

Verix eVo ACT

Programmers Guide



Verix eVo ACT Programmers Guide
© 2010 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, Verix eVo terminals are registered trademarks of VeriFone. Other brand names or trademarks associated with VeriFone products and services are trademarks of VeriFone, Inc.

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

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

VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA
(800) VeriFone (837-4366)

www.verifone.com

VeriFone Part Number DOC00310, Revision A



CONTENTS

CHAPTER 1 Introduction	Audience	11
	Organization	11
	References	12
	Manual Conventions	13
	Conventions	13
	Syntax Notation	13
	Acronyms	14
	Code Examples	15
 CHAPTER 2 Getting Started with Verix eVo ACT	Shared Libraries	17
	Building Applications Referencing a Shared Library	17
	Library Resides in Flash	18
 CHAPTER 3 Programmers Quick Reference	Function Calls	19
 CHAPTER 4 Application Idle Engine	Integrating the AIE into an Application	33
	Programmer-Defined Functions	35
	Application-Defined Idle Loop Function	35
	Handling CANCEL Key	35
	Slow Poll function	35
	Fast Poll function	35
	Application Idle Tables	36
	Branch Table	36
	Function Table	39
	Event Detection	41
	System Events	41
	Periodic Events	41
	Application Idle Functions	42
 CHAPTER 5 Message/Data Entry Engine	Message File	43
	TXOFILE Utility	43
	File Type and Compression Type	46
	Message File Format	46
	Message/Data Entry Functions	47
 CHAPTER 6 ISO 8583 Message Interface Engine	ISO 8583 Engine Overview	50
	ISO 8583 Bitmaps	50
	Message Structure	50
	ISO 8583 Engine Design	51

Engine Components	51
Engine Files	52
Incorporating the Engine	52
Global Data and Definitions	53
Global Defines	53
Global Variables	54
Global Structures	54
Typical Host/Terminal Packet Structure	55
Application Design	56
Map Construction	56
Map Manipulation Routines	56
Field Table Construction.	57
Variant Fields	59
Computed Fields	61
Convert Table	62
Conversion Routines.	64
Packet Assembly and Disassembly	66
ISO 8583 Message Interface Functions	67

CHAPTER 7

ISO 8583 Protocol Engine

ISO 8583 Protocol Engine Data Structure	69
Programmer-Defined Functions	73
mdm_tr_8583	73
def_valid_8583	74
prot_8583()	75
check_8583_tpdu_msg_id()	76
process_8583_request()	77

CHAPTER 8

PIP Engine

Compiling Source Code Modules	79
Overview.	79
Programmer-Defined Functions/Data.	81
PIP Files	81
PIP Engine and Application Construction Toolkit	82
Modular Design	82
ISO 8583 Protocol.	82
ISO 8583 Message Engine.	82
The Field Table	83
The Convert Table	88
Variant Fields	88
Computed Fields.	89
ISO 8583 Protocol Engine	89
Protocol 8583 Data Structure	90
Programmer-Defined Setup Function	90
Integrating the PIP Engine Into Your Application	91
AMEX Validation Functions	99
The Validation Process.	99
PIP Functions	101
Troubleshooting PIP Application Modules	102
PIP/ISO 8583 Glossary	103

CHAPTER 9 Data Capture Engine	Keyed File System	105
	Application Data	106
	Data Capture Functions	106
	Example Program.	106
 CHAPTER 10 Modem Engine	Modem Engine Functions and Macros.	107
	Macros	107
	MDM_READ_BLK()	108
	MDM_RESET_ERROR()	108
	MDM_STATUS().	109
	MDM_WRITE_BLK()	110
	XMODEM.H Listings	111
	XMODEM.H Structures and Unions.	117
	Modem Functions.	118
	Example Program.	118
 CHAPTER 11 Report Formatter	Incorporating the Report Engine in an Application.	120
	Conditional Printing	121
	Report Conversion Utility	121
	Report Format Files	122
	Report Format File Guidelines	122
	Errors	126
	Example Format File	127
	Global Variables Declaration File.	127
	Data Structures	127
	Global #defines.	127
	Global Structures	127
	Sample Output	128
	Report Formatter Functions	128
	Example Program.	128
 CHAPTER 12 Database Library	Database Features.	129
	File Formats	129
	Database File Format	129
	Index File Format	130
	File Storage Structure	130
	File Compression	131
	Database Library Functions	131
 CHAPTER 13 Printer Drivers	Verix eVo Based Terminal ITPs	133
	Downloadable User Fonts	133
	Downloadable Printer Logo	134
	Printing Monochrome Bitmap.	134
	Printer Driver Functions	134
	Example Program.	135

CHAPTER 14

Function Calls

AC Library Function Categories	138
AC Library Function Calls	140
act_kbd_pending_test()	141
append_char()	141
atox()	142
beep()	142
bitfun()	143
card_parse()	144
chars2int()	148
chk_card_rdr()	149
chk_luhn()	150
CHK_TIMEOUT()	151
clock_data()	152
clock2int()	154
ctons1()	155
cvt_ym()	156
delete_char()	156
display()	157
display_at()	158
display_new()	159
do_compression()	160
dsp_strs()	162
ERR_BEEP()	162
f_dollar()	163
fieldcnt()	164
fieldfix()	165
fieldray()	166
fieldvar()	167
set_chars_per_key_value()	168
getkbd_entry()	169
getxy()	175
insert_char()	176
insert_decimal()	176
int2str()	177
julian_date()	178
KBHIT()	178
key_card_entry()	179
keyin_amt_range()	181
keyin_mapped()	183
LEAP_YR()	184
load_bmp()	185
long2str()	187
MAX()	188
MEMCLR()	189
msg_display_at()	190
msg_display_new()	191
MIN()	192
month_end()	193
mult_strcat()	193
NORM_BEEP()	194
ntocs()	194

p_set_baudformat()	195
pad()	196
pause()	197
prompt()	198
prompt_at()	199
purge_char()	200
range()	201
range_vlr()	203
scroll_buffer()	204
set_imeout()	205
sgetf()	206
SLEEP()	209
sputf()	210
strn2int()	212
str2digit()	213
str2dsp_len()	214
str2int()	215
str2long()	216
track_parse()	217
view_buffer()	218
DVLR Function Calls	219
delete_dvlr()	219
insert_dvlr()	220
read_dvlr()	220
seek_dvlr()	221
write_dvlr()	222
IVLR Function Calls	223
delete_ivlr()	223
insert_ivlr()	223
read_ivlr()	224
replace_ivlr()	224
seek_ivlr()	225
write_ivlr()	226
SVC Function Calls	227
SVC_CLOCK()	227
SVC_KEY_NUM()	228
SVC_KEY_TXT()	229
SVC_TICKS()	231
Application Idle Engine Function Calls	232
Application Idle Engine Examples	232
aie_main()	233
appl_idle_get_cancel_poll_time()	233
appl_idle_get_fast_poll_time()	234
appl_idle_get_idle_loop_time()	234
appl_idle_get_slow_poll_time()	234
appl_idle_set_cancel_loop_time()	234
appl_idle_set_fast_poll_time()	235
appl_idle_set_idle_poll_time()	235
appl_idle_set_slow_poll_time()	235
Message/Data Entry Engine Function Calls	236
msg_get()	236
msg_select_file()	237

ISO 8583 Message Interface Engine Function Calls	238
asc_to_asc()	239
asc_to_bcd()	239
asc_to_str()	239
av2_to_str()	240
av3_to_av3()	240
av3_to_str()	240
bcd_to_asc()	241
bcd_to_bcd()	241
bcd_to_snz()	241
bcd_to_str()	242
bi2_to_hst()	242
bi3_to_hst()	242
bin_to_hst()	243
bit_to_bit()	243
bv2_to_str()	243
get_dst_8583()	244
get_fn_8583()	244
get_src_8583()	244
hst_to_bi2()	245
hst_to_bi3()	245
hst_to_bin()	245
iso8583_main()	246
map_clear()	246
map_man()	247
map_reset()	247
map_set()	248
map_test()	248
process_8583()	249
set_dst_8583()	249
set_src_8583()	250
str_to_asc()	250
str_to_av2()	250
str_to_av3()	251
str_to_bcd()	251
str_to_bv2()	251
str_to_xbc()	252
xbc_to_str()	252
PIP Engine Function Calls	253
check_8583_tpdu_msg_id()	253
find_field_index()	254
pip_main()	254
pip_trans()	255
prot_8583()	258
prot8583_main()	260
save_8583_tpdu_msg_id()	260
set_tpdu_length()	261
Data Capture Engine Function Calls	262
dce_key_cvt()	263
DCE_GETCFG_C()	265
DCE_GETCFG_S()	265
DCE_PUTCFG_C()	266

DCE_PUTCFG_S()	266
Modem Engine Function Calls	267
inInitModem()	268
inOpenModem()	269
vdCheckDataMode()	270
vdSetEchoMode()	270
xhayes_control()	271
xhayes_display()	272
xhayes_flush()	274
xhayes_response()	275
xhayes_send_cmd()	276
xhayes_status()	277
xmdm_check_status()	278
xmdm_checkline()	279
xmdm_clear()	280
xmdm_close()	282
xmdm_dial_status()	283
xmdm_failed_output_pending()	284
xmdm_flush()	284
xmdm_get_line_dial()	285
xmdm_hangup()	287
xmdm_init()	288
xmdm_input_pending()	290
xmdm_open()	291
xmdm_output_pending()	293
xmdm_receive_data()	294
xmdm_send_data()	295
xmdm_set_attrib()	296
xmdm_set_protocol()	297
Report Formatter Function Calls	298
Report Formatter Examples	298
formater_close()	299
formater_line()	300
formater_main()	301
formater_open()	302
formater_set_flags()	303
formater_template()	303
Database Library Function Calls	304
Database Library Examples	304
db_close()	304
db_delete()	305
db_get_key()	306
db_open()	308
db_read()	309
db_remove()	310
db_seek_key()	311
db_seek_record()	312
db_write()	313
Printing Monochrome Bitmap	313
print_image()	313
Print Pak 3700 Function Calls	315
p3700_close()	315

p3700_dnld_char()	316
p3700_dnld_font_file()	317
p3700_dnld_graphic_file()	318
p3700_id()	319
p3700_init()	320
p3700_mode()	321
p3700_print()	321
p3700_print_graphic()	322
p3700_select_font()	322
p3700_sel_tbl_dnld_char()	323
p3700_status()	324

APPENDIX A Example Programs

EXFIELD.C	326
EXTMOUT.C	329
EXTRANS.C	331
Data Capture Engine Example Program	336
Modem Engine Example Program	338
Report Formatter Example Program	342
P3700 Example Program	345



Introduction

This manual provides information on the Verix eVo ACT Library, including modules supported within the library, function call descriptions, and example code and applications.

Audience

This manual is aimed at programmers using Verix eVo ACT to develop applications for Verix eVo based terminals. To use this manual you should have an understanding of the following:

- C programming concepts and terminology
- Windows 2000 or Windows XP or Windows Vista.

Organization

The organization of the manual is as given in [Table 1](#):

Table 1 Organization

Chapter	Description
Chapter 1, Introduction	Provides information on the Verix eVo ACT Library, including modules supported within the library.
Chapter 2, Getting Started with Verix eVo ACT	Lists the advantages of using Verix eVo ACT and describes the procedure to build an application using ACL as a shared library.
Chapter 3, Programmers Quick Reference	Lists the system function calls and its description for Verix eVo Operation System.
Chapter 4, Application Idle Engine	Describes the features of AIE, describes the process flow of Integrating the AIE into the application, and lists the different function calls, events and its usage.
Chapter 5, Message/Data Entry Engine	Lists the tasks that Message/Data Entry Engine is responsible, describes the message file, and TXOFIL Utility.
Chapter 6, ISO 8583 Message Interface Engine	Provides an overview on ISO 8583 Engine, ISO 8583 Interface Engine components, files, global data definition, variables, and structures.
Chapter 7, ISO 8583 Protocol Engine	Describes about the ISO 8583 Protocol Engine data structure and programmer defined functions.
Chapter 8, PIP Engine	Describes the PIP Engine, modular design that is adopted, ISO 8583 Message Engine, programmer-defined function calls, and the structure of the 8583 protocol used.
Chapter 9, Data Capture Engine	Explains about the Data Capture Engine, key files and application data.
Chapter 10, Modem Engine	Describes about the Modem Engine and macros used in that Modem Engine.

Table 1 Organization (continued)

Chapter	Description
Chapter 11, Report Formatter	Explains the report formatter and the steps to use it.
Chapter 12, Database Library	Lists the features of database library, file formats, and file storage structure.
Chapter 13, Printer Drivers	Describes information about the Verix eVo based Terminal ITPs, downloadable fonts, logos. Explains about different dot matrix printers that communicate with the host terminal through the RS-232 port.
Chapter 14, Function Calls	Describes the different function calls for various modules.
Appendix A, Example Programs	Provides example codes for different programming functions.

References

Refer to the following set of manuals for additional information on the Verix eVo OS or the Verix eVo TX OS:

- AMEX Express 3000 PIP Terminal Technical Specifications

In addition to VeriFone programming support manuals listed in the introduction section of this manual, refer to the following industry manuals for PIP application development:

- ISO 8583 Bank Card Originated Messages—Interchange Message Specifications—Content for Financial Transactions (part number ISO 8583: 1987(E))
- AMEX Express 3000 PIP Terminal Technical Specifications

Manual Conventions

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

Conventions

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

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

NOTE



Note points out interesting and useful information.

CAUTION



Caution points out potential programming problems.

The various conventions used throughout this manual are listed in [Table 2](#).

Table 2 Conventions

Abbreviation	Definition
bps	bits per second
KB	KiloByte
MIN	minimum (value)
ms	Millisecond (one-thousandth of a second)
sec	seconds

Syntax Notation

This library reference uses the following syntax notation:

```
char *fgets(char *strg, int n,
FILE *strm);
```

```
%[*][width][modifier]
```

```
(stream,format [,argument,...]);
```

```
S_IREAD | S_IWRITE
```

The `Courier` typeface is used for source code, DOS commands, and filenames.

Items in braces are optional, such as an optional argument to a function.

A series of three dots (either in a column or in a line) signifies that an item can be repeated any number of times.

A vertical bar separates optional choices.

variable

Variable names (identifiers) or information to be supplied by the programmer appears in italics.

0x7F

Hexadecimal values are denoted with a leading 0x.

All source code characters are case sensitive. For example, `fprint()` and `FPRINT()` may not be the same facilities. All source code punctuation must be included as shown.

Acronyms

The acronyms used in this manual are listed in [Table 3](#).

Table 3 Acronym Definitions

Acronym	Definitions
AC	Application Construction
ACL	Application Construction Library
ACT	Verix eVo ACT
AIE	Application Idle Engine
ADR	Address
AMEX	American Express
API	Application Programmers Interface
ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
ABA	American Bankers Association
BLN	Backward Length
C	C-Language Program
CAS	Credit Authorization System
CB	Control Byte
CLR	Clear
CCITT	Comite Consultatif International de Telegraphique et Telephonique
COM	Common Object Model
CR	Carriage Return
CRC	Cyclic Redundancy Check
CTS	Clear To Send
CVLR	Compressed Variable Length Records
DATA REC	Data Record
DCD	Data Carrier Detect
DCP	Distributed Credit Authorization System (CAS) Processor
DOS	Disk Operating System
DVLR	Double Variable Length Records
EOF	End Of File
ESC	Escape
FLEN	Forward Length
HDLC	High-Level Data Link Control

Table 3 **Acronym Definitions** (continued)

Acronym	Definitions
ID	Identification
IATA	International Air Transport Association
ISO	International Organization for Standardization
ITP	Internal thermal printer
IVLR	Integer Variable Length Records
MAC	Message Authentication Code
N/A	Not Applicable
NMS	Network Management System
OS	Operating System
PC	Personal Computer
PIP	Plural Interface Processing (also referred to as split-dial)
POS	Point-Of-Sale
PVN	Providian Financial Corporation
RAM	Random-Access Memory
REV_OFF	No Reversal Processing
REV_ON	Normal Reversal Processing
REV_ONLY	Process Only a Pending Reversal
RI	Ring Indicator
ROC	Record of Charge
RRN	Retrieval Reference Number
RX	Receive
SDK	Software Development Kit
SDLC	Synchronous Data Link Control
SDS	Software Development Suite
TNMS	Telecommunication Network Management System
TPDU	Transport Protocol Data Unit
TX	Transmit
TXO	Transaction eXpress Option
VFI	VeriFone
VLR	Variable Length Records
VPN	VeriFone Part Number

Code Examples

Links to *code example* text files included on the Verix eVo ACT CD are provided throughout this manual. The `.txt` file opens in your specified default text editor.

NOTE

Writing your own applications is recommended.

Links are referenced by the underlined word Example; click on the following section title to bring up a text file example:

Example

NOTE

So that the entire code example can be read on screen, ensure that the word-wrap option is enabled in your preferred text editor.



Getting Started with Verix eVo ACT

Verix eVo ACT is a collection of C code modules designed to reduce the effort required to design and develop applications for VeriFone Verix eVo based terminals.

Verix eVo ACT provides the following benefits:

- Reduces design, coding, and testing time. Commonly used functions and procedures are simply incorporated into your new and existing applications designs.
- Consistent methodology for common procedures. Using Verix eVo ACT ensures that applications complete the same tasks in the same way. This reduces the amount of effort to debug new code and eases the effort required for quality control.
- Easier maintenance. One of the time consuming tasks of code maintenance is the learning curve required when a new programmer must enhance or modify an application. Verix eVo ACT provides code that the programmer can already be familiar with and can concentrate on the unique aspects of the application.
- Optimization and error trapping. The Verix eVo ACT modules provide error trapping and have been compiled with full optimization. This helps to provide the most efficient code for the application programmer.
- Code leveraging. The application programmer can develop additional specialized functions using the Verix eVo ACT modules as a base. This helps control the size of the application by reusing code from the toolkit. The components of the toolkit use this principle as the various engines call AC Library (ACL) functions to complete their tasks.

Shared Libraries

This chapter describes using the ACL as a shared library.

In a multi-application environment, more than two applications can reside on a single Verix eVo based terminal. However, the memory space available on the terminal is limited. To efficiently utilize the memory on the terminal, use a common copy of the ACL for all applications. Verix eVo ACT allows the application to be *linked* to the library as a shared library.

Building Applications Referencing a Shared Library

This section explains the procedure for building an application using ACL as a shared library. Change the project make file (typically the SMK file) to build the terminal application for the shared version of ACL. For example, build the sample.c file with the shared version of ACL. The following is a sample .MAK file with the changes required for using ACL as a shared library:

Example

The linked example code file demonstrates use of MAK().

Library Resides in Flash

In the previous example, the sample application is built so that the shared ACL resides in the terminal flash memory. The following is an example download file, where `setdrive.f` designates the flash partition, `setgroup.15` is the flash memory partition, GID 15, and `setdrive.i` designates the RAM partition:

```
*go=sample.out
setgroup.15
-iact.lib.p7s
setdrive.f
-iact.lib
setdrive.i
setgroup.1
-isample.out
-isample.out.p7s
```

NOTE

As Verix eVo ACT library is dependent on EESL library, EESL library should be downloaded to the terminal along with Verix eVo ACT shared library.



Programmers Quick Reference

This section provides programmers quick access to system function calls, CONFIG.SYS variables, device variables, error codes, download procedures, instructions on erasing flash, and keypad key return values for the Verix eVo operating system.

Function Calls The functions listed in [Table 4](#) are arranged by device or purpose. Refer to the function description for associated parameters and 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 on to jump to the function description.

Table 4 Function Calls

Function Call	Description	Page
ISO 8583 Message Interface Engine		
mdm_tr_8583 Calls		
<pre>int mdm_tr_8583 (int h_device, int mode, unsigned char *req_buf, unsigned req_size, unsigned char *rcv_buf, unsigned rcv_limit, unsigned timeout, unsigned carrier_timeout)</pre>	Transmits and receives ISO 8583 packets.	73
def_valid_8583 Calls		
<pre>int def_valid_8583(int *parms, int com_result);</pre>	Validates the received packets.	74
<pre>int prot_8583 (COMM_8583_DATA *control_data, int rev_opt, unsigned resp_timeout);</pre>	Transmits and receives a transaction using the ISO 8583 protocol.	75
<pre>int check_8583_tpdu_msg_id (COMM_8583_DATA *control, BYTE orig_tpdu, BYTE *orig_msg_id);</pre>	Checks the current TPDU address.	76
<pre>int process_8583_request (COMM_8583_DATA *control_data, int mode, unsigned current_timeout, BYTE *request_buf, unsigned request_size, BYTE *tpdu_buf, BYTE *msg_id_buf);</pre>	Sends and receives an 8583 request.	77
Modem Engine		
Macros Function Calls		
<pre>int MDM_READ_BLK (int h_modem, struct Opn_Blks mdm_blk);</pre>	Reads opened modem device communication parameters into a Modem Opn_Blks structure.	108
<pre>int MDM_RESET_ERROR(h_modem);</pre>	Resets the latched error conditions of parity, framing, and data overrun.	108
<pre>int MDM_STATUS(int h_modem);</pre>	Obtains the opened modem device communication status and copies it to the global structure status_info.	109

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int MDM_WRITE_BLK (int h_modem, struct Opn_Blz mdm_blk);</code>	Initializes or resets modem device communication parameters.	110
AC Library		
AC Library Function Calls		
<code>int act_kbd_pending_test(int targetcode);</code>	Checks if the target character is present in the keyboard buffer.	141
<code>int append_char(char *string, char c);</code>	Accepts a pointer to a null-terminated string specified by string, and appends a character specified by c.	141
<code>BYTE atox(BYTE character);</code>	Converts an ASCII hex character to a binary byte value.	142
<code>void beep(int type);</code>	Activates the beep feature of the terminal.	142
<code>int bitfun(UINT *map, BYTE bit, BYTE set);</code>	Performs bit manipulation on an unsigned integer.	143
<code>int card_parse (char *track, struct TRACK *parsed, char *order);</code>	Accepts data from the card reader and parses it to the selected track data.	144
<code>int chars2int(char *s_buffer, int i_num);</code>	Converts a string of up to five (ASCII) numeric characters to its integer equivalent.	148
<code>int chk_card_rdr(int h_card);</code>	Checks if data is available from the card reader port.	149
<code>int chk_luhn(char *buffer);</code>	Accepts a null-terminated string containing digits, and verifies that the last digit in the string is the correct MOD 10 check digit for the preceding digits in the string.	150
<code>int CHK_TIMEOUT (int h_clock, unsigned long time_val);</code>	Uses the <code>ioctl()</code> function to compare a value specified by <code>time_val</code> to the current clock tick value.	151
<code>int clock_data (int clock, int i_type, char *clock_buffer, char *s_dest_buf);</code>	Provides access to the terminal system clock.	152
<code>int clock2int (int h_clock, unsigned char *buffer, int *year, int *month, int *day, int *hour, int *min, int *sec, int *wday);</code>	Converts a date time string to the equivalent integer values.	154
<code>char *ctons(unsigned char *string);</code>	Converts a counted string to standard C-type null-terminated string.	155
<code>int cvt_ym(char *ym);</code>	Computes the total number of months from the year 0000 A.D. to the time given in a buffer in <code>yyyymm</code> format.	156
<code>int delete_char(char *string, int del_pos);</code>	Deletes a single character from a null-terminated string.	156
<code>int display(char *display_string);</code>	Accepts a pointer to a null-terminated string and writes it to the display without repositioning the cursor prior to the write, or clearing any portion of the display.	157

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int display_at</code> (unsigned int column, unsigned int line, char *display_string, unsigned int clear);	Repositions the cursor by column and line number before to displaying a string and optionally, clears the display.	158
<code>int display_new(char *message);</code>	Displays the message in column one of the current line, then clears the display to the end of line.	159
<code>int do_compression</code> (int mode, int compress_type, char *in_buf, char *out_buf, int in_len);	Compresses or uncompresses a record using one of the following compression types: no compression, 6BIT, VFI or BCD.	160
<code>int dsp_strs(char * va_alist, ...);</code>	Accepts one or more null-terminated strings and displays the string on the STDOUT device.	162
<code>void ERR_BEEP();</code>	Activates the beep feature of the terminal. The function produces an error beep that is lower in pitch than a normal beep.	162
<code>void f_dollar</code> (char *data_ptr, int prec, int dol_flag, int dol_format);	Formats a null-terminated string as a dollar amount.	163
<code>int fieldcnt</code> (char *buf, int start, int count, char *dest);	Copies the <i>n</i> th counted field from a source buffer specified by <i>buf</i> . <code>fieldcnt()</code> starts the first counted field at the position specified by <i>start</i> , and moves down to the field specified by <i>count</i> .	164
<code>int fieldfix</code> (char *buf, int start, int size, char *dest);	Copies a data string from a source buffer to a destination buffer starting at an offset specified by <i>start</i> up to a fixed length specified by <i>len</i> , or to end of the source buffer.	165
<code>int fielfray</code> (char *buf, int start, char stop, char *dest);	Copies a data string from a source buffer to a destination buffer.	166
<code>int fieldvar</code> (char *buf, int fldnum, unsigned char sep, char *dest);	Copies <i>n</i> th variable data field separated by a field delimiter.	167
<code>int getkbd_entry</code> (int h_clock, char *msg, char *outbuf, unsigned wait, unsigned type, char szKeyMap[][CHAR_PER_KEY], int KeyMapSize, int va_alist, ...);	Provides display and keyboard entry functions for a variety of input applications.	169
<code>int getxy(unsigned *x, unsigned *y);</code>	Return Values the <i>x</i> and <i>y</i> coordinates of the cursor relative to the current window.	175
<code>int insert_char(char *string, int pos, char c);</code>	Inserts a character into a string at a specified position.	176
<code>int insert_decimal(char *buf);</code>	Inserts a decimal point at the third character from the right in a dollar amount string.	176
<code>void int2str(char *dest, int val);</code>	Converts an integer value to an ASCII null-terminated string.	177
<code>int julian_date</code> (unsigned year, unsigned month, unsigned day);	Converts a specified date to a Julian calendar value.	178
<code>int KBHIT(void);</code>	Uses <code>kbd_pending_count()</code> to determine the number of unprocessed key entries in the keyboard buffer.	178

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int key_card_entry (int h_clock, int h_card, char *data, unsigned int type, unsigned int wait, unsigned int max, unsigned int min, char *buffer, unsigned int message_num, char szKeyMap[][CHAR_PER_KEY], int inKeyMapSize);</code>	Accepts the card-type data from either the keyboard or card reader.	179
<code>int keyin_amt_range (char *out_buf, int amt_fmt, long max, long min, int frac);</code>	Uses the SVC_KEY_NUM() routine to accept numeric entries suitable for use in monetary formatted amounts.	181
<code>char keyin_mapped(unsigned long key_map);</code>	Accepts a bitmapped long integer indicating valid keys for data entry.	183
<code>int LEAP_YR(int year);</code>	Determines if the year specified is a leap year.	184
<code>int load_bmp (int h_file, UINT tran_code, BYTE *b_map, BYTE *msg_id, BYTE *p_code);</code>	Assists in building the ISO 8583 message packets (required in PIP applications) by loading a bitmap, message ID and processing code from a formatted file.	185
<code>void long2str(char *dest, long val);</code>	Converts a signed long integer to a string.	187
<code>MAX(val1, val2);</code>	Returns the greater of two values.	188
<code>char *MEMCLR(char *buf, unsigned size);</code>	Clears a block of memory using the standard C <code>memset()</code> function.	189
<code>int msg_display_at (unsigned int column, unsigned int line, unsigned int message_num, char *buffer, unsigned int clr);</code>	Retrieves specified message from the message file.	190
<code>int msg_display_new (unsigned int message_num, char *buffer);</code>	Retrieves the specified message from the message file, repositions the cursor by column and line number prior to displaying the string, and optionally clears the display.	191
<code>MIN(val1, val2);</code>	Returns the lesser of two values.	192
<code>int month_end(int month, int year);</code>	Returns the number of days in the month for the specified year.	193
<code>int mult_strcat(BYTE *outbuf, char va_alist, ...);</code>	Concatenates multiple strings of text in the destination buffer.	193
<code>void NORM_BEEP(void);</code>	Activates the beep feature of the system. This produces a beep higher in tone than an error beep.	194
<code>int ntocs(char *dest_buf, char *src_buf);</code>	Converts standard C-type null-terminated strings to counted strings commonly used in C.	194
<code>int p_set_baudformat (int h_comm_port, int baud_rate, char *data_format);</code>	Initializes the specified communications port for printing with specified baud rate and data format.	195
<code>int pad (char *pdest_buf, char *psrc_buf, char pad_char, int pad_size, int align);</code>	Accepts a null-terminated string (source) and adds characters as needed to produce a null-terminated destination string of the length specified by the call.	196
<code>int pause(unsigned time);</code>	Waits a designated number of 10-ms intervals.	197

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int prompt</code> (<code>int h_clk</code> , <code>char *display_string</code> , <code>unsigned int wait</code> , <code>unsigned int opt</code>);	Displays a null-terminated string at the current line, column 1, for a specified time duration or until a key is pressed, whichever occurs first.	198
<code>int prompt_at</code> (<code>int h_clock</code> , <code>unsigned col</code> , <code>unsigned lin</code> , <code>char *msg</code> , <code>unsigned wait</code> , <code>unsigned opt</code>);	Displays a null-terminated string at a specified column and line for a specified time duration or until a key is pressed, whichever occurs first.	199
<code>int purge_char(char *buffer, char rem_char);</code>	Removes all occurrences of a specified character from a null-terminated string.	200
<code>int range(RANGE_PARMS *range_data);</code>	Searches a table stored in a keyed file.	201
<code>int range_vlr</code> (<code>char *acct</code> , <code>int start</code> , <code>char *data</code> , <code>char *file_name</code>);	Searches a VLR file for a record that bounds the given account number.	203
<code>int scroll_buffer</code> (<code>char *buf</code> , <code>int inc</code> , <code>unsigned long valid_keys</code> , <code>unsigned scroll_offset</code> , <code>char scroll_left</code> , <code>char scroll_right</code> , <code>char exit_key</code> , <code>char *key_buf</code>);	Displays a message in the current display window allowing the user to scroll through the message using application-selected keys.	204
<code>unsigned long set_ftimeout</code> (<code>int h_clock</code> , <code>unsigned int time</code> , <code>unsigned long gradient</code>);	Sets an interval timer based on the 1/64-second system tick clock.	205
<code>char sgetf(char *ss, char *cs, char *args, ...);</code>	Compares characters of an input string to a control string.	206
<code>void SLEEP(unsigned int time);</code>	Causes program execution to be suspended by the specified number of 10-ms increments.	209
<code>char sputf(char *result_store, char *next_cs,</code> <code>char *args, ...);</code>	Modified version of the standard C routine <code>sprintf()</code> .	210
<code>int strn2int(char *buffer, int cnt);</code>	Converts <code>cnt</code> bytes of buffer to an integer value.	212
<code>int str2digit(char *source);</code>	Accepts a string and purges all characters that are not digits.	213
<code>int str2dsp_len</code> (<code>char *source</code> , <code>unsigned offset</code> , <code>short dsp_wid</code> , <code>char dir</code>);	Calculates the number of characters required to fill a specified number of display positions.	214
<code>int str2int(char *buffer);</code>	Accepts a null-terminated string and returns an equivalent integer value.	215
<code>long str2long(char *string);</code>	Converts string data containing ASCII decimal digits (0 to 9) into a long integer number.	216
<code>int track_parse</code> (<code>struct TRACK *parsed</code> , <code>unsigned char tk_option</code>);	Parses track data.	217
<code>int view_buffer</code> (<code>char *buf</code> , <code>int inc</code> , <code>unsigned long key_map</code>);	Displays a string in the current display window and allows the user to use the [*] and [#] keys to scroll to the right and left, respectively, when viewing strings that are larger than the current display.	218
DVLR Function Calls		
<code>int delete_dvlr(int handle, int count);</code>	Deletes one or more double variable length records from a DVLR file.	219
<code>int insert_dvlr</code> (<code>int handle</code> , <code>const char *buffer</code> , <code>int size</code>);	Inserts a double variable length record into a DVLR file.	220

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int read_dvlr(int handle, char *buffer, int size);</code>	Reads a double variable length record from a DVLR file.	220
<code>long seek_dvlr (int handle, long offset, int origin);</code>	Seeks the specified number of records forward or backward in a DVLR file.	221
<code>int write_dvlr (int handle, const char *buffer, int size);</code>	Writes a double variable length record to a DVLR file.	222
IVLR Function Calls		
<code>int delete_ivlr(int h_file, unsigned int count);</code>	Deletes one or more integer counted records from an IVLR file.	223
<code>int insert_ivlr (int h_file, const char *buffer, int rec_size);</code>	Inserts an integer counted record into an IVLR file.	223
<code>int read_ivlr(int h_file, char *data, int size);</code>	Reads an integer counted record from an IVLR file.	224
<code>int replace_ivlr (int h_file, char *buffer, int rec_size);</code>	Replaces a record at the current position in an IVLR file.	224
<code>long seek_ivlr (int h_file, long rec_num, int origin);</code>	Moves the file pointer to the specified record in an IVLR file.	225
<code>int write_ivlr (int h_file, const char *data, int size);</code>	Writes an integer counted record to an IVLR file.	226
SVC Function Calls		
<code>int SVC_CLOCK(int action, char buffer, int limit);</code>	Allows the user to read or set the current time.	227
<code>int SVC_KEY_NUM (char *dest_buff, int max_digits, int fraction, int punctuate);</code>	Uses keyboard, display, and beeper input to retrieve a formatted decimal number (counted string).	228
<code>short SVC_KEY_TXT (char *dest, short type, short max, short min, char szKeymap[][CHAR_PER_KEY], int KeyMapSize);</code>	Uses keyboard, display, and beeper to retrieve formatted data input.	229
<code>int SVC_TICKS(int action, long *longadr);</code>	Allows the user to check if a tick value has expired or read the systems tick counter.	231
Application Idle Engine Function Calls		
<code>int aie_main (BRANCH_TBL_ENTRY *idletable, PF_TABLE *appltbl, AIEPROC idle loop, AIEPROC fastpoll, AIEPROC slowpoll, AIEPROC activate, AIEPROC deactivate);</code>	The <code>main()</code> of the program is defined by the application.	233
<code>long appl_idle_get_cancel_poll_time();</code>	Returns the value of the cancel detect timer.	233
<code>long appl_idle_get_fast_poll_time();</code>	Returns the value of the fast poll timer.	234
<code>long appl_idle_get_idle_poll_time();</code>	Returns the value of the idle loop timer.	234
<code>long appl_idle_get_slow_poll_time();</code>	Returns the value of the slow poll timer.	234
<code>void appl_idle_set_cancel_loop_time(long time);</code>	Sets the value of the cancel poll timer.	234
<code>void appl_idle_set_fast_poll_time(long time);</code>	Sets the value of the fast poll timer.	235
<code>void appl_idle_set_idle_poll_time(long time);</code>	Sets the idle poll timer.	235
<code>void appl_idle_set_slow_poll_time(long time);</code>	Sets the value of the slow poll timer.	235
Message/Data Entry Engine Function Calls		
<code>char *msg_get(unsigned message_num, char *buf_ptr);</code>	Retrieves a record from the message file.	236

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int msg_select_file(char *file_name);</code>	Opens a message file, reads the first record that contains the initialization information, and sets the local file handle and compression variables.	237
ISO 8583 Message Interface Engine Function Calls		
<code>void asc_to_asc(int n);</code>	Copies <i>n</i> bytes from source to destination, then advances the global pointers <code>dst_8583</code> and <code>src_8583</code> by <i>n</i> .	239
<code>void asc_to_bcd(int n);</code>	Converts <i>n</i> ASCII digits to packed BCD and stores them in the destination.	239
<code>void asc_to_str(int n);</code>	Moves <i>n</i> characters from source to destination then appends a null into destination.	239
<code>void av2_to_str(int c);</code>	Converts a 1-byte counted string into a null-terminated ASCII string and stores the result in the destination.	240
<code>void av3_to_av3(int c);</code>	Copies a 2-byte counted string from source to destination.	240
<code>void av3_to_str(int c);</code>	Converts a 2-byte counted ASCII string to a null-terminated ASCII string.	240
<code>void bcd_to_asc(int n);</code>	Expands a packed BCD sequence into its ASCII equivalent.	241
<code>void bcd_to_bcd(int n);</code>	Moves <i>n</i> BCD nibbles from source to destination.	241
<code>void bcd_to_snz(int n);</code>	Copies a fixed length BCD field to a string.	241
<code>void bcd_to_str(int n);</code>	Copies <i>n</i> BCD digits from the source to the destination and then appends a null character in the destination.	242
<code>void bi2_to_hst(int c);</code>	Converts a 1-byte counted string to a null-terminated ASCII hex string.	242
<code>void bi3_to_hst(int c);</code>	Converts a 2-byte counted string to a null-terminated ASCII hex string.	242
<code>void bin_to_hst(int n);</code>	Converts <i>n</i> bytes of binary digits into <i>2n</i> bytes of ASCII hex digits.	243
<code>void bit_to_bit(int n);</code>	Copies <i>n</i> bits from the source to the destination. Always moves in whole bytes.	243
<code>void bv2_to_str(int c);</code>	Converts a 1-byte counted BCD string to a null-terminated ASCII string.	243
<code>void get_dst_8583(unsigned char *dst);</code>	Gets the destination buffer of the ISO8583 engine.	244
<code>int get_fn_8583(void);</code>	Returns the erroneous field number during packet packing or unpacking.	244
<code>void get_src_8583(unsigned char *src);</code>	Gets the source buffer of the ISO8583 engine.	244
<code>void hst_to_bi2(int c);</code>	Converts a null-terminated ASCII hex string to a 1-byte counted string and stores the result.	245

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>void hst_to_bi3(int c);</code>	Converts a null-terminated ASCII hex string to a 2-byte counted string and stores them in the destination.	245
<code>void hst_to_bin(int n);</code>	Converts <i>n</i> bytes of ASCII hex digits to binary and stores the result.	245
<code>void iso8583_main (converters *pConvertTable, fn Ret1, fn Ret2, unsigned char *szdst_8583, unsigned char *szsrc_8583);</code>	Passes pointers to a convert table, function pointers to user-defined functions that return the value of variant fields, and two buffers to hold the values of the source and the destination (when conversion routines are used).	246
<code>void map_clear(unsigned char *map, int max_fn);</code>	Resets all the bits in the designated map.	246
<code>void map_man (unsigned char *va_alist, ...);</code>	Turns bits on or off in the specified map.	247
<code>void map_reset(unsigned char *map, int field_no);</code>	Turns off the bit associated with the field number.	247
<code>void map_set(unsigned char *map, int field_no);</code>	Turns on the bit associated with the field number.	248
<code>int map_test(unsigned char *map, int field_no);</code>	Returns the status of the bit in the bitmap corresponding to the field number.	248
<code>int process_8583 (int how, field struct *field_tbl, unsigned char *map, unsigned char *buffer, int limit);</code>	Processes packing or unpacking of a packet by checking the bits in the map.	249
<code>void set_dst_8583(unsigned char *dest);</code>	Sets the destination buffer of the ISO8583 engine.	249
<code>void set_src_8583(unsigned char *src);</code>	Sets the source buffer of the ISO8583 engine.	250
<code>void str_to_asc(int n);</code>	Converts a null-terminated ASCII string to fixed-size ASCII string with blank padding on the right, if necessary.	250
<code>void str_to_av2(int c);</code>	Converts a null-terminated ASCII string to a 1-byte counted string.	250
<code>void str_to_av3(int c);</code>	Converts a null-terminated ASCII string to a 2-byte counted string.	251
<code>void str_to_bcd(int n);</code>	Converts a null-terminated ASCII string to fixed-size BCD with zero padding on the left if necessary.	251
<code>void str_to_bv2(int c);</code>	Converts a null-terminated ASCII string (containing digits only) to a 1-byte counted BCD string.	251
<code>void str_to_xbc(int n);</code>	Converts a null-terminated ASCII string to a BCD string while preserving the first byte of the source.	252
<code>void xbc_to_str(int n);</code>	Converts a BCD string to a null-terminated ASCII string while preserving the first byte of the source.	252
PIP Engine Function Calls		
<code>int check_8583_tpdu_msg_id (COMM_8583_DATA *control, unsigned char *orig_tpdu, unsigned char *orig_msg_id);</code>	Verifies a valid match between the current values of the TPDU and message ID found in the field table, and the values passed in the call.	253

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>field_struct *find_field_index (int search_num, field_struct *ptr);</code>	Searches a field table array for a specific field entry.	254
<code>void pip_main (unsigned char *tmap, char *tmsgid, char *tproccode, Fxn func);</code>	Passes pointers to the map buffer, pointer to the message ID buffer, pointer to processing code, and a function pointer to user-defined <code>set_trans_field</code> function.	254
<code>int pip_trans (int trans_type, HOST_8583_DATA *host_struct);</code>	Provides advice message processing and an interface to the ISO 8583 Protocol Engine.	255
<code>int prot_8583 (COMM_8583_DATA *control_data, int rev_opt, unsigned resp_timeout);</code>	Interface to the ISO 8583 Message Engine, application-defined validation function, application-defined communications function, and processes reversals as specified in the ISO 8583 standard.	258
<code>int prot8583_main(RevFxn func);</code>	Passes a function pointer to a callback function used to modify the contents of reversal data if required by the specific host.	260
<code>void save_8583_tpdu_msg_id (COMM_8583_DATA *control, unsigned char *dest_tpdu, unsigned char *dest_msg_id);</code>	Copies the TPDU and message ID from the application variable to the destination buffer.	260
<code>int set_tpdu_length(int inTPDULength);</code>	Allows the user to set the length of the TPDU field for the PIP processing	261
Data Capture Engine Function Calls		
<code>int dce_key_cvt (unsigned char process, char *file, char *key, unsigned char cnvt, DCE_KEY_DATA data);</code>	Permits application data to be written to and read from keyed files, and completes any required data conversion using functions provided by the ACL.	263
<code>int DCE_GETCFG_C(char *key, DCE_KEY_DATA data);</code>	This macro call is the same as <code>dce_key_cvt()</code> , except that the filename is supplied as <code>CONFIG.SYS</code> , the process is supplied as <code>read</code> , and the data type is supplied as <code>character</code> .	265
<code>int DCE_GETCFG_S(char *key, DCE_KEY_DATA data);</code>	This macro call is the same as the <code>dce_key_cvt()</code> function, except that the filename is supplied as <code>CONFIG.SYS</code> , the process is supplied as <code>read</code> , and the data type is supplied as <code>string</code> .	265
<code>int DCE_PUTCFG_C(char *key, DCE_KEY_DATA data);</code>	This macro call is the same as <code>dce_key_cvt()</code> , except that the filename is supplied as <code>CONFIG.SYS</code> , the process is supplied as <code>write</code> , and the data type is supplied as <code>character</code> .	266
<code>int DCE_PUTCFG_S(char *key, DCE_KEY_DATA data);</code>	This macro call is the same as <code>dce_key_cvt()</code> , except that the filename is supplied as <code>CONFIG.SYS</code> , the process is supplied as <code>write</code> , and the data type is supplied as <code>string</code> .	266
Modem Engine Function Calls		
<code>int inInitModem (int h_modem, int i_max_wait, int va_alist, ...);</code>	Initializes the modem by receiving environment variable numbers as parameters.	268

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int inOpenModem (int *h_modem, char *port, int Baud, int Format);</code>	Opens the modem, as specified in <code>port</code> and obtains the resulting modem response.	269
<code>void vdCheckDataMode(int mode);</code>	Sets the parameter for checking the data mode before sending any command to the modem.	270
<code>void vdSetEchoMode(int setecho);</code>	Sets the parameter for echo mode ON or OFF in the command response.	270
<code>int xhayes_control (int h_modem, int h_clock, int max_wait, char command, char *buffer);</code>	Sends a Hayes command to the modem and returns a Hayes status code.	271
<code>int xhayes_display(int hayes_code, char *buffer);</code>	Translates a Hayes modem response code to its descriptive text equivalent, storing the string at the specified buffer address.	272
<code>int xhayes_flush(int h_modem);</code>	Clears the command response buffer from the modem device.	274
<code>int xhayes_response (int h_modem, int h_clock, int max_wait);</code>	Reads a Hayes modem command response and converts the command into a Hayes code integer value.	275
<code>int xhayes_send_cmd(int h_modem, char *cmd_buff);</code>	Converts the input string into Hayes format and sends the resulting command string to the modem.	276
<code>int xhayes_status (int h_modem, int h_clock, int wait_time);</code>	Obtains the Hayes response from the modem, and validates it, depending on the Hayes command executed, as specified by the command parameter.	277
<code>int xmdm_check_status (int h_modem, unsigned char stat_map);</code>	Returns the current modem signal information, including CTS, DCD, framing error, overrun error, parity error, and break or abort.	278
<code>int xmdm_checkline(int h_modem);</code>	Checks if a active telephone line is connected to the terminal.	279
<code>int xmdm_clear (int h_modem, char *sz_mdm_name, int h_clock, int wait_time, int rate, int format);</code>	Closes and re-opens the modem device.	280
<code>int xmdm_close (int h_modem, int output_pend, int input_pend);</code>	First obtains modem status, then, closes the modem device based on input parameters and modem communication status.	282
<code>int xmdm_dial_status (int h_modem, int h_clock, int max_wait);</code>	Obtains the results from a Hayes dialing command.	283
<code>int xmdm_failed_output_pending(int h_modem);</code>	Determines if there are any failed output messages pending.	284
<code>int xmdm_flush(int h_modem);</code>	Clears the modem data read() receive buffer.	284
<code>int xmdm_get_line_dial (int h_modem, char *dial_string, int *iwrite, int h_clock, int max_wait);</code>	Checks for the presence of a phone line.	285
<code>int xmdm_hangup (int h_modem, int h_clock, int max_wait);</code>	Instructs the modem unit to go on-hook (disconnect) from the phone line.	287

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>int xmdm_init</code> (<code>int h_modem</code> , <code>char *sz_mdm_name</code> , <code>int h_clock</code> , <code>int max_wait</code> , <code>int rate</code> , <code>int format</code>);	Initializes the modem by opening the device and setting the communications parameters to the specified protocol.	288
<code>int xmdm_input_pending</code> (<code>int h_modem</code>);	Determines if there are any input messages pending.	290
<code>int xmdm_open</code> (<code>int *h_modem</code> , <code>char *path</code> , <code>int h_clock</code> , <code>int max_wait</code> , <code>int rate</code> , <code>int format</code>);	Opens the modem device specified in <code>path</code> and obtains the resulting modem responses.	291
<code>int xmdm_output_pending</code> (<code>int h_modem</code>);	Determines if there are any output messages pending.	293
<code>int xmdm_receive_data</code> (<code>int h_modem</code> , <code>char *buffer</code> , <code>int min</code> , <code>int max</code> , <code>int max_wait</code>);	Waits for a specified period of time or until it receives any number of bytes greater than zero.	294
<code>int xmdm_send_data</code> (<code>int h_modem</code> , <code>char *buffer</code> , <code>int buff_len</code> , <code>int max_wait</code>);	Provides a timed transmission of data in <code>buffer</code> to the modem device.	295
<code>int xmdm_set_attrib</code> (<code>int h_modem</code> , <code>int rate</code> , <code>int format</code> , <code>int flush</code>);	Changes a single modem attribute, without affecting previously established modem attributes.	296
<code>int xmdm_set_protocol</code> (<code>int h_modem</code> , <code>int rate</code> , <code>int format</code>);	Initializes the communication data protocol attributes of the currently opened modem.	297
Report Formatter Function Calls		
<code>void formater_close</code> (<code>FORMATER *formdata</code>);	Closes the template file and returns to the caller.	299
<code>int formater_line</code> (<code>FORMATER *formdata</code> , <code>int start_line</code> , <code>int stop_line</code> , <code>int print_blank_lines</code> , <code>unsigned long condition</code> , <code>int *error_line</code>);	Formats a range of report lines. Formatting of conditional lines is based on the <code>condition</code> parameter.	300
<code>void formater_main</code> (<code>g_data *gvardata</code>);	Accepts a handle of <code>g_data</code> to be used by the application for formatting.	301
<code>int formater_open</code> (<code>int handle</code> , <code>FORMATER *formdata</code> , <code>char *template</code> , <code>int (*output_initializer)()</code> , <code>int time_out</code>);	Accepts the handle of an open output device, a <code>FORMATER</code> structure address that holds information about the current formatter job, the name of the report template file, a pointer to the output initialization function, and a time-out value that initializes the output device.	302
<code>unsigned long formater_set_flags</code> (<code>int va_alist</code> , ...);	Builds the condition flag from user application-specific variables.	303
<code>int formater_template</code> (<code>FORMATER *formdata</code> , <code>char *template</code>);	Specifies which template file to use.	303
Database Library Function Calls		
<code>short db_close</code> (<code>DB_FILE *db_ptr</code>);	Closes the database and index files.	304
<code>short db_delete</code> (<code>DB_FILE *db_ptr</code> , <code>long rec_num</code> , <code>long rec_cnt</code>);	Deletes one or more records from the database and index files.	305
<code>long db_get_key</code> (<code>DB_FILE *db_ptr</code> , <code>void *key_struct</code> , <code>void *Matchstr</code> , <code>long action</code> , <code>char *databuf</code> , <code>int dlen</code> , <code>long *rec_num</code>);	Finds the specified key value.	306

Table 4 **Function Calls** (continued)

Function Call	Description	Page
<code>short db_open (DB_FILE *db_ptr, char *filename, int key_len, int mode);</code>	Creates a database file and the corresponding index file and updates the database file structure.	308
<code>short db_read (DB_FILE *db_ptr, void *key_struct, char *data_buf, int dlen, long rec_num);</code>	Retrieves a specified record from the database file.	309
<code>short db_remove(char *filename);</code>	Deletes the database file and the corresponding index files.	310
<code>long db_seek_key (DB_FILE *db_ptr, void *key_struct, void *Matchstr, long action, long *rec_num);</code>	Searches the index file records for the specified key value.	311
<code>long db_seek_record (DB_FILE *db_ptr, long rec_num, int origin);</code>	Searches for a specified record in an index file and modifies the file pointer position accordingly.	312
<code>short db_write (DB_FILE *db_ptr, void *key_struct, char *data_buf, unsigned int data_len, long rec_num);</code>	Writes/updates a record in both the database and index files.	313
Print Pak 3700 Function Calls		
<code>int p3700_close(short h_comm_port);</code>	Waits for all pending data to transmit to the printer, then returns.	315
<code>short p3700_dnld_char (short h_comm_port, unsigned char *rd_buf, unsigned short write_bytes);</code>	Transmits a single character to the printer.	316
<code>short p3700_dnld_font_file (short handle, short h_font_file, short font_table);</code>	Downloads a font file set to the printer.	317
<code>short p3700_dnld_graphic_file (short h_comm_port, short h_graphic_file);</code>	Downloads a graphic logo file into the printer.	318
<code>int p3700_id(short h_comm_port, short id_time_out);</code>	Sends the <ESC>i command (request printer ID) to the ITP, and waits for a response.	319
<code>int p3700_init (short h_comm_port, short time_out);</code>	Sets the printer to native mode by sending the <GS><ESC>c<GS> command.	320
<code>unsigned char *p3700_mode (unsigned char mode, unsigned char* buf);</code>	Converts special print characters to a valid printer command sequence.	321
<code>int p3700_print (short h_comm_port, unsigned char *buf);</code>	Sends a text string to the ITP.	321
<code>short p3700_print_graphic (short h_comm_port, short imageId, short offset);</code>	Prints a graphic file already in printer memory.	322
<code>short p3700_select_font (short h_comm_port, short font_size, short font_table);</code>	Selects the font table to use for printing or downloading.	322
<code>short p3700_sel_tbl_dnld_char (short h_comm_port, unsigned char * font_buf, short font_table, short font_size, short font_bytes);</code>	Selects the font table, and then downloads a single font character to the printer.	323
<code>int p3700_status (short h_comm_port, short stat_time_out);</code>	Sends the <ESC>d command to the ITP and waits for it to respond with its status byte.	324
Printing Monochrome Bitmap		
<code>print_image(int offset, char * filename);</code>	Prints the monochrome bitmap.	313

Application Idle Engine

In a typical transaction environment, the terminal waits for an event (card swipe, keypress, or incoming dialup) to continue the application. This is known as the *idle state*. From the idle state, every application is required to execute tasks that are triggered by such events. Other tasks, such as opening devices or setting flags, are only required to execute during power up or on return from another code module.

