriptions of function calls used for the Verix V printer device driver.

## get\_fc\_config()

Retrieves current hardware flow control configuration. The buffer contents returned correspond to that described in [set\\_fc\\_config\(\)](#).

**Prototype**    `int get_fc_config(int fd, char *buffer);`

### **Return Values**

Success: 0

Failure: -1, `errno` set to EBADF, `fd` is not a valid open file descriptor.

-1, `errno` set to EACCES, access violation attempting to write to `buffer`.

## **set\_fc\_config()**

This function is provided for performance tuning for high-speed communications, to select/deselect hardware flow control, and select between the printer and auxiliary ports. Under normal operation, the defaults should be more than adequate.

---

**NOTE**

Poor configuration settings can adversely affect communication performance.

---

**Prototype**

```
int set_fc_config(int fd, char *buffer);
```

Sets the hardware flow control configuration as follows:

Buffer, Byte 0: Bit 0: DTR for no receive choke.

Bit 1: RTS for no receive choke.

Bit 2: DTR for receive choke.

Bit 3: RTS for receive choke.

Bit 4: CTS for transmit choke.

Bit 5: unused.

Bit 6: hardware flow control enabled.

Bit 7: unused.

Byte 1: Bits 0–7: receive choke trigger level.

Byte 0, bits 1:0 indicate the state that the RTS and DTR output lines are set to when hardware flow control is enabled and the driver is ready to receive data (1 = active, 0 = inactive).

Byte 0, bits 3:2 indicate the states that the RTS and DTR output lines are set to when hardware flow control is enabled and the driver is not ready to receive data (1 = active, 0 = inactive).

Byte 0, bit 4 indicates if the CTS input line must be set for the driver to transmit data when hardware flow control is enabled (that is, if bit 4 is 0, the transmitter is always active; if bit 4 is 1 and CTS goes low, transmission is suspended).

Byte 0, bit 6 indicates whether hardware flow control is enabled (1 = enabled, 0 = disabled).

Note that if the transmit FIFO depth is greater than 1, there may be characters already in the transmit FIFO that are transmitted even after the CTS line goes low. Ensure that the device connected to the communication port is able to buffer the same number of characters equal to the transmit FIFO depth.

If hardware flow control is disabled, byte 1, bits 5:0 are ignored and RTS and DTR can be manually controlled by calling `set_serial_lines()`. Conversely, if hardware flow control is enabled, attempts to manually control the state of the RTS and DTR output lines with `set_serial_lines()` are ignored.

Byte 1 indicates how close to full the operating system's receive FIFO (not the hardware FIFO on the UART device) must be before receive flow control is triggered. The operating system's FIFO is 256 bytes deep. A receive choke condition is triggered when the FIFO is full to within `<rx choke trigger level>` of the top.



**NOTE**  
Setting byte 1 to a value that is too high wastes buffer space, slows down communications, and could result in overrun errors.

The DTR output of the UART controls the MUXing of the serial printer port between the internal printer port and the external auxiliary port. If the DTR bit is set to 1, the internal printer port is selected; if the DTR bit is set to 0, the external printer port is selected. It is *not* recommended to use the DTR output for flow-control purposes. To select the internal printer port, the DTR bit should be set to 1 for both the no receive choke and the receive choke conditions. To select the external printer port, the DTR bit should be set to 0 for both the no receive choke and receive choke conditions. By default, the internal printer port is selected.

The default hardware flow control configuration is as follows:

Byte 0:

Bit	Interpretation	Default
0	DTR for no receive choke	1
1	RTS for no receive choke	1
2	DTR for receive choke	1
3	RTS for receive choke	1
4	CTS for transmit choke	0
6	hardware flow control	0

Byte 1: receive choke trigger level: 36

To avoid loss of data, `set_fc_config()` can only be called after calling `open()`, but before calling `set_opn_blk()`. Calling `set_fc_config()` at any other time has no effect on FIFO status.

**Return Values**

Success: 0

Failure: -1 `errno` set to `EBADF`, `fd` is not a valid open file descriptor.

-1 `errno` set to `EINVAL`, `set_opn_blk( )` called.

-1 `errno` set to `EACCES`, access violation attempting to read from buffer.

**NOTE**

Results returned when sending this function to the IPP port read 0 (success), however, results are not viable as this port has no flow control (that is, Tx and Rx are the only lines available on this port).

## Internal PIN Pad Port (COM5)

The Verix V internal PIN pad (VVIPP) is a software-only emulation of the IPP7 chip used in Verix 68K platforms such as the Omni 3600 and Omni 3750. IPP7 is also used in non-Verix 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 ARM document for more details.

COM5 is the communication port assigned to the internal PIN pad (IPP). In the Omni 3xxx platform, the IPP7 chip connects via UART to the main CPU. In Predator, the IPP7 chip and UART are emulated in software.

Predator IPP is backwards compatible with IPP7. Existing Verix 3xxx applications that use IPP7 works with Predator IPP. The only significant difference is Predator IPP can hold up to 10 triple length keys. The IPP appears as COM5 and the EPP (second external RS-232) port appears as COM2. The IPP is a device that supports state-of-the-art measures to provide security for PIN-based transactions.

### NOTE



The IPP7 functionality is emulated in software, there is no hardware.

## Files Used to Compile Driver

Below are the files currently used to compile this driver:

### **ipp.c**

Main module that includes all other source files directly or indirectly. Non-volatile and transient data, `open()`, `close()`, `read()`, `write()`, `control()`, `status()`, and `mod_init()` are defined here.

### **ippemu.c and ippemu.h**

Packet decoding and processing are implemented in these files.

### **asciectl.h**

This file defines names for ASCII control characters such as STX and ETX, among others.

### **getpin.c and getpin.h**

These files implement keyboard PIN entry functions. These functions are exported for the security module via `InstallSysVec()`.

### **giske.c and giske.h**

GISKE (Global Interoperable Secure Key Exchange) functions in here. GISKE defines a format for secret keys. It supports single, double, and triple length DES keys with key attributes. GISKE keys are encrypted and MAC'd.

**Clear IPP Keys  
Upon Certificate  
Tree Removal**

When Verix V file authentication detects a corrupt file authentication certificate tree, 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.

## open()

Opens the IPP. On success, the IPP handle is returned and this handle can be used for read(), write(), and close(). The pointer deviceName must point to the string “/dev/com5”.

### Prototype

```
int open(const char *deviceName, int unused);
```

### Return Values

- Success: IPP handle is returned and this handle can be used for read(), write(), and close(). The pointer deviceName must point to string “/dev/com5”.
- Failure: If the IPP is already owned by another task, this fails and returns -1, errno is set to ENODEV.

## read()

Transfers data from the card reader scan into the buffer.

**Prototype**    `int read(int handle, char *buffer, int size);`

**Parameters**

handle	The value returned from open().
buffer	A pointer to the data area.
size	The maximum number of bytes to read.

**Return Values**

Success:	>0 indicates the number of bytes returned in the buffer. A return value of zero (0) means there is no data.
Failure:	A return value of -1 means an error has occurred and errno has been set. If errno is equal to EBADF, the handle is not valid. If errno is equal to EINVAL, set_opn_blk() has not been called or the buffer pointer is invalid.

## write()

Transfers a single complete IPP packet or a single character from the buffer into the IPP. Incomplete, incorrectly framed packets, overly large, or multiple packets in a single write are rejected. The valid start-of-packet characters are STX and SI. The valid end-of-packet characters are ETX and SO. The only single characters accepted are ACK, NAK, and EOT.

**Prototype** `int write(int handle, char *buffer, int size);`

### Parameters

handle	The value returned from open().
buffer	A pointer to the data area.
size	The number of bytes to write.

### Return Values

Success:	The actual number of bytes written. If the return value is equal to size, the packet was transferred to the IPP.
Failure:	If the return value is to -1 errno is set. If errno is equal to EBADF the handle is not valid.  If errno is equal to EINVAL, this means set_opn_blk() has not been called, the buffer is too large to be a valid IPP packet, the buffer pointer is not valid, the single character was not one of [ACK, NAK, EOT], the packet has a bad LRC, or the packet is not framed correctly.

## **close()**

---

Releases ownership of the IPP. All unread data is lost.

**Prototype**    `int write(int handle);`

**Parameters**    `handle`     The value returned from `open()`.

**Return Values**

Success:    0 if `close` is successful.

Failure:    -1 and sets `errno` if not. If `errno` is equal to `EBADF`, the handle is invalid.

## **set\_opn\_blk()**

Initializes/reinitializes the communication parameters.

**Prototype**    `int set_opn_blk(int handle, const struct Opn_Blk*ob)`

**Parameters**

handle      The value returned from open().

Ob           A pointer to an Opn\_Blk structure.

**Return Values**

Success:    0 if successful.

Failure:    -1 if error is detected. Errno is set to EBADF if the handle is not valid. Errno is set to EACCES if the pointer is not valid.

There is no UART hardware, this call stores the contents of ob in transient memory. Calling `get_opn_blk()` returns the stored values. No checking of any kind is done on the values.

## get\_opn\_blk()

Copies the current `opn_blk()` structure into the caller's buffer.

**Prototype**    `int get_opn_blk(int handle, struct *Opn_Blk ob)`

**Parameters**

handle      The value returned from `open()`.

Ob           A pointer to an `Opn_Blk` structure.

**Return Values**    If the `opn_blk` structure is not previously set, `get_opn_blk()` returns zero and the caller's OB structure is set to zero.

## **get\_port\_status()**

Copies the status information to the caller's 4-byte buffer:

- 1st byte Number of input messages pending.
- 2nd byte Number of failed output messages pending (always 0).
- 3rd byte Number of output slots (always 1).
- 4th byte Current signal information. Always 0 because there is no UART.

**Prototype**    `int get_port_status(int handle, char *buf4);`

**Parameters**

- handle      The value returned from `open()`.
- buf4        A pointer to a four-character buffer.

**Return Values**    If `opn_blk` structure is not previously set, `get_port_status( )` returns zero.

## **reset\_port\_error()**

---

Has no effect and the corresponding error indicators are always 0. In general for Verix V communication ports, *reset\_port\_error()* resets the error indicators for parity, framing, and overrun errors, and the break indicator.

**Prototype**    `int reset_port_error(int handle);`

**Parameters**

handle      The value returned from *open()*.

buf4        A pointer to a four-character buffer.

**Return Values**

Success:    Always returns zero (0).

Failure:    -1 and sets *errno* to EBADF if the handle is not valid.

## **set\_serial\_lines()**

---

Uses the parameters in buffer to set/reset DTR, RTS, and BRK.

**Prototype**    `int set_serial_lines(int handle, const char *mask);`

**Parameters**

`handle`    The value returned by `open()`. The mask is ignored.

**Return Values**

Success:    Always returns zero (0) as there is no UART hardware.

Failure:    -1 and sets `errno` to `EBADF` if the handle is not valid.

## **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**    For compatibility with Omni 3xxx, IPP\_power(1) returns 0.  
IPP\_power(0) return -1 and set errno to EINVAL.

## **set\_fifo\_config()**

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.

### **Prototype**

```
int set_fifo_config(int handle, const char *settings);
```

### **Parameters**

handle      The value returned by `open()`.

setting      Points to two bytes.

### **Return Values**

Success: 0

Failure: -1 if an error is detected. Sets `errno` to `EBADF` if the handle is not valid.  
`Errno` is set to `EACCES` if the point is not valid.

## **get\_fifo\_config()**

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.

**Prototype**    `int get_fifo_config(int handle, char *settings);`

**Parameters**

handle     The value returned by `open()`.  
setting    Points to two bytes.

**Return Values**

Success:   0  
Failure:   -1 if an error is detected. Sets `errno` to `EBADF` if the handle is not valid.  
              `Errno` is set to `EACCES` if the point is not valid.

## TerminatePinEntry()

Ends the PIN entry session. `set_opn_blk()` must be called before calling `TerminatePinEntry()`.

**Prototype** `int get_fifo_config(int handle, char *settings);`

**Parameters**

`handle` The value returned by `open()`.

`setting` Points to two bytes.

**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.

## 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: Non-zero, with the only failure condition an invalid buffer pointer. errno is unchanged.

## USB External Serial (COM6)

Apart from the regular UART port accessible via the Handy-Link connector, the Vx670 terminal allows connection to other serial devices via the USB UART dongle. The Vx570 terminal also communicates with COM6 to support USB to RS-232 converter device.

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.

The `EVT_USB` bit is set in all running tasks' event masks whenever the OS detects a USB device connection or removal (e.g., 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 V terminal is in the DOWNLOAD NEEDED state, the System Mode polls several devices in a round robin fashion looking for ENQ from a download host. COM6 for the Verix V 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 V terminal at the DOWNLOAD NEEDED prompt without entering System Mode.

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 System Mode 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.

## USB Secure Mode

A multi-application-compatible mechanism for application software is supported to “secure” its communications with a USB device. Securing communications means:

- Powering off all external USB ports (for internal USB device such as V<sup>x</sup>670 WiFi).
- Powering off all external USB ports except the one being secured (for external USB device).

### NOTE



When concurrent use of USB devices is required, security is not possible.

An internal USB device physically located inside the V<sup>x</sup>670 casing (e.g., WiFi) is considered secure, and is never powered off.

By default, the OS always powers on all USB ports, allowing backward compatibility with previous OSes in terms of enumeration and device initialization.

USB-based devices can be configured in two modes—single USB device mode and multi-USB device mode. USB modes are specified through `open()` and `set_usb_multi_device()` APIs. When in single device mode, all other external USB ports are powered off; when in multi-USB device mode, simultaneous use of USB devices is allowed.

The USB device mode is not affected by docking/undocking (or Multiport plug in/out). Once single USB device mode is turned on, other ports remain powered off until the single USB device mode is turned off (and the application prompts that simultaneous USB device usage is ok).

When a device is in single USB device mode and the device is closed, all external ports are powered back on.

## open()

Opens the USB device.

**Prototype** int open(const char \*id, int flags);

**Parameters**

id	Device name (e.g. DEV_COM3).
flags	Flags are used for files and not devices. Determines if the device is to be opened in either single USB device or multi-USB device mode. O_SINGLE_DEV: the device is to be opened in single-USB-device mode. This has no effect if used on a non-USB device. O_MULTI_DEV: also defined (as 0x0) and can be used to open USB devices in multi-USB-device mode.

**Return Values**

Success	Device opened.  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.
Failure	Returns -1 with errno set to: <ul style="list-style-type: none"><li>• EBUSY – device already opened by another application.</li><li>• EBADF – device not available, device is physically not connected or the external USB ports are powered off. Call get_usb_device_bits() to investigate.</li><li>• EAGAIN – if single-USB-device mode is requested but one or more external USB devices are already open (trying to open "/dev/wln1" in single-USB-device mode while an external USB device is already open).</li></ul>

**NOTE**



If a device is currently enumerated/initialized, open( ) is blocked until initialization is complete.

## **close()**

---

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

**Prototype**    `int close(int hdl);`

## **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**

```
set_usb_multi_device(int hdl, int onOff);
```

### **Parameters**

hdl      Device name for an already opened USB device.

onOff     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 (e.g., calling this function on a Vx570 with COM3 handle will fail with `errno=EINVAL`).

## get\_usb\_device\_bits()

Gets the status of all supported USB devices.

**Prototype**

```
0unsigned long get_usb_device_bits (void);

#define UDB_FLASH 1<<0
#define UDB_ETHER 1<<1
#define UDB_WIFI 1<<2
#define UDB_COM3 1<<3
#define UDB_COM6 1<<4
#define UDB_KBYD 1<<7
#define UDB_IBHUB 1<<4
```

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 serial
UDB_KYBD	7	USB keyboard
UDB_IBHUB	4	DUET USB hub

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 will be set. All other device bits will be 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 will be set (assuming WiFi is populated in the terminal) and the single-USB-device mode bit will be set.
- If single-USB-device mode is not set for any device, the bits will be set for all devices that are enumerated and initialized. The single-USB-device mode bit will be cleared.

## USB External Ethernet (ETH1)

On Vx570, the ASIX AX88772 USB Ethernet is a permanently attached internal device. On Vx670 and Vx670-based units, the USB Ethernet is an external device. No more than one USB Ethernet device may be connected at any time. In the instance that more than one device is connected, only the first device connected is recognized.

The Vx700 PIN pad supports Ethernet over USB driver, similar to the current Vx570 unit with some changes in the connection. Power is added to the connection to eliminate external power source.

For more information, refer to [USB Ethernet \(ETH1\)](#) section in [Appendix i](#).

## USB Internal WiFi (WLN1)

The Conexant CX3889 (Partagas) WiFi chip is used as the WiFi device for Vx670 terminal. This allows applications and libraries to read and write Ethernet packets, and convert the Ethernet packets to/from 802.11 wireless LAN packets then send/receive them over the radio.

For more information, refer to [USB Internal WiFi \(WLN1\)](#) section in [Appendix i](#).

## 9-Bit Serial Interface

The Vx700 PIN pad supports the Multi-Drop Bus (MDB) protocol. 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.

## 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 V-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()`.

The Vx700 PIN pad adds one more data format 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 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.

**Prototype**    `int write_9bit(int com, short *buffer, int count);`

### Parameters

com	Handle of the desired COM port.
buffer	Pointer to a short array.
count	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.

**Prototype**    `int read_9bit(int com, short *buffer, int count);`

### Parameters

com	Handle of the desired COM port.
buffer	Pointer to a short array.
count	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.

## Bluetooth Modem Support

The Vx670 OS supports the Bluetooth I/O module and the Bluetooth base station with Silicon Laboratories Si2434 modem chip set.

---

### NOTE



The use of synchronous communication with other base stations is not supported.

---

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 V OS 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 System Mode 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 V<sup>x</sup>670 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 V OS 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 V 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 V<sup>x</sup>670 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.

**General Communication Device Functions**

This section describes APIs that are used by all of the Verix V on-board communication devices.

## close()

Disables the device.



**NOTE** The application is responsible for ensuring that all data written to the port was transmitted before issuing `close()`.

Issuing `close()` without checking `get_port_status()` or calling `SVC_WAIT()` may cause loss of transmitted characters.

**Prototype** `int close(int comm_handle);`

### Parameters

`comm_handle` Value returned from `open()`.

### Return Values

Success: 0

Failure: -1 and `errno` set to `EBADF`: `comm_handle` is not a valid open file descriptor.

## 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 handle, void *parms);`

### Parameters

`handle`    Open serial port.

`parms`    Points to a one-character string which is the same as the `type` parameter passed to `SVC_ZONTALK()`, 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: `write()` to file failed.  
              -7: Timed out waiting for host.  
              -8: Lost carrier from modem.  
              -9: Sent too many NAKs.  
              -10: `set_timer()` failed.  
              -11: `write()` to COMx failed.  
              -13 and `errno` set to -EACCES: 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 system mode, and reinitiate the download.

## get\_component\_vars()

Returns the non-volatile data for this communication port.

**Prototype**    int get\_component\_vars(int handle, char \*buffer, int len);

**Parameters**

len	If len is less than the full component data size, the data are truncated. len must be large enough to handle the filename and date (18 bytes), else EINVAL returns.
-----	---

Data format returned is:

char filename[12];	the filename of the component 0 — padded
char filedate[6];	BCD date and time of filename file: ymdhms
char data[variable];	component-specific data as defined below:

Open Block (OpnBlk):  
 unsigned char rate;  
 unsigned char format;  
 unsigned char protocol;  
 unsigned char parameter;

Line status (SigStat):

Current signal information returned in byte 4:

- 0x80: set if break/abort detected
- 0x40: always 0
- 0x20: set if CTS detected
- 0x10: set if ring indicator present
- 0x08: set if DCD present
- 0x04: set if frame error detected
- 0x02: set if overrun error detected
- 0x01: set if parity error detected

### Return Values

Success: 0

Failure: -1 and errno set to EBADF: comm\_handle is not a valid open file descriptor.

-1 and errno set to EINVAL: No set\_opn\_blk() command issued or the caller's buffer is an invalid size.

-1 and errno set to ENXIO: USB device not present.

## get\_fifo\_config()

Retrieves current FIFO configuration. The returned buffer contents correspond to what is described [set\\_fifo\\_config\(\)](#).

**Prototype**    `int get_fifo_config(int fd, char *buffer);`

### **Return Values**

Success: 0

Failure: -1 and `errno` set to EBADF: `fd` is not a valid open file descriptor.

-1 and `errno` set to ENXIO: USB device not present.

## get\_opn\_blk()

Copies the current `Opn_Blk` structure into the caller's buffer. The caller is responsible for ensuring that the buffer is large enough to hold the `Opn_Blk` structure.

### **Prototype**

```
int get_opn_blk(int port, struct SetOpnBlk *ob);
```

### **Return Values**

Success: 0

Failure: -1 and `errno` set to EBADF: `comm_handle` is not a valid open file descriptor.

-1 and `errno` set to ENXIO: USB device not present.

## get\_port\_status()

---

Copies current status information to caller's 4-byte buffer:

- Byte 1: Number of input messages pending.
- Byte 2: Number of failed output messages pending.
- Byte 3: Number of output messages available:
  - async mode: message is number of bytes
  - sync mode: message is number of packets

**NOTE**


There may not be enough buffers available for all available slots. Each slot corresponds to a single `write()` performed by the application. See [write\(\)](#).

- 
- Byte 4: Current signal information:

- 0x80: always 0.
- 0x40: always 1.
- 0x20: set if CTS detected (COM1, COM2 and COM3).
- 0x08: set if DCD detected (COM2 and COM3 only).
- 0x04: set if frame error detected.
- 0x02: set if overrun error detected.
- 0x01: always 0.

**NOTE**


The ARM architecture does not support parity or break detection.

**Prototype**

```
int get_port_status(int handle, char *four);
```

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_handle, &buffer) != 0)
    printf("\fOutput pending");
close (comm_handle);
```

**Return Values**

Success: 0: No output pending.

Failure: 1: Output on the port is pending output.

-1 and `errno` set to ENXIO: USB device not present.



**NOTE** 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).

## 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.

### Prototype

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

### Parameters

`*device_name` Device name, such as `/DEV/COM1`, `/DEV/COM2`, `/DEV/COM3`, `/DEV/COM4`, or `/DEV/COM5`.

System globals can also be used, such as `DEV_COM1`, `DEV_COM2`, `DEV_COM3`, `DEV_COM4`, and `DEV_COM5`.

### Return Values

Success: 0

Failure: -1 and `ERRNO` set to `EBADF`. This means the device is not connected. This is a normal value for a Vx810 operating as a stand-alone device.

-1 and `ERRNO` set to `ENOENT`. The device is not connected and there is no modem profile loaded. This would be a normal value for a Vx810 operating as a stand-alone device in which no modem profile has been loaded.

-1 and `ERRNO` set to `EACCES`. The application is attempting to open the modem before the OS has finished loading the modem profile.

-1 and `ERRNO` set to `EBUSY`. The application is attempting to open the modem before the OS has finished loading the modem profile and USB Security is enabled.

## read()

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.

### Prototype

```
int read(int comm_handle, char *buffer, int size);
```

### Parameters

<code>comm_handle</code>	Value returned from <code>open()</code> .
<code>size</code>	Maximum number of bytes to read.
<code>*buffer</code>	Pointer to the data area.

### Return Values

Success: Number of bytes read.

Failure: `-1` and `errno` set to `EINVAL`: No `set_opn_blk()` command issued or the caller's buffer is an invalid size.

`-1` and `errno` set to `EBADF`: `comm_handle` is not a valid open file descriptor.

`-1` and `errno` set to `ENXIO`: USB device not present.

## **reset\_port\_error()**

Resets parity, framing, and overrun error conditions.

**Prototype**    `int reset_port_error(int port);`

**Parameters**    `port`    Resets error conditions on the port.

**Return Values**

Success:    0

Failure:    -1 and `errno` set to EBADF: `comm_handle` is not valid open file descriptor.

              -1 and `errno` set to ENXIO: USB device not present.

## **set\_fifo\_config()**

Sets the ARM UART and receive (Rx) FIFO length as follows:

- buffer, Byte 0, bit 0:
  - 1 = FIFO enabled   • 0 = FIFO disabled
- buffer, Byte 0, bits 1–5 unused
- buffer, Byte 0, bits 6–7: Receive FIFO settings:
  - 00 = 4 bytes              • 01 = 8 bytes
  - 10 = 12 bytes (default)   • 11 = 15 bytes
- buffer, Byte 1: Transmit (Tx) FIFO settings:
  - 00 = empty              • 01 = 4 bytes
  - 10 = 10 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.

### **Prototype**

```
int set_fifo_config(int fd, char *buffer);
```

To avoid loss of data, `set_fifo_config()` can only be called after calling `open()`, but before calling `set_opn_blk()`.

### **Return Values**

- |          |   |
|----------|---|
| Success: | 0   |
| Failure: | <ul style="list-style-type: none"> <li>–1 and <code>errno</code> set to <code>EBADF</code>: <code>fd</code> is not a valid open file descriptor.</li> <li>–1 and <code>errno</code> set to <code>EINVAL</code>: <code>set_opn_blk()</code> already called.</li> <li>–1 and <code>errno</code> set to <code>EACCES</code>: Access violation attempting to read from buffer.</li> <li>–1 and <code>errno</code> set to <code>ENXIO</code>: USB device not present.</li> </ul> |

## **set\_opn\_blk()**

Uses the fields of the `Opn_Blk` structure, `*ob`, to initialize/reinitialize the communication parameters. See [The Opn\\_Blk\(\) Structure](#) for more information.

**Prototype**    `int set_opn_blk(int port, struct Opn_Blk *ob);`

**Parameters**    `*ob`    Initializes/reinitializes the communication parameters.

**Return Values**

Success:    0

Failure:    -1 and `errno` set to EBADF: `comm_handle` is not a valid open file descriptor.

              -1 and `errno` set to ENXIO: USB device not present.

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 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`.

## set\_serial\_lines()

Uses the parameters in `buffer` to set/reset DTR, RTS, and BRK.



**NOTE** 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`.

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.



**NOTE** 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.

### Prototype

```
int set_serial_lines(int fd, char *buffer);
```

### Return Values

Success: 0

Failure: -1 and `errno` set to `EINVAL`: No `set_opn_blk()` command issued or the caller's buffer is an invalid size.

-1 and `errno` set to `EBADF`: `fd` is not a valid open file descriptor.

-1 and `errno` set to `EACCES`: Access violation attempting to read from buffer.

-1 and `errno` set to `ENXIO`: USB device not present.

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()`.

## 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.

### Prototype

```
int write(int comm_handle, char *buffer, int size);
```

### Parameters

comm_handle	Value returned from <code>open()</code> .
*buffer	Pointer to the data area.
size	Maximum number of bytes to write.

### Return Values

Success: Number of bytes written.

Failure:

-1 and `errno` set to `ENOSPC`: Not enough buffer space available.

-1 and `errno` set to `EBADF`: `comm_handle` is not a valid open file descriptor.

-1 and `errno` set to `ENXIO`: USB device not present.



## CHAPTER 12

### Security/Crypto Library

This chapter describes Verix V 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:

- |   |   |   |
|---|---|---|
| • <a href="#">rsa_calc()</a>            | • <a href="#">crypto_write()</a>        | • <a href="#">crypto_read()</a>         |
| • <a href="#">AES()</a>                 | • <a href="#">SHA1()</a>                | • <a href="#">DES()</a>                 |
| • <a href="#">iPS_SetPINParameter()</a> | • <a href="#">GenerateRandom()</a>      | • <a href="#">isAttacked()</a>          |
| • <a href="#">iPS_GetPINResponse()</a>  | • <a href="#">iPS_SelectPINAlgo()</a>   | • <a href="#">iPS_RequestPINEntry()</a> |
| • <a href="#">iPS_InstallScript()</a>   | • <a href="#">iPS_CancelPIN()</a>       | • <a href="#">iPS_GetScriptStatus()</a> |
| • <a href="#">iPS_ExecuteScript()</a>   | • <a href="#">iPS_UninstallScript()</a> |   |

## 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()
- rsa\_calc()
- DES()
- AES()

### Random Number Generation

- GenerateRandom()

### Attack Detection

- isAttacked()

### 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.

On V<sup>X</sup>670, the DS3610 has two event capture registers that record any tamper event that meets the qualification of time, temperature, temperature rate-of-change, voltage, or frequency. 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 /TEI 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 of DS3610. 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 F4` on System Mode 2 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.

---

**NOTE**



The new Dallas RTC/security DS3610 chip on V<sup>X</sup>670 handles the real-time clock (RTC), GEK storage, tamper event logging, and random number generation.

---

## **crypto\_read()**

Reads a maximum of count bytes of encrypted data from the open file associated with handle, 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 handle, char *buffer, int count);
```

### **Parameters**

`handle` File handle.

`buffer` 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 handle.