The Application Idle Engine (AIE) assists the application programmer in managing the idle state; it acts as a traffic manager. When an event occurs at the idle state, such as a keypress or card swipe, the engine references an application-defined table to determine which function to execute. The format and functions of the Application Idle tables are described in this manual.

NOTE

- AIE will not support the dual keypress events.
 - V^X680 terminal does not have alpha and functional keys.
-

Integrating the AIE into an Application

AIE is designed to be used as a generic module. It provides most of the functionality required to process events from the idle state; however, it does not provide the functionality unique to each application. The engine cannot know if the application must monitor the system clock, perform an upload at a specified time, and what processing to perform each time the terminal powers up. To help meet the applications requirements, AIE assists in creating the following application routines:

- System initialization (power up)
- Cancel key processing
- Display the idle prompt (as required)
- Idle state processing

Figure 1 illustrates the AIE role in the application.

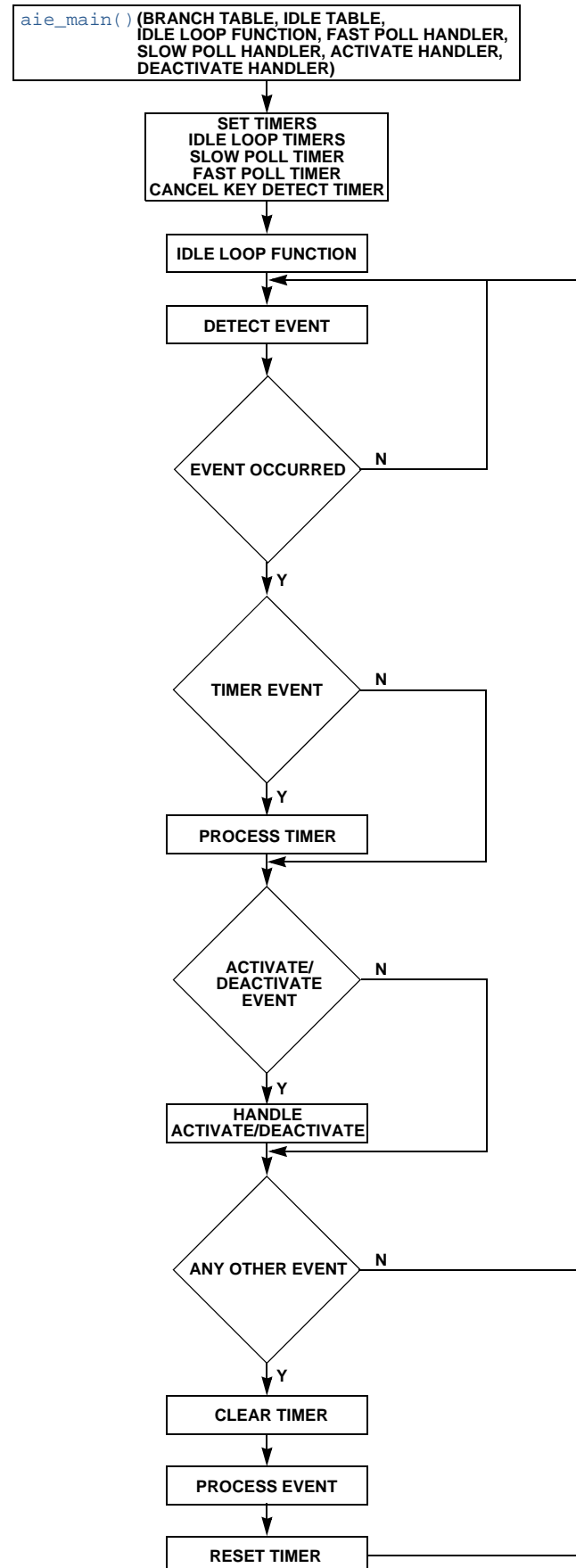


Figure 1

Application Processing Flow Chart

Programmer-Defined Functions

Figure 1 shows functions in the application idle process that must be written by the programmer. These functions allow the application to easily meet the specific requirements within the framework of the AIE.

Application Initialization

The application defines the `main()` of the program. The functionality defined by `appl_coldboot()` can be directly invoked by the application. Before calling `aie_main()`, the application should typically execute tasks performed during system initialization, such as:

- opening devices to be accessed by the application (card reader, modem, and so on)
- setting or testing flags (such as a download-needed flag)
- performing file maintenance
- moving CONFIG.SYS data to a file
- setting up user passwords. If the cancel key trap is armed in this function, it is overwritten (see Figure 1). In addition, this function must return TRUE (1). Any other return value causes AIE to exit.

Application-Defined Idle Loop Function

The Idle Loop function is passed as a parameter to `aie_main()` and is the main loop in the engine. It is called immediately after the timers are set, and then once per cycle. This function can be used for timed events, such as:

- print spooler management,
- automatic upload or download of data,
- idle display management,
- automatic settlement, and
- error processing.

Since this function is called once per cycle, the code must be very efficient. Executing complex or time-consuming operations in this function delays event processing of events.

Handling CANCEL Key

Handling the cancel key is the responsibility of the application. Use the `KEY_CANCEL` parameter of `act_kbd_pending_test()`, to determine if a cancel key is pending and the appropriate action is taken. In the idle state, AIE periodically checks (50 ms) for the cancel key and flushes the keyboard.

Slow Poll function

The Slow Poll function, defined by the application, is called by AIE once every 250 ms during the normal idle state. The programmer can create code to determine if programmer-defined events have occurred and handle them.

Fast Poll function

The Fast Poll function, defined by the application, is called by AIE once every 50 ms during the normal idle state. The programmer can create code to determine if programmer-defined events have occurred and handle them.

Application Idle Tables

Both the slow poll and fast poll functions can be used to perform any periodic activity.

One powerful feature of the AIE is its ability to use tables to control event processing. Two tables are required by the engine: the Branch Table and the Function Table. These must be passed as parameters to `aie_main()`, however, create additional tables to meet the requirements of the application. To understand the relationship between Branch Table and Function Table and how they are used by the engine, refer to [Figure 2](#).

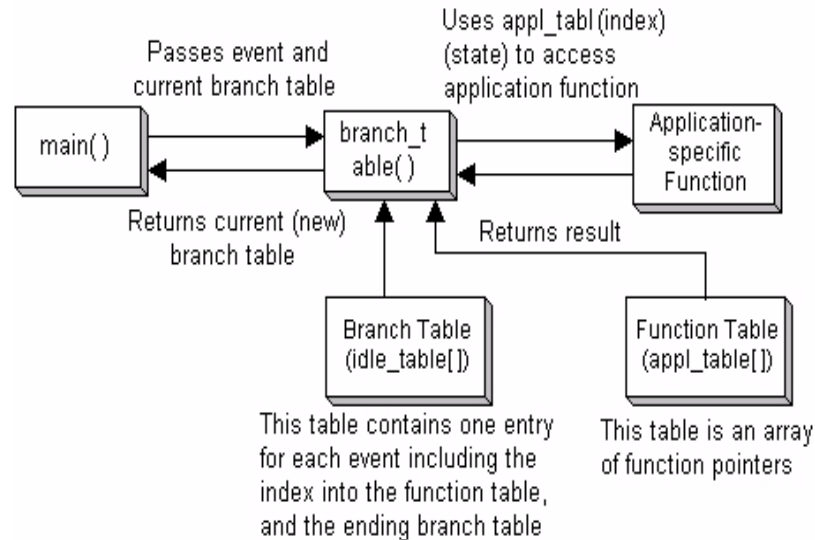


Figure 2 AIE Event Processing Table

Branch Table

The Branch Table contains an index to function pointers defined in the Function Table. Each time an event occurs in the idle state, the event processor accesses the Branch Table to determine which function to process. Each entry in the Branch Table has three members:

- The event to process (defined in `APPLIDL.H`)
- The index into the Function Table
- The current (new) Branch Table to access on return from the function

The Branch Table function increases flexibility and reduces code space by allowing creation of a table-driven application. To do this, create two tables that reference the events to process (refer to the `idle_table[]` and `appl_table[]` examples) and the `branch_table()` function does the rest. After executing the event, `branch_table()` returns a pointer to the next table used to process events.

NOTE

To switch to the new Branch Table, `BRANCH_EXIT` or any non negative value can be returned from the event handler functions. The following example shows that the `KEY_CR` event sets `func_table[]` as the new Branch Table. After executing the `KEY_CR` event handler function, `func_table[]` becomes the active table and waiting for events. `func_table[]` does not become the active Branch Table unless the function referenced by `FT_FUNC` returns `BRANCH_EXIT` or any non negative value. In this case, if any non negative value is returned from the handler function, the event is processed in the active table (`func_table[]`) and not in the idle table.

Example

The linked code example file demonstrates use of `branch_table()`.

NOTE

`BRANCH_TBL_ENTRY` is defined in `APPLIDL.H` (using typedef).

In addition to the programmer-defined table entries (for example, keypress or card swipe), the engine recognizes five other table entries:

- `END_TABLE` (required)
- `COMMON_FUNC` (optional)
- `ENTRY_FUNC` (optional)
- `ERROR_FUNC` (optional)
- `EXIT_FUNC` (optional)

Using any of the above four optional entries to perform common routines will reduce code space and increase application flexibility. If any of these entries are used in the application, the entry must be included in both the Branch Table and the Function Table, as well as write the function that executes when the entry is processed (see the `BRANCH_TBL_ENTRY` example above).

END_TABLE

This entry is required to truncate the table. It must be the last entry in the table. All entries occurring below `END_TABLE` are ignored. In addition, this entry can be used to process invalid application events, that is, events returned by the event handler functions (callback functions) and not listed in the active Branch Table. For example, in `idle_table[]` only `KEY1` - `KEY5` are mapped and event handler functions are provided. If any of the event handler functions return `KEY6`

which is not listed in the active table, the function associated with the END_TABLE entry is processed. The END_TABLE function can be used for error beep and display “INVALID KEY” etc. However it can still perform any other tasks required by the application. The END_TABLE function will not get executed for the system generated events which are not listed in the table. Refer to the [Example](#) for more details.

COMMON_FUNC

This entry executes prior to any other event for example, used to display messages or set transaction flags.

ENTRY_FUNC

This entry executes if Branch Table entry is called with the current event set to 0 (zero). After execution, ENTRY_FUNC must return the next valid event to process (refer to APPLIDL.H) for example, use ENTRY_FUNC to display a menu. When the user makes a selection from the menu, ENTRY_FUNC returns the keypress as the next event to process.

ERROR_FUNC

This entry executes if the value returned by an event is a negative value, which is then passed to ERROR_FUNC. The return values from ERROR_FUNC have the following significance:

- Positive integers (≥ 0): Process the return as the next event within the same table.
- Negative integers (< 0): Process the absolute value of the return as the next event, under the next (branch) table, as indicated by the current table entry.
- BRANCH_EXIT: Exits with no new events; forces the engine under control of the idle_table.

EXIT_FUNC

This entry executes prior to passing control to another table, or when BRANCH_EXIT is returned under the current table. For example, it can be used to hang up the phone, complete a print job, or perform other housekeeping tasks.

The previously described entries can be placed anywhere in the table (with the exception of END_TABLE); however, the entries are always processed in the following order:

- 1 COMMON_FUNC
- 2 ENTRY_FUNC
- 3 Standard event (for example, a keypress or card swipe)
- 4 ERROR_FUNC
- 5 EXIT_FUNC

Function Table

The Function table is an array of pointers to integer functions. Each element in the table is a name of an application-defined function written to process an event. Every entry in the Branch Table must have a corresponding entry in the Function Table.

Example

```
PF_TABLE appl_table[] =
{
    sale_tran,
    force_tran,
    credit_tran,
    void_tran,
    func_main,
    idle_common,
    idle_exit,
    idle_error,
    not_available,
    (PF_TABLE) END_TABLE
};
```

NOTE



To make referencing the function name to the corresponding table entry value easier, include the #define directive above the function name. The #define values must start at 0 (zero) and continue in ascending order.

Example

```
appl_table[] with #defines
PF_TABLE appl_table[] =
{
    #define FT_SALE      0
    sale_tran,

    #define FT_FORCE     1
    force_tran,

    #define FT_CREDIT    2
    credit_tran,

    #define FT_VOID      3
    void_tran,

    #define FT_FUNC      4
    func_main,

    #define FT_COMMON    5
    idle_common,

    #define FT_EXIT      6
    idle_exit,
```

```
#define FT_ERROR      7
idle_error,

#define NO_KEY        8
not_available,

(PF_TABLE) END_TABLE
};
```

These functions must be integer functions with no parameters. In addition, the functions must return one of the following values:

- A value < 0 indicates an error. In this case, the `ERROR_FUNC` entry executes (if included in `idle_table[]` and `appl_table[]`).
- `BRANCH_EXIT`: Continues processing (defined in `APPLIDL.H`).
- A valid event number to process (as defined in `APPLIDL.H`). Returning an event number causes the event to be processed as if it was executed from the idle state.

Event Detection

AIE detects events for all devices opened by the application.

System Events

System events are as follows:

APPL_CARD	Magnetic card reader event
C1_IN	COM1 event
C2_IN	COM2 event
BARIN	Barcode reader event
PIPE_IN	Pipe event
C3_IN	COM3 event
C4_IN	COM4 event
C5_IN	COM5 event
ICC1_IN	Smart card reader1 event
DEACTIVATE_IN	Deactivate event
ACTIVATE_IN	Activate event
SYSTEM_IN	System event
CONSOLE_IN	Console event
CLK_IN	Clock event
TIMER_IN	Timer event
USB_IN	USB event
C6_IN	COM6 event
WLN_IN	WLAN event
USER_IN	User event
SHUTDOWN_IN	Shutdown event
USB_CLIENT_IN	USB slave event
C8_IN	COM8 event
SOCKET_IN	Socket event

NOTE



To avoid wasting system resources, do not set traps at the idle state for these devices.

Periodic Events

AIE sets up four timers to handle:

- idle loop processing,
- cancel key detection,
- slow poll handling, and

- fast poll handling.

The programmer may define Slow Poll and Fast Poll functions that will be called periodically. These need to be passed as parameters to [aie_main\(\)](#). The Idle Loop function defined by the application is called periodically when the idle loop processing timer expires.

Similarly, in the idle state, on a Cancel key detect timer, the AIE checks for the Cancel Key and flushes the keyboard.

All of the following four timers are configurable and can be set using the calls specified in the [Application Idle Engine Function Calls](#). The default values are:

- Idle timer - 750 msec
- Cancel timer - 50 msec
- Slow Poll timer - 250 msec
- Fast Poll timer - 50 msec

Application Idle Functions

The application idle functions are described in the [Application Idle Engine Function Calls](#) section.



Message/Data Entry Engine

The Message/Data Entry Engine is a group of functions that control the display of prompts and messages, and reads data from the keypad or card reader. This library assists application development in the following tasks:

- retrieve messages,
- clear the display,
- display messages,
- set the cursor position,
- wait a specified time for a keypress, and
- accept input from the keypad or cardreader.

Other functions in this library support manual- or auto-scrolling through messages, which is useful for displaying long messages or menus.

Message File

The functions in the Message/Data Entry library are designed to retrieve prompts and messages from a message file created by program called TXOFILE. The format of the message file and instructions on how to create it are provided in this section.

While other types of files are discussed, the Message Engine is designed to use only message-type files that use any of the available compression types.

TXOFILE Utility

TXOFILE is a PC-based utility that allows quick and easy generation of TXO-compatible files that can be downloaded to the terminal. Using a DOS program to generate application modules provides two important advantages:

- The application code space is reduced because the data is stored in a file.
- This method provides greater flexibility since the data is not compiled as part of the application. If the data in the file needs to be modified, only the file is updated and downloaded to the terminal, rather than through the application.

Files generated by TXOFILE typically contain data that is not modified by your application program (for example, prompts and messages). Although TXOFILE is discussed here in conjunction with the Message/Data Entry Library, it also supports additional file and compression types that allow you to generate binary files, and so forth. The following sections list the file and compression types supported by TXOFILE. Before you select a file and compression type, review the file type and compression type considerations discussed later in this section.

Compression Types

- 8 bit-to-6 bit (lowercase characters are converted to uppercase)
- 8 bit-to-4 bit (uppercase and lowercase characters are preserved)
- BCD (only digits are compressed)
- No compression

File Types

- VLR
- CVLR
- DVLR
- Fixed
- Keyed
- Message (ASCII)

TXOFIL Command Line Options

The input file should have the following string if two byte addressing is used. This string should precede any valid messages.

```
"/* Byte addressing: 2 */"
```

To install TXOFIL, copy `txofile.exe` and `msgfmt.exe` file to the same directory on your PC. Follow standard DOS pathing conventions (place the file in a PATHed directory or edit the `AUTOEXEC.BAT` PATH line). To display the command line syntax and compression/file type/format, enter TXOFIL with no command line arguments.

TXOFIL Syntax

```
txofile [-options] input_filename output_filename
```

TXOFIL Options

The options are case sensitive. Each option must be preceded by a dash:

```
-p (specifier) (format - Intel or Motorola)
```

- i Indicates that the input file uses two byte addressing and bytes need to be swapped.
- m Indicates that the output file should be generated for Motorola format, that is, the bytes are not swapped.

```
-c (type#) (compression type)
```

- 0 No compression (default)
- 1 8-to-6 bit

- 2 8-to-4 bit (VFI)
- 3 BCD

-t(specifier) (file type)

- c CVLR
- d DVLR
- f Fixed
- k Keyed
- m Message file (default)
- v VLR

NOTE



The default values are no compression and message file.

Example

Assume that input.txt has messages specified in two byte addressing. Add the string “/* Byte Addressing:2 */” to input.txt. This string should precede the list of valid messages. Run txofile.exe as follows:

```
txofile -pi input.txt output.dat
```

The generated message file output.dat can be used with an application running on a PC.

Recommended Usage

Run TXOFILE utility twice for the same input file; once with -pi option and once without the -pi option.

Example

```
txofile -pi input.txt filename.iat
```

This generates the message file for a PC.

```
txofile input.txt filename.dat
```

This generates the message file for a terminal. Note the change in the file extension for the two output files.

In the application, select the message file as follows:

```
#ifdef _WINDOWS
msg_select_file(filename.iat)
#else
msg_select_file(filename.dat)
#endif
```

You do not need to change the application for SDS compilation. You also do not need to download *.iat files to the terminal.

File Type and Compression Type

The file type and compression type options you specify on the command line are determined by the type of data being stored in the file and how it is accessed by your application. The following information explains how TXOFILE handles the various file types.

CVLR, Fixed Length, and Keyed Files

You cannot specify a compression type parameter. TXOFILE automatically compresses CVLR and keyed files using 8-bit-to-4-bit compression. Fixed length and DVLR files are not compressed.

Message Files

These include a two-byte header record. Byte 1 (0x02) indicates the length of the header record. Byte 2 indicates the type of compression used. It is recommended that you select 8-to 6-bit compression for message files.

Hex, Octal, and Decimal Values

TXOFILE supports hex, octal, and decimal values in a quoted string. For example, \x0F is stored in the output file as one byte (0x0F). These values must be preceded by \n where n is one of the following:

- x or X: The two letters/digits immediately following the \X or \x are treated as two hex digits and are stored as one byte in the output file.
- 0–7: The three digits immediately following the \ are treated as three octal digits and are stored as one byte in the output file.
- d or D: The three digits immediately following the \D or \d are treated as three decimal digits and are stored as one byte in the output file.

NOTE



Values represented by hex, octal, or decimal cannot exceed 255 decimal.

Message File Format

A message file is an ASCII file that contains one #define directive for each message (refer to the following example). The #define is not required, however, it allows you to relate the record number to the associated message.

```
#define ENTER_CODE 1
/* Message number for "ENTER CODE" */
#define PRIMARY_PHONE 2
/* Message number for "PRIMARY PHONE#" */
#define ALT_PHONE 3
/* Message number for "ALT PHONE #" */
#define MERCHANT_ID 4
/* Message number for "MERCHANT ID #" */
```

Each line in the message file must conform to the following format:

```
#define [name] [value]
```

```
/* Message number for "[message]" */
```

When TXOFIL processes the input file, the value becomes the record number and the quoted string within the comment becomes the message text in the output file.

NOTE

In the message (quoted string), only include ASCII characters. Escape sequences (for example, hex or decimal) are not valid unless they are indicated by \x0F, \187, \d032, and so on.

Message/Data Entry Functions

The message/data entry functions are described in [Message/Data Entry Engine Function Calls](#).

ISO 8583 Message Interface Engine

The ISO 8583 Message Interface Engine is an interface engine designed to aid in the assembly and disassembly of data packets conforming to the ISO 8583 standard. The engine is beneficial for the application programmer by providing a single routine to complete both assembly and disassembly.

The engine allows for the handling of *computed* fields, such as the MAC, whose value is based on the preceding bytes in the ISO packet. Verix eVo ACT includes two additional modules that rely on the ISO 8583 Message Interface Engine, the ISO 8583 Protocol Engine, and the PIP Engine.

The PIP Engine is a specific ISO 8583 implementation using PIP. The ISO 8583 Protocol Engine assists the PIP Engine in building, sending, receiving, and unpacking an ISO 8583 message. Other ISO 8583 applications can also use the ISO 8583 Message Interface Engine and the ISO 8583 Protocol Engine.

Figure 3 illustrates how the ISO 8583 Message Interface Engine interfaces to the ISO 8583 Protocol Engine.

NOTE



This engine can only be used with Verix eVo ACT, (VPN P006-212-02), for Verix eVo terminals.

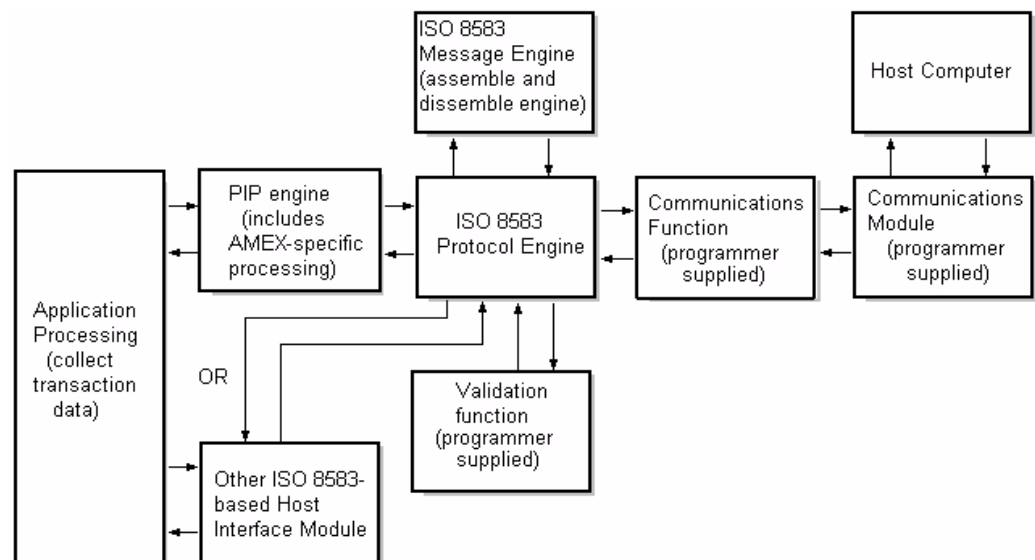


Figure 3 ISO 8583 Message Interface

ISO 8583 Engine Overview

Services of the financial industry include the exchange of electronic messages relating to financial transactions. The ISO 8583 international standard specifies a common interface by which bankcard-originated messages relating to financial transactions can be exchanged between private systems. The standard has nothing to do actual transmittance and receipt of data, but concentrates on the construction of the data packets.

ISO 8583 Bitmaps

The ISO 8583 standard identifies 126 possible fields that can be included in a data packet. For each data packet, various combinations of these data fields are included depending on the type of request or response represented by the data packet. Each of these data fields is represented by a single bit in one of two bitmap elements in the data packet.

The first bitmap field is always included in the data packet. The first bit is used to indicate whether the second bitmap field exists. The ISO 8583 standard arranges the various data fields so that the most common data packets can be formed using only the first bitmap field, which reduces transmission size.

Each of the remaining 63 bits represents a particular data field as described in the ISO 8583 standard. The application programmer must review the host interface specification to ensure the correct fields are included for each message type. For every data field included in the data packet, the corresponding bit in the bitmap is set to one. Any unused fields must set their corresponding bit to zero. The bitmap is completed as the data packet is assembled, and is used at the destination to disassemble the data.

Figure 4 represents a partial first bitmap field.

Bitmap = i

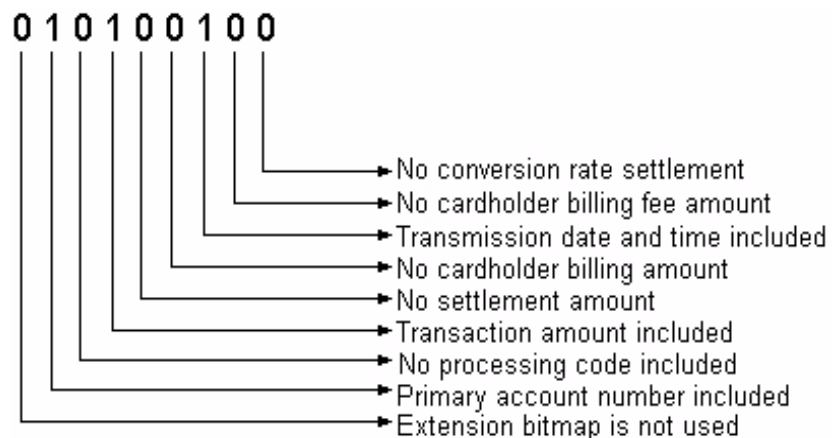


Figure 4 Partial First Bitmap Field

Message Structure

Each ISO 8583 message is constructed in the following sequence:

- 1 Message type identifier
- 2 One or more bit maps
- 3 A series of data elements in the order of bitmap representation

The message identifier is a four-digit numeric field used to describe each message class and function. Digits one and two identify the class of message, while digits three and four represent the message function or the transmission mode when digits one and two range between 01 and 08. For example:

Identifier = 0100

Message class is 01, authorization message

Message mode is 00, transaction processed, interactive

The second message component is one or more bitmap(s), as previously described.

The third message component and its data content is made up of a series of data elements. Data elements can be of fixed or variable length. The actual length of any variable-length-data element is provided in its fixed-length prefix. Note that the message structure does not preclude the use of additional data elements in a message as required for national interchange or private use. Some fields are reserved for private and other uses. If necessary, additional bitmaps can be included and the appropriate data fields defined. Additional field definitions require mutual agreement by the parties involved.

Applications developed for systems using the ISO 8583 international standard must format all messages accordingly. The construction of each message is a tedious job, and most programmers do not really want to know how the messages are constructed. Most programmers like to manipulate data in a form that is convenient for their requirements, not in a foreign form required by the interface. This implies that the data resides in a subset of the application variables and stored in formats selected by the programmer.

ISO 8583 Engine Design

This engine is designed to be table driven. A single routine is used for the assembly and disassembly of ISO 8583 packets. The assembly and disassembly of ISO 8583 packets is driven by the following structures:

Maps	One or more collections of 64 bits that drive packet assembly and indicate what is in a message.
Field table	Defines all the fields used by the application.
Convert table	Defines data-conversion routines.
Variant tables	Optional tables used to define variant fields.

The [process_8583\(\)](#) routine is used for the assembly and disassembly of ISO 8583 packets.

Engine Components

The ISO 8583 Interface Engine is divided into two major parts:

- A set of map manipulation and data conversion routines, and
- A set of routines to handle packet assembly and disassembly.

Engine Files

The ISO 8583 Message Interface Engine consists of the following files:

Engine Header File	ISO8583.H	Function headers and <code>#define</code> preprocessors for the bitmap manipulation and data conversion routines.
		Contains the primary data structures, required global variables, and function headers for the primary interface routine.
	APPL8583.C	File to be modified by the application programmer to include the variables used to interface to the assembly and disassembly routines. Contains the field tables, variant field tables (if any), and the two required integer functions for processing variant fields. This file is listed at the end of this manual section.
Sample Application Files	APPL8583.H	Companion header file for <code>APPL8583.C</code> .

Incorporating the Engine

To incorporate this engine into an application, modify the `APPL8583.C` and `APPL8583.H` files so that each has all the application variables required in the bit map and set up the map properly. Compile `APPL8583.C` and link it with your application and the ISO 8583 library.

Use the following procedures to transmit or receive an ISO 8583 packet using the ISO 8583 Interface Engine:

To transmit an ISO 8583 packet

- 1 Set data values in the application variables for those to transmit.
- 2 Call the `prot8583_main()` routine.
This constructs the complete message and returns the number of bytes in the constructed message.
- 3 Call `write()` to transmit the message.

To receive a message

- 1 Call `read()` to receive the message.
- 2 Call the `process_8583()` routine.
This results in all fields being deposited into the application variables.
- 3 Use the values in the application variables.

NOTE



The routines provided by this engine simplify the handling of bitmapped messages. The determination of which message type to send and which fields to include in that message is the responsibility of the application programmer. Similarly, validation of a host response and the fields it provides is the responsibility of the application.

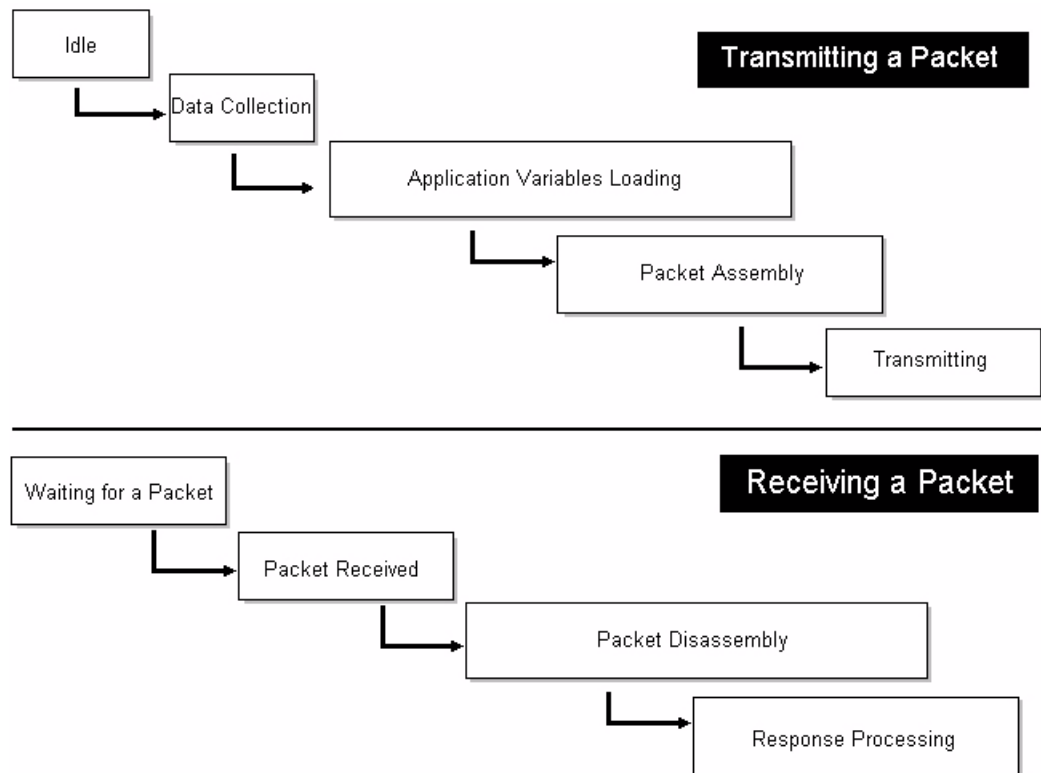


Figure 5 Transaction Processing Flowchart

Global Data and Definitions

Global Defines

The following `#define` preprocessors are used by the 8583 Interface Engine and can be found in `ISO8583.H`:

```

#define BIT_MAP_SIZE 8
#define map_bytes(fn) (((fn) - 1) * (BIT_MAP_SIZE * 8)) /
+ 1) * BIT_MAP_SIZE)
#define bit_map(name, fn) unsigned char name[map_bytes(
fn)]
#define STOP 0x8000
#define SKIP 0x4000
#define FIELD_MASK 0x0fff
#define OFF 0x4000
#define LEGAL_FIELD(map, fn) ((fn) >= 1) && ((fn) <= (map_size(map) *
BIT_MAP_SIZE))
#define LS_MAP_BIT 0x80
#define ASC_ASC 0
#define AV3_AV3 1
#define BIT_BIT 2
#define BCD_BCD 3
#define BCD_ASC 4

```

```
#define ASC_STR 5
#define BCD_STR 6
#define BCD_SNZ 7
#define AV2_STR 8
#define BV2_STR 9
#define AV3_STR 10
#define XBC_STR 11
#define BIN_HST 12
#define BI2_HST 13
#define BI3_HST 14
```

Global Variables

The following global variable can be found in LIB8583.C:

```
unsigned char pad_nibble_8583 = 0;
```

The following global variables are used by the 8583 Interface Engine and can be found in ISO8583.H:

```
converters convert_table[];
unsigned char *src_8583;
unsigned char *dst_8583;
int fn_8583;
unsigned char map_8583 [];
```

Global Structures

The following global structures are used by the Interface Engine and can be found in ISO8583.H:

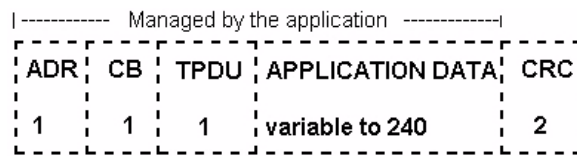
```
typedef struct
{
    int field_num; /* 1st col - field number */
    int packet_sz; /* 2nd col - packet size */
    int convert_idx;
    /* 3rd col - index into covert table */
    void *reference;
    /* 4th col - pointer to variable or table */
    int var_sz; /* 5th col - variable size */
} field_struct;

typedef struct
{
    unsigned int variant1; /* 1st col - target */
    unsigned int variant2; /* 2nd col - target */
    int packet_sz; /* 3rd col - packet size */
    int convert_idx; /* 4rd col - index into covert table */
    void *reference;
    /* 5th col - pointer to variable or table */
    int var_sz; /* 6th col - variable size */
} variant_struct;

typedef void (*(converters[2])) ();
```

Typical Host/ Terminal Packet Structure

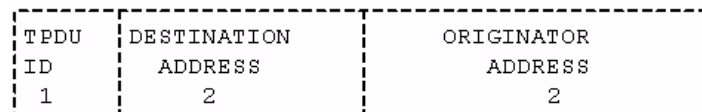
The structure of a terminal/host message is:



where,

- ADR: is the HDLC (SDLC) poll address (normally 0x30).
- CB: is the HDLC control byte.
- TPDU: is the Transport Protocol Data Unit (see below).
- CRC: is the HDLC checksum (CCITT CRC).

The structure of the request message TPDU is:



where,

- TPDU ID: identifies TPDU type:
 - 0x60 = Transactions
 - 0x68 = NMS/TNMS
- DESTINATION ADDRESS: is the network international identifier
- ORIGINATOR ADDRESS—Identifies the individual terminal or process originating the transaction.

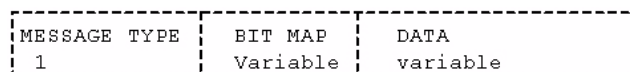
The structure of the response message TPDU is:



where,

- TPDU ID: Identifies the TPDU type. This is the same value as in the request message.
- DESTINATION ADDRESS: This is the same as the originator address from the request message.
- ORIGINATOR ADDRESS: This is the same as the destination address from the request message.

The structure of APPLICATION DATA is:



where,

- MESSAGE TYPE: is four BCD digits
- BIT MAP: 64 or 128 bits numbered from the left, starting with 1.
- DATA: The following rules apply:
 - All data elements begin on a byte boundary.
 - Fixed-length, n -type fields with an odd length are right-justified to a byte boundary, and zero-filled on the left.
 - All lengths for variable-length fields are represented in BCD, right-justified to a byte boundary, and zero-filled on the left.
 - The length indicator for a variable-length field is a count of data elements to follow. It does not include itself in the count.
 - Variable-length, n -fields with an odd length are left-justified within a field and zero-filled.

Application Design

Map Construction

When designing an application that uses the ISO 8583 Interface Engine, the programmer must understand how to construct maps and field tables. The following sections describe these two topics in detail.

A map is one or more 64-bit collections used to define which fields to include in an outgoing packet and to indicate which fields are present in an incoming packet. There is a direct correspondence between the bits in the map and the order of the data fields in the packet. The first bit that is on in the map (moving from bit 1 to 128) defines the number of the first data field. Bits 1 to 64 constitute the primary bit map, while bits 65 to 128 make up the optional secondary bit map. Bit number 1 indicates the presence of the secondary bit map, with bit 65 indicating the presence of a tertiary map. The current ISO 8583 standard only defines fields up to 128, and hence it uses only two maps.

The application programmer typically creates a map for each packet which the application is required to send. These maps may be handled in any one of the following ways:

- Create a table of maps used by the application then refer to a particular map with an index into this table.
- Use individually named maps.
- Create each map on the fly just before packet assembly.
- Use a mixture of the above methods.

The ISO 8583 Engine handles maps with up to 256 bits.

Map Manipulation Routines

In most cases, maps are created once for each particular message type and then used without change. Occasionally however, it can be necessary to manipulate these maps. To assist in this process, the ISO 8583 Interface Engine provides the following map manipulation functions:

- map_clear()
- map_test()
- map_set()
- map_reset()
- map_man()

Field Table Construction

Use the field table to define all fields required by the application and transmitted by the host. There are as many rows in the field table as there are unique fields. Each row in the table has the following five fields:

Field Num	The number of the field. Corresponds to the bit set in the bitmaps.
8583 Size	The number of units called for by the 8583 packet. These units could be single bits, 4-bit BCD digits, or 8-bit bytes.
Convert Index	An index into the Convert table, as explained in Variant Field Strategy .
Application Variable	A pointer to the actual variable or place where the data resides within the application. May also point to a second table, as is the case for variant fields.
Variable Size	Number of units called for by variable.

The header file, ISO8583.H, defines the following structure which is used for the field table:

```
typedef struct {
int field_num;
    /* 1st col - field number */
int packet_sz;
    /* 2nd col - packet size */
int convert_idx;
    /* 3rd col - index into convert table */
void *reference;
    /* 4th col - pointer to variable or table */
int var_sz;
    /* 5th col - variable size */
}
field_struct;
```

The following #define preprocessors are provided in APPL8583.H:

```
#define VARIABLE(name)
(void *) name, sizeof(name)
#define TABLE(name)
(void *) name, 0
#define FUNCTION(name)
(void*) name, 0
```

If the application specified data length (8583 packet_sz) is less than the actual data length, then it will fail returning an error (-4).

Correct the value of 8583 packet_sz to reflect the maximum length of the data field and inspect 8583 packet_sz field while using the following conversions:

AV2_STR, BV2_STR, AV3_STR, HST_BI2, HST_BI3, STR_AV2, STR_BV2, STR_AV3, BI2_HST, BI3_HST, and AV3_AV3

The following is an example of the field table:

```
field_struct field_table [] =
{
    /* Fld 8583 Convert Variable name and size */
    /* no. sz index */
    { 0, 10, BCD_STR, VARIABLE(tpdu) },
    { 0, 4, BCD_STR, VARIABLE(message_id) },
    { 2, 19, BV2_STR, VARIABLE(pri_acct_num) },
    { 3, 6, BCD_STR, VARIABLE(proc_code) },
    { 4, 12, BCD_SNZ, VARIABLE(amount) },
    { 11, 6, BCD_STR, VARIABLE(sys_trace) },
    .
    .
    .
    { 32, 11, BV2_STR, VARIABLE(acq_ins_id) },
    { 35+ SKIP, 37, AV2_STR, VARIABLE(discard) },
    { 37, 12, ASC_ASC, VARIABLE(retr_ref_num) },
    .
    .
    { 43, 40, ASC_STR, VARIABLE(card_acc_name) },
    { 44, 0, COMPUTE, FUNCTION(build_44) },
    { 60+ STOP, 0, VARIANT, TABLE(field_60) },
};
```

This field table must define all fields that the host will send, including those not used by the application. This is necessary to allow the engine to skip them. However, to skip a field, the engine must know how long that field is. See field 35 in the example above.

Packets are assembled and disassembled from left to right, that is, from low field number to high field number. During packet assembly, an index is used to reference entries in the field table. This index increments to the next field called for by the driving bitmap.

Use the following steps to construct the field table:

- 1** Create column one for each field used by the application or transmitted by the host.
- 2** Fill in the values for the second column taken from your host interface document.

Use the field sizes specified by the particular host with which the application is communicating.

- 3 Make a list of the types of fields required by the ISO 8583 packets from the host interface document.

Here is the start of the list:

BCD Binary coded decimal. Length given by packet size column.

BV2 BCD preceded by 2-digit length.

AV3 ASCII string preceded by a 3-digit length.

These are the symbolic names used for the first part of the entries in the third column (the names to the left of the underscore). See the header file ISO8583.H for a complete list of all supported data types.

- 4 Decide on the format the application will store and manipulate the data for each field.

Give each of these types a descriptive name, as described in step 3. Use the same name if the forms are the same. These are the symbolic names used for the second part of the entries in the third column (the names to the right of the underscore). There are also two special `#defines`, `COMPUTE` and `VARIANT`, to use with computed or variant fields. Use one of the following three defines for the fourth column: `VARIABLE`, `TABLE`, or `FUNCTION`:

- `VARIABLE`: if the data is stored in a simple program variable.
- `TABLE`: specifies variant fields.
- `FUNCTION`: specifies computed fields.

See [Variant Fields](#) and [Computed Fields](#).

Variant Fields

The 8583 specification defines fields 60–63 as *private* fields. This means that these fields can be used by the application in any way. Following are several possibilities for the usage of these fields:

- Always use them the same way to store the same type of data. If this is done, these private fields be used as if they were defined fields and are no more difficult to handle than any other field.
- Let their usage and contents to vary depending of the type of message. Since there are only four of these fields and since a lot of data simply does not fit into the other defined fields, most applications are forced to think of these private fields as variant fields. Accordingly, the ISO 8583 Interface Engine provides a mechanism to manipulate variant fields. Note that this mechanism can be ignored or modified by the programmer— this is only one method and works in most cases. For completeness, two other schemes are provided that could be used to handle variant fields.
- Define special conversion routines that use other application variables to determine exactly what they should do.

- Always move the field onto a C structure that uses a union to offer different views of the data.

Variant Field Strategy

Refer the previous field table example and note the following for field 60:

```
{ 60+ STOP, 0, VARIANT, TABLE( field_60) },
```

Column three normally has an index into the convert table. The define VARIANT is a special value defined to be one greater than the last valid index into the convert table. If this value is used it simply tells the engine “the contents of this field depend on the two additional integers”. These two integers can be the first two digits of the processing code and the message type. Accordingly, the pointer in the fourth column, instead of pointing at an application variable points at a table of the following form:

```
variant_struct field_60[] =
{
    /* Variant1, Variant2 8583 Size Convert index, variable name and size
    */
    { 00, 0x0220, 12, BCD_STR, VARIABLE(orig_amount) },
    { 02, 0x0220, 12, BCD_STR, VARIABLE(orig_amount) },
    { 90, 0x0810, 6, BCD_STR, VARIABLE(batch_num) },
    { 92+ STOP, 0x0500, 6, BCD_STR, VARIABLE(batch_num)
    },
};
```

The format of a variant field table is simply that of the basic field table with two new columns added to the left. When the interface engine encounters a variant field, it moves to the variant field table and then sequentially searches for a row where the integers specified by the first two columns match two integers provided by the application.

If a match is not found, the engine aborts the assembly or disassembly and returns an error code. Due to the sequential search, the variant field tables must be ordered with the most-common definition first and the least common last.

Exactly what these two integers represent is up to the application, as they are provided by two functions found in the file APPL8583.C. The above table uses the first two digits of the processing code and the message . Accordingly, these two functions are:

```
unsigned int return_variant1()
{
    unsigned char x [3];
    /* Most significant 2 digits of proc_code */
    strncpy( x, proc_code, 2);
    return atoi( x);
}

unsigned int return_variant2()
```

Computed Fields

```
{
    return atoi( message_id);
}
```

When the match is found, the interface engine proceeds to use the other columns of the table just as with non-variant fields.

A computed field is one whose value depends on either the partially assembled ISO 8583 packet or some other real time data. In other words, the value of the field prior to calling the `process_8583()` routine to build the packet cannot be determined. Such an example would be a MAC. The value of this field is determined using the assembled ISO 8583 packet. The interface engine caters to these types of fields by providing a mechanism whereby during assembly a call will be made to a routine written by the user that will be responsible for providing the value of the fields. This user-written routine should be included in `APPL8583.C` and must be of the form:

```
/******
/* Example of a computed field. */
/* Such a routine must have four parameters. */
/* how - how= 0 = pack; how= 1 = unpack */
/* buffer - Pointer to working buffer. Copy of value
/*   passed to the main process_8583 driver. */
/* psz - Number of units in packet. Must be set by */
/*   this routine. */
/* vsz - Number of bytes in variable. Must be set by
/*   this routine. */
/******
int compute_field (how, buffer, psz, vsz)
    int how; unsigned char *buffer;
    int *psz; int *vsz;
{
    *psz = 19 /*value from second column of field table */
    *vsz = sizeof (field_02); /* size of data in application variable */
    if (how == 0) /* We are building a packet */
        src_8583 = (unsigned char *) field_02; /* Pointer to
        source data */
    else
        dst_8583 = (unsigned char *) field_02;
        /* Pointer to destination data */
    return
}
```

If a computed field is encountered while the interface engine is building a packet, it will first call a user-written routine of the form shown above. This initializes the data value, establishes the construction pointers, sets the sizes, and ultimately returns the conversion type. The mechanism then has all the data as if it were presented explicitly in the field table. The value will then be properly formatted into the packet and the construction process will continue.

Convert Table

The convert table contains pairs of pointers that point to simple routines used to convert the data between the form required by the ISO 8583 packet and the application. For example, it may be most natural for the C application to store the expiration date as a null-terminated string. The packet, however, requires that it be stored as four BCD digits. Therefore, two conversion routines are required: One to change a null-terminated string to packed BCD, and one to do the reverse.

Scan column three of the field table to determine which conversion routines are required. Each conversion results in a pair of routines: one named A_to_B() and the other B_to_A(). The convert table contains as many rows as there are unique conversion pairs. For example:

Assembly	Disassembly	Description
str_to_bcd()	bcd_to_str()	ASCII, null-terminated, to BCD
asc_to_bcd()	bcd_to_asc()	ASCII, count from field table, to BCD

A #define is created for each row in the convert table, which is then used in column 3 of the field table. For example:

```
#define BCD_STR 1 /* this type of define */
/* found in ISO8583.H */
#define BCD_ASC 2 /* this type of define */
/* found in ISO8583.H */
```

The convert table is created using the following type declaration:

```
typedef void (*(converters[2])) ();
/* a pair of pointers */
/* to functions */
```

The ISO8583.H file contains the #defines for conversion types, and the ISO 8583 library contains the routines that perform the following conversions:

Table 5 ISO 8583 Library Conversion Routines

Convert Index	Description
0	Fixed-size ASCII to same.
1	3-digit ASCII counted to same.
2	Fixed-size bits to same.
3	Fixed-size BCD to same.
4	Fixed-size BCD to ASCII with same number of digits.
5	Fixed-size ASCII to null-terminated string.
6	Fixed-size BCD to null-terminated string.
7	Fixed-size BCD to null-terminated string (drop lead zeros).
8	2-digit ASCII counted to null-terminated string.
9	2-digit BCD counted to null-terminated string.
10	3-digit ASCII counted to null-terminated string.
11	Fixed-size "signed" (C/D) BCD to null-terminated string.
12	Fixed-size binary to ASCII hex.

Table 5 **ISO 8583 Library Conversion Routines** (continued)

Convert Index	Description
13	2-digit counted binary to null-terminated ASCII hex string.
14	3-digit counted binary to null-terminated ASCII hex string.