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 handle, const char *buffer, int count);
```

**Parameters**

`handle` File handle.

`buffer` Pointer to the buffer holding the input data.

`count` Number of bytes to write.

**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, 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 V-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

ucAesOption      Specifies the operation type and key length:

- 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

pucAesKey8N      Pointer to 8N-byte key block (N=2, 3 or 4).

pucInputData      Pointer to 16-byte input block.

pucOutputData      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

ucDeaOption	Algorithm:
	<ul style="list-style-type: none"><li>• DESX1KE (02h): DEAX encryption with single-length key</li><li>• DESX1KD(03h): DEAX decryption with single-length key</li><li>• DESX2KE (04h): DEAX encryption with double-length key</li><li>• DESX2KD(05h): DEAX decryption with double-length key</li><li>• DESX3KE (06h): DEAX encryption with triple-length key</li><li>• DESX3KD(07h): DEAX decryption with triple-length key</li><li>• DESE (08h): DEA encryption with single-length key</li><li>• DESD (09h): DEA decryption with single-length key</li><li>• TDES2KE (0Ch): TDEA encryption with double-length key</li><li>• TDES2KD (0Dh): TDEA decryption with double-length key</li><li>• TDES3KE (0Eh): TDEA encryption with triple-length key</li><li>• TDES3KD (0Fh): TDEA decryption with triple-length key</li></ul>
pucDeaKey8N	Pointer to 8N-byte key block (N=1, 2 or 3).
pucInputData	Pointer to 8-byte input block.
pucOutputData	Pointer to 8-byte output block.

### Return Values

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**    random8    Pointer to the 8-byte buffer where the random value transfers.

**Return Values**

Success:    0, executed

Failure:    -1 with errno set to EACCES, 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);`

**Parameters**    None

### **Return Values**

Success:    0, no attack occurred since keys were last loaded.

Failure:    1, an attack occurred and encrypted files are lost.

## 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

msg	Array of unsigned 16-bit integers holding the input.
mod	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.
wds	Number of 16-bit unsigned shorts in msg and mod.
exp	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.
result	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 errno set to EACCES, invalid buffer pointer.

-1 with errno set to EINVAL, 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.

## VeriShield Security Script Functions

The Verix V 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 V 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 a PC tool and converted into a downloadable file (`*.vso`) for the Verix V. 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.

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 V-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:

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• <a href="#">iPS_GetScriptStatus()</a></li> <li>• <a href="#">iPS_InstallScript()</a></li> </ul> | <ul style="list-style-type: none"> <li>• <a href="#">iPS_ExecuteScript()</a></li> <li>• <a href="#">iPS_UninstallScript()</a></li> </ul> |
|--|--|

See [VeriShield Security Scripts](#) for more information.

The functions used to manage PIN entry are:

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• <a href="#">iPS_CancelPIN()</a></li> <li>• <a href="#">iPS_RequestPINEntry()</a></li> <li>• <a href="#">iPS_SetPINParameter()</a></li> </ul> | <ul style="list-style-type: none"> <li>• <a href="#">iPS_GetPINResponse()</a></li> <li>• <a href="#">iPS_SelectPINAlgo()</a></li> </ul> |
|---|---|

The [Key Loading Functions](#) are:

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• <a href="#">iPS_DeleteKeys()</a></li> <li>• <a href="#">iPS_LoadSysEncKey()</a></li> <li>• <a href="#">iPS_LoadMasterEncKey()</a></li> </ul> | <ul style="list-style-type: none"> <li>• <a href="#">iPS_LoadSysClearKey()</a></li> <li>• <a href="#">iPS_LoadMasterClearKey()</a></li> <li>• <a href="#">iPS_CheckMasterKey()</a></li> </ul> |
|---|---|

## iPS\_GetScriptStatus()

Checks if a VeriShield security script file is installed in the Verix V terminal and, if so, returns the name of the script.

## Prototype

```
int iPS_GetScriptStatus(unsigned char ucScriptNumber,
                      unsigned char *pucINNname);
```

## Parameters

**ucScriptNumber** Script number. Range [0..7]

**pucINName** 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**

pucINName    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 <code>pusOUTDataSize</code> . If the macro returns more data than the output buffer can contain, an error <code>E_VS_BAD_LENGTH</code> is returned and nothing is copied into the output buffer.

### **Return Values**

Success:	0, executed.
Failure:	<code>EBADF</code> , caller does not own crypto device.  <code>EACCES</code> , invalid buffer pointer.  <code>E_VS_SCRIPT_NOT_LOADED</code> , script is not loaded, not authenticated, or not accessible from the current GID.  <code>E_VS_MACRO_NOT_EXIST</code> , macro does not exist in this script.  <code>E_VS_BAD_LENGTH</code> , <code>usINDataSize</code> is less than of the value expected by the macro or <code>usOUTDataSize</code> is less than the number of bytes the macro is attempting to return.  <code>E_VS_BAD_CHAINING</code> , bad sequence of macro (see Chaining Mechanism).  <code>E_VS_SYSTEM_ERROR</code> , 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**

ucScriptNumber 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.

The PIN entry functions described in this section are only to be used with VeriShield Secure Scripts.

---

## **iPS\_CancelPIN()**

---

Cancels the PIN processing.

**Prototype**    `int iPS_CancelPIN(void);`

**Parameters**    None

**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 functions returns the number of PIN digits entered and the last non-numeric key pressed.

### **Prototype**

```
int iPS_GetPINResponse(int * piStatus, PINRESULT * pOUTData);
```

### **Parameters**

<b>piStatus</b>	OK(0x00): Done. The PIN is stored internally and is ready to be processed by a script command. <ul style="list-style-type: none"><li>• 0x01: Unit is idle.</li><li>• 0x02: Collecting PIN.</li><li>• 0x05: Aborted by user (the &lt;CANCEL&gt; key ()) was pressed).</li><li>• 0x06: No PIN entered (only if this option is turned on).</li><li>• 0x0A: Aborted by user. The &lt;CLEAR&gt; 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 ucOption.bit4 has been set using the <a href="#">iPS_SetPINParameter()</a> function.</li></ul>
<b>pOUTData</b>	Pointer to the <code>PINRESULT</code> context structure giving the parameters used with the <code>iPS_GetPINResponse()</code> function. This struct is defined in the <code>svc_sec.h</code> file.  This structure will return different information depending on the status of the PIN session. If <code>*piStatus</code> is equal to: <ul style="list-style-type: none"><li>• OK(0x00): done.<ul style="list-style-type: none"><li>- <code>pOUTData-&gt;nbPinDigits</code> Number of PIN digits entered (PIN length).</li><li>- <code>pOUTData-&gt;encPinBlock</code> 8-byte buffer contains no relevant information.</li></ul></li><li>• 0x01: PIN pad is idle<ul style="list-style-type: none"><li>- <code>pOUTData</code> contains no relevant information.</li></ul></li></ul>

- 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).
    - 0xD: Last key pressed was the <ENTER> ( ) key.
    - 0x8: Last key pressed was the <CLEAR> ( ) key.
- 0x05 or 0xA: 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:
    - 0x18: <CANCEL> ( ) key
    - (0x08: <CLEAR> ( ) key
- 0x06: No PIN entered (Only if this option is turned on)
  - pOUTData contains no relevant information.

### ***Return Values***

Success: 0, executed.

Failure: EBADF: Caller does not own crypto device.

EACCES: 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 30 seconds Verix V adds one token until the maximum 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 Verix V, 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.

EACCES, 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\_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 <a href="#">iPS_SetPINParameter()</a> 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.
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 <a href="#">iPS_GetPINResponse()</a> function is 0x0A.
ucOption.bit5..7	RFU - Must be 0

### 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 KVC (key verification code) argument is irrelevant in Verix V 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.

- 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 secure 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"><li>• 00: Key set defined in VSS #0</li><li>• 01: Key set defined in VSS #1</li><li>• ...</li><li>• 07: Key set defined in VSS #7</li></ul>
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.

## Parameters

`ucKeySetID` Key set identifier.

- 00: Key set defined in VeriShield Security Script #0
  - 01: Key set defined in VeriShield Security Script #1
  - ...
  - 07: Key set defined in VeriShield Security Script#7

ucKeyID

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);

<b>Parameters</b>	
ucKeyID	Key identifier. <ul style="list-style-type: none"><li>• 0x00 = VSS_KLK (16 bytes)</li></ul>
pucINKeyValue	16-byte buffer containing the encrypted keys.

<b>Return Values</b>	
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.



## APPENDIX A

### System Mode

Actions such as diagnosing hardware problems, changing the system configuration, downloading new applications, and updating the operating system can be done via the System Mode functions.

- Press F2 and F4 (or the ENTER key and the 7 key) at the same time and enter the password to invoke System Mode.
- Assign files and applications to groups for access control.
- Use the system and file group passwords to secure applications and information on the terminal.
- Use the System Mode menus and submenus to configure terminals; download, test, and debug applications; and perform routine tests and terminal maintenance.

System Mode is used exclusively by those responsible for configuring, deploying, and managing on-site terminal installations.

### When to Use System Mode

Use the System Mode functions to perform different subsets of related tasks:

- **Application programmers** configure a development terminal, download development versions of the application program, then test and debug the application until it is validated and ready to be downloaded to other terminals.
- **Deployers of terminals to end-user sites** perform the specific tasks required to deploy a new 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 System Mode menus and execute the corresponding procedures.

### Local and Remote Operations

The System Mode operations available on a 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 System Mode 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 System Mode 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 System Mode operations.

## Verifying Terminal Status

The 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 and the terminal is in download mode. On all terminals, the COM1 serial port is open and waiting for direct download (on terminals that support it, the USB port is also ready for direct download). Press F2 and F4 to perform the required download.



**NOTE** Enter System Mode by simultaneously pressing F2 and F4, or the 7 and ENTER keys. On Vx700 Unattended Payment Terminal, only the key pair 7 and  is available. The function keys F1 – F4 are not supported.

- If an application program is loaded into 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 System Mode

To prevent unauthorized use of the System Mode menus, the terminal OS requires a system password each time you enter System Mode. To access the System Mode password entry screen, simultaneously press the F2 and the F4 keys, or the 7 and  keys. The default factory-set system password is “Z66831” use the following key sequence to enter this password:

1 ALPHA ALPHA 6 6 8 3 1 .

For the Vx700 PIN pad that employs a variant of the EBS 100 keypad mapping, use the following key sequence to enter this password:

9 HELP HELP HELP HELP 6 6 8 3 1 .

System Mode prompts for the System Mode entry password. If the password is entered correctly, then System Mode terminates all application tasks and enters System Mode. Cell phone/multi-key entry is not activated for password entry. If incorrect password is entered, the console ownership is returned to the pre-empted application along with a notification event.

The System Mode password must be between 5 and 10 characters long. Otherwise, System Mode 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 System Mode and displays the first System Mode main menu, SYS MODE MENU 1. You can now toggle through all System Mode main menus.



If you change a password but forgot it later on, no password recovery method is available. Without the password, you are unable to access System Mode 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.

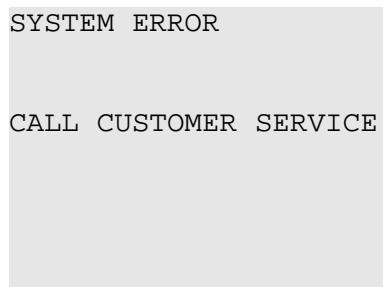
**File Integrity Check** At startup, the System Mode queries the state of the file systems and determines if they are in a corrupt state using the SVC\_CHECKFILE( ) system call.

RAM CHKSUM ERROR G1 I : ABCD.DAT	RAM CHKSUM ERR G1 F : ABCD.DAT
DELETE F2	DELETE F2
CLEAR ALL RAM F3	CLEAR ALL FLASH F3
IGNORE F4	IGNORE F4

The file ABCD.DAT represents the corrupt file.

- The DELETE option prompts for the GID 1 password before removing the file with the CHECKSUM error.
- The CLEAR ALL option prompts for the GID 1 password before removing all files. In the flash file system, all files are removed without exception and the Flash is also coalesced/defragmented. In the RAM file system, all files except CONFIG.SYS in group 1 are removed. The unprotected variables from CONFIG.SYS are also removed.

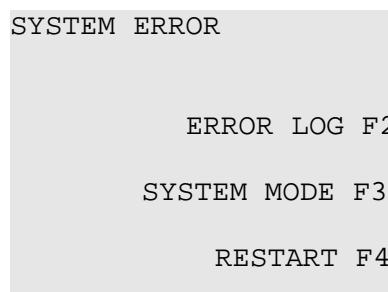
To avoid system errors while clearing the RAM or Flash file systems, System Mode must be running from the boot sector. If the boot sector is corrupted, the unit must be returned for repair.



- The IGNORE option ignores the corrupt file and proceeds to the normal start-up.

**System Error Handler**

In the event of a Bus Error or an Address error, the system logs debug information in kernel memory and restarts. System mode will test for this condition on each start-up. When necessary, the system displays:



- ERROR LOG displays the error log screens.
- SYSTEM MODE prompts for the System Mode entry password and then proceeds to the first interactive menu.
- RESTART reboots the system.

The screen will be displayed again but the error log can be viewed in System Mode.

**Developer Download**

Configure \*SMDL to enable polling for direct download on COM1 during the start-up sequence and before the copyright screen. If \*SMDL =1, a direct download is attempted during start-up. The system looks for ENQs on the line, trying both 115,200 and 19,200 bits/s. If no data is detected, normal start-up resumes.



This option is provided only as a convenience for application developers. Do not enable for terminals deployed into service.

**Resumable Download**

In a resumable full download, System Mode does not remove files until all split files are received and checked for a correct checksum, unlike in a regular full download where System Mode removes files before the download starts, which means losing the application when the download fails. Refer to [Resumable Download](#) for more information.

**IP/Dial Download**

To protect the application files, the OS does not allow files or parameters to be cleared from the GID if configured to be protected—all requests to clear either GIDs 1 or 15 will be ignored. RAM and Flash files are also protected and cannot be deleted. These files will remain in the terminal even if the user selects the “CLEAR MEM” option. Refer to [IP Persistence](#) for more information.

## Start-up Screens

On start-up, System Mode displays the terminal model, OS version information, and copyright notice for three seconds. Display duration can be extended by pressing any key—each key press restarts the three second timer.

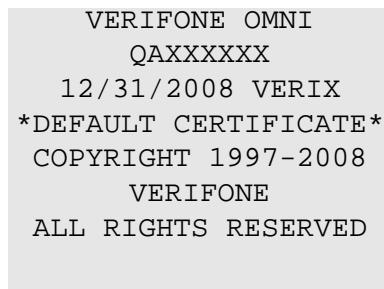
The OS information displayed on the screen is obtained from `SVC_INFO_EPROM()` and `SVC_VERSION_INFO()`.



**Figure 22 Initial Screen**

### Default Certificate

If the terminal detects that the default certificate is in use, the screen below is displayed for three seconds:



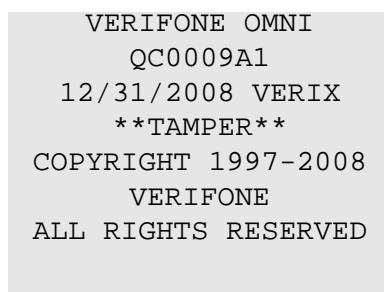
**Figure 23 Initial Screen with Default Certificate**

### Error Detection Screen

When errors are detected, System Mode alerts the user and displays the corresponding error screen.

#### Tamper Detect

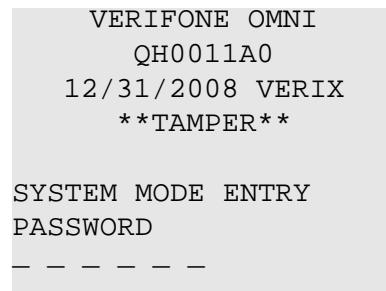
If the terminal detects that it may have been tampered with, it displays the following screens:



**Figure 24 Tamper Screen on an 8-Line Display**

The "/\* \* T A M P E R \* \*/" line on the screen blinks, to continue, press either the  or the  keys.

On an 8-line PIN pad, the following screen is displayed:



**Figure 25 Tamper Screen on an 8-Line PIN Pad Display**

**NOTE**

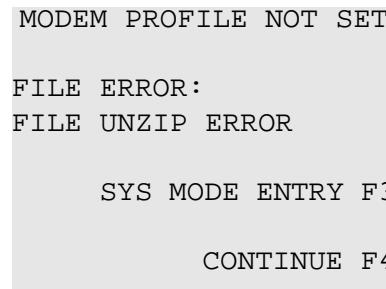


On the Verix V Pin pads, two passwords are required to clear the Tamper flag—the System Mode and the GID 1 passwords.

On the initial screen, enter the System Mode password. Select IPP KEY LOAD menu. You are required to enter the GID 1 password. The TAMPER flag is cleared by pressing F1 if available and key 2 on UPT units.

### Country Profile Error

The modem profile containing country configuration is loaded if the modem is present at startup. If the modem detects a problem with the Modem Country Profile, the following screen is displayed:



- SYS MODE ENTRY bring the user to the System Mode Entry Password screen.
- CONTINUE ignores the error.

For more information on modem profile loading errors, refer to [Table 57 \\*MERR values](#).

## Interactive System Mode Entry at Start-Up

At start-up, on the copyright screen display, there is a three second timing window that allows user to assert two simultaneous key pair presses, F2 and F4 keys or the 7 and  keys, indicating the request to enter interactive System Mode. This occurs before the default \*GO processing to enable the user to intercept application start-up. The numeric UPT only supports the key pair 7 and  keys for System Mode entry.

The 7 and  key pair or the F2 and F4 key pair can also be used to enter System Mode at any time after the application starts.

---

**NOTE**

On the V<sup>x</sup>700 unit, only the 7 and  key pair are available for System Mode entry.

---

## ZIP File Support

System Mode scans GIDs 1 through 45 for the CONFIG.SYS variable \*UNZIP. When set, System Mode invokes UNZIP decompression on the specified file.

## VeriShield File Authentication Support

System Mode is responsible for initiating VeriShield file authentication and key certificate management functions. If a file fails authentication, a message will be displayed. A key must be pressed to clear the message. The file will not be executed.

## Automatic Flash Coalesce

The System Mode variable \*DEFrag checks the Flash file system for deleted files and optionally coalesce (defragmented) the Flash.

- If \*DEFrag is not defined, set to zero, or is non-numeric, the Flash is coalesced if there are deleted files.
- If \*DEFrag is greater than zero, the Flash is coalesced if the amount of Flash freed up after the coalesce is equal to or greater than the value of \*DEFrag in K bytes.
- If \*DEFrag is less than zero, the Flash is never coalesced. The System Mode Flash defrag function is also available for manual coalesce.

## \*GO and \*ARG Processing

System Mode runs before any other applications are started. It is responsible for processing the \*GO environment variable in CONFIG.SYS. Each group can have its own CONFIG.SYS file but the application with the highest privilege will start-up the other applications.

System Mode starts up the application specified by \*GO in CONFIG.SYS for GID 31. If no variable exists in GID 31, it starts the application defined by \*GO in GID 1, which is the ‘sponsoring application.’

If \*ARG exists in CONFIG.SYS, its contents are passed as command line arguments to the process started by \*GO processing. If there is no \*GO environment variable in GID 1 CONFIG.SYS file, System Mode displays the following screen:

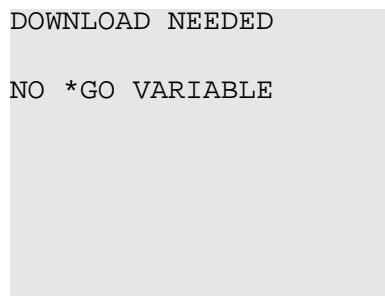


Table 58 enumerates the error messages and their description:

**Table 58 \*GO Error Messages**

Error Message	Description
NO *GO VARIABLE	No *GO environment variable in GID 1 or 31 CONFIG.SYS file.
*GO FILE NOT FOUND	*GO is set but <code>run()</code> system call fails because the executable file is missing.
*GO NOT AUTHENTICATED	*GO is set but <code>run()</code> system call fails because the executable file was not authenticated.
NOT ENOUGH MEMORY	*GO is set but <code>run()</code> system call fails because there is not enough memory to execute the file.
INVALID *GO VARIABLE	Default error condition. The above 3 error conditions do not hold true, but the system could not run *GO.

System Mode then polls the port/s for a download and also polls the keyboard for the interactive System Mode entry keystroke pair. The ports polled are dependent on the specific platform. Some Verix V platforms poll more than one port in sequence. Platforms that support USB memory stick devices or SD memory card devices will load from those devices if the stick or card is present.

**Table 59 Polled Ports and Devices**

COM Port	Terminal/PIN Pad
COM1	Vx510, Vx570, Vx610, Vx670, Vx700, V5
COM2	Vx800, Vx810 PIN pad
USB device/COM6	Vx570, Vx510G, Vx670, Vx810, Vx700
USB memory stick	Vx570, Vx670, Vx700, Vx810
SD memory card	Vx700, Vx810

During FULL download. The memory erased and coalesced is dependent on the chosen download interface. All platforms, except the Vx670, erase all GID 1 RAM and Flash and coalesce/defragment the Flash.

The V<sup>x</sup>670 offers two download interfaces, ORIGINAL and ENHANCED. The ENHANCED download is the same as the interface described above for all other platforms. If the ORIGINAL download interface is active, RAM is erased in the GIDs where the files are downloaded. No Flash files are erased.

## System Mode User Interface Conventions

The default 6x8 font is employed for robustness and does not require specific font files to reside in the file system. Since future versions may be constrained to upper case only when implemented in the Keyboard and Display Controller for space reasons, System Mode prompts are all in upper case. Most of the System Mode static text are encapsulated in a separate file to allow the menus and prompts in English to be easily translated.

The top line of each menu presents a static title to indicate that System Mode is running. The user navigates through the System Mode menus using either function keys/user addressable keys or numeric prompts.

### Function Keys

On terminals/keypads supporting them, function keys are arranged with four unmarked keys at the bottom of the display and four or six keys to the right side of the display or keypad. They are marked F1 – F4 (4 keys) or F0 – F5 (6 keys).

#### NOTE



The V<sup>x</sup>700 unit does not have function keys. It relies instead on numeric prompts.

In the following discussion Fn will be used unless a specific key is referred to.

The bottom line, line 8 or line 16, is reserved for labels for the function keys below the screen for navigational purposes such as  $\uparrow$  and  $\downarrow$ . Three of four available function keys are left on the right of the screen for menu selections in the 8-line display and four or six on the 16-line display. Each label associated with an Fn key contains a key name and a prompt, and are right justified on the screen.

Within specific functions, the or red CANCEL key consistently returns the user to the previous screen, not including password or GID entry screens. The or green ENTER key indicates the completion of text or numeric entry.

Holding down the or CLEAR key (generates Clear key code) resets a currently active edit field. Otherwise, all navigation will be performed via function keys.

### Group ID

Some System Mode operations require a file group context—seeing the files associated with a particular application in the RAM (I:) or the flash (F:) file system, download to a particular file group only, or run the CONFIG.SYS editor for a specific application which is in a specific file group, or update the password for a specific file group.

To support these file groups, a group identifier has been appended to the path within the file systems. This group is a byte that is associated with the executable of the process that is running. The System Mode executable will contain a field in its header that identifies GID 1.

If a multi-level directory structure was employed, then a file operation could require multiple file updates which would be difficult to recover in the case of restarting from a power loss. Groups allow the file system to remain flat, at the same time, avoid namespace collisions by two separate applications.

## File Group Conventions

A hierarchical rights relationship exists in the file groups. The System Mode and the ‘sponsoring application’ are configured to run in GID 1. GID 0 is reserved for operating system components. GID 1 has the right to employ the `set_group()` system call to values greater than or equal to one. Processes running in GIDs 2 through 14 have the right to use `set_group()` to access their own group and GID 15, which is a public area accessible to any group.

Each group can have its own unique CONFIG.SYS file. Variables used by System Mode will reside in the CONFIG.SYS file in group one.

## System Mode Menus

Verix V uses a 6 x 8 pixel default font for System Mode, which provides consistency over all platforms. The System Mode is displayed in the following display sizes with 21 characters per line:

- 4 lines, 32 x 64 pixel - available on both PIN pads and terminals.
- 8 lines, 64 x 64 pixel - has two variants, standard and numeric.
- 16 lines, 128 x 64 pixel - has two variants, the 4-screen key variant that uses the standard System Mode screen flow, and the 6-screen key variant.

### NOTE

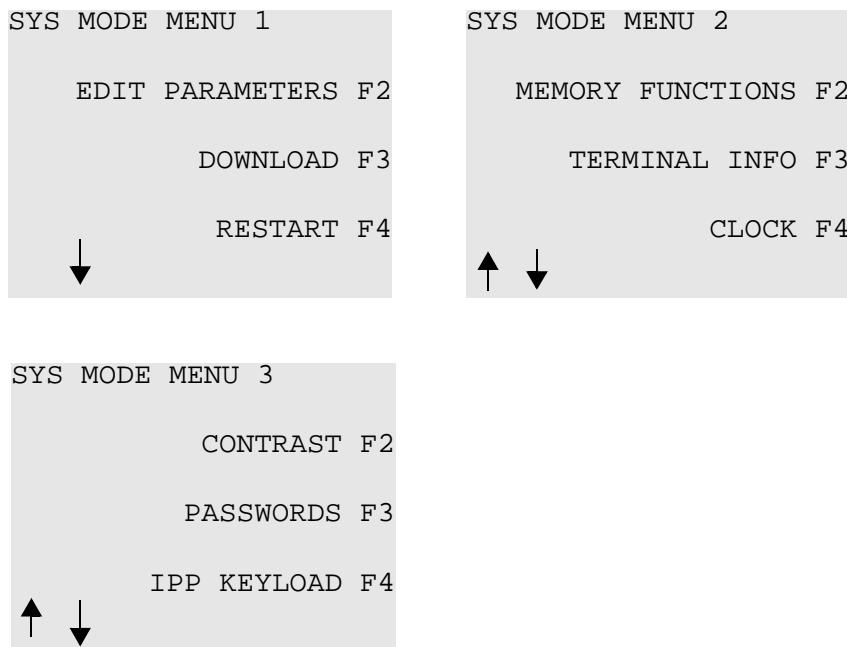


Refer to the terminal's/PIN pad's respective Reference Manual for a more detailed discussion of the System Mode screen sequence.

## Eight-Line Standard Interface

The eight-line user interface is similar for Vx570, Vx510 and Vx610 terminals, except where there are feature or hardware differences.

The three main menus in the standard 8-line interface are as follows:



System Mode Menu 1 allows the user to edit parameters, perform a download, and restart the terminal.

System Mode Menu 2 provides the user with memory, terminal, and clock information.

System Mode Menu 3 allows the user to adjust the terminal's display contrast, manage the System Mode and file group passwords, and check the IPP key loading mode.

### Back-To-Back Download

Pressing either the star (\*) key or the pound (#) key on each System Mode menu triggers a back-to-back download (or unit-to-unit download). If the star key is pressed, the back-to-back download screen is displayed, prompting the user for GID 1 password. If the pound key is pressed, the GID 1 password is prompted. Enter the correct password to start terminal send.

The System Mode upload screen is displayed, indicating the file being transferred and the progress status.

```
SYS MODE UPLOAD  
I: CONFIG.SYS  
*****  
UPLOADING NOW
```

The System Mode download screen appears, similar to the screen in the System Mode download section.

```
SYS MODE DOWNLOAD      G1  
*****  
DOWNLOADING NOW
```

### Interactive Group Selection

When an operation requires a confirmation or needs to change the group context, a group entry screen is prompted:

```
SYS MODE FILE  
  
FILE GROUP _1
```

If the group is changed, the password for the new group is prompted.

```
SYS MODE FILE  
GROUP 1 PASSWORD  
_____
```

A group password is required when editing a group's CONFIG.SYS file, downloading files to a group, and changing passwords and load IPP keys. System Mode remembers any passwords successfully entered during a given session. These passwords are not reentered when toggling between groups for different operations.

### File Group Operations

For operations that require a group context, the current group is displayed on the screen. [Table 60](#) illustrates the group display on specific operations.

**Table 60      File Group Display**

Operation	File Group
RAM CHKSUM Error	Display Group
Flash CHKSUM Error	Display Group
Clock	N/A
Contrast	N/A
Restart	N/A
Download	Prompt for Group
Clear RAM	Prompt for Group
Clear FLASH	Prompt for Group
Configuration	N/A
Editor	Prompt for Group
Passwords	Prompt for Group
Remote Diags	N/A
Error Log	Display Group
Debugger	Prompt for Group
Screen Diag	N/A
Keyboard Diag	N/A
Mag Card Diag	N/A
IPP Diag	N/A
IPP Key Load	Prompt for Group
Printer Diag	N/A
RAM Directory	Display Group (Vx670 Prompt for Group)
Flash Directory	Display Group (Vx670: Prompt for Group)

- Display Group - displays the group where an error is detected.
- Prompt for Group - prompts for the group to act on and the group password, then displays the group on subsequent screens.

## Download Interface

Verix V offers a variety of download interfaces. These download communications are platform-specific and are enumerated in [Table 61](#).

**Table 61** Download Interfaces

Download COM	Vx510	Vx610	Vx570	Vx670	V5	Vx700 PIN Pad	Vx810 PIN Pad	Vx800
COM1	x	x	x	x	x	x		x
COM2	x		x		x	x	x	x
COM3 (MODEM)	x	x	x	x	x	x	x	
							(base)	
COM8						x	x	
COM6 (USB UART)			x	x		x	x	
USB Device				x		x	x	x
TCP/IP	x	x	x	x		x	x	
SD Card						x	x	
USB Memory Stick			x	x		x	x	

TCP/IP requires a TCP/IP application and the \*ZTCP variable must be set to a TCP/IP application name, except for the Vx510 terminal.

Below are possible error messages if the download does not complete successfully:

**Table 62** System Mode Download Error Messages

Error Message	Description
NO LINE	Phone line in use.
NO DIAL TONE	No dial tone.
NO CARRIER	Could not establish communications.
BUSY	Busy signal.
NO ENQ FROM HOST	Host did not send ENQ.
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 RESP FR HOST	Timed out waiting for packet from host.

Refer to the VeriCentre or ZonTalk 2000 documentations for error messages generated by the download host.

Error messages for IPP DIAG and IPP KEY LOAD are reported on the last line of the screen. They have the form “\*\* 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 given in errno.h and negative numbers map to values generated by the System Mode application.

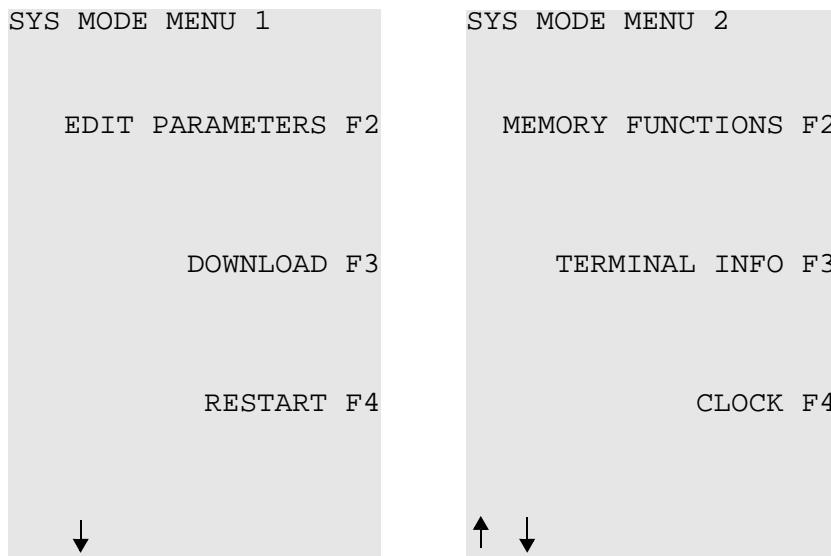
**Table 63 IPP DIAG/Key Load Error Codes**

Error Code	Description
-1	Can't open COM5 (errno set).
-2	Can't communicate with IPP.
-3	COM5 read/write error (errno set).
-4	Timeout waiting for start of packet.
-5	Timeout waiting for next byte, <ACK>, etc.
-6	Received too many <NAK> responses.
-7	Received <EOT> before end of transaction.

### 16-Line, Four Function Keys Interface

The 16-line interface, 4-screen key menu system has the same screen flow as the standard 8-line mode with few exceptions—the download media and the modem System Information are platform-specific, and if contactless is supported, contactless diagnostics may be offered in the Diags and Logs.

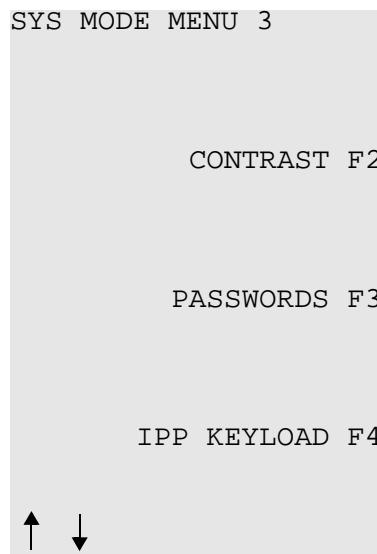
The three main menus in the 16-line interface are as follows:



System Mode Menu 1 allows the user to edit parameters, perform a download, and restart the terminal.

System Mode Menu 2 provides the user with memory, terminal, and clock information.

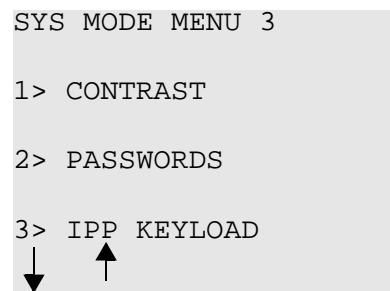
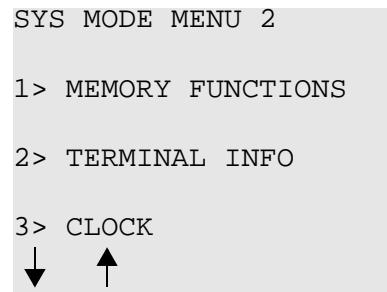
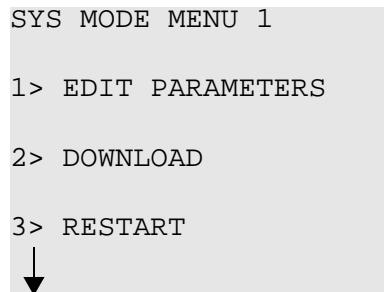
System Mode Menu 3 allows the user to adjust the terminal's display contrast, manage the System Mode and file group passwords, and check the IPP key loading mode.



### Eight-Line Interface with Numeric Prompts

This interface is available on platforms that do not offer user definable screen keys, such as the Vx700 PIN pad. The screen flow is the same as the standard 8-line interface with few exceptions—the menus that offer choices and data input become two menus, and the menu choices are left justified or arranged at the bottom of the screen and are numbered sequentially.

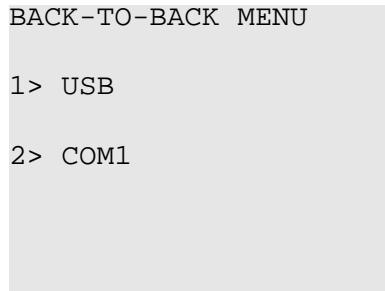
The three main menus in the standard 8-line interface with numeric prompts are as follows:



Press the corresponding keypad number to make the given choice.

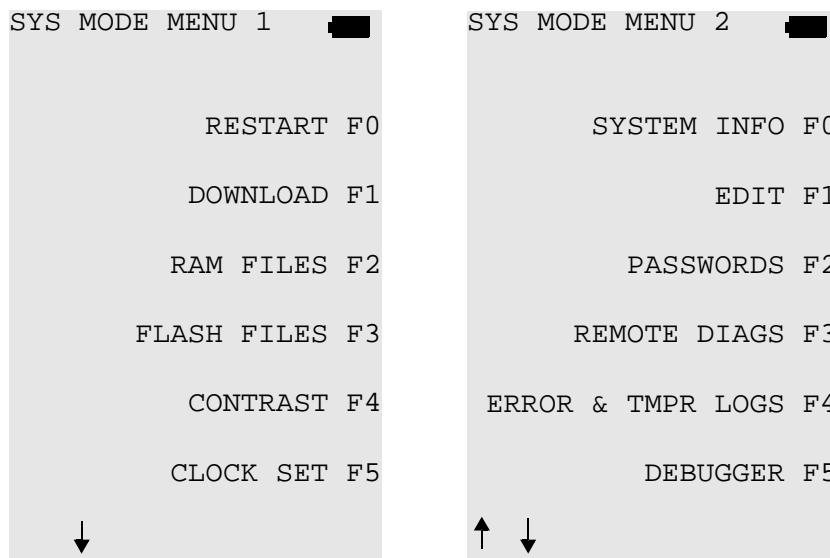
### Back-To-Back Download

Back-to-back is activated from any one of the three main menus. Press the 4 key to activate upload. Press the 5 key to activate download. The screen below appears after entering the password. After selecting the appropriate interface, the standard 8-line uploading or downloading screen is displayed.



### 16-Line, Six Function Keys Interface

The 16-line menu structure features four main menus with five user choices such as in the V<sup>x</sup>670 terminal. Below are the four main menus in the 16-line, six function keys interface:



System Mode Menu 1 allows the user to restart the unit, perform full or partial downloads to the terminal, clear RAM and Flash files, adjust the display contrast, and set the terminal clock.

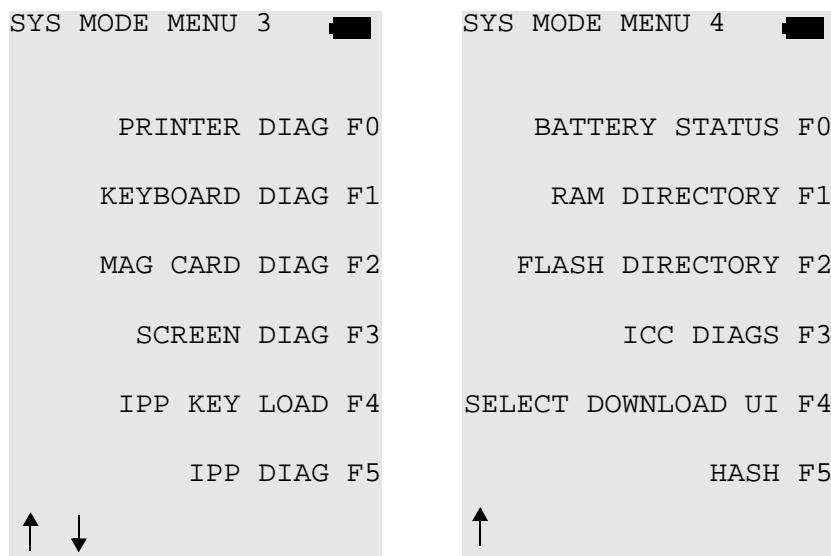
System Mode Menu 2 lets the user view the terminal configuration information, edit CONFIG.SYS or other keyed file, manage System Mode and file group passwords, view error logs, and perform application debugging operations.

System Mode Menu 3 allows the user to display printer information and run printer keyboard, magnetic card and display panel tests, and check the IPP keyloading mode.

## SYSTEM MODE

### System Mode Menus

System Mode Menu 4 lets the user view the battery status, file directory information for all the groups within the memory partition, run ICC diagnostics, and choose the desired download user interface.



### Four-Line Interface

On a normal startup, the four-line interface such as that used by the Vx800, presents the following screen sequence:

VERIFONE OMNI VMT  
QE0008A0 04/18/2007  
(C)2007 VERIFONE INC.  
ALL RIGHTS RESERVED

The copyright notice is displayed

VERISHIELD PROTECTED  
\*DEFAULT CERTIFICATE\*

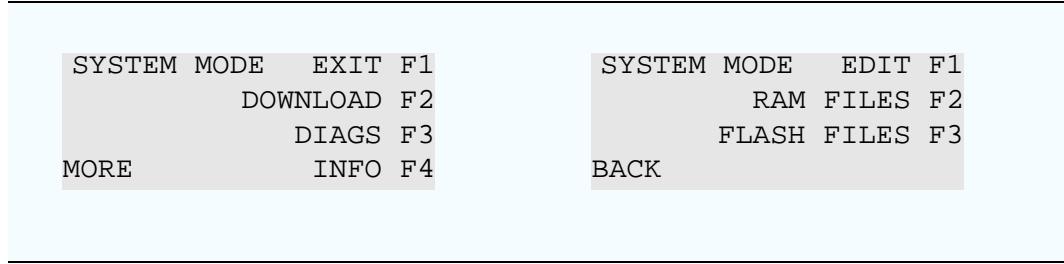
Default certificate is displayed.

If there was tamper attempt, the tamper screen appears:

\* \* T A M P E R \* \*  
\*DEFAULT CERTIFICATE\*

Default certificate is displayed.

Below are the main menus in the four-line interface:



System Mode Menu 1 allows the user to exit System Mode, perform full or partial downloads to the terminal, run hardware and software diagnostics, and view the terminal configuration information.

System Mode Menu 2 lets the user edit the CONFIG.SYS file or other keyed files, and clear the RAM and Flash files.

---

## **SYSTEM MODE**

### *System Mode Menus*



## APPENDIX B

# 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 triple-DES 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 V terminals.

In its default configuration, the Verix V-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 V 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, triple-DES, AES, and RSA

Refer *VeriShield Security Scripts, 21883* for more information.





## APPENDIX C

### 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 system mode 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. System mode 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 F3` from SYS MODE MENU 6 (`IPP KEYLOAD F4`, SYS MODE MENU 3 on Vx670) they are normally prompted to enter the group 1 password, using the standard prompt dialog. However, password entry is not required if the default password (Z66831) is in use. This is intended to facilitate initial key loading for newly deployed terminals.

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



\*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.

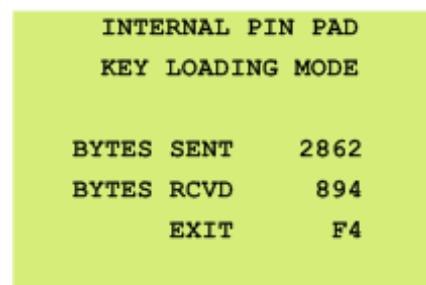
E, O, D, and R also set Fmt\_A7E1, Fmt\_A7O1, Fmt\_DTR, and Fmt\_RTS, respectively.

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)

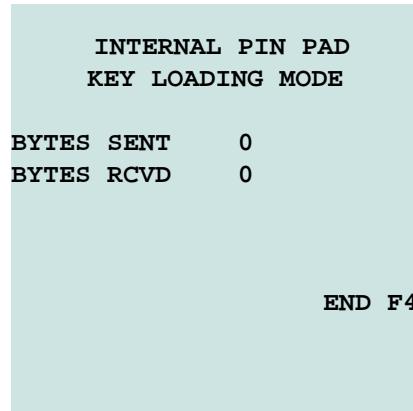
\*IPPMKI is intended to support key loading software with fixed communication requirements; the baud rate probably has no significant effect on performance given the small amount of data involved. Note that COM1 settings are independent of COM5 (IPP) settings.

When pass-through mode starts, the following screen displays:



**Figure 26      Sample IPP Key Load Screen for Vx5xx/Vx610 Terminals**

Figure 27 shows sample IPP Key Load screen for Vx670 terminal



**Figure 27 Sample IPP Key Load screen for Vx670 Terminal**

The byte counts are updated to provide feedback to the user. Pressing CLEAR ends passthrough mode, closes COM5, and returns to SYS MODE MENU 6 (SYS MODE MENU 3 on Vx670).

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 returns to SYS MODE MENU 6 (SYS MODE MENU 3 on Vx670).

## 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 system mode application, as follows:

- -1 Can't open COM5 (`errno` set)
- -2 Can't 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 Vx670, two new timeouts are enforced during IPP keyloading. If no data is received from the host within 60 seconds, the program will terminate and return to System Mode. 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 new `system_integrity_check` has also been added to the System Mode. 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. 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 Vx510, Vx570, Vx610, Vx670, Vx810, V5 PIN pad, and Vx700 operating systems conform to the PCI PED requirements for initiating the IPP key loading operation.

The System Mode IPP key loading process requires users to enter the System Mode password followed by the GID 1 password. In compliance to the PCI PED, the requisite System Mode 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 System Mode password or any GID password from the System Mode Passwords Menu.

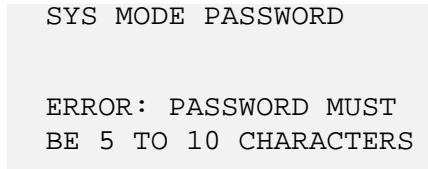
### To change the password manually

- 1 Navigate to the System Mode 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

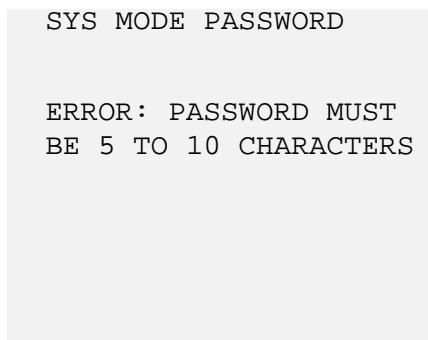
Attempting to enter a new password with less than five characters results in an error message accompanied by a beeping sound.

Figure 28 shows the error message on a four-line display.



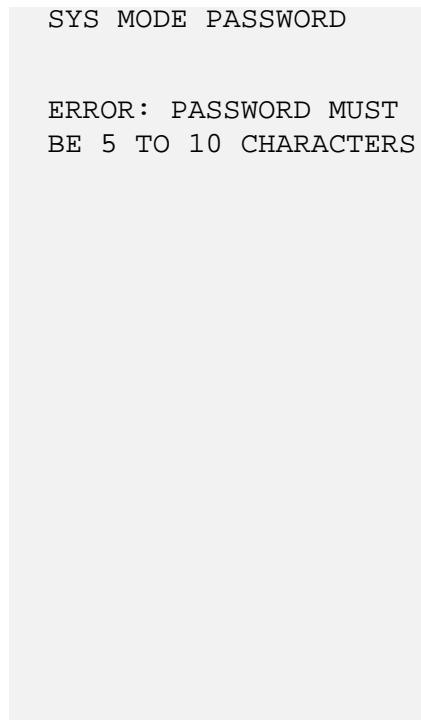
**Figure 28 Error Message on a Four-Line Display**

Figure 29 shows the error message on an eight-line display.



**Figure 29 Error Message on an Eight-Line Display**

Figure 30 shows the error message on a 16-line display.



**Figure 30 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 System Mode 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 System Mode 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 System Mode 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):

ABCDEFGHIJKLMNPQRSTUVWXYZ

0123456789

. , " - +#!;:@=&/ \%\$ \_

## OS Upgrade

When upgrading from earlier Verix V OS, 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.

## Default Password

The OS sets a default password of Z66831 for System Mode 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 System Mode session.

The GID 1 password entry is also required when IPP KEY LOAD operation is restarted (i.e., Key Load session is terminated for any reason such as session timeout or CANCEL key is accidentally pressed).

**OS Expansion**

Initial versions of the merged OS for V<sup>x</sup>510 and V<sup>x</sup>610 fit within the existing allotment of seven sectors, however, a subsequent version is allotted nine sectors. The two extra sectors (128 KB) are largely empty and are reserved for future OS enhancements.

The OS expansion process is performed by two consecutive downloads. The first download launches a special application or “wizard” that when executed, prepares the terminal for the second download, which loads the new nine-sector OS into the memory. The wizard converts the flash memory image of the seven-sector OS into a nine-sector OS by shifting the contents of the flash file system up by two sectors and inserting a dummy 128-KB file at the start of the flash file system. By shifting the flash file system up in place, the wizard preserves the existing applications and data files, which do not need to be loaded again, once the expansion process is completed. The dummy file acts as a placeholder for the new OS sectors. Once the wizard’s task is done, the second download overlays the seven-sector OS and dummy file.

The following critical conditions must be met to ensure success of the process.

- 1** Ensure that at least 128 KB of flash is available before shifting flash memory. The new OS claims this amount of memory for its use. Although the wizard checks before moving data, the user is advised to ensure that this condition is met beforehand by, for example, performing batch settlements, deleting transaction batch files, and coalescing memory.
- 2** Do not perform a system restart and protect against power failures while shifting flash memory. Aborting the shift operation in the middle of the flash erase/write sequence and restarting it from the beginning leaves the flash file system in a corrupted state. The only recourse is re-initializing the flash file system. The user is advised to ensure that this condition is met beforehand by, for example, use of an UPS.
- 3** Do not perform flash file operations after shifting flash memory. The dummy 128 KB file must remain at the start of the flash file system when the nine-sector OS is downloaded.

At least 660 KB of SRAM must be available before downloading the nine-sector OS. The upgrade logic requires this amount of memory for temporary

use only. Although the wizard checks before moving data, the user is advised to ensure that this condition is met beforehand (as in condition #1).

**NOTE**



The downloaded files of the new OS, plus a 64 Kb sector work buffer, must fully reside in the SRAM before the files are decoded and programmed into flash by the upgrade logic. Otherwise, the sector work buffer will be the spare flash sector that resides at the end of the seven-sector OS. This leads to the corruption of sector 8 of the nine-sector OS as it is loaded into flash.

Once the wizard and the nine-sector Operating Systems are ready, the seven-sector merged Operating Systems will be withdrawn. From that point on, all new Operating Systems created for the V<sup>x</sup>510 and V<sup>x</sup>610 are nine sectors, which eliminate the chance of accidentally downgrading the OS from nine sectors to seven (which leaves the flash file system in a corrupted state).

## Upgrading to the Nine-Sector OS

There is only one fully verified upgrade sequence for the seven-sector OS in the old hardware.

### To upgrade to the Nine-Sector OS

- 1 Download the latest ECO-released seven-sector OS.

This ensures that only the latest flash memory drivers and OS upgrade code are used.

- 2 Download and execute the wizard.

- 3 Download the special ECO-released nine-sector OS. They are designed to ignore sectors eight and nine, and hence, does not suffer from any ill effects if the upgrade process corrupts sector eight.

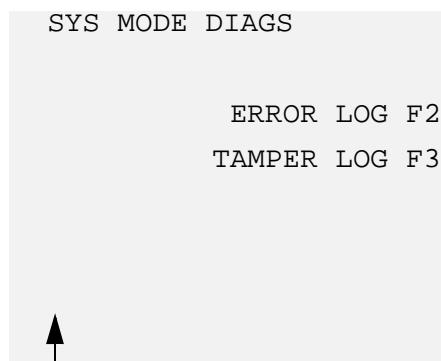
## Hardware Specifics for V<sup>x</sup>510 and V<sup>x</sup>610 Units

Below are hardware enhancements on the V<sup>x</sup>510 and V<sup>x</sup>610 terminals.

### DS3610 Chip

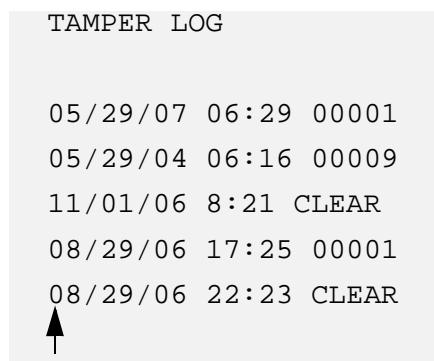
The PCI PED hardware on the V<sup>x</sup>510 and V<sup>x</sup>610 terminals uses the DS3610 chip, similar to the V<sup>x</sup>670 terminal. This replaces the old MAX6900 chip and the RNG and tamper circuits, which handles the real-time clock (RTC), GEK storage, tamper event logging, and random number generation (RNG).

Access to the tamper log is allowed via the System Mode.



**Figure 31      System Mode Diagnostics Screen**

Selecting F3 (Tamper Log) in the System Mode Diagnostics screen displays the Tamper Log Screen.



**Figure 32      Sample Tamper Log Screen**

If there is no data to display, the screen shows <EMPTY>.

### Expanded Memory Options

The OS is compiled with the mirror bit flash memory option enabled and a flash address limit of 16 MB.

### Reject Certain OS Downloads

In cases where the wrong OS version is loaded into the hardware, the Vx510/Vx610 terminals provide a software mechanism that rejects OS downloads that do not match the OS class currently executing.

In a merged Operating Systems scenario, the new OS works well in both the old and new hardware. However, the old OS works exclusively in the old hardware, and not in the new hardware. In this case, the new OS checks the firmware name to verify the correct code or pattern before accepting the download. The new OS uses "10" in the Major Software ID Field of the firmware name.

**Example** The OS version QA0008xx can be overwritten with the newer QA0010xx and QB0108xx with QB0110xx, but never the other way around. This method disallows downgrading a new OS to an old OS for use with old hardware. This is needed to block an attempt to downgrade the OS from nine sectors to seven.

In an unmerged Operating System, the old OS works exclusively with old hardware, and the new OS works exclusively with new hardware. This prevents the old (or new) OS from authenticating the new (or old) OS after the download files have been transferred. Thus, the download is rejected. Checking the OS firmware name is advantageous as it prevents cross-loading the wrong platform's OS (i.e., cannot load QB010xxx into Vx510 terminal). Unmerged Operating Systems are avoided because each set of changes to the OS source code leads to the creation and release of a pair of Operating Systems, each for the old and new hardware.





## APPENDIX D

# IPP Communications Packets

This appendix describes the required packet commands of the IPP for MS (Master Session) or DUKPT operations supported by the Verix V OS.

## Advanced Programming in IPP

For programmers familiar with Verix IPP6/IPP7 MS and DUKPT features, the Omni 5xxx IPP (VVIPP) has almost all of the same features. The differences are summarized in [Table 64](#).

**Table 64 Differences in Omni 5xxx 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. IPP7 is limited to at most three triple-length keys.

## Minor Differences by Packet

### <SI>0103<SO> PROM Checksum

The value of the checksum does not match IPP7 because Verix V 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 V-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

IPP7 can hold at most three triple-length keys. In VVIPP, all ten key locations can hold a single-, double-, or triple-length key.

### <STX>75..<ETX> DUKPT Accept and Encrypt PIN/Data Authentication Response

ANSI DUKPT MAC is only defined for 3DES DUKPT. VVIPP returns error code 8 if ANSI DUKPT MAC is requested when using 1DES DUKPT. IPP7 returns undefined results in this case.

## Packets

The packet set is similar to that used for external PIN pads, such as the PINpad 1000, however, unlike previous IPPs, the Omni 5xxx 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 Omni 5xxx 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.



Omni 5xxx 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 65 lists packets used in both MS and DUKPT sessions.

**Table 65      Common Packets**

Packet	Description
01	Interactive diagnostic routine
05	Transfer serial number
06	Request PIN pad serial number

**Table 65 Common Packets**

<b>Packet</b>	<b>Description</b>
09	Response to Packet 01
11	PIN pad connection test
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
M04	Read Permanent Unit Serial Number (IPP8 Emulation)

Table 66 lists packets supported by IPP for MS.

**Table 66 IPP Supported Packets for MS**

<b>Packet</b>	<b>Description</b>
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 67 lists packets supported by IPP for DUKPT.

**Table 67 IPP Supported Packets for DUKPT**

<b>Packet</b>	<b>Description</b>
90	Load initial key
91	Confirm initial key
75	Encrypt PIN/authentication data response
78	Encrypt PIN/authentication data test request
76	PIN entry test request
71	Response PIN entry test request of "76"
Z60	Accept and encrypt PIN request (VISA mode)
Z63	Accept and encrypt PIN, custom PIN entry requirements (VISA mode)
Z69	Accept and encrypt PIN/data authentication request (VISA mode)
73	Response PIN block
19	Select a DUKPT Engine (IPP8 Emulation)
25	Check the DUKPT Engine (IPP8 Emulation)

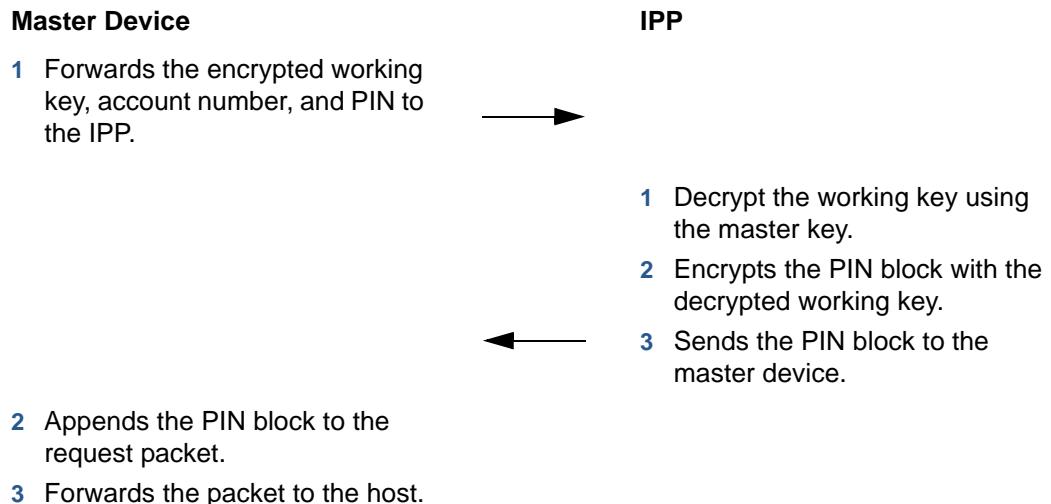
<p><b>Packet Acknowledgement and Timing</b></p>	<p>The IPP only responds to commands that have the proper packet format. The packet can be in the form of</p> <p>&lt;STX&gt;msg&lt;ETX&gt;[LRC]</p> <p>or</p> <p>&lt;SI&gt;msg&lt;SO&gt;[LRC]</p> <p>according to the specific command. The IPP returns &lt;ACK&gt; 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 &lt;STX&gt;&lt;ETX&gt; framing (for example, &lt;SI&gt;&lt;SO&gt;, &lt;SI&gt;&lt;ETX&gt;, or &lt;STX&gt;&lt;SO&gt;), &lt;ACK&gt; is returned if the LRC is valid; only the specified framing is processed.</p> <p>This rule also applies to &lt;SI&gt;&lt;SO&gt; 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.</p> <p>The response message from the IPP follows the &lt;ACK&gt; if the packet command has a response. However, the timing varies from different commands.</p>
<p><b>Encryption</b></p>	<p>There are two methods of PIN encryption in IPP:</p> <ul style="list-style-type: none"><li>• MS</li><li>• DUKPT</li></ul>

#### **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 33 illustrates an MS encryption session.



**Figure 33 Master Session Encryption Example**

#### DUKPT Method

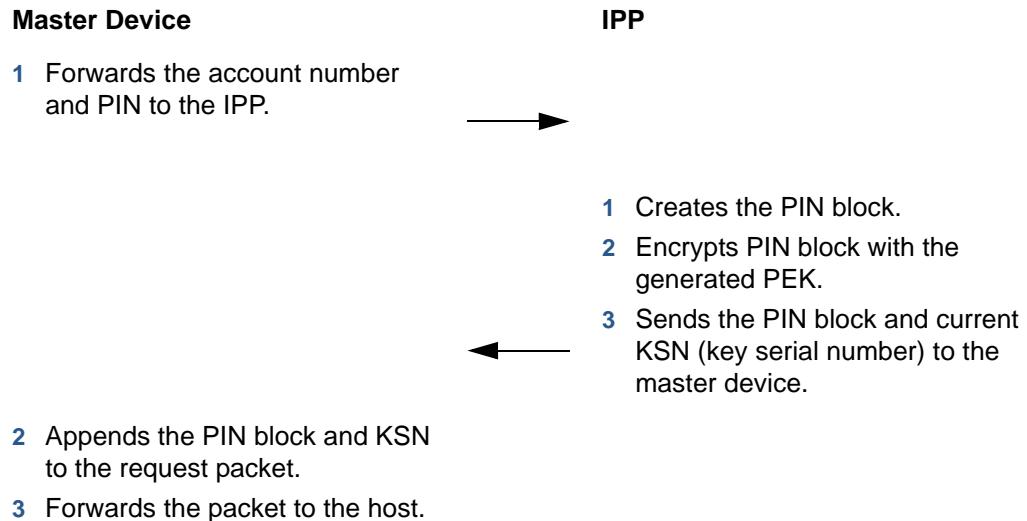
The IPP encrypts the customer's PIN according to the ANSI X9.8 standard and VISA's ANSI X9.24 DUKPT key management method, based on the ANSI X3.92 DES algorithm implemented in the IPP firmware.

Before actual operation, each IPP must be loaded with a unique initial KSN (key serial number) and a unique initial PEK (PIN Encryption Key). And the encryption counter of the IPP is set to zero. The initial PEK is generated by encrypting the initial KSN using appropriate derivation key.

The encryption per transaction of IPP during actual operation is as follows:

- 1 The master device sends an account number and a PIN to the IPP.
- 2 The IPP generates the clear-text PIN block using the account number and PIN.
- 3 Using the generated PEK based on the encryption counter which is updated after each transaction, the IPP do a special encrypt to the PIN block using the DES algorithm and PEK, then sends the encrypted PIN block with current KSN (the concatenation of the initial KSN and the encryption counter) to the master device.
- 4 The master device then appends the encrypted PIN block and current KSN to a request packet and forwards the completed request packet to the host.

Figure 34 illustrates the DUKPT method of encryption.



**Figure 34 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.

## **Restrict the Speed of the PIN Encryption Operation**

PIN encryption is limited to one per 30 seconds on average to deter an exhaustive PIN search. The algorithm is best explained in terms of tokens in a bucket.

A PIN encryption request is only accepted if there is a token in a bucket. A token is placed in the bucket every 30 seconds, with a maximum of 127 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 Omni 5xxx IPP. Omni 5xxx IPP7 is backward compatible with IPP6 and IPP5. Exceptions to this rule are noted.

## **GISKE**

GISKE (Global Interoperable Secure Key Exchange) is an industry standard key block format for secure transfer of secret keys between two devices that share a secret key. Both master and session keys can be in GISKE format. The GISKE KLK (Key Loading Key) is used to encrypt and authenticate master keys. Master keys can be remotely updated using this key. GISKE is designed for secure transfer of double- and triple-length 3DES keys. For more details on GISKE refer *GISKE Key Block Spec, VPN 22986*.

## **Key Management Switching**

The rules for key management switching (see [Packet 17: Set IPP7 Key Management Mode](#)) are shown in [Table 68](#).

### **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 68 Key Management Switching Rules**

Rules	To 1DES (VISA)	To 1DES (SPAIN) <sup>a</sup>	To Mixed Mode	To 3DES	To SM <sup>a</sup>
From 1DES <sup>b</sup> (VISA)	NC	E	NC	2/3K	E
From 1DES <sup>a</sup> (SPAIN)	E	NC	E	E	E
From Mixed mode <sup>c</sup>	1K	E	NC	2/3K	E
From 3DES <sup>d</sup>	E	E	E	NC	E
From SM <sup>a</sup>	E	E	E	E	NC
Key Mode	1DES and 3DES Key Usage Rules <sup>e</sup>				
1DES only <sup>b</sup>	<ul style="list-style-type: none"> <li>• Load and use of 1DES MS keys allowed<sup>f</sup></li> <li>• Load KLK allowed</li> <li>• Load 3DES master keys allowed</li> <li>• Use of 3DES master keys not allowed</li> <li>• Load 3DES session keys not allowed</li> <li>• Use of 3DES session keys not allowed</li> <li>• Key attributes verified<sup>g</sup>, except key usage = ‘AN’ – ANY is allowed</li> <li>• GISKE key block verified<sup>h</sup></li> </ul>				
Mixed mode <sup>c</sup>	<ul style="list-style-type: none"> <li>• Load and use 1DES or 3DES MS keys allowed</li> <li>• Load KLK allowed</li> <li>• 1DES master keys used for 1DES session keys</li> <li>• 3DES master keys used for 1DES and 3DES keys</li> <li>• Key attributes verified, except: key usage = ‘AN’ – ANY is allowed</li> <li>• GISKE key block verified</li> </ul>				
3DES only <sup>d</sup>	<ul style="list-style-type: none"> <li>• Load and use 3DES MS keys allowed</li> <li>• Load KLK allowed</li> <li>• Load 1DES master keys not allowed</li> <li>• Use of 1DES master keys not allowed</li> <li>• Load 1DES session keys not allowed</li> <li>• Use of 1DES session keys not allowed</li> <li>• Key attributes verified; no exceptions allowed</li> <li>• GISKE key block verified</li> </ul>				
<p>a. Spain and SM modes not supported in Verix V. Keys are erased as specified.</p> <p>b. Least secure mode.</p> <p>c. For transition period.</p> <p>d. Most secure mode.</p> <p>e. The key management register is set using <a href="#">Packet 17: Set IPP7 Key Management Mode</a>.</p> <p>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.</p> <p>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.</p> <p>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.</p>					

## 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 IPP7 key attributes, key version, and key length.

On erasure, the master key usage attribute is set to 0, the version is set to 0, and the length is set to 1DES.



Each key has its own key attribute register, key version register, and key length register.

The register listed in [Table 69](#) applies to 1DES master key, 3DES master key (GISKE), and KLK (GISKE). The original GISKE (ASCII-hex) key usage attribute value is saved in RAM (2 bytes).

**Table 69 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.



The key version comparison is only compared to the key it is replacing, not to any other keys.

Table 70 lists the key length register values for 1DES, 3DES, and three-key 3DES.

**Table 70 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 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 Omni 5xxx, all master key locations 0–9 can hold single-, double-, or triple-length DES keys. Omni 5xxx IPP7 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 71** 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 72** Packet 05 Format

Data Element	Characteristic	Comments
<SI>	1H	Shift in, value: 0Fh
Packet Type	2AN	Value: 05
[vvv]	3AN	PIN pad version number
[dddddd]	6N	Release date -- format: YYMMDD
[p]	1AN	Production facility code
[bb]	2AN	Production batch code
[nnnn]	4N	Serial # for group ID 0001–9999
<SO>	1H	Shift out, value: 0Eh
{LRC}	1H	Error check

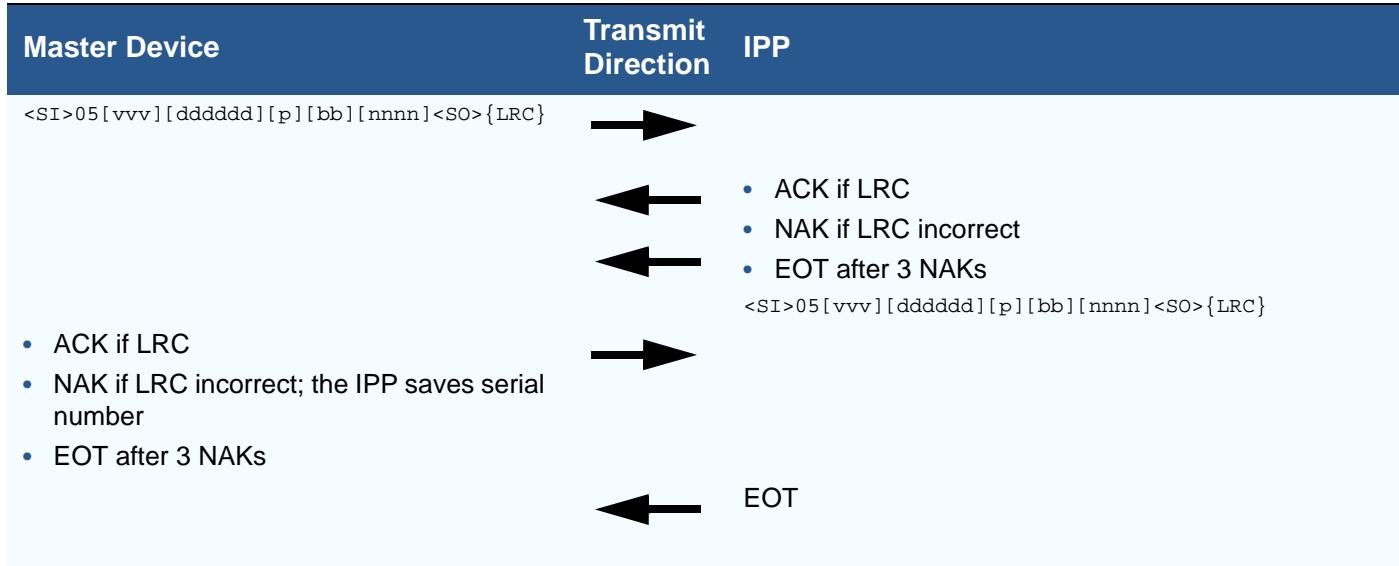
**Packet 05 Length:**

- MAX: 21 characters
- MIN: 21 characters

**Packet 05 Example:**

Set the IPP serial number to 246880401A001234:

```
<SI>05246880401A001234<SO>{LRC}
```

**Table 73** **Packet 05 Communication Protocol**
**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 74** **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 75** Packet 06 Format

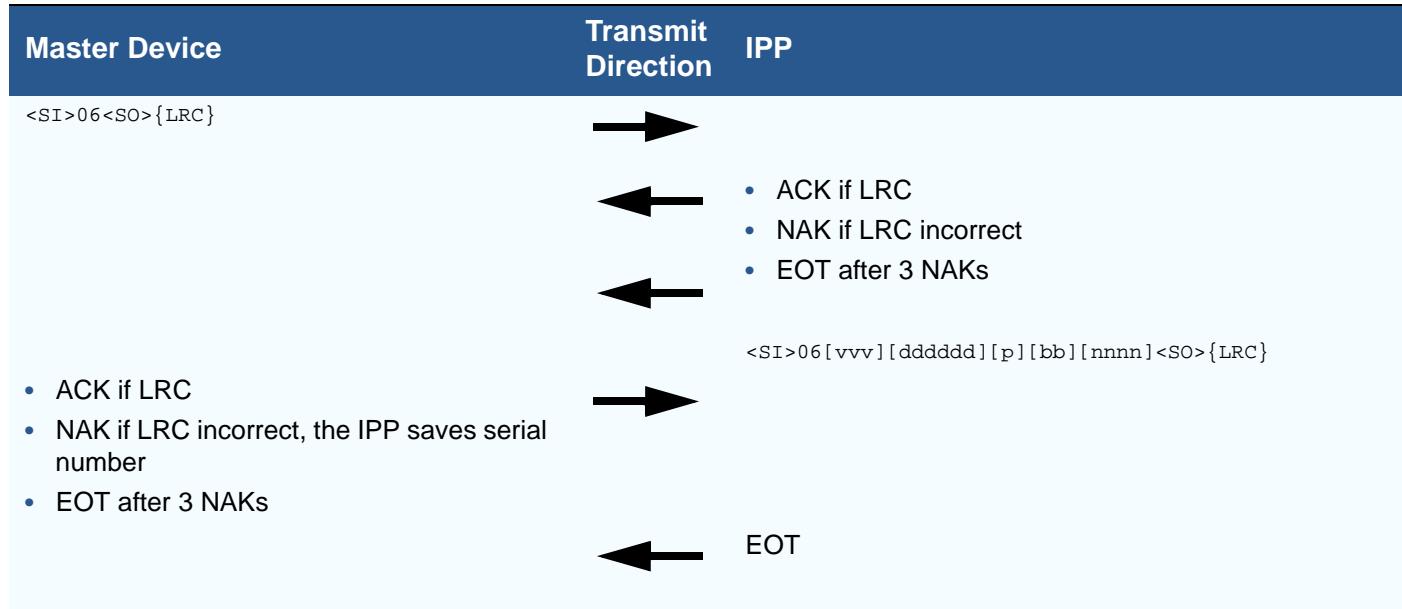
Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 06
[vvv]	3AN	PIN Pad Version Number
[dddddd]	6N	Release Date, Format: YYMMDD
[p]	1AN	Production Facility Code
[bb]	2AN	Production Batch Code
[nnnn]	4N	Serial # for Group ID 0001 - 9999
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

**Packet 06 Length:**

- MAX: 21 characters
- MIN: 21 characters

**Response Sample Packet**

&lt;SI&gt;06246880401A001234SO&gt;{LRC}

**Table 76** Packet 06 Communication Protocol**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 77](#).

**Table 77** Packets 09 and 14 Communication Protocol

Master Device	Transmit Direction	IPP
<b>00 Current Baud Rate</b>		
<SI>0100<SO>{LRC}	→	
	←	ACK/NAK/EOT
	←	<SI>14yyyy<SO>{LRC} where, yyyy indicates the current baud rate: <ul style="list-style-type: none"><li>• 1200</li><li>• 2400</li><li>• 4800</li><li>• 9600, or</li><li>• 19200</li></ul>
ACK/NAK/EOT	→	
	←	EOT to terminate process.
<b>01 RAM Test/One-Time</b>		
<SI>0101<SO>{LRC}	→	
	←	ACK/NAK/EOT
	←	<SI>14RAM TST BEGIN<SO>{LRC}
ACK/NAK/EOT	→	
	←	<SI>14RAM TST OK<SO>{LRC} or <SI>14BAD RAM<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process.

**Table 77** Packets 09 and 14 Communication Protocol (continued)

Master Device	Transmit Direction	IPP
<b>02 RAM Test/Continuous</b>		
<SI>0102<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>14RAM TST BEGIN<SO>{LRC}
ACK/NAK/EOT	→	
ACK	→	
	←	<SI>14RAM TST OK<SO>{LRC} or <SI>14BAD RAM<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process.
<b>03 PROM Checksum Test</b>		
<SI>0103<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>14xx<SO>{LRC} where, xx is the one-byte PROM internal checksum. There are two checksums inside the IPP:
		<ul style="list-style-type: none"> <li>The PROM checksum, which is 2-bytes long and is located at 7FFE/7FFF. This checksum is for manufacturing purposes.</li> <li>The PROM internal checksum.</li> </ul>
ACK/NAK/EOT	→	
	←	EOT to terminate process.

**Table 77** Packets 09 and 14 Communication Protocol (continued)

Master Device	Transmit Direction	IPP
<b>06 Serial Number Check</b>		
<SI>0106<SO>{LRC}	→	IPP6 and earlier
	←	ACK/NAK/EOT
	←	<SI>14xxxxxxxxxxxxxx<SO>{LRC} where, xxxxxxxxxxxxxxxx indicates the serial number of the IPP. Length is 16 digits, for example, 1234567890123456.
ACK/NAK/EOT	→	
	←	EOT to terminate process.
<b>07 UART Loopback Test</b>		
<SI>0107<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>09<SO>{LRC}
ACK/NAK/EOT	→	
<SI>09<SUB>PROCESSING<SO>{LRC}	→	
	←	ACK/NAK/EOT
	←	<SI>09<SUB>PROCESSING<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process.

**Table 77** Packets 09 and 14 Communication Protocol (continued)

Master Device	Transmit Direction	IPP
<b>08 IPP PROM Version Number</b>		
<SI>0108<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>14IPPx vvvvxxxx MM/YY<SO>{LRC} where: <ul style="list-style-type: none"><li>• vvv: 4-digit software ID number. For IPP5, 0PGP.</li><li>• 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.</li><li>• MM/YY: date of software. For example, MM/YY = 05/95 means the software was created May 1995.</li></ul>
ACK/NAK/EOT	→	
	←	EOT to terminate process.
<b>09 Reset IPP</b>		
<SI>0109<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>14RESET COMPLETE<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process. (The IPP restarts. Insert a delay before sending data to the IPP.)

**Table 77** Packets 09 and 14 Communication Protocol (continued)

Master Device	Transmit Direction	IPP
<b>10 Clear IPP</b>		
<SI>0110<SO>{LRC}	→	IPP7
	←	ACK/NAK/EOT
	←	<SI>14CLR COMPLETE<SO>{LRC}
ACK/NAK/EOT	→	
	←	EOT to terminate process.

**Packet 11: PIN Pad Connection Test**

The master device uses this packet to check the connection with the IPP. If the connection is good, the master device receives an 'ACK' from the IPP within 1 second. Else, it assumes that the IPP is not connected.

**Table 78** 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**

&lt;SI&gt;11&lt;SO&gt;{LRC}

**Table 79** Packet 11 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>11<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>

**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**

Omni 5xxx supports this packet but it has no effect. Verix V-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 RAM. After a power cycling memory test or loss of backup battery power, the baud rate setting is reset to the default.

**Table 80      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"> <li>• 1 = 1200 bps (default)</li> <li>• 2 = 2400 bps</li> <li>• 3 = 4800 bps</li> <li>• 4 = 9600 bps</li> <li>• 5 = 19200 bps</li> </ul>
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

**Packet 13 Length:**

- MAX: 6 characters
- MIN: 6 characters

**Table 81** Packet 13 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>13x<SO>{LRC}	→	
	←	ACK if LRC okay; NAK if LRC incorrect.
	←	<SI>14yyyy<SO>{LRC} where,  x = baud rate code      yyyy = string for selected baud rate <ul style="list-style-type: none"><li>• 1                          • 1200 (default)</li><li>• 2                          • 2400</li><li>• 3                          • 4800</li><li>• 4                          • 9600</li><li>• 5                          • 19200</li></ul> The baud rate code must be in the range 1–5; all other codes are ignored and directly echo [EOT] with the baud rate unchanged.
ACK/NAK	→	
	←	EOT to terminate process (the PIN pad uses the new baud rate accordingly).

### Packet 15: Set IPP Key Management Mode

This packet command changes the secret key management mode that the IPP uses for the transaction. The IPP supports two modes of secret key management:

- IPP5 or VISA MASTER SESSION+DUKPT mode

VISA MASTER SESSION+DUKPT mode covers MS and DUKPT and MAC process of standard ANSI MAC. The Omni 5xxx IPP does not include SEMP/4B mode, and erases keys when this mode is selected.

#### NOTE



In the Omni 5xxx 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 82 IPP Request Packet 15 Format**

Data Element	Field	Length	Comments
<SI>	Start of packet	1	Shift In.
15	Packet type	2	Control Character- 0Fh 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"> <li>• “SPAIN” – Spain SEMP/4B mode</li> <li>• “VISA” – IPP mode</li> <li>• Other characters – no change</li> </ul>
[SO]	End of packet	1	Shift Out.
{LRC}	Block code check	1	Control Character- 0Eh Error Check Character

**Response Packet Format**

&lt;SI&gt;15[Key Code]&lt;SO&gt;{LRC}

**Table 83 IPP Response Packet 15 Format**

Data Element	Field	Length	Comments
<SI>	Start of packet	1	Shift In.
15	Packet type	2	Control Character- 0Fh 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"> <li>• “SPAIN” – Spain SEMP/4B mode</li> <li>• “VISA” – IPP mode</li> <li>• Other characters – no change</li> </ul>
[SO]	End of packet	1	Shift Out.
{LRC}	Block code check	1	Control Character- 0Eh Error Check Character

If the terminal receives the response without any errors, then it sends ACK to the IPP. The IPP then sends <EOT> { ASCII CODE is 04 } to terminate the session.

**Packet 15 Example:**

```
<SI>15SPAIN<SO>{LRC}      ( set Spain SEMP/4B mode)
<SI>15VISA<SO>{LRC}       ( set MS / DUKPT mode)
```

**NOTE**

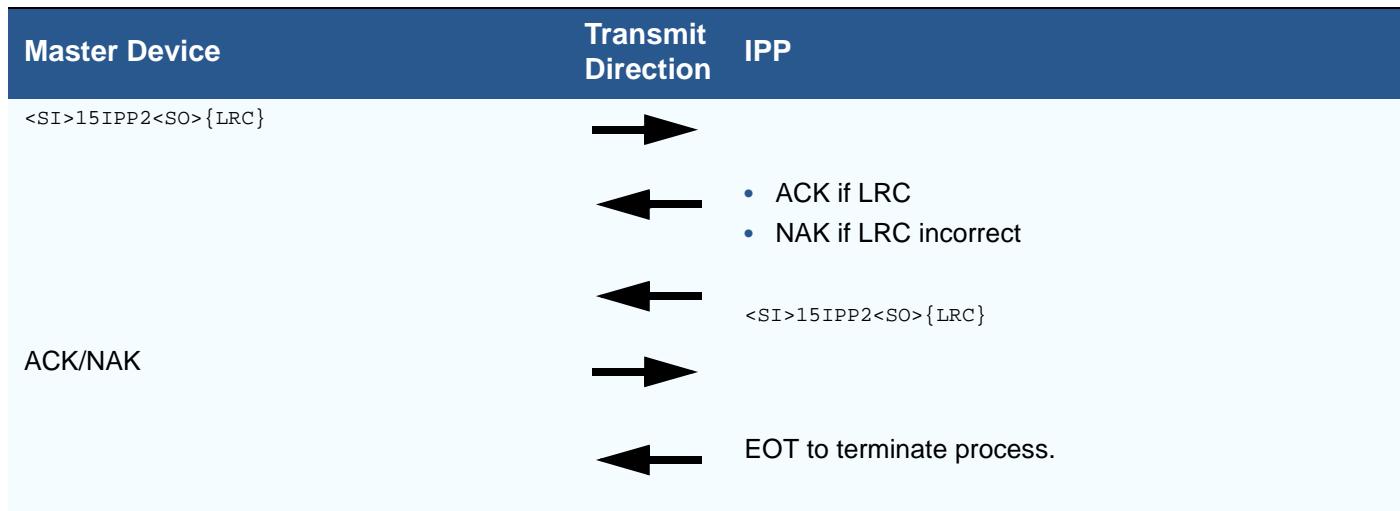
In IPP6, the following packet 15 variation is included for compatibility purposes only, and does not result in the key information being erased.

**Table 84**      **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 85**      **Packet 15 Communication Protocol****Packet 17: Set IPP7 Key Management Mode**

This packet sets or clears a number of control switches in the key management options register for IPP7 key management configuration. IPP7 supports the following additional functions (as compared to IPP6):

- Triple DES (3DES) DUKPT support
- GISKE MS Key support
- Zero (0) key support

Note that the new MAC alternatives apply only when GISKE is active, and are selected by key attributes and not the key management switch.

For compatibility, the default key management mode for IPP7 is set to IPP5 mode (MS- DUKPT or single DES mode). Once a new key management scheme is selected, it is retained during power cycles.

Setting a new mode causes the IPP7 to erase all existing keys or non-volatile security values stored for secure messaging.

**Incoming Packet Format:**

<SI>17[KMM][PINER]<SO>{LRC}

**Table 86      Packet 17 Format**

Data Elements	Characteristics	Comments																															
<SI>	1H	Shift In, value: 0Fh																															
Packet Type	2AN	Value: 17																															
Key Management Mode	2AH	The two ASCII hex digits are concatenated big-endian, to produce a single control byte. The key management mode register (8 bits) in IPP7 is as follows:																															
[KMM]		<table border="1"> <thead> <tr> <th>Bit</th><th>0</th><th>1</th><th>2</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1DES MS (default)</td></tr> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>Mixed mode (1DES and 3DES GISKE)</td></tr> <tr> <td>0</td><td>1</td><td>0</td><td>0</td><td>3DES GISKE MS</td></tr> <tr> <td>1</td><td>1</td><td>0</td><td>0</td><td>Secure messaging (not supported in Omni 5xxx).</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Bit 3</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0</td><td>1DES DUKPT (default)</td></tr> <tr> <td>1</td><td>3DES DUKPT</td></tr> </tbody> </table>	Bit	0	1	2	Description	0	0	0	0	1DES MS (default)	1	0	0	0	Mixed mode (1DES and 3DES GISKE)	0	1	0	0	3DES GISKE MS	1	1	0	0	Secure messaging (not supported in Omni 5xxx).	Bit 3	Description	0	1DES DUKPT (default)	1	3DES DUKPT
Bit	0	1	2	Description																													
0	0	0	0	1DES MS (default)																													
1	0	0	0	Mixed mode (1DES and 3DES GISKE)																													
0	1	0	0	3DES GISKE MS																													
1	1	0	0	Secure messaging (not supported in Omni 5xxx).																													
Bit 3	Description																																
0	1DES DUKPT (default)																																
1	3DES DUKPT																																

**Table 86**      **Packet 17 Format** (continued)

Data Elements	Characteristics	Comments
<b>Bit 4 Description</b>		
0	Zero key support off (default)	
1	Zero key support on	
<b>Bit 5 Description</b>		
0	Zero GISKE session key support off (default)	
1	Zero GISKE session key support on	
<b>Bit 6 Description</b>		
0	N/A	
1	Clear all MS master keys and KLK	
<b>Bit 7 Description</b>		
0	MAC empty working key support off (default)	
1	MAC empty working key support on	

**Table 86** 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.
<b>Note:</b> This field was added for IPP8 emulation.		<b>Bit 0 ( DUKPT Engine "1") Description</b>
	0	1DES DUKPT - Default
	1	3DES DUPKT
		<b>Bit 1 ( DUKPT Engine "2") Description</b>
	0	1DES DUKPT - Default
	1	3DES DUPKT
		Bit 2 ~ 3
		-----
		Reserved
		Example
	<b>Engine=</b>	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 87** Packet 17 Set IPP Key Management Mode

Master Device	Transmit Direction	IPP
<SI>17[KMM][PINER]<SO>{LRC}		
		<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>
		<SI>17[KMM][PINER]<SO>{LRC}
<ul style="list-style-type: none"> <li>ACK if LRC and key management echo is OK</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> <li>EOT if LRC is correct, but key management echo is not OK.</li> </ul>		
		EOT to terminate process. The IPP saves the new key management accordingly.

**Notes**

- 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 (not SPAIN).
- 2 When the IPP receives packet 17 to change KM modes (for example, to 3DES or SM mode), the master device must know the new specification and functions associated with the IPP. If the IPP is not in the “old single DES” mode (IPP5/6), the IPP ignores packet 15 and will not allow itself to be switched to SPAIN mode unless the KMM register is set to IPP5/6 mode.
- 3 SPAIN mode is a submode of the single DES (IPP5/6) KMM register setting. A change from 1DES to 3DES or mixed mode will disable SPAIN mode.
- 4 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.
- 5 The master device must delay for at least 500 ms before sending a packet to the IPP when the KMM is switched from IPP7 to SM or from SM to IPP7.
- 6 Switching from SM to IPP7 mode causes a factory reset. The IPP clears the contents of RAM and communication to the IPP is reset to the default, 1200 baud, 7 data bits, even parity, and 1 stop bit (7E1).
- 7 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 IPP7 Key Management Mode

Checks the setting in the IPP7 key management options register.

### Request Packet Format

<SI>18<SO>{LRC}

**Table 88      Packet 18 Format**

Data Elements	Characteristics	Comments
<SI>	1H	Shift In, value: 0Fh
Packet Type	2AN	Value: 18

**Table 88** Packet 18 Format (continued)

Data Elements	Characteristics	Comments
Key Management Mode [KMM]	2AH	The two digits are concatenated big-endian, to produce a single control byte. The key management mode register (8 bits) in IPP7 is as follows:
		<b>Bit</b>
	<b>0    1    2    Description</b>	
	0    0    0    Old single DE	
	1    0    0    Mixed mode (1DES and 3DES GISKE).	
	0    1    0    3DES GISKE MS	
	1    1    0    Secure messaging (not supported in OmniEvent)	
		<b>Bit 3   Description</b>
	0    1DES DUKPT	
	1    3DES DUKPT	
		<b>Bit 4   Description</b>
	0    Zero key support off	
	1    Zero key support on.	
		<b>Bit 5   Description</b>
	0    Zero GISKE session key support off	
	1    Zero GISKE session key support on	
		<b>Bit 6   Description</b>
	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).	
		<b>Bit 7   Description</b>
	0    MAC empty working key support off.	
	1    MAC empty working key support on.	

**Table 88** 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.
<b>Note:</b> This field was added for IPP8 emulation.		<b>Bit 0 ( DUKPT Engine "1") Description</b>
	0	1DES DUKPT - Default
	1	3DES DUPKT
		<b>Bit 1 ( DUKPT Engine "2") Description</b>
	0	1DES DUKPT - Default
	1	3DES DUPKT
		Bit 2 ~ 3
		-----
		Reserved
		Example:
	Engine=	1      2
	DEMF = 0x30	0      1DES    1DES
	0x31	1      3DES    1DES
	0x32	2      1DES    3DES
	0x33	3      3DES    3DES
<SO>	1H	Shift Out, value: 0Eh
{LRC}	1H	Error Check

**Packet 18 Length:**

- MAX: 8 characters
- MIN: 8 characters

**Table 89** Packet 18 Check IPP7 Key Management Mode

Master Device	Transmit Direction	IPP
<SI>18<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>
	←	
• ACK/NAK	→	<SI>18[KMM][PINER]<SO>{LRC}
	←	EOT to terminate process.

**Packet 18 Examples:**

The following examples show the response packet,  
 <SI>18[KMM][PINER]<SO>{LRC} from the IPP.

- 1 1DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18000<SO>{LRC}

- 2 Mixed MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18010<SO>{LRC}

- 3 3DES MS mode, zero key support off, zero GISKE session key support off, and 1DES DUKPT mode:

<SI>18020<SO>{LRC}

- 4 1DES MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18080<SO>{LRC}

- 5 Mixed MS mode, zero key support off, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18090<SO>{LRC}

- 6 3DES MS mode, zero key support off, zero GISKE session support off, and 3DES DUKPT mode:

<SI>180A0<SO>{LRC}

- 7 1DES MS mode, zero key support on, zero GISKE session support off, and 1DES DUKPT mode:

<SI>18100<SO>{LRC}

- 8** Mixed MS mode, zero key support on, zero GISKE session support on, and 1DES DUKPT mode:

<SI>18310<SO>{LRC}

- 9** 3DES MS mode, zero key support off, zero GISKE session key support on, and 1DES DUKPT mode:

<SI>18220<SO>{LRC}

- 10** 1DES MS mode, zero key support on, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18180<SO>{LRC}

- 11** Mixed MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

<SI>18390<SO>{LRC}

- 12** 3DES MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

<SI>182A0<SO>{LRC}

- 13** 1DES MS mode, zero key support on, zero GISKE session key support off, and 3DES DUKPT mode:

<SI>18580<SO>{LRC}

- 14** Mixed MS mode, zero key support on, zero GISKE session key support on, and 3DES DUKPT mode

<SI>18790<SO>{LRC}

- 15** 3DES MS mode, zero key support off, zero GISKE session key support on, and 3DES DUKPT mode:

<SI>186A0<SO>{LRC}

### Packet Z60: Accept and Encrypt PIN (VISA Mode)

On receipt of the Z60 packet, Omni 5xxx 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 90      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

**Table 90** **Packet Z60 Format** (continued)

Data Elements	Characteristics	Comments
[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"> <li>• 16AH – 1DES, key-only format</li> <li>• 120AH – GISKE key block format. For more details on GISKE refer <i>GISKE Key Block Spec</i>, VPN 22986.</li> <li>• (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.</li> <li>• (1DES only) If zero key support mode is disabled, the passed key is used regardless of the encrypted key value.</li> <li>• Zero GISKE session key support for GISKE key block format communication protocol. (see <a href="#">Using a Session Key</a>).</li> <li>• 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.</li> </ul>
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error Check

### **Packet Z60 Length:**

- Maximum: 147 characters
  - Minimum: 32 characters

## **Sample Packet Z60 for MS 1DES:**

<STX>Z60.0123456789012345678<FS>0123456789012345<ETX>{ LRC }

## **Sample Packet Z60 for DUKPT:**

<STX>Z60.0123456789012345678<FS>DUKPT ENCRYPTION<ETX>{LRC}

## **Sample Packet Z60 for MS GISKE:**

### Errors returned by write()

Some packet format errors are caught when the packet is written to the IPP. In this case, `write()` returns -1 and `errno` set. The packet is not ACKed or NAKed, and no response packet returns.

Z60 Format Error	errno
No <FS>	EINVAL
PIN entry too fast.	EACCESS
See <a href="#">Restrict the Speed of the PIN Encryption Operation</a> .	

### Packet Z63: Accept and Encrypt PIN— Custom PIN Entry Requirements (VISA Mode)

On receipt of the Z63 packet, Omni 5xxx reads the user's PIN from the keyboard, echoing to the display [echo char] for each digit accepted. The PIN length can be between [min len] and [max len] digits, inclusive, or 0 if the [NULL allowed flag] is set. There are two variations of these request packets: MS and DUKPT.

#### Sample Packet Z63 for MS

```
Request    <STX>Z63.[acct num]<FS>[working key][min len][max len]
           [NULL allowed flag][echo char]<ETX>{LRC}
Response   <STX>71.0[PIN len][PIN block format]
           [encrypted PIN block]<ETX>{LRC}
```

#### Sample Packet Z63 for DUKPT

```
Request    <STX>Z63.[acct num]<FS>DUKPT ENCRYPTION[min len][max len]
           [NULL allowed flag][echo char]<ETX>{LRC}
Response   <STX>73.00000[key serial number][encrypted PIN block]<ETX>{LRC}
```

Note that [min len] and [max len] are two-character ASCII digits that represent values between 04 and 12, inclusive. [max len] should not be less than [min len] that is:

$$04 \leq [\text{min len}] \leq [\text{max len}] \leq 12$$

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 91** Packet Z63 Format

Data Elements	Characteristics	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	3AN	Value: Z63
Packet Delimiter	1A	Value: (.), 2Eh [aaa...aa] 8-19N Card account number
<FS>	1H	Field Separator; Value: 1Ch
[www...www]	16AH or 120AH	Encrypted working key or (encrypted session key) DUKPT. DUKPT ENCRYPTION means DUKPT is ENCRYPTION selected. Otherwise, it is the working key of MS encrypted under the master key. GISKE is used here for 3DES session key support.  Size of [www...www] indicates which packet format is used: <ul style="list-style-type: none"> <li>• 16AH: 1DES, key-only format</li> <li>• 120AH: GISKE key block format. For more details on GISKE refer <i>GISKE Key Block Spec</i>, VPN 22986.</li> <li>• (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.</li> <li>• (1DES only) If zero key support mode is disabled, the passed key is used regardless of the encrypted key value.</li> <li>• Zero GISKE session key support for GISKE key block format communication protocol. (see Using a Session Key).</li> <li>• 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.</li> </ul>
[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 the Speed of the PIN Encryption Operation](#).

**Packet M04: Read Permanent Unit Serial Number**

**NOTE**

This packet is added for IPP8 emulation.

**Request Packet Format**

<SI>M04<SO>{LRC}

**Table 92** **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 93** **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 94** Packet M04 Communication Protocol

Master Device	Transmit Direction	IPP
<SI>M04<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> <li>ACK if LRC okay.</li> <li>NAK if LRC incorrect (EOT after 3 NAKs).</li> </ul>
	←	<SI>M04[PUSN]<SO>{LRC}
• ACK if LRC okay • NAK if LRC incorrect (EOT after 3 NAKs).	→	
	←	EOT terminates session.

## MS-Specific Packets

The following packets are specific to MS 1DES and 3DES operations. The default mode for the IPP at power up is MS 1DES.

### Packet 02: Transfer Master Key

The master device uses this packet to send a master key to the IPP. The response from the IPP to the master device depends on the value of the key management option register.

**Table 95 MS Packet 02 Format**

Data Element	Characteristic	Comments
<STX>	1H	Shift In, Value: 0Fh
Packet Type	3AN	Value: 02
[n]	1N	Address or key usage identifier.  1DES: <ul style="list-style-type: none"><li>• Master key address is 0-9</li></ul> 3DES: <ul style="list-style-type: none"><li>• Master key address for double- or triple-length keys is 0-9, 'F<sup>a</sup></li></ul>
[hhh...hh]	16H	Master key in ASCII. <ul style="list-style-type: none"><li>• 16Ah: 1DES mode for single-length key</li><li>• 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.</li></ul>
<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 96**.

**Table 96 Default Key Attributes**

Key Attributes	Value	Hex	Definition
Key usage	AN	0x41, 0x4E	Any; no special restrictions
Key Algorithm	D	0x44	DES

**Table 96 Default Key Attributes**

Key Attributes	Value	Hex	Definition
Key mode of use	N	0x4E	No special restrictions
Key version	00	0x30, 0x30	version = zero
Key length	1	0x31	single-length key

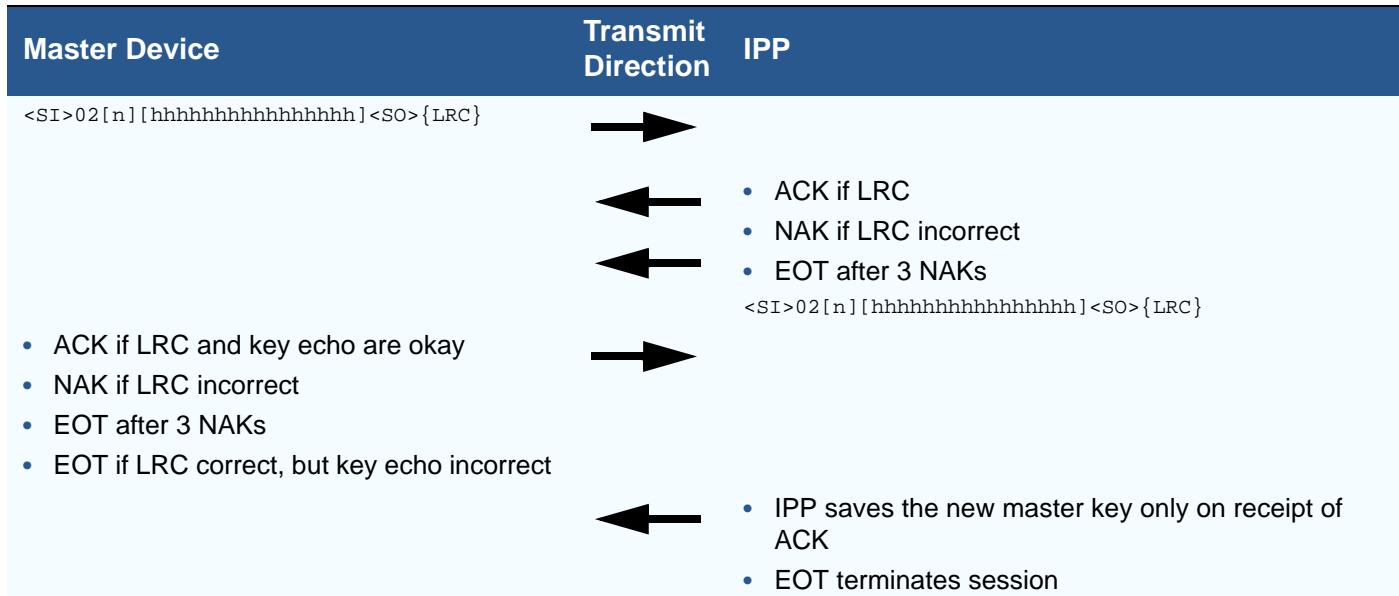
The single-DES communication protocol between the master device and the IPP as follows:

#### **Sample Packet 02 in Key-only Format**

This sample packet requests the IPP to load master key 0123456789ABCDEF into location '0'.