The convert indices in ISO8583.H are as follows:

```
#define ASC_ASC 0
#define AV3_AV3 1
#define BIT_BIT 2
#define BCD_BCD 3
#define BCD_ASC 4
#define ASC_STR 5
#define BCD_STR 6
#define BCD_SNZ 7
#define AV2_STR 8
#define BV2_STR 9
#define AV3_STR 10
#define XBC_STR 11
#define BIN_HST 12
#define BI2_HST 13
#define BI3_HST 14
```

The convert table itself is actually created with an initialized variable in APPL8583.C:

```
converters convert_table [] = {
    { asc_to_asc, asc_to_asc },
    { av3_to_av3, av3_to_av3 },
    { bit_to_bit, bit_to_bit },

    { bcd_to_bcd, bcd_to_bcd },
    { asc_to_bcd, bcd_to_asc },
    { str_to_asc, asc_to_str },
    { str_to_bcd, bcd_to_str },
    { snz_to_bcd, bcd_to_snz },
    { str_to_av2, av2_to_str },
    { str_to_bv2, bv2_to_str },
    { str_to_av3, av3_to_str },
    { str_to_xbc, xbc_to_str },
    { hst_to_bin, bin_to_hst },
    { hst_to_bi2, bi2_to_hst },
    { hst_to_bi3, bi3_to_hst }
};
```

The following convert indices are defined in ISO8583.H:

```
#define VARIANT -1
```

```
#define COMPUTE (VARIANT- 1)
```

These indices identify the variant and computed fields.

Each conversion routine is called with a single integer parameter, obtained from column two of the field table. In addition, each routine must advance the following two global character pointers:

`src_8583` Points at the source data. On an incoming packet, this points at the packet. On an outgoing packet, this is set to the value in the fourth field of the field table.

`dst_8583` Points at the destination data with reverse orientation from `src_8583`.

In addition, the following global variables are available to the conversion routine and can be used as necessary:

`fn_8583` Current field number being worked on.

`map_8583` Copy of current working map.

As an example of one of these conversion routines, the following converts a packed BCD string of `cnt` digits into a null-terminated string of ASCII digits:

```
void bcd_to_str (cnt)
int cnt;
{
    bcd_to_asc(cnt);
    *dst_8583++ = 0;
}
void bcd_to_asc(cnt) int cnt; {
    while (cnt > 0)
    {
        *dst_8583++ = get_bcd_to_asc(cnt-- & 1 ?
        *src_8583++ :
        *src_8583 >> 4);
    }
} static unsigned char get_bcd_to_asc( bcd) unsigned char bcd;
{
    return (bcd & 0xf) + '0';
}
```

Conversion Routines

The following routines are included in Verix eVo ACT to perform all data conversions required to format ISO 8583 packets. Refer to [ISO 8583 Message Interface Engine Function Calls](#) for details about these routines.

- `asc_to_asc()` copies *n* bytes from source to destination. It then advances the global pointers `dst_8583` and `src_8583` by *n*.

- `av3_to_av3()` copies a 2-byte counted string from source to destination. The 2-byte count is in BCD format. If the value of count bytes exceeds n , no move is performed and the operation is terminated.
- `bit_to_bit()` copies n bits from the source to destination, always in whole bytes.
- `bcd_to_bcd()` moves n BCD nibbles from source to destination.
- `asc_to_bcd()` converts n ASCII digits to packed BCD and stores them in the destination.
- `bcd_to_asc()` expands n nibbles of packed BCD into ASCII equivalents. It assumes all the BCD digits are in the range 0–9.
- `str_to_asc()` converts a null-terminated ASCII string to fixed-size (n) ASCII. The resulting string is padded with spaces or truncated, as necessary, to n bytes.
- `asc_to_str()` moves n characters from source to destination then appends a null in the destination.
- `str_to_bcd()` converts a null-terminated ASCII string to fixed-size (n) BCD nibbles with zero padding on the left.
- `bcd_to_str()` copies n BCD digits from the source to destination, then appends a null character in the destination. Leading zeros in the source are preserved.
- `bcd_to_snz()` copies n nibbles of BCD to a null-terminated ASCII string. Leading zeros are removed, but are included in the conversion count.
- `str_to_av2()` converts a null-terminated ASCII string to a 1-byte counted string. The count is in BCD form. If the size of the ASCII string exceeds n bytes, no move occurs and the operation terminates.
- `av2_to_str()` converts a 1-byte counted string into a null-terminated ASCII string and stores the result in the destination. The count is in BCD form. If the value of count byte exceeds n , no move occurs and the operation terminates.
- `str_to_bv2()` converts a null-terminated ASCII string (contains digits only) to a 1-byte counted BCD string. The count is in BCD form. If the size of the ASCII string exceeds n bytes, no move occurs and the operation terminates.
- `bv2_to_str()` converts a 1-byte counted BCD string to a null-terminated ASCII string. If the value of count byte exceeds n , no move occurs and the operation terminates.
- `str_to_av3()` converts a null-terminated ASCII string to a 2-byte counted string. The count is in BCD form. If the size of the ASCII string exceeds n bytes, no move occurs and the operation terminates.
- `av3_to_str()` converts a 2-byte counted ASCII string to a null-terminated ASCII string. The byte count is in BCD form. If the value of count byte exceeds n , no move occurs and the operation terminates.
- `str_to_xbc()` converts a null-terminated ASCII string to a BCD string while preserving the first byte of the source (C = Credit, D = Debit).

- `bcd_to_str()` converts n bytes of a BCD string to a null-terminated ASCII string, preserving leading zeros.
- `hst_to_bin()` converts n bytes of ASCII hex digits to binary and stores them in the destination.
- `bin_to_hst()` converts n bytes of binary into $2n$ bytes of ASCII hex digits.
- `hst_to_bi2()` converts a null-terminated ASCII hex string to a 1-byte counted binary and stores it in the destination. The count is in BCD format and is only half the length of the ASCII hex string. If the size of the ASCII string exceeds n bytes, no move occurs and the operation terminates.
- `bi2_to_hst()` converts a 1-byte counted string of binary to a null-terminated ASCII hex string. If the value of count byte exceeds n , no move occurs and the operation is terminates.
- `hst_to_bi3()` converts a null-terminated ASCII hex string to a 2-byte counted binary and stores it in the destination. The count is in BCD format and is only half the length of the ASCII hex string. If the size of the ASCII hex string exceeds n bytes, no move occurs and the operation terminates.
- `bi3_to_hst()` converts a 2-byte counted string of binary to a null-terminated ASCII hex string. If the value of count bytes exceeds n , no move occurs and the operation is terminates.

Packet Assembly and Disassembly

The routine used for the assembly and disassembly of ISO 8583 packets is `process_8583()`. It accepts five parameters from the caller. These parameters are as follows:

- `how`: An integer type that determines if the function should assemble or disassemble a packet. 0 indicates assembly; a 1 indicates disassembly.
- `field_tbl`: A pointer to a struct `field_struct` that is the field table.
- `map`: An unsigned char pointer that points to the map defining the packet being processed. This parameter is ignored when disassembling an incoming packet.
- `buffer`: An unsigned char pointer that points to a buffer area used in both assembling and disassembling a packet.
- `limit`: An integer type that denotes the maximum size for the packet being processed.

The function prototype is as follows:

```
int process_8583(how, field, map, buffer, limit)
int how;
field_struct *field_tbl;
unsigned char *map;
unsigned char *buffer;
int limit;
```

This function processes assembly or disassembly of a packet by checking the bits in the map. The return values for this integer function are as follows:

- Success: >= 0: Assembly: Number of bytes placed into the buffer. Use for writes.
 Disassembly: Number of bytes in the buffer not yet processed. Should be zero.
- Failure: -1: Field not defined in the field table. The variable fn_8583 contains the field in error.
- 2: Exceeded the destination buffer limit while packing.
- 3: Exceeded the source buffer limit while unpacking.
- 4: Variable-length field conversion exceeded limit parameter.
- 5: No matching variant field definition.

The following global variables are used in this function and are defined in the ISO8583.H header file:

```
extern unsigned char *src_8583; /* Source pointer */
extern unsigned char *dst_8583; /* Destination pointer */
extern int fn_8583; /* Field number */
extern unsigned char map_8583[]; /* Map */
```

ISO 8583 Message Interface Functions

The ISO 8583 message interface functions are described in [ISO 8583 Message Interface Engine Function Calls](#).



ISO 8583 Protocol Engine

The ISO 8583 Protocol Engine builds and sends ISO 8583 packets to a host. To accomplish this, it interfaces with the ISO 8583 Message Engine. The ISO 8583 Protocol Engine adheres to all requirements in the ISO 8583 standard.

The ISO 8583 Protocol Engine interfaces to the programmer defined validation function used to determine if an appropriate response was received. This engine interfaces to the programmer-defined communication function used to transmit and receive request and response messages. It processes reversals as specified in the ISO 8583 standard.

The PIP Engine requires the ISO 8583 Protocol Engine, but the ISO 8583 Protocol Engine does not require the PIP Engine. It does, however, require the ISO 8583 Message Engine.

Although this engine was written to support the PIP Engine and the ISO 8583 process, it could be used to interface to any engine that uses ISO 8583 packets for processing. This section treats the ISO 8583 Protocol Engine as a generic module. See [PIP Engine](#) for specific implementation of the ISO 8583 Protocol Engine

This engine can only be used with Verix eVo ACT, (VPN P006-212-02), for Verix eVo based terminals.

ISO 8583 Protocol Engine Data Structure

The ISO 8583 Protocol Engine requires a data structure to provide information and application data variables. By using a structure to contain this information, the interface to the ISO 8583 Protocol Engine is simplified. The protocol 8583 data structure is defined in the PROT8583.H header file.

Prototype

```
typedef struct c8583dat
{
    field_struct *p_fld_table;
    int *comm_handle;
    unsigned comm_timeout;
    unsigned wait_for_carrier;
    unsigned char *map;
    int (* validation) (void *, int);
    void *v_parms;
    unsigned v_parms_size;
    int (* transceiver) (int, int, unsigned char *, unsigned,
        unsigned char *, unsigned, unsigned, unsigned);
}
```

```

    unsigned char *transmit;
    unsigned int transmit_size;
    unsigned transmit_limit;
    unsigned char *receive;
    unsigned receive_size;
    unsigned receive_limit;
    unsigned char *reversal;
    unsigned reversal_size;
    unsigned reversal_limit;
    unsigned char *base_name;
    unsigned char rev_msg_id[5];
    int state;
} COMM_8583_DATA;

```

Parameters

<code>p_fld_table</code>	The ISO 8583 Protocol Engine calls the ISO 8583 Message Engine to assemble and disassemble messages for each transaction. The <code>p_fld_table</code> member is a pointer to the applications field table that contains all the required information.
<code>comm_handle</code>	A pointer to the communications device. This handle is passed to the communications routine as required.
<code>comm_timeout</code>	A time-out value specifying the length of time to wait for a response from the host. This parameter is passed to the communications routine as required.
<code>wait_for_carrier</code>	A time-out value passed to the communications routine to specify the period of time to wait if the carrier is not present. In applications where the call to the host is initiated early in the transaction, the carrier may already be present when the message is available for transmission. In other cases, it is necessary for the communications routine to wait for carrier to indicate the connection to the host is complete.
<code>map</code>	A pointer to the completed bitmap for the transaction. This bitmap can be constructed by the application in various ways or at various times, including as a file.
<code>validation</code>	<p>A function pointer called after receiving a response to a request message. The function should return a 0 if the response is valid, or -1 to -14 if the validation routine fails. Any positive return value is taken as a request to wait for a new response packet. The return value is, therefore, the number of seconds to wait for the next response. The <code>#define RETRY_CURRENT_TIMEOUT</code> may be returned to request a new response packet using the current time-out value.</p> <p>Two parameters are passed to this function: a pointer to the validation structure, and the result of an internal validation of the TPDU and response message ID by the Protocol 8583 Engine (refer to check_8583_tpdo_msg_id() for a list of valid values for this parameter).</p>

v_parms	A pointer to the application validation structure, v_parms_size specifies the size of that structure. By using a structure to contain the validation parameters, parsing is reduced and comparisons are easier.
v_parms_size	For AMEX transactions, the response message is validated by comparing the amount, systems trace number, and terminal ID to the request message. During disassembly, some application variables are overwritten (such as, the transaction amount) by the field information in the response.
	Note: To preserve this information, the validation structure is created to hold the values needed to validate the response.
	v_parms_size builds the reversal and advice files, and is widely used throughout the ISO 8583 Protocols Engine. v_parms_size must be set to include the RRN field. It uses the space to manage the RRN under time-out conditions.
transceiver	A function pointer used as the entry point to the communications routine. This routine is called to send or receive any messages.
transmit	These three parameters define the buffer used to contain outgoing messages. The buffer must be large enough to contain the largest transmit message and the size of the validation structure. transmit_size is the size of the message to send. transmit_limit is the maximum size of the buffer.
transmit_size	
transmit_limit	
receive	These three parameters define the buffer used to contain inbound messages. The buffer must be large enough to contain the largest message that will be received. receive_size is the size of the message to be received in the buffer. receive_limit is the maximum size of the buffer.
receive_size	
receive_limit	
reversal	The ISO 8583 Protocol Engine provides reversal processing. These three parameters process reversal requests and must be identical to the transmit buffer requirements (size and limit).
reversal_size	
reversal_limit	
base_name	String used to construct a separate reversal file. The file extension .REV is appended to base_name when the reversal is created. This allows the ISO 8583 Protocol Engine to support different hosts and separate reversal files.
rev_msg_id[5]	Like all other ISO 8583 messages, reversals require a message ID. The ISO 8583 Protocol Engine must be able to create a reversal for any transaction at any time. rev_msg_id[] must be loaded with the appropriate reversal message ID for the host.

state

Flag to determine the processing state of the ISO 8583 Protocol Engine. Uses TRAN_REQ (transaction request), REV_REQ (reversal request), or NEW_REV_REQ (new reversal), as defined in `PROT8583.H`. The state is included in the control structure so that the application can determine the current transaction being processed. This is useful in controlling the types of messages displayed during communications. The application treats this as a read-only parameter.

Programmer-Defined Functions

Although the ISO 8583 Protocol Engine provides interfaces to the validation and communications functions, the programmer is responsible for writing these functions. This allows programmers to meet their application-specific requirements within the framework of the ISO 8583 standard. For examples on the user routines and how they should look, see the example file `txrx8583.c` supplied with the Verix eVo ACT library.

mdm_tr_8583

A user-defined function that transmits and receives ISO 8583 packets. See the `TXRX8583.C` file in the Verix eVo ACT library.

Prototype

```
int mdm_tr_8583(int h_device, int mode, unsigned char *req_buf, unsigned
               req_size, unsigned char *rcv_buf, unsigned rcv_limit,
               unsigned timeout, unsigned carrier_timeout);
```

Parameters

<code>h_device</code>	Device handle.
<code>mode</code>	Transmit, receive, or TX/RX mode of modem.
<code>req_buf</code>	Buffer containing request packet.
<code>req_size</code>	Size of request packet.
<code>rcv_buf</code>	Buffer to hold received packet.
<code>rcv_limit</code>	Maximum size of receive packet.
<code>timeout</code>	Maximum wait period to wait for receive packet.
<code>carrier_timeout</code>	Maximum wait period for the carrier to become active.

Return Values

The following returns inform the ISO 8583 Protocol Engine that a specific transmit/receive failure occurred.

Success: The number of characters received.

Failure: Negative: Failure.

TR_CLR_KEY: The [CLEAR] key was pressed.

TR_LOST_CARRIER: The carrier was lost.

TR_RD_FAIL: The transmit failed.

TR_TO_RESPONSE: Transmit error. Any other negative value returns as a failure of the ISO 8583 Protocol Engine.

def_valid_8583

A user-defined function that validates the received packets. See the `TXRX8583.C` file in the Verix eVo ACT library.

Prototype

```
int def_valid_8583(int *parms, int com_result);
```

Parameters

<code>parms</code>	Parameter used by the routine.
<code>com_result</code>	Result of the return from the communications call.

Return Values

Success:	0
Failure:	Positive: Wait the number of seconds specified in the return value. Negative: Failure. RETRY_CONTROL_TIME: Wait another time-out.

prot_8583()

Transmits and receives a transaction using the ISO 8583 protocol. The PIP Engine of Verix eVo ACT is in compliance with the ISO 8583 Messaging standard, manage a potential power failure by creating a reversal file (when applicable) immediately before the transaction request is sent, and delete the reversal upon appropriate (non-reversal) return from the communication function.

NOTE



The application variables must all be loaded prior to calling this function.

A valid field table must be declared and all appropriate information loaded. The standard global variables for transmit, receive, store, sizes, limits, and so on must be declared and appropriate for the packets being built. The modem engine is expected to be available if the provided communication routines are to be used. The ISO 8583 engine for packet assembly/disassembly must be available.

Prototype

```
int prot_8583(COMM_8583_DATA *control_data, int rev_opt, unsigned
             resp_timeout);
```

Parameters

<code>control_data</code>	Protocol engine structure for this transaction.
<code>rev_opt</code>	Reversal option: <ul style="list-style-type: none"> • REV_OFF, • REV_ON, • or REV_ONLY.
<code>resp_timeout</code>	Time-out period for response.

Return Values

Returns a negative return value if the function fails. `errno` is set with the actual error code.

Success: 1: Received packet has been unpacked into the application

Failure: TR_CLR_KEY: Clear key pressed.

TR_LOST_CARRIER: Carrier lost by modem engine.

TR_RD_FAIL: Transmit failed in modem engine.

CREATE_REVERSAL: A reversal was created.

PROT8583_UNPACK_ERROR: Unpack routine failed.

TR_TO_RESPONSE: A transmit error occurred in the modem routine.

REQ_FAILED: Request failed.

PROT8583_FAILED_REVERSAL: Reversal failed.

PROT8583_FAILED_TRANSMIT: Transmit failed.

PROT8583_PARM_ERROR: An illegal parameter passed.

PROT8583_REVERSAL_FILE_ERROR: An error occurred accessing the reversal file.

PROT8583_CREATED_REVERSAL: The request resulted in a reversal created.

PROT8583_CLR_KEY: Clear key pressed.

-17: Field not defined in the field table.

-18: Exceeded specified limit when parsing outgoing packet.

-19: Incoming packet exceeded specified limit.

-20: Incoming field size too large.

-21: No matching variant in variant table.

check_8583_tpdu_msg_id()

Checks the current TPDU address. Validates the message ID and TPDU in the response message against the TPDU and message ID from the request message. The return value is passed to the validation function for processing.

Prototype

```
int check_8583_tpdu_msg_id(COMM_8583_DATA *control, BYTE orig_tpdu, BYTE
                           *orig_msg_id);
```

Parameters

control	Protocol engine structure for this transaction.
orig_tpdu	Request TPDU.
orig_msg_id	Request message ID.

Return Values

Success:	1: TPDU and message ID match.
Failure:	Negative: message ID and TPDU do not match.
	PROT8583_RCV_TPDU_ERROR: TPDU failed.
	PROT8583_RCV_MSG_ID_ERROR: Message ID failed.
	PROT8583_BAD_TPDU_MSG_ID: Both the message ID and TPDU failed.

process_8583_request()

Sends and receives an 8583 request. Unpacks the response, validates the TPDU address, the message ID, and calls the users validation routine. A validation function must be provided by the user. This routine calls [process_8583_request\(\)](#) and if successful, calls [check_8583_tpdu_msg_id\(\)](#). The return value of a successful call is the return value of the user-defined validation function.

Prototype

```
int process_8583_request(COMM_8583_DATA *control_data, int mode, unsigned
                        current_timeout, BYTE *request_buf, unsigned
                        request_size, BYTE *tpdu_buf, BYTE *msg_id_buf);
```

Parameters

control_data	Protocol data structure for this transaction.
mode	Transmit, receive, or TX/RX mode of modem.
current_timeout	Passed to user TXRX8583 routine as a time-out.
request_buf	Buffer containing packet to send.
request_size	Size of packet to send.
tpdu_buf	TPDU address for this transaction.
msg_id_buf	Message ID for this transaction.

- Return Values**
- Success:

Value of validation routine.
- Failure:

PROT_8583_UNPACK_ERROR.
- Positive:

Communication error.

PIP Engine

Several transaction processors, most notably American Express, require certain applications to analyze information entered by the user (typically the account number) and route the transaction request to the appropriate host computer. The ability to route transactions to multiple hosts is referred to as PIP.

This section describes the key design criteria and issues that must be addressed to successfully implement the PIP Engine in an application. In addition to describing each of the components required to provide PIP capability, example modules are included to illustrate how the components interface.

A glossary of PIP and ISO 8583 terminology is included in [PIP/ISO 8583 Glossary](#).

NOTE

To ensure a successful PIP implementation, be familiar with the terminology, protocol, and processing requirements of each host the application must support.

This engine can only be used with the Verix eVo ACT, (VPN P006-212-02), for Verix eVo based terminals.

Compiling Source Code Modules

Most of the PIP Engine is provided in library form. Certain files, such as the communications example TXRX8583 are provided in source form. This example code can be incorporated into the application and modified as necessary to meet the applications communications requirements.

The source files must be compiled and linked with the application in the same manner as all other application files. These files become, in effect, another application file and should be processed in the same way.

Overview

The PIP Engine (and associated components) provides the ability to perform the following:

- ISO 8583 message assembly and disassembly
- ISO 8583 protocol processing (including reversal management)
- AMEX host protocol processing (including advice management)

NOTE

The AMEX host protocol processing component can be adapted for other hosts.

Each PIP host incorporates the ISO 8583 protocol from the application construction engine. This protocol was originally developed for communications between financial institutions and has been adapted for use in the POS environment. It is generally used in conjunction with SDLC communications links. ISO 8583 defines the request/response and error processing scenarios. Adherence to these rules ensures that independently developed applications can communicate with each other.

The main component of the ISO 8583 protocol is the message. Each message (either a request or response) is passed between the originator and the processor. The message is composed of a TPDU, message ID, bitmap, and a variable number of data fields. The key to the message is the bitmap. Each field that appears in a message is explicitly defined and assigned a number. When a field occurs in the message, the bit corresponding to the field is set in the bitmap. In this way, the number of fields in each message can vary. This allows the message to contain only those fields necessary for the type of request or response being processed.

The PIP Engine relies on various ISO 8583 engines and application-specific code to provide the functionality required to support PIP. By allowing specific processing to be completed by the application programmer, much more flexibility is gained in application design, communications device management, displays, and data storage.

Figure 3 illustrates how the PIP Engine interfaces to both the application program and various ISO 8583 engines.

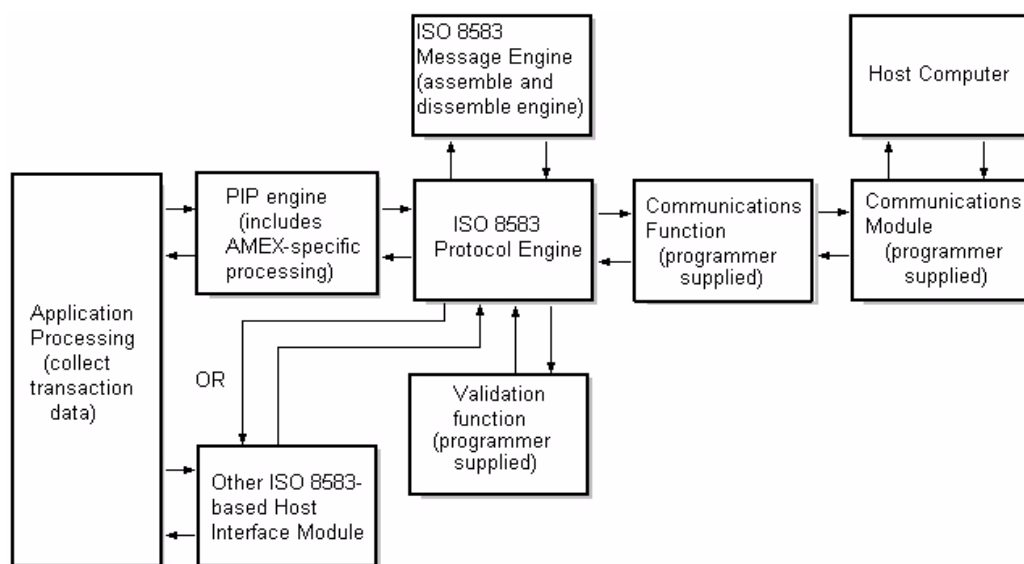


Figure 3 PIP Engine Interface

The PIP Engine interfaces to the application program and the ISO 8583 Protocol Engine. The ISO 8583 engines can be used to interface to other ISO 8583-based hosts. The ISO 8583 Protocol Engine interfaces to the ISO 8583 Message Engine, a validation function that validates the host response message, and communications functions that transmits and receive messages (both of these functions must be provided by the application). The communications function interfaces to a communications module that processes communications to and from the host computer. Although the ACL is not illustrated in the diagram, it is also used at a lower level by both the PIP and Protocol 8583 Engines.

Figure 3 does not illustrate the supporting data structures. These structures are crucial to the implementation of the PIP and ISO 8583 Engines. The application must declare and properly initialize these data structures. Many of the requirements of PIP are satisfied by simply completing the data declarations and initialization. However, other requirements must be considered when planning code requirements.

Several data structures and five functions must be completed to support the PIP Engine. This manual clearly defines the data structures and the purpose of their members. PIP does not dictate the form in which data must be managed. The application requirements, by engine, are listed in the following table.

ISO 8583 Message Engine	<ul style="list-style-type: none"> • Field Table • Convert Table • Variant field table(s) • Bitmap • <code>return_variant{}</code> • <code>return_variant2{}</code>
ISO 8583 Protocol Engine	<ul style="list-style-type: none"> • <code>COMM_8583_DATA</code> structure • Validation Function • Communications Function
PIP Engine	<ul style="list-style-type: none"> • <code>HOST_8583_DATA</code> structure • <code>set_trans_fields{}</code>

PIP Files

Some files are created and maintained by the PIP and ISO 8583 Protocol Engines. Each host that uses the ISO 8583 Protocol Engine and supports reversals requires a file to store the pending reversal. This file is managed by the ISO 8583 Protocol Engine and is deleted once the reversal is sent to the host.

The PIP Engine creates two files for every host that supports advice. Advice is a transaction sent to the host to inform it of some previous action or offline transaction. On occasion, AMEX uses a stand-in host. All transactions approved by the stand-in must be forwarded to the capture host in the form of an advice message. One file is an index file and the other file contains the actual advice transactions. These files are completely managed by the PIP Engine and *must* not be altered.

PIP Engine and Application Construction Toolkit

The PIP Engine is a modular component in a larger application. The interface requirements are defined and must be strictly adhered to. The PIP Engine also requires additional modular components in Verix eVo ACT.

PIP requires that application include the following components from Verix eVo ACT:

- ACL
- ISO 8583 Message Engine
- ISO 8583 Protocol Engine

There are several options available to complete interface requirements of the PIP Engine. Recommendations and examples are provided in this manual. Feel free to employ other techniques that are compatible with the overall application design, provided that the application meets the stated interface requirements.

Modular Design

A modular design approach enhances the flexibility of ISO 8583 programming and the PIP Engine. Many applications may require support for one or more hosts based on the ISO 8583 protocol. However, the requirements of these hosts may vary greatly.

ISO 8583 Protocol

Virtually any ISO 8583 host can be supported by making calls directly to the ISO 8583 Protocol Engine level rather than at the PIP level. In fact, the ISO 8583 Protocol Engine can be used in applications that do not require PIP-level processing.

Communications

The PIP Engine requires a user-defined communications routine as an external interface to the application used to process transmit and receive messages between the terminal and host. The ISO 8583 Protocol Engine makes no assumptions about the link to the host. This means that you can easily connect to devices other than the modem. This call is only required to send the request and obtain a response. This simple definition also means that other hosts, including async hosts, can be supported with the same communications routine (as long as calling and return requirements are met). For example, you may be connected to a PC for standard transaction processing using the COM1 port and to the modem port for dial backup processing.

ISO 8583 Message Engine

The ISO 8583 Message Engine supports the message processing requirements of the ISO 8583 Protocol. The functionality provided by this engine includes:

- Assembly and disassembly of packets based on the ISO 8583 standard
- Management of the bitmap that defines the valid fields in each packet
- Conversion of data packet fields in request and response messages

The ISO 8583 Message Engine uses a field table, a variant field table, and a converter table.

NOTE



Although the ISO 8583 Message Engine is being discussed in the context of PIP, it can be used in any application that is required to support the ISO 8583 standard. For more information about the ISO 8583 Message Engine, refer to ISO 8583 Message Interface Engine.

The Field Table

The field table is a master table that lists every data field that can be included in any request or response packet. This is necessary to provide the engine with the required information to access the application data elements, as well as complete the necessary data conversion.

Field Table Structure

The field table describes every field that can be included in any request or response packet. This table is used by the ISO 8583 Message Engine to assemble and disassemble packets. Each table entry contains the field number, the length of the field in the packet, an index into the conversion table for converting the packet fields to and from the application data types, a pointer to the application data variable, and the size of the application data element.

The table is an array of the following structure:

```
typedef struct
{
    int field_num;
    int packet_sz;
    int convert_idx;
    void *reference;
    int var_sz;
} field_struct;
```

NOTE



The field table structure is defined in the ISO8583.H file.

Parameters

<code>field_num</code>	<p>The field number corresponds to a bit position in the bitmap for the ISO 8583 packet. When a packet is assembled or disassembled, the bitmap indicates the next field to assemble or parse. The first ISO 8583 field is bit two, as bit one is used as a bitmap extension indicator.</p> <p>There must be a minimum of two elements in the field table array: the TPDU and the message ID. Each of these fields is expected to be packed BCD values and must be loaded into the appropriate application variables by the <code>set_trans_fields()</code> function. In addition to the field number, the following two special entries can be used in this field:</p> <ul style="list-style-type: none"> • SKIP indicates that the engine should skip over this field, because it is being sent by the host but is not used by the application. • STOP indicates that this is the last field in the table. <p>Note: The stop entry must be included in the table.</p>
<code>packet_sz</code>	<p>Specifies the length of the field (or the maximum length for variable length fields) in the ISO 8583 message. To determine the correct field length, refer to the host documentation.</p>
<code>convert_idx</code>	<p>Each field in an ISO 8583 message is required to be in a specific data format. The ISO 8583 Message Engine includes a variety of conversion functions that convert the data format defined for the field to the required ISO 8583 data format (for example, BCD to string). The convert index is an index into the convert table (see The Convert Table) that indicates what conversion function pair to use for the message field.</p> <p>In addition to the convert index, the following two special entries can be made in the convert index field:</p> <ul style="list-style-type: none"> • VARIANT indicates that the field varies depending on the message. In this case, a variant field table is used as the application variable. The ISO 8583 Message Engine accesses the variant field table to determine the conversion required for the field. • COMPUTE indicates that the contents of the field are calculated at the time the message is assembled/disassembled. The application variable field becomes a pointer to a function that performs the required calculation.
<code>reference</code>	<p>Contains a pointer (variable name) to the application variable, variant field table, or compute function required for the field. These variables are usually character arrays (for example, <code>char xxx[xx];</code>) when using the standard ISO 8583 conversion functions. Conversion functions (for example, long to BCD) must be written and included in the convert table when using a data format for variables other than a character array.</p>
<code>var_sz</code>	<p>Specifies the size of the application variable for this field. This performs limits checking when disassembling messages.</p> <p>An example of a completed field table follows. For more information about how to construct the field table, refer to ISO 8583 Message Interface Engine.</p>

Applications specifying a value for 8583 `sz` smaller than the actual data length to be converted may have been successful with earlier versions. Such applications will now fail, returning an error value (-4). Programmers should correct the value of 8583 `sz` to accurately reflect the maximum length of the data field.

Carefully inspect the 8583 `sz` field when using the following conversions: AV2_STR, BV2_STR, AV3_STR, HST_BI2, HST_BI3, STR_AV2, STR_BV2, STR_AV3, BI2_HST, BI3_HST, and AV3_AV3.

```
field_struct field_table [] =
{
    /* Field Packet Convert Variable Variable */
    /* # Size Index Name Size */
    { 0, 10, BCD_STR, (void *) tpdu, sizeof(tpdu) },
    { 0, 16, BIT_BIT, (void *) t_msg_id, sizeof(t_msg_id) },
    { 3, 24, BIT_BIT, (void *) t_proc_code, sizeof(t_proc_code) },
    { 4, 12, BCD_SNZ, (void *) t_amount, sizeof(t_amount) },
    { 5+ SKIP, 12, BCD_STR, (void *) discard, sizeof(discard) },
    {10+ SKIP, 8, BCD_STR, (void *) discard, sizeof(discard) },
    {11, 6, LNG_BCD, (void *)& t_sys_trace, sizeof(t_sys_trace) },
    {12, 6, BCD_STR, (void *) t_time, sizeof(t_time) },
    {13, 4, BCD_STR, (void *) t_date, sizeof(t_date) },
    {14, 4, BCD_STR, (void *) t_card. exp, sizeof(t_card. exp) },
    {15+ SKIP, 4, BCD_STR, (void *) discard, sizeof(discard) },
    {21+ SKIP, 4, BCD_STR, (void *) discard, sizeof(discard) },
    {22, 3, BCD_STR, (void *) t_posem, sizeof(t_posem) },
    {23+ SKIP, 3, BCD_STR, (void *) discard, sizeof(discard) },
    {24, 3, BCD_STR, (void *) t_nii, sizeof(t_nii) },
    {25, 2, BCD_STR, (void *) t_poscc, sizeof(t_poscc) },
    {26+ SKIP, 2, BCD_STR, (void *) discard, sizeof(discard) },
    {34+ SKIP, 28, AV2_STR, (void *) discard, sizeof(discard) },
    {35, 37, BV2_STR, (void *) t_card. track, sizeof(t_card. track) },
    {36+ SKIP, 104, AV2_STR, (void *) discard, sizeof(discard) },
    {37, 12, ASC_STR, (void *) t_key. rrn, sizeof(t_key. rrn)},
    {38, 6, ASC_STR, (void *) t_auth_code, sizeof(t_auth_code) },
    {39, 2, ASC_STR, (void *) t_resp_code, sizeof(t_resp_code) },
    {40+ SKIP, 3, ASC_STR, (void *) discard, sizeof(discard) },
    {41, 8, ASC_STR, (void *) t_host_data. tid, sizeof(t_host_data. tid)},
    {42, 15, ASC_STR, (void *) t_host_data. mid, sizeof(t_host_data.mid)},
    {43+ SKIP, 40, ASC_STR, (void *) discard, sizeof(discard) },
    {44, 2, AV2_STR, (void *) t_add_resp, sizeof (t_add_resp) },
    {45, 79, AV2_STR, (void *) t_card. track, sizeof(t_card. track) },
    {46+ SKIP, 999, AV3_STR, (void *) discard, sizeof(discard) },
    {53+ SKIP, 16, BCD_STR, (void *) discard, sizeof(discard) },
    {54, 12, AV3_STR, (void *) t_tip, sizeof(t_tip) },
    {55+ SKIP, 999, AV3_STR, (void *) discard, sizeof(discard) },
    {59+ SKIP, 999, AV3_STR, (void *) discard, sizeof(discard) },
}
```

```

{60, 0, VARIANT, (void *) variant_60, 0 },
{61, 8, AV3_STR, (void *) t_product_code, sizeof(t_product_code)},
{62, 10, AV3_STR, (void *) t_key. ticket, sizeof(t_key. ticket) },
{63, 0, VARIANT, (void *) variant_63, 0 },
{64+ SKIP+ STOP, 64, BIT_BIT, (void *) discard, sizeof(discard) },
};

```

The first two elements, the TPDU and message ID, have special meaning. These elements must always be present in the field table. Each of these elements has a field number of 0, which means they do not have bitmap assignments. More information about each of these special fields follows.

The TPDU

The Transport Protocol Data Unit (TPDU) routes the SDLC message. It consists of a transaction type identifier, an originator address, and a destination address. For AMEX transactions, the transaction type identifier is either 0x60 for application messages or 0x68 for NMS/TNMS transactions. The typical application only uses 0x60.

NOTE



Refer to [Typical Host/Terminal Packet Structure](#) for a diagram of each of the host/terminal packet formats.

The originator and destination addresses are each two bytes. The TPDU should be provided by the host. The TPDU is the only SDLC component that must be provided (other than opening the port in SDLC mode).

NOTE



In a request message, the destination address occurs before the originator address. In a response message, the originator address occurs before the destination address. This distinction is important.

The Message ID

The message ID indicates the type of message and message function. The message ID is an ISO 8583 requirement, not an SDLC requirement. The message ID is four digits and is compressed to two bytes when placed in an ISO 8583 packet. A description of each of these digits is in the following table.

Position	Description
First Digit	The version number (always 0).
Second Digit	<p>Message class. This variable depends on the requested transaction. The following list identifies valid values for the message class. A host may not use every message class.</p> <ul style="list-style-type: none"> • 1 = Authorization • 2 = Financial capture • 3 = File update • 4 = Reversals • 5 = Reconciliation • 8 = Maintenance
Third Digit	<p>Message function. This identifies a particular operation for a message type (as in a request for authorization or a financial capture response). Valid values are:</p> <ul style="list-style-type: none"> • 0 = Request • 1 = Response to a request • 2 = Advice • 3 = Response to an advice • 4 = Notification • 5–9 = These codes are reserved for ISO use
Fourth Digit	Transaction originator. This position is always 0 for requests and responses.

An example of a typical message ID is a 0200 request message (financial capture request). This message is sent to a host to request authorization for a transaction that, if approved, is electronically captured and settled. The host responds with a 0210 response message (financial capture response).

For more information about the message ID fields, refer to the *AMEX 3000 PIP Terminal Technical Specifications* document.

The Convert Table

The convert table contains pairs of pointers to functions that allow the ISO 8583 Message Engine to convert application data to packet data and vice versa. The PIP Engine includes a standard set of conversion routines and a standard convert table, which provide all the conversions needed for processing AMEX transactions.

NOTE



The ISO 8583 Message Engine Reference section contains instructions for adding additional conversions to support other host packet requirements. The function references for each of the conversion functions are located in the [ISO 8583 Message Interface Engine Function Calls](#) section.

The convert table must contain all the pairs are used by the application to convert the data. Do not change the name of this table, `convert_table`. Entries may be added to the end of this table. However, do not modify existing entries

```
converters convert_table [] =
{
    {asc_to_asc, asc_to_asc}, /* ASC_ASC 0 */
    {av3_to_av3, av3_to_av3}, /* AV3_AV3 1 */
    {bit_to_bit, bit_to_bit}, /* BIT_BIT 2 */
    {bcd_to_bcd, bcd_to_bcd}, /* BCD_BCD 3 */
    {asc_to_bcd, bcd_to_asc}, /* BCD_ASC 4 */
    {str_to_asc, asc_to_str}, /* ASC_STR 5 */
    {str_to_bcd, bcd_to_str}, /* BCD_STR 6 */
    {str_to_bcd, bcd_to_snz}, /* BCD_SNZ 7 */
    {str_to_av2, av2_to_str}, /* AV2_STR 8 */
    {str_to_bv2, bv2_to_str}, /* BV2_STR 9 */
    {str_to_av3, av3_to_str}, /* AV3_STR 10 */
    {str_to_xbc, xbc_to_str}, /* XBC_STR 11 */
    {hst_to_bin, bin_to_hst}, /* BIN_HST 12 */
    {hst_to_bi2, bi2_to_hst}, /* BI2_HST 13 */
    {hst_to_bi3, bi3_to_hst}, /* BI3_HST 14 */
};
```

Variant Fields

Variant fields depend on the type of message being sent and the function of that message. An example of a variant field might be the private fields 60 to 63. These fields can be any length up to 999 bytes, and can be used for any purpose agreed upon by the host and requesters.

The field table allows a special entry, `VARIANT`, in the convert index to indicate that a variant field table exists to further define this field. When the ISO 8583 Message Engine encounters this entry, it selects the indicated field table and continues to process the field entry for the message.

NOTE



To use variant fields, you must create two functions: `unsigned int return_variant1()` and `unsigned int return_variant2()`. For more information on how to write these functions, refer to [PIP Engine and Application Construction Toolkit](#).

Computed Fields

In some cases, a field may need to be computed. This computation may depend on other fields in the packet, other application variables, the entire packet (such as, a checksum), or other types of calculations.

The field table allows a special entry, `COMPUTE`, in the convert index to indicate that a computed field table exists to further define this field. When the ISO 8583 Message Engine encounters this entry, it calls the function pointed to in the field table entry and accepts the result as the contents of the packet field.

NOTE



Computed fields are not currently required to interface with the AMEX host. This information is included in the event an alternate host requires this type of field.

ISO 8583 Protocol Engine

In addition to assembling/disassembling data packets (as described in the ISO 8583 Message Engine section), the application must support the ISO 8583 protocol. The ISO 8583 Protocol Engine was developed to provide this support. This engine includes:

- an interface to the ISO 8583 Message Engine to assemble and disassemble packets,
- an interface to the programmer-defined validation function that determines if an appropriate response was received,
- an interface to the programmer-defined communications function that transmits and receives request/response messages, and
- processes reversals as specified in the ISO 8583 Messaging standard.

PIP Engine has the following power-fail recovery mechanism:

- Determines if reversals are allowed, and that a transaction request is to be generated. Create a contingent reversal file for that transaction request just prior to the internal call to the programmer-supplied communication and validation functions.
- On successful return of the programmer-supplied communication and validation functions, or on return of any error not considered a reversal condition, delete this contingent reversal file.

If a power failure occurs during this period of vulnerability, an internal routine checks for a reversal file at the beginning of the next transaction request session. If this file exists, it sends a reversal request as appropriate. No intervention by the application is necessary.

Protocol 8583 Data Structure

The ISO 8583 Protocol Engine uses the following data structure (defined in PROT8583.H) to provide information and application data variables.

```
typedef struct c8583dat
{
    field_struct *p_fld_table;
    int *comm_handle;
    unsigned comm_timeout;
    unsigned wait_for_carrier;
    unsigned char *map;
    int (* validation)();
    void *v_parms;
    unsigned v_parms_size;
    int (* transceiver)();
    unsigned char *transmit;
    unsigned int transmit_size;
    unsigned transmit_limit;
    unsigned char *receive;
    unsigned receive_size;
    unsigned receive_limit;
    unsigned char *reversal;
    unsigned reversal_size;
    unsigned reversal_limit;
    unsigned char *base_Purposefname;
    unsigned char rev_msg_id [5];
    int state;
} COMM_8583_DATA;
```

Programmer-Defined Setup Function

To use the PIP Engine, write a function called `set_trans_fields()`. This function is called at the beginning of the PIP Engine and must set up the following information:

- the processing code,
- the message ID,
- the TPDU, and
- the bitmap.

This information is used to build the request packet. In addition, this information must be preserved as a structure to ensure that it is not overwritten by the data included in the host response.

NOTE



For more information on how to write this function, refer to [Integrating the PIP Engine Into Your Application](#).

Integrating the PIP Engine Into Your Application

Integrating the PIP Engine and its associated modules into the application is a six-step process:

- 1 Plan and define all required tables and data structures.
- 2 Write and test the `return_variant1()` and `return_variant2()` functions.
- 3 Write the `set_trans_fields()` function.
- 4 Write a communications (transceiver) function.
- 5 Write a validation function.
- 6 Make the appropriate calls to `pip_trans()` from the application.

This process must be clearly understood before beginning the integration.

Step 1 - Plan and define all required tables and data structures

Declare the data elements required by each of the following engines:

ISO 8583 Message Engine	Field table
	Variant field tables (if required)
	Convert table
ISO 8583 Protocol Engine	COMM_8583_DATA structure
	Validation structure
PIP Engine	HOST_8583_DATA structure

Declarations and examples can be found with the engine files (when appropriate).

Application Variable

In addition to declaring the tables and data structures listed above, also declare a variable named `gu_clr_state` (`unsigned int`) and initialize it to 0. This variable can be used to support clear key processing in the `pip_trans()` and `prot_8583()` functions. Each of these functions aborts processing and returns to the caller if this variable is set to any value other than 0 (zero). This is a convenient way to interrupt the function processing when the clear key is pressed.

Step 2 - Develop the `return_variant1` and `return_variant2` functions

Variant field tables use two additional pieces of information to determine the description of the field. The functions `return_variant1()` and `return_variant2()` must return values that are compared to entries in the variant field tables to indicate if the current entry in the field table is to be used, or if the message engine should continue to process the variant field table. The ISO 8583 Message Engine provides functions for `return_variant1()` and `return_variant2()` used to access the variant field tables constructed for the AMEX host. These functions are generally useful in processing any ISO 8583-based host.

When processing the field tables, the ISO 8583 Message Engine calls `return_variant1()` and `return_variant2()`, then compares these values to the values in the first two positions of each array element of the variant field table. The convert index, variable name, and variable size fields have the same function as their counterparts in the field table.

The following examples show the `return_variant1()` and `return_variant2()` functions and two sample variant field tables for fields 60 and 63. In these examples, `return_variant1()` returns the first two digits of the processing code and `return_variant2()` returns the message ID (each converts the data to integer form).

```
/* If variant fields are used, this integer valued function must return a
value which will be used to match against the values in the first field of
variant field tables. */
```

```
unsigned int return_variant1()
{
    unsigned char x [3];
    MEMCLR (x, 3);
    /* Most significant 2 digits of proc_code. */
    SVC_UNPK4 (x, t_proc_code, 1);
    return (str2int (x));
}
```

```
/* If variant fields are used, this integer valued function must return a
value which will be used to match against the values in the second field
of variant field tables. */
```

```
unsigned int return_variant2()
{
    unsigned char x [5];
    MEMCLR (x, 5);
    SVC_UNPK4 (x, t_msg_id, 2);
    return (str2int (x));
}
```

```
/* Type definition for an individual member of a variant field table
array. */
```

```
typedef struct
{
    unsigned int variant1; /* Target. */
    unsigned int variant2; /* Target. */
    int packet_sz; /* Packet size. */
    int convert_idx; /* Index into convert table. */
    void *reference; /* Pointer to variable or table. */
    int var_sz; /* Variable size. */
} variant_struct;
```

```
variant_struct variant_60[] =
```

```

{
    /* Vari- Vari- Pack Convert Variable Variable ant 1 ant 2 Size Index
    Name Size */
    { 00, 320, 37, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 02, 320, 37, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 22, 320, 37, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 09, 320, 37, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 29, 320, 37, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 92, 500, 6, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 96, 500, 6, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 97, 500, 6, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 92, 510, 12, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 96, 510, 12, AV3_STR, (void *) or_msg, sizeof (or_msg)},
    { 00, 420, 22, AV3_STR, (void *) ah_dat.dcp_orig_msg, sizeof
      (ah_dat.dcp_orig_msg)},
    { 02, 420, 22, AV3_STR, (void *) ah_dat.dcp_orig_msg, sizeof
      (ah_dat.dcp_orig_msg)},
    { 22+ STOP, 420, 22, AV3_STR, (void *) ah_dat.dcp_orig_msg, sizeof
      (ah_dat.dcp_orig_msg)},
};

/* Variant structure for field 63.
* 0500 = Reconciliation
* 0320 = Batch upload - sale, force, credit, void (debit and credit )
* 0200 = Sale
*/
variant_struct variant_63 [] =
{
    /* Vari- Vari- Pack Convert Variable Variable ant 1 ant 2 SizeIndex
    Name Size */
    { 00, 200, 112, BIT_BIT, (void *) bat_tot, sizeof (bat_tot)},
    { 00, 210, 40, AV3_STR, (void *) bat_tot, sizeof (bat_tot)},
    { 00, 110, 40, AV3_STR, (void *) bat_tot, sizeof (bat_tot)},
    { 02, 320, 112, BIT_BIT, (void *) bat_tot, sizeof (bat_tot)},
    { 22, 320, 112, BIT_BIT, (void *) bat_tot, sizeof (bat_tot)},
    { 22, 330, 40, AV3_STR, (void *) bat_tot, sizeof (bat_tot)},
    { 20, 330, 40, AV3_STR, (void *) bat_tot, sizeof (bat_tot)},
    { 97, 500, 36, AV3_STR, (void *) bat_tot, sizeof (bat_tot)},
    { 93, 510, 42, ASC_ASC, (void *) bat_tot, sizeof (bat_tot)},
    { 94+ STOP, 510, 42, ASC_ASC, (void *) bat_tot, sizeof (bat_tot)},
};

```

Step 3 - Write the set_trans_fields function

Write a `set_trans_fields()` function (an example is provided with the PIP Engine files). When this function call ends, the following must be complete:

- the system trace number is properly set (increment as needed),
- the processing code is loaded,

- the message ID is loaded,
- the TPDU is loaded,
- the bitmap is loaded, and
- the validation structure is initialized.

The following is an example of a `set_trans_fields()` function. It is not necessary to follow this example exactly. In fact, it is necessary to alter the processing of this function depending on the way handle data storage is handled.

```
/* This function uses a file to store the bitmap, message ID, & processing
code. */
int set_trans_fields(host, transaction_type);
int host;
int transaction_type;
{
int f_handle; /* Handle used to access the file. */
extern BYTE source_map[8];
/* Variable used to contain the mandatory bit fields for the transaction.
*/
/* Open b- m- p file, return if an error occurs. */
if(- 1 == (f_handle = open(" bmp. lod", O_RDONLY)))
    return(f_handle);

/* This function reads the bitmap, message ID, and processing code into
the indicated variables. These are the global values whose addresses
appear in the field table. */
load_bitmap(f_handle, transaction_type, source_map, t_msg_id, t_proc_code);
close(f_handle);
/* Load the TPDU. */
strcpy(tpdu, " 6001000000");

/* Increment the systems trace number. This is maintained as a LONG value
and is converted when placed in the ISO 8583 packet. The conversions
bcd2lng() and lng2bcd() are used for this purpose. */

t_sys_trace++;

/* The validation structure for this application contains the address of
the systems trace number and the original value of the trace number. This
makes validation easier. */
input_val_struct. p_trace = &t_sys_trace;
input_val_struct. b_trace = t_sys_trace;
```

```

/* Some transactions, such as adjustments, require the RRN of the original
transaction be sent as a field in the request. This application saves the
value of the request RRN since the value in the field table will be
overwritten when the response is processed. */

strcpy(input_val_struct., t_key. rrn);

/* The terminal ID is a validation parameter. The original terminal ID
must be compared to the terminal ID in the response message. Since the
original will be lost when the response is unpacked, this step saves the
original value for comparison. The pointer is used to conveniently access
the message ID during validation. */

input_val_struct. p_tid = (unsigned char *) t_host_data. tid;

strcpy(input_val_struct. b_tid, t_host_data. tid);

/* The requirements for the above message ID apply to the amount which
means the amount must be processed in a similar manner. */

input_val_struct. p_amount = (unsigned char *)
t_amount;

strcpy(input_val_struct. b_amount, t_amount);

/* set_trans_fields should return 1 if it is successful. Any negative
value will abort. */

return(1);
} /* end set_trans_fields() */

```

Step 4 - Write the communications function

The user-defined communications function (also called the transceiver function) ensures that no restrictions are placed on the way communications are accomplished. The ISO 8583 Protocol Engine makes no assumptions about how the application communicates and permits processing through a serial port, modem, and so on.

The calling syntax and return values are defined (refer to the following table). This interface definition allows the ISO 8583 Protocol Engine to correctly call the communications routine, as well as return meaningful results on the success or failure of the communications process.

Use `#defines` (listed in `PROT8583.H`) when writing the communications function. This ensures that the proper values are used, makes the source code more readable, and is easier to remember.

The function can have any name. The name of the function is placed in the transceiver member of the `COMM_8583_DATA` structure. The function can also be called directly by the application to complete other communication requirements. The example provided is named `mdm_tr_8583()`.

The following syntax is used when calling the communications function:

```
int mdm_tr_8583 (int h_device, int mode, unsigned char *req_buf, unsigned
req_size, unsigned char *rcv_buf, unsigned rcv_limit, unsigned timeout,
unsigned carrier_timeout);
```

The `mode` parameter must be either TXRX or RX_ONLY (defined in PROT8583.H). When `mode` is TXRX, the communications function sends the data in the `req_buf` and waits for a response. RX_ONLY is used to request that the communications function wait for an additional response message.

The AMEX host may send a “Please Wait” message in response to a request. When this happens, the validation routine should request an additional response packet with a 60 second time-out. Additionally, any message that fails validation results in a request for another request packet.

The carrier time out may not be required, depending on the method used by the application to establish the link to the host. It is highly recommended that the application perform a pre-dial to initiate the connection with the host as early as possible in the transaction. The following table lists the required return values for the transceiver function.

Success: TR_SUCCESS (1): Request successful.

Failure: TR_BAD_PARM (-33): Incorrect parameter in call.

TR_TO_CARRIER (-34): Time out waiting for carrier (no reversal).

TR_WR_FAIL (-35): Write to port failed (no reversal).

TR_TO_RESPONSE (-36): Time out waiting for response (create reversal).

TR_RD_FAIL (-37): Read port failed (create reversal).

TR_LOST_CARRIER (-38): Connection to host lost after request was sent (create reversal).

TR_CLR_KEY (-47): Clear key press (no reversal). This value should not be returned once the request has been sent.

```
int mdm_tr_8583 (int h_device, int mode, unsigned char *req_buf, unsigned
req_size, unsigned char *rcv_buf, unsigned rcv_limit, unsigned timeout,
unsigned carrier_timeout);
```

```
{
int ret_val = TR_SUCCESS;
unsigned carrier_delay;
unsigned long receive_delay;
char mdm_status [5];
/* Carrier delay is in seconds == 50 ms * 20 * delay. */
carrier_delay = carrier_timeout * 20;
/* Wait for carrier to be available only if the mode is TXRX. */
while ((0 == (ret_val = mdm_check_status (h_device, MDM_DCD))) && TXRX ==
mode)
{
```



```

/* Do carrier delay in 50- ms increments. */
if (gu_clr_state)
{
    ret_val = TR_CLR_KEY;
    break;
}
else if (carrier_delay-- > 0)
{
    SVC_WAIT (50);
}
else
{
    /* Timed out waiting for carrier, set * ret_val and exit. */
    ret_val = TR_TO_CARRIER;
    break;
}
} /* End while wait clr or carrier. */
/* Continue if not timed out for carrier and [CLEAR] key has not been *
pressed. */

if (0 < ret_val)
{
    /* If mode 0, then send request. */
    if (TXRX == mode)
    {
        /* Send the request message. */
        if (req_size != (ret_val = write (h_device, req_buf, req_size)))
        {
            /* Write fail to device, either port error, no buffers, or * not
            enough characters written */
            errno = ret_val;
            ret_val = TR_WR_FAIL;
        }
    }
}
/* End transmit. */

/* Begin receive processing. */
if (0 < ret_val )
{
    /* Set wait for response timeout.*/
    receive_delay = set_itimeout (gh_clock, timeout, TM_SECONDS);
    while (0 >= mdm_input_pending (h_device))
    {
        if (0 == mdm_check_status (h_device, MDM_DCD))
        {

```

```

        ret_val = TR_LOST_CARRIER; break;
    }
    /* Wait for timeout or received message packet */
    if (TM_EXPIRED==CHK_TIMEOUT (gh_clock, receive_delay))
    {
        ret_val = TR_TO_RESPONSE;
        break;
    }
} /* End while. */

/* If ret_val is still positive, we have a packet to read. */
if (0 < ret_val)
{
    /* Inbound packet ready, set ret_val to error if read fails. */
    if (0 >= (ret_val = read (h_device, rcv_buf, rcv_limit)))
    {
        errno = ret_val; ret_val = TR_RD_FAIL;
    }
}
/* End receive. */
}
/* End wait for response. */
}
/* End if carrier and not [CLEAR] key. */
/* If ret_val is still positive, we succeeded. If not, one of * the steps
failed. The first error ends the process. */
return(ret_val);
} /* End mdm_tr_8583().*/

```

Step 5 - Write the validation function

Validation refers to verifying that the response received for a request is appropriate. This, of course, varies from one host to host. The ISO 8583 Protocol Engine always validates the TPDU and message ID, and passes the result of the validation to the validation routine of the application for further processing. The ISO 8583 Protocol Engine will not abort the transaction, even if the TPDU and message ID are both invalid. The engine relies on the validation of the application to determine if the response is to be accepted or rejected.

The results of the validation are application dependent. Several options are possible. The value returned by the validation routine dictates the action to take. A positive return value indicates that the ISO 8583 Protocol Engine should call the communications routine to wait for another response packet using the return value as the time out in seconds. The validation routine may also return the `#define RETRY_CONTROL_TIMEOUT (-15)` to request another response message using the time out in the ISO 8583 Protocol Engine data structure. The validation routine may also return `CREATE_REVERSAL`.

These values are defined in `PROT8583.H`. Returning `CREATE_REVERSAL` instructs the ISO 8583 Protocol Engine to create a reversal record. This record is sent to the host (as a separate transaction) preceding the next transaction. The reversal message informs the host to eliminate the transaction identified (as though the transaction did not occur). When validating AMEX responses, any invalid response should cause the application to continue to wait for an additional response.

AMEX Validation Functions

The ISO 8583 Protocol Engine provides two example validation functions that properly validate AMEX host transactions: `amex_validation_low()` and `amex_validation_high()`.

Since these functions are provided on the distribution diskette in source form, they must be modified to support additional hosts. The functions can be used as is for AMEX transaction processing. If the application is interfacing to other hosts, write validation function(s) for each host.

`amex_validation_low()` Function

AMEX can respond with one of two special messages: The “Please Wait” message means that the host is processing the request and the terminal should allow another time period (usually 60 seconds) for the response. The “Hang Up” message means that there is a problem processing the request (the host is not available) and the terminal should abort the transaction request.

`amex_validation_low()` is responsible for detecting each of these response messages. This function, called from `amex_validation_high()`, either processes one of these two messages as indicated or continues to validate the response.

`amex_validation_high()` Function

The second validation routine is `amex_validation_high()`. This function is used as the validation routine in the ISO 8583 Protocol Engine data structure. This function calls the `amex_validation_low()` function. If the response is “Please Wait” or “Hang UP”, it is processed accordingly. Validation continues for all other responses.

The Validation Process

The next step in the validation process is to determine the proper validation parameters. Reversals are processed prior to actual transactions. When a reversal exists, two transactions are pending when the validation routine is called. The validation routine tests the state in the `COMM_8583_DATA` structure to determine if the validation is for a transaction request or a reversal. If the request is for a reversal, the reversal parameters are copied from the beginning of the reversal buffer to the local validation structure; otherwise, the transaction validation parameters are used.

The transaction amount, systems trace number, and terminal ID are now compared. The transmitted and received values must match. The `amex_validation_high()` routine pads the amounts to a minimum of three characters to ensure proper comparison. The validation routine expects the systems trace number to be a global long, and it expects the terminal ID and amounts to be strings. If other data types are used, change the comparisons in `amex_validation_high()` accordingly.

Another special case occurs if a DCP response is received for a reversal request. The PIP Engine must create an advice message that is forwarded to the AMEX host at a later time. The `amex_validation_high()` function tests for this by comparing the message ID to 0400, the reversal message ID. If the response is for a reversal, then the value of field 44 is tested. If this field is 02, then a DCP response was received for the reversal and the validation routine returns `AMEX_DCP_REVERSAL`. Upon receiving this return code, the advice is created.

The validation routine returns a 0 if the response is valid for the request. This is accepted by the ISO 8583 Protocol Engine, and the value either returns to the caller, or processes the transaction request. This scenario varies with previous transaction activity.

The ISO 8583 Protocol Engine includes a validation structure used with the provided validation routines. Again, this structure may be modified as needed. You are responsible for ensuring that the application meets AMEX validation criteria.

The validation structure is listed below, followed by an explanation of its members.

```
typedef struct
{
    unsigned char b_amount[13];
    unsigned char b_tid[9];
    long int b_trace;
    unsigned char b_rrn[13];
    unsigned char *p_amount;
    unsigned char *p_tid;
    long int *p_trace;
    unsigned char *curr_key;
    COMM_8583_DATA *comm_struct;
} VALIDATION_STRUCT;
```

```
unsigned char b_amount[13];
unsigned char b_tid[9];
long int b_trace;
unsigned char[13];
```

These parameters are loaded with the data for the current transaction. This amounts to a backup of the transaction data, since an inbound message must be disassembled and the original transaction data is overwritten.

```
unsigned char *p_amount;
unsigned char *p_tid;
long int *p_trace;
```

The pointers in the validation structure are there to simplify the references to the data in the field table. The same addresses could be accessed using the pointer to the field table in the COMM_8583_DATA structure.

```
unsigned char *curr_key;
```

Pointer to the application variable containing the key used for the advice file. This should be the ROC for AMEX host transactions.

```
COMM_8583_DATA *comm_struct;
```

Pointer to simplify referencing the COMM_8583_DATA structure for the validation routine.

**Step 6 - Write the
application
functions to call the
PIP Engine**

After completing the required table and functions described in steps one through five, integrate the PIP Engine into the application. Refer to the \TONLINE and \TPRO8583 subdirectories on the installation diskettes for examples of how to call the ISO 8583 and PIP functions.

PIP Functions

The PIP functions are described in [PIP Engine Function Calls](#).

Troubleshooting PIP Application Modules

This section presents problem scenarios and possible solutions.

Problem	The terminal sends a request, a response is received, but the terminal appears to be stuck during receive. The terminal is still connected to the host.
Solution	<ul style="list-style-type: none"> The response packet failed the validation routine. Ensure that the validation structure (see page 100 for example) is loaded with the correct values. The returned response amount is padded with leading zeroes. Ensure that these characters are not included in the validation comparison. The terminal may have received a 0820 message ("Please Wait"). The terminal should wait for the next response packet.
Problem	The request packet is not transmitted and the PIP module returns an error.
Solution	<ul style="list-style-type: none"> The packet being built exceeds the size of the buffer. Increase the size of the transmit buffer. A variant field table is not set up correctly. The packet assembler could not find a matching message ID/ processing code pair in one of the variant tables. Check the variable <code>fn_8583</code> to determine which field failed.
Problem	The terminal receives a valid response packet, but the PIP module returns an error.
Solution	<ul style="list-style-type: none"> The transaction may have been denied by the host. Check the request packet for errors. A destination field is too small to hold the response data. Check the <code>fn_8583</code> variable to determine which field failed. Increase the size of this field. The variant field table is not correctly set up. The packet disassembler may not have found a matching message ID/ processing code pair in one of the variant tables. Check the variable <code>fn_8583</code> to determine which field failed.
Problem	The request packet has missing or extra fields.
Solution	<ul style="list-style-type: none"> Additional or missing fields in the response packet can usually be traced to the bitmap used to build the packet. Extra fields are caused by the setting (on) of extra bits, while missing fields are caused by bits not being turned on. Other causes of missing fields are missing application variable data or an incorrect field table conversion.
Problem	Advice , reversals, TPDUs, and/or bitmap overwritten by the RRN field.
Solutions	<ul style="list-style-type: none"> Set <code>v_parm_size</code> in the comm structure to include <code>b_rrn[]</code> in the validation structure.
Problem	The RRN field is not being set on advice and/or reversals.
Solution	<ul style="list-style-type: none"> The application must set the bitmap of the RRN field even though RRN does not exist.

PIP/ISO 8583 Glossary

Table 5 PIP Term Definitions

Term	Definition
Advice	A transaction sent to the host to inform it of some previous action or offline transaction. On occasion, AMEX uses a stand-in host. All transactions approved by the stand-in must be forwarded to the capture host in the form of an advice message.
Bitmap	64 bits numbered from 1 to 64, left to right. One or more bitmap(s) are required before each ISO 8583 message to indicate the individual fields that occur in the data packet. Bit one is an extension bit. If this bit is set, another 64 bitmaps follows. Bit 64 is reserved as a Message Authenticator Code (MAC). The MAC field is not used for AMEX transactions.
Closed Batch	A batch that is no longer used to store new transactions, but still contains details of previous transactions. For AMEX batches, a batch is closed when it is settled with the host. Closed AMEX batches are deleted and are no longer available.
Conversion Index	A reference to an entry in the <code>convert_table[]</code> . Each field required for the ISO 8583 Message Engine requires a conversion routine to convert the data from the application data type format to that used in the ISO 8583 message, and vice versa. The conversion index is entered into the field table and the <code>process_8583()</code> call uses this index to access the proper conversion routine for each field as required.
Conversion Table	An array containing pointers to routines that convert application data to the appropriate ISO 8583 fields. This table is required when using the ISO 8583 Message Engine.
DCP	Distributed Credit Authorization System (CAS) Processor. This is a "stand-in" host used by AMEX when the primary host is unavailable.
Descriptor Code	A set of two-digit numbers assigned to a terminal to describe the purchases for transactions.
Field Table	An array containing information for every field that is either to be assembled or disassembled during communications with an ISO 8583 host. This field is required by the ISO 8583 Message Engine. The first two array elements are always reserved for the TPDU and message ID. The array elements indicate the field number, packet field size, conversion table index, pointer to the application variable or variant field table, and the size of the application variable.
Message Definition Table	A table illustrating the various fields used when communicating with an ISO 8583 host. This table indicates the required and optional fields, the format for the data, and the length of the data field. This table generally includes a description of the use of each of the fields.
Offline Transaction	A transaction completed without communication with the host. A variety of transactions can be completed offline, such as a transaction that is less than the designated minimum transaction amount (floor limit).

Table 5 **PIP Term Definitions** (continued)

Term	Definition
On line Transaction	A transaction completed by sending the details of the transaction to the host and waiting for a host response to determine if the transaction is approved or denied.
Open Batch	The current batch file used to store the details of new transactions.
Reversal Transaction	A transaction sent to the host to eliminate or “reverse” the last sent transaction. This is usually to correct a problem that occurred during the transaction such as a time out or loss of connection.
ROC	Record of charge. A physical record of a transaction or receipt. A ROC can be printed from a terminal with a printer either external or internal. The ROC is usually signed by the cardholder.
ROC Number	A number assigned to a transaction and printed on a receipt for the cardholder to sign. This number can be preprinted at the bottom of some AMEX credit or credit forms.
RRN	Retrieval Reference Number (field 37). A 12-character unique reference number assigned to a transaction by the host. This number references a transaction when further processing is needed. This field is returned from the host in the response message.



Data Capture Engine

The Data Capture Engine provides programmers working in the Verix eVo operating environment an easy way to manage the transfer of application data to and from keyed data files. By using the Data Capture Engine, the programmer eliminates the need to design, code, and test the functionality provided by this engine.

Using a standard engine also provides the benefit of application maintenance. Programming with the Data Capture Engine requires less time to implement enhancements and make other required changes to the application.

The Data Capture Engine relies on the functionality provided by the Verix eVo standard C Library and the ACL. By making use of these two library sources, further leverage is gained by reusing functions, thereby further reducing test efforts and application size.

Keyed File System

The design of the Data Capture Engine is based on the use of the keyed file system provided by Verix eVo. Keyed files contain records identified by a programmer-specified name or key. This key writes and reads data to and from the file allowing random access to the individual data records. The Verix eVo operating system uses compressed variable length records (CVLR) for keyed files. This introduces two limitations:

- Keyed files employ a data compression mechanism. This limits the type of data that can be stored in the file. The compression technique converts any lowercase alphabetic character to its uppercase equivalent, and packs ASCII digits “0” through “9” into four bits. This limits the data being stored to bytes of 0x5F or less. Bytes greater than 0x5F return a different character when read from a keyed file.
- The second limitation pertains to the length of the record. The data portion of each record contains a byte indicating the length of the data as its first byte. This length includes the length byte. This limits the number of characters that can be written to a maximum of 254 characters (254 data characters + length byte = 255); however, these are compressed bytes. Depending on the number of consecutive numeric character pairs in the data portion of the record, up to 508 characters can be stored (if all characters are numeric).

Application Data

Keyed files can be opened by the application prior to their use, but this is not required. The file must exist, however. If a specified file is not open at the time access is required, the file is opened, the requested operation completed, and the file is closed. There must be a file handle available for the operation to proceed. If the file does not exist or a file handle is not available, an error value returns.

Most application data is integer, long integer, character, or string data types. This data, residing in RAM, must be stored and retrieved from data files to ensure that it is not lost in the event of a power failure or similar event. The access of this data is also random in nature.

The keyed file mechanism provides random access, but the data must be converted to a compatible format (ASCII) for storage and converted again when retrieved. These required conversions are accomplished by the Data Capture Engine.

Data Capture Functions

The data capture functions are described in [Data Capture Engine Function Calls](#).

Example Program

A data capture example program can be found in the [Data Capture Engine Example Program](#) section.



Modem Engine

The Application Construction Modem Engine provides programmers an easy way to manage the modem device. By using the Modem Engine, the need to design, code, and test the functionality provided by this engine is eliminated.

The Modem Engine relies on the functionality provided by the Standard C Library and the ACL. By making use of these two library sources, further leverage is gained by reusing functions, thereby further reducing testing efforts and application size.

Modem Engine Functions and Macros

The Modem Engine is made up of 27 functions and five macros. These functions and macros are general-purpose routines that ease the use and management of the modem device. Function prototypes, macro definitions, structure declarations, and `#defines` for many return and error codes are contained in the `XMODEM.H` header file (see [XMODEM.H Listings](#)).

Macros

The Modem Engine comprises the following macros:

- `MDM_READ_BLK()`: Obtains the communications parameters.
- `MDM_RESET_ERROR()`: Resets latched error conditions.
- `MDM_STATUS()`: Obtains the status of the modem.
- `MDM_WRITE_BLK()`: Sets the communications parameters.

MDM_READ_BLK()

Reads opened modem device communication parameters into a Modem Opn_Blк structure. This function can save the current modem settings to allow single parameter changes or to restore the modem state at a later time.

Prototype

```
#include <XMODEM.H>
int MDM_READ_BLK(int h_modem, struct Opn_Blк mdm_blk);
```

Parameters

h_modem	Opened modem device handle.
Opn_Blк mdm_blk	Communication parameters.

Return Values

Success: SUCCESS: No errors occurred.
Failure: FAILURE: The error type can be determined by checking `errno`.

NOTE



Opn_Blк is a structure of type `struct Opn_Blк` containing communications parameters (refer to the target terminals programmers manual for Opn_Blк structure definition).

Example

The linked example code file demonstrates use of `MDM_READ_BLK ()`.

MDM_RESET_ERROR()

Resets the latched error conditions of parity, framing, and data overrun. These conditions remain active until the modem device is reset. This function clears errors that sometimes occur during connection or resets an error that occurs during communication.

Prototype

```
#include <XMODEM.H>
int MDM_RESET_ERROR(h_modem);
```

Parameters

h_modem	Opened modem device handle.
---------	-----------------------------

Return Values

Success: SUCCESS: No errors occurred.
Failure: FAILURE: The error type can be determined by checking `errno`.

Example

The linked example code file demonstrates use of `MDM_RESET_ERROR ()`.

MDM_STATUS()

Obtains the opened modem device communication status and copies it to the global structure `status_info`.

Prototype

```
#include <XMODEM.H>
int MDM_STATUS(int h_modem);
```

Parameters

`h_modem` Device handle for opened modem.

Return Values

Success: SUCCESS: No errors occurred.
Failure: FAILURE: Cannot access modem status.

NOTE



If 0 returns, the modem status is copied to the global structure `status_info`.

The last element of the status array `m_stat` and contains the current signal information bit settings set up as follows:

- 0x80: break/abort detected
- 0x40: always zero
- 0x20: CTS detected (not connected, always zero)
- 0x10: RI present (not connected, always zero)
- 0x08: DCD present
- 0x04: frame error detected – latched error
- 0x02: overrun error detected – latched error
- 0x01: parity error detected – latched error

Example

The linked example code file demonstrates use of `MDM_STATUS()`.

MDM_WRITE_BLK()

Initializes or resets the modem device communication parameters. The modem must be initialized after opening before access is allowed.

Prototype

```
#include <XMODEM.H>
int MDM_WRITE_BLK(int h_modem, struct Opn_Blkc mdm_blk);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>Opn_Blkc mdm_blk</code>	Structure containing communications parameters.

Return Values

Success:	SUCCESS: No errors occurred.
Failure:	FAILURE: The error type can be determined by checking <code>errno</code> .

NOTE



For `Opn_Blkc` parameters, refer to the target terminals programmers manual.

Example

The linked example code file demonstrates use of `MDM_WRITE_BLK()`.

XMODEM.H Listings

Table 6 lists the descriptions of return codes, function parameter constants, unions, and structures found in XMODEM.H.

Table 6 XMODEM.H Error Return Codes

#define	Value	Meaning
E_LATCH	-5	Check latched error.
E_READ_CMD	-6	read() function failed.
E_FW_READ_CMD	-6	Modem is opened, read() firmware error occurred at modem device initialization.
E_FW_ONLY_CR	-7	Modem opened, firmware status initialize reply is only <CR>.
E_ONLY_CR	-7	Hayes response returned only a <CR>.
E_HR_TIMEOUT	-8	Hayes response function timed out.
E_FW_TIMEOUT	-8	Hayes response firmware time expired.
E_FW_STATUS	-9	Modem opened, Hayes "ERROR" returned on firmware initialization.
E_WRITE_CMD	-10	Unsuccessful write(); write() returned -1.
E_NOCARRIER	-11	DCD not present.
E_MI_READ_CMD	-14	Modem opened; CONFIG.SYS *MI variable contains an invalid Hayes command causing read() to return -1.
E_MI_ONLY_CR	-15	Modem opened; *MI = <CR> as the only Hayes response during the open.
E_MI_TIMEOUT	-16	Hayes response *MI wait timed out during modem open.
E_MI_STATUS	-17	Modem opened; Hayes "ERROR" response to *MI initialize during the open; invalid Hayes command in *MI.
E_INVALID_RESP	-18	Invalid modem response received during read().
E_STATUS_CMD	-19	Get modem device status failed; ioctl() returned -1.
E_READ	-23	Error reading modem buffer; read() returned -1.
E_INPUT_PENDING	-24	Input waiting at modem.
E_OUTPUT_FAILED	-25	Closing modem device failed due to output pending from a previous write() to modem.
E_READ_BLK	-26	Error reading modem parameters; ioctl() returned -1.
E_WRITE_BLK	-27	Error writing modem parameters; ioctl() returned -1.
E_H_FLUSH	-28	Error trying to flush Hayes command path.
E_M_FLUSH	-29	Error trying to flush modem data path before the close() device call.
E_CLOSE	-31	Failure to close modem device; close() returned -1.

Table 6 **XMODEM.H Error Return Codes** (continued)

#define	Value	Meaning
E_OPEN	-32	Failure to open modem device, open() returned -1.
E_PARTIAL	-33	Modem sent only part of the buffer.
E_INVALID_PARM	-34	Invalid parameter passed.
FAILURE	-1	Generic unsuccessful function execution.
E_X_FEATURE_NOT_SUPPORTED	-45	Terminal does not support the feature.
E_USBMDM_REMOVED	-46	USB modem removed.
E_USBMDM_SETSRLFAILED	-47	USB modem set serial failed.
E_WRITE	-35	Failure in writing to the modem.
E_X_OUT_PEND	-36	External modem has output pending.
E_X_IN_SESSN	-37	External modem is in data mode.
E_X_MI_TIMEOUT	-38	External modem MI response.
E_X_MI_STATUS	-39	External modem MI bad response.
E_X_HR_TIMEOUT	-40	External modem timeout.
E_X_BAD_RESP	-41	External modem bad dial response.
E_X_NOCARRIER	-42	External modem no carrier.
E_X_PARTIAL	-43	External modem partial send.
E_X_LATCH	-44	External modem latch error.

Table 7 **Non-error XMODEM.H Return Codes**

#define	Value	Meaning
OUTPUT_PENDING	1	Output waiting at modem; device is not closed.
INPUT_PENDING	2	Modem not closed; input pending at modem; modem status byte for input pending is greater than zero.
OUTPUT_FAILED	3	Modem not closed; failed output records exist from modem status information.
X_OUTPUT_PEND	4	Modem output pending.
X_INPUT_PEND	5	Modem input pending.
SUCCESS	0	Successful modem function execution.

Table 8 **Valid Hayes Numeric Response Codes**

#define	Value	Meaning
HAYES_OK	0	Hayes "OK" status returned from read() request.
CONNECT	1	Hayes "CONNECT" (300 baud).
CONNECT_300	1	Hayes "CONNECT" 300 baud alternate.
RING_DETECT	2	Hayes "RING" status.
NO_CARRIER	3	Hayes "NO CARRIER" status.
HAYES_ERROR	4	Hayes "ERROR" status.
CONNECT_1200	5	Hayes "CONNECT" 1200 status.
NO_DIALTONE	6	Hayes no dial tone status.

Table 8 **Valid Hayes Numeric Response Codes** (continued)

#define	Value	Meaning
BUSY_DETECT	7	Hayes "BUSY" busy detect.
NO_ANSWER	8	Hayes silence not detected.
CONNECT_2400	10	Hayes "CONNECT" 2400 status.
CONNECT_4800	11	Hayes "CONNECT" 4800 status.
CONNECT_9600	12	Hayes "CONNECT" 9600 status.
CONNECT_7200	13	Hayes "CONNECT" 7200 status.
CONNECT_12000	14	Hayes "CONNECT" 12000 status.
CONNECT_14000	15	Hayes "CONNECT" 14000 status.
CONNECT_19200	16	Hayes "CONNECT" 19200 status.
CONNECT_38400	17	Hayes "CONNECT" 38400 status.
CONNECT_57600	18	Hayes "CONNECT" 57600 status.
CONNECT_115200	19	Hayes "CONNECT" 115200 status.
CONNECT_75TX_ 1200RX	22	Hayes "V.23 originate connection establish" status.
CONNECT_1200TX_ 75RX	23	Hayes "V.23 answer connection establish" status.
DELAYED	24	Hayes "COUNTRY SPECIFIC BLACKLIST" status.
BLACKLISTED	32	Hayes "number dialed is blacklisted" status.
DATA	35	Hayes "data modem connection" status.
CARRIER_300	40	Hayes "0-300 data rate" status.
CARRIER_1200_75	44	Hayes "V.23 backward channel carrier" status.
CARRIER_75_1200	45	Hayes "V.23 forward channel carrier" status.
CARRIER_1200	46	Hayes "1200 data rate" status.
CARRIER_2400	47	Hayes "2400 data rate" status.
CARRIER_4800	48	Hayes "4800 data rate" status.
CARRIER_7200	49	Hayes "7200 data rate" status.
CARRIER_9600	50	Hayes "9600 data rate" status.
CARRIER_12000	51	Hayes "12000 data rate" status.
CARRIER_14400	52	Hayes "14400 data rate" status.
COMPRESSION_ CLASS5	66	Hayes "MNP CLASS 5 COMPRESSION 2400" status.
COMPRESSION_ V42BIS	67	Hayes "MODEM CONNECT AT V42BIS" status.
COMPRESSION_ NONE	69	Hayes "MODEM CONNECT WITH NO DATA COMPRESSION" status.
PROTOCOL_NONE	70	Hayes "NO ERROR CORRECTION PROTOCOL" status.
PROTOCOL_LAPM	77	Hayes "PROTOCOL LAPM" status.
PROTOCOL_ALT	80	Hayes "PROTOCOL ALT" status.
VFI_NO_LINE	90	VeriFone-defined "NO PHONE LINE" status.
VFI_DIALDONE	91	VeriFone dial done; dialing completed.

Table 9 Valid Silicon Laboratories USB Modem Numeric Response Code

#define	Value	Meaning
SLABS_OK	0	Silicon Labs Modem "OK" status.
SLABS_CONNECT_300	1	Silicon Labs Modem "CONNECT" (300) baud.
SLABS_RING_DETECT	2	Silicon Labs Modem "INCOMINGRING" status.
SLABS_NO_CARRIER	3	Silicon Labs Modem "NO CARRIER" status.
SLABS_SLABS_ERROR	4	Silicon Labs Modem "ERROR" status.
SLABS_CONNECT_1200	5	Silicon Labs Modem "CONNECT 1200" Status.
SLABS_NO_DIALTONE	6	Silicon Labs Modem "No dialtone" status.
SLABS_BUSY_DETECT	7	Silicon Labs Modem "Busy detect".
SLABS_NO_ANSWER	8	Silicon Labs Modem "Silence not detected".
SLABS_RINGING	9	Silicon Labs Modem "RINGING detected".
SLABS_CONNECT_2400	10	Silicon Labs Modem "CONNECT 2400" Status.
SLABS_CONNECT_4800	11	Silicon Labs Modem "CONNECT 4800" Status.
SLABS_CONNECT_9600	12	Silicon Labs Modem "CONNECT 9600" Status.
SLABS_CONNECT_19200	14	Silicon Labs Modem "CONNECT 19200" Status.
SLABS_CONNECT_7200	15	Silicon Labs Modem "CONNECT 7200" Status.
SLABS_CONNECT_12000	16	Silicon Labs Modem "CONNECT 12000" Status.
SLABS_CONNECT_14400	17	Silicon Labs Modem "CONNECT 14400" Status.
SLABS_CONNECT_16800	18	Silicon Labs Modem "CONNECT 16800" Status.
SLABS_CONNECT_21600	19	Silicon Labs Modem "CONNECT 21600" Status.
SLABS_CONNECT_24000	20	Silicon Labs Modem "CONNECT 24000" Status.
SLABS_CONNECT_26400	21	Silicon Labs Modem "CONNECT 26400" Status.
SLABS_CONNECT_28800	22	Silicon Labs Modem "CONNECT 28800" Status.
SLABS_CONNECT_31200	23	Silicon Labs Modem "CONNECT 31200" Status.
SLABS_CONNECT_33600	24	Silicon Labs Modem "CONNECT 33600" Status.

Table 9 **Valid Silicon Laboratories USB Modem Numeric Response Code** (continued)

#define	Value	Meaning
SLABS_CIDM	30	Silicon Labs Modem "Caller ID mark detected".
SLABS_FLASH	31	Silicon Labs Modem "Hook switch flash".
SLABS_STAS	32	Silicon Labs Modem "UK CID state tone alert signal detected".
SLABS_OVERCURRENT	33	Silicon Labs Modem "Over current condition".
SLABS_BLACKLISTFULL	40	Silicon Labs Modem "Black list is full".
SLABS_BLACKLISTED	41	Silicon Labs Modem "Attempted number is black listed".
SLABS_NOLINE	42	Silicon Labs Modem "No phone line is present".
SLABS_LINEINUSE	43	Silicon Labs Modem "Telephone line in use".
SLABS_POLARITYREVERSAL	44	Silicon Labs Modem "Polarity reversals was detected".
SLABS_NOPOLARITYREVERSAL	45	Silicon Labs Modem "Polarity reversals was not detected".
SLABS_CONNECT_56000	52	Silicon Labs Modem "CONNECT 56000" Status.
SLABS_CONNECT_32000	60	Silicon Labs Modem "CONNECT 32000" Status.
SLABS_CONNECT_48000	61	Silicon Labs Modem "CONNECT 48000" Status.
SLABS_CONNECT_28000	63	Silicon Labs Modem "CONNECT 28000" Status.
SLABS_CONNECT_29333	64	Silicon Labs Modem "CONNECT 29333" Status.
SLABS_CONNECT_30666	65	Silicon Labs Modem "CONNECT 30666" Status.
SLABS_CONNECT_33333	66	Silicon Labs Modem "CONNECT 33333" Status.
SLABS_CONNECT_34666	67	Silicon Labs Modem "CONNECT 34666" Status.
SLABS_CONNECT_36000	68	Silicon Labs Modem "CONNECT 36000" Status.
SLABS_CONNECT_37333	69	Silicon Labs Modem "CONNECT 37333" Status.
SLABS_PROTOCOLNONE	70	Silicon Labs Modem "No protocol".
SLABS_CONNECT_75	75	Silicon Labs Modem "CONNECT 75" Status.
SLABS_PROTOCOLV42	77	Silicon Labs Modem "V.42 protocol".
SLABS_PROTOCOLV42BIS	79	Silicon Labs Modem "V.42bis protocol".
SLABS_MNP2PROTOCOL	80	Silicon Labs Modem "MNP2 protocol".
SLABS_MNP3PROTOCOL	81	Silicon Labs Modem "MNP3 protocol".

Table 9 Valid Silicon Laboratories USB Modem Numeric Response Code (continued)

#define	Value	Meaning
SLABS_MNP4PROTOCOL	82	Silicon Labs Modem "MNP4 protocol".
SLABS_MNP5PROTOCOL	83	Silicon Labs Modem "MNP5 protocol".
SLABS_CONNECT_38666	90	Silicon Labs Modem "CONNECT 38666" Status.
SLABS_CONNECT_40000	91	Silicon Labs Modem "CONNECT 40000" Status.
SLABS_CONNECT_41333	92	Silicon Labs Modem "CONNECT 41333" Status.
SLABS_CONNECT_42666	93	Silicon Labs Modem "CONNECT 42666" Status.
SLABS_CONNECT_44000	94	Silicon Labs Modem "CONNECT 44000" Status.
SLABS_CONNECT_45333	95	Silicon Labs Modem "CONNECT 45333" Status.
SLABS_CONNECT_46666	96	Silicon Labs Modem "CONNECT 46666" Status.
SLABS_CONNECT_49333	97	Silicon Labs Modem "CONNECT 49333" Status.
SLABS_CONNECT_50666	98	Silicon Labs Modem "CONNECT 50666" Status.
SLABS_CONNECT_52000	99	Silicon Labs Modem "CONNECT 52000" Status.
SLABS_CONNECT_53333	100	Silicon Labs Modem "CONNECT 53333" Status.
SLABS_CONNECT_54666	101	Silicon Labs Modem "CONNECT 54666" Status.
SLABS_UNOBTAINABLENUM	102	Silicon Labs Modem "Unobtainable number".

NOTE



This is applicable only for Vx670 terminal. For more information, refer to the Silicon Laboratories Reference Manual.

Table 10 Latched Modem Device Data Error Conditions

#define	Value	Meaning
E_PARITY	1	Parity error on data received.
E_OVERRUN	2	Data overrun error on data received.
E_PARITY_OVERRUN	3	Parity and data overrun error on data received.
E_FRAMING	4	Framing error on data received.
E_PARITY_FRAME	5	Parity and framing error on data received.

Table 10 Latched Modem Device Data Error Conditions (continued)

#define	Value	Meaning
E_FRAME_OVERRUN	6	Framing and data overrun error on data received.
E_PARITY_FRAME_OVERRUN	7	Parity, framing, and data overrun errors received.

Table 11 XMODEM.H Valid Communication Protocol Attribute Requests

#define	Value	Meaning
MDM_BAUD_RATE	1	All request to change the opened modem device baud rate.
MDM_FORMAT	2	All request to change the opened modem device data format.
MDM_PROTOCOL	3	All request to change the opened modem device protocol.
MDM_SDLC_TYPE	4	SDLC request to change the opened modem device SDLC parameter.
MDM_STX_CHAR	5	Packet request to set the opened modem device STX character flag.
MDM_ETX_CHAR	6	Packet request to set the opened modem device ETX character flag.
MDM_COUNT	7	Packet request to set the opened modem device packet count.
P_CUSTOM	15	N/A custom protocol contained in application-defined function while setting protocol.

XMODEM.H Structures and Unions

The following structures and unions are used by functions and macros to set or retrieve mode device parameters.

union all_protocols

Used by `xmdm_init()` and `xmdm_set_protocol()` to change or set the modem device parameters. `all_protocols` is a union of the possible structures of each protocol type.

```
union all_protocols
{
    struct s_char /* Character mode protocol */
    {
        int rate; /* Baud rate */
        int format; /* Data format (parity,# data bits, #stopbits) */
    } char_mode;
    struct s_packet /* Packet mode protocol */
    {
        int rate; /* Baud rate */
        int format; /* Data format */
        int stx_char; /* Packet start of transmission character */
    }
}
```

```

int etx_char;    /* Packet end of transmission character */
int count;      /* Packet checksum used: 0, 1, or 2*/
} pkt_mode;

    struct s_sdslc      /* SDLC Protocol */
    {
        int rate;      /* SDLC connect baud rate */
        int format;    /* SDLC data format */
    } sdslc_mode;
struct custom      /* Custom Protocol */
{
    int custvarA;      /* Custom protocol variables*/ int custvarB;
    int custvarC;
    int custvarD;
    int (* custom_init)(); /* Custom protocol function address */
} custom_mode;
};

```

struct reject_struct

The reject record structure used by [MDM_RESET_ERROR\(\)](#) macro while calling `ioctl()`.

```

struct reject_struct
{
    char *buf_addr;
    /* Location to store reject queue record */
    int buf_size;
    /* Size of reject queue record storage location */
};

```

struct stat_struct

The modem status structure used by `ioctl()` during [xmdm_check_status\(\)](#), [xmdm_input_pending\(\)](#), [xmdm_failed_output_pending\(\)](#) and [xmdm_output_pending\(\)](#) to store the current modem status.

```

struct stat_struct
{
    unsigned char m_stat[4];
    unsigned char max_slots;
};

```

Modem Functions

The modem functions are described in [Modem Engine Function Calls](#).

Example Program

A Modem Engine example program can be found in the [Modem Engine Example Program](#) section.



Report Formatter

Most VeriFone applications are required to send formatted data to a printer or display. The most common examples of formatted data are sales receipts and batch reports. The AC report engine simplifies the task of formatting data for display or printer output.

The AC report conversion utility, `FORMCVT.EXE`, is a PC-based program that generates a binary template file from an ASCII report format file. The ASCII report format file contains information defining the layout, variables, and print characteristics that define a report; it can be created using any ASCII text editor.

Based on the information stored in the binary template file, the report engine formatter handles all formatting of data, buffer positioning, and sending of lines and special commands to a device driver. This approach to report generation reduces and even eliminates the need to modify application code when report formats change, since all of the formatting information is stored in a file independent of the application code.

This section discusses the usage of the report formatter functions. These functions are called in a C application to generate reports. Design of an ASCII report format file and generation of a binary template file is also discussed.

The report formatter consists of a set of general functions that format and output data. To use the formatter, the application must specify:

- what data to format,
- how to format the data, and
- where to output the formatted data.

The report conversion utility, `FORMCVT.EXE`, assists in answering the first two questions. `FORMCVT` uses a file (`G_VARS.H`) that defines report global variables and an ASCII report format file (`APPLIC.H`) to generate a variable index file and a binary template file for the report.

The `FORMCVT` header files `G_VARS.H` (provided by the programmer) and `APPLIC.H` (produced by `FORMCVT`) files are used by the application and the report formatter to define variables used in the report. The binary template file is used by the report formatter to determine how to format the variable data.

The report formatter determines where to send the formatted data based on an output initializer function. This function name is provided to the formatter at the start of the report.

The output initializer function (called by `formater_open()`) accepts a pointer to a `FORMATER` structure, defined in `<FORMATER.H>`, as an input parameter. The output initializer function is responsible for initializing the following members of the `FORMATER` structure:

<code>direct</code>	If 1, there is a direct interface to the output device and the report formatter should monitor system buffer usage. Otherwise, a spooled interface is used.
<code>h_comm_port</code>	Handle for the communications port.
<code>(*output_close)()</code>	Device close function.
<code>(*output_print)()</code>	Device output line function.
<code>(*output_mode)()</code>	Driver-dependent codes conversion function.

In addition, the output initializer function should perform any device initialization. Sample output initializer functions are provided with the report engine. These sample functions initialize direct output through an Application Construction printer driver to the ITP of the terminal.

The following steps show how to use the report conversion utility and report formatter.

Incorporating the Report Engine in an Application

- 1 Create an ASCII report format file and corresponding `GVAR.S.H` file.

All variables referenced in the report format file must be defined in `GVAR.S.H`. `GVAR.S.H` may also contain other variables not referenced in the report format file. See the global variables declaration file.

- 2 Run the report engine conversion utility, `FORMCVT`, to convert the ASCII Report Format file to a binary template file.

`FORMCVT` generates a variable index file, `APPLIC.H`.

`FORMCVT` command line syntax is as follows:

```
formcvrt TEMPLATE[.TXT] [-fTEMPLATE.FRM] [-hAPPLIC.H]
```

- 3 Determine the output initializer function(s).

An output initializer function initializes the variables that specify functions interfacing with the output device; it also opens and initializes the output device. The output initializer functions listed in step four are provided with the report engine for a direct interface to the following printers: `P150DIR.C`, `P250DIR.C`, and so on.

- 4 Use the following functions to generate a report from the application:

- `formater_open()`: Initializes the report by opening the specified binary template file and calling the specified output initializer function.
- `formater_set_flags()`: Builds an unsigned long bitmap flag for report templates that contain conditional lines.

- `formater_line()`: Outputs one or more report lines.
- `formater_close()`: Closes the binary template file and output device.

5 Compile the application modules.

These modules include `GVAR.S.H` and `APPLIC.H`.

6 Link the application modules with the report engine formatter module `FORMATR3.G` or `FORMATR4.G` and any device driver module, such as `DR3.G` or `DR4.G`, to create a downloadable file.

7 Download this file along with the binary template file(s) to the terminal.

For instance, an application can consist of one `.OUT` file and four template files. To download this application:

- a** Create an ASCII text file called `DOWNLD.SYS`. The contents of the file can be:

```
IAMEX001.OUT
ISPOT.FRM
IEOD.FRM
IEOF.FRM
IRECEIPT.FRM
```

In this example, `AMEX001.OUT` is the downloadable application, the rest are report template files.

- b** Type the following command at DOS prompt:

```
DDL -FDOWNLD.SYS
```

Conditional Printing

The AC report engine supports conditional printing. At report generation, the report engine formatter determines which lines are printed by comparing a user-defined condition variable (see [Report Formatter Example Program](#)) with corresponding conditional codes in the template file. If the conditions evaluate to a Boolean true value, the line prints. This allows a variety of reports to be generated using a single template.

Report Conversion Utility

This utility is a stand-alone program running under the DOS operating system. The command line `FORMCVT` syntax is as follows:

```
FORMCVT INPUT_FILE[.TXT] [-fTEMPLATE.FRM] [-hTEMPLATE.H]
```

- `INPUT_FILE` is the ASCII report format file that contains the report format. If the extension is omitted, it defaults to `.TXT`.
- `-f` specifies an alternate binary output filename. If this option is omitted, the template filename is the same as the input file name with a `.FRM` extension.
- `-h` specifies an alternate output variable index filename. If this option is omitted, `FORMCVT` generates an index filename `APPLIC.H`. This file is used by the report engine functions; do not change these functions.

FORMCVT also requires the input variable definition file `GVAR.S.H`. FORMCVT assumes this file exists in the current directory. `GVAR.S.H` contains the definitions of all variables referenced in the report format file. If an application uses multiple binary template files, `GVAR.S.H` contains the cumulative definitions for all the global variables used in these reports.

NOTE

`GVAR.S.H` and `APPLIC.H` must be included the application at compile time.

Report Format Files

A report format file is a FORMCVT ASCII input file that defines the layout, variables and print characteristics for a report. FORMCVT then generates the corresponding binary file used by the report engine formatter during report generation.

Some examples of report layout commands are “center a literal string” and “left justify a variable”. Examples of print characteristic commands are “print in red” and “print in compressed mode”. All commands supported by FORMCVT are listed in [Report Format File Guidelines](#).

To include global variable values in a report, the variable names must be referenced in the report format file. These variables must also be provided to FORMCVT in `GVAR.S.H`. At report generation, the report engine formatter formats these variable values in the report.

The report format file and `GVAR.S.H` can include the following types of variables:

- character (char)
- string (character array)
- integer
- long integer
- structure elements (the structure path must be provided)
- array (of any type)

Floating point and double types are not supported. To include floating point values in a report, declare the report variable as a string and format the floating point value in the string before calling the report engine formatter.

Report Format File Guidelines

The following rules apply to the format and content of each line of the ASCII report format file. References are made to the example format file.

- 1 Start each report line with a line number.

Valid line numbers range from 1 to 127. Leading zeros are not required, but can be added for text alignment.

Line numbers do not have to be sequential, but each must be greater than or equal to previous line number. Skipped line numbers can be printed as blank lines or simply ignored.

In the [Example Format File](#), note that line 3 has been skipped. A parameter in the call to `formater_line()` within the application specifies how lines 3, 9, 14, and 17 are printed: either as blank lines or skipped.

At report generation, the report engine formatter can be instructed to print one line or a number of lines. Lines can also be reprinted any number of times.

- 2** The character immediately following the line number must be a colon (:), an ampersand (&), or an equal sign (=).

Both the ampersand and the equal sign indicate that the line is conditional, and they are followed by a value defining this condition. The colon signals the end of the conditional portion and the beginning of the text portion of the line.

Conditional values can be specified in three forms: “B” (binary), “D” (decimal), or “X” (hexadecimal). For example, the following values are equivalent:

&X000A

&B1010

&D0010

All values are numerically right justified, making these values also equivalent to:

&X00000001

&X000001

&X1

At report generation, a four-byte, bitmap-conditional flag is passed to the report engine formatter. This value is logically ANDed with the conditional value of lines with an ampersand conditional operator, or compared for a match with the conditional value of lines with an equal sign conditional operator. The line prints, if the ampersand/AND or the equal/MATCH is TRUE.

When multiple conditionals are defined for a given line number, the first one evaluating to true prints. At this point, the report engine formatter begins processing the next sequential line number in the sequence.

NOTE



A line number followed only by a colon always evaluates to true.

Example

```
01&X1: "ONE"
01: "TWO"
01&X2: "THREE"
02=X02000301: "FOUR"
02: "NEXT"
```

In this example, assume a conditional value of 0x02000301 is passed to the report engine formatter.

The result of ANDing 0x02000301 and 0x00000001 (from the first line in the example) evaluates to TRUE, and so the string constant "ONE" prints. The formatter then skips to the line number 02. The result of matching 0x02000301 to 0x02000301 evaluates to TRUE, so the string constant "FOUR" prints.

If a conditional value of 0x01000000 is passed to the report engine formatter, the string constants "TWO" and "NEXT" print because the second and fifth lines in the example are the first (and in this case, the only) lines in each line-set to evaluate to TRUE.

It is important to note that the string constant "THREE" is never printed, because the second line (TWO) always evaluates to TRUE.

3 Use standard C-style comments (`/* */`) in GVAR.S.H and the report format file.

In GVAR.S.H, the comments can appear anywhere in the file, except preceding a variable on the same line.

Acceptable:

```
int item; /* Item Purchase Number */
```

Unacceptable:

```
/* Item Purchase Number */ int item;
```

In the report format file, a comment must occupy the entire line. Embedded comments are not allowed.

Acceptable:

```
/* The next line is line one */
01&X0001:1.40, \Cmerchant
```

Unacceptable:

```
01&X0001:1.40, \Cmerchant /* line 1 */
```

For ease of reading, white space characters (blank lines, spaces, tabs, and so on) can be used in these files.

NOTE

In the GVAR.S.H file, variable declarations must start in column one.

4 Each line in the report format file is composed of fields.

The first field begins immediately after the colon that terminates the conditional value. Subsequent fields are separated by semicolons (;). Each line can contain multiple fields.

In the example, [Example Format File](#), line 2 uses only the first field, while line 10 uses the first and second fields.

Each field follows the following rules of syntax:

```
[start.end,] [options] {global_var}
    {[options] string}
```

start Represents the printer column where the field begins.

end Represents the printer column where the field ends.

options Represents any of the special print characteristics that can be used.

Only one data element can be specified per field. This can be either of the global variables defined in `GVAR$H` or a string constant.

Example

```
01:1.13,\Ctemp
```

The value of `temp` is centered on the first line of the report starting in column one and ending in column 13.