```
<SI>0200123456789ABCDEF<SO>{LRC}
```

**Table 97 Packet 02 Key-Only Communication Protocol**



#### **GISKE Key Block Format**

3DES communication protocol between the master device and the IPP is as follows:

#### **Sample Packet 02 in GISKE Key Block Format:**

This sample packet requests that the IPP load the 120-byte GISKE key block into address 0

```
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF012345678901234567890123:"
```

```
<SI>
02000123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234
56789ABCDEF0123456789ABCDEF012345678901234567890123<SO>{LRC}
```

**Table 98** Packet 02 Response Format

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 02
[n]	1N	Response code (0–7): <ul style="list-style-type: none"> <li>• 0 = No error</li> <li>• 1 = Error: IPP in incorrect KM mode</li> <li>• 2 = Error: incorrect key usage, mode of use, algorithm, or key length</li> <li>• 3 = Version error</li> <li>• 4 = Error: KLK already exists or new KLK was not encrypted from the previous KLK</li> <li>• 5 = GISKE decryption or MAC error</li> <li>• 6 = Error: master key address does not match the address range described in <a href="#">Packet 02: Transfer Master Key</a></li> <li>• 7 = Error: inappropriate master key addressing</li> </ul>
<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:**

&lt;SI&gt;020&lt;SO&gt;{LRC}

**Table 99** 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"> <li>• ACK if LRC</li> <li>• NAK if LRC incorrect</li> </ul>
	←	<ul style="list-style-type: none"> <li>• EOT after 3 NAKs</li> </ul>
	→	<SI>02[n]<SO>{LRC}
<ul style="list-style-type: none"> <li>• ACK if LRC, no errors, and key echo okay</li> <li>• NAK if LRC incorrect</li> <li>• EOT after 3 NAKs</li> <li>• EOT if LRC correct, but key echo incorrect</li> </ul>	→	<ul style="list-style-type: none"> <li>• IPP saves new key only on receipt of ACK</li> <li>• EOT terminates session</li> </ul>

## 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 100 MS Packet 04 Format**

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 04
[a]	1N	Master Key address: 0–9 KLK: F
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

### MS Packet 04 Length:

- MAX: 6 characters
- MIN: 6 characters

### Packet 04 Communication Protocol

Packet 04 has two types of communication format: key-only and GISKE key block format. The communication format depends on the IPP key management setting and the length of the key at address [a]. The use of the communication protocol is as follows:

IPP Key Management Setting	Key Length at Address [a]	Communication Protocol Used
1DES mode	1DES (single-length key)	Key-only format (IPP5/IPP6)
	3DES (single-, double-, or triple-length key)	GISKE key block format <sup>a</sup>
Mixed or 3DES mode	1DES (single-length key)	GISKE key block format
	3DES (single-, double-, or triple-length key)	GISKE key block format

- a. If a single-, double-, or triple-length key stored in the IPP contains the key attribute information described in the GISKE specification, this indicates that master device must be compatible with the IPP7 (3DES) specification. Therefore, the master device can understand the GISKE key block format communication protocol.

### Packet 04 Key-only Format

To check if the master key is loaded at address 5, the request sample packet 04 for key-only format is

```
<SI>045<SO>{LRC}
```

**Table 101 Response Packet 04 Key-only Format**

Data Element	Characteristic	Comments
<SI>	1H	Shift In, Value: 0Fh
Packet Type	2AN	Value: 04
[r]	1AN	Response code: • 0 = No master key at address [a] • F = Master key present at address [a]
<SO>	1H	Shift Out, Value: 0Eh
{LRC}	1H	Error Check

**Table 102 Response Packet 04 Key-only Format Communication Protocol**

Master Device	Transmit Direction	IPP
<SI>040<SO>{LRC}	→	
	←	<ul style="list-style-type: none"> <li>ACK if LRC okay</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> <li>PIN pad checks requested address [a].</li> </ul>
	←	
• ACK if LRC okay • NAK if LRC incorrect • EOT after 3 NAKs	→	<SI>04[r]<SO>{LRC}
	←	EOT

**Response Packet 04 Key-only Format Length:**

- MAX: 6 characters
- MIN: 6 characters

**Response Packet 04 Key-only Format Example:**

&lt;SI&gt;040&lt;SO&gt;{LRC}

**Packet 04 GISKE Key Block Format**

The GISKE key block format communication protocol is used when the IPP is in mixed or 3DES mode. The original GISKE (ASCII-hex) key usage attribute value is saved in RAM (2 bytes).

&lt;SI&gt;042&lt;SO&gt;{LRC}

**Table 103 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"> <li>• 0 = No master key at address [a]</li> <li>• F = Master key present at address [a]</li> </ul>
Key Usage Attribute (KUA)	2AH	Only when master key is present at address [a]: <ul style="list-style-type: none"> <li>• AN: ANY: The key is available in the IPP, but was not loaded using GISKE format.</li> <li>• D0: Data encryption</li> <li>• I0: IV</li> <li>• T0: control vector</li> <li>• K0: key encryption or wrapping</li> <li>• G0: MAC generation</li> <li>• M0: MAC verification</li> <li>• P0: PIN encryption</li> <li>• V0: PIN verification</li> <li>• C0: CVK (card verification key)</li> <li>• B0: BDK (base derivation key [A])</li> <li>• 00: ISO 9797-1 MAC algorithm 1 (1–56 bits)</li> <li>• 10: ISO 9797-1 MAC algorithm 1 (1–112 bits)</li> <li>• 20: ISO 9797-1 MAC algorithm 2 (2–112 bits)</li> <li>• 30: ISO 9797-1 MAC algorithm 3 (3–112 bits)</li> <li>• 40: ISO 9797-1 MAC algorithm 4 (4–112 bits)</li> <li>• 50: ISO 9797-1 MAC algorithm 5 (5–56 bits)</li> <li>• 60: ISO 9797-1 MAC algorithm 5 (5–112 bits)</li> </ul>
Algorithm	1AH	(optional) Only if the master key is present at address [a]. The value is stored in the Key Attributes register. <ul style="list-style-type: none"> <li>• D: DES [0]</li> <li>• R: RSA [1]</li> <li>• A: AES [2]</li> <li>• S: DSA [3]</li> <li>• T: TDES [4]</li> <li>• U: Unknown [5]</li> <li>• E: Elliptic Curve [6]</li> <li>• [7]–[F] = Reserved</li> </ul> <p><b>Note:</b> To save storage space in RAM, 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 103 Response Packet 04 GISKE Key Block Format (continued)**

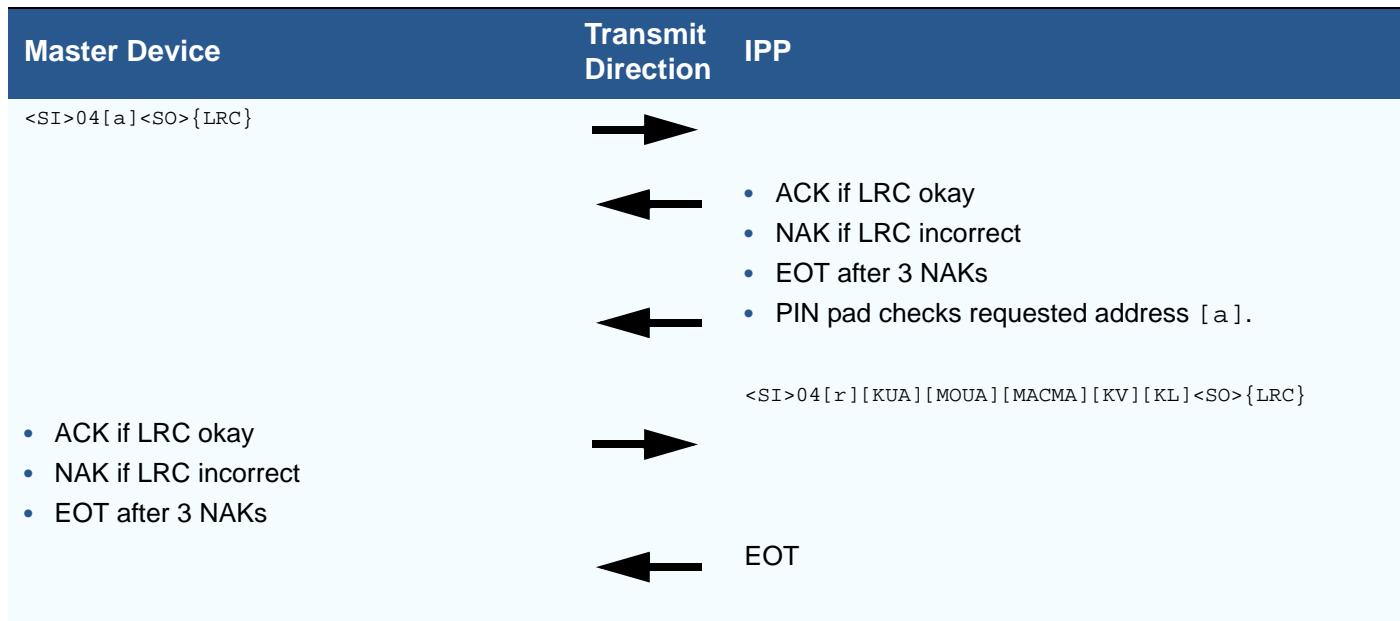
Data Element	Characteristic	Comments
Mode of Use Attribute (MOUA)	1AH	(optional) Only if the master key present at address [a]. The value is stored in the key attributes register. <ul style="list-style-type: none"><li>• N: No special restrictions [0]</li><li>• E: Encryption only [1]</li><li>• D: Decryption only [2]</li><li>• S: Signature only [3]</li><li>• O: IV [4]</li><li>• G: MAC generate [5]</li><li>• V: MAC verify [6]</li><li>• C: Calculate = generate or verify) [7]</li><li>• [6]–[F]: Reserved</li></ul> <p><b>Note:</b> To save storage space in RAM, the mode of use 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 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><b>Note:</b> 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"><li>• 1: single-length key</li><li>• 2: double-length key</li><li>• 3: triple-length key</li></ul> <p><b>Note:</b> 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 104 Response Packet 04 GISKE Block Format Communication Protocol**
**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 105 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:**

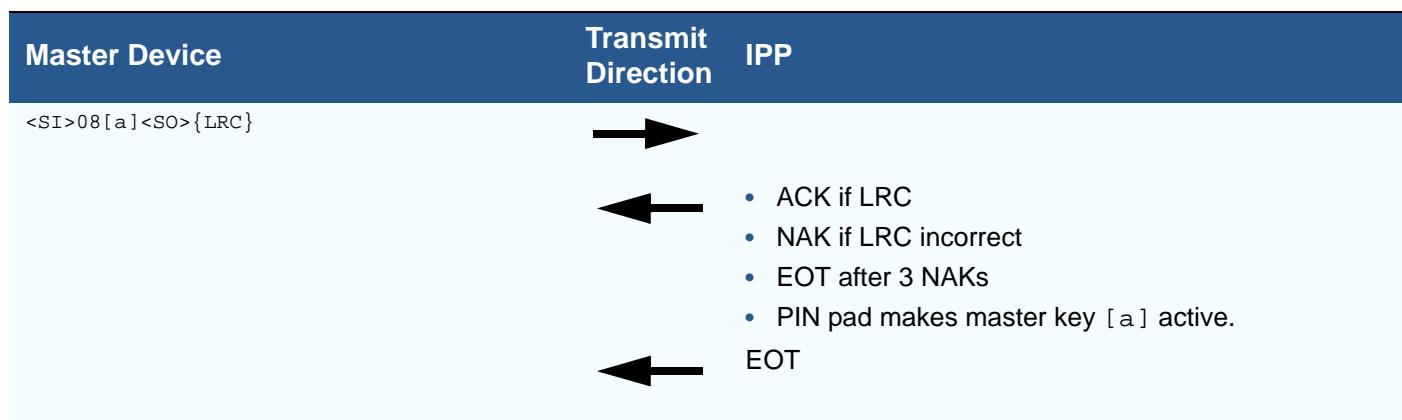
- MAX: 6 characters
- MIN: 6 characters

**MS Packet 08 Example:**

To select Master Key 7:

```
<SI>087<SO>{LRC}
```

**Table 106 MS Packet 08 Communication Protocol**



#### Notes

- 1 The 1DES and 3DES key usage rules (see [Rules for Loading the Master Key \(MS only\)](#)) applies when selecting a master key. If the selecting master key is not available or is not applicable due to the 1DES or 3DES key usage rule, no response is returned to the master device.
- 2 If the master key address does not contain any key, the IPP still sets the currently selected key as the active master key due to a backward compatibility requirement.

#### MS Packet 71: Transfer PIN Block

Response packet to [Packet Z60: Accept and Encrypt PIN \(VISA Mode\)](#) and [Packet Z63: Accept and Encrypt PIN–Custom PIN Entry Requirements \(VISA Mode\)](#). The IPP encrypts the formatted clear-text PIN block and sends the cipher-text PIN block to the master device.

**Table 107 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 108 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"> <li>• 1 = no master key</li> <li>• 2 = account or working key error</li> <li>• 3 = PIN too long.</li> <li>• 4 = PIN too short / non-decimal digits in PIN.</li> <li>• 5 = PIN pad used as DUKPT<sup>a</sup></li> <li>• 6 = Master key attributes error</li> <li>• 7 = KOF/GISKE working key attributes error, key attributes: key usage, algorithm, mode of use, key version, or key length</li> </ul>
<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 IPP7: 1DES or 3DES. All keys associated with DUKPT are erased when switching between 1DES and 3DES DUKPT modes.

### Packet 19: Select a DUKPT Engine

The application sends this packet to the IPP to select one of the DUKPT engines ("0", "1", or "2"). It is recommended that the application always send this packet first before sending a DUKPT packet (eg. packet Z60, Z63, 76, Z69 and 90).

**NOTE**



This packet was added for IPP8 emulation.

**Table 109 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

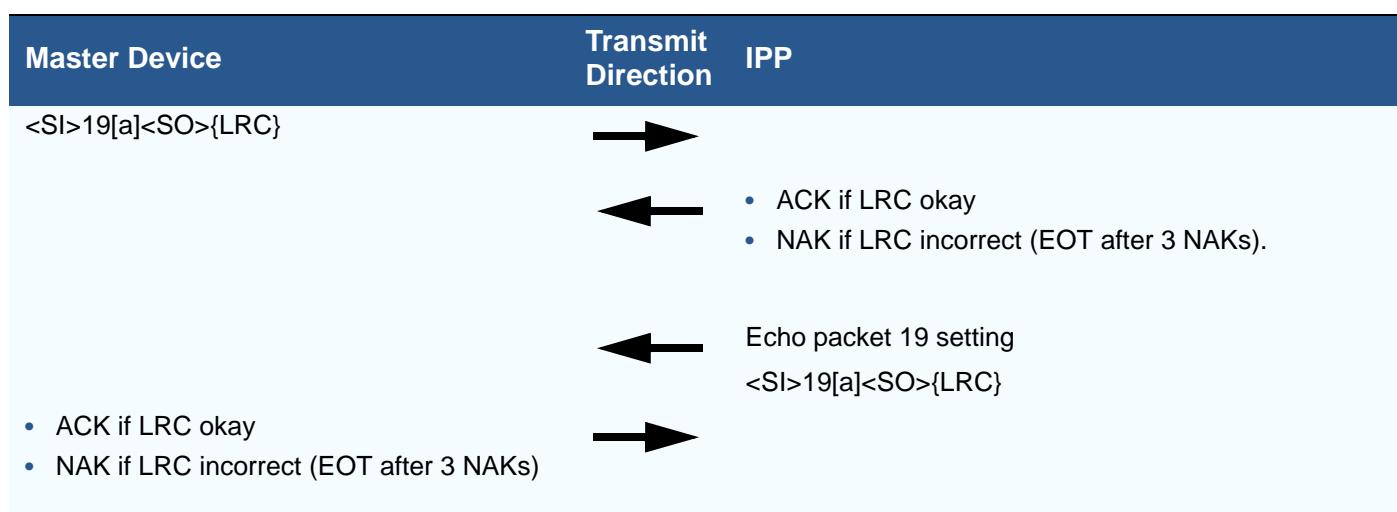
Minimum: 6 characters

**Sample Packet:**

To select second DUKPT Engine:

<SI>192<SO>{LRC}

**Table 110 DUKPT Packet 19 Communication Protocol**



**Table 110 DUKPT Packet 19 Communication Protocol (continued)**

Master Device	Transmit Direction	IPP
		...
		IPP changes DUKPT engine
		...
		EOT to terminate process.



- 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").



This packet is added for IPP8 emulation.

***Request Packet Format***

<SI>25<SO>{LRC}

**Table 111 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 112 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:

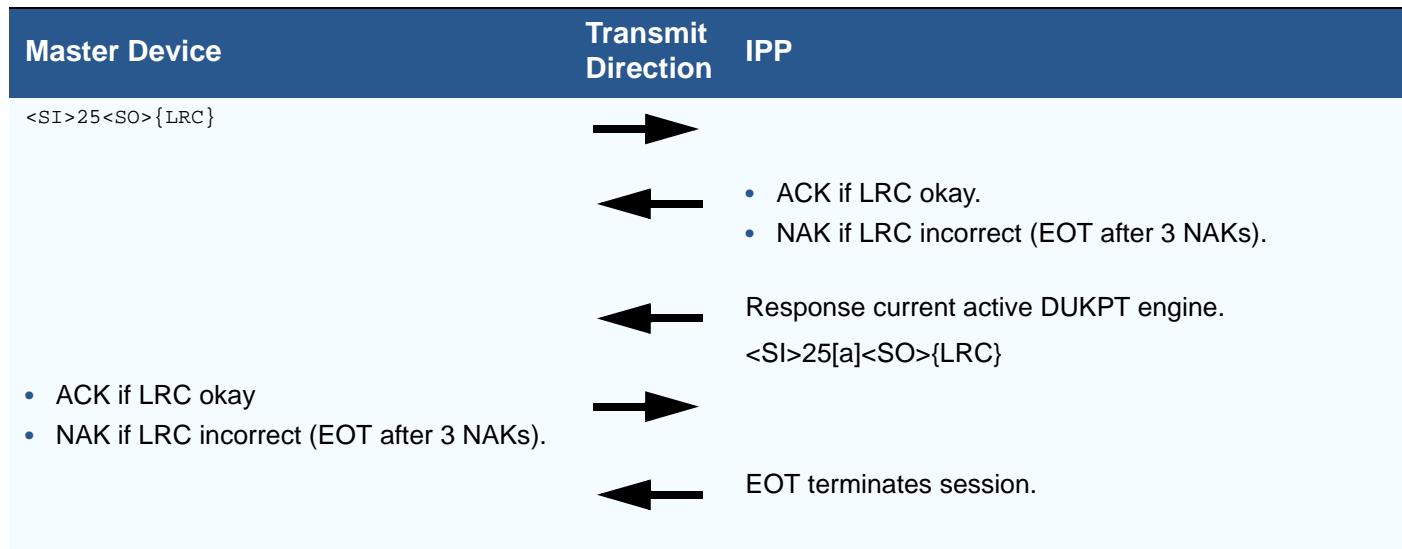
To Check DUKPT Engine:

<SI>25<SO>{LRC}

Response packet, DUKPT Engine "1" = active DUKPT Engine:

<SI>251<SO>{LRC}

**Table 113 Packet 25 Communication Protocol**



**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 114 DUKPT Packet 73 Format**

Data Element	Characteristic	Comments
<STX>	1H	Start of Text, Value: 02h
Packet Type	2AN	Value: 73

**Table 114 DUKPT Packet 73 Format**

Data Element	Characteristic	Comments
Packet Delimiter	1A	Value: (.), 2Eh
00000	5N	Value: 00000
[KSN]	10–20AH	Key serial number; hex. (leading Fs suppressed). Presented only if a PIN is entered; length is 0 if no PIN is entered.
Encrypted PIN Block	16AH	The 64-bit encrypted PIN block represented as 16 hexadecimal digits.
<ETX>	1H	End of Text, Value: 03h
{LRC}	1H	Error check

**DUKPT Packet 73 Length:**

- MAX: 47 characters
- MIN: 27 characters

**DUKPT Packet 73 Example:**

<STX>73.0000001234567890123456789123456<ETX>{LRC}

Packet 73 is the response packet to [Packet Z60: Accept and Encrypt PIN \(VISA Mode\)](#), the PIN request packet with no errors detected. If errors are detected in the Z60 or Z63 packet, the response packet is in the following format:

**Table 115 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"> <li>• 1 = no key</li> <li>• 2 = account error</li> <li>• 3 = PIN too long</li> <li>• 4 = PIN too short / non-decimal PIN digit in PIN</li> <li>• 5 = PIN pad used as MS<sup>a</sup></li> <li>• 6 = PIN pad has over 1 million transactions</li> </ul>
<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 116 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 117 DUKPT Packet 90 Communication Protocol**

Master Device	Transmit Direction	IPP
90 Packet		→
		←
		• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs • Initialization of 21 Future Keys
		Packet 91 with confirmed status
• ACK if LRC • NAK if LRC incorrect • EOT after 3 NAKs	→	

**DUKPT Packet 91:** 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 118 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"><li>• 0 = Confirmed</li><li>• 1 = Not confirmed</li><li>• 2 = (IPP7 only) Error; incorrect key length. Confirmation status 2 only applies to IPP7. It indicates that the length of the initial PIN encryption key does not comply with 1DES or 3DES DUKPT mode, as follows:</li></ul>
	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 119 DUKPT Packet 91 Communication Protocol**

Master Device	Transmit Direction	IPP
		Packet 91
<ul style="list-style-type: none"> <li>• ACK if LRC</li> <li>• NAK if LRC incorrect</li> <li>• EOT after 3 NAKs</li> </ul>		

**DUKPT Packet 76:  
PIN Entry Test  
Request** Directly presets the PIN code '1234' to do encryption and send response packet 71.

**Table 120 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



The amount filed must be present in the packet command, but the format is not confirmed.

**Table 121 DUKPT Packet 76 Communication Protocol**

Master Device	Transmit Direction	IPP
76 Packet	   	<ul style="list-style-type: none"> <li>• ACK if LRC</li> <li>• NAK if LRC incorrect</li> </ul> <ul style="list-style-type: none"> <li>• ACK if LRC</li> <li>• NAK if LRC incorrect</li> <li>• EOT after 3 NAKs</li> </ul>

**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 122 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 123 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"> <li>• 1 = no key</li> <li>• 2 = account error</li> <li>• 5 = CID field error</li> <li>• 6 = PIN pad has over 1 million transactions</li> </ul>
<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 Omni 5xxx.

### DUKPT Z69 Packet: Accept and Encrypt PIN / Data Authentication Request

On receipt of the Z69 packet, Omni 5xxx 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 124 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 125 DUKPT Packet Z69 Communication Protocol**

Master Device	Transmit Direction	IPP
Z69 Packet	→	
	←	<ul style="list-style-type: none"> <li>ACK of LRC okay</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>
	←	Packet 75 with confirmed status
		<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>

### DUKPT Packet Z69 Example:

VISA:

<STX>Z6901234567890<FS>C19.99<ETX>{LRC}

ANSI:

&lt;STX&gt;Z6901234567890&lt;US&gt;C19 . 99&lt;ETX&gt;{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 126 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>7500000000D04396624CF6892B04A000002468000048D5D7AF0333800FD<ETX>{LRC}
PC ---> PINPAD : <ACK>
```

Packet 75 is the response packet to packet Z69 or packet 78, PIN request, when no errors are detected in the request packet. If errors are detected in packet Z69 or packet 78, the response packet is in the following format:

**Table 127 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"> <li>• 1 = no key</li> <li>• 2 = account error</li> <li>• 3 = PIN too long</li> <li>• 4 = PIN too short/non-decimal digit in PIN</li> <li>• 5 = C D field error/not DUKPT mode</li> <li>• 6 = PIN pad has over 1 million transactions</li> <li>• 7 = amount error</li> <li>• 8 = ANSI MAC not allowed when using 1DES DUKPT</li> </ul>
<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:**

&lt;STX&gt;751&lt;ETX&gt;{LRC}

**Packet 78: DUKPT Accept and Encrypt PIN/Data Authentication Test Request****NOTE**

Packet 78 requests PIN encryption and MAC processing using a fixed PIN of '1234'. The response packet is packet 75.

This packet is used for testing and should not be used by applications.

**Table 128 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 129 DUKPT Packet 78 Communication Protocol**

Master Device	Transmit Direction	IPP
78 Packet	→	
	←	<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> </ul>
	←	Packet 75 with PIN = 1234
<ul style="list-style-type: none"> <li>ACK if LRC</li> <li>NAK if LRC incorrect</li> <li>EOT after 3 NAKs</li> </ul>	→	

## 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 130 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"> <li>• 4 = the last packet</li> <li>• 5 = the first/middle packet</li> </ul> Binary Data: Range: 6–7 <ul style="list-style-type: none"> <li>• 6 = the last packet</li> <li>• 7 = the first/middle packet</li> </ul>

**Table 130 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 <sup>a</sup>	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 131 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  Binary: X = 0, 2, 4 – 26, 28	224	00–99	
GISKE Key Block Format mode	ASCII: X = 0, 1, 2 – 14, 15  Binary: X = 0, 2, 4 – 12, 14	120	00 – 99	Due to size of GISKE key block, the size of message is reduce to 120 bytes.

## **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 132     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"> <li>• 0 = no error and MAC follows</li> <li>• 1 = ready for next Z66 packet and no MAC follows</li> <li>• 2 = out-of-order error and no MAC follows</li> <li>• 3 = [pointer] error and no MAC follows</li> <li>• 4 = [second key] error and no MAC follows</li> <li>• 5 = packet frame error and no MAC follows</li> <li>• 6 = [flag] error</li> <li>• 7 = [message] error</li> <li>• 8 = [working key] error/GISKE key usage, algorithm, mode of use, or key length error</li> <li>• 9 = incorrect key attributes of the master key (first or second)</li> </ul>
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 133 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

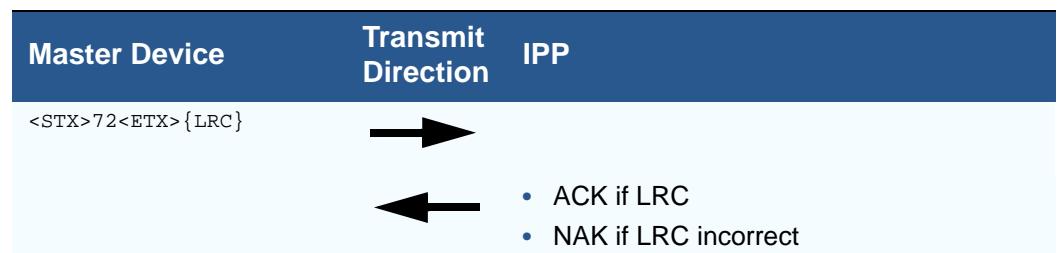
**Packet 72 Length:**

- MAX: 5 characters
- MIN: 5 characters

**Packet Z72 Example:**

<STX>712 <ETX>{LRC}

**Table 134 Packet 72 Communication Protocol**



**MAC Module Design**

**ANSI (Standard) MAC Algorithms**

The algorithm to calculate the MAC is fully compatible with the ANSI X9.19 1986, *Financial Institution Retail Message Authentication* specification. Within this standard, there are two modes of operation: CBC (Cipher Block Chaining) and CFB-64 (64-bit Cipher Feedback). In IPP5 implementation (that is, IPP5 and higher, including IPP6, IPP7, and Omni 5xxx 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 135](#).

**Table 135 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.





## APPENDIX E

### Special Topics

This appendix discusses topics that cover more than one subject area.

#### Smart Card PIN Entry using an External PIN Pad

The external PIN pad port supports transmit and receive signals and provides +9V power for external devices such as, the PINpad 1000SE or the SC5000.

The port can be used for asynchronous communications, at baud rates from 300–115200 and *only* supports character mode.

Pin	Signal	Pin	Signal
1	n/c	6	RXD
2	VPinPAD (+9V DC regulated power) <sup>a</sup>	7	TXD
3	n/c	8	n/c
4	n/c	9	n/c
5	GND	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 V 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. Verix V provides only core cryptographic services.

## Security Considerations

Verix V-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 V 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 V OS to generate a *master* key using `gen_master_key()`. The Verix V OS 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 V OS to generate a *session* key and encrypt it with the master key, using `gen_session_key()`. Verix V 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 V 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
<code>KEY_PARITY_ZERO</code>	1	Set key parity bits to zero.
<code>KEY_PARITY EVEN</code>	2	Set key parity even.
<code>KEY_PARITY ODD</code>	3	Set key parity odd.
<code>KEY_SEEDED</code>	4	Use input key value as a seed to generate the key.
<code>KEY_ERASE</code>	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**

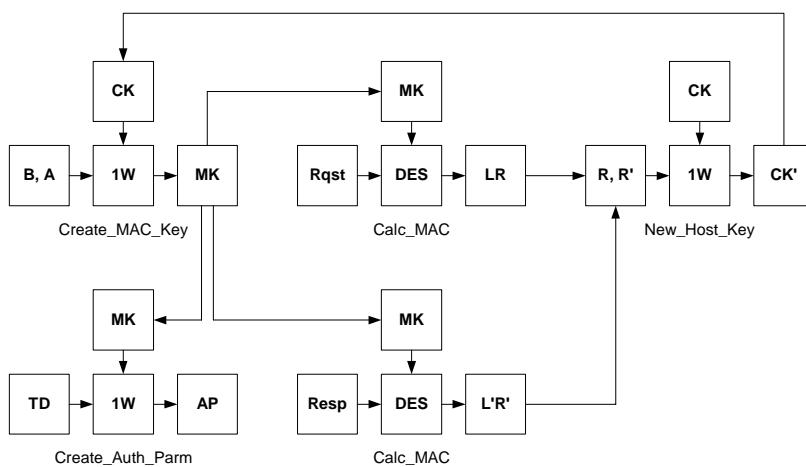
- |   |                        |
|---|------------------------|
| 0 | No master key present. |
| 1 | Master key present.    |

## Support for APACS40 Cryptographic Functions

The Verix V 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 35 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 35 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 *buffer, 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	Represent the low-order four bytes of the eight-byte MAC computed for
resp_residue	the request message and the response message.

### **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: 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: 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 EBUSY: Another caller owns the feature.

-1 and errno set to EBADF: Called file is missing, not properly initialized, or corrupt.

-1 and errno set to EINVAL: 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 V 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.



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\(\)](#)
- [SVC\\_CS2AZ\(\)](#)
- [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 `buffer`. 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 V-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 *buffer, int bufsize, int format,
          int precision);
```

### Parameters

<code>val</code>	Floating point value to convert.
<code>buffer</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:  $\geq 0$ : Number of bytes stored in `buffer`.

The result is equivalent to:

```
sprintf(buf, "%.<precision><format>", d);
```

## Itoa()

---

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 *buffer, int radix);`

### Parameters

`value`    The number to convert to an ASCII string. The result is stored in `buffer` as a zero-terminated string.

`buffer`    Where the converted number will be stored.

`radix`    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 *buffer, int radix);`

### Parameters

- |                     |  |
|---------------------|--|
| <code>value</code>  | The number to convert to an ASCII string.  |
| <code>buffer</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 <a href="#">ltoa()</a> . |

**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***

source	Input string to convert.
dest	Address to stored converted string.
size	Size of buffer at dest.

### **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 compressed variable length records. 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\(\)](#), [write\\_vlr\(\)](#), and [write\\_cvlr\(\)](#) 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  $\leq 255$ .

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**

## SVC\_UNPK4()

Decompresses ASCII data compressed by [SVC\\_PACK4\(\)](#). See [write\(\)](#), [write\\_vlr\(\)](#), and [write\\_cvlr\(\)](#) for details of the compression algorithm, and [SVC\\_PACK4\(\)](#) for a linked code example. See also [read\(\)](#), [read\\_vlr\(\)](#), and [read\\_cvlr\(\)](#).

`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  $\leq 55$ .

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 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).

### 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 [SVC\\_CS2AZ\(\)](#) 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 [SVC\\_CS2AZ\(\)](#) for a description of counted strings. See also [itoa\(\)](#), [ultoa\(\)](#), and [SVC\\_2INT\(\)](#).

**Prototype**    void SVC\_INT2 (unsigned int value, char \*buffer);

**Parameters**

value    The number to convert.

buffer    The address of buffer to store the string.

**Example**

## 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 [SVC\\_CS2AZ\(\)](#) for a description of counted strings.

**Table 136 Example Conversions**

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.

## **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\(\)](#).



## APPENDIX F

### Download Operations

The three download methods for Verix V-based terminals are *direct*, *remote*, and *back-to-back*. This chapter provides download precautions and describes procedures for downloading applications and files to Verix V-based terminals. Some operations are not explained in detail; there may be references to pertinent documentation.

To successfully transfer files into a Verix V-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 V Operating System Programming Tools Reference Manual*, VPN - 23231 for more information.
- Perform remote downloads using VeriCentre, a VeriFone remote download tool. Remote downloads use the public telecommunications network (phone line), or the TCPIP (available on some terminals). VeriCentre is described in the online manuals on the VeriCentre CD-ROM.

---

**NOTE**

To support ECSSecure, the OS includes terminal serial number from the MIB in the download sign-on packet. See *ECSSecure*, VDN - 23773, for details.

---

**CAUTION**

The CONFIG.SYS variables \*SMPW and \*PW should only be set using characters that are supported on the terminal keypad, otherwise, access to the System Mode 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.

Not all devices' keypads are the same. Caution should be exercised in creating System Mode and GID passwords. Valid characters for the passwords are those that can only be entered from the device keypad. The allowable characters include:

ABCDEFGHIJKLMNPQRSTUVWXYZ

0123456789

.,\*"- +#!;:@=&/ \%\$\_



On the Vx810 PIN pad, VeriCentre-initiated downloads are not supported since they require the use of a modem. However, modem tests can be performed on the Vx810 DUET base station because it has a built-in modem. Modem tests can be performed if the Vx810 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.

## OS Download Precautions

Downloading an operating system into a Verix V-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 RAM file space.

**Note:** Loading into RAM is the default.

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.

- 
- |  |  |
|--|--|
| <ul style="list-style-type: none"><li>• Download operating system files separately from other applications.</li><li>• Always download the entire operating system.</li></ul> | <p>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.</p> <p>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.</p> |
|--|--|

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 (System Mode):

- 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 System Mode and selecting RESTART or by power cycling.
  - a Press F2 and F4 to enter System Mode during the first screen (copyright notice) display.



You can enter System Mode anytime by pressing F2 and F4 (or ENTER and 7) but the terminal does not accept any downloads if an application is currently running. If an application is loaded, you must enter System Mode before the application launches.

- 2 Select DOWNLOAD from SYS MODE MENU 2 (SYS MODE MENU 1 on Vx670). Enter the file group where the files should be stored (see [Verix V Support for File Groups](#)).
- 3 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.

- 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 153](#)) 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 for Vx670.

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.

## Error Messages

[Table 137](#) provides explanations of error messages that appear on the host screen.

**Table 137 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.

**Table 137 Error Messages** (continued)

Error Message	Description
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.
Argument files nested too deeply	Argument files specified in an -f 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 system mode or an application can provide a way to selectively delete files.
Insufficient RAM	Some files must be deleted. Files can be deleted in system mode 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 -23 to +23 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.

**Table 137 Error Messages** (continued)

Error Message	Description
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

## 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.

- Connect Verix V-based terminals using an RS-232 serial cable through the COM1 ports.



On Vx670 terminal, COM1 is available via the Handy-Link connector and a power-barrel connector.

### Hardware Requirements

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.



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.

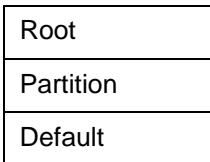
If the customer does replace/remove a certificate, files previously authenticated by the removed certificate must be reauthenticated on the sending terminal prior to performing a successful back-to-back download.

### 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.

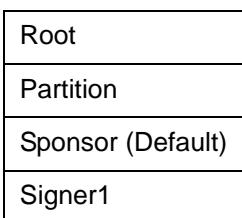
The default or factory certificate tree consists of three certificates: *root*, *partition*, and *default*. Files are authenticated using the default certificate. See [Figure 36](#).



**Figure 36     Factory Default Certificate Tree**

The customer replaces the default certificate with an sponsor certificate and adds a signing certificate (see [Figure 37](#)). At this point, all new files must be authenticated with the signing certificate.

**Initial Customer Certificate Tree:** Replace sponsor certificate (authenticated by the default certificate) and add new Signer1 (authenticated by sponsor).

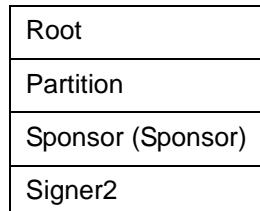


**Figure 37     Initial Customer Certificate Tree**

At this point, the customer 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 36](#).

**Second Customer Certificate Tree:** Replacement sponsor certificate (authenticated by the sponsor) and add new Signer2 (authenticated by sponsor).



**Figure 38 Second Customer Certificate Tree**

### Valid Back-to-Back Downloads

The following shows back-to-back downloads in the *valid* direction:

- Factory default → factory default
- Initial customer → factory default
- Second customer → initial customer

### Invalid Back-to-Back Downloads

The following shows back-to-back downloads in the *invalid* direction:

- Second customer → factory default
- Factory default → 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 default (one revision) works because the sponsor certificate is authenticated by the default certificate and can be authenticated in the factory default unit.



**NOTE** You cannot initiate a back-to-back download from a second customer unit to a factory default unit (two revisions) because the sponsor certificate in the second customer certificate tree is authenticated by the sponsor certificate not present in the factory default 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 Place the terminal into System Mode (press F2 and F4 simultaneously).
- 2 Enter the System Mode password.
- 3 Press the star ('\*') key.

**4** Enter the Group 1 password.

The terminal screen displays “SYS MODE UPLOAD” and “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 Back-To-Back

Back-to-back downloads are available over COM1 and USB. Press the 4 key to start upload, or press the 5 key to start download on any of the three main System Mode menus. In the Back-to-Back Menu screen, the user can select from either of the two communications options—USB or COM1. This is also available on the V<sup>x</sup>700 units.

## Logical File Transfers

Logical file transfers transfer all files in the user file system. No system files in RAM or in the boot sectors of flash 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 RAM or flash to accommodate all files being transferred. Clear and defragment the flash on the Target terminal prior to initiating a download. The download can also fail if the RAM or flash memory sizes of the receiving terminal are less 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 Flash

Coalescing the flash memory may free enough memory space to clear a flash memory problem. If not, files must be deleted. Files can be deleted in System Mode or an application can provide a way to selectively delete files.

## Insufficient RAM

Some files must be deleted. Files can be deleted in System Mode 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.

## File Name Extensions and GIDs

By default, downloaded files are stored in Group 1 of the RAM 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 RAM 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).

## 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 included in the VVDTK is used to sign files. See the help files included with the tool for specifics on using VeriShield.

The Verix V checks the signature using the *key certificate* stored in the terminal. If the certificate passes the authentication process, Verix V concludes that the file is signed by someone who knows the secret key and that the file is valid.

Terminals are shipped from manufacture with a default key certificate stored in the memory. The VVDTK includes key and certificate files for the default key—`k2sign.key` and `k2sign.crt`. These files can be used to authenticate files as long as the default certificate remains in the terminal. Since the default key is known to many people, it should be replaced by a secret key before the terminal is deployed. The process of obtaining and installing keys are not described in this manual. It is assumed that the developer has the proper key 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 RAM, even if the file it signs is in flash. Both files *must* be loaded to the same file group.

### NOTE



See DDL- Download Utility in the Verix, V Operating System Programming Tools Reference Manual for information on using the DDL utility.

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 RAM:

```
ddl hello.out hello.p7s *go=hello.out
```

- Download to a flash file:

```
ddl SETDRIVE.F hello.out SETDRIVE.I hello.p7s *go=f:hello.out
```

or, more briefly, since RAM 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 *****
SYSTEM CERTIFICATE
K2PART.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.

```
***** VERIFYING FILES *****
CHECK CERTIFICATE
OWNR.CRT

** AUTHENTIC **
```

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
MYFILE.P7S
MYFILE.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.

### **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 linked.

### Prototype

```
int authenticate(const char *signature_file_name);
```

### Parameter

signature\_file\_name     The authenticated file name.

## **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**

source	Authenticated file
target	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. EACCES - if the cryptographic signatures of file source and target are not equal.



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 V 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.



**CAUTION**  
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 V 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 RAM 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	RAM	Current
3\Y.OUT	Y.OUT	RAM	3
3\Y.DAT	Y.DAT	RAM	3
F3\Z.OUT	F:Z.OUT	Flash	3
F\MAIN.OUT	F:MAIN.OUT	Flash	Current
15\COURIER.FON	COURIER.FON	RAM	15
F15\ARIAL.FON	F:ARIAL.FON	Flash	15
15F\ARIAL.FON	15F/ARIAL.FON	RAM	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 flash group (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 RAM. 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 138](#)). 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 138 displays a list of error codes.

**Table 138      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 139.

For messages generated by the download host, see the documentation for the VeriCentre host software.

At the end of a System Mode-initiated download, the last error message remains on screen until a key is pressed.

**Table 139      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 139      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.

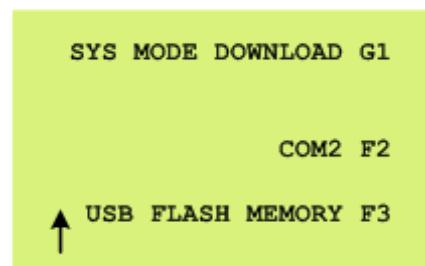
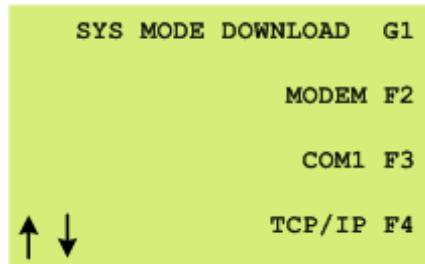
## User Mode TCP/IP Download Support

**NOTE**



The \*COMBO variable is ignored for System Mode downloads.

System mode allows the port to be selected using menus like the following two screens:



**NOTE**



USB FLASH MEMORY is displayed only on platforms that support USB and where the memory stick is plugged in.

If the user selects TCPIP then the system considers several possibilities, based on a new variable in CONFIG.SYS.

If the user selects TCP/IP when the option is unavailable, an error tone is heard and a message is displayed.

SYS MODE DOWNLOAD G1

TCP/IP NOT PRESENT

**NOTE**



To go back to System Mode menu, press Enter or 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.

System Mode 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.

Prior to invoking `run()`, the System Mode 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 TCPIP button is pressed.

SYS MODE DOWNLOAD G1

NO \*ZTCP VARIABLE

If \*ZTCP is present and is pointing it to an non-existent program, the terminal displays message as shown below.

```
SYS MODE DOWNLOAD G1
```

```
<Program name>  
NOT FOUND
```

If the specified program is not authenticated, the message is displayed as shown below.

```
SYS MODE DOWNLOAD G1
```

```
<Program name>  
NOT AUTHENTICATED
```

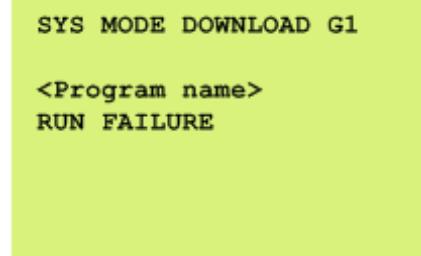
If the program fails to run, the message is displayed as shown below.

```
SYS MODE DOWNLOAD G1
```

```
<Program name>  
RUN FAILURE
```

If the user selects COM2 on a Vx610 terminal and if \*ZTCP is not present but the module has an IP stack (WiFi), the terminal should “dial” for an IP download. If the required environment variables are not set (i.e., \*ZP), the terminal requests for it from the user.

When the user selects COM2 on a Vx610 terminal, 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.

## USB Flash Memory Download

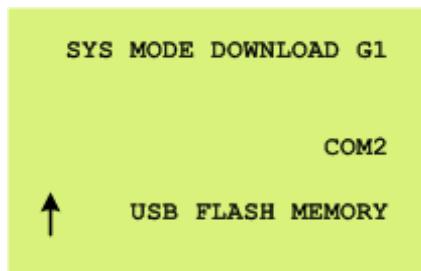
The OS provides the USB Flash Memory Download functionality in the Verix V terminals. It allows the user to download a single file, VERIFONE.ZIP, into the current group from the USB flash memory. This download can be through a System Mode or an automatic download.

## USB System Mode Download

System Mode allows the user to select USB FLASH MEMORY option to download from USB flash. The System Mode menu screen is as shown below:

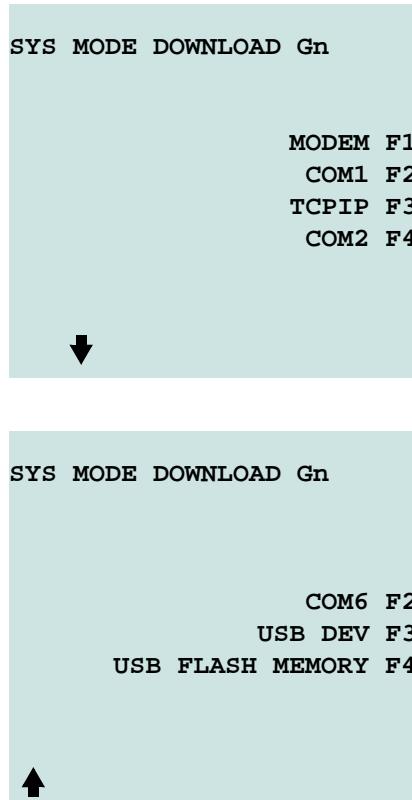


**NOTE** Display and confirmation messages are the same across all platforms, except when otherwise stated.



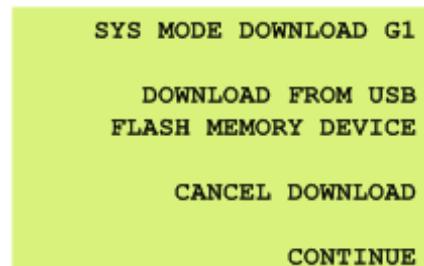
**Figure 39      Sample Vx5xx/Vx610 SYS MODE DOWNLOAD Screen**

Figure 40 shows sample System Mode Download screen.



**Figure 40      Sample Vx670 SYS MODE DOWNLOAD Screen**

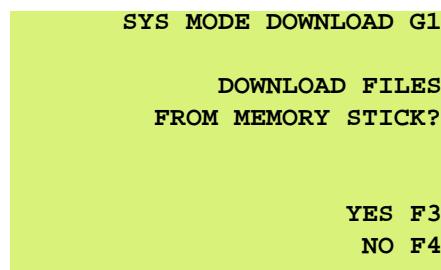
If the user selects the USB FLASH MEMORY option, then the System Mode prompts the user to confirm it.



Once confirmed, the terminal task proceeds to copy the file into the terminal's RAM file system in the current active group. When the files are successfully copied, System Mode displays the DOWNLOAD COMPLETE message.



If the user selects CONTINUE, from the DOWNLOAD FROM USB FLASH MEMORY DEVICE screen, the option DOWNLOAD FILES FROM MEMORY STICK appears. The user has the option to either download every file or allow the OS to get all the files.



If a single file exists and the user selects NO, the download is cancelled and the OS returns to System Mode Menu 1. If multiple files exist, the user has the option whether to download or not to download each file by name. This is also true for SD Memory on terminals with SD capability.

While downloading, in addition to copying the file VERIFONE.ZIP into the RAM of the current group, the system mode also sets the CONFIG.SYS variable \*UNZIP=VERIFONE.ZIP in the current group. On restart, the archive is unzipped.

## USB Flash Auto-Download

On Vx670 and units that support USB Flash, auto-download provides a faster way to download VERIFONE.ZIP from the USB flash drive compared to the USB System Mode download. Auto-download is initiated when the terminal is in idle state. When no applications are running, the Verix V OS monitors the USB flash drive for the file VERIFONE.ZIP and then copies it to the RAM 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 V 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 RAM files in GID1 before copying VERIFONE.zip to RAM, GID1.
- Start and End – The Verix V OS 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 V OS 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 Vx810 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 Vx810 PIN pad is at the DOWNLOAD NEEDED screen.

- 2** Set CONFIG.SYS \*SMDI=1, insert the external memory device or start the download at the download host, and restart or power cycle the Vx810 PIN pad.

The OS automatically performs a download from any one of the following sources:

- Default Serial Port (COM2 on Vx570)
- 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 System Mode 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 Vx810 unit at the same time.

**Multiple ZIP file Downloads**

New System Mode 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 "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 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 System Mode 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**

The Vx810 PIN pad has a single SD card slot. 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 on the Vx810 PIN pad is solely used for application downloads. When present, the System Mode 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.

## **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 V 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.



**CAUTION** As soon as the download starts to maximize the free Flash and RAM space in the terminal, old application files can be cleared. When the download fails, the terminal is rendered unusable.

Alternatively, old applications can be left in the terminal during download, allowing it to be overwritten once new application files are unzipped. However, there must be enough space in the terminal to support this.

In a resumable full download, System Mode does not remove files until all split files are received and checked for a correct checksum, unlike in a regular full download where System Mode 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 by Verix V.

### \*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 System Mode checks for the split files in all groups every time a System Mode 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.



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 140 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 System Mode UI Control	N/A	N/A	N/A

- If \*ZRDL exists, the System Mode 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 System Mode. The counter decrements for each download retry.

**NOTE**

\*ZTRY is not decremented on each retry.

### System Mode Download

If \*ZRDL exists, the System Mode 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. System mode will not wait for a keypress to continue, unlike in a regular download.

### Clear Split Files

In the System Mode 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.

Refer to the System Mode menu of an eight-line standard interface (e.g., Vx510) for RDL screen shots.

### 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 If files in Flash and free Flash space is less than the size of the split file, defragment Flash to recover space taken by #mapp.zip~1 split file.
- 6 Repeat steps for #mapp.zip~2 until #mapp.zip~LAST.

If there is enough free Flash space, the entire file can be reassembled with only one defragmentation cycle at the end to recover space taken by the deleted split

files. Otherwise, Flash defragmentation cycle may be needed for each split file. It is safe to defragment with no restart because no applications are running.



**NOTE** Defragmentation cycles are minimized by downloading the split parts in reverse order of processing. VeriCentre controls the order in which files are downloaded, thus, it must ensure the optimal file ordering (i.e., ~LAST, ~9...~1, ~CRC).

VeriCentre increases or reduces the number of defragmentation cycles by configuring the value of \*DEFRAG—the higher the value, the lesser the time spent defragmenting Flash (increases the chance of running out of Flash free space); the lower the value, the longer the time spent for defragmentation. Setting the value to \*DEFRAG=0 is the safest but slowest option.

## Defragmenting Flash

In the current System Mode unzip process, Flash is defragmented between unzipping each ZIP file. Old files are not automatically removed prior to unzipping, which can cause unzip to fail due to lack of Flash free space.

Manual file removal and defragmentation is allowed but requires System Mode password. VeriCentre can remove old files but if it removes them and the download fails, the terminal is unusable.



**NOTE** It is ideal to remove old files and defragment to recover free space after a valid ZIP file is received and before unzipping the file.

### \*UNZIP2

On startup, System Mode 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 to. 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 program 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 System Mode 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.

The System Mode then defragments Flash to recover the space taken by the deleted files. The variable \*DEFRAG=0 in GID2 ensures this happens. System Mode unzips the file F:/#bigfat.zip, after unzipping, the file is removed. Finally, after processing \*UNZIP2 in all groups, System Mode 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 V 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
/file	Removes “file” from I: in the current group
./	Removes ALL files from I: in the current group
I:./	Removes ALL files from I: in the current group
F:2/	Removes ALL files from F: in group 2
*:./	Removes ALL files from both I: and F: in the current group
*/	Removes ALL files from I: in ALL groups
F:*/	Removes ALL files from F: in ALL groups
I:*/	Removes ALL files from I: in ALL groups
*:*/	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 \*ZRD is defined—this allows the terminal to run old applications in case the new application download fails.



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.

System Mode always clears RAM and Flash files before the download proceeds. If the download fails, the terminal is left without an application.

```
SYS MODE DOWNLOAD Gnn
*****WARNING*****
CLEAR Application
FROM GROUP Gnn?

YES F3
NO F4
```

### SVC\_ZONTALK() download()

RAM and Flash 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:

- \*GUARD =1 Protection in GID 1 is enabled
- \*GUARD =15 Protection in GID 15 is enabled
- \*GUARD =1, 15 Protection in GID1 and 15 is enabled
- No \*GUARD 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 System Mode—it can be disabled or deleted at System Mode.
- 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 the “CLEAR MEM” option. When this happens, either of the following messages is displayed:

RAM & FLASH CLEARED  
GID 1 GUARDED  
COALESCING FLASH

ALL RAM & FLASH CLEAR  
GUARDED GIDS RETAINED  
COALESCING FLASH

---

**NOTE**

The above screens are samples only. Other screens are possible.

When a download is performed from System Mode, 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.



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 GID1, and add new files.



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 System Mode, but still allows individual files and variables to be updated or removed via downloads or applications.



## APPENDIX G

### Power Management

The Verix V OS 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 V OS 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 µA.



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

---

Use Figure 41 as reference for features of the Vx5xx/Vx610 terminals.

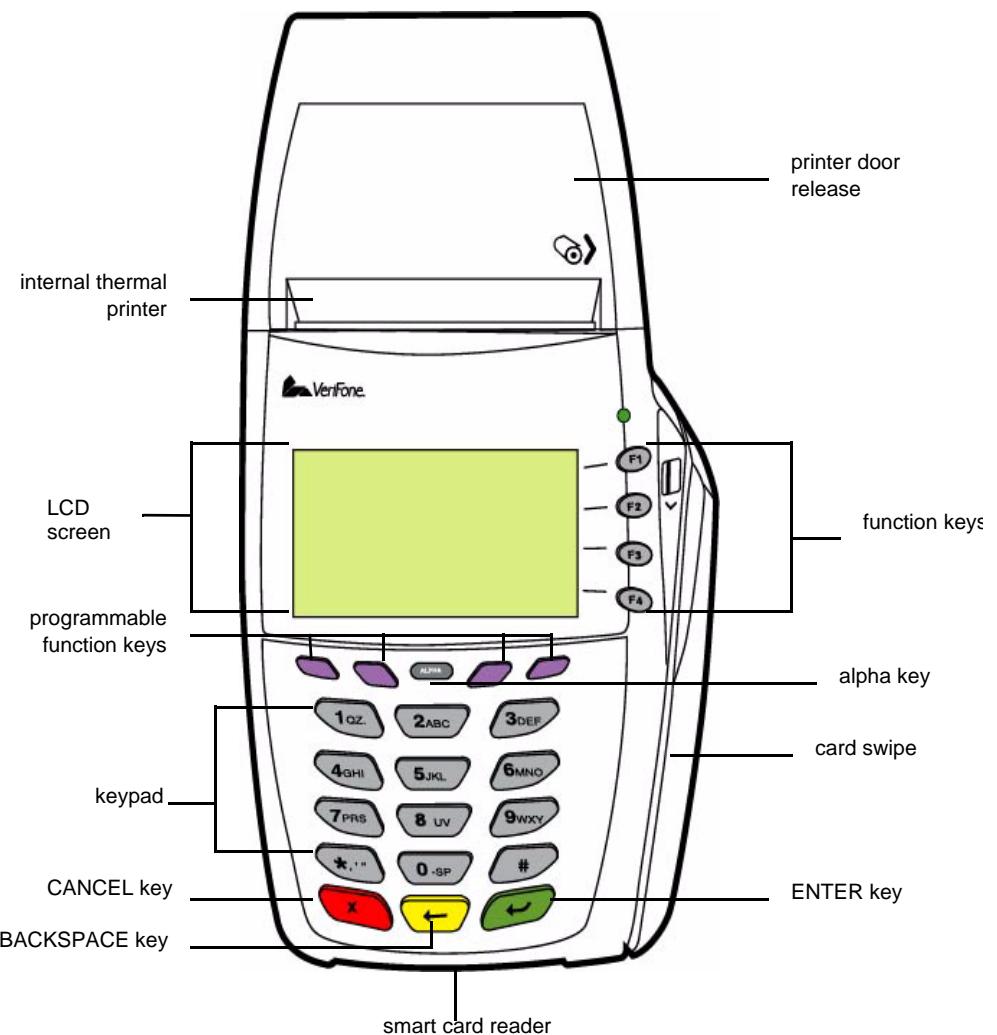
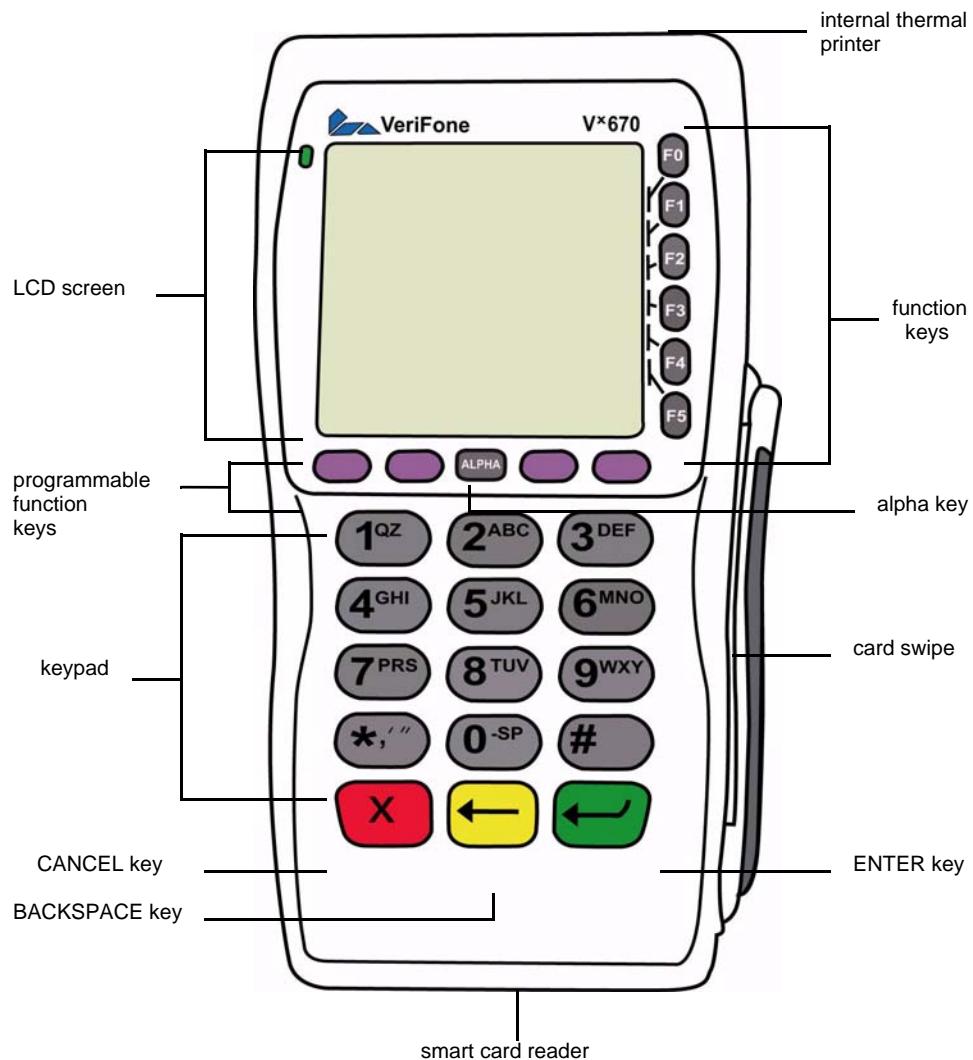


Figure 41 Vx5xx/Vx610 Terminal Features

Use [Figure 42](#) as reference for the features of the Vx670 terminal.



**Figure 42** Vx670 Terminal Features

**NOTE**



The Vx670 terminal has two additional function keys, F0 and F5.

## 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 V 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 will enter 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 will wake 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 then use `get_dock_sts()` to determine the current state of the unit.

**NOTE**

On Vx670, 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.

## Power Supply Control on V<sup>x</sup>700 PIN Pad

The V<sup>x</sup>700 PIN pad accepts 5-32 VDC power from either the DC IN jack or the COM 1 port. Internally, the V<sup>x</sup>700 PIN pad uses DC-DC converters to generate 1.8VDC, 3.3 VDC, and 5.0 VDC. Control of the 3.3 VDC and 5.0 VDC power supplies is similar to that of the V<sup>x</sup>670 terminal.

### 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.

## Printer, Battery, and Radio Interaction

The Verix V 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 (a Texas Instruments BQ2000T Battery Charger chip for V<sup>x</sup>610 and ISL6253 Battery charger chip for V<sup>x</sup>670). Both V<sup>x</sup>610 and V<sup>x</sup>670 terminals support a battery capacity of 1800 mAH.

---

### NOTE



The V<sup>x</sup>810 and the V<sup>x</sup>700 units do not operate from a battery and must be connected to an external power source to operate. These units do not support an integrated printer. If the application calls any of the battery API functions, the OS returns either of the errors `ENODEV` or `EINVAL`.

The V<sup>x</sup>810 DUET contains a micro-controller chip connected to a thermal print mechanism.

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

The BQ2000T Battery Charger Chip on V<sup>x</sup>610 handles all aspects of the battery charging. It automatically detects when the charging should begin and end. This renders the OS to only perform the enabling and disabling of the charger chip, which under normal operating conditions, is always enabled. On the other hand, the ISL6263 Battery Charger Chip on V<sup>x</sup>670 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 V<sup>x</sup>610 reads the thermistor via the ADC Channel 4, and the V<sup>x</sup>670 reads the thermistor via Channel 0. If the temperature is too low or too high, charging should be disabled.

On Vx670, 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 now 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 Vx510 GPRS, a power management for the printer similar to the Vx610 PIN pad is implemented, which calculates and informs the printer driver of the available power. The printer driver uses this information to print the maximum number of dots, printing as fast as available power allows.

**NOTE**



On V5 terminal, the OS increases printer speed from 9 lines per second to 12.5 lines per second by increasing the power to the printer mechanism.

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 will indicate a mechanism error.

## **Application Interface**

Application participation in power management is indirect but absolutely essential for the success of the system. Each Vx6xx 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 V API.

**Function Calls** Applications must use the system calls listed in Table 141 for Verix V terminals:

**Table 141 Power Management Functions for Verix V Terminals**

Function	Description														
<code>int get_battery_sts(void);</code>	<p>Indicates battery status, and returns 1 if the battery level is OK; 0 if battery level is low, and -1 if the unit does not have a battery. 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.</p> <p>If the battery desires a different threshold, the <code>get_battery_value</code> may be used.</p>														
	<table> <thead> <tr> <th>Condition</th><th>State</th><th>Return Value/Action</th></tr> </thead> <tbody> <tr> <td>Battery voltage &lt; 6.300 volts</td><td>Battery Critical</td><td>Unit will turn itself off</td></tr> <tr> <td>Battery Remaining Charge &lt; 100 mAh</td><td>Battery Low</td><td>Return 0</td></tr> <tr> <td>None of the above</td><td>Battery OK</td><td>Return 1</td></tr> </tbody> </table>			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
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													
<code>int get_battery_value(int type);</code>	<p>Returns the requested battery values. The application may request Battery Full Capacity, Remaining Capacity, and Voltage.</p> <table> <thead> <tr> <th>Name</th><th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>FULLCHARGE</td><td>0 - 2500</td><td>Theoretical MAX charge, in mAH</td></tr> <tr> <td>REMAININGCHARGE</td><td>0 - 2500</td><td>Remaining charge, in mAH.</td></tr> <tr> <td>BATTERYVOLTAGE</td><td>6000-8500</td><td>Battery voltage, in mV.</td></tr> </tbody> </table>			Name	Value	Description	FULLCHARGE	0 - 2500	Theoretical MAX charge, in mAH	REMAININGCHARGE	0 - 2500	Remaining charge, in mAH.	BATTERYVOLTAGE	6000-8500	Battery voltage, in mV.
Name	Value	Description													
FULLCHARGE	0 - 2500	Theoretical MAX charge, in mAH													
REMAININGCHARGE	0 - 2500	Remaining charge, in mAH.													
BATTERYVOLTAGE	6000-8500	Battery voltage, in mV.													
<code>int get_dock_sts(void);</code>	<p>If no battery is present, -1 is returned.</p> <p>A result of 0 indicates the unit is docked; -1 indicates it is undocked.</p>														

**Table 141 Power Management Functions for Verix V Terminals**

Function	Description												
<code>int get_powersw_sts(void);</code>	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.												
<code>int set_backlight(int mode);</code>	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.												
<code>int set_com1_pwr(const char *sigs);</code>	<p>This API controls power on Vx610 and Vx670 terminals.</p> <p>This function will always return 0.</p> <p>sigs is a character that contains the following bit patterns:</p> <table> <tr> <td><code>#define COM1_PWR_ON (1&lt;&lt;0)</code></td><td>Turns on the COM1 level driver chip.</td></tr> <tr> <td></td><td>Available on Vx610 and Vx670 terminals.</td></tr> <tr> <td></td><td>Powerup default is ON.</td></tr> <tr> <td><code>#define COM1_PWR_PIN_ON (1&lt;&lt;1)</code></td><td>Turns on the power pin on the COM1 port.</td></tr> <tr> <td></td><td>Available only on Vx610 terminal.</td></tr> <tr> <td></td><td>Powerup default is OFF.</td></tr> </table>	<code>#define COM1_PWR_ON (1&lt;&lt;0)</code>	Turns on the COM1 level driver chip.		Available on Vx610 and Vx670 terminals.		Powerup default is ON.	<code>#define COM1_PWR_PIN_ON (1&lt;&lt;1)</code>	Turns on the power pin on the COM1 port.		Available only on Vx610 terminal.		Powerup default is OFF.
<code>#define COM1_PWR_ON (1&lt;&lt;0)</code>	Turns on the COM1 level driver chip.												
	Available on Vx610 and Vx670 terminals.												
	Powerup default is ON.												
<code>#define COM1_PWR_PIN_ON (1&lt;&lt;1)</code>	Turns on the power pin on the COM1 port.												
	Available only on Vx610 terminal.												
	Powerup default is OFF.												
	<p>See: <a href="#">Example</a></p> <p><code>int BatteryRegs(char * buffer);</code></p> <p>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.</p> <p>This function is available on Vx670 terminals only.</p> <p>On Vx610 terminals, this returns &lt;0.</p>												
<code>int get_battery_initialization_status(void)</code>	<p>Returns the initialization status. The battery is considered initialized when it has been fully charged within a terminal and the Remaining Charge value accurately calibrated. The return code is 1 if battery is initialized, 0 if not initialized, and -1 if there is no battery.</p> <p>This function is available on Vx670 terminals only.</p>												

**Table 141 Power Management Functions for Verix V Terminals**

Function	Description
int SVC_SLEEP(void);	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 will go to sleep after 50 ms. Upon waking up, the internal *POW variable is restored. *POW in group1 CONFIG.SYS is never changed in this function. If SVC_SLEEP is not used, the OS sleep features works the same as in Vx610.
int SVC_SHUTDOWN(void);	This function is available on Vx670. On Vx700, this forces the unit to sleep mode.
int set_com_pwr(int port, cons char *sig)	This commands the terminal to turn itself off. If the function detects that it is not a Vx610 terminal, it returns -1. If the Vx610 terminal is being powered by an external power pack, it returns -1, otherwise this function returns 0.
	On Vx700, since there is no real "OFF" state while the PIN pad is powered on, this returns -22 and *TURNON and *TURNOFF are not used.
	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, or -1 if -1 with errno set to EBADF, if COM port is not opened, -1 with errno set to EINVAL, if open block is not set.
	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.

### 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 V 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 will trigger a console event for the current owner of the console. Setting \*GKE=2 will generate 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.



## APPENDIX H

### WWAN

This appendix describes the application interface to the WWAN (Wireless Wide Area Network) for the Vx510, Vx610, and Vx670 terminals.

The goal of using a WWAN is portability. The Vx510/Vx610/Vx670 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 V 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.

---

The following function calls are specific to the radio modem:

## **get\_radio\_sts()**

---

Returns the status of the signal RAD\_INT and RAD\_INT2.

**Prototype**    `int get_radio_sts(int hdl, char *sigs);`

**Parameters**

`hdl`                  Handle of the COM port.  
    `*sigs`                 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.

---

## **get\_port\_status()**

Copies current port status information to caller's 4-byte buffer as follows and returns a result code indicating whether or not any output is currently queued for the printer.

**Prototype**    `int get_port_status(int handle, char*buffer);`

## **reset\_port\_error()**

Has no effect and the corresponding error indicators are always 0. In general for Verix V communication ports, `reset_port_error()` resets the error indicators for parity, framing, and overrun errors, and the break indicator.

***Prototype***    `int reset_port_error(int handle);`

## 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.

**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 *EMXXXX Development Kit AT Command Reference Manual*, E-2130394 Revision A from Sierra.

On Vx610 terminal, the Kyocera M200 CDMA radio module is supported in place of the Sierra EM3420 CDMA radio module. Communication to the wireless modules is over the Vx610 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 Vx610 COM2 port and the new Kyocera and current radio modules.

**Table 142 COM 2 Port Connections**

<b>COM2 Lines</b>	<b>Direction</b>	<b>Kyocera CDMA Lines</b>	<b>Siemens GPRS Lines</b>	<b>CO WiFi Lines</b>
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. See `set_radio_ctl()` for more information. The other two are inputs to the CPU from the radio modules. See `get_radio_sts()`.

**Table 143 Radio Control Lines Connections**

<b>COM2 Lines</b>	<b>Direction</b>	<b>Kyocera CDMA Lines</b>	<b>Siemens GPRS Lines</b>	<b>CO WiFi Lines</b>
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)
```