```
01:1.13,\Ctemp;20.30,"\W\CTESTING"
```

The string `TESTING` prints double-wide (`\W`) and centered in the field (`\C`) between columns 20 and 30.

NOTE



The starting and ending column numbers must be separated by a period and must be in range for the printer device used. A comma (,) must follow the ending column number.

5 Starting and ending column numbers must be specified for every field.

This makes the template easier to maintain and eliminates positional ambiguities. However, when printing a string constant, only the starting column of the field must be specified. The formatter begins printing at the fields starting column and continues until the end of the string is reached, or to the maximum width allowed by the printer.

If the width of a field exceeds the width defined by the starting and ending columns, the field is truncated.

The default justification for global variables is right justified.

The default justification for string constants is left justified.

6 Special print characteristics and layout commands are expressed in the form of a two-character string, “\X”, where X is any one of the following:

A:	Print the following string or variable in Arabic.
B:	Print the following string or variable in black.
C:	Center the following string or variable in the specified column range preceding this command.
D:	Print the following string or variable in red.
E:	Turn off printing in red.
F:	Set the character size to 5 x 7 and enable underline printing.
G:	Print logo file.
H:	Print the following string or variable in double height.
I:	Reinitialize the printer.
J:	Set the character size to 7 x 7.
L:	Left justify the following string or variable in the specified column range preceding this command.
N:	Restore the printer to normal printing mode.
P:	Eject six blank lines in the printer.
R:	Right justify the following variable or string in the specified column range preceding this command.
S:	Enable printing in compressed mode.
T:	Select font size and font table.
W:	Print the following string or variable in double width.
X:	Hexadecimal printing using the selected font table.
Z:	Use ASCII character set.
% (percent)	Specifies the pad character to use when printing variables. The default pad character is a space (ASCII 32). This character pads the width of the field regardless of justification.

These special print characters can be specified in or out of the literal strings. In the example, line 1 centers and prints the contents of the global variable merchant in double width.

Line 13 of the example prints the literal string “TOTAL: \$”, also in double width.

Errors If FORMCVT detects any errors, it displays an error message and aborts the parsing operation.

Example Format File

An example demonstrating the use of conditional lines is given in the [sample format file](#).

Global Variables Declaration File

All the variables shown in the [Example Format File](#) must be declared in the global variables declaration file GVAR.S.H.

Each line in this file should contain only one variable declaration, and each declaration must start in column 1. C-style comments are allowed in this file, with the exception that comments cannot be placed in front of variables.

```
/* declarations for global variables used in the report */
int item;
char amount[13];
char tip[13];
char total[13];
char merchant[20];
char date_time[20];
char t_card_acct[30];
char t_card_exp[5];
char t_auth_code[9];
char t_invoice[7];
```

Data Structures

The following [Global #defines](#) table, defines the global variables found in file APPLIC.H.

Global #defines

The following are defined in the file APPLIC.H:

Table 12 **Global #defines**

#Define	Value	Meaning
G_C	1	Indicates a variable of character type.
G_S	2	Indicates a string variable.
G_I	3	Indicates a variable of integer type.
G_L	4	Indicates a variable of long integer type.

Global Structures

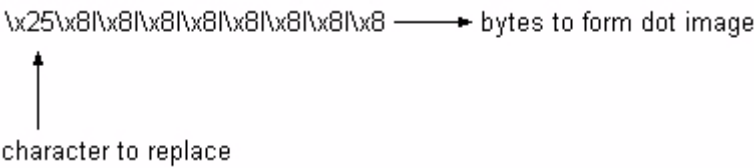
The following structure is used by the report formatter to store the information about all global variables used in the report generation process.

```
typedef struct
{
void *addr;
int type;
} g_data;
g_data data[ ];
```

The address of the variable is stored in the first element of this structure and the type, as shown in the [Global #defines](#) section above, is stored in the second element, type. The number of elements in data[] is application dependent.

Sample Output

The following sample receipts demonstrate the use of conditional true and conditional false output, and are based on the examples provided on the preceding pages.



Report
Formatter
Functions

The report formatter functions are described in [Report Formatter Function Calls](#).

Example
Program

A report formatter example program can be found in the [Report Formatter Example Program](#) section.

Database Library

The Database Library supports database file operations (for example, open, read, write, and delete) with key-based search of data records.

The Database Library provides function calls to create and manipulate a database. This library is based on an index (keyed) file system to allow specified search key(s) to access individual records in the database. These search keys provide random access to the database. There are two major advantages in using this keyed file system:

- A structure can be defined for an index (key) file. The structure contains the list of search keys required to access the database records (for example, card number and server ID). Utilization of search keys provides a fast access method into the database.
- Database records can be compressed externally to save system memory.

Database Features

The Database Library:

- allows random access to the database records through an index file,
- stores variable length records in database files,
- handles compressed records, and
- provides sequential access (in both the forward and backward directions) to the data records.

File Formats

Database and index file formats are discussed in this section.

Database File Format

The database contains variable length records. Each record is composed of a forward length component, the actual data, and a backward length component. This record format is referred to as Double Variable Length Record (DVLR). The DVLR format allows searching the database in both directions, forward and backward.



Figure 4 DVLR Format

Figure 4 shows the DVLR format of a data record. FLEN stands for forward length, and BLLEN for backward length. Both the FLEN and BLLEN are two bytes. Data Record indicates the actual data stored.

Index File Format

Each time a database file is open (create) using the API call `db_open()`, a corresponding index (keyed) file is created. Each index file record includes the search keys used to access records in the database. The minimum requirement for each index file record is a field of LONG used to store the offset address of the associated database record. The index record format can be expanded by adding fields to define each search key. The following example illustrates an index file record structure:

```
typedef struct
{
    long rec_num;
    short which_track; // 0 = Manual entry, 1 = Track 1, 2 =
    Track 2
    char hostno[3]; // Host number
    char server[4]; // Server ID
    char ticket[29]; // Ticket number, auto_download field
    char Card_type[3]; // Two character card type
}DB_IDX_KEY
```

When calling `db_open()`, `sizeof(DB_IDX_KEY)` should be used as the parameter for key length.

File Storage Structure

Figure 5 shows the file storage structure. Header, Rec1, Rec2, and Rec3 represent the index records. Data Rec 1, 2, and 3 represent the data portion of the records. Each data record consists of FLEN, Data Rec, and BLEN. Each index record stores the offset for a particular data record.

For example, Rec1 stores an offset value of 0. Assume the length of Data Rec 1 is 21, then,

$FLEN + \text{Length of Data Rec 1} + BLEN = 25$ or $(2 + 21 + 2 = 25)$

where, $FLEN = BLEN = 2$ bytes.

Now, Rec2 stores an offset value of 25. Similarly, if the length of Data Rec 2 is 50, Rec3 stores an offset value of $25 + (2 + 50 + 2) = 79$. So, if a data record is deleted in the database, the corresponding data and index records are removed and only the offsets are recomputed.

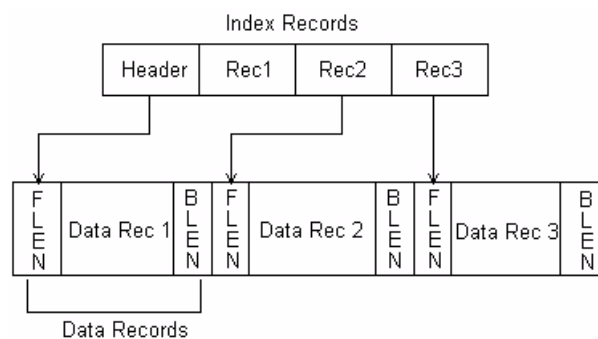


Figure 5 File Storage Structure

NOTE



The record numbers start at zero. The delete operation slows down the application as the offsets must be updated.

File Compression

Although the Database Library can read and write compressed records, a compression function is not provided in this library. Compress or decompress database records prior to calling any of the Database Library functions.

**Database
Library
Functions**

The Database Library functions are described in [Database Library Function Calls](#).

Printer Drivers

The printer driver provides an easy way to manage printing and use various printer attributes provided.

Various types of printing supported by the printer driver are as follows:

- LOGO type printing
- Printer Font Type printing (PFT)
- Printing monochrome bitmaps

Verix eVo
Based Terminal
ITPs

The ITP is a thermal array printer that communicates with the host terminal through the RS-232 port. The data format of the ITP is fixed at 8 bits, no parity, 1 stop bit (8N1) at 19200 bps. It has 64 font tables used to download user-defined fonts for printing.

The ITP has the following additional features:

- Downloadable user fonts
- Downloadable user logo

Downloadable
User Fonts

Lists the sizes of the fonts that can be downloaded to the Verix eVo based terminal ITPs.

Table 13 Font Sizes

Font Resolution	Font Size	Bytes Per Character	Font Tables Required
5X8	2	8	1
8X14, 42 column mode ^a	3	14	2
8X14, 32 column mode	4	14	2
16X16	1	32	4
24X24	5	72	9
32X32	6	128	16
48X48	7	288	36
64X64	8	512	64

a. Indicates the default font when the printer is powered on.

Font Table 0 is reserved for built-in fonts, such as 5 x 8 in 24 column mode, 8 x 14 in 42 column mode, and 8 x 14 in 32 column mode. User font loads can start from font table 1 through table 64. The user must specify where to load the different font tables for use by the application. For example, if the user downloads a 16 x 16 font at font table 1, the next available font table for a new font download is 5 because the 16 x 16 font occupies four font tables.

As the fonts download into the flash of the printer, they are not erased from memory even if the printer is switched off. However, fonts can be overwritten by downloading new fonts to the same font table.

Downloadable Printer Logo

The ITP can handle logos of various sizes, with a maximum dimension of 384 horizontal dots by 240 vertical dots (384X240). Only one image can be downloaded at a time to the ITP.

NOTE



Downloading a new logo into the ITP erases the any other logo stored in flash.

The distinctive features of the ITP are:

- Selectable characters-per-line through the <ESC> F command; one of 42, 32, or 24 lines.
- User-defined graphics download through the <ESC> g command.
- Remote self-test through the <ESC> s command.
- User-defined logo download.
- Printer firmware download.

NOTE



The printer ID for the V*520 and V*680 will be P.

Printing Monochrome Bitmap

This feature enables printing of monochrome bitmaps independent of the printers. This feature is supported in Verix eVo terminals.

- [Printing Monochrome Bitmap](#)

Printer Driver Functions

The printer driver functions are described in the following section:

- [Print Pak 3700 Function Calls](#)

Example Program

The following is the example program of printer driver:

- P3700 Example Program



Function Calls

This chapter discusses the [AC Library Function Categories](#) and provides function call descriptions for the following modules:

- AC Library Function Calls
- DVLR Function Calls
- IVLR Function Calls
- SVC Function Calls
- Application Idle Engine Function Calls
- Message/Data Entry Engine Function Calls
- ISO 8583 Message Interface Engine Function Calls
- PIP Engine Function Calls
- Data Capture Engine Function Calls
- Modem Engine Function Calls
- Report Formatter Function Calls
- Database Library Function Calls
- Print Pak 3700 Function Calls

AC Library Function Categories

The functions in this library are organized into one of four functional groups:

- string,
- console I/O,
- device, and
- utility.

The level for each group indicated in the following table, refers to the functional level of the routine. The AC Library is self-referencing, meaning that some functions in the library depend on other functions also in the library. Understanding function levels helps to show the relationship of one function to others in the library. The four groups are as follows:

Level	Description
0	Macros that do not depend on any ACL function.
1	Compiled C functions that can use level 0 macros.
2	Functions that use at least one function from level 1.
3	The most complex functions in the ACL — can use ACL functions from any level.

Each function in the AC Library is listed by category and level in [Table 14](#).

Table 14 Verix eVo ACT Library Functions by Category

Function	Level	Function	Level	Function	Level
File Functions		Console I/O Functions			
delete_dvlr/ivlr	3			scroll_buffer	2
do_compression	3				
insert_dvlr/ivlr	3	beep	1		
read_dvlr/ivlr	3	bitfun	1	str2dsp_len	1
seek_dvlr/ivlr	3				
write_dvlr/idvlr	3	display	1	view_buffer	3
Device Functions					
card_parse	3	display_at	2	String Functions	
chk_card_rdr	1			append_char	1
CHK_TIMEOUT	0	display_new	2	atoi1	
clock2int	3		2	chars2int	1
clock_data	3	dsp_strs	2	chk_luhn	2
		ERR_BEEP	0	ctons1	1
p_set_baudformat	1	getkbd_entry	3	delete_char	1
			3	f_dollar	2
set_ftimeout	2		1	fieldcnt	1

Table 14 Verix eVo ACT Library Functions by Category (continued)

Function	Level	Function	Level	Function	Level
SLEEP	0	getxy	1	fieldfix	1
track_parse	2		1	fieldray	1
Utility Functions		KBHIT	0	fieldvar	2
		KBHIT_ALT	0	insert_char	1
cvt_ym	3	keyin_amt_range	3	insert_decimal	3
julian_date	2	keyin_mapped	1	int2str	2
LEAP_YR	0			long2money	3
load_bmp	3	NORM_BEEP	0	long2str	3
MAX	0	pause	1	mult_strcat	1
MEMCLR	0			ntocs	1
MIN	0	prompt	3	pad	2
month_end	2			purge_char	1
range_vlr	3	prompt_at	3	range	3

AC Library Function Calls

This section describes the following AC library function calls:

- act_kbd_pending_test()
- append_char()
- atox()
- beep()
- bitfun()
- card_parse()
- chars2int()
- chk_card_rdr()
- chk_luhn()
- CHK_TIMEOUT()
- clock_data()
- clock2int()
- ctons1()
- cvt_ym()
- delete_char()
- display()
- display_at()
- display_new()
- do_compression()
- dsp_strs()
- ERR_BEEP()
- f_dollar()
- fieldcnt()
- fieldfix()
- fieldray()
- fieldvar()
- getkbd_entry()
- getxy()
- insert_char()
- insert_decimal()
- int2str()
- julian_date()
- KBHIT()
- key_card_entry()
- keyin_amt_range()
- keyin_mapped()
- LEAP_YR()
- load_bmp()
- long2str()
- MAX()
- MEMCLR()
- msg_display_at()
- msg_display_new()
- MIN()
- month_end()
- mult_strcat()
- NORM_BEEP()
- ntocs()
- p_set_baudformat()
- pad()
- pause()
- prompt()
- prompt_at()
- purge_char()
- range()
- range_vlr()
- scroll_buffer()
- set_imeout()
- sgetf()
- SLEEP()
- sputf()
- strn2int()
- str2digit()
- str2dsp_len()
- str2int()
- str2long()
- set_chars_per_key_value()
- track_parse()
- view_buffer()

act_kbd_pending_test()

Checks if the target character is present in the keyboard buffer.

Prototype

```
#include <actutil.h>
int act_kbd_pending_test(int targetcode);
```

Parameters

targetcode Character key code to be checked by the function. Valid values are:

- KEY_CANCEL
- KEY0 to KEY9
- KEY_a to KEY_g

Return Values

Success: 1: The target character is present in the keyboard buffer.

Failure: 0: Target character is not present in the keyboard buffer.

append_char()

Accepts a pointer to a null-terminated string specified by string, and appends a character specified by c.

If the buffer passed is an empty string, the character is inserted as the first character of the string. The buffer is unaffected if the passed character is null.

Prototype

```
#include <aclstr.h>
int append_char(char *string, char c);
```

Parameters

string Buffer with null-terminated string.

c Character to append.

Return Values

Length of the string, including the appended character.

Dependencies

Verix eVo SDK

NOTE



Bounds checking are not performed. The calling routine is responsible for ensuring that space is available for the additional character.

See Also

[insert_char\(\)](#), [delete_char\(\)](#), [purge_char\(\)](#), [pad\(\)](#)

Example

The linked example code file demonstrates use of [append_char\(\)](#). Also see the example in the [EXTRANS.C](#) section.

atox()

Converts an ASCII hex character to a binary byte value.

Prototype

```
#include <aclstr.h>
BYTE atox (BYTE character);
```

Parameters

character Character to convert.

Return Values

Success: 0–15: Successful operation.
Failure: 0xFF: Error (invalid character).

Example The linked example code file demonstrates use of atox().

beep()

Activates the beep feature of the terminal. Based on the `type` parameter, `beep()` produces one of two types of beeps, both can occur on a conditional or unconditional basis.

A conditional beep is based on the value of the `*BP` parameter in the system `CONFIG.SYS` file. If the `*BP` value is zero or empty, the beep executes, therefore, default allows a conditional beep. A non-zero value prevents a conditional beep.

Prototype

```
#include <aclconio.h>
void beep(int type);
```

Parameters The following are valid type values defined in `ACLCONIO.H`:

type Normal, Error, or Conditional.

- `NORML`: Normal beep
- `ERROR`: Error beep
- `C_NORML`: Conditional normal beep
- `C_ERROR`: Conditional error beep

Return Values The function returns on completion of the beep. No value is returned.

Dependencies Verix eVo SDK

See Also [NORM_BEEP\(\)](#), [ERR_BEEP\(\)](#)

Example The linked example code file demonstrates use of beep().

bitfun()

Performs bit manipulation on an unsigned integer.

Prototype

```
#include <aclconio.h>
int bitfun (UINT *map, BYTE bit, BYTE set);
```

Parameters

- map** Pointer to an unsigned integer bitmap.
- bit** Bit to manipulate: 0–15, counting from the right.
- set** Action to take on bit:
- 0 clears the bit
 - 1 sets the bit
 - 2 checks the bit

Return Values

Defined for CHECK_BIT only.

Success: 1: Bit is set.

Failure: 0: Bit is clear.

Example

The linked example code file demonstrates use of bitfun().

card_parse()

Accepts data from the card reader and parses it to the selected track data.

Prototype

```
#include <acldev.h>
int card_parse(char *track, struct TRACK *parsed, char *order);
```

Parameters

`track` Card reader data.

`parsed` Parsed output.

`order` Track selection order.

Return Values

Success: 1,2,3: Track number that was read (successful).

Failure: TK_NOT_FOUND (-1): None of the tracks specified in `order` contained a valid data status byte.

 INVLD_ORDER (-2): The order contains a character that is not 1, 2, or 3.

 BAD_READ (-3): An attempt to parse a track field in the card read data fails.

 INVLD_FORMAT (-4): Invalid format.

 ACT_FRAMING_ERROR (-6): Track status byte from card read is 2, 3, or 4. Parsed track is still stored in structure `TRACK`, and `errno` contains the parsed track.

The individual tracks returned from `read()` are formatted as:

```
[C][S][track data]
```

where,

- `[C]` is a single byte representing the length of the string containing the track data, including the count byte.
- `[S]` is a one byte status of the read for the track.
- `[track data]` is the complete track read of the track (if available). This field is omitted for tracks not available on the terminal; however, the count byte and status are provided for all three tracks.

Three of these fields are read from the card device regardless of how many physical tracks are read by the terminal. The tracks appear in order from track 1 to track 3, for example,

```
[C1][S1][track1][C2][S2][track2][C3][S3][track3]
```


The supported tracks comply with standard specifications as follows:

Track 1 format	<p>IATA track 1 format consisting of a maximum of 79 alphanumeric characters. The track format is:</p> <pre>[ss][fc][PAN][fs][name][fs][other][es][lrc]</pre> <p>where,</p> <ul style="list-style-type: none"> [ss] Start sentinel of “%” (0x25) [fc]: one-byte format code. [PAN] Primary account number; maximum 19 digits. [fs] Field separator of “{” (0x7B) [name]: Name of the card holder; maximum 26 characters [other] Can contain the following: <ul style="list-style-type: none"> • 4-digit expiration date^a • 3-byte restriction or type^a • 5-byte offset of PVN^a • 16 bytes of discretionary data [es] End sentinel of “?” (0x3F). [lrc] 1-byte longitudinal redundancy check value
Track 2 format	<p>ABA track 2 format consists of a maximum of 40 numeric characters. The track format is:</p> <pre>[ss][PAN][fs][other][es][lrc]</pre> <p>where,</p> <ul style="list-style-type: none"> [ss] Start sentinel of “,” (0x3B). [PAN] Primary account number; maximum 19 digits. [fs] Field separator of “=” (0x3D). [other] Can contain the following: <ul style="list-style-type: none"> • 4-digit expiration date^b • 3-byte restriction or type^b • 5-byte offset of PVN^b • 5 bytes of discretionary data [es] End sentinel of “?” (0x3F). [lrc] 1-byte longitudinal redundancy check value.
Track Format 3	<p>ISO 4909 track 3 format consisting of a maximum of 107 alphanumeric characters. This standard was published in October, 1973; other formats are now in use. The track format is:</p> <pre>[ss][fc][PAN][fs][secure][other][es][lrc]</pre> <p>where,</p> <ul style="list-style-type: none"> [ss] Start sentinel of “%” (0x25) [fc]: Two-byte format code. [PAN] Primary account number; maximum 19 digits. [fs] Field separator of “=” (0x3D).

- a. Required by VISA and MasterCard Track 2 format.
b. Required by VISA and MasterCard Track 3 format.

[secure]	<p>Name of the card holder, 26 characters max</p> <p>Can contain the following:</p> <ul style="list-style-type: none"> • 3-byte country code • 1-byte field separator of “=” (0x3D) • 3-byte currency code • 1-byte currency exponent • 4-byte amount authorized per cycle • 4-byte amount remaining this cycle • 4-byte cycle begin • 2-byte cycle length • 1-byte re-entry count • 6-byte PIN control parameters <p>or</p> <ul style="list-style-type: none"> • 1-byte field separator of “=” (0x3D) • 1-byte interchange control • 2-byte PAN service restriction • 2-byte SAN-1 service restriction • 2-byte SAN-2 service restriction • 4-byte expiration date • 1-byte field separator of “=” (0x3D) • 1-byte card sequence number • 9-byte card security number <p>or</p> <ul style="list-style-type: none"> • 1-byte field separator of “=” (0x3D)
[other]	<p>Can contain the following:</p> <ul style="list-style-type: none"> • Optional first secondary account number • Optional second secondary account number • 1-byte relay marker • 6-byte cryptographic check <p>or</p> <ul style="list-style-type: none"> • 1-byte field separator of “=” (0x3D) • Discretionary data
[es]	End sentinel of “?” (0x3F).
[lrc]	1-byte longitudinal redundancy check value.

[card_parse\(\)](#) processes tracks based on the `order` parameter. On terminals that read multiple tracks on a single pass, the application can select which is the preferred track. For example, a terminal can read both track 1 and track 2 on a card swipe.

The information on track 1 provides information not available on track 2, and the track 1 information is preferred. If the data in track 1 is invalid or missing, then track 2 data is read and returned. By specifying the order for processing as “12”, [card_parse\(\)](#) always attempts to obtain track 1, but returns track 2 if track 1 fails.

If a non-existent track is specified in the `order` parameter, `card_parse()` processes as though the track existed but was invalid.

`card_parse()` places the track information in a caller-supplied structure `TRACK` (defined in `ACLDEV.H`). This structure contains the following data elements and sizes:

```
struct TRACK
{
char acct [23];/* account number */
char exp [5];/* expiration date */
char name [27];/* name for track 2 */
char type [4];/* required by VISA/ MC */
char PVN [6];/* required by VISA/ MC */
char disc [17];/* tracks 1 and 2 only */
char track [108];/* raw track data */
}
```

Not all fields are available on all tracks; the following table indicates the fields available after parsing. This function clears all fields in the structure prior to parsing. Any fields not used for a particular track are zero-filled.

	Track1	Track2	Track3
acct	X	X	X
exp	X	X	X
name	X	—	—
type	X	X	—
PVN	X	X	—
disc	X	X	—
track	X	X	X

The `track` field is an exact copy of the data portion of the track field of the card read. This can be used for inclusion in packets or additional parsing, if required. The actual parsing of the track field is accomplished by calling the ACL routine `track_parse()`.

For tracks 1 and 2, if discretionary data is not available, `card_parse()` returns success and the discretionary field is null.

Dependencies ACL `MEMCLR()`, `track_parse()`

NOTE



Data must be formatted as read from card reader.

Example

The linked example code file demonstrates use of `card_parse`.

chars2int()

Converts a string of up to five (ASCII) numeric characters to its integer equivalent.

Prototype

```
#include <aclutil.h>
int chars2int (char *s_buffer, int i_num);
```

Parameters

s_buffer Pointer to a buffer containing the string to convert.
i_num Number of characters in the buffer.

Return Values

Success: Converted integer.

See Also

str2int()

Example

The linked example code file demonstrates use of chars2int().

CAUTION

This is an integer function. Be sure to convert values in the correct range. If the string exceeds five characters, it is truncated on the right. For example, -12345 convert to -1234.

Any invalid string character is ignored. For example, 1A2B3 converts to 123.

chk_card_rdr()

Checks if data is available from the card reader port. This function is similar to the keyboard function [KBHIT\(\)](#).

NOTE

Use of this function on platforms where no magnetic card device exists will always return 0.

Prototype

```
#include <acldev.h>
int chk_card_rdr(int h_card);
```

Parameters

`h_card` Handle of card reader port.

Return Values

Success: ≥ 0 : Data is available.

Failure: 0: No card data is available.

 -1: An error related to the card reader occurred, see `errno`.

Dependencies

Verix eVo SDK

NOTE

This function reads the card data and writes the data back to the card reader. So the complete data read from the card can not exceed the value of `CARD_SIZE` as defined in `ACLDEV.H` (the amount of space allocated for the read). If the bytes read are less than one, no write to the card reader is executed. The card reader device must be open.

Example

The linked example code file demonstrates use of `chk_card_rdr()`.

chk_luhn()

Accepts a null-terminated string containing digits, and verifies that the last digit in the string is the correct MOD 10 check digit for the preceding digits in the string.

Prototype

```
#include <aclstr.h>
int chk_luhn(char *buffer);
```

Parameters

buffer String to check.

Return Values

Success: 1: The MOD 10 check digit is valid.

Failure: 0: The MOD 10 check digit is invalid.

 -1: The string has no character, more than 25 characters, or less than 2 characters.

Calls `SVC_MOD_CK()` to complete the check and returns the result of `SVC_MOD_CK()`.

Dependencies

Verix eVo SDK

NOTE



String limits are 25 maximum and 2 minimum.

Example

The linked example code file demonstrates use of `chk_luhn()`.

CHK_TIMEOUT()

Uses the `ioctl()` function to compare a value specified by `time_val` to the current clock tick value. `CHK_TIMEOUT()` returns TRUE(1) if the comparison value is greater than the current clock tick value.

Prototype

```
#include <acldev.h>
int CHK_TIMEOUT(int h_clock, unsigned long time_val);
```

Parameters

<code>h_clock</code>	Clock handle.
<code>time_val</code>	Time value in ticks.

Return Values

Success:	1: Time out not expired.
Failure:	0: Time out expired.

See Also

[set_itimeout\(\)](#)

Example

The linked example code file demonstrates use of `CHK_TIMEOUT()`.

clock_data()

Provides access to the terminal system clock.

Prototype

```
#include <aclddev.h>

int clock_data (int clock, int i_type, char * clock_buffer,
               char * s_dest_buf);
```

Parameters	clock	Clock handle.
	i_type	Type of action to perform with clock: <ul style="list-style-type: none">• 1 = set clock• 0 = get clock data
	clock_buffer	Command string defining the desired clock output string. Only used on get clock data operations (i_type = 0). The command should have a combination of punctuation (spaces, commas, slashes, and so on) and command codes from the following table.

Clock		
Code	Data	Examples
A	2-digit year	1988 = 88
B	2-digit month	July = 07
C	2-digit date	1 through 31
D	2-digit hours	12 hour clock (that is, 10 A.M. = 10, 1 P.M. = 01)
E	2-digit minutes	00 through 59
F	2-digit seconds	00 through 59
G	A or P	A.M. = A P.M. = P
H	4-digit year	1987 = 1987
I	3-letter month	January = JAN
J	Full month name	January = JANUARY
K	2-digit hours	24-hour clock 10 A.M. = 10 1 P.M. = 13
L	3-letter day	Monday = MON
M	M	adds M to a.m. and p.m. (see G)
N	Full day	Monday = MONDAY

O	colon(:)	colon for time (8:30 or 14:22:13)
P	equal sign (=)	equal sign instead of colon (8=30 or 14=22=13)
Q	numeric day	Sunday = 0 Monday = 1 ... Saturday = 6

s_dest_buf

Destination buffer for the clock set string or clock output string. On a clock set operation this buffer should contain a string with the desired date/time to set the system clock in *yyyymmddhhmmss* format. On get clock data operations, this buffer is the destination for the formatted clock output string.

Return Values

Success: 1: No problems.

Failure: -1: Could not access clock.

clock2int()

Converts a date-time string to the equivalent integer values. The date-time string can be provided to the function or obtained by reading the clock device.

Prototype

```
#include <acldev.h>

int clock2int(int h_clock, unsigned char *buffer, int *year, int *month,
              int *day, int *hour, int *min, int *sec, int *wday);
```

Parameters

<code>h_clock</code>	Clock handle, used only when buffer is null.
<code>buffer</code>	Date-time string to convert. If null, current time is read. If date-time is provided in buffer, it must be in the format <i>yyyymmddhhmissw</i> , <ul style="list-style-type: none"> • where, <i>yyyy</i> is the year, • <i>mo</i> is the month, • <i>dd</i> is the day, • <i>hh</i> is the hour, • <i>mi</i> is the minutes, • <i>ss</i> is the seconds, and • <i>w</i> is the day of week. Day-of-week values range from 0 to 6, with Sunday being 0. If the date value provided in <code>buffer</code> is not properly formatted, the results are unpredictable.
<code>year</code>	Converted year.
<code>month</code>	Converted month.
<code>day</code>	Converted day.
<code>hour</code>	Converted hour.
<code>min</code>	Converted minutes.
<code>sec</code>	Converted seconds.
<code>wday</code>	Converted weekday.

Return Values

Success: > 0: Success.
Failure: -1: Error reading clock.

Dependencies

ACL, [strn2int\(\)](#).

Example

The linked example code file demonstrates use of `clock2int()`.

ctons1()

Converts a counted string to standard C-type null-terminated string. A counted string consists of a count byte followed by the string data. The count bytes value is the number of data characters in the string, including the count byte. For example, the word COUNT contains five characters. To create a counted string, a single byte of 6 is inserted in front of the C in COUNT making the total length of the string 6 bytes. This would be the equivalent of \006COUNT.

The resulting null-terminated string occupies the same space as the original counted string.

NOTE

It is important to remember that the length of a counted string is indicated by the length byte rather than a null at the end. One advantage of the counted string is that it contains data that can contain nulls. When using counted strings, be extremely careful about using C functions designed to operate on null-terminated strings.

The memory space for a counted string and a null-terminated string are the same. Allocate one byte for each data character, plus one byte for either the null or the count byte.

Prototype

```
#include <aclstr.h>
char *ctons(unsigned char *string);
```

Parameters

`string` String to convert.

Zero in `string[0]` is invalid. `ctons1()` treats this condition as an empty null-terminated string and does not modify the string.

If `string[0]` is 1, the counted string contains only the count byte. In this case, `string[0]` is set to 0 creating an empty null-terminated string.

Return Values

This function returns a pointer to the converted null-terminated string.

A counted string must have a minimum length of one (the count byte).

Dependencies

Verix eVo SDK

See Also

`ntocs()`

Example

The linked example code file demonstrates use of `ctons1()`.

cvt_ym()

Computes the total number of months from the year 0000 A.D. to the time given in a buffer in *yyyymm* format.

NOTE



The year must be between 1969 and 2068.

Prototype

```
#include <actutil.h>
int cvt_ym (char * ym);
```

Parameters

ym Year and month for conversion in *yyyymm* format.

Return Values

Success: Positive: Number of months.
Failure: Negative: -1 if date or year out of bounds.

delete_char()

Deletes a single character from a null-terminated string. The length of the string is decremented by one. The *del_pos* parameter specifies the characters offset into the string, with zero being the beginning of the string.

Prototype

```
#include <aclstr.h>
int delete_char(char *string, int del_pos);
```

Parameters

string String from which to delete a character.
del_pos Zero-based position of the character to delete.

Return Values

Success: ≥ 0 : Length of the modified string.
Failure: -1: *pos* was larger than the allowable string length or the position value was negative.

Dependencies

Verix eVo SDK

See Also

[insert_char\(\)](#), [DVL R Function Calls](#), [pad\(\)](#), [purge_char\(\)](#)

Example

The linked example code file demonstrates use of `delete_char()`.

display()

Accepts a pointer to a null-terminated string and writes it to the display without repositioning the cursor prior to the write, or clearing any portion of the display. At completion, the cursor is positioned after the last written character.

This is the equivalent of:

```
write(STDOUT, display_string, strlen (display_string))
```

Prototype

```
#include <aclconio.h>
int display(char *display_string);
```

Parameters

`display_string` Null-terminated string.

Return Values

Success: ≥ 0 : Number of characters actually written to the display. This return value is the actual return value from `write()`.

Failure: -1: An error occurred in the `write()` function.

Dependencies

Verix eVo SDK

See Also

[display_at\(\)](#)

Example

The linked example code file demonstrates use of `display()`.

display_at()

Repositions the cursor by column and line number before displaying a string and optionally, clears the display.

The cursor is repositioned using the `gotoxy()` function. Applications written for single-line displays should always specify line 1 as the line number; however, the `column` parameter can be useful to position displays.

Multiline applications should provide a line number appropriate for the intended platform. Refer to the programmers manual for the target terminal.

Prototype

```
#include <aclconio.h>

int display_at(unsigned int column, unsigned int line, char
               *display_string, unsigned int clear);
```

Parameters

<code>column</code>	Display column.
<code>line</code>	Display line position.
<code>display_string</code>	Null-terminated string.
<code>clear</code>	Clear display option.
The following three <code>clr</code> options are available and are defined in <code>ACLCONIO.H</code> :	
<ul style="list-style-type: none"> • <code>CLR_LINE</code>: Clears the entire line selected for the display operation. • <code>CLR_EOL</code>: Writes the display message, and then clears to the end of the line. • <code>NO_CLEAR</code>: Does not clear any portion of the display or any other value. 	

Return Values

Success:	≥ 0 : Number of characters actually written to the display. This return value is the actual return value from <code>write()</code> .
Failure:	-1: An error occurred in the <code>write()</code> function.

Dependencies

ACL [display\(\)](#)

NOTE



Column and line values are not checked to ensure they are appropriate for the terminal.

See Also

[display\(\)](#)

Example

The linked example code file demonstrates use of `display_at()`.

display_new()

Displays the message in column one of the current line, then clears the display to the end of the line. The message must be null-terminated.

Prototype

```
#include <aclconio.h>
int display_new (char *message);
```

Parameters

`message` Pointer to the buffer containing the null-terminated message to display.

Return Values

Positive number indicates the number of characters actually written to the display. This is the value returned from `write()`.

Success: 0:

Failure: -1: Indicates an error.

See Also [display\(\)](#), [display_at\(\)](#)

Example The linked example code file demonstrates use of `display_new()`.

do_compression()

Compresses or uncompresses a record using one of the following compression types: no compression, 6BIT, VFI, or BCD.

Prototype

```
#include <aclfile.h>
int do_compression (int mode, int compress_type, char *in_buf,
                    char *out_buf, int in_len);
```

Parameters

mode	Mode to execute:	
	<ul style="list-style-type: none">COMPRESS_RECORD; Size of compressed data.UNCOMPRESS_RECORD; Size of uncompressed data.	
compress_type	Type of compression:	
	NO_COMPRESSION	Copies in_buf to out_buf.
	COMPRESS_6BIT	Data compression operates on a subset of the ASCII character set (0x20–0x60). Lowercase characters are replaced with uppercase equivalents. Control characters (ASCII values less than 32 or greater than 127) are not preserved. This type of compression is not completely reversible.
	COMPRESS_VFI	Only digits 0–9 are compressed. Any non-numeric value is not compressed. Non-numeric values can be expanded (8-to-4 COMPRESSION); lowercase characters are replaced with uppercase equivalents.
	COMPRESS_BCD	Compresses (Binary Coded Decimal) digits 8-to-4. Non-numeric data is not compressed.
in_buf	Pointer to the record to compress or uncompress.	
	<ul style="list-style-type: none">mode = COMPRESS_RECORD, the buffer contains the record to compressmode = UNCOMPRESS_RECORD, the buffer contains the record to uncompress	
out_buf	Pointer to the buffer storing the compressed or uncompressed record.	
in_len	Length of the input record to compress or uncompress.	

Return Values

Success:	0:
Failure:	-1: Invalid input length.

Calling `do_compression()` with `NO_COMPRESSION` is the equivalent of `memcpy()`.

Ensure that `out_buf` is large enough to contain the compressed/uncompressed record. In both cases, `out_buf` is updated with the compressed/uncompressed record.

Using the `COMPRESS_VFI` compression type on character records results in a compressed record larger than the original (uncompressed) record (for example, "ABCD" = F41F42F43F44 compressed).

`COMPRESS_BCD` inserts a length count ahead of a compressed numeric string. Any alpha character remains the same.

Example The linked example code file demonstrates use of `do_compression()`.

dsp_strs()

Accepts one or more null-terminated strings and displays the string on the STDOUT device.

This routine calls the `write()` function once for each string parameter. The display is not cleared and the function uses the current line and column for the start of the display.

Prototype

```
#include <aclconio.h>
int dsp_strs(char * va_alist, ...);
```

Parameters

`va_alist` Argument prototype for display string(s).

Return Values

Success: ≥ 0 : Number of characters actually written to the display for the last string. This return value is the actual return value from the `write()` function.

Failure: -1: An error occurred in the `write()` function.

Dependencies

Verix eVo ACT Library `write()`, (see [page 157](#) for `write()` example).

CAUTION



The final argument must be a null. The behavior of the function is undefined if the parameters are not null-terminated strings or the trailing null is omitted.

Example

The linked example code file demonstrates use of `dsp_strs()`.

ERR_BEEP()

Activates the beep feature of the terminal. The function produces an *error* beep, that is lower in pitch than a *normal* beep.

Prototype

```
#include <aclconio.h>
void ERR_BEEP();
```

Return Values

The function returns before completion of the beep. No value is returned.

This function is implemented as a macro.

Dependencies

`error_tone()`, `normal_tone()`.

See Also

[beep\(\)](#), [NORM_BEEP\(\)](#)

Example

The linked example code file demonstrates use of `ERR_BEEP()`.

f_dollar()

Formats a null-terminated string as a dollar amount. Options include insertion of the dollar sign (\$), specification of decimal places, and separator characters.

NOTE



`f_dollar()` is not bound by an internally allocated buffer. The only string size limitation is that the data buffer must be large enough to accommodate the input data with all format characters.

On input, `*data` is the null-terminated string to be formatted. On output, `*data` is the corresponding dollar formatted string.

Prototype

```
#include <aclstr.h>

void f_dollar(char *data_ptr, int prec, int dol_flag, int dol_format);
```

Parameters

`data_ptr` Pointer to the I/O buffer.

`prec` Number of decimal digits.

`dol_flag` \$ format flag. Valid values are defined in ACLSTR.H as follows:

Value	Define	Explanation
• 0	DOLLAR_FMT_OFF	Input string unchanged; no formatting.
• 1	DOLLAR_FMT	Format without leading \$.
• 2	DOLLAR_IN_FMT	Format with leading \$.

`dol_format` Radix and separator format. Valid values are defined in ACLSTR.H as follows:

Value	Define	Radix	Separator
• 0	DOL_RDXPSEPN	period	none
• 1	DOL_RDXNSEPN	none	none
• 2	DOL_RDXPSEPC	period	comma
• 4	DOL_RDXCSEPN	comma	none
• 6	DOL_RDXCSEPP	comma	period

If `dol_format` is not equal to a valid equate, the default of DOL_RDXPSEPN is used.

Return Values None

Dependencies Verix eVo SDK

Verix eVo ACT Library [insert_char\(\)](#).

See Also [sprintf\(\)](#)

Example

The linked example code file demonstrates use of `f_dollar()`.

fieldcnt()

Copies the *n*th counted field from a source buffer specified in `buf`. `fieldcnt()` starts the first counted field at the position specified by `start` and moves down to the field specified by `count`. The string is then copied to the destination buffer.

Counted fields are defined by a count byte and a series of data bytes. The value of the count byte is the length of the data string plus the count byte. Counted fields can contain null characters.

Offset values begin at 0 for the first position in the source string. If the return value is 0 or -1, a null character is returned in the destination buffer.

Prototype

```
#include <aclstr.h>
int fieldcnt(char *buf, int start, int count, char *dest);
```

Parameters

`buf` Pointer to source buffer.

`start` Offset of counted fields.

`count` Field number to copy.

`dest` Destination buffer.

Return Values

Success: ≥ 0 : Actual number of characters copied.

Failure: -1: The field does not exist, a parameter was out of range, or a count byte was zero.

Dependencies

Verix eVo SDK

NOTE



IA null character is appended to the destination buffer. Since the counted field can also contain null bytes, the first null byte is not necessarily the end of the data. Verify the return value by the actual number of characters in the field.

If the source buffer is not properly formatted, the result is unpredictable.

See Also

`fieldfix()`, `fieldvar()`, `fieldray()`

Example

The linked example code file demonstrates use of `fieldcnt()`.

fieldfix()

Copies a data string from a source buffer to a destination buffer starting at an offset specified by `start` up to a fixed length specified by `size`, or to end of the source buffer.

If the function reaches the end of the buffer before copying all the characters indicated by `size`, the function performs a partial copy and returns a value less than the specified size value.

The function returns the actual number of characters being copied or an error code (-1). If the return value is minus one or zero, a null character is returned to the destination buffer; otherwise, a null character is added at the end of the destination buffer.

Prototype

```
#include <aclstr.h>
int fieldfix(char *buf, int start, int size, char *dest);
```

Parameters

<code>buf</code>	Source buffer.
<code>start</code>	Starting offset (0-based).
<code>size</code>	Number of characters to copy (1-based).
<code>dest</code>	Destination buffer.

Return Values

Success:	≥ 0 : Number of characters in destination buffer; zero if no characters were copied.
Failure:	-1: Start offset was beyond the data string length or the offset was negative.

Dependencies

Verix eVo SDK

NOTE



This function does not perform error checking. The destination buffer must be large enough to contain the field data to copy and a null character.

See Also

[fieldfix\(\)](#), [fieldvar\(\)](#), [fieldcnt\(\)](#)

Example

The linked example code file demonstrates use of `fieldfix()`. Also see the [EXFIELD.C](#) example program.

fieldray()

Copies a data string from a source buffer to a destination buffer. It starts at a fixed offset specified in `start` and copies up to a stop character specified in `stop`, or the end of the buffer. If the end of the buffer is reached before copying all the characters indicated by the delimiter, a partial copy is performed.

The function returns the actual number of characters being copied or an error code (-1). If the return value is -1 or zero, a null character is returned to the destination buffer. Otherwise, a null character is appended to the data in the destination buffer.

Prototype

```
#include <aclstr.h>
int fieldray(char *buf, int start, char stop, char *dest);
```

Parameters

<code>buf</code>	Source buffer.
<code>start</code>	Starting offset (0-based).
<code>stop</code>	Stop character at which copying terminates (this character is not copied to the destination buffer).
<code>dest</code>	Destination buffer.

Return Values

Success:	≥ 0 : Number of characters being copied.
Failure:	-1: Start offset was beyond or before the data string length.

Dependencies

Verix eVo SDK

NOTE



This function does not perform error checking. The destination buffer must be large enough to accommodate the data.

See Also

[fieldfix\(\)](#), [fieldvar\(\)](#), [fieldcnt\(\)](#)

Example

The linked example code file demonstrates use of `fieldray()`.

fieldvar()

Copies *n*th variable data field separated by a field delimiter. The first variable data field starts at the beginning of the source buffer and ends at the first encountered delimiter. If the end of the buffer is reached before copying all the characters indicated by the delimiter, a partial copy is performed.

The function returns the actual number of characters copied or an error code (-1). If the return value is minus one or zero, a null character is returned in the destination buffer. Otherwise, a null character is appended to the end of the destination buffer.

Prototype

```
#include <aclstr.h>
int fieldvar(char *buf, int fldnum, unsigned char sep, char *dest);
```

Parameters

buf	Buffer with message to scroll.
fldnum	Field number to copy (1-based).
sep	Field separator.
dest	Destination buffer.
inc	Scroll increments.
valid_keys	Valid key map (Refer Table 15 for valid keys)

Return Values

Success: ≥ 0 : Number of characters being copied; zero if no characters were copied.
Failure: -1: The field does not exist.

Dependencies

Verix eVo SDK
Verix eVo ACT Library [fieldray\(\)](#)

NOTE



This function does not perform error checking. The destination buffer must be large enough to contain the field data to copy, plus one null character.

See Also

[fieldfix\(\)](#), [fieldvar\(\)](#), [fieldcnt\(\)](#)

Example

The linked example code file demonstrates use of `fieldvar()`.

NOTE



When the parameter ending with a delimiter is passed to `fieldvar()` API, it assumes that there is another variable present and returns zero.

set_chars_per_key_value()

Gets the number of characters per key map from the user and sets the new value in the library.

User needs to call this API if the default characters per key (Five) provided by library does not suit their requirement. User needs to call this API before calling [getkbd_entry\(\)](#), [key_card_entry\(\)](#), and [SVC_KEY_TXT\(\)](#).

Once this API is called, the new value is retained for the life of the application. If the user wants to change to a different value or default value, this API needs to be called with a new value.

If the user calls this API with a value 7, the key map buffer should be as shown below:

```
char szKeyMap[MAX_ALPNUM_KEYS][7]={"0- +%_", "1QZ.\/", "2ABC&a",
"3DEF%d", "4GHI*g", "5JKL/j", "6MNO~m", "7PRS^p", "8TUV[t", "9WXY]w",
"*, \" : ", "#=: $?! " };
```

Users can define their own characters set in each array element. If the value for character per key is set to 7, an array `char szKeyMap[MAX_ALPNUM_KEYS][7]` has to be defined. Only 6 chars can be provided and one char is used for NULL.

It is the responsibility of the application to provide the array `szKeyMap[][CHAR PER KEY]` passed to the Verix eVo ACT library functions. The value of CHAR PER KEY can be modified, but the array should be in accordance with the modified value of CHAR PER KEY. If the value and the array size mismatches, then the results are unpredictable.

NOTE



`set_chars_per_key_value()` function allows the user to set any positive value. It gets set to default value (6), when 0 or any negative value is given.

Prototype

```
#include <aclconio.h>
void set_chars_per_key_value (short shCharPerKey)
```

Parameters

short shCharPerKey	Application-defined number of chars per key
--------------------	---

Return Values

None.

getkbd_entry()

Provides display and keyboard entry functions for a variety of input applications. This function combines the display, delay, key detection, data input, and data validation routines.

Prototype

```
#include <aclconio.h>

int getkbd_entry(int h_clock, char *msg, char *outbuf, unsigned wait,
                unsigned type, char szKeyMap[][CHAR_PER_KEY],
                int KeyMapSize, int va_alist, ...);
```

Parameters

h_clock	Clock device handle.
msg	Message to display.
outbuf	Where to hold the return data.
wait	Wait time, in 10-ms increments.
type	Type of data entry desired, valid values are:

- NUMERIC
- ALPHANUM
- NUMERIC|PASSWORD
- ALPHANUM|PASSWORD
- AMOUNT
- KEYMAP long va_alist

Variable arguments that depend on type:

- If type is NUMERIC, ALPHANUM, NUMERIC|PASSWORD or ALPHANUM_PASSWORD:

```
getkbd_entry(h_clock, msg, out_buf, wait, type, char
SzKeyMap[][CHAR_PER_KEY], int KeyMapSize, int max, int
min),
```

int max maximum number of digits (max= 9)

int min minimum number of digits (min=0)

- If type is KEYMAP:

```
getkbd_entry(h_clock, msg, out_buf, wait, type,
SzKeyMap, KeyMapSize, err_wait, err_msg, keys)
```

int err_wait = length of time to display err_msg, in half-second increments.

char *err_msg = error message displayed for invalid entries
long keys bitmap of the keys that can be entered.

- If type is AMOUNT:

```
getkbd_entry(h_clock, msg, out_buf, wait, type,
szKeyMap[][CHAR_PER_KEY], KeyMapSize, frac, max, min)
```

long max = maximum value allowed.

long min = minimum value allowed.

int frac =number of decimal places desired.

```
szKeyMap [ ]
{CHAR_PER_KEY}
```

Key map specifying the mapping of the logical alphanumeric keys to the physical keys on the keypad.

```
char szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY]=
{ "0- +%", "1QZ.\\", "2ABC&", "3DEF%", "4GHI*",
  "5JKL/", "6MNO~", "7PRS^", "8TUV[", "9WXY", "*, '\":",
  "#=: $?" };
```

For Vx680 terminal, the key map specifying the mapping of the logical alphanumeric keys to the physical keys on the keypad is:

```
char szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY_VX680]=
{ "0- +%", "1QZqz.\\", "2ABCabc&", "3DEFdef%",
  "4GHIghi*",
  "5JKLjkl/", "6MNOmno~", "7PRSprs^", "8TUVtuv[",
  "9WXYwxy]", "*, '\":",
  "#=: $?" }
```

By default, in the Verix eVo ACT library, CHAR_PER_KEY is defined as 6 and for Vx680 terminals, the CHAR_PER_KEY_VX680 is defined as 8. Hence the application can have five chars per key in the key-mapping array. User can modify this value by calling [set_chars_per_key_value\(\)](#). Based on the value set by the user, the szKeyMap array should be passed to the library APIs. There is no validation done on the array size by the library.

KeyMapSize

Size of the specified key map.

va_alist

One or more null-terminated strings to concatenate. The argument list is null-terminated (defined in `STDIO.H`).

Return Values

- Success: > 0: Length of string stored in out_buf.
- Failure: 0: Time out occurred.
- 1: Device read error.
 - 2: Parameter out of range.
 - 3: clear key pressed (not provided for mapped entry type).
 - 4: Invalid entry type specified.
 - 5: User pressed enter key when null data entry is permitted.

NOTE



All five input parameters are required. No parameter can be omitted from any call. The prompt can be suppressed by supplying a null message parameter. [getkbd_entry\(\)](#) will wait for the data entry without updating the display.

The display capability of this function uses [prompt\(\)](#) to display an optional message for n number of 10-ms increments, as specified by the wait parameter. During this display time, any keypress terminates the display and begins the data entry. Should the user fail to enter the data during the display period, the function

ends and returns zero. If the data entry fails, the content of the output buffer is unchanged. If an out-of-range amount is entered, [getkbd_entry\(\)](#) beeps and an invalid value remains displayed. The user can edit the entry using the backspace key. All data is returned as a null-terminated string.

NOTE

If the value of outbuf is F1-F4 or ALPHA key, a blank pixel is displayed when printed on the screen.

Data entry can be divided into three categories: standard entry, mapped key entry, and amount entry. The data entry category is specified by the `type` parameter and determines the syntax of the [getkbd_entry\(\)](#) call. #defines for the type options are supplied in `ACLCONIO.H`.

Standard Mode Data

Standard mode data entry is selected when `type` is `NUMERIC`, `ALPHANUM`, `NUMERIC | PASSWORD`, and `ALPHANUM | PASSWORD`.

This mode entry is suitable for entering account numbers, expiration dates, passwords, sequence numbers, and so on.

Standard entry options permit `ALPHANUM` and `NUMERIC` data entry in either clear mode—where each character is echoed to the screen as it is entered—or `PASSWORD` mode—where an asterisk (*) character is written to the screen for each character entered at the keyboard. This option uses [SVC_KEY_TXT\(\)](#).

Password type selection is accomplished by ORing either the `ALPHANUM` or `NUMERIC` options with the `PASSWORD` option. If the `min` parameter is zero, null data entry is permitted and the user can only press the enter key. If the `min` parameter is not zero and the user presses the enter key, [getkbd_entry\(\)](#) error beeps and starts another data entry time out.

Cell Phone Mode

The cell phone mode is specific to Vx680 terminals. This feature enables the user to key in the alpha numeric text without using the alpha keys, which are traditionally used in other terminals. Each key has the corresponding key value and alphabets. The user can click once to display the numeric value and click twice, thrice and four times to display capital letter of the respective keys. Fifth, sixth and seventh click displays those respective keys in small letters. The ninth click will display the special character. Refer to the [getkbd_entry\(\)](#) API for more details on key mapping.

The time out value for displaying the character simultaneously is set by #KEY_TIMER environmental variable. If the value of the environmental variable is greater than 20, then the variable value is set as 20. If the value of the environmental variable is less than 0, then 2 is set as the default value.

NOTE

Cell phone mode for keying in characters is supported only in Vx680 terminal.

Mapped Key Entry

Mapped key entry accepts a bitmapped long integer indicating the valid keys for the entry. This is useful to prompt the user with a display and restricting input to a limited number of keystrokes, such as Y/N entries.

The key map is created by ORing the values of the valid keys. #defines are provided in ACLCONIO.H for all standard Verix eVo terminal keys, as well as the screen-addressable keys.

Mapped key entry returns after the time out specified expires or a key contained in the key map is entered. A single key is accepted and its null-terminated value is placed in the output buffer.

Mapped key entry also accepts an error message and error display time. If an error message is supplied, a prompt message should also be provided. The error message is displayed for the specified number of half seconds after any invalid keypress. If data entry starts while the error message is displayed, the routine immediately reads and processes the new input. If no entry occurs during the error message display, the prompt message redisplay. This display restarts the time out, providing another timed entry period. The error message can also provide a secondary prompt, such as:

```
PromptMsg = ABORT TRANS?
```

```
Error Msg = 9= YES OR 6= NO
```

The following table lists the defines for the individual key definitions.

Table 15 **Key Definitions**

Key	Define
[0]	KM_0
[1]	KM_1
[2]	KM_2
[3]	KM_3
[4]	KM_4
[5]	KM_5
[6]	KM_6
[7]	KM_7
[8]	KM_8
[9]	KM_9

Table 15 **Key Definitions** (continued)

Key	Define
[*]	KM_STR
[#]	KM_PND
[CANCEL]	KM_ESC
[CLEAR]	KM_CLR
[BACKSPACE]	KM_BS
[ALPHA]	KM_AL
[FUNC/ ENTER]	KM_CR
[a]	KM_a: leftmost screen-addressable key
[b]	KM_b: second from the left screen-addressable key
[c]	KM_c: third from the left screen-addressable key
[d]	KM_d: fourth from the left screen-addressable key (additional screen-addressable keys)
[e]	KM_e: fifth screen-addressable key
[f]	KM_f: sixth screen-addressable key
[g]	KM_g: seventh screen-addressable key
[h]	KM_h: eighth screen-addressable key
[f0]	KM_f0:ninth screen-addressable key
[f5]	KM_f5:tenth screen-addressable key
All keys	KM_ALL
No keys	KM_NONE
Up Arrow key	KM_DOWN
Down Arrow key	KM_UP

Amount Mode Data

The entry amount is selected by `AMOUNT` type. Amount entry uses `SVC_KEY_NUM()` to accept numeric entries suitable for use in monetary-formatted amounts. `SVC_KEY_NUM()` restricts the maximum number of input characters to 15, and the maximum number of decimal places to 10. This routine further restricts input to those values that can be represented by a signed long value (-2147483648 to 2147483648), including decimal places. For example, 1000000 may represent:

- \$1,000,000: 0 decimal places
- \$100,000.0: 1 decimal place
- \$1.000000: 6 decimal places

Positive and negative values are permitted. The alpha key (if present) can toggle a “-” sign at the beginning of the data input. No checks are provided and overrange conditions are undefined.

The `frac` parameter allows the user to specify the number of decimal places included in formatting the input. `max` and `min` parameters must provide enough digits to permit the use of a specified number of decimal places. This is useful in international monetary environments and in applications using tax tables, gasoline prices, and so on.

When specifying negative `max` or `min` parameters, it is not possible to specify a negative value between 0 and -1. This is due to the inability to preserve leading zeros.

Formatting the amount on the display can be controlled by ORing any of the following values with the `AMOUNT` in the `type` parameter:

- `RDXPSEPN` radix decimal point and no separator: 1000000.00
- `RDXPSEPC` radix decimal point and comma separator: 1,000,000.00
- `RDXCSEPN` radix comma no separator: 1000000,00
- `RDXCSEPP` radix comma and decimal point separator: 1.000.000,00

If a format is not specified in the `type`, `RDXSEPN` is the default.

Example

```
getkbd_entry(h_clock, msg, output, 100, AMOUNT | RDXPSEPC,
             szKeyMap[][CHAR_PER_KEY], KeyMapSize, 2, 100000L, 100L);
```

Displays `msg` for up to one second using the amount input, a decimal point as radix, and a comma separator using two decimal places with 1,000.00, as the maximum value, and 1.00 as the minimum value.

`szKeyMap [] {CHAR_PER_KEY}` The key map specifying the mapping of the logical alphanumeric keys to the physical keys on the keypad.

Example

```
char szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY]= {"0- +%", "1QZ.\\",
"2ABC&", "3DEF%", "4GHI*", "5JKL/", "6MNO~", "7PRS^", "8TUV[ ", "9WXY]",
"*, '\\":", "#=:.$?" };
```

`KeyMapSize` Size of the specified key map.

Dependencies

Verix eVo SDK

[ACL prompt\(\)](#), [ctons1\(\)](#), [keyin_mapped\(\)](#), [ERR_BEEP\(\)](#), [keyin_amt_range\(\)](#), [set_chars_per_key_value\(\)](#)

NOTE



Only limited parameter verification is performed. If the wait parameter is zero, no time-out value is used. For `AMOUNT` type entry, `max` cannot be less than `min` and `frac` cannot be larger than the number of characters required to represent the larger of `max` or `min`. For `AMOUNT` type, values between zero and minus one cannot be specified, such as, 001 with a fractional value of 2.

A user entering mapped `type` data with a map passed with no bits set “on” (1), is unable to register any keypress. The routine continues to loop each time a user presses a key, and returns when the time out expires.

For mapped key entry, if an error message is specified but the error time parameter is zero, the message displays briefly and is then overwritten by the prompt (if provided).

See Also [prompt\(\)](#), [prompt_at\(\)](#), [display\(\)](#), [display_at\(\)](#), [keyin_mapped\(\)](#), [keyin_amt_range\(\)](#)

Example The linked example code file demonstrates use of [getkbd_entry\(\)](#). Also see the [EXTRANS.C](#) example program.

getxy()

Returns the x and y coordinate values of the cursor relative to the current window. The upper-left corner of the window is position 1,1. The x and y values provided by this function can be used in a [gotoxy\(\)](#) call to correctly position the cursor within a window. This function differs from [wherecur\(\)](#), which returns the physical position of the cursor. For example, if a window is defined using the coordinates 3,1, 10,1 and the cursor is in the “home” position, [getxy\(\)](#) returns 1,1, and [wherecur\(\)](#) returns 3,1.

Prototype

```
#include <aclconio.h>
int getxy(unsigned *x, unsigned *y);
```

Parameters

x x coordinate.
y y coordinate.

Return Values

Success: 0: Success.
Failure: -1: The current cursor position is outside the boundaries of the current window.

Dependencies Verix eVo SDK

See Also [window\(\)](#), [gotoxy\(\)](#), [wherecur\(\)](#)

Example The linked example code file demonstrates use of [getxy\(\)](#).

insert_char()

Inserts a character into a string at a specified position. This function permits passing a null as the insert character, which can be used to truncate a null-terminated string at specified position.

Prototype

```
#include <aclstr.h>
int insert_char(char *string, int pos, char c);
```

Parameters

<code>string</code>	Buffer containing the string.
<code>pos</code>	Position to insert character.
<code>c</code>	Character to insert.

Return Values

Success:	> 0: Length of the modified string.
Failure:	-1: Position was larger than string length, or position was negative or buffer is NULL.

NOTE

Bounds checking are not performed. The calling routine must ensure space is available for the inserted character(s).

See Also [DVLR Function Calls](#), [delete_char\(\)](#), [purge_char\(\)](#), [pad\(\)](#), [sputtf\(\)](#)

Example The linked example code file demonstrates use of `insert_char()`.

insert_decimal()

Inserts a decimal point at the third character from the right in a dollar amount string. The string is left padded with zeros if the length is less than three (for example, 1 is converted to 0.01).

Prototype

```
#include <aclstr.h>
int insert_decimal(char *buf);
```

Parameters

<code>buf</code>	Buffer to store the string to convert. String buffer must be large enough to hold result.
------------------	---

Return Values

Success:	0: Length of the converted string (positive number).
Failure:	-1: If the input buffer is null, the first character is not “-” or a numeric, or if any character is not numeric.

Example The linked example code file demonstrates use of `insert_decimal()`.

int2str()

Converts an integer value to an ASCII null-terminated string. This function assumes base 10 and does not support a radix, but does allow negative input values.

This function is based on the SVC_INT2 function in the C Standard library. The function converts the resulting counted string to a null-terminated string.

Prototype

```
#include <aclstr.h>
void int2str(char *dest, int val);
```

Parameters

dest Output buffer. The destination buffer must be large enough to hold the resulting characters and the null terminator (destination buffer plus two bytes).

val Value to convert.

Return Values

None.

Dependencies

Verix eVo SDK

Verix eVo ACT Library [ctons1\(\)](#), [insert_char\(\)](#)

NOTE

Overflow is undefined.

See Also

[strn2int\(\)](#), [long2str\(\)](#), [str2long\(\)](#)

Example

The linked example code file demonstrates use of `int2str()`.

julian_date()

Converts a specified date to a Julian calendar value. The Julian date is the day number with reference to January 1 of the specified year. This is useful in determining the number of days between one date entry and another. The macro [LEAP_YR\(\)](#) determines if the specified year is a leap year.

Prototype

```
#include <acldev.h>
#include <aclutil.h>
int julian_date(unsigned year, unsigned month, unsigned day);
```

Parameters

`year` Year number to convert.

`month` Month number to convert.

`day` Day number to convert.

Return Values

Success: > 0: Julian date for the specified month, day and year.

Failure: INVALID_TIME: month or day is invalid.

Dependencies

Verix eVo SDK, Verix eVo ACT Library [LEAP_YR\(\)](#)

See Also

[LEAP_YR\(\)](#), [month_end\(\)](#)

Example

The linked example code file demonstrates use of `julian_date()`.

KBHIT()

Uses `kbd_pending_count()` to determine the number of unprocessed key entries in the keyboard buffer. Key data is not read from the keyboard buffer. This function is implemented as a macro.

Prototype

```
#include <aclconio.h>
int KBHIT(void);
```

Return Values

Success: > 0: Number of keys in buffer.

Failure: 0: No keys in buffer.

See Also

`kbd_pending_count()`

Example

The linked example code file demonstrates use of `KBHIT()`.

key_card_entry()

Accepts the card-type data from either the keyboard or card reader. The following are conditions that must be met before calling `key_card_entry()`:

- The message file must be open.
- Limited parameter verification is performed.
- The input buffer must be large enough to hold the data.

Prototype

```
#include <message.h>

int key_card_entry(int h_clock, int h_card, char *data, unsigned int type,
                  unsigned int wait, unsigned int max, unsigned int min,
                  char *buffer, unsigned int message_num, char
                  szKeyMap[][CHAR_PER_KEY], int inKeyMapSize);
```

Parameters

<code>h_clock</code>	Clock handle.
<code>h_card</code>	Card reader handle.
<code>data</code>	Buffer for entered data.
<code>type</code>	Numeric or alphanumeric.
<code>wait</code>	Number of increments (10-ms increments).
<code>max</code>	Maximum keyboard entry allowed.
<code>min</code>	Minimum keyboard entry allowed.
<code>buffer</code>	Retrieved message or message to display.
<code>message_num</code>	Message number to display or 0 if there is a message in the buffer.
<code>szKeyMap [] [CHAR_PER_KEY]</code>	The key map specifying the mapping of the logical alphanumeric keys to the physical keys on the keypad. For example:

```
szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY]= { "0- +%", "1QZ.\\", "2ABC&",
"3DEF%", "4GHI*", "5JKL/", "6MNO~", "7PRS^", "8TUV[ ", "9WXY]",
"*, '\":", "#=: $? " };
```

By default, in Verix eVo ACT library, `CHAR_PER_KEY` is defined as 6. Hence the application can have five chars per key in the key-mapping array. User can modify this value by calling [set_chars_per_key_value\(\)](#). Based on the value set by the user, the `szKeyMap` array should be passed to the library APIs. There is no validation done on the array size by the library.

<code>KeyMapSize</code>	Size of the key map specified.
-------------------------	--------------------------------

Return Values

Success: Positive: Source of entry as indicated:

- 0x31 = keyboard
- 0x32 = swipe

Input is in the data buffer. If swiped, full track data returns. If manual entry, the user entry returns null-terminated. `max` and `min` parameters are not verified. For more details on MSR data format refer to the `read()` API, under Magnetic Card Reader section in *Verix eVo OS Programmers Manual*.

Failure

Negative:

- 0 = time out
- -2 = parameter out of range or invalid prompt number
- -4 = invalid entry type specified
- -1 = device read error
- -3 = clear key pressed
- -5 = the user pressed the enter key when null data entry is permitted

See Also `set_chars_per_key_value()`

keyin_amt_range()

Uses the `SVC_KEY_NUM()` routine to accept numeric entries suitable for use in monetary formatted amounts. Returns a null-terminated string to the callers buffer.

The `SVC_KEY_NUM()` routine restricts the maximum number of input characters to 15 and the maximum number of decimal places to 10. This routine further restricts input to values that can be represented by a signed long value (2147483647 to -2147483648), including decimal places. For example, 1000000 can represent:

- \$1,000,000: 0 decimal places
- \$100,000.0: 1 decimal place
- \$1.000000: 6 decimal places.

Positive and negative values are permitted. The alpha key on the terminal can toggle a - sign at the beginning of the data input. No checks are provided and overrange conditions are undefined.

The `frac` parameter allows the caller to specify the number of decimal places included in formatting the input. The `max` and `min` parameters must provide enough digits to permit the use of a specified number of decimal places. This is useful in international monetary environments and in applications using tax tables, gasoline prices, and so on.

When specifying negative `max` or `min` parameters, it is not possible to specify a negative value between zero and minus one. This is due to the inability to preserve leading zeros.

`#defines` are provided for the formatting of the data as follows:

- `RDXPSEPN` radix decimal point and no separator: 1000000.00
- `RDXPSEPC` radix decimal point and comma separator: 1,000,000.00
- `RDXCSEPN` radix comma no separator: 1000000,00
- `RDXCSEPP` radix comma and decimal point separator: 1.000.000,00

For example,

```
keyin_amt_range(output, RDXPSEPC, 100000L, 100L, 2);
```

Uses the decimal point as the radix, a comma separator, with an 1,000.00 maximum value and 1.00 minimum value, and uses two decimal places. The formatting selection applies to the data in the buffer after data entry. The user only sees the radix during data entry not the separator.

If a zero value is selected for the `frac` parameter, the radix is still displayed, but with no trailing characters.

`keyin_amt_range()` requires the user to press at least one numeric key and never returns a null string in `out_buf`.

NOTE



Only limited parameter verification is performed. `max` cannot be less than `min`, and `frac` cannot be larger than the number of characters required to represent the larger of `max` or `min`. Values between zero and minus one cannot be specified, such as, 001 with a fractional value of 2.

Prototype

```
#include <aclconio.h>

int keyin_amt_range(char *out_buf, int amt_fmt, long max, long min,
                    int frac);
```

Parameters

<code>out_buf</code>	Buffer to store the return data.
<code>amt_fmt</code>	Formatting specifier.
<code>max</code>	Maximum amount value allowed.
<code>min</code>	Minimum amount value allowed.
<code>frac</code>	Decimal places allowed.

Return Values

Success:	> 0: The length of the result buffer, including format characters.
Failure:	-1: Device read error.
	-2: Parameter out of range.
	-3: clear key pressed.

Dependencies

Verix eVo SDK, ACL: [prompt\(\)](#), [ctons1\(\)](#), [keyin_mapped\(\)](#), [ERR_BEEP\(\)](#)

See Also

[prompt\(\)](#), [prompt_at\(\)](#), [display\(\)](#), [display_at\(\)](#), [keyin_mapped\(\)](#)

Example

The linked example code file demonstrates use of `keyin_amt_range()`. See also the [EXTRANS.C](#) example program.

keyin_mapped()

Accepts a bitmapped long integer indicating valid keys for data entry. This is useful when prompting the user with a display and restricting input to a limited number of keystrokes, for example, Y/N entries. The key map is created by ORing the values of all the allowed keys. #defines are provided in ACLCONIO.H for all standard Verix eVo keys, as well as the screen-addressable keys.

Prototype

```
#include <aclconio.h>
char keyin_mapped(unsigned long key_map);
```

Parameters

key_map Keys allowed for entry.

Return Values

Success: > 0: Key code value of valid key entered.

Failure: 255: Error reading keyboard.

 251: Key press was not in the map.

This routine returns after each keypress, allowing the caller to take appropriate error processing actions as required by the application.

Dependencies

Standard TXO C Library

See Also

[keyin_amt_range\(\)](#)

Example

The linked example code file demonstrates use of `keyin_mapped()`.

LEAP_YR()

Determines if the specified year is a leap year. A leap year is defined as:

"Years evenly divisible by four but not evenly divisible by 100, or years evenly divisible by 400."

Prototype

```
#include <aclutil.h>
int LEAP_YR(int year);
```

Parameters

`year` Year to check.

Return Values

Success: 1: Leap year.

Failure: 0: Not a leap year.

Dependencies

Verix eVo SDK

See Also

[month_end\(\)](#)

Example

The linked example code file demonstrates use of LEAP_YR().

load_bmp()

Assists in building the ISO 8583 message packets (required in PIP applications) by loading a bitmap, message ID, and processing code from a formatted file. The file must be created using the TXOFIL utility (see [TXOFIL Utility](#)). [Table 16](#) lists the record, transaction code, transaction bitmap defines, message IDs, and processing codes.

Each line in the bitmap file is a VLR record.

h_file must be the handle to an open and properly formatted VLR file.

ISO 8583 hosts normally require bitmaps to be 8 bytes in length. The message ID and processing code are packed BCD values (where, "0200" converts to 0x02 0x00).

Table 16 **Codes**

Code	Definition
Record	
00–03 =	Bitmap index, message ID index, and processing code index.
04–06 =	ISO 8583 bitmaps.
07–08 =	ISO message IDs.
09–12 =	ISO processing codes.
Tran Codes	
00 Tran Code 0 =	"04,07,09" authorization only.
01 Tran Code 1 =	"05,08,09" sale/void.
02 Tran Code 2 =	"06,08,09" force online.
03 Tran Code 3 =	"06,08,11" refund.
Transaction Bitmap Defines	
04 Bitmap 1 =	"\x30\x20\x05\x80\x00\xc0\x00\x00" authorization only.
05 Bitmap 2 =	"\x30\x20\x05\x80\x00\xc0\x00\x00" sale.
06 Bitmap 3 =	"\x30\x38\x05\x80\x00\xc0\x00\x00" refund/force online.
Message IDs	
07 Message Id 1 =	"\x01\x00" authorization only.
08 Message Id 2 =	"\x02\x00" financial transaction.
Processing Codes	
09 Proc Code 1 =	"\x00\x40\x00" sale, offline sale, authorization.
10 Proc Code 2 =	"\x02\x40\x00" void DB, DB adjust.
11 Proc Code 3 =	"\x20\x40\x00" refund, offline refund.

Prototype

```
#include <aclutil.h>

int load_bmp (int h_file, UINT tran_code, BYTE *b_map, BYTE *msg_id, BYTE
               *p_code);
```

Parameters

<code>h_file</code>	Bitmap file handle.
<code>tran_code</code>	Transaction code. This is the index to the record that contains the record numbers for the bitmap, message ID, and processing code.
<code>b_map</code>	Pointer to the bitmap buffer.
<code>msg_id</code>	Pointer to the message ID buffer.
<code>p_code</code>	Pointer to the processing code buffer.

Return Values

Success:	≥ 0
Failure:	-1: File with <code>errno</code> containing the error code. -2: Bad index format.

Example

The linked example code file demonstrates use of `load_bmp()`.

long2str()

Converts a signed long integer to a string. If the long value is negative, a minus sign is placed at the beginning of the string. The function converts 2147483647 as the largest possible positive integer before it goes negative (four bytes). For example, the large positive integer 21234567890 is converted to a corresponding negative string as -2103731502. The large negative integer 11234567890 is converted to a corresponding string as 1650333998.

Prototype

```
#include <aclstr.h>
void long2str(char *dest, long val);
```

Parameters

dest Destination buffer address.
val Long integer to convert.

Return Values

None.

NOTE

No error checking is performed. This function assumes a long val of four bytes. The value 0x80000000 is forced to -2147483648.

See Also

strn2int(), str2long()

Example

The linked example code file demonstrates use of long2str().

MAX()

Returns the greater of two values. This function is implemented as a macro, and as such, the data types of val1 and val2 are determined by the caller. The caller must ensure that val1 and val2 are compatible data types. The macro expands to:

```
((val1) > (val2)) ? (val1) : (val2))
```

Because of the implementation as a macro, take care when passing parameters that are actually expressions that modify variables (for example, i++). These expressions may be evaluated twice.

Prototype

```
#include <aclutil.h>  
MAX(val1, val2);
```

Parameters

val1 First value to compare.
val2 Second value to compare.

Return Values

Success: The larger of the two values.

Dependencies

Verix eVo SDK

See Also

[act_kbd_pending_test\(\)](#)

Example

The linked example code file demonstrates use of MAX().

MEMCLR()

Prototype `#include <aclutil.h>`
 `char *MEMCLR(char *buf, unsigned size);`

Parameters

<code>buf</code>	Area to clear.
<code>size</code>	Size of the buffer.

Return Values Void pointer to the memory block cleared.

Dependencies Verix eVo SDK

See Also `memset()`

Example The linked example code file demonstrates use of MEMCLR(). See also the [EXTRANS.C](#) example program.

msg_display_at()

Retrieves the specified message from the message file, repositions the cursor by column and line number prior to displaying the string, and optionally clears the display. The following are conditions that must be met before calling `msg_display_at()`:

- The message file must be open.
- The buffer must be large enough to hold the retrieved message.
- Column and line values are not verified to ensure they are appropriate for the terminal.

Prototype

```
#include <message.h>

int msg_display_at(unsigned int column, unsigned int line, unsigned int
                  message_num, char *buffer, unsigned int clr);
```

Parameters

<code>column</code>	The column to display at.
<code>line</code>	The line to display on.
<code>message_num</code>	The message number to display.
<code>buffer</code>	Buffer to store retrieved message.
<code>clr</code>	Valid values are: <ul style="list-style-type: none"> • <code>CLR_LINE</code> • <code>CLR_EOL</code> • <code>NO_CLEAR</code>

Return Values

Success:	Positive: The number of characters actually written to the display. This is the value returned from <code>write()</code> .
Failure:	Negative: -1 on error; -2 message file not open or message not found

See Also `display_at()`

msg_display_new()

Retrieves the specified message from the message file, repositions the cursor by column and line number prior to displaying the string, and optionally clears the display. The following are conditions that must be met before calling msg_display_new():

- The message file must be open.
- The buffer must be large enough to hold the retrieved message.
- Uses msg_display_at().

msg_display_new() is prototyped in message.h

Prototype

```
#include <message.h>
int msg_display_new(unsigned int message_num, char *buffer);
```

Parameters

message_num	The message number to display.
buffer	Buffer to store retrieved message.

Return Values

Success:	Positive: The number of characters actually written to the display. This is the value returned from write().
Failure:	Negative: -1 on write error; -2 message file not open.

MIN()

Returns the lesser of two values. This function is implemented as a macro and, as such, the data types of val1 and val2 are determined by the caller. The caller must ensure that val1 and val2 are compatible data types. The macro expands to:

```
((val1) < (val2)) ? (val1) : (val2))
```

Because of the implementation as a macro, take care when passing parameters that are actually expressions that modify variables (for example, `i++`). These expressions may be evaluated twice.

Prototype

```
#include <aclutil.h>  
MIN(val1, val2);
```

Parameters

val1 First value to compare.
val2 Second value to compare.

Return Values

The lesser of the two values.

Dependencies

Verix eVo SDK

See Also

[MAX\(\)](#)

Example

The linked example code file demonstrates use of MIN().

month_end()

Returns the number of days in the month for the specified year. Use the macro [LEAP_YR\(\)](#) to determine if the specified year is a leap year.

Prototype

```
#include <aclutil.h>
int month_end(int month, int year);
```

Parameters

month	Requested month.
year	Requested year.

Return Values

Success:	The number of days in the month for the indicated year.
Failure:	-1 = Invalid month value (out of range).

Dependencies Verix eVo ACT Library [LEAP_YR\(\)](#)

See Also [LEAP_YR\(\)](#)

Example The linked example code file demonstrates use of month_end().

mult_strcat()

Concatenates multiple strings of text in the destination buffer.

NOTE



The destination buffer must be large enough to hold the result.

Prototype

```
#include <aclstr.h>
int mult_strcat(BYTE *outbuf, char va_alist, ...);
```

Parameters

outbuf	Pointer to the destination buffer storing the concatenated string.
va_alist	One or more null-terminated strings to concatenate. The argument list is null-terminated (defined in <code>STDIO.H</code>).

Return Values

Success:	Number of characters stored in the destination buffer.
	0: No strings concatenated.

Example The linked example code file demonstrates use of mult_strcat().

NORM_BEEP()

Activates the beep feature of the system. This produces a beep higher in tone than an error beep.

This function is implemented as a macro; it returns before completion of the beep.

Prototype

```
#include <aclconio.h>
void NORM_BEEP(void);
```

Return Values None.

Dependencies normal_tone()

See Also beep(), ERR_BEEP()

Example The linked example code file demonstrates use of NORM_BEEP().

ntocs()

Converts standard C-type null-terminated strings to counted strings commonly used in C.

NOTE



A counted string cannot contain more than 254 characters. This restriction is required since, the count must be the first byte of the string and the maximum value for a single byte is 255. The count byte is included in the count. The destination buffer must be at least as long as the source buffer.

Prototype

```
#include <aclstr.h>
int ntocs(char *dest_buf, char *src_buf);
```

Parameters

dest_buf	Address to store counted string.
src_buf	Null-terminated string address.

Return Values

Success:	String converted.
Failure:	-1: The source or destination buffer is null.

Dependencies Verix eVo SDK

See Also ctons1()

Example The linked example code file demonstrates use of ntocs().

p_set_baudformat()

Initializes the specified communications port for printing at a specified baud rate and data format. Accepts a device handle to the printer port, the baud rate, and the data format as parameters. Printer characteristics automatically initialized include: character mode, auto-enable, flow control, and RTS assertion. The following are conditions that must be met before calling p_set_baudformat:

- The caller must open() the communications port.
- Valid baud rates are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

If an invalid rate is specified, the rate defaults to 9600.

- Valid data formats are: 7E1 7N1, 7O1, 8E1, 8N1 and 8O1.

If an invalid format is specified, format defaults to 7E1.

Prototype

```
#include <acldev.h>
int p_set_baudformat(int h_comm_port, int baud_rate, char *data_format);
```

Parameters

h_comm_port Handle for printer port.
baud_rate Baud rate.
data_format Data format.

Return Values

Success: 0
Failure: -1 on error.

Dependencies

Verix eVo SDK

Example

The linked example code file demonstrates use of p_set_baudformat().

pad()

Accepts a null-terminated string (source) and adds characters as required to produce a null-terminated destination string of the length specified by the call. The location of the source in the destination (left, center, or right) is controlled through the `align` parameter. The pad character is specified by the caller, and can be any value, including null.

The destination string is an exact duplicate of the source string if the source string is equal to or longer than the desired length, or the length specified is negative or zero. `pad()` does not truncate the source string to the specified pad length.

The source and destination buffers can be the same buffer. In this case, the source contents are altered; otherwise, the source buffer contents are unchanged.

NOTE



The caller must ensure that the destination buffer is the greatest of the two pad length or source buffer. No bounds checking is performed. Passing null as the `pad_char` parameter results in the destination buffer being filled with `pad_size` null characters. The previous contents of the destination buffer are destroyed.

Prototype

```
#include <aclstr.h>

int pad(char *pdest_buf, char *psrc_buf, char pad_char, int pad_size,
        int align);
```

Parameters

pdest_buf	Stores the padded string.	
psrc_buf	Source string to pad.	
pad_char	Pad with this character.	
pad_size	Size of the padded string.	
align	Indicates the position of the source data in the destination string. Valid values are:	
Align at:	Effect	#define (ACLSTR.H)
• 0x00	Source at the beginning	LEFT or LEFTJUSTIFY
• 0x80	Source at the end	RIGHT or RIGHTJUSTIFY
• 0x88	Source centered	CENTER or CENTERJUSTIFY
• other value	Source at the beginning	

Return Values

Success:	> 0: The number of characters added to the source string to produce the destination string.
Failure:	0: The source length was greater than or equal to the pad length, or the pad length was negative.

Dependencies

Verix eVo SDK

See Also

[DVL R Function Calls](#), [insert_char\(\)](#), [delete_char\(\)](#)

Example

The linked example code file demonstrates use of `pad()`.

pause()

Waits a designated number of 10-ms intervals. A keypress terminates the function. If the `time` parameter is zero, the function does not pause and returns immediately with a zero return value. `pause()` runs until either the time interval has expired or a keypress is detected. If a keypress interrupts `pause()`, the key value is read and discarded.

Prototype

```
#include <aclconio.h>
int pause(unsigned time);
```

Parameters

`time` Number of 10-ms intervals to pause.

Return Values

Success: 0: Pause interval successfully expired.
Failure: 1: Keypress occurred before designated pause interval expired.
 -1: Keypress error condition.

Dependencies Verix eVo SDK

See Also [SLEEP\(\)](#)

Example The linked example code file demonstrates use of `pause()`.

prompt()

Displays a null-terminated string at the current line, column 1, for a specified duration or until a key is pressed, whichever occurs first. The key pressed remains in the keyboard buffer and can be read on return from `prompt()`.

The `opt` parameter determines how the display is cleared before writing the `display_string` on the display. The `wait` parameter specifies the number of 10-ms increments to display the prompt message.

Prototype

```
#include <aclconio.h>

int prompt(int h_clk, char *display_string, unsigned int wait,
           unsigned int opt);
```

Parameters

<code>h_clk</code>	Handle to clock.
<code>display_string</code>	String to display.
<code>wait</code>	Wait time in 10-ms increments.
<code>opt</code>	Clear display option. The clear options, defined in <code>ACLCONIO.H</code> , determine how the display clears before writing the prompt message. Valid clear options are: <ul style="list-style-type: none"> • <code>CLR_LINE</code>: Completely clears the current line, but does not change the cursor position. • <code>CLR_EOL</code>: Clears from the end of message to the end of the line. • <code>NO_CLEAR</code>: Writes to the display without clearing.

Return Values

Success:	0: Time expired without a keypress.
Failure:	1: A key was pressed before the time out.

Dependencies

Verix eVo SDK
ACL [prompt_at\(\)](#)

See Also [prompt_at\(\)](#), [display\(\)](#), [display_at\(\)](#)

Example The linked example code file demonstrates use of `prompt()`.

prompt_at()

Displays a null-terminated string at the specified column and line for a specified duration or until a key is pressed, whichever occurs first. The key pressed remains in the keyboard buffer and can be read on return from `prompt_at()`.

The `opt` parameter determines how the display is cleared before writing the `msg` string to the display. The `wait` parameter specifies the number of 10-ms increments to display the prompt message.

Prototype

```
#include <aclconio.h>

int prompt_at(int h_clock, unsigned col, unsigned lin, char *msg,
              unsigned wait, unsigned opt);
```

Parameters

<code>h_clock</code>	Clock handle.
<code>col</code>	Display column location.
<code>lin</code>	Display line location.
<code>msg</code>	Message to display.
<code>wait</code>	Display time in 10-ms increments.
<code>opt</code>	Clear display option. The clear options, defined in <code>ACLCONIO.H</code> , determine how the display will clear before writing the prompt message. Valid clear options are: <ul style="list-style-type: none"> • <code>CLR_LINE</code>: Completely clears the line specified. • <code>CLR_EOL</code>: Clears from the end of message to the end of the line. • <code>NO_CLEAR</code>: Writes to the display without clearing.

Return Values

Success:	0: Time expired without a keypress.
Failure:	1: A key was pressed before the time out occurred.

Dependencies

Verix eVo SDK
Verix eVo ACT Library [display_at\(\)](#)

See Also [prompt\(\)](#), [display\(\)](#), [display_at\(\)](#)

Example

See also the [EXTRANS.C](#) example program. The linked example code file demonstrates use of `prompt_at()`.

purge_char()

Removes all occurrences of a specified character from a null-terminated string. The target string is not modified if it is empty, contains no purge characters, or the purge character is null.

NOTE



This function does *not* purge null characters.

Prototype

```
#include <aclstr.h>
int purge_char(char *buffer, char rem_char);
```

Parameters

<code>buffer</code>	Buffer to remove character from.
<code>rem_char</code>	Character to remove.

Return Values

Success: Number of characters deleted from the target string.

Dependencies

Verix eVo SDK

See Also

[DVLR Function Calls](#), [insert_char\(\)](#), [pad\(\)](#)

Example

The linked example code file demonstrates use of `purge_char()`.

range()

Searches a table stored in a keyed file. `range()` operates on a specified file passed as an input parameter. This file is assumed to be a keyed file, stored as compressed records. Each record consists of a key value and associated key data.

Each record within the file can be uniquely identified by a key value. The key value is created from the concatenation of up to four characters, plus a one- or two-digit integer corresponding to a record number.

`range()` assumes the associated key value data within the file is formatted to create a range table as follows:

```
<minimum range> <maximum range> <range data>
```

Each record of the file has a minimum and maximum range field, and associated range data, separated with blanks. The maximum length of each cannot exceed 20 characters.

`range()` provides access into the range table. The programmer specifies a value to be *range*d or compared. The range function accesses the range table and compares the minimum and maximum range fields in search of a matching record. Should the value fall within the minimum and maximum ranges, the associated range data is stored and the record in which the match is found returns.

NOTE



The file consists of CVLRs. Compression is intended to operate on a subset of the ASCII character set. All numeric characters (range 30h–39h) are stored using only four bits. All supported non-numeric characters (ranges 00h–02Fh and 3Ah–5Fh) are stored using eight bits (that is, without compression). All other characters (range 60h–FFh) are stored in eight bits and corrupted. Ensure that such data is not included in the file records.

Prototype

```
#include <aclstr.h>
int range(RANGE_PARMS *range_data);
```

Parameters

`range_data` Range parameters defining the table and search criteria.

Return Values

Success: 0: The label length is greater than 4 or less than 1, or the start record is greater than 99 or less than 1.
> 0: The record accessed in which a range match was found.

Failure: -1: Last record accessed at the time in which processing is complete and a match has not been found— this is the equivalent of one greater than the last record number of the file if the entire file is searched.

RANGE_PARMS Structure

All input parameters are passed in a structure of type RANGE_PARMS, as defined in ACLSTR.H.

```
typedef struct
{
    char *acct_num;
    /* 'range' or 'account number' searching for */
    char *label;
    /* Table record label identifier; Maximum of 4 characters. This string is
    concatenated with the offset record number to a unique record identifier
    into the table (label_entry_id). */
    char *data_fld;
    /* Place in which to store found range data. Passed as null if no data is
    to be stored. */

    int start;
    /* Offset record number in which to begin range search. This integer is
    concatenated with the label to create a unique record identifier into the
    table (label_entry_id). */
    char *file_name;
    /* File in which the range table exists. */
} RANGE_PARMS;
```

Dependencies Verix eVo SDK, getkey(), strcmp(), strncmp(), strcpy()

Example The linked example code file demonstrates use of range().

range_vlr()

Searches a VLR file for a record that bounds the given account number. range_vlr() operates on the specified file passed as an input parameter. This parameter is assumed to be a VLR file, and each record consists of data as follows:

NOTE



range_vlr() does not perform error checking and the lengths of the range numbers in the records become the lengths of the comparison.

```
<minimum range_fld><SPACE><maximum range_fld><SPACE><range data>
```

Prototype

```
#include <aclfile.h>

int range_vlr (char *acct, int start, char *data, char *file_name);
```

Parameters

acct	Range or account number being searching for.
start	Offset record number at which to begin the range search.
data	Buffer to store found range data. This is passed as null if there is no data to store.
file_name	File where the range table exists.

Return Values

Success: 0: Range not found.

Failure: -1: Start record number is < 1 or the account is null.

-2: File cannot be opened; `errno` is intact.

> 0: The positive value of the record accessed in which a range match was found.

scroll_buffer()

Displays a message in the current display window allowing the user to scroll through the message using application-defined keys. Scrolling occurs at increments specified by the calling function.

The initial display position of the message is specified in the `scroll_offset` parameter. The amount of text to scroll is specified in the `inc` parameter.

Use the calling routine to specify the key(s) to terminate the display. One of these keys can be specified as a special exit key.

`#defines` for constructing the key map are included in `ACLCONIO.H`. To accept one or more keys to terminate the display, OR the desired key values together. If the exit key is selected, set `key_buf[0]`.

NOTE



If the value of `key_buf` is F1-F4 or ALPHA key, a blank pixel is displayed when printed on the screen.

Prototype

```
#include <aclconio.h>

int scroll_buffer(char *buf, int inc, unsigned long valid_keys,
                unsigned scroll_offset, char scroll_left,
                char scroll_right, char exit_key, char *key_buf);
```

Parameters

<code>buf</code>	Buffer with message to scroll.
<code>inc</code>	Scroll increments.
<code>valid_keys</code>	Valid key map. (Refer Table 15 for valid keys.)
<code>scroll_offset</code>	Initial display position in scroll increments (1 based).
<code>scroll_left</code>	Scroll left key.
<code>scroll_right</code>	Scroll right key.
<code>exit_key</code>	Exit key.
<code>key_buf</code>	Buffer to hold key used to terminate the display.

Return Values

Success: Message position, based on `inc`, when the user terminated the display.

Example

The linked example code file demonstrates use of `scroll_buffer()`.

set_itimeout()

Sets an interval timer based on the 1/64-second system tick clock. Used in conjunction with [CHK_TIMEOUT\(\)](#). The time values are approximate.

Rollover is not processed by this routine. The system tick count is reset each time the system powers up or on exit from system mode. This results in a rollover after more than two years of continuous operation.

Prototype

```
#include <alcdev.h>

unsigned long set_itimeout(int h_clock, unsigned int time,
                          unsigned long gradient);
```

Parameters

<code>h_clock</code>	Clock device handle.
<code>time</code>	Number intervals for time out.
<code>gradient</code>	Timeout units.

Return Values

Success:	> 0: Current system tick value + (time * interval), where, interval is TM_TICKS, TM_SECONDS, or TM_MINUTES, as defined in ACLDEV.H.
Failure:	INVALID_TIME (0): Invalid interval.

See Also [CHK_TIMEOUT\(\)](#)

Example See also the [EXTMOUT.C](#) programming example. The linked example code file demonstrates use of [set_itimeout\(\)](#).

sgetf()

Compares characters of an input string to a control string. `sgetf()` is the modified version of the standard C routine `sscanf()`. The `sgetf()` version offers advantages of being much smaller than its standard C counterpart. `sgetf()` does not however, support floating point and some unsigned options.

Parsing terminates if any of the following occurs:

- the end of the control string is reached,
- the end of the input string is reached, or
- a conversion mismatch of the input and control strings.

Prototype

```
#include <aclstr.h>
```

```
char sgetf(char *ss, char *cs, char *args, ...);
```

Parameters

<code>ss</code>	Pointer input string.
<code>cs</code>	Pointer control string. The control string can contain format directives or literals as follows:
Flags	<p>Flags act as modifiers to the format directive and control how the data is extracted. All flags are optional.</p> <ul style="list-style-type: none"> • *: Overrides storage assignment of this value. Without the "*" option, data extracted from the input string is stored in the corresponding address parameter. If the "*" is included in the control string, data extracted from the input string is not stored in an address in the parameter list. • \$: Applies only to integer and long conversions. This flag causes the parser to ignore a single instance of "." or "," in the input string. Example: If the input string is "65.11" and the control string is "%\$i", then 6511 is stored.
Width	<p>Numeric value that specifies the maximum field width to extract. This can be used in conjunction with the i, l, b, c, s types. Note that if zero is specified as the maximum field width, at least one character of the input string is extracted.</p>
Literals	<p>Literals can be intermixed with directives in the control string. There is a one-to-one correlation of a literal in the control string to a literal in the input string, with one exception, the space character. A single space character as a literal in the control string corresponds to any number of blank spaces in the data string.</p>
Directives	<p>All format directives are preceded by a percent ("%") sign. Directives are handled serially, so that the first directive starts at the beginning of the string and the second immediately following it. Standard directives follow the format:</p> <pre>%[flags][width]<type></pre>

Type This mandatory field specifies what kind of data is expected in the input string and how it should be assigned. Supported values are:

- **i** = Decimal integer: Parsing begins at the first character of the input string and ends on the first non-numeric. Exception: the "\$" flag allows a single radix ("." or ",") to be included in the numeric string.
- **l** = Decimal long integer: Parsing begins at the first character of input string and ends on the first non-numeric. Exception: the "\$" flag allows inclusion of a single radix ("." or ",") in the numeric string.
- **c** = Character: Parsing begins at the first character in the input string. Example: If the input string is "987", then " " is stored.
- **s** = String: Parsing begins at the first non-space character of the input string and ends at the first space or null character. Example: If the input string is "987", then "987" is stored as a string (null-terminated).
- **[]** = String: Characters specified between brackets in the control string describe the set of characters in the input string to be assigned as a string (null-terminated). The first character found in the data string that is not part of the bracketed set terminates the string. This option differs from the **s** option in that the string parsing is not terminated by a space.
- **[1–9]** A range of characters can be specified within the brackets by putting a hyphen ("-") character between the lower and upper limits. (The first character must have a lower ASCII value than the second.)
- **[^]**: Alternatively, bracketed characters can be used as a set of exclusion characters to terminate the string. This is specified by placing a caret ("^") in the first position following the open bracket. All characters are copied from the input string to the assigned string-up until the point at which one of these exclusion characters or null terminator is encountered.

- Examples
- If the input string is "123AB45," then the control string "%[1-8]" extracts the string "123". Parsing terminates at the character "A".
 - If the input string is "123AB45," then the control string "%[A-Z]" extracts the string "AB".
 - If the input string is "<STX>12AB78<ETX> <LRC>," then the control string "%\002%[\^003]" assigns "12AB78" to a string.

args Variable number of pointers to results of format conversion.

Return Values

Success: Pointer to the last character parsed in the input string. The return value can be used as the input string pointer for a subsequent call.

NOTE

`errno` is initialized to zero at beginning of routine and set to `EINVAL` if:

- a syntax error occurs
- input the string does not match control string, or
- the end of the input string is reached and is not finished parsing the control string.

Dependencies

Verix eVo SDK

See Also

`sprintf()`

Example

The linked example code file demonstrates use of `sgetf()`.

SLEEP()

Causes program execution to be suspended by the specified number of 10-ms increments. This function is implemented as a macro utilizing the `SVC_WAIT()` function.

The maximum time value allowed is one minute. Program suspension is not interrupted by a keypress or other event. To perform program suspension use [pause\(\)](#), which is terminated by a keypress.

`SVC_WAIT()` executes a foreground loop until the approximate number of milliseconds specified in `time` elapse.

Prototype

```
#include <acldev.h>
void SLEEP(unsigned int time);
```

Parameters

`time` Sleep time in 10-ms increments.

Return Values

None.

Dependencies

Verix eVo SDK

See Also

`SVC_WAIT()`, [pause\(\)](#)

Example

The linked example code file demonstrates use of [SLEEP\(\)](#).

sprintf()

Modified version of the standard C routine `sprintf()`. However, it does not support all standard `sprintf()` directives.

Control strings and control string literals are handled in a similar manner as the standard `sprintf()`.

NOTE

`sprintf()` is not bound by an internally allocated buffer. The only string size limitation is that the result buffer must be large enough to accommodate all expanded format characters.

The following directives are supported:

- i integer
- u unsigned
- c character
- s string

Flags are also handled in the same manner as `sprintf()`, with the exception of the zero flag.

The standard `sprintf()` function uses the zero flag to specify that a 0 is used as the pad character should the output value contain fewer digits than the minimum field width. This zero flag is not supported in the ACL `sprintf()`. `sprintf()` does, however, support the same functionality through the standard minimum field width parameter. If the minimum field width parameter contains a leading zero, it is interpreted as the 0 flag, wherein zero is used as the pad character should the output value contain fewer digits than the minimum field width.

- = Center justify flag can be used in addition to left justify. For example, `sprintf(result, "%=10s", "hello");` accomplishes the following:
 result: " hello "
- + Sign flag. This flag immediately follows the justification flag (-, =), and proceeds the minimum field width indicator. The + flag causes the output to contain a + if the input value is positive and a - if the input value is negative.

Examples:

```
long l = 1234567L;
sprintf(buff, "%-+10.2l$0i", (char *)&l); or
sprintf(buff, "%=+10.2l$0i", (char *)&l);
result: $+1234.56
change long l = -1234567L;
sprintf(result, "%-+10.2l$0i", (char *)&l); or
sprintf(buff, "%=+10.2l$0i", (char *)&l);
result: $-1234.56
```

[\$] n Dollar formatting. The position of the \$ flag within the control string is important. Dollar formatting must immediately proceed the i, u, c, or s directive. * can be used in place of the n format value; the next parameter in the input list is then used as the n format value, as follows:

n Format Values			
Value	Define	Radix	Separator
0	DOL_RDXPSEPN	period	none
1	DOL_RDXNSEPN	none	none
2	DOL_RDXCSEPP	comma	period
4	DOL_RDXCSEPN	comma	none
6	DOL_RDXPSEPC	period	comma

Definition: n specifies type of separator formatting.

Radix refers to the character separating whole numbers from decimal digits.

Separator refers to the character separating thousands digits.

As with standard `sprintf`, “*” designates the next parameter in the input list as the format variable. `sputf()` uses the same convention when specifying minimum field width, precision, and dollar format.

Prototype

```
#include <aclstr.h>
char sputf(char *result_store, char *next_cs, char *args, ...);
```

Parameters

<code>result_store</code>	Control string specifying formatting to occur.
<code>next_cs</code>	Variable length list of pointers to control string.
<code>args</code>	Parameters and input data.

Return Values

Success: The length of the result buffer.

Failure: Unpredictable value if error encountered.

Dependencies

Verix eVo SDK, STDIO.H, CTYPE.H, ACL: [pad\(\)](#), [ctons1\(\)](#), [long2str\(\)](#), [insert_char\(\)](#), [int2str\(\)](#), [f_dollar\(\)](#)

See Also

[sprintf\(\)](#), [printf\(\)](#), [f_dollar\(\)](#)

Example

The linked example code file demonstrates use of `sputf()`.

strn2int()

Converts `cnt` bytes of `buffer` to an integer value. Non-numeric characters in `buffer` are included in the count of bytes to convert, but are ignored during the conversion.

Prototype

```
#include <aclstr.h>
int strn2int(char *buffer, int cnt);
```

Parameters

<code>buffer</code>	Data to convert to an integer value; maximum 40 characters.
<code>cnt</code>	Number of bytes to convert.

Return Values

Equivalent integer value of the string. If `cnt` is greater than 40 or less than 0, `strn2int()` returns 0.

Failure: 0: `cnt > 40` or no numeric characters in `cnt` bytes of `buffer`.

Dependencies

Verix eVo SDK, ACL [str2int\(\)](#)

Example

The linked example code file demonstrates use of [strn2int\(\)](#).

str2digit()

Accepts a string and purges all characters that are not digits. If a minus sign ("-") is found in the first character position of the string, it is retained in the resulting string.

NOTE



The original string passed is destroyed; all non-digit characters are purged.

This routine is used in the `str2long()` function.

Prototype

```
#include <aclstr.h>
int str2digit(char *source);
```

Parameters

`source` Pointer to null-terminated string to process.

Return Values

Success: Number of digit characters in the converted string not including the optional "-" sign flag.

Failure: -1: String is empty.

Dependencies

Verix eVo SDK

See Also

[purge_char\(\)](#)

Example

The linked example code file demonstrates use of `str2digit()`.

str2dsp_len()

Calculates the number of characters required to fill a specified number of display positions. Characters are counted in the source buffer, starting at a specified offset in the given direction (FORWARD or REVERSE).

Prototype

```
#include <aclconio.h>

int str2dsp_len(char *source, unsigned offset, short dsp_wid, char dir);
```

Parameters

source	String to count.
offset	Starting offset.
dsp_wid	Display width.
dir	Count direction. Valid values are: <ul style="list-style-type: none">• FORWARD• REVERSE

Return Values

Success:	≥ 0 : Number of characters that can fit in dsp_wid.
Failure:	-1: Source buffer is null. -2: Offset was greater than the string length.

Example

The linked example code file demonstrates use of str2dsp_len().

str2int()

Accepts a null-terminated string and returns an equivalent integer value. Except for a leading minus sign ("-"), non-numeric characters in the string are ignored. The basis for this function is `SVC_2INT()`, which converts the null-terminated string to a counted string.

NOTE

The input string cannot be longer than 40 characters. Overflow and underflow is undefined. `str2int()` returns the same value for a string containing only ASCII zeros, and a string with more than 40 characters.

Prototype