## **set\_radio\_ctl ()**

Controls the settings of RAD\_MOD, RAD\_RST, and RAD\_OFF.

**Prototype**    int set\_radio\_ctl (int hdl, const char \*sigs);

**Return Values**    Unused bits 3-7 should be set to 0.

Success        0

Failure        -1, errno set to EACCESS.

-1, errno set to EINVAL when executed on a terminal that does not support set\_radio\_ctl().

**Example:**    The following example sets RAD\_OFF, RAD\_RST, and RAD\_MOD to 1.

```
char sigs = 0x07;  
set_radio_ctl(hdl, &sigs);
```

## get\_radio\_sts ()

Returns the status of RAD\_INT and RAD\_INT2.

**Prototype**    `int get_radio_sts (int hdl, char *sigs);`

**Return Values**    Unused bits 2-7 should be set to 0 or 1. Never assume the unused bits will be 0.

Success        0

Failure        -1, errno set to EACCESS.

-1, errno set to EINVAL when executed on a terminal that does not support get\_radio\_ctl().

## Wireless Module ID EEPROM

The Vx610 terminal 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 Vx610 terminal 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 Vx670, 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 /EMERGOFF pin on the radio module, thus shutting the radio off.

### Vx670 CR GPRS

The Vx670 CR GPRS is a cost reduction version of current Vx670 GPRS terminal. It uses the Siemens MC55i GPRS radio and the MRD531 MSR decoder chip similar to those used in all other Verix V terminals, except Vx670.

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 Vx670 CR GPRS OS also runs on Vx670 Classic—the OS reads the hardware configuration information from the Manufacturing Information Block (MIB) to determine which hardware version is present.

The Vx670 CR 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.

---

### Vx510 GPRS

The Vx510 GPRS terminal is 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 Vx670 CR 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)
```

## WiFi

On Vx610, the WiFi radio module has a Compact Flash interface. The ConnectOne iChip bridges between the serial and CF interface. The iChip supports an onboard TCP/IP stack. For more details refer to the *AT +iProgrammers Manual* from ConnectOne.

On Vx670, the WiFi device is Conexant CX3889 USB WiFi chip (Partagas). Actual throughput is expected to be 3-5 Mbits/s. The USB WiFi hardware does not have a built-in IP stack , thus, IP download requires loading Comm Server in the terminal. The Comm Server can be loaded in the factory using the protected file feature (adding 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 more information, refer to [USB Internal WiFi \(WLN1\)](#) section on [Appendix i](#).



## APPENDIX I

# USB Support

This section details the USB features supported on Vx570, Vx670, and Vx670-based units.

## USB Flash Drive

On terminals that support it, this enables the System Mode 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)

On Vx570, some of the devices are permanently connected (Ethernet) while others are plugged during operations. The USB Ethernet is independent of the modem but both can operate simultaneously.

On Vx670, the USB Ethernet is an external device different from that of the Vx570, 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.

The Vx810 PIN pad supports the existing Vx570 USB to Ethernet hardware, packaged in a “dongle” cable instead of mounted internally as in the Vx670 terminal. When this cable is used, applications can access the Ethernet using DEV\_ETH1 (“/DEV/ETH1”). The application interface is identical to that of the Vx570 terminal. Applications cannot receive Ethernet disconnect events because the Ethernet dongle cable is also the Vx570 power cable. The only way to disconnect the Ethernet is to disconnect the power.

The DUET base station may contain an optional USB Ethernet device. The OS detects its presence the same way it detects the USB Ethernet dongle in the stand-alone Vx810 PIN pad. The OS behaves exactly the same as in the stand-alone Vx810 PIN pad. 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.

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.



**NOTE** The V<sup>x</sup>700 PIN pad supports Ethernet over USB driver, similar to the current V<sup>x</sup>570 PIN pad with some changes in the connection. Power is added to the connection to eliminate external power source.

## USB Ethernet Functions

This section details the APIs used in USB-based Ethernet.

## **open()**

Returns the handle for reading or writing the Ethernet device.

**Prototype**    `int open("/dev/eth1", int unused);`

**Parameters**    `/dev/eth1`      Ethernet device name.

**Return Values**    Success:      The handle is returned.

Failure:      `-1` and `errno` set to `EBUSY`.

## read()

This API allows the user to read one packet from the Ethernet device.

**Prototype**    `int read(int eth_handle, char *buffer, int count);`

**Parameters**

<code>eth_handle</code>	The handle value returned from <code>open()</code> .
<code>*buffer</code>	Buffer to hold the maximum packet size of 1514 bytes.
<code>count</code>	Size of the buffer in bytes.

**Return Values**

Success: It returns the number of bytes read.

0: No data is read.

Failure: -1: `errno` set to `EBADF`.



**NOTE**

Any count less than 1514 is considered to be an error: the result will be -1 with `errno` set to `EINVAL`.

## write()

This API allows the user to write one packet for the Ethernet device.

### Prototype

```
int write(int eth_handle, const char *buffer, int count);
```

### Parameters

eth_handle	The handle value returned from <code>open()</code> .
*buffer	The buffer holds the outgoing packet of size "count" bytes. The packet can be as small as 60 bytes.
count	Number of bytes in the outgoing packet from the buffer. It ranges from 60 and 1514 bytes.

### Return Values

Success:	Number of bytes written.
Failure:	-1 and errno set to ENOSPC. -1 and errno set to EBADF.

## close()

---

This API releases the Ethernet device.

**Prototype**    `int close(int eth_handle);`

**Parameters**    `eth_handle`    The handle value returned from `open()`.

**Return Values**

Success:	0: The device successfully closed.
Failure:	-1 and <code>errno</code> set to <code>EBADF</code> : hdl is not a valid open file descriptor.

## get\_enet\_status()

This API checks whether Ethernet link is live or not. The status is returned in the status4 array.

**Prototype**    `int get_enet_status(int hdl, char *status4);`

### Parameters

hdl	The handle is the value returned by <code>open()</code> .
*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 MAC (Media Access Control) address.

**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 causes the USB Ethernet device to ignore classes of packets specified by the rx\_control parameter.

**Prototype**    `int set_enet_rx_control(int hdl, int rx_control);`

### **Parameters**

`hdl`                  The handle is the value returned by `open()`.

`rx_control`         The `rx_control` parameter is bit-mapped. The two least significant bits have been assigned functions. All other bits are reserved and should be set to zero.

        bit 1: 1 = allow reception of multicast packets.

                  0 = disallow reception of multicast packets.

        bit 0: 1 = allow reception of broadcast packets.

                  0 = disallow reception of broadcast packet.

---

### **CAUTION**



- If broadcast packets are disallowed, the Address Resolution Protocol will not work.
  - This command only affects messages received after the command has been sent to the Ethernet chip. Any broadcast or multicast packets received by the Ethernet chip prior to the user's invocation of this command will be queued up and delivered to the user in response to `read()` commands.
- 

### **Return Values**

Success:        0

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

## USB External Dial (COM3)

The modem port COM3 is also a USB device on V<sup>x</sup>670. The V<sup>x</sup>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 DUET base station contains a USB modem device, which is functionally the same as the USB modem dongle in the V<sup>x</sup>670 terminal. The USB modem driver code is actually imported from the V<sup>x</sup>670 OS without any changes except to recognize the new DUET USB modem PID.

### NOTE



The modem does not operate unless a model profile is loaded on the V<sup>x</sup>810 PIN pad. Note that the V<sup>x</sup>810 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 V<sup>x</sup>810 PIN pad can be connected to a DUET, but the modem does not operate until a modem profile is present in the V<sup>x</sup>810 PIN pad memory and loaded into the modem device. Modem profile naming rules and loading conventions are identical to that of the V<sup>x</sup>670 terminal.

The USB modem in the V<sup>x</sup>810 DUET operates exactly the same as the USB modem in the V<sup>x</sup>670 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.

## V<sup>x</sup>810 DUET Modem Device API

When the V<sup>x</sup>810 PIN pad operates in a stand-alone mode, all modem related APIs return -1 with `errno` set to ENODEV.

When the V<sup>x</sup>810 PIN pad is connected to a DUET, the modem related APIs return normal response codes (e.g., `open()` returns a valid handle). In this mode, the V<sup>x</sup>810 PIN pad presents the same API and behavior as the V<sup>x</sup>670 terminal when the external modem dongle is connected.

## USB External Serial (COM6)

V<sup>x</sup>670 allows connection to other serial devices via the USB UART dongle, apart from the regular UART port. The USB dongle has a USB connector on one end and an RJ-45 jack on the other. For more information, refer to [USB External Serial \(COM6\)](#).

On the V<sup>x</sup>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<sup>x</sup>670 terminal base.

When the Vx810 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 Vx810 or on a Vx810 with DUET.

The presence of the VID/PID Hub indicates to the OS that it is running on a Vx810 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 Vx810 DUET. the Vx810 DUET does not contain USB device power control circuitry so the USB Hub cannot control the device power.

## **USB Internal WiFi (WLN1)**

Vx670 uses the Conexant CX3889 USB WiFi chip (Partagas) as its WiFi device. It is connected internally so it cannot be physically connected and disconnected. However, when the Vx670 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 will result in connect and disconnect events, respectively.

The driver accepts the following vendor and product identification combinations. Production 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 144      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 is 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 chips MAC address. This addresses is stored so the status function `get_enet_MAC()` can returned the MAC address. The entire packet is also stored in the Readahead buffer so that applications will also see this packet.

The Conexant Partagas 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 V API function `remove()` as in:

```
remove( "F:#PRISMA09.X2" )  
remove( "F:#PRISMA09.X2.P7S" );
```

or by using DDL as in:

```
dd1 -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 `set_enet_rx_control()` both do not apply to WiFi. The status function `get_enet_MAC()` is supported due to many applications directly using this feature.



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](#) section on [WWAN](#).

**USB WiFi Functions** The following functions are used in Vx670 USB WiFi:

## open()

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

**Prototype**    `int open (" /dev/wln1 ", int unused );`

### Return Values

Success: Handle.

Failure: -1 and `errno` set to `EBUSY` (returned by the OpSys if another task currently owns the device) or `ETIMEOUT` (if the device is connected but not completely initialized).

## read()

Allows the current owner to read one packet from the WiFi device.

**Prototype** `int read (int handle, char *buffer, int count);`

**Return Values**

Success: Number of bytes transferred.

Failure: -1 and `errno` set to `EINVAL`.

The caller should ensure that the buffer can hold the maximum packet size that can be received (1514 bytes).

---

**NOTE**



Packet truncation is impossible since the caller must specify the maximum count of 1514 bytes. Any count less than 1514 is considered an error.

---

## write()

This allows the current owner to write one packet for the WiFi device.

**Prototype**    `int write (int handle, const char *buffer, int count);`

### Return Values

Success: Number of bytes transferred.

Failure: -1 and `errno` set to `ENOSPC`. This means that the buffer space does not exist for the entire packet and the call fails with no data transferred.

Ethernet packets must not contain more than 1514 bytes. If the caller's count parameter is larger, the call is rejected. The result will be -1 with `errno` set to `EINVAL`. Unlike USB Ethernet, there is no minimum packet size because PIMFOR packets can be smaller than the minimum legal Ethernet data packet size.

---

### NOTE



A count of 0 in this case is not considered an error. Instead, `errno` will not be set.

---

## **close()**

This call releases the ownership of the WiFi device so that another application can use it.

**Prototype**    `int close (int handle);`

### **Return Values**

Success: 0

Failure: -1 and `errno` set to `EBADF`. This means that the buffer space does not exist for the entire packet and the call fails with no data transferred.

---

#### **NOTE**



Failure only occurs if the caller is not the current owner.

---

## **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 Conexant Partagas 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 Partagas chip intercepts management packets so they are not transmitted over the radio. Refer to the [Partagas] Conexant CX3889 Driver Programmers Manual, Document # DO-412064-TC Draft 1D, April 28, 2005, for more information.

WiFi data and management packets are received/sent to the Verix V `read()` / `write()` API functions after opening the WiFi device “/DEV/WLN1”. Verix V 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. See [Partagas] Conexant CX3889 Driver Programmers Manual, Document # DO-412064-TC Draft 1D, April 28, 2005, for additional information on PIMFOR. But the PIMFOR magic number has changed from one byte 0x88 to 4 bytes 0x50494D34. See the memo “3889 Host Interface Description.doc” for new PIMFOR format.

**Table 145 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 146 defines PIMFOR values and their description.

**Table 146 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 Verix V.
- The Verix V 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.



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 describes how the PIMFOR packet should look like in Verix V:

**Table 147 PIMFOR Packet on Verix V**

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 148 describes the firmware loading process.

**Table 148 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 149 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 149 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

## 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 V<sup>x</sup>810 PIN pad cannot switch between HID client and RS-232 client without a restart. The mechanism to determine the USB client device is a new CONFIG.SYS parameter, \*USBCLIENT. The \*USBCLIENT is allowed to have two values; HID and RS232. If no \*USBCLIENT CONFIG.SYS parameter is present, the OS uses RS232 as the default.

**NOTE**



This is a change from the Verix V terminal, which uses HID as the default USB client mode.

On V<sup>x</sup>510 GPRS, 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.

### 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 V<sup>x</sup>810 PIN pad. The System Mode 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. The Verix PIN pads such as the V<sup>x</sup>800 and the V<sup>x</sup>810 PIN pad provide a VXUART.inf file for the customer. The .inf file is installed to Windows and it identifies the connected device as either a "V<sup>x</sup>810 PIN Pad" or "V<sup>x</sup>800." When the OS boots on this mode, the standard DDL can be used on the host. The System Mode 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 V<sup>x</sup>810 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 V<sup>x</sup>810 PIN pad.

A developer cable with USB device and USB Host connector is available. When used, the V<sup>x</sup>810 PIN pad inherits the USB connect/disconnect tones in the V<sup>x</sup>670 terminal. The tones may sound different on the V<sup>x</sup>670 terminal and the V<sup>x</sup>810 PIN pad depending on the buzzer hardware and case design. The difference in tones is outside the control of the OS.

The CONFIG.SYS variable, \*USBMEM, is used for USB Host operations as on previous Verix V platforms.

On V<sup>x</sup>510 GPRS, the USB Host port provides 500mA at 5 volts to power up the USB devices it supports. [Table 150](#) presents the PID, VID, and the device driver of the USB Host port supported devices.

**Table 150      Supported Devices**

<b>USB Devices</b>	<b>PID</b>	<b>VID</b>	<b>Device Driver</b>
PP1000se none CTLS	0X1000	0X193A	DEV_USBSER
V <sup>x</sup> 810	0X0207	0X11CA	DEV_USBSER
PL2303 USB to RS-232 converter device (Teletec and ViVo cables)	0x2303	0x067b	DEV_COM6



If the V<sup>x</sup>810 unit has a CTLS module, the cable for use in the V<sup>x</sup>510 GPRS/V<sup>x</sup>810 connection requires an external power source.

### **Supported Devices**

The V<sup>x</sup>510 GPRS USB Host port supports the following USB devices:

- COM6/PL2303 USB to RS-232 Converter Device

The V<sup>x</sup>510 GPRS supports the Qx120 device via COM6 using the Teletec and ViVotech cables. IT also supports the VeriFone USB UART dongle currently on V<sup>x</sup>670, V<sup>x</sup>810, and V<sup>x</sup>570 devices. COM6 also supports System Mode and auto-download processes.

- PINpad1000se and V<sup>x</sup>810 Device

The Vx510 GPRS supports the PINpad 1000SE and Vx810 devices by emulating USBSER.SYS.

**NOTE**



The Vx810 PIN pad and the Vx810 DUET are both supported because they report the same VID and PID.

These devices are opened using the “/DEV/USBSER” or DEV\_USBSER device name. These also use the APIs `set_event_bit()` and `get_event_bit()`.

The following device bits for these devices are defined on SVC.H. The API `get_usb_device_bits()` returns the following:

```
#define UDB_VX810      (1<<8)  
#define UDB_PP1000SE   (1<<9)
```

The OS presents the standard device API to the application and issues `open()`, `close()`, `read()`, and `write()` functions.

The OS allows the application to use the USB interface the same way it would use a standard UART-based COM port. The OS returns “success” for API calls like `set_opn_blk()` even when there is no actual effect on the USB interface. The `set_opn_blk()` is called before starting to communicate with the USB devices to allow existing applications, which expect a standard COM port interface, to use the USB port with minimal code changes.

The functions `set_opn_blk()`, `reset_port_error()`, `set_serial_lines()`, `set_fifo_config()`, `get_open_blk()`, `get_fc_config()`, `set_fc_config()`, `get_port_status()`, and `get_fifo_config()` returns “success” since there is no physical UART.

### Other Supported USB Devices

The following USB devices are also supported on the Vx510 GPRS Host Port. They are previously implemented on the Vx570 terminal and Vx700/Vx810 PIN pads.

- Heron D130 barcode reader device
- MSO300 Fingerprint scanner device
- USB keyboard device
- VFI Ethernet dongle

## USB Thermal Printer

The Vx810 DUET contains a micro-controller (MCU) connected to a thermal print mechanism. The OS communicates with the printer MCU through the USB UART, which is functionally the same as the USB UART dongle on the Vx670 terminal.

The Vx670 USB UART driver attaches the USB UART PID to the printer device (COM4) for the Vx810 PIN pad. A new USB device definition, UDB\_COM4, is set in the USB device status.



**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 V 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 printer driver sends the value of \*PRTFNT, if present, to the MCU. On the Vx810 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 printer driver sends the value of \*PRTLGO, if present, to the MCU. On Vx810 DUET, \*PRTLGO is not used, thus, it has no effect on the printer operation.

#### Printer ID

Because the Vx810 DUET printer driver provides a completely different architecture, the command "Request Printer ID" (<ESC> "i") returns a value of "Q" instead of "P" as in other Verix V printer drivers.

#### 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.

#### MCU Firmware download

The Vx810 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 Vx810 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 System Mode. 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 System Mode 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 Vx810 PIN pad is connected to a DUET base station, the OS returns a valid handle.

## Print Buffer Management

The Vx810 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 Vx810 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 System Mode. The hardware is not power cycled in a soft reboot as opposed to a hard reboot where power is cycled.

The Vx810 DUET printer behaves like an external printer, and is not aware that the main CPU in the Vx810 has restarted. The MCU retains any pending print data in its print buffer before the Vx810 PIN pad is restarted. This data needs to be flushed so that the next time the Vx810 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).

## **USB Device Drivers**

The Verix OS supports the following USB device drivers:

- USB fingerprint reader
- USB barcode scanner
- USB keyboard
- USB to RS-232 converter
- Metrologic Barcode scanner
- MC5727 USB driver

Refer to [System Devices](#) for more information.





## APPENDIX J

### ASCII Table

An ASCII table for the Vx510/Vx610 Verix® V Operating System display is presented as [Figure 43](#) 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 “^\_”.

## ASCII TABLE

### Control Characters

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	1	1	1	1	1	1	1	1	1	1	1	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	
Most-Significant Nibble	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	
	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0	
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	
	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-	
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	
	~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	
	p	q	r	s	t	u	v	w	x	y	z	{		}	~	█	

Figure 43 ASCII Table For Verix V-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 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 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 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.