```
#include <aclstr.h>
int str2int(char *buffer);
```

Parameters

`buffer` Buffer containing string to convert; cannot exceed 40 characters.

Return Values

Success: Equivalent integer value of string.

Failure: 0: The input string is greater than 40 characters.

Dependencies

Verix eVo SDK, ACL `ntocs()`

See Also

`long2str()`, `int2str()`, `str2long()`

Example

The linked example code file demonstrates use of `str2int()`.

str2long()

Converts string data containing ASCII decimal digits (0 to 9) into a long integer number. The string is parsed from the beginning to the end of the buffer, or until a null is encountered.

This function ignores all non-digit characters, except a minus sign ("-"). The minus sign is interpreted as a sign flag and must be located in the first character position of the string. A sign flag occurring in any other position in the string is ignored.

NOTE

The module does not perform error checking. Overflow and underflow conditions are undefined. `str2long()` returns the same result for a string containing only ASCII zeros and a string with more than 40 characters.

This routine is the inverse of `long2str()`. This routine differs from the standard `atol()` routine in that it ignores non-numeric characters (except for a sign flag in the first byte).

Prototype

```
#include <aclstr.h>
long str2long(char *string);
```

Parameters

`string` Null-terminated string to convert.

Return Values

Long integer converted value.

Success: Long integer converted value.

Failure: -1: Buffer is NULL.

Dependencies

Verix eVo SDK

See Also

[long2str\(\)](#), [int2str\(\)](#), [str2int\(\)](#)

Example

See also the [EXTRANS.C](#) example program. The linked example code file demonstrates use of `str2long()`.

track_parse()

Parses track data. The `TRACK` structure inputs the unformatted data in to the function, and also holds the results of the parsing. The `tk_option` parameter specifies the track number to parse.

Track data is passed to the function in field `track` of structure `parsed`. For information on the format of individual tracks, see `card_parse()`. See `ACLDEV.H` for I/O structure definitions.

Prototype

```
#include <acldev.h>
int track_parse(struct TRACK *parsed, unsigned char tk_option);
```

Parameters

`parsed` I/O structure.
`tk_option` Track to parse.

Return Values

Success: 1
Failure: `TK_NOT_FOUND`: Tracks specified in `tk_option` contained an invalid data status byte.
 `INVLD_FORMAT`: Data could not be parsed.

TRACK Structure

```
struct TRACK
{
char acct [23]; /* account */
char exp [5]; /* expiration date */
char name [27]; /* name */
char type [4]; /*
type req'd by VISA & MC */
char PVN [6]; /* PVN req'd by VISA & MC */
char disc [17]; /* track 1 and 2 only */
char track [108]; /* raw track data */
}
```

Example

The linked example code file demonstrates use of `track_parse()`.

view_buffer()

Displays a string in the current display window and allows the user to use the [*] and [#] keys to scroll to the right and left, respectively, when viewing strings larger than the current display.

view_buffer() also allows the calling routine to specify the key or keys allowed to terminate viewing the string. Pressing any key other than [*], [#], and the specified termination key produces an error beep, and the display does not change.

view_buffer() does not permit scrolling past the end of the message string and does not scroll beyond the beginning of the string. A scroll increment is specified in the call. This is the number of characters to move either right or left for each press of the [*] or [#] key.

#defines are included in ACLCONIO.H for constructing maps for the termination key map. To accept one or more keys to terminate the display, simply OR the desired key values together. For example, KM_BS | KM_CR permits the enter or backspace key to end the display.

If inc is less than 1, 1 is used as the scroll increment. inc cannot be greater than the display window size.

NOTE



In Vx700 terminal, instead of [*] and [#] keys, down and up arrow keys are used to scroll to the right and left, respectively.

Prototype

```
#include <aclconio.h>
int view_buffer(char *buf, int inc, unsigned long key_map);
```

Parameters

buf	Buffer with message to scroll.
inc	Scroll increment.
key_map	Map of termination keys.

Return Values

Success: 1

Dependencies

keyin_mapped(), ERR_BEEP()

See Also

keyin_mapped()

Example

See also the [EXTRANS.C](#) example program. The linked example code file demonstrates use of view_buffer().

DVLr Function Calls

DVLr functions allow manipulation of double variable length (counted) records (DVLrs). The following DVLr function calls are described:

- `delete_dvlr()`
- `insert_dvlr()`
- `read_dvlr()`
- `seek_dvlr()`
- `write_dvlr()`

A file using DVLrs provides the ability to search both forward and backward through a file. Each record is composed of a forward-length count, the data, and a backward-length count.

Forward-length count = 2 + DATA LENGTH + 2.

Backward-length count = 2 + DATA LENGTH + 2.

In addition, you can use DVLr functions with compressed or uncompressed records.



To avoid file pointer positioning errors, do not mix file access methods. For example, do not write a record using `write_dvlr()` one time and `write()` the next time.

delete_dvlr()

Deletes one or more double variable length records from a DVLr file.



The file position indicator should be positioned at the start of a record to guarantee that the function deletes the correct amount of data.

Prototype

```
#include <aclfile.h>
int delete_dvlr(int handle, int count);
```

Parameters

`handle` File handle returned by `open()`.

`count` Number of records to delete, starting at the current file position.

Return Values

Positive number indicates successful operation.

Value	Description
0	Attempt was made to read at EOF or the record size is 0.
-1	Failure, with <code>errno</code> containing the error code.

insert_dvlr()

Inserts a double variable length record into a DVLR file.

NOTE

The file position indicator must be positioned at the beginning of the DVLR. If it is not, the file may become corrupt.

Prototype

```
#include <aclfile.h>
int insert_dvlr(int handle, const char *buffer, int size);
```

Parameters

handle File handle returned by `open()`.
buffer Pointer to the data to insert.
size Maximum number of bytes to insert into the file.

Return Values

Number of bytes successfully inserted (positive number).

Success: 0: Attempt made to read at EOF or the record size is 0.

Failure: -1: Failure, with `errno` containing the error code.

read_dvlr()

Reads a double variable length record from a DVLR file.

NOTE

To read the record successfully, the file position indicator must be positioned at the beginning of a normal DVLR.

After reading the record, the file position indicator is advanced to the start of the next record, even if all of the bytes in the record are not copied to the buffer.

Prototype

```
#include <aclfile.h>
int read_dvlr(int handle, char *buffer, int size);
```

Parameters

handle File handle returned by `open()`.
buffer Pointer to buffer storing the incoming record.
size Maximum number of bytes to read from the file.

Return Values

Number of bytes successfully read (positive number).

Success: 0: Attempt was made to read at EOF or the record size is 0.

Failure: -1: Failure, with `errno` containing the error code.

seek_dvlr()

Seeks the specified number of records forward or backward in a DVLR file.

NOTE



To seek past the correct quantity of data, the file position indicator must be initially positioned at the beginning of the DVLR.

Prototype

```
#include <aclfile.h>
long seek_dvlr(int handle, long offset, int origin);
```

Parameters

- handle** File handle returned by `open()`.
- offset** Number of records to seek forward or backward. Use a negative value to seek backward.
- origin** Position to offset from. Valid values are:
- `SEEK_SET` Searches the beginning of file.
 - `SEEK_CUR` Searches the file pointer current position.
 - `SEEK_END` Searches the end of the file.

The above functions are available in SVC.H.

Return Values

A positive number indicates absolute offset (in bytes) into the file after performing the seek.

Failure: -1: Indicates a failure, with `errno` containing the error code.

write_dvlr()

Writes a double variable length record to a DVLR file.

NOTE

The file position indicator must be positioned at the beginning of the DVLR. If it is not, the file may become corrupt.

If the file position indicator is positioned on an existing DVLR, that record is replaced with the new data (as if a [delete_dvlr\(\)](#)/[insert_dvlr\(\)](#) was performed). If the file position indicator is positioned at the end of the file, the data is appended to the file.

Prototype

```
#include <aclfile.h>

int write_dvlr(int handle, const char *buffer, int size);
```

Parameters

handle File handle returned by `open()`.

buffer Pointer to the data to write.

size Maximum number of bytes to write to the file.

Return Values

Number of bytes successfully written (positive number).

Success: 0: Attempt was made to read at EOF or the record size is 0.

Failure: -1: Failure, with `errno` containing the error code.

IVLR Function Calls

IVLR functions allows manipulation of integer variable length records (counted). The following IVLR function calls are described:

- `delete_ivlr()`
- `insert_ivlr()`
- `read_ivlr()`
- `replace_ivlr()`
- `seek_ivlr()`
- `write_ivlr()`

CAUTION



To avoid file pointer positioning errors, do not mix file access methods. For example, do not write a record using `write_ivlr()` one time and `write()` the next time.

delete_ivlr()

Deletes one or more integer counted records from an IVLR file.

Prototype

```
#include <aclfile.h>
int delete_ivlr(int h_file, unsigned int count);
```

Parameters

- `h_file` Handle returned by `open()`.
- `count` Number of records to delete, starting at the current file position.

Return Values

- Success: Positive: Number of integers deleted
- Failure: -1: Failure with `errno` containing the error code as determined by `delete()` or `read()`.
- 2: Attempt to delete data past the end of file.

insert_ivlr()

Inserts an integer counted record into an IVLR file.

Prototype

```
#include <aclfile.h>
int insert_ivlr(int h_file, const char *buffer, int rec_size);
```

Parameters

- `h_file` Handle returned by `open()`.
- `buffer` Pointer to data to insert.
- `rec_size` Maximum number of bytes to insert into the file.

Return Values

- Success: Positive: Number of bytes inserted
- Failure: -1: Indicates a failure, with `errno` containing the error code as determined by `insert()`.

read_ivlr()

Reads an integer counted record from an IVLR file.

Prototype

```
#include <aclfile.h>
int read_ivlr(int h_file, char *data, int size);
```

Parameters

<code>h_file</code>	File handle returned by <code>open()</code> .
<code>data</code>	Pointer to buffer storing the incoming data.
<code>size</code>	Maximum number of bytes to read from the file.

Return Values

Success:	Positive: Number of characters read.
Failure:	-1: Indicates a failure, with <code>errno</code> containing the error code as determined by <code>read()</code> .

replace_ivlr()

Replaces a record at the current position in an IVLR file.

Prototype

```
#include <aclfile.h>
int replace_ivlr(int h_file, char *buffer, int rec_size);
```

Parameters

<code>h_file</code>	Handle returned by <code>open()</code> .
<code>buffer</code>	Pointer to data to insert.
<code>rec_size</code>	Maximum number of bytes to insert into the file.

Return Values

Success:	Positive: Number of bytes in the new record.
Failure:	-1: Indicates a failure, with <code>errno</code> containing the error code as determined by <code>insert()</code> or <code>read()</code> .

seek_ivlr()

Moves the file pointer to the specified record in an IVLR file.

NOTE

To seek past the correct quantity of data, the file position indicator must be initially positioned at the beginning of the IVLR.

Prototype

```
#include <aclfile.h>
long seek_ivlr(int h_file, long rec_num, int origin);
```

Parameters

- | | |
|----------------------|---|
| <code>h_file</code> | Handle returned by <code>open()</code> . |
| <code>rec_num</code> | Record number to search. <code>int origin</code> ; Position to offset from. Valid values are: <ul style="list-style-type: none">• <code>SEEK_SET</code>: From the beginning of file.• <code>SEEK_CUR</code>: From the current position of the file pointer.• <code>SEEK_END</code>: From the end of the file. |
| <code>origin</code> | Position to offset from. Valid values are: <ul style="list-style-type: none">• <code>SEEK_SET</code>: From the beginning of file.• <code>SEEK_CUR</code>: From the current position of the file pointer.• <code>SEEK_END</code>: From the end of the file. |

Return Values

- | | |
|----------|--|
| Success: | Positive: New value of the file position indicator (in total number of bytes from the beginning of the file). |
| Failure: | -1: Indicates that the specified record number was not found. Backward seeking is not supported; a record number previous to the current record number cannot be sought. |

write_ivlr()

Writes an integer counted record to an IVLR file.

Prototype

```
#include <aclfile.h>
int write_ivlr(int h_file, const char *data, int size);
```

Parameters

<code>h_file</code>	Handle returned by <code>open()</code> .
<code>data</code>	Pointer to the data to write.
<code>size</code>	Maximum number of bytes to write to the file.

Return Values

Success:	Positive: Number of characters written.
Failure:	-1: Indicates a failure, with <code>errno</code> containing the error code as determined by <code>write()</code> .

SVC Function Calls

This section discusses the following SVC function calls:

- `SVC_CLOCK()`
- `SVC_KEY_TXT()`
- `SVC_KEY_NUM()`
- `SVC_TICKS()`

SVC_CLOCK()

Allows the user to read or set the current time.

NOTE



This function is supported on EPROM version 10 and higher.

Prototype

```
#include <svctxo.h>
int SVC_CLOCK(int action, char *buffer, int limit);
```

Parameters

- | | |
|---------------------|--|
| <code>action</code> | <ul style="list-style-type: none"> • 0 = places the current time into <code>buffer</code> (reads clock). • 1 = writes the contents of <code>buffer</code> to the clock (sets clock). |
| <code>buffer</code> | Stores the time information. |
| <code>limit</code> | Indicates the maximum number of characters: <ul style="list-style-type: none"> • 15 when reading the clock. • 14 when writing to the clock. |

Return Values

- Success: `bytes`: The number of bytes read or written.
- Failure: -1: Failure

Example

The linked example code file demonstrates use of `SVC_CLOCK()`.

SVC_KEY_NUM()

Uses keyboard, display, and beeper input to retrieve a formatted decimal number (counted string). The following are used for key presses:

- The alpha key is used to toggle between positive and negative numbers. A negative number is displayed with leading-minus sign (-).
- The backspace key is used to delete the last character entered.
- The clear key is used to abort the read.

Prototype

```
include <svctxo.h>
int SVC_KEY_NUM(int dest_buff, int max_digits, int fraction, int
                punctuate);
```

Parameters

dest_buff	Specifies where the resulting counted string of digits is stored.		
max_digits	Specifies the maximum number of digits permitted (excluding punctuation marks). This must be less than or equal to 15.		
fraction	Specifies the number of digits to the right of the radix. For example, the fractional value. This must be less than or equal to 10. If the user presses [ENTER] without additional input, the destination buffer contains a counted string of zeros in the format specified in fraction.		
	0 - fraction	bytes_read = 2.	buffer = 0.
	1 - fraction	bytes_read = 3.	buffer = 0.0
	2 - fraction	bytes_read = 4.	buffer = 0.00
	Specifies the style of punctuation requested:		
	0 - Radix = point	No separator	12345678.90
	2 - Radix = point	Separator = comma	12,345,678.90
punctuate	4 - Radix = comma	No separator	12345678,90
	6 - Radix = comma	Separator = point	12.345.678,90

Return Values

Success:	bytes_read: The integer value containing the actual number of bytes stored in the destination buffer (including punctuation characters) on a successful read.
Failure:	1: The user pressed the clear key. The clear key trap is disabled when this function is called.

Example

The linked example code file demonstrates use of SVC_KEY_NUM().

SVC_KEY_TXT()

Uses keyboard, display, and beeper to retrieve formatted data input. The following are used for key presses:

- The control keys in the fourth column of the keyboard (backspace, alpha, and enter) provide primitive editing options.
- The enter key is used to terminate the string, assuming the `minimum_allowed` parameter has been satisfied.
- Each of the twelve imprinted keys on the keyboard represents several characters: the alpha key scrolls through each of the possible characters. The scroll sequence is inscribed on the key caps, with the following two exceptions:

[0]: "0", "-", " " (space), "+"

[#]: "#", "!", ".", "@", "=", "&"

- The backspace key is used to delete the last character entered.
- The clear key is used to abort the read.

The `SVC_KEY_TXT()` does not accept input from a function or screen-addressable key, and beeps on a keypress from one of these keys.

Prototype

```
#include <txosvc.h>

short SVC_KEY_TXT(char *dest, short type, short max, short min,
                  char szKeymap[][CHAR_PER_KEY], int KeyMapSize);
```

Parameters

`szKeyMap[][CHAR_PER_KEY]` The key map specifying the mapping of the logical alphanumeric keys to the physical keys on the keypad.

```
szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY] = { "0- +%", "1QZ.\\", "2ABC&",
"3DEF%", "4GHI*", "5JKL/", "6MNO~", "7PRS^", "8TUV[ ", "9WXY]",
"*, '\\":", "#=: $?" };
```

By default, in the Verix eVo ACT library, `CHAR_PER_KEY` is defined as 6. Hence the application can have five chars per key in the key-mapping array. User can modify this value by calling `set_chars_per_key_value()`. Based on the value set by the user, the `szKeyMap` array should be passed to the library APIs. There is no validation done on the array size by the library.

`KeyMapSize` Size of the key map specified.

`dest` The location for the entered data to be returned (should be large enough to receive `max_allowed` bytes plus one count byte).

`type` Specifies the type of data entry allowed:

- 0 = Numeric keys only.
- 1 = Alphanumeric keys.

- 2 = Numeric keys for password. Each key is echoed with an asterisk.
- 3 = Alphanumeric keys for password. Each key is echoed with an asterisk.

max	The maximum number of bytes that can be entered.
min	The minimum number of bytes allowed.

Return Values

- Success:
- bytes_read: Integer value containing the actual number of bytes read on a successful read.
- Failure:
- 1: The user has pressed the clear key to cancel the read operation. The clear key trap is disabled when this function is called.

Example The linked example code file demonstrates use of [SVC_KEY_TXT\(\)](#).

See Also [set_chars_per_key_value\(\)](#)

SVC_TICKS()

Allows the user to check if a tick value has expired or read the systems tick counter.

Prototype

```
#include <svctxo.h>
int SVC_TICKS(int action, long *longadr);
```

Parameters

<code>action</code>	Determines the behavior of the function. Valid values are: <ul style="list-style-type: none"> • 0 = Compares the specified longword value against the systems current longword tick counter. The return value indicates if the users value has expired: <ul style="list-style-type: none"> • 0 = expired • 1 = pending • 1 = Copies the systems longword tick counter to the callers longword.
<code>longadr</code>	Pointer to store the tick counter.

Return Values

Return values are based on the value of action.

	Action:	Returned Value
Success:	1:	Always returns 1.
Failure:	0:	<ul style="list-style-type: none"> • 0 = users tick expired. • 1 = users tick is pending (not expired).

This function is supported on EPROM version 10 and higher.

See Also

`tick_compare()`

Example

The linked example code file demonstrates use of `SVC_TICKS()`.

Application Idle Engine Function Calls

This section covers the Branch Table entry, Function Table, and idle loop processing for the application engine, and presents [Application Idle Engine Examples](#). The following application engine functions calls are described:

- `aie_main()`
- `appl_idle_get_cancel_poll_time()`
- `appl_idle_get_fast_poll_time()`
- `appl_idle_get_idle_loop_time()`
- `appl_idle_get_slow_poll_time()`
- `appl_idle_set_cancel_loop_time()`
- `appl_idle_set_fast_poll_time()`
- `appl_idle_set_idle_poll_time()`
- `appl_idle_set_slow_poll_time()`

branch_tbl_entry

```
typedef struct branch
{
    short event;
    short func;
    struct branch *next_table;
} Branch_Tbl_Entry;
```

Function Table

```
typedef short(*PF_TABLE)(short state);
```

Idle Loop processing, Slow Poll, Fast Poll

The idle loop processing, slow poll, and fast poll functions should be of AIE Proc.

```
typedef short(* AIE_Proc)();
```

Application Idle Engine Examples

Idle Loop Function

This function is the `main()` loop in the engine and is called once per cycle. In this example, `appl_idle_loop()` calls an `auto_settle()` function and an `auto_download()` function. Based on a certain condition being met, the `auto_download()` function performs a full or partial VeriCentre download and the `auto_settle()` function performs an automatic settlement with the host.

Example

The linked example code file demonstrates use of `appl_idle_loop()`.

aie_main()

Starts the application idle engine. The `main()` of the program is defined by the application. After performing the application initialization, [Application Idle Engine Examples](#) should be invoked. [Application Idle Engine Examples](#) is the idle engine processing loop. See [Figure 1](#) for details on processing.

Prototype

```
# include <applidl.h>

int aie_main(BRANCH_TBL_ENTRY *idletable, PF_TABLE *apptable,
            AIEPROC idle loop, AIEPROC fastpoll, AIEPROC slowpoll,
            AIEPROC activate, AIEPROC deactivate);
```

Parameters

idletable	Branch Table structure to search for the event.
apptable	Function Table structure containing the function pointers to the application event handling routines.
loop	Pointer to the application-defined idle loop function.
fastpoll	Pointer to the application-defined fast poll function.
slowpoll	Pointer to the application-defined slow poll function.
activate	The application-defined routine to call when a activate event occurs.
deactivate	The application-defined routine to call when a deactivate event occurs

Return Values

TRUE.

NOTE



The Application Idle Engine uses the EVT_TIMER supported by the Verix V OS for the slowpoll, fastpoll and idlepoll functionalities. The results are unpredictable when the application uses the EVT_TIMER along with Application Idle Engine, as there might be a clash of EVT_TIMER event by the library as well as the application.

appl_idle_get_cancel_poll_time()

Returns the value of the cancel detect timer. This timer determines the periodicity of the AIE checks for the cancel key.

Prototype

```
#include <applidl.h>

long appl_idle_get_cancel_poll_time();
```

Return Values

The value of the cancel poll timer.

appl_idle_get_fast_poll_time()

Returns the value of the fast poll timer. This timer determines the periodicity of the fast poll function that is called in the idle state.

Prototype

```
#include <applidl.h>
long appl_idle_get_fast_poll_time();
```

Return Values

Success: The value of the fast poll timer.

appl_idle_get_idle_loop_time()

Returns the value of the idle loop timer. This timer determines the periodicity of the idle poll function that is called in the idle state.

Prototype

```
#include <applidl.h>
long appl_idle_get_idle_poll_time();
```

Return Values

Success: The value of the idle poll timer.

appl_idle_get_slow_poll_time()

Returns the value of the slow poll timer. This timer determines the periodicity of the slow poll function that is called in the idle state.

Prototype

```
#include <applidl.h>
long appl_idle_get_slow_poll_time();
```

Return Values

Success: The value of the slow poll timer.

appl_idle_set_cancel_loop_time()

Sets the value of the cancel poll timer.

Prototype

```
#include <applidl.h>
void appl_idle_set_cancel_loop_time(long time);
```

Parameters

time The new value of the cancel poll timer.

Return Values

Void.

appl_idle_set_fast_poll_time()

Sets the value of the fast poll timer. The value set by the user should be in multiples of 50. If the value entered is not in multiples of 50, then it will be rounded to the nearest multiples of 50. Any negative value entered will also be taken as 50.

Prototype `#include <applidl.h>`
`void appl_idle_set_fast_poll_time(long time);`

Parameters

<code>time</code>	The new value of the fast poll timer.
-------------------	---------------------------------------

Return Values Void.

appl_idle_set_idle_poll_time()

Sets the idle poll timer. The value set by the user should be in multiples of 50. If the value entered is not in multiples of 50, then it will be rounded to the nearest multiples of 50. Any negative value entered will also be taken as 50.

Prototype `#include <applidl.h>`
`void appl_idle_set_idle_poll_time(long time);`

Parameters

<code>time</code>	The new value of the idle poll timer.
-------------------	---------------------------------------

Return Values Void.

appl_idle_set_slow_poll_time()

Sets the value of the slow poll timer. The value set by the user should be in multiples of 50. If the value entered is not in multiples of 50, then it will be rounded to the nearest multiples of 50. Any negative value entered will also be taken as 50.

Prototype `#include <applidl.h>`
`void appl_idle_set_slow_poll_time(long time);`

Parameters

<code>time</code>	The new value of the slow poll timer.
-------------------	---------------------------------------

Return Values Void.

Message/Data Entry Engine Function Calls

This section describes the following message/data entry function calls:

- `msg_get()`
- `msg_select_file()`

`msg_get()`

Retrieves a record from the message file.

NOTE



Assumes `msg_select_file()` has been called. This routine does not check for buffer overflow. If the message file is compressed, `msg_get()` decompresses it.

Prototype

```
#include <message.h>
char *msg_get(unsigned message_num, char *buf_ptr);
```

Parameters

<code>message_num</code>	Number of the text message to retrieve.
<code>buf_ptr</code>	Pointer to the buffer receiving the text data (uses global variables <code>msg_file</code> and <code>msg_compress</code>).

Return Values

Success: Pointer to null-terminated text string.
Failure: NULL.

Global variable `errno` contains the error code.

See Also `msg_select_file()`

msg_select_file()

Opens a message file, reads the first record that contains the initialization information, and sets the local file handle and compression variables.

NOTE

The following global variables are set:

`msg_file` = file handle.

`msg_compress` = Compression used; Any previously opened message file is closed.

Compression on/off is determined at the time the message file is built. This is transparent to the user.

Prototype

```
#include <message.h>
int msg_select_file(char *file_name);
```

Parameters

`file_name` Message filename where record is located.

Return Values

Success: Positive: Number of bytes read from the first record in the message file.

Failure: Negative: Number indicates an error.

See Also [msg_get\(\)](#)

ISO 8583 Message Interface Engine Function Calls

This section describes the following ISO 8583 message interface function calls:

- asc_to_asc()
- asc_to_bcd()
- asc_to_str()
- av2_to_str()
- av3_to_av3()
- av3_to_str()
- bcd_to_asc()
- bcd_to_bcd()
- bcd_to_snz()
- bcd_to_str()
- bi2_to_hst()
- bi3_to_hst()
- bin_to_hst()
- bit_to_bit()
- bv2_to_str()
- get_dst_8583()
- get_fn_8583()
- get_src_8583()
- hst_to_bi2()
- hst_to_bi3()
- hst_to_bin()
- iso8583_main()
- map_clear()
- map_man()
- map_reset()
- map_set()
- map_test()
- process_8583()
- set_dst_8583()
- set_src_8583()
- str_to_asc()
- str_to_av2()
- str_to_av3()
- str_to_bcd()
- str_to_bv2()
- str_to_xbc()
- xbc_to_str()

NOTE



Conversion functions provided in the ISO 8583 Engine are not intended to be called directly from an application. These functions are called indirectly through the variable `convert_table` and `process_8583` procedure.

asc_to_asc()

Copies *n* bytes from source to destination, then advances the global pointers dst_8583 and src_8583 by *n*.

Prototype `#include <iso8583.h>`
 `void asc_to_asc(int n);`

Parameters *n* Number of bytes to copy.

Return Values Void.

Dependencies Verix eVo SDK

asc_to_bcd()

Converts *n* ASCII digits to packed BCD and stores them in the destination. If *n* is an odd number, a leading zero is inserted in the destination. This function assumes only ASCII digits are in the source, therefore, no verification for digits is done.

Prototype `#include <iso8583.h>`
 `void asc_to_bcd(int n);`

Parameters *n* Number of ASCII digits to convert.

Return Values Void

Dependencies Verix eVo SDK

asc_to_str()

Moves *n* characters from source to the destination, then appends a null into the destination. This routine makes a null-terminated string out of *n* bytes. If there are more characters than *n* bytes in the source, they are not copied to the destination.

Prototype `#include <iso8583.h>`
 `void asc_to_str(int n);`

Parameters *n* Number of characters to move.

Return Values Void

Dependencies Verix eVo SDK.

av2_to_str()

Converts a 1-byte counted string into a null-terminated ASCII string and stores the result in the destination. The count is in BCD form.

Prototype

```
#include <iso8583.h>
void av2_to_str(int c);
```

Parameters

c Perform no conversion if length > c.

Return Values Void

Dependencies Standard TXO C Library, [asc_to_asc\(\)](#).

av3_to_av3()

Copies a 2-byte counted string from the source to the destination. The 2-byte count is in BCD format. For example, the count for a string of length 128 bytes is stored as 01, 40 in ASCII, which is 01, 28 in BCD representation. The count bytes are also copied to the destination.

Prototype

```
#include <iso8583.h>
void av3_to_av3(int c);
```

Parameters

c Perform no conversion if length > c.

Return Values Void

Dependencies Verix eVo SDK

av3_to_str()

Converts a 2-byte counted ASCII string to a null-terminated ASCII string. The byte count in the source is in BCD form.

Prototype

```
#include <iso8583.h>
void av3_to_str(int c);
```

Parameters

c Perform no conversion if length > c.

Return Values Void

Dependencies Standard TXO C Library, [asc_to_asc\(\)](#).

bcd_to_asc()

Expands a packed BCD sequence into its ASCII equivalent. It assumes all the BCD digits are in the range 0–9. No error checking is done by this function.

Prototype `#include <iso8583.h>`
 `void bcd_to_asc(int n);`

Parameters `n` Number of BCD nibbles.

Return Values Void

Dependencies Standard TXO C Library

bcd_to_bcd()

Moves *n* BCD nibbles from the source to the destination. Since there are two nibbles per byte and everything is byte aligned, this is done as byte moves.

Prototype `#include <iso8583.h>`
 `void bcd_to_bcd(int n);`

Parameters `n` Number of BCD nibbles to move.

Return Values Void

Dependencies Verix eVo SDK

bcd_to_snz()

Copies a fixed-length BCD field to a string. Leading zeros are removed, but are included in the conversion count. For example, if a BCD string 00012598 is converted to a string without leading zeros and a count of 6, then the results is an ASCII string of 125.

Prototype `#include <iso8583.h>`
 `void bcd_to_snz(int n);`

Parameters `n` Number of BCD digits to copy.

Return Values Void

Dependencies Standard TXO C Library, [bcd_to_str\(\)](#)

bcd_to_str()

Copies *n* BCD digits from the source to the destination and then appends a null character in the destination. Leading zeros in the source are preserved.

Prototype

```
#include <iso8583. h>
void bcd_to_str(int n);
```

Parameters

n Number of BCD digits.

Return Values Void

Dependencies Standard TXO C Library, [bcd_to_asc\(\)](#)

bi2_to_hst()

Converts a 1-byte counted string to a null-terminated ASCII hex string. No error checking is performed by this function.

Prototype

```
#include <iso8583. h>
void bi2_to_hst(int c);
```

Parameters

c Perform no conversion if length > *c*.

Return Values Void

Dependencies Standard TXO C Library, [bin_to_hex\(\)](#)

bi3_to_hst()

Converts a 2-byte counted string to a null-terminated ASCII hex string. No error checking is performed by this function.

Prototype

```
#include <iso8583. h>
void bi3_to_hst(int c);
```

Parameters

c Perform no conversion if length > *c*.

Return Values Void

Dependencies Standard TXO C Library, [bin_to_hex\(\)](#), [get_len_v2\(\)](#)

bin_to_hst()

Converts n bytes of binary digits into $2n$ bytes of ASCII hex digits. This always results in an even number of hex digits. No error checking is performed by this function.

Prototype `#include <iso8583. h>`
`void bin_to_hst(int n);`

Parameters n Number of digits to convert.

Return Values Void

Dependencies Standard TXO C Library, bin_to_hex()

bit_to_bit()

Copies n bits from the source to the destination. Always moves in whole bytes.

Prototype `#include <iso8583. h>`
`void bit_to_bit(int n);`

Parameters n Number of bits to copy.

Return Values Void

Dependencies Verix eVo SDK

bv2_to_str()

Converts a 1-byte counted BCD string to a null-terminated ASCII string.

Prototype `#include <iso8583. h>`
`void bv2_to_str(int c);`

Parameters c Perform no conversion if length $> c$.

Return Values Void

Dependencies Verix eVo SDK

get_dst_8583()

Prototype `#include <iso8583.h>`
`void get_dst_8583(unsigned char *dst);`

Parameters `dst` Buffer to store the destination buffer of the ISO 8583 engine.

Return Values None

Dependencies Used in all the conversion routines that must obtain the destination buffer of the ISO 8583 engine.

get_fn_8583()

Returns the erroneous field number during packet packing or unpacking. The application calls this function to obtain the erroneous field after `process_8583()` fails, returning an -1 error. This implies that the field is not defined in the field table.

Prototype `#include <iso8583.h>`
`int get_fn_8583(void);`

Return Values Success: The field number that had the error during message packing or unpacking.

Dependencies Obtain a meaningful results, the application should call `get_fn_8583()` only after the packing/unpacking fails.

get_src_8583()

Prototype `#include <iso8583.h>`
`void get_src_8583(unsigned char *src);`

Parameters `src` Buffer to store the source buffer of the ISO 8583 engine.

Return Values None

Dependencies Used in all the conversion routines that must obtain to get the source buffer of the ISO 8583 engine.

hst_to_bi2()

Converts a null-terminated ASCII hex string to a 1-byte counted string and stores the result. The count is in BCD format and is only half the length of the ASCII hex string. No error checking is performed by this function.

Prototype `#include <iso8583. h>
void hst_to_bi2(int c);`

Parameters `c` Perform no conversion if length > c.

Return Values Void

Dependencies Standard TXO C Library, hex_to_bin()

hst_to_bi3()

Converts a null-terminated ASCII hex string to a 2-byte counted string and stores them in the destination. The count is in BCD format and is only half the length of the ASCII hex string. No error checking is performed by this function.

Prototype `#include <iso8583. h>
void hst_to_bi3(int c);`

Parameters `c` Perform no conversion if length > c.

Return Values Void

Dependencies Standard TXO C Library, hex_to_bin()

hst_to_bin()

Converts *n* bytes of ASCII hex digits to binary and stores the result. The source must contain an even number of hex digits and the count; *n*, must be an even number. No error checking is performed by this function.

Prototype `#include <iso8583. h>
void hst_to_bin(int n);`

Parameters `n` Number of ASCII hex digits to convert.

Return Values Void

Dependencies Standard TXO C Library, hex_to_bin()

iso8583_main()

Passes pointers to a convert table, function pointers to a user-defined functions that returns the value of variant fields, and two buffers to hold the values of the source and the destination (when conversion routines are used). These pointers are used by the ISO Message Engine.

Prototype

```
# include <iso8583.h>

void iso8583_main(converters *pConvertTable, fn Ret1, fn Ret2,
                  unsigned char *szdst_8583, unsigned char *szsrc_8583);
```

Parameters

pConvertTable	Pointer to a convert table.
Ret1	Function pointer to a user-defined function that returns the value of variant fields.
Ret2	Function pointer to a user-defined function that returns the value of variant fields.
szdst_8583	Buffer to store the destination buffer of the ISO 8583 engine.
szsrc_8583	Buffer to store the source buffer of the ISO 8583 engine.

Return Values

None

Dependencies

For any application, [iso8583_main\(\)](#) should be called in `main()` before using the ISO Message Engine.

map_clear()

Resets all bits in the designated map. The `max_fn` parameter indicates the maximum field number in the map (64, 128, 192, or 256). This is necessary to ensure the correct construction of maps that exceed 64 bits.

Prototype

```
#include <iso8583.h>

void map_clear(unsigned char *map, int max_fn);
```

Parameters

map	Pointer to bit map.
max_fn	Maximum field number in map.

Return Values

Void

Dependencies

Verix eVo SDK

See Also

[map_test\(\)](#), [map_set\(\)](#), [map_reset\(\)](#), [map_man\(\)](#)

Example

The linked example code file demonstrates use of `map_clear()`.

map_man()

Turns bits on or off in the specified map. The default is to set (turn on) the bit in the map associated with each field number. To reset a bit, simply add the constant 0xFF to the desired field number. Since there are a variable number of parameters, add the constant STOP to the last parameter. If `field_no` is illegal for the given map, the operation on that particular bit is ignored.

Prototype `#include <iso8583. h>`
`void map_man(unsigned char *va_alist, ...);`

Parameters

<code>map</code>	Pointer to the bitmap.
<code>field_no</code>	Field (bit number) to reset.

Return Values Void

Dependencies Standard TXO C Library

See Also [map_clear\(\)](#), [map_test\(\)](#), [map_set\(\)](#), [map_reset\(\)](#)

Example The linked example code file demonstrates use of `map_man()`.

map_reset()

Turns off the bit associated with the field number. If `field_no` is illegal for the given map, this function does nothing.

Prototype `#include <iso8583. h>`
`void map_reset(unsigned char *map, int field_no);`

Parameters

<code>map</code>	Pointer to the bitmap.
<code>field_no</code>	Field (bit number) to reset.

Return Values Void

Dependencies Standard TXO C Library, `LEGAL_FIELD()`

See Also [map_clear\(\)](#), [map_test\(\)](#), [map_set\(\)](#), [map_man\(\)](#)

Example The linked example code file demonstrates use of [map_reset\(\)](#).

map_set()

Turns on the bit associated with the field number. If `field_no` is illegal for the given map, this function does nothing.

Prototype

```
#include <iso8583. h>
void map_set(unsigned char *map, int field_no);
```

Parameters

<code>map</code>	Pointer to the bitmap.
<code>field_no</code>	Field (bit number) to set.

Return Values

Void

Dependencies

Standard TXO C Library, `LEGAL_FIELD()`

See Also

[map_clear\(\)](#), [map_test\(\)](#), [map_reset\(\)](#), [map_man\(\)](#)

Example

The linked example code file demonstrates use of [map_set\(\)](#).

map_test()

Returns the status of the bit in the bitmap corresponding to the field number. The returned value is 0 if the bit is off, and 1 if it is on.

Prototype

```
#include <iso8583. h>
int map_test(unsigned char *map, int field_no);
```

Parameters

<code>map</code>	Pointer to the bitmap.
<code>field_no</code>	Field (bit number) to check.

Returns Values

Success:	1: The bit is set.
Failure:	0: The bit is off or an illegal field number was passed by the caller.

Dependencies

Verix eVo SDK

See Also

[map_clear\(\)](#), [map_set\(\)](#), [map_reset\(\)](#), [map_man\(\)](#)

Example

The linked example code file demonstrates use of [map_test\(\)](#).

process_8583()

Processes packing or unpacking of a packet by checking the bits in the map. For each bit that is set, it either appends the formatted data to the outgoing packet or disassembles the packet and stores the data into proper application variables as indicated by the parameter `how`.

Prototype

```
#include <iso8583.h>

int process_8583(int how, field_struct *field_tbl, unsigned char *map,
                unsigned char *buffer, int limit);
```

Parameters

<code>how</code>	Pack or unpack.
<code>field_tbl</code>	Field table.
<code>map</code>	Bitmap to pack.
<code>buffer</code>	Buffer address.
<code>limit</code>	Maximum buffer size.

Return Values

Success:	<p>≥ 0: Assembly: number of bytes placed in <code>buffer</code>, which is used for write. Disassembly: Number of bytes in <code>buffer</code> not processed, which should be zero.</p>
Failure:	<p>-1: Field not defined in field table. The variable <code>fn_8583</code> contains the field in error.</p> <p>-2: Stored more in <code>buffer</code> than limit.</p> <p>-3: Exceeded size of valid data in <code>buffer</code>.</p> <p>-4: Stored more than variable limit on an incoming message.</p> <p>-5: No matching variant field definition.</p>

Dependencies

Verix eVo SDK

Example

The linked example code file demonstrates use of `process_8583()`.

set_dst_8583()

Sets the destination buffer of the ISO 8583 engine.

Prototype

```
#include <iso8583.h>

void set_dst_8583(unsigned char *dest);
```

Parameters

<code>dest</code>	Data to set the destination buffer of the ISO 8583 engine.
-------------------	--

Return Values

None

Dependencies

Used in all conversion routines that must set the destination buffer of the ISO 8583 engine.

set_src_8583()

Sets the source buffer of the ISO 8583 engine.

Prototype

```
#include <iso8583.h>
void set_src_8583(unsigned char *src);
```

Parameters

src Data to set the source buffer of the ISO 8583 engine.

Return Values

None

Dependencies

Used in all the conversion routines that must set the source buffer of the ISO 8583 engine.

str_to_asc()

Converts a null-terminated ASCII string to fixed-size ASCII string with blank padding on the right, if necessary. If the count specified is less than the length of the string, the string is truncated.

Prototype

```
#include <iso8583.h>
void str_to_asc(int n);
```

Parameters

n Size of the ASCII string.

Return Values

Void

Dependencies

Standard TXO C Library, [asc_to_asc\(\)](#)

str_to_av2()

Converts a null-terminated ASCII string to a 1-byte counted string. The count is in BCD format.

Prototype

```
#include <iso8583.h>
void str_to_av2(int c);
```

Parameters

c Required to be 1.

Return Values

Void

Dependencies

Verix eVo SDK

str_to_av3()

Converts a null-terminated ASCII string to a 2-byte counted string. The count is in BCD format.

Prototype `#include <iso8583. h>`
 `void str_to_av3(int c);`

Parameters `c` Required to be 1.

Return Values Void

Dependencies Verix eVo SDK

str_to_bcd()

Converts a null-terminated ASCII string to fixed-size BCD with zero padding on the left, if necessary. If the length of the string is greater than the count specified, the characters beyond the count in the string are not converted.

Prototype `#include <iso8583. h>`
 `void str_to_bcd(int n);`

Parameters `n` Size of the BCD string.

Return Values Void

Dependencies Standard TXO C Library, max_asc_to_bcd()

str_to_bv2()

Converts a null-terminated ASCII string (containing digits only) to a 1-byte counted BCD string. The count is in BCD format. If the length of the string is an odd number, a trailing zero is appended to the destination.

Prototype `#include <iso8583. h>`
 `void str_to_bv2(int c);`

Parameters `c` Required to be 1.

Return Values Void

Dependencies Verix eVo SDK

str_to_xbc()

Converts a null-terminated ASCII string to a BCD string while preserving the first byte of the source. The first byte must contain one of the two following values:

- C: Denotes a credit transaction.
- D: Denotes a debit transaction.

Prototype

```
#include <iso8583. h>
void str_to_xbc(int n);
```

Parameters

n Size of the BCD string.

Return Values

Void

Dependencies

Standard TXO C Library, [str_to_bcd\(\)](#)

xbc_to_str()

Converts a BCD string to a null-terminated ASCII string while preserving the first byte of the source. The first byte must contain one of the two following values:

- C: Denotes a credit transaction.
- D: Denotes a debit transaction.

Prototype

```
#include <iso8583. h>
void xbc_to_str(int n);
```

Parameters

n Size of null-terminated ASCII string.

Return Values

Void

Dependencies

Standard TXO C Library, [bcd_to_str\(\)](#)

PIP Engine Function Calls

This section describes the following PIP Engine function calls:

- `check_8583_tpdu_msg_id()`
- `find_field_index()`
- `pip_main()`
- `pip_trans()`
- `prot_8583()`
- `prot8583_main()`
- `save_8583_tpdu_msg_id()`
- `set_tpdu_length()`

check_8583_tpdu_msg_id()

Verifies a valid match between the current values of the TPDU and message ID found in the field table, and the values passed in the call. This comparison requires that the TPDU and message ID are stored as packed numeric (BCD) values.

This function is used by `pip_trans()` and `prot_8583()`. In addition, this function can be used in other places in the application.

Both the TPDU and the message ID are used in the transmit packet.

Prototype

```
#include <prot8583.h>
#include <aclfile.h>
int check_8583_tpdu_msg_id(COMM_8583_DATA *control, unsigned char
                           *orig_tpdu, unsigned char *orig_msg_id);
```

Parameters

<code>control</code>	Pointer to the data structure used by the ISO 8583 Protocol Engine.
<code>orig_tpdu</code>	Pointer to the buffer containing the original TPDU.
<code>orig_msg_id</code>	Pointer to the buffer containing the original message ID.

Return Values

Success: 1: The TPDU and message ID match.

Failure: `errno` set to `PROT8583_RCV_TPDU_ERROR`: TPDU is incorrect.

`errno` set to `PROT8583_RCV_MSG_ID_ERROR`: Message ID is incorrect.

`errno` set to `PROT8583_BAD_TPDU_MSG_ID`: Both the TPDU and message ID are incorrect.

find_field_index()

Searches a field table array for a specific field entry. This function is used by the [pip_trans\(\)](#) function.

CAUTION



There is no mechanism within the function to determine the validity of the `field_struct`-type pointer passed to it. Passing an invalid `field_struct` pointer to the function causes invalid results.

Prototype

```
#include <amexhost.h>
#include <aclstr.h>
#include <iso8583.h>
field_struct *find_field_index(int search_num, field_struct *ptr);
```

Parameters

`search_num` The field for which to search.

`ptr` The field table array to search.

Return Values

Success: The address of the array element.

Failure: NULL if requested field is not found.

pip_main()

Passes pointers to the map buffer, pointer to the message ID buffer, pointer to processing code, and a function pointer to user-defined `set_trans_field` function. The first three buffers are filled by the application before these are used by the PIP Engine.

Prototype

```
#include <portdef.h>
void pip_main(unsigned char *tmap, char *tmsgid, char *tproccode,
               Fxn func);
```

Return Values

None

Dependencies

For any application, [pip_main\(\)](#) must be called in `main()` before using the PIP Engine. Also, note that the fourth parameter can be null if the application wants to use the default `set_trans_field` function provided in PIP engine.

pip_trans()

Provides advice message processing and an interface to the ISO 8583 Protocol Engine. This function sets up AMEX host (PIP) processing, and defines the transaction type and initializes the host data structure.

NOTE



- Verix eVo ACT library checks for the cancel key operation during PIP transaction. If the `DISABLECANKEYPIP config.sys` entry is set to 1, then Verix eVo ACT library ignores the cancel key pressed during the PIP transaction.
- If the `DISABLECANKEYPIP config.sys` entry is set to 0, or the config entry is not found, then Verix eVo ACT library checks for the cancel key press and aborts the PIP transaction if it is pressed.
- For more information on the PIP specification, refer to the *AMEX Express 3000 PIP Terminal Technical Specifications* (available from American Express).

Prototype

```
#include <amexhost.h>
#include <aclstr.h>
#include <iso8583.h>
int pip_trans(int trans_type, HOST_8583_DATA *host_struct);
```

Parameters

`trans_type` Informs `pip_trans()` of the requested transaction. Valid values are:

- `A_AUTH_TRANS` `0x0028` /* `0x0028` or `00101000b` */
- `A_REFUND_TRANS` `0x0009` /* `0x0009` or `00001001b` */
- `A_SALE_TRANS` `0x0008` /* `0x0008` or `00001000b` */
- `A_VOID_CREDIT` `0x000d` /* `0x000d` or `00001101b` */
- `A_VOID_DEBIT` `0x000c` /* `0x000c` or `00001100b` */
- `A_REFUND_OFFLINE` `0x0011` /* `0x0011` or `00010001b` */
- `A_SALE_OFFLINE` `0x0010` /* `0x0010` or `00010000b` */
- `A_VOID_CR_OFFLINE` `0x0015` /* `0x0015` or `00010101b` */
- `A_VOID_DB_OFFLINE` `0x0014` /* `0x0014` or `00010100b` */
- `A_ADJUST_CREDIT` `0x0013` /* `0x0013` or `00010011b` */
- `A_ADJUST_DEBIT` `0x0012` /* `0x0012` or `00010010b` */
- `A_SEND_ADVICE` `0x000b` /* `0x000b` or `00001011b` */
- `A_SEND_REVERSAL` `0x000a` /* `0x000a` or `00001010b` */
- `A_CLOSE_BEGIN` `0x0018` /* `0x0018` or `00011000b` */
- `A_CLOSE_DETAIL` `0x0038` /* `0x0038` or `00111000b` */
- `A_DCP_REVERSAL` `0x0048` /* `0x0048` or `01001000b` */
- `A_CLOSE_END` `0x0078` /* `0x0078` or `01111000b` */

`host_struct` Pointer to the structure used by the PIP Engine that contains all the required information to complete a transaction with the AMEX host.

Return Values

Success: 1: Transaction type and structure data successfully passed to application.

Failure: AMEXHOST_ADJUST_ERROR (-64)
 AMEXHOST_DUPKEY_ERROR (-65)
 AMEXHOST_VOID_ERROR (-66)
 AMEXHOST_INVADV_ERROR (-67)
 AMEXHOST_INV_OPTION (-68)
 PIP_NOT_ONLIN (-69)
 AMEX_VOICE_AUTH (-70)
 AMEX_DCP_APPROVAL (-71)
 AMEX_NO_ADVICE (-72)
 AMEX_DCP_REVERSAL (-73)
 HOST_ERROR (-74)

NOTE

`errno` may provide additional information.

The following options are defined in AMEXHOST. H. Transaction codes are bitmapped according to the following table (with the exception of close packets and transmission of advice packets):

Bit Position	Description
1	1 = Credit / 0 = Debit
2	1 = Adjustment transaction
3	1 = Void transaction
4	1 = Online transaction
5	1 = Offline transaction
6	1 = Close packets
7	1 = Close upload
8	1 = Final close

`host_struct` must be properly initialized following the HOST_8583_DATA structure as defined in AMEXHOST. H.

The `pip_trans()` function only uses the lower byte of the `trans_type` parameter, which it passes to the AMEX host module. This module uses the lower byte to determine the type of transaction being processed (as defined in `AMEXHOST.H`). The AMEX host module passes the `trans_type` parameter to the `set_trans_fields()` function (application defined). This function may use both the lower and upper bytes to complete any additional transaction information required by the application.

NOTE

The lower byte should only use assigned values defined in `AMEXHOST.H`. The upper byte can be assigned any value.

In the following example, the application uses the lower byte to define a valid AMEX transaction, and the upper byte to define the bitmap, message ID, and processing code associated with the transaction.