**volatile RAM** Storage area in RAM for all global and local data and the system stack and heap.

**WWAN** Wireless Wide Area Network



## Symbols

( 232, 234, 238, 241  
\* 142, 143, 144  
\*COM2RB 143  
\*COM3RB 143  
\*COMBO 142  
\*DOT0 143  
\*DOT1 143  
\*MERR 143  
\*MN 143  
\*USBMEM 145  
\*ZTCP 145  
\*ZX 145

## Numerics

1DES master key 621

## A

abort 48  
acronym list 24  
activate\_task() 229  
AES() 594  
Allows 259  
ALPHA key 219  
alpha\_shift() 230  
ANSI X9.19 message authentication code 690  
applications  
    event word 191  
    execute on startup 150  
    sleep mode 191  
    sponsoring 150  
ARM UART receive FIFO 582  
ASCII table 803  
asynchronous cards 509  
ATS packet form 524  
automatic file decompression 725  
auto-repeating keys 227

## B

backlight 256, 266  
back-to-back downloads 150  
battery level 45, 755  
battery status 45, 755

BatteryRegs() 756  
baud rate 511  
baud rates 515  
beeper functions. See *functions, beeper*  
BRK 584  
buffers  
    application 368  
    device 586  
    device driver 586  
    insufficient 177  
    key buffer 256  
    OS 341  
    port status 577  
    transmit 586  
BWT 494, 503, 506

## C

card swipe 342  
card\_pending() 339  
CCITT algorithm 322  
certificate trees 716  
    default 717  
    factory 717  
    initial customer 717  
    second customer 718  
    synchronized 717  
certificates for file authentication 720  
character mode initialization 511  
character set 399  
checksum 328  
checksum validation 148  
choke 536  
choke trigger level 536  
clock 365  
clock functions See *functions*.  
clock handle 366  
clock ticks 372  
close file 115  
clr\_timer() 194  
clreol() 232  
clrscr() 233  
coalesce 136  
coalesce flash 138  
coalesce memory 137

COM2HW **142**  
COM3HW **142**  
communication sequence **525**  
compress ASCII data **704**  
compressed files **725**  
compressed variable-length record (CVLR) **62, 81, 87, 95, 704**  
    files **74**  
CONFIG.SYS **62, 141, 720**  
CONFIG.SYS environment variables  
    \*APNAME **142**  
    \*ARG **142, 147**  
    \*B **142, 147**  
    \*BCM **142**  
    \*DBMON **143**  
    \*DEFFRAG **143, 149**  
    \*DIAG **143**  
    \*FA **143, 150**  
    \*FILE **73, 143, 150**  
    \*GO **62, 143, 150**  
    \*IPPMKI **143, 151**  
    \*MERR **152**  
    \*MIN **152**  
    \*name **146**  
    \*PIPE **144**  
    \*PW **144, 153**  
    \*SMDL **144, 153**  
    \*SMGIDS **144**  
    \*SMPW **144, 154**  
    \*SMUI **144**  
    \*TIME **144, 154**  
    \*UNZIP **144, 154**  
    \*VALID **145, 154**  
    \*ZA **145, 532**  
    \*ZINIT **145**  
    \*ZM **532**  
    \*ZP **145**  
    \*ZRESET **145**  
    \*ZRESP **145**  
    \*ZSWESC **145**  
    \*ZT **145, 532**  
CHKSUM **142, 148**  
UNZIP **144, 154**  
console  
    determine the handle for **246**  
    ownership **228**  
    sharing **161**  
    transfer ownership **228**  
console functions. See *functions, console*  
contrast setting **256**  
contrast\_down() **234**  
    contrast\_up() **235**  
conventions  
    filenames **72**  
    measurement **23**  
counted strings **282**  
country setting for terminal **284**  
crash log **285**  
CRC **318**  
    CRC functions. See *functions, CRC*.  
    CRC, cyclic redundancy check **318**  
    CRC16 algorithm **324**  
    CRC32 algorithm **326**  
crypto libraries **587**  
crypto\_read() **590**  
crypto\_write() **591**  
cryptography **720**  
cryptography functions **593**  
CTS **536**  
current key **688**  
current owner variable **695**  
cursor  
    positioning **250**  
    turn on/off **267**  
CVLR. See also, *compressed variable-length record*. **62**  
CWT **503**  
**D**  
data  
    format **511**  
date setting **368**  
    time setting **720**  
date/time **364**  
date2days() **364**  
datetime2seconds() **364**  
day-of-week setting **368**  
days2date() **364**  
decode tables **343**  
decoder **343**  
decompress ASCII data **705**  
decompress files **725**  
decompression  
    application interface **726**  
    implementation **728**  
    performance **728**  
    user interface **727**  
decrypt\_session\_data() **488, 684**  
decryption **488, 684**  
default font. See *font*  
delete

key **119**  
record pair **119**  
deleting files **101**  
`delline()` **236**  
DES encryption **491, 686**  
`DES()` **595**  
device  
    handles **332**  
    names **332**  
device drivers  
    console **228**  
    serial printer **534**  
device management functions. See *functions*,  
*device management*  
device ownership **335, 336**  
devices  
    beeper **374**  
    capability enumeration **499**  
    console **209**  
    magnetic card reader **337**  
    real-time clock **362**  
`dir_flash_coalesce()` **137**  
`dir_flash_coalesce_size()` **138**  
`dir_get_attributes()` **121**  
`dir_get_file_date()` **122**  
`dir_get_file_size()` **123**  
`dir_get_first()` **125**  
`dir_get_next()` **126**  
`dir_get_sizes()` **127**  
`dir_put_file_date()` **128**  
`dir_reset_attributes()` **129**  
`dir_set_attributes()` **130**  
Direct Download Utility (DDL) **711**  
`disable_hot_key()` **237**  
`disable_key_beeps()` **238**  
display **209**  
    ASCII table **803**  
display contrast **241**  
download  
    result messages **728**  
download fonts **433**  
download function call **573**  
`download()` **573**  
downloads **573**  
    and file authentication **721**  
back-to-back **712, 716**  
    certificate revisions **718**  
    interrupting **714, 719**  
    invalid direction **718**  
    valid direction **718**  
download-and-resume **531**  
from system mode **713**  
full **531**  
partial **531**  
drivers **280**  
    console **228**  
    mag driver return codes  
        **MAG\_BADJS 343**  
        **MAG\_BADLRC 343**  
        **MAG\_BADTRK 343**  
        **MAG\_NODATA 343**  
        **MAG\_NOERR 343**  
        **MAG\_NOETX 343**  
        **MAG\_NOSTX 343**  
        **MAG\_PARITY 343**  
        **MAG\_REVETX 343**  
    See also, *device drivers* **534**  
`DTR` **536, 584**  
`DUKPT`  
    **IPP7 662**

## E

`_exit()` **163**  
`EBC` **503**  
`EMV PINs` **682**  
`EMV smart card` **682**  
`enable_hot_key()` **239**  
`enable_key_beeps()` **240**  
encrypted session key **683**  
encryption key **682**  
environment functions. See *functions*.  
error beep **374**  
error codes  
    API interface  
        **EACCES 342, 689, 690, 691, 693**  
        **EBADF 339, 340, 342, 689, 690, 691, 692, 693, 694, 695**  
        **EINVAL 342**  
        **ENODEV 341**  
        **ENOSPC 341, 689, 690, 691, 693, 694**  
        **EBADF 123, 160, 178, 583**  
        **EINVAL 200, 370**  
        **ENODEV 84**  
        **ENOMEM 181**  
        **ENOSPC 200**  
    files  
        **EEXIST 82**  
        **EFBIG 79**  
        **ENOENT 102**  
        **ENOSPC 79**  
    flash

- EACCES **134, 136**
- EBADF **135**
- EBUSY **134, 136, 138**
- EINVAL **135**
- ENOSPC **136**
- keyed files
  - EBADF **119**
- RS-232 port
  - EACCES **582**
  - EBADF **575, 576, 582**
  - EINVAL **574, 582**
- serial printer port
  - EACCES **535, 538, 584**
  - EBADF **535, 538, 584**
  - EINVAL **538**
- error log **285**
- error message
  - flash insufficient **715, 719**
- error messages
  - insufficient flash **715, 719**
  - insufficient RAM **715, 719**
- error\_tone() **378**
- errors
  - frame **577**
  - overrun **537, 577**
  - parity **577**
- event **191**
- event functions. See *functions, event*
- event mask **203**
- events
  - EVT\_CLOCK **367**
  - EVT\_DEACTIVATE **256**
  - EVT\_KBD **228, 256**
- exception handling **193**
- exceptions **193**
- external PIN pad **681**
  
- F**
- \*FA **150**
- fatal exceptions **285**
- FIFO
  - configuration **575**
- FIFO functions. See *functions, FIFO*.
- FIFOs **311**
- file attribute functions. See *functions, file attribute*
- file authentication **716, 720**
  - \*FA variable and **150**
  - certificates
    - default **717**
    - initial customer certificate tree **717**
    - partition **717**
- root **717**
- Signer1 **717**
- Signer2 **718**
- sponsor **717**
- downloads and **721**
- k2sign.crt default certificate file **721**
- K2sign.key default key file **721**
- key certificate and **720**
  - signature file **720**
- file decompression, automatic **725**
- file directory functions. See *functions, file directory*
- file extension **75**
- file functions. See *functions*.
- file groups **72, 726**
  - file sharing and **76**
  - Group 0 **72, 76**
  - Group 1 **72, 76, 719**
  - Group 15 **76**
  - group ID feature **76**
  - Groups 2-14 **76**
- system mode password setting variable **144**
- visibility **76**
- file groups, and decompressing files **726**
- file handle **83, 115**
  - termination **116**
- file information functions. See *functions*.
- file lock **161**
- file management guidelines **75**
- file manager **75, 76, 77, 134, 135**
- file naming conventions **72**
- file pointer **82**
- file position pointer **90, 95**
- file positioning functions. See *functions, file positioning*.
- file signing tool **720**
- file size **97**
- file size sloshing **113**
- file updates **99**
- files
  - .p7s
- automatic decompression in terminal memory **725**
- close file **115**
- code file **159**
- compressed and downloads **725**
- conventions for **71**
- create a new file **82**
- decompression, and groups **726**
- deleting files **101**
- file directory **120**
  - first file determination **125**

subsequent file determination **126**  
file directory information **127**  
file handle **73, 83, 115**  
file position pointer **90, 95**  
filenames **72**  
    and group ID **76**  
    private name space for **76**  
font **268**  
getting file size information **97**  
Group 0 **76**  
groups **72**  
insert data into **94**  
keyed **74, 116**  
maintaining file headers **77**  
maximum number open setting variable **143**  
new file **82**  
padding **76**  
paired **74**  
position pointer in 83  
    See Also *seek pointer, seek*  
primary user group **72**  
renaming files **106**  
SETDRIVE **720**  
SETGROUP **720**  
signature **150, 720**  
signing **720**  
slosh **79**  
sloshing **113**  
store files **71**  
transfer data to/from **86**  
update information **99**  
variable-length (VL) **74**  
write access **82**  
writes  
    seek to the end **82**  
fixed expansion area feature **80**  
fixed-length records **91**  
flash  
    \*DEFRAG CONFIG.SYS variable **143**  
    amount installed **139**  
    coalesce **138, 149**  
    coalescing **134**  
    defragmentation **143**  
    defragmenting **149**  
    determine size **283**  
    erase **136**  
    file restrictions **134**  
    file system **159**  
    full **722**  
    insufficient error message **715, 719**  
    insufficient FLASH error **715, 719**  
    memory **71**  
flash directory functions. See *functions*.  
flash file system **85, 134, 137**  
    file manager and **134**  
    file manager functions not supported **135**  
    writes to **136**  
flash files  
    close **136**  
    open **136**  
flow control **535, 536, 584**  
font  
    font file **268**  
font data organization **399**  
font definition file **260**  
font files  
    .fon **210**  
Font Generation tool **268**  
fonts  
    5 × 8 **432**  
    8 × 14 **432**  
    default **209, 228**  
    default209  
    download fonts **433**  
    font files **210**  
    Font Generation tool **268**  
    tables **432**  
forward slash **76**  
frame error **577**  
functions  
    beeper  
        close() **377**  
        error\_tone() **378**  
        key\_beeps() **379**  
        normal\_tone() **380**  
        open() **381**  
        sound() **382**  
    clock  
        date2days **364**  
        datetime2seconds **364**  
        days2date **364**  
        seconds2datetime **364**  
        secs2time() **364**  
        SVC\_VALID\_DATE **364**  
        time2secs() **364**  
    console  
        activate\_task() **229**  
        alpha\_shift() **230**  
        BatteryRegs() **756**  
        close() **231**  
        clreol() **232**  
        clrscr() **233**  
        contrast\_down() **234**  
        contrast\_up() **235**

delline() **236**  
disable\_hot\_key() **237**  
disable\_key\_beeps() **238**  
enable\_hot\_key() **239**  
enable\_key\_beeps() **240**  
get\_battery\_initialization\_status() **756**  
get\_console() **246**  
get\_font() **247**  
get\_font\_mode() **248**  
getcontrast() **241**  
getfont() **242**  
getgrid() **243**  
getinverse() **244**  
gotoxy() **250**  
inverse\_toggle() **252**  
put\_graphic() **258**  
read() **259**  
screen\_size() **261**  
set\_backlight() **266**  
set\_bcm() **212**  
set\_com1\_pwr() **756**  
set\_cursor() **267**  
set\_font() **268**  
set\_hot\_key() **269**  
setinverse() **264**  
SVC\_INFO\_DISPLAY() **270**  
SVC\_INFO\_KBD() **271**  
whereloc() **272**  
wherewin() **273**  
wherewinloc() **274**  
window() **275**  
write() **276**  
write\_at() **277**

CRC

- SVC\_CRC\_CALC() **320**
- SVC\_CRC\_CALC\_L() **321**
- SVC\_CRC\_CCITT\_L() **322**
- SVC\_CRC\_CITT\_M() **323**
- SVC\_CRC\_CRC16\_L() **324**
- SVC\_CRC\_CRC16\_M() **325**
- SVC\_CRC\_CRC32\_L() **326**
- SVC\_LRC\_CALC() **327**
- SVC\_MEMSUM() **328**
- SVC\_MOD\_CK() **329**

device management

- get\_name() **334**
- get\_owner() **335**
- set\_owner() **336**

event

- clr\_timer () **194**
- peek\_event () **195**
- read\_event() **197**

set\_timer () **200**  
SVC\_WAIT() **202**  
wait\_event () **203**

FIFO

- SVC\_CHK\_FIFO() **312**
- SVC\_CLR\_FIFO() **313**
- SVC\_GET\_FIFO() **314**
- SVC\_PUT\_FIFO() **315**
- SVC\_READ\_FIFO() **316**
- SVC\_WRITE\_FIFO() **317**

file access

- read() **87**
- read\_cvr() **87**
- read\_vlr() **87**
- write() **89**
- write\_cvr() **89**
- write\_vlr() **89**

file attribute

- get\_file\_attributes() **109**
- get\_file\_max() **110**
- reset\_file\_attributes() **111**
- set\_file\_attributes() **112**
- set\_file\_max() **113**

file directory

- dir\_get\_attributes() **121**
- dir\_get\_file\_date() **122**
- dir\_get\_file\_size() **123**
- dir\_get\_first() **125**
- dir\_get\_next() **126**
- dir\_get\_sizes() **127**
- dir\_put\_file\_date() **128**
- dir\_reset\_attributes() **129**
- dir\_set\_attributes() **130**

file information

- get\_file\_date() **99**
- get\_file\_size() **98**
- SVC\_CHECKFILE() **100, 148**

file positioning

- Iseek() **91**
- seek\_cvr() **91**
- seek\_vlr() **91**

files

- close() **115**
- delete() **95**
- delete\_cvr() **95**
- delete\_vlr() **95**
- insert() **94**
- insert\_cvr() **94**
- insert\_vlr() **94**
- remove() **102**

flash

- dir\_flash\_coalesce() **137**

dir\_flash\_coalesce\_size **138**  
SVC\_FLASH\_SIZE() **139**

get\_env() **157**

IPP  
    get\_port\_status() **391**

keyed files  
    getkey() **118**  
    putkey() **119**

magnetic card reader  
    card\_pending() **339**  
    close() **340**  
    open() **341**  
    read() **342**

Master/Session **683**  
    decrypt\_session\_data() **684**  
    gen\_master\_key() **685**  
    gen\_session\_key() **686**  
    test\_master\_key() **687**

One Way **691**

pipe interface  
    close() **180**  
    open() **181**  
    pipe\_connect() **182**  
    pipe\_init\_char() **178, 184**  
    pipe\_init\_msg () **43, 168, 172, 173, 174, 175, 176, 178**  
    pipe\_init\_msg() **185**  
    pipe\_pending() **186**  
    read() **187**  
    write() **188**

power management  
    get\_battery\_sts() **45, 755**  
    get\_battery\_value **45, 755**  
    get\_dock\_sts() **45, 755**  
    get\_powersw\_sts() **46, 756**

put\_env() **158**

real-time clock  
    close() **366**  
    open() **367**  
    read() **368**  
    write() **370**

Reset\_Key **688**

RS-232  
    close() **572**  
    download() **573**  
    get\_component\_vars() **574**  
    get\_fifo\_config() **575**  
    get\_opn\_blk() **576**  
    get\_port\_status() **577**  
    open() **579**  
    read() **580**  
    reset\_port\_error() **581**

set\_fifo\_config() **582**  
set\_opn\_blk() **583**  
SVC\_ZONTALK() **531**  
write() **586**

security/crypto library  
    AES() **594**  
    crypto\_read() **590**  
    crypto\_write() **591**  
    DES() **595**  
    GenerateRandom() **596**  
    isAttacked() **597**  
    rsa\_calc() **598**  
    SHA1() **599**

serial printer port  
    get\_fc\_config() **535**  
    set\_fc\_config() **536**  
    set\_serial\_lines() **584**

service  
    SVC\_VALID\_DATE() **364**  
    SVC\_WAIT() **373**

smart card  
    decrypt\_session\_data() **488**  
    gen\_master\_key() **489**  
    gen\_session\_key() **491**  
    test\_master\_key() **492**

string conversions  
    Itoa() **698**  
    strnlwr() **700**  
    strnupr() **701**  
    SVC\_2INT() **709**  
    SVC\_AZ2CS() **707**  
    SVC\_CS2AZ() **710**  
    SVC\_DSP\_2\_HEX() **703**  
    SVC\_HEX\_2\_DSP() **702**  
    SVC\_INT2() **708**  
    SVC\_PACK4() **704**  
    SVC\_UNPK4() **705**  
    ultoa() **699**

SVC\_RESTART **189**

system information  
    get\_component\_vars() **280**  
    SVC\_CHK\_PASSWORD() **282**  
    SVC\_FLASH\_SIZE() **283**  
    SVC\_INFO\_COUNTRY() **284**  
    SVC\_INFO\_CRASH() **285**  
    SVC\_INFO\_DISPLAY() **270, 286**  
    SVC\_INFO\_EPROM() **287**  
    SVC\_INFO\_HW\_VERS() **288**  
    SVC\_INFO\_LIFETIME() **290**  
    SVC\_INFO\_LOTNO() **291**  
    SVC\_INFO\_MFG\_BLK() **293**  
    SVC\_INFO\_MOD\_ID() **294**

- SVC\_INFO\_MODELNO() **297**
- SVC\_INFO\_OS\_HASH() **310**
- SVC\_INFO\_PARTNO() **298**
- SVC\_INFO\_PIN\_PAD() **299, 556**
- SVC\_INFO\_PORT\_MODEM() **301**
- SVC\_INFO\_PRNTR() **302**
- SVC\_INFO\_PTID() **303**
- SVC\_INFO\_RESET() **304**
- SVC\_INFO\_SERLNO() **305**
- SVC\_PORT\_IR() **300**
- SVC\_RAM\_SIZE() **308**
- SVC\_VERSION\_INFO() **309**
- tasks
  - \_exit() **163**
  - get\_task\_id() **165**
  - get\_task\_info() **166**
  - run **167**
  - set\_group() **169**
- time
  - date2days() **364**
  - datetime2seconds() **364**
  - days2date() **364**
  - read\_clock() **365**
  - SVC\_VALID\_DATE() **364**
  - SVC\_WAIT() **373**
- TXO. See *TXO functions*.
- VSS
- iPS\_CancelPIN() **606**
- iPS\_CheckMasterKey() **613**
- iPS\_DeleteKeys() **614**
- iPS\_ExecuteScript() **603**
- iPS\_GetPINResponse() **607**
- iPS\_GetScriptStatus() **601**
- iPS\_InstallScript() **602**
- iPS\_LoadMasterClearKey() **615**
- iPS\_LoadMasterEncKey() **616**
- iPS\_LoadSysClearKey() **617**
- iPS\_LoadSysEncKey() **618**
- iPS\_RequestPINEntry() **609**
- iPS\_SelectPINAlgo() **610**
- iPS\_SetPINParameter() **611**
- iPS\_UninstallScript() **604**
- G**
  - gen\_master\_key() **489, 685**
  - gen\_session\_key() **491, 686**
  - GenerateRandom() **596**
  - get PIN request **683**
  - get\_battery\_initialization\_status **756**
  - get\_battery\_sts() **45, 755**
  - get\_component\_vars() **280, 574**
- H**
  - hardware flow control
    - See also *flow control*. **536**
  - high-speed communication **536**
  - host key reset **694**
  - host key update **693**
  - hot key **237, 269**
- get\_console() **246**
- get\_dock\_sts() **45, 755**
- get\_env() **141, 157**
- get\_fc\_config() **535**
- get\_file\_attributes() **109**
- get\_file\_date() **99**
- get\_file\_max() **77, 110**
- get\_file\_size() **98**
- get\_font() **247**
- get\_font\_mode() **248**
- get\_group **76**
- get\_name() **334**
- get\_owner() **335**
- get\_port\_status() **391, 577**
- get\_powersw\_sts() **46, 756**
- get\_task\_id() **165**
- get\_task\_info() **166**
- getcontrast() **241**
- getfont() **242**
- getgrid() **243**
- getinverse() **244**
- getkey() **118**
- GID
  - password **282**
  - see also, *groups* **720**
- GISKE
  - key attribute **620**
  - KLK key loads **621**
- global variables
  - device names **332**
- gotoxy() **250**
- Group 0 **76**
- Group 1 **72, 76**
- Group 15 **76**
- Group ID **76**
- group password compare **282**
- groups
  - Group 1 **720**
  - Group 15 **720**
  - public **720**
- Groups 2-14 **76**

hot key status **249**

## I

infrared device **300**

int **34, 39, 43**

internal PIN pad key loading **143**

interrupts **363**

inverse\_toggle() **252**

IPP

key attributes **619**

key length **619**

key version **619**

IPP7 **619**

DUKPT modes **662**

iPS\_CancelPIN() **606**

iPS\_CheckMasterKey() **613**

iPS\_DeleteKeys() **614**

iPS\_ExecuteScript() **603**

iPS\_GetPINResponse() **607**

iPS\_GetScriptStatus() **601**

iPS\_InstallScript() **602**

iPS\_LoadMasterClearKey() **615**

iPS\_LoadMasterEncKey() **616**

iPS\_LoadSysClearKey() **617**

iPS\_LoadSysEncKey() **618**

iPS\_RequestPINEEntry() **609**

iPS\_SelectPINAlgo() **610**

iPS\_SetPINParameter() **611**

iPS\_UninstallScript() **604**

isAttacked() **597**

## K

k2sign.crt **721**

k2sign.key **721**

key **91, 621**

beep **374**

delete **119**

maximum length **116**

key attributes, IPP **619**

key certificate **720**

key FIFO **228**

key loading **612**

key\_beeps() **379**

keyed file functions. See *functions, keyed files*

keyed files **74, 116**

rules **116**

key-loading program **688**

keypad **209**

ALPHA key use **219**

color coded function keys **214**

keys **219**

APACS40 **688**

auto-repeating **227**

color coded function **214**

current **688**

current, reset **694**

host **690, 691, 693, 694**

hot key **161**

MAC **689**

programmable function keys **64**

seed **688, 694**

session **491, 686**

keys and file authentication **720**

key-value pairs **62**

KLK (GISKE) **621**

## L

LCD display **209**

load security script keys **612**

lock files **161**

LRC, longitudinal redundancy check **327**

Iseek() **82**

Itoa() **698**

Luhn check **329**

## M

MAC **689**

MAC value **622**

MAG\_BADJS **343**

MAG\_BADLRC **343**

MAG\_BADTRK **343**

MAG\_NODATA **343**

MAG\_NOERR **343**

MAG\_NOETX **343**

MAG\_NOSTX **343**

MAG\_PARITY **343**

MAG\_REVETX **343**

magnetic card reader **337**

magnetic card reader functions. See *functions, magnetic card reader*

main task **159**

management information block (MIB) **228**

manufacturing block **293**

master key **489, 682, 685**

master key sessions **487**

Master/Session **597**

master-session key management **682**

measurement conventions **23**

memory  
  coalesce **137**  
flash  
  coalesce **137**  
  file transfers **719**  
  insufficient FLASH error **715, 719**  
padding **75**  
RAM  
  file transfers **719**  
  insufficient RAM error **715, 719**  
memory management unit (MMU) **57**  
memory sloshing **113**  
**MIB 228, 256**  
model number **297**  
modem  
  determine port number **301**  
  escape to command mode **145, 155**  
  handshake **525**  
  initialization string setting variable **145, 155**  
  reset string setting variable **145**  
  response setting variable **145**  
modes  
  character **512**  
  dot graphic mode **399**  
  double width mode **399**  
  SDLC **525**  
  system mode **64**  
    system mode entry **226, 713**  
Modulo 8 information packets **524**  
multitasking **159**

## N

new file **82**  
no receive choke **537**  
non-volatile RAM **57**  
normal beep **374**  
normal\_tone() **380**

## O

On **771**  
One Way function **691**  
operating system  
  version information **309**  
**Opn\_Blk 511, 576**  
overrun error **577**  
overrun errors **577**

## P

.p7s file See *file authentication*  
packet format

Modulo-128 **525**  
Modulo-8 **525**  
supervisory **525**  
unnumbered information **525**  
packet formats **524**  
padding files **76**  
parity error **577**  
part number **298**  
password **282, 720**  
passwords **578**  
peek\_event() **195**  
PIN block **683**  
PIN entry functions **605**  
PIN pad **299, 556**  
  communications parameter variable **143**  
  external **681**  
  key loading **143**  
PINs **682**  
pipe **159**  
pipe interface functions. See *functions*.  
pipe\_connect() **182**  
pipe\_init\_char() **178, 184**  
pipe\_init\_msg() **43, 168, 172, 173, 174, 175, 176, 178**  
pipe\_init\_msg() **185**  
pipe\_pending() **186**  
pipes **177**  
  anonymous **181**  
  character **177**  
  closing **180**  
  function calls for configuring **178**  
  handles **182**  
  maximum number **177**  
  message **177**  
  naming **181**  
play **375**  
play\_RTTTL() **375**  
port number **301**  
ports  
  auxiliary **536**  
  COM for SDLC **525**  
  COM2 **516, 681**  
  external PIN pad (COM2) **516, 681**  
  external printer **537**  
  internal printer **537**  
  PIN pad **539**  
  printer **536**  
  RS-232 **515**  
  serial **512**  
  serial printer **537**  
post\_user\_event() **196**

power management **46, 756**

printer

    determine type installed **302**

protocol-specific parameters **511**

put\_env() **141, 158**

put\_graphic() **258**

putkey() **119**

## R

RAM

    determine size of **308**

    error messages

        insufficient RAM **715, 719**

    non-volatile **71**

    RAM-based file system **85**

read() **259**

read\_clock() **365**

read\_event() **197**

read\_evt **198**

read\_user\_event () **199**

ready-to-receive packets **524**

real-time clock **362**

real-time clock functions. See *functions, real-time clock.*

receive choke **537**

receive choke condition **537**

receive choke trigger level **537**

receive FIFO **537**

receive flow control

    See also *flow control.* **537**

rename files **106**

reset\_file\_attributes() **111**

reset\_port\_error() **581**

Returns **244**

RS-232

    signals **526**

RS-232 port functions. See *functions.*

RS-232C compliance **515**

rsa\_calc() **598**

RTS **536, 584**

run() **167**

run\_thread() **168**

## S

screen\_size() **261**

SDLC initialization **524**

SDLC protocol **524**

    IBM packet formats **524**

    Modulo 8 information packets **524**

    unnumbered information packet formats **525**

secret key **720**

secs2time() **364**

security **720**

    security libraries **587**

        PIN entry functions

            CancelPIN **606**

            GetPINResponse **607**

            RequestPINEntry **609**

            SelectPINAlgo **610**

            SetPINParameter **611**

        VeriShield security scripts functions

            ExecuteScript **603**

            GetScriptStatus **601**

    security scripts

        key management **597**

            DUKPT **597**

    seed key **688**

    seek **82**

    seek pointer **83, 84, 89, 90, 95**

    seek\_cvlr() **82**

    seek\_vlr() **82**

    Selects **264**

    sem\_init() **174**

    sem\_post() **176**

    sem\_wait() **175**

    Semaphores **170**

    serial number **305**

    serial port functions. See *functions, RS-232 port.*

    session key **682**

    session keys **491, 686**

    set\_backlight() **266**

    set\_bcm() **212**

    set\_com1\_pwr() **756**

    set\_cursor() **267**

    set\_fc\_config() **536**

    set\_file\_attributes() **112**

    set\_file\_max() **77, 113**

    set\_font() **268**

    set\_group() **169**

    set\_hot\_key() **269**

    set\_owner() **336**

    set\_serial\_lines() **584**

    set\_timer () **200**

    setinverse() **264**

    Setting **212**

    signature file **720**

    sloshing **76, 113**

    smart card functions. See *functions, smart card*

    smart card reader

        ICC socket locations **362**

- smart card support **487**  
sound **382**  
sponsoring application **150**  
**SRAM 85**  
string conversion functions. See *functions, string conversions.*  
string conversions **696**  
**strnlwr() 700**  
**strnupr() 701**  
supervisory packet format **524**  
**SVC\_2INT() 709**  
**SVC\_AZ2CS 707**  
**SVC\_CHECKFILE() 148**  
**SVC\_CHK\_FIFO() 312**  
**SVC\_CHK\_PASSWORD() 282**  
**SVC\_CLR\_FIFO 313**  
**SVC\_CRC\_CALC() 320**  
**SVC\_CRC\_CALC\_L() 321**  
**SVC\_CRC\_CCITT\_L() 322**  
**SVC\_CRC\_CCITT\_M() 323**  
**SVC\_CRC\_CRC16\_L() 324**  
**SVC\_CRC\_CRC16\_M() 325**  
**SVC\_CRC\_CRC32\_L() 326**  
**SVC\_CS2AZ() 710**  
**SVC\_DSP\_2\_HEX() 703**  
**SVC\_FLASH\_SIZE 283**  
**SVC\_FLASH\_SIZE() 139**  
**SVC\_GET\_FIFO() 314**  
**SVC\_HEX\_2\_DSP() 702**  
**SVC\_INFO\_COUNTRY() 284**  
**SVC\_INFO\_CRASH() 285**  
**SVC\_INFO\_DISPLAY 270, 286**  
**SVC\_INFO\_DISPLAY() 270**  
**SVC\_INFO\_EPROM() 287**  
**SVC\_INFO\_HW\_VERS() 288**  
**SVC\_INFO\_IR() 300**  
**SVC\_INFO\_KBD() 271**  
**SVC\_INFO\_LIFETIME() 290**  
**SVC\_INFO\_LOTNO 291**  
**SVC\_INFO\_MFG\_BLK() 293**  
**SVC\_INFO\_MOD\_ID() 294**  
**SVC\_INFO\_MODELNO() 297**  
**SVC\_INFO\_OS\_HASH() 310**  
**SVC\_INFO\_PARTNO 298**  
**SVC\_INFO\_PIN\_PAD() 299, 556**  
**SVC\_INFO\_PORT\_MODEM() 301**  
**SVC\_INFO\_PRNTR() 302**  
**SVC\_INFO\_PTID() 303**  
**SVC\_INFO\_RESET() 304**  
**SVC\_INFO\_SERLNO() 305**  
**SVC\_INT2() 708**  
**SVC\_LRC\_CALC() 327**  
**SVC\_MEMSUM() 328**  
**SVC\_MOD\_CK() 329**  
**SVC\_PACK4() 704**  
**SVC\_PUT\_FIFO() 315**  
**SVC\_RAM\_SIZE() 308**  
**SVC\_READ\_FIFO() 316**  
**SVC\_RESET 73**  
**SVC\_RESTART 73, 189**  
**SVC\_UNPK4() 705**  
**SVC\_VALID\_DATE 364**  
**SVC\_VALID\_DATE() 364**  
**SVC\_VERSION\_INFO() 309**  
**SVC\_WAIT() 202, 373**  
**SVC\_WRITE\_FIFO() 317**  
**SVC\_ZONTALK() 531**  
system  
    date setting **368**  
    day setting **368**  
    library **332**  
    receive FIFO **537**  
    time setting **368**  
    time-out **584**  
    timing **363**  
system devices **331**  
    access **331**  
system information functions. See *functions, system information.*  
system mode **64, 577**  
    downloads from **713**  
    entry **226, 713**  
    local and remote operations **577**

**T**

- task ID **695**  
tasks **159, 162**  
    device ownership **160**  
    end a task **160**  
    hot key **161**  
    locking files during use **161**  
    one starts another **189**  
terminal  
    ASCII table for display **803**  
    identification number **303**  
    last reset date **304**  
    life span determination **290**  
    password **578**  
    serial number **305**

verify status **578**  
terminal management **228**  
terminal reset **73**  
terminal security **720**  
test\_master\_key() **492, 687**  
Threads **170**  
time setting **368**  
time2secs() **364**  
timers, cancelling **194**  
timestamp **99**  
Toggles **252**  
transmit FIFO depth **536**  
traps **57**  
triple-track decode sequence **343**  
triple-track mag stripes **343**  
TXO functions  
  getscrollmode **245**  
  insline **251**  
  kbd\_pending\_count() **253**  
  kbd\_pending\_test() **254**  
  putpixelcol() **257**  
  resetdisplay() **260**  
  SVC\_INFO\_KBD() **271**  
  wherecur() **272**  
  wherewin() **273**  
  wherewincur() **274**  
  window() **275**

**U**  
ultoa() **699**  
unnumbered information packet formats **525**  
USB **771**  
  device functions  
    get\_usb\_device\_bits() **562**  
  External Dial (COM3) **521, 780**  
  External Ethernet (ETH1) **563, 771**  
  External Serial (COM6) **557, 780**  
  Flash Drive **771**  
  Internal WiFi (WLN1) **563, 781**  
utilities  
  strings **696**

**V**  
variable-length record (VLR) **62, 74, 81, 87, 94**  
variables  
  \*GO **72**  
  global **332**  
VeriCentre **155, 725**  
VeriShield **145, 154, 720**  
VeriShield security script functions **600**

version, OS **309**  
VLR. See *variable-length record*.  
void **39**

**W**  
wait\_event () **203**  
wait\_evt() **204**  
wherecur() **272**  
wherewin() **273**  
wherewincur() **274**  
window() **275**  
write() **276**  
write\_at() **277**

**Z**  
ZonTalk  
  CONFIG.SYS \*Z series variables **145, 155**



VeriFone, Inc.  
2099 Gateway Place, Suite 600  
San Jose, CA, 95110 USA.  
1-800-VeriFone  
[www.verifone.com](http://www.verifone.com)

# Verix® V Operating System

## *Programmers Manual*