```
#include "amexhost.h"
/* A_ADJUST_CREDIT and A_ADJUST_DEBIT are defined in AMEXHOST.H. */
#define ADJUST_CREDIT (0x1000 | A_ADJUST_CREDIT)
/* 0x0013 or 00010011b */
#define ADJUST_DEBIT (0x1100 | A_ADJUST_DEBIT)
/* 0x0012 or 00010010b */
/* File Format
* Records 0 - 20 = Bitmap Index, Message ID Index, and Processing Code
Index
* Records 21 - 35 = ISO 8583 Bitmaps
* Records 36 - 45 = ISO Message IDs
* Records 46 - 53 = ISO Processing Codes
*/
10 Tran Code 10="33, 42,52"
/* AMEX Sale */
11 Tran Code 11="33, 42,53"
/* AMEX Refund */
/* Transaction bitmaps defines for AMEX and Citi */
33 Bitmap 8 ="\ x30\ x20\ x05\ x80\ x00\ xc0\ x00\ x0c"
/* AMEX Financial */
/* Message IDs for AMEX and Citi */
42 Message ID 2 ="\ x02\ x00"
/* Financial Transaction */
/* Processing Codes for AMEX and Citi */
52 Proc Code 1 ="\ x00\ x40\ x00"
/* Sale, Offline Sale, Auth */
53 Proc Code 2 ="\ x02\ x40\ x00"
/* Void DB, DB Adjust */
```

prot_8583()

This function is called by the PIP Engine and provides the following functionality:

- an interface to the ISO 8583 Message Engine to assemble and disassemble packets.
- an interface to the application-defined validation function used to determine if an appropriate response was received.
- an interface to the application-defined communications function used to transmit and receive request/response messages.
- processes reversals as specified in the ISO 8583 standard.

NOTE



- Verix eVo ACT library checks for the cancel key operation during PIP transaction.
- If the DISABLECANKEYPIP `config.sys` entry is set to 1, then Verix eVo ACT library ignores the cancel key pressed during the PIP transaction.
- If the DISABLECANKEYPIP `config.sys` entry is set to 0 or the config entry is not found, then Verix eVo ACT library checks for the cancel key press and aborts the PIP transaction if the cancel key is pressed.

Prototype

```
#include <prot8583.h>
#include <aclfile.h>
int prot_8583(COMM_8583_DATA *control_data, int rev_opt,
              unsigned resp_timeout);
```

Parameters

<code>control_data</code>	Must be a properly initialized COMM_8583_DATA structure.
<code>rev_opt</code>	Determines the reversal processing required and must be one of the following, (as defined in PROT8583.H): <ul style="list-style-type: none"> • REV_ON: Normal reversal processing. • REV_OFF: No reversal processing. • REV_ONLY: Process only a pending reversal, if one exists.
<code>resp_timeout</code>	Defines the time, in seconds, allowed to receive a response to a request. This time-out is passed to the transceiver function and is not referenced internally.

Return Values

Success:	PROT8583_SUCCESS
Failure:	PROT8583_PARM_ERROR: Illegal parameter passed in the call. PROT8583_REVERSAL_FILE_ERROR: Error occurred accessing the reversal file. PROT8583_FAILED_REVERSAL: Error occurred processing the reversal. PROT8583_CREATED_REVERSAL: Request resulted in a reversal beings created.

PROT8583_FAILED_TRANSMIT: Request could not be processed.

PROT8583_CLR_KEY: `gu_clr_state` variable was set to 1 on a clear key press.

PROT8583_UNPACK_ERROR: Error occurred unpacking the response.

-17: A field is not defined in the field table.

-18: While parsing the outgoing packet, more data was stored in the buffer than the specified limit (packet size).

-19: While unpacking the incoming packet, the packet size is larger than the size specified in the field table.

-20: While unpacking the incoming packet, the data parsed for a variable is larger than the size specified in the field table.

-21: No matching variant in the variant table.

The `errno` variable returns additional error information. Always examine `errno` if the function fails. Error return values are defined in `PROT8583.H`. Any errors passed from the transmit/receive and any errors from the validation process are also returned.

NOTE


If `resp_timeout` is zero, the member in the control data structure, `comm_timeout`, must be greater than zero.

Two functions must be provided: the transmit/receive function and the validation function. Both of these functions must have a return type of `int`.

The validation function is called after receiving a response. This function can return the following values:

Success: 0

Failure: -1 to -14: Use if the validation routine fails and you do not want to wait for another response packet.

≥ 1: Indicates the new time-out value (in seconds) to use to wait for another response packet.

RETRY_CURRENT_TIMEOUT: Waits for another response packet using the current time-out value.

prot8583_main()

Passes a function pointer to a callback function used to modify the contents of reversal data if required by the specific host. If the application does not want to update the contents, a null pointer is passed.

Prototype

```
#include <prot8583.h>
int prot8583_main(RevFxn func);
typedef int(*RevFxn) (COMM_8583_DATA *control_data);
```

Parameters

RevFxn func User defined function pointer to modify the reversal data.

Return Values

None.

Dependencies

[prot8583_main\(\)](#) must be called by the application before performing any PIP-related transactions.

save_8583_tpdu_msg_id()

Copies the TPDU and message ID from the application variable to the destination buffer. This function is primarily used by the [pip_trans\(\)](#) and [prot_8583\(\)](#) functions.

Prototype

```
#include <prot8583.h>
#include <aclfile.h>
void save_8583_tpdu_msg_id(COMM_8583_DATA *control, unsigned char
                           *dest_tpdu, unsigned char *dest_msg_id);
```

Parameters

control Pointer to the structure used to find the variable declared in the field table.

dest_tpdu Pointer to the buffer to store the TPDU.

dest_msg_id Pointer to the buffer to store the message ID.

Return Values

None.

set_tpdu_length()

Allows the user to set the length of the TPDU field for the PIP processing.

NOTE



The device handle must be valid.

Prototype

```
#include <iso8583.h>
int set_tpdu_length(int inTPDULength);
```

Parameters

`inTPDULength` Length of the TPDU field. Pass zero if absent.

Return Values

Success: `TPDU_PARAM_SUCCESS`

Failure: -1 to -14: Use if the validation routine fails and you do not want to wait for another response packet. `INVLD_ORDER` (-2)

`TPDU_PARAM_INVALID` (< 0)

`TPDU_PARAM_ERROR` (> `MAX_TPDU_LENGTH` (32 bytes))

NOTE



`set_tpdu_length()` must be called by the application if the TPDU field is present and its length is not 5 bytes (default value).

Data Capture Engine Function Calls

The Data Capture Engine allows short integers, long integers, characters, and strings to be stored and retrieved from keyed files. The calling application specifies the application variable indicating the source or destination of the data, the files for storage, the operation desired (write or read), and the type of conversion required. The Data Capture Engine completes the conversion and accesses the data file as required. This is accomplished using a single function, `dce_key_cvt()`.

This section describes the following data capture function calls:

- `dce_key_cvt()`
- `DCE_PUTCFG_C()`
- `DCE_GETCFG_C()`
- `DCE_PUTCFG_S()`
- `DCE_GETCFG_S()`

The application programmer can create other macros as required, but should not modify the standard macros provided in `ACDCE.H`. Any modification of standard macros can introduce confusion and problems in the maintenance of the application.

dce_key_cvt()

Access function for Data Capture Engine keyed file. This function permits application data to be written to and read from keyed files, and completes any required data conversion using functions provided by the ACL.

The Data Capture Engine does not attempt to create a keyed file; the file must exist prior to requesting any Data Capture Engine operation. The specified file does not have to be open; this function opens the file, performs the operation, and closes it automatically. If the file is not already open, a file handle is obtained from the operating system, and released on completion of the operation.

The data is not validated to be of the type specified in the call. The caller is expected to call the Data Capture Engine and correctly specify the data type. The result of requesting an operation with an incorrect data type is undefined.

Prototype

```
#include <acdce.h>

int dce_key_cvt(unsigned char process, char *file, char *key, unsigned
               char cnvt, DCE_KEY_DATA data);
```

Parameters

process	Valid values are: <ul style="list-style-type: none"> • DCE_READ • DCE_WRITE
file	Name of the keyed file.
key	Keyed record identifier.
cnvt	Data type valid values are: <ul style="list-style-type: none"> • DCE_I • DCE_UI • DCE_L • DCE_UL • DCE_C • DCE_S
data	Source or destination data address.

Return Values

Success:	1
Failure:	-1: Filename is null.
	-2: Key is null or does not exist.
	-3: Unsupported data type specified.
	-4: File handle or file not available.

NOTE

- When reading previously stored information, the data type can be specified as a character string. This may be useful when the data is to be displayed, printed, and so on.
- For example, storing the value 100 to a keyed file results in the conversion of the value to the ASCII string "100". If the value is then read as a string variable, the string "100" is placed in the application buffer. This does not affect the ability to later read the data as a signed or unsigned integer.
- This concept applies to each of the macros listed in this manual that end with _I, _L, _UI and _UL.

See Also `putkey()`, `getkey()`

Dependencies **ACL:** `str2digit()`, `str2int()`, `str2long()`, `int2str()`, `long2str()`

DCE_GETCFG_C()

This macro call is the same as [dce_key_cvt\(\)](#), except that the filename is supplied as CONFIG.SYS, the process is supplied as read, and the data type is supplied as character.

Prototype

```
#include <acdce.h>
int DCE_GETCFG_C(char *key, DCE_KEY_DATA data);
```

Parameters

key Keyed record identifier.

data Destination data address.

Return Values

Success: 1

Failure: -2: key is null or does not exist.

See Also Application Construction Engine Data Capture function: [dce_key_cvt\(\)](#)

DCE_GETCFG_S()

This macro call is the same as the [dce_key_cvt\(\)](#) function, except that the filename is supplied as CONFIG.SYS, the process is supplied as read, and the data type is supplied as string.

Prototype

```
#include <acdce.h>
int DCE_GETCFG_S (char *key, DCE_KEY_DATA data);
```

Parameters

key Keyed record identifier.

data Address of source data.

Return Values

Success: 1

Failure: -2: key is null or does not exist.

See Also Application Construction Engine Data Capture function: [dce_key_cvt\(\)](#)

DCE_PUTCFG_C()

This macro call is the same as [dce_key_cvt\(\)](#), except that the filename is supplied as CONFIG.SYS, the process is supplied as write, and the data type is supplied as character.

Prototype

```
#include <acdce.h>
int DCE_PUTCFG_C(char *key, DCE_KEY_DATA data);
```

Parameters

key Keyed record identifier.
data Address of source data.

Return Values

Success: 1
Failure: -2: key is null.

See Also Application Construction Engine Data Capture function: [dce_key_cvt\(\)](#)

DCE_PUTCFG_S()

This macro call is the same as [dce_key_cvt\(\)](#), except that the filename is supplied as CONFIG.SYS, the process is supplied as write, and the data type is supplied as string.

Prototype

```
#include <acdce.h>
int DCE_PUTCFG_S(char *key, DCE_KEY_DATA data);
```

Parameters

key Keyed record identifier.
data Address of source data.

Return Values

Success: 1
Failure: -2: key is null.

See Also Application Construction Engine Data Capture function: [dce_key_cvt\(\)](#)

Modem Engine Function Calls

This section describes the following modem engine function calls:

- `inInitModem()`
- `inOpenModem()`
- `vdCheckDataMode()`
- `vdSetEchoMode()`
- `xhayes_control()`
- `xhayes_display()`
- `xhayes_flush()`
- `xhayes_response()`
- `xhayes_send_cmd()`
- `xhayes_status()`
- `xmdm_check_status()`
- `xmdm_checkline()`
- `xmdm_clear()`
- `xmdm_close()`
- `xmdm_dial_status()`
- `xmdm_failed_output_pending()`
- `xmdm_flush()`
- `xmdm_get_line_dial()`
- `xmdm_hangup()`
- `xmdm_init()`
- `xmdm_input_pending()`
- `xmdm_open()`
- `xmdm_output_pending()`
- `xmdm_receive_data()`
- `xmdm_send_data()`
- `xmdm_set_attrb()`
- `xmdm_set_protocol()`

inInitModem()

Initializes the modem by receiving environment variable numbers as parameters. It adds “M” with each number passed by the application and obtains the command strings specified by this environment variable.

Prototype

```
#include <XMODEM.H>

int inInitModem(int h_modem, int i_max_wait, int va_alist, ... );
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>max_wait</code>	Maximum amount time, in seconds, to wait for a response.
<code>va_alist</code>	Environment variable number of init string commands to send.

Return Values

Success:	SUCCESS: Initialization successful.
Failure:	E_INVALID_PARM: Invalid parameter passed.
	E_WRITE: Write to the modem fails.
	value < 0: Hayes response for the command.

Example

```
int max_wait;
int h_modem;
result = inInitModem(h_modem,5, 1,2,3,4,5,6,-1);
```

NOTE



This function can be used to send any number of initialize commands. It should be passed with numeric values. The function appends “M” to the passed numeric, and looks for the environment variables. So, for the above example, it looks for environment variables “M1”, “M2”, “M3”, “M4”, “M5,” and “M6”. The parameters should be terminated with -1.

If the CONFIG.SYS environment variable is non-populated, it skips and continues to the next CONFIG.SYS environment variable. The user is informed about this by means of LOG_PRINTF statement.

inOpenModem()

Opens the modem, as specified in `port` and obtains the resulting modem response. `h_modem` contains the handle returned by the `open()` call. It opens the port and sets the protocol parameters. If `port` is null or empty, then the device MDM_PORT is opened.

This call is a simplified version of `xmdm_open()`.

NOTE



This function opens the modem port, sets the baud and formats parameters. It does not send any initialization commands. To initialize the port, use `inInitModem()`.

Prototype

```
#include <XMODEM.H>

int inOpenModem(int *h_modem, char *port, int Baud, int Format);
```

Parameters

<code>h_modem</code>	Pointer to the modem handle.
<code>port</code>	Pointer to the modem port string.
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format.

Return Values

Success:	SUCCESS: Execution successful.
Failure:	E_WRITE_BLK: Failure to write port setup with MDM_WRITE_BLK().
	E_OPEN: open() command unsuccessful.

Example

```
int retVal;
int h_modem;
retVal = inOpenModem (&h_modem, NULL, Rt_2400, Fmt_A7N1);
```

vdCheckDataMode()

Sets the parameter for checking the data mode before sending any command to the modem. [xhayes_send_cmd\(\)](#) checks for this and if set to TRUE, checks for the data mode before sending the command. If set to FALSE, it does not perform any data mode checking. Default is TRUE for backward compatibility.

Prototype

```
#include <XMODEM.H>
void vdCheckDataMode(int mode);
```

Parameters

mode	Parameter to enable or disable check for data mode before sending the command.
------	--

Return Values None.

Example

```
vdCheckDataMode( TRUE );
vdCheckDataMode( FALSE );
```

vdSetEchoMode()

Sets the parameter for echo mode ON or OFF in the command response. It is used in conjunction with [xmdm_open\(\)](#). If echo mode is set to TRUE, the user must provide the init string with echo on. If echo mode is set to FALSE, the library initializes the port with echo off.


Prototype

```
#include <XMODEM.H>
void vdSetEchoMode(int setecho);
```

Parameters

setecho	Parameter to set or reset the echo mode command response.
---------	---

Return Values None.

NOTE  Call function before calling [xmdm_open\(\)](#). If set to TRUE, [xmdm_open\(\)](#) adds “E1” to the user-defined init string.

If set to FALSE, [xmdm_open\(\)](#) adds “E0” to the user-defined command string.

The default value is FALSE for backward compatibility.

Example

```
vdSetEchoMode( TRUE );
vdSetEchoMode( FALSE );
```

xhayes_control()

Sends a Hayes command to the modem and returns a Hayes status code.

NOTE



`xhayes_control()` uses `xhayes_send_cmd()` to send the command string and `xhayes_status()` to return the numeric response code once the command is sent.

Prototype

```
#include <XMODEM.H>

int xhayes_control(int h_modem, int h_clock, int max_wait, char command,
                  char *buffer);
```

Parameters

<code>h_modem</code>	Modem device handle.
<code>h_clock</code>	Clock device handle.
<code>max_wait</code>	Maximum wait time (seconds). If no limit on the amount of time in which to access the modem is desired, set <code>max_wait</code> to zero so that the function never times out, but only returns with the first response from the modem.
<code>command</code>	Command to execute.
<code>buffer</code>	Buffer address. For the Hayes "Dial" command ("D"), <code>buffer</code> must be a string containing the phone number and any other valid dial command characters, plus four additional bytes used internally for Hayes formatting. For the Hayes "Check Line" command ("-") and "Hang Up" command ("H"), <code>buffer</code> must be at least four bytes in length.

Return Values

Success:	0 - 91: Numeric Hayes response (see Table 17).
Failure:	E_FORMAT_CMD: Formatting of the Hayes command failed. E_WRITE_CMD: Hayes write command failed. E_HR_TIMEOUT: Allotted time specified by <code>max_wait</code> exceeded. E_READ_CMD: Command to access the Hayes response failed. E_ONLY_CR: Only a single carriage return was received in reply. Command was not "D", "H", or "-".

Example

The linked example code file demonstrates use of `xhayes_control()`.

xhayes_display()

Translates a Hayes modem response code to its descriptive text equivalent, storing the string at the specified buffer address.

Prototype

```
#include <XMODEM.H>
int xhayes_display(int hayes_code, char *buffer);
```

Parameters

hayes_code Hayes modem response code.

buffer Buffer address.

Return Values

Success: SUCCESS: Hayes response code is valid.

Failure: See [Table 17](#).

NOTE



XMODEM.H contains a list of valid Hayes response codes. The buffer must be at least 14 bytes in length to accommodate the returned string.

Invalid response codes are returned in `buffer`, and the function returns with FAILURE.

Table 17 **Hayes Failure Codes**

Numeric	Hayes Code	Hayes String
0	HAYES_OK	"OK"
1	CONNECT_300	"CONNECT 300"
2	RING_DETECT	"RING"
3	NO_CARRIER	"NO CARRIER"
4	HAYES_ERROR	"ERROR"
5	CONNECT_1200	"CONNECT 1200"
6	NO_DIALTONE	"NO DIAL TONE"
7	BUSY_DETECT	"BUSY"
8	NO_ANSWER	"NO ANSWER"
10	CONNECT_2400	"CONNECT 2400"
11	CONNECT_4800	Hayes "CONNECT 4800" status.
12	CONNECT_9600	Hayes "CONNECT 9600" status.
13	CONNECT_7200	Hayes "CONNECT 7200" status.
14	CONNECT_12000	Hayes "CONNECT 12000" status.
15	CONNECT_14000	Hayes "CONNECT 14000" status.
16	CONNECT_19200	Hayes "CONNECT 19200" status.
17	CONNECT_38400	Hayes "CONNECT 38400" status.
18	CONNECT_57600	Hayes "CONNECT 57600" status.
19	CONNECT_115200	Hayes "CONNECT 115200" status.

Table 17 Hayes Failure Codes (continued)

Numeric	Hayes Code	Hayes String
22	CONNECT_75TX_1200RX	Hayes "V.23 originate connection establish" status.
23	CONNECT_1200TX_75RX	Hayes "V.23 answer connection establish" status.
24	DELAYED	Hayes "COUNTRY SPECIFIC BLACKLIST" status.
32	BLACKLISTED	Hayes "number dialed is blacklisted" status.
35	DATA	Hayes "data modem connection" status.
40	CARRIER_300	Hayes "0-300 data rate" status.
44	CARRIER_1200_75	Hayes "V.23 backward channel carrier" status.
45	CARRIER_75_1200	Hayes "V.23 forward channel carrier" status.
46	CARRIER_1200	Hayes "1200 data rate" status.
47	CARRIER_2400	Hayes "2400 data rate" status.
48	CARRIER_4800	Hayes "4800 data rate" status.
49	CARRIER_7200	Hayes "7200 data rate" status.
50	CARRIER_9600	Hayes "9600 data rate" status.
51	CARRIER_12000	Hayes "12000 data rate" status.
52	CARRIER_14400	Hayes "14400 data rate" status.
66	COMPRESSION_CLASS5	Hayes "MNP CLASS 5 COMPRESSION 2400" status.
67	COMPRESSION_V42BIS	Hayes "MODEM CONNECT AT V42BIS" status.
69	COMPRESSION_NONE	Hayes "MODEM CONNECT WITH NO DATA COMPRESSION" status.
70	PROTOCOL_NONE	Hayes "NO ERROR CORRECTION PROTOCOL" status.
77	PROTOCOL_LAPM	Hayes "PROTOCOL LAPM" status.
80	PROTOCOL_ALT	Hayes "PROTOCOL ALT" status.
90	VFI_NO_LINE	"VFI NO LINE"
91	VFI_DIALTONE	"VFI DIAL TONE"

Example

The linked example code file demonstrates use of xhayes_display().

xhayes_flush()

Clears the command response buffer from the modem device. The `read()` function is called until the response buffer is empty. This function returns immediately.

NOTE

Issuing a call to `xhayes_flush()` immediately after `xhayes_send_cmd()` or `write()`, effectively results in the buffer not being cleared. Some commands, such as “H” (hangup/go on-hook), require more than one second of real time to complete and elicit a valid Hayes response to the request. A 100-ms delay is generally adequate between commands and responses.

`xhayes_flush()` can be useful when called just before a dial command to remove old responses that may still be in the modem devices response queue.

Prototype

```
#include <XMODEM.H>
int xhayes_flush(int h_modem);
```

Parameters

`h_modem` Opened modem device handle.

Return Values

Success: SUCCESS: The Hayes response buffer cleared successfully.

Failure: E_READ_CMD: `read()` function returned FAILURE.

Example

The linked example code file demonstrates use of `xhayes_flush()`.

xhayes_response()

Reads a Hayes modem command response and converts the command into a Hayes code integer value. Use [xhayes_response\(\)](#) to obtain a valid Hayes response code following any modem command.

Prototype

```
#include <XMODEM.H>

int xhayes_response(int h_modem, int h_clock, int max_wait);
```

Parameters

h_modem	Opened modem device handle.
h_clock	Clock device handle.
max_wait	Maximum amount time, in seconds, to wait for a response. If no timeout is desired, set max_wait to zero

Return Values

The value returned is the Hayes numeric response value, which is an integer and is in the allowed range of valid Hayes numeric response codes, see [Table 8](#).

Failure: E_READ_CMD: The command to access status response was unsuccessful.

E_HR_TIMEOUT: A response cannot be obtained in less than or equal to the number of seconds specified by max_wait.

E_ONLY_CR: Hayes response returned from the modem is a single carriage return only.

NOTE



[xhayes_display\(\)](#) can be called to convert the Hayes response return value to a display string. If echo is enabled then the echo response is ignored and the Hayes numeric response is returned.

Example

The linked example code file demonstrates use of [xhayes_response\(\)](#).

xhayes_send_cmd()

Converts the input string into Hayes format and sends the resulting command string to the modem.

If [vdCheckDataMode\(\)](#) is set to TRUE, this function checks if the modem is in data mode or command mode before sending the command.

If [vdCheckDataMode\(\)](#) is set to FALSE, this function does not perform any checking before sending the command.

By default, [vdCheckDataMode\(\)](#) is set to TRUE so the function performs a data mode check.

Prototype

```
#include <XMODEM.H>
int xhayes_send_cmd(int h_modem, char *cmd_buff);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>cmd_buff</code>	Command buffer string. <code>cmd_buff</code> points to the command buffer string that contains a valid, null-terminated string, such as "D555-1234", "H" or "S0= 1". It must also be large enough to accommodate two additional command characters, "AT" and <CR> (three bytes total). Upon return, <code>cmd_buff</code> contains the fully formatted Hayes command. If the returned length of <code>cmd_buff</code> is not equal to the return value of xhayes_send_cmd() , an error occurred.

Return Values

The number of bytes written to the modem.

Failure: E_WRITE_CMD: Unsuccessful write to modem. This value is returned as a result of `write()` returning an error code.

NOTE



The value returned is the number of bytes sent to the modem. Any value less than 3 should be interpreted as a send command error, since "AT\r" is the shortest possible command.

Example

The linked example code file demonstrates use of [xhayes_send_cmd\(\)](#).

xhayes_status()

Obtains the Hayes response from the mode. This function calls the [xhayes_response\(\)](#) and returns the response.

Prototype

```
#include <XMODEM.H>
int xhayes_status(int h_modem, int h_clock, int wait_time);
```

Parameters

<code>h_modem</code>	Modem handle.
<code>h_clock</code>	Clock device handle.
<code>wait_time</code>	Specifies the maximum amount of time, in seconds, to receive the Hayes response. Set this to zero if no time out is desired.

Return Values

Refer to the [xhayes_response\(\)](#) return values.

Example

The linked example code file demonstrates use of [xhayes_status\(\)](#).

xmdm_check_status()

Returns the current modem signal information, including CTS, DCD, framing error, overrun error, parity error, and break or abort. The `stat_map` parameter indicates which signal(s) to check. Each signal is defined in `XMODEM.H` and can be combined for example, `E_PARITY_FRAME | E_FRAME_OVERRUN`.

Prototype

```
#include<XMODEM.H>
int xmdm_check_status(int h_modem, unsigned char stat_map);
```

Parameters

- | | |
|-----------------------|--|
| <code>h_modem</code> | Modem handle. |
| <code>stat_map</code> | Signal(s) of interest. Valid values are: <ul style="list-style-type: none">• <code>E_PARITY</code>• <code>E_OVERRUN</code>• <code>E_PARITY_OVERRUN</code>• <code>E_FRAMING</code>• <code>E_PARITY_FRAME</code>• <code>E_FRAME_OVERRUN</code>• <code>MDM_BREAK</code>• <code>MDM_CTS</code>• <code>MDM_DCD</code> |

Return Values

- | | |
|----------|---|
| Success: | 1: At least one signal in <code>stat_map</code> was set. |
| Failure: | 0: None of the signals in <code>stat_map</code> were set.
-1: Modem status command failure: <code>ioctl()</code> failed. |

Example

The linked example code file demonstrates use of `xmdm_check_status()`.

xmdm_checkline()

Checks if an active telephone line is connected to the terminal.

Prototype

```
#include <XMODEM.H>
int xmdm_checkline(int h_modem);
```

Parameters

h_modem Opened modem device handle.

Return Values

Success: SUCCESS: Phone line present.

Failure: VFI_NO_LINE: Phone line not present.

 E_WRITE: Write to the modem failed.

 E_HR_TIMEOUT: A response could not be obtained. Timed out.

Example

The linked example code file demonstrates use of [xmdm_checkline\(\)](#).

xmdm_clear()

Closes and re-opens the modem device.

NOTE

The modem device should be in initialized state before calling `xmdm_clear()` API. If `h_modem` is non-zero, `xmdm_clear()` refers to the opened modem device, if `h_modem` is zero, then `xmdm_clear()` API behaves similar to `xmdm_open()` API. `xmdm_set_protocol()` API is used to initialize the modem.

Prototype

```
#include <XMODEM.H>

int xmdm_clear(int h_modem, char *sz_mdm_name, int h_clock, int wait_time,
               int rate, int format);
```

Parameters

<code>h_modem</code>	Modem handle.
<code>sz_mdm_name</code>	Modem access path.
<code>h_clock</code>	Clock device handle.
<code>wait_time</code>	Maximum response time (seconds).
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format.

Return Values

Success: > 0: Successful initialization, open modem handle value returned.

Failure: E_OPEN: Failure to open modem with open().

E_CLOSE: Failure to close modem with close().

E_FW_READ_CMD: Modem opened. Unsuccessful read() while trying to access firmware status.

E_FW_ONLY_CR: Modem opened. Hayes response received during firmware open was single carriage return character, <CR>.

E_FW_TIMEOUT: Allotted time in which to receive Hayes response expired while obtaining firmware status.

E_FW_STATUS: Modem opened. Hayes error response as result of firmware initialization.

E_MI_READ_CMD: Modem opened. Unsuccessful read() while trying to access the CONFIG.SYS environment variable *MI.

E_MI_ONLY_CR: Modem opened. Hayes response received during *MI initialization was <CR>.

E_MI_TIMEOUT: Allotted time in which to receive Hayes response expired while obtaining *MI status.

E_MI_STATUS: Modem opened. Hayes error response as result of generating invalid Hayes command from *MI environment variable in CONFIG.SYS.

E_X_HR_TIMEOUT: Allotted time in which to receive Hayes response expired while obtaining *MI status in Vx680 terminal.

E_X_MI_TIMEOUT: Allotted time in which to receive Hayes response expired while obtaining *MI status in Vx520 terminal.

Example The linked example code file demonstrates use of `xmdm_clear()`.

xmdm_close()

First obtains modem status, then, closes the modem device based on input parameters and modem communication status. [xmdm_close\(\)](#) acts as a safe close so that, for example, if output is currently pending, the modem is not closed.

Prototype

```
#include <XMODEM.H>

int xmdm_close(int h_modem, int output_pend, int input_pend);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>output_pend</code>	Non-zero value causes the function to test for pending output. If output exists, the function does not close the modem, and an error code returns.
<code>input_pend</code>	Non-zero value causes the function to test for pending input. If input exists, the function does not close the modem and an error code returns.

Return Values

Success:	SUCCESS: Successful close.
Failure:	<p>OUTPUT_PENDING: Modem not closed; modem output pending and <code>output_pend</code>, is non-zero. Allow time for this condition to clear or call xmdm_flush().</p> <p>INPUT_PENDING: Modem not closed; input pending at modem, and <code>input_pend</code> is non-zero.</p> <p>OUTPUT_FAILED: Modem not closed; failed output records exist and <code>output_pend</code> is non-zero.</p> <p>FAILURE: <code>close()</code> function failed.</p> <p>E_STATUS_CMD: Error obtaining initial modem status.</p> <p>E_M_FLUSH: Flushing output buffer failed.</p>

Example

The linked example code file demonstrates use of `xmdm_close()`.

xmdm_dial_status()

Obtains the results from a Hayes dial command.

Prototype

```
#include <XMODEM.H>
int xmdm_dial_status(int h_modem, int h_clock, int max_wait);
```

Parameters

- h_modem** Opened modem device handle.
- h_clock** Clock device handle.
- max_wait** Maximum amount time, in seconds, to wait for a response.
- If **max_wait** is zero, `xmdm_dial_status()` does not time out. The default wait-for-carrier detect after dialing for Verix eVo modems is 30 seconds. This 30 second wait can be changed through the S7 modem register or by setting the *S7 environment variable in CONFIG.SYS.
- The *S7= 55 entry causes the modem device to wait up to 55 seconds for answer and carrier detect.

Return Values

- Failure:** **E_HR_TIMEOUT:** Hayes response not obtained before **max_wait** elapsed.
- E_READ_CMD:** Error accessing modem status.
- E_ONLY_CR:** Hayes response = <CR> only. Any other response received (such as, "OK") causes `xmdm_dial_status()` to return a value of **E_INVALID_RESP**.
- E_INVALID_RESP:** Hayes response does not pertain to a dial command; response is invalid.

Value	Description
CONNECT_300	Dial and connect successful; 300 baud.
NO_CARRIER	No carrier detected.
HAYES_ERROR	Hayes error occurred in the dial command.
CONNECT_1200	Dial and connect successful; 1200 baud.
NO_DIALTONE	Indicates no dialtone present. This status exists only if dialtone is enabled with the S60= 0 command (the default setting).
BUSY_DETECT	Indicates line is busy. This status exists only if busy detection is enabled. The X4 command (default) enables both dialtone and busy detect.
NO_ANSWER	Hayes silence; phone not answered.
VFI_NO_LINE	VeriFone unique; phone line does not exist.
VFI_DIALDONE	VeriFones dial done; dialing complete.

Example

The linked example code file demonstrates use of `xmdm_dial_status()`.

xmdm_failed_output_pending()

Determines if there are any failed output messages pending.

Prototype

```
#include <XMODEM.H>
int xmdm_failed_output_pending(int h_modem);
```

Parameters

h_modem Opened modem device handle.

Return Values

Success: SUCCESS: No failed output pending.

Failure: FAILURE: Status access failure.

1: At least one failed output message pending.

Example

The linked example code file demonstrates use of `xmdm_failed_output_pending()`.

xmdm_flush()

Clears the modem data read receive buffer.

NOTE



This function can be useful just before closing the modem. Also see `xmdm_close()` and `xmdm_set_attrib()`, which both attempt to flush the output buffer.

Prototype

```
#include <XMODEM.H>
int xmdm_flush(int h_modem);
```

Parameters

h_modem Opened modem device handle.

Return Values

Success: SUCCESS: Modem buffer and reject queue flushed.

Failure: E_READ: Error clearing modem input buffer.

Example

The linked example code file demonstrates use of `xmdm_flush()`.

xmdm_get_line_dial()

Checks for the presence of a phone line. If the line is available, [xmdm_get_line_dial\(\)](#) begins dialing the number set in `dial_string`. This causes the modem to dial, and begin waiting for the carrier. If the carrier is detected, the modem connects and goes online. `max_wait` determines how many seconds to wait for the dial response.

Prototype

```
#include <XMODEM.H>

int xmdm_get_line_dial(int h_modem, char *dial_string, int *iwrite,
                      int h_clock, int max_wait);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>dial_string</code>	Phone number to dial.
<code>iwrite</code>	<code>write()</code> return value.
<code>h_clock</code>	Clock device handle.
<code>max_wait</code>	Maximum amount time, in seconds, to wait for a response.

NOTE



`dial_string` must be a null-terminated string containing valid dialing information (see [Table 18](#)), and must be large enough to accommodate the four extra command characters used by this function. The longest Hayes command that can be sent is 40 bytes.

When [xmdm_get_line_dial\(\)](#) returns, `iwrite` contains the number of bytes written to the modem command buffer, and `dial_string` contains the complete dial command string whose string length should be equal to `iwrite` for a successful dial.

If `max_wait` is set to zero, this function does not time out, but returns when the first Hayes response is received.

Return Values

- Failure:
- `E_INVALID_RESP`: Obtained invalid dial status response such as "OK".
 - `E_WRITE_CMD`: Command to request line status failed.
 - `E_READ_CMD`: Command to access line status failed.
 - `E_HR_TIMEOUT`: Could not obtain line status response in time specified in `max_wait`.
 - `E_ONLY_CR`: Hayes response = <CR> only.

Value	Description
CONNECT_300	Connection made; 300 baud.
CONNECT_1200	Connection made; 1200 baud.
NO_CARRIER	Connection not established.
HAYES_ERROR	Modem error occurred.
NO_DIALTONE	No dial tone present on the phone line.
BUSY_DETECT	Called number was busy.
NO_ANSWER	Called number did not answer.
VFI_NO_LINE	VeriFone-defined; no phone line status.
VFI_DIALDONE	VeriFone-defined; dialing completed status.

CAUTION

If auto-answer is enabled and a call comes in just before dialing out, `xmdm_get_line_dial()` does not consider this a valid Hayes response to dialing, and returns the value `E_INVALID_RESP`. This can be unpredictable. Setting auto-answer to off (`ATS0= 0`) just before dialing prevents this condition.

Table 18 **Dial Command String Modifiers**

Hex Code	ASCII Code	Command Description
30h–39h	'0'–'9'	Digits pulse/tone dialed.
41h–44h	'A'–'D'	Digits pulse/tone dialed.
23h, 2Ah	'#', '*'	Digits tone dial only.
50h	'P'	Switch to pulse dial.
54h	'T'	Switch to tone dial.
56h	'V'	Verify answer tone (Bell 103/ Bell 212).
57h	'W'	Wait for dial tone or (S7) seconds.
2Ch	','	Comma; wait for (S8) seconds.
3Bh	';'	Semicolon; eliminate the handshake.
20h	' '	Space/blank is ignored.
2Dh	'-'	Dash/hyphen is ignored.
Any other characters or unimplemented digits cause a return code of 4 (ERROR).		

Example

The linked example code file demonstrates use of `xmdm_get_line_dial()`.

xmdm_hangup()

Instructs the modem unit to go on-hook (disconnect) from the phone line.

Prototype

```
#include <XMODEM.H>
int xmdm_hangup(int h_modem, int h_clock, int max_wait);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>h_clock</code>	Clock device handle.
<code>max_wait</code>	Parameter is not used in the library, but is retained for backward compatibility.

Return Values

Success:	SUCCESS: Successful execution.
Failure:	E_READ: Error reading modem buffer; read() returned -1. E_WRITE_BLK: Failure to write modem parameters in Vx520 terminal. E_USBMDM_SETSRLFAILED : Failure to write modem parameters in Vx680 terminal.

NOTE



Calling `xmdm_hangup()` to a modem that is currently on-hook has no effect. The hangup function always terminates the connection, regardless of who originated the call. (Closing the modem also guarantees a disconnect.)

Example

The linked example code file demonstrates use of `xmdm_hangup()`.

xmdm_init()

Initializes the modem by opening the device and setting the communications parameters to the specified protocol. If the modem device is already open, it is closed and then re-opened. If the modem is not open, the `h_modem` parameter must be set to zero. This function is equivalent to calling `xmdm_clear()` and `xmdm_set_protocol()`.

Prototype

```
#include <XMODEM.H>

int xmdm_init(int h_modem, char *sz_mdm_name, int h_clock, int max_wait,
              int rate, int format);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>sz_mdm_name</code>	Path to modem device.
<code>h_clock</code>	Clock device handle.
<code>max_wait</code>	Maximum amount time, in seconds, to wait for a response.
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format.

Return Values

Success: > 0: Successful initialization; modem device handle is returned.

Failure: E_OPEN: Failure to open modem with `open()`.

E_CLOSE: Failure to close modem with `close()`; modem handle on input was non-zero, but did not reference an opened device.

FAILURE: Error writing modem protocol parameters.

Table 19 **Firmware Errors**

Value	Description
E_FW_READ_CMD	Modem opened; Unsuccessful <code>read()</code> while trying to access firmware status.
E_FW_TIMEOUT	Allotted time to receive the Hayes response expired while obtaining firmware status. This function does not time out if <code>max_wait</code> is set to zero.
E_FW_STATUS	Modem opened; Hayes error response as result of firmware initialization.
E_FW_ONLY_CR	Modem opened; Hayes response received during firmware open was a single <CR>.

Table 20 CONFIG.SYS *MI Initialization Errors

Value	Description
E_MI_READ_CMD	Modem opened; Unsuccessful read() = -1 while trying to access *MI entry. This is received when *MI contains a bad Hayes command.
E_MI_TIMEOUT	Allotted time to receive the Hayes response expired while obtaining *MI status.
E_MI_STATUS	Modem opened; Hayes error response as result of generating invalid Hayes command from *MI command in CONFIG.SYS.
E_MI_ONLY_CR	Modem opened; Hayes response received during *MI initialization is only a single <CR>.

Example

The linked example code file demonstrates use of `xmdm_init()`.

xmdm_input_pending()

Determines if there are any input messages pending.

Prototype

```
#include <XMODEM.H>
int xmdm_input_pending(int h_modem);
```

Parameters

`h_modem` Modem handle.

Return Values

Success: SUCCESS: No input pending.

Failure: 1: At least one input message pending.

 FAILURE: Status access failure.

Example

The linked example code file demonstrates use of `xmdm_input_pending()`.

xmdm_open()

Opens the modem device specified in `path` and obtains the resulting modem responses.

Prototype

```
#include <XMODEM.H>

int xmdm_open(int *h_modem, char *path, int h_clock, int max_wait, int
              rate, int format);
```

Parameters

<code>h_modem</code>	Pointer to the modem handle.
<code>path</code>	Pointer to the modem path string.
<code>h_clock</code>	Clock device handle.
<code>max_wait</code>	Maximum amount time, in seconds, to wait for a response.
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format.

Return Values

Value	Description
HAYES_OK	Successful execution.
E_OPEN	open() command unsuccessful.
E_FW_READ_CMD	Firmware error; initial read() failed.
E_FW_TIMEOUT	Firmware error; too long to execute.
E_FW_STATUS	Firmware error; Hayes response status.
E_FW_ONLY_CR	Firmware error; modem opened; Hayes response received during firmware open was single carriage return character, <CR>.
E_WRITE_BLK	Failure to write port setup with MDM_WRITE_BLK().

Table 21 CONFIG.SYS *MI Entry Errors

Value	Description
E_MI_READ_CMD	read() failed.
E_MI_TIMEOUT	Modem busy; too long to execute. This function should not timeout. If it does, set <code>max_wait</code> to zero so it cannot timeout.
E_MI_STATUS	Hayes response status is ERROR.
E_MI_ONLY_CR	Modem opened. Hayes response received during *MI initialization was a single carriage return character.

NOTE



The modem device remains inactive until a call is made to initialize the parameters such as baud rate, format, and so on. See the `xmdm_set_protocol()` function. If `path` is null or empty, then the device MDM_PORT is opened.

NOTE

It is highly recommended to flush the port after `xmdm_open` using `xmdm_flush()`. `*MI CONFIG.SYS` environment variable is used to specify the modem init string in the Verix eVo terminal. It is used while opening the modem. Since E0V0Q0 is added by the library, the user is not required to provide this as part of `*MI` variable.

Example

The linked example code file demonstrates use of `xmdm_open()`.

xmdm_output_pending()

Determines if there are any output messages pending.

Prototype `#include <XMODEM.H>`
 `int xmdm_output_pending(int h_modem);`

Parameters

<code>h_modem</code>	Opened modem device handle.
----------------------	-----------------------------

Return Values

Success:	SUCCESS: No output pending.
Failure:	FAILURE: Status access failure.
	1: At least one output message pending.

Example The linked example code file demonstrates use of `xmdm_output_pending()`.

xmdm_receive_data()

Waits until a specified incoming data, with maximum positive number of bytes from the modem port or a pause in reception, or a timeout expires on a first come first basis. If there are more data pending than specified, it will be ignored.

Prototype

```
#include <XMODEM.H>

int xmdm_receive_data(int h_modem, char *buffer, int min, int max,
int max_wait);
```

Parameters

<code>h_modem</code>	Handle to the modem comport.
<code>buffer</code>	Pointer to receive buffer.
<code>min</code>	Not used in the library. Maintained for backward compatibility.
<code>max</code>	Maximum number of characters to receive. Must be greater than zero.
<code>max_wait</code>	Maximum receive time in 10-ms increments.

Return Values

Success:	> 0: Number of bytes read
Failure:	E_INVALID_PARM: Invalid parameter passed.
	E_READ: Error reading input data, read() = -1.
	E_X_HR_TIMEOUT: Time out reached before 'max' number of bytes read.
	E_X_NO_CARRIER: Carrier lost while receiving data.

Example

The linked example code file demonstrates use of `xmdm_receive_data()`.

xmdm_send_data()

Provides a timed transmission of data in `buffer` to the modem device.

The maximum transfer length allowed by this function is 32767 bytes. In packet mode, transfers should be limited to 254 bytes. The number of bytes actually transmitted depends on available modem buffer space.

Prototype

```
#include <XMODEM.H>

int xmdm_send_data(int h_modem, char *buffer, int buff_len, int max_wait);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>buffer</code>	Data to send to modem.
<code>buff_len</code>	Buffer length.
<code>max_wait</code>	Maximum transmit time in 10-ms increments.

NOTE



If `max_wait` is zero (0), the routine makes a single attempt to transfer data to the modem. If the execution of the data transfer fails, `xmdm_send_data()` returns FAILURE.

Return Values

Success:	> 0: Contains the number of bytes transferred. This value should equal <code>buff_len</code> on successful transfer. Since the output is buffered by the modem device and sent at a controlled rate, the output is pending at the time this function returns. To determine if all buffered output was sent, call <code>xmdm_checkline()</code> until a value of zero returns.
Failure:	FAILURE: Error writing to modem port. E_NOCARRIER: Carrier lost while trying to send data. E_PARTIAL: Sent only part of buffer. Check <code>errno</code> for the number of bytes sent. E_STATUS_CMD: Unable to obtain modem status when sending buffers larger than 200 bytes.

If only part of the buffer is transmitted,

E_PARTIAL is returned, with the number of bytes sent in `errno`.

If `max_wait` is a non-zero value, `xmdm_send_data()` times the transmission, and returns with a time out error if the buffer could not be transmitted in the specified amount of time. In a time-out condition, the actual number of bytes sent returned in `errno`.

Example

The linked example code file demonstrates use of `xmdm_send_data()`.

xmdm_set_attrb()

Changes a single modem attribute without affecting previously established modem attributes. This function is for on-the-fly changes when the modem has already been initialized. [xmdm_set_attrb\(\)](#) was written with the character, packet, and SDLC standard protocols in mind.

Specific `#defines` can be declared to make this function useful with custom protocols.

`attrib` identifies which modem attribute to change. `value` designates the value to assign to this attribute.

Prototype

```
#include <XMODEM.H>
int xmdm_set_attrb(int h_modem, int rate, int format, int flush);
```

Parameters

<code>h_modem</code>	Opened modem device handle.
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format.
<code>flush</code>	Flush buffers control flag. Once the modem attribute is reset, you can clear the Hayes and modem data. If this is the case, set flush to 1; otherwise, set flush to 0. Valid values are as follows: <ul style="list-style-type: none"> • 1 = Flush buffer. • 0 = Buffer not flushed.

Return Values

Success:	SUCCESS: Successful update of attribute.
Failure:	FAILURE: Invalid attribute, <code>attrib</code> . E_WRITE_BLK: Unable to update system modem attributes. E_READ: Unable to flush modem data buffer.

NOTE



Each attribute has a corresponding identification number. Attributes can be specific to a particular protocol. These values are from XMODEM.H.

Example

The linked example code file demonstrates use of [xmdm_set_attrb\(\)](#).

xmdm_set_protocol()

Initializes the communication data protocol attributes of the currently opened modem.

NOTE



The library supports a structure definition for each type of protocol. The programmer should load the appropriate structure with the intended attributes for the desired protocol. `xmdm_set_protocol()` initializes the modem with this information.

Prototype

```
#include <XMODEM.H>
int xmdm_set_protocol(int h_modem, int rate, int format);
```

Parameters

<code>h_modem</code>	Modem device handle.
<code>rate</code>	Rate of connection.
<code>format</code>	Connection format. Any parameter that is passed to the <code>format</code> variable, <code>Fmt_AFC</code> and <code>Fmt_DTR</code> gets appended which is a default setting for open block format.

Return Values

Success:	SUCCESS: Normal function completion.
Failure:	FAILURE: Unable to write in the modem initialization block. A negative number is returned for the following data communication baud rates: <ul style="list-style-type: none"> • 2400: -3 error value is returned. • 4800: -4 error value is returned. • 38.4 K: -7 error value is returned.

Example

The linked example code file demonstrates use of `xmdm_set_protocol()`.

Report Formatter Function Calls

This section describes the following report formatter function calls:

- `formater_close()`
- `formater_line()`
- `formater_main()`
- `formater_open()`
- `formater_template()`
- `formater_set_flags()`

Report Formatter Examples

Example

Generates the .FRM file using the FORMCVT tool. A sample text file for generating the .FRM file as follows.

```
1:1,"ITP Test Report"
3:1.30,"\C\DVALUECARD"
4:1.40,"\C\D"SPOT \W\HREPORT\B"
6:1,"\NTERM ID";14,"DATE";24,"TIME";32,"BONUS";38,"ART"
7:1.10,\L\Dterm_id;14.21,date;24.28,v_time;32.32,bonus;34,"WORK\B"
9:1,"TOTAL";19,"TOTAL"
10:3,"INT = $";10.16,int_ttl;21,"CUM = $";28.34,cum_ttl
12:1,"BILL";19,"BILL"
13:3,"INT = $";10.16,int_bil;21,"CUM = $";28.34,cum_bil
15:1,"CREDIT CARD";19,"CREDIT CARD"
16:3,"INT = $";10.20,int_crd;21,"CUM = $";28.34,cum_crd
20:1,"SUBJECT TO";19,"S_____ "
21:1,"\D\W\HLATER AUDIT\B";
22:10.30,\Nint_crd
23:1,"\N\Dtest value:";25.28,test;30.39,\Cname
25:1.12,\L\Dterm_id;13.26,\Bterm_id;27.40,\Cterm_id
27:1.12,\Lterm_id;13.18,\Lterm_id;27.40,\Rterm_id
29:1.12,\Lint_ttl;13.26,int_ttl;27.40,\Cv_time
31:3.23,"string constant";24,"test";30.39,"on left"
33:2.14,\D\Wtest;16.20,\B\Htest
35:2.14,\N\Ctest;16.23,\Ltest;30.39,test
37:2.14,\Cbonus;16.23,\Lbonus;30.39,bonus
39&x80000000:1,"Condition line x80000000"
39&x40000002:1,"Condition line x40000002"
39&x1:1,"Condition line x00000001"
41:1,"That's all folks..."
```

formater_close()

Closes the template file and returns to the caller.

Prototype `void formater_close (FORMATER *formdata);`

Parameters

<code>formdata</code>	Pointer to a FORMATER structure.
-----------------------	----------------------------------

Return Values None

Dependencies Verix eVo SDK

formater_line()

Formats a range of report lines. Formatting of conditional lines is based on the `condition` parameter. The formatted data is buffered into 200-byte buffers and passed to the output print function (assigned in the initialized output function).

NOTE



The following special printing modes are not handled by this function. They are processed by the output driver.

- Print red
- Restore print color to black
- Print double width
- Print double height
- Restore to normal print mode
- Send initialization command to printer

If a string constant or variable has a length longer than the column range specified in the template file, it is truncated to fit the specified range. The default for formatting a string constant is left justification; the default for formatting a variable is right justification.

When formatting a range of line numbers, problems may occur with system buffer usage. In this case, `formater_line()` returns -1 and stores the next line number to format in `error_line`. Using the contents of `error_line`, these types of errors can be processed.

Prototype

```
int formater_line(FORMATER *formdata, int start_line, int stop_line,
                 int print_blank_lines, unsigned long condition,
                 int *error_line);
```

Parameters

<code>formdata</code>	Pointer to a FORMATER structure.
<code>start_line</code>	Starting line number. If the line is not defined in the template and <code>print_blank_lines</code> is non-zero, blank lines print until the line number in the template is reached.
<code>stop_line</code>	Last line number to print. If this value is <code>TO_END</code> (-1), the template is processed through to the last line.
<code>print_blank_lines</code>	If zero, print undefined lines as blank lines; otherwise, ignore undefined lines.
<code>condition</code>	Value ANDed with the condition in each template record. If the result is not 0, the line prints; otherwise it is ignored. Note: Any line in the template that does not specify a condition is always printed.
<code>error_line</code>	Pointer to an <code>int</code> containing a line number being printed when the error occurred.

Note: The formatter keeps print data in 200-byte buffers. The value returned is the first line number in the current buffer.

Return Values

Success: 0

Failure: -1: An error condition. `error_line` is set to indicate where to resume, and `errno` may have been set by `output_print()`.

Example

```
while (formater_line (formatdata, start_line, TO_END, print_blank_lines,
condition, error_line) != 0)
start_line = *error_line;
```

Dependencies

Verix eVo SDK

formater_main()

Accepts a handle of `g_data` to be used by the application for formatting.

NOTE



Call `formater_main()` before using any other function.

Prototype

```
void formater_main(g_data *gvardata);
```

Parameters

`g_data gvardata` Structure holding application-specific information required by the formatter.

formater_open()

Accepts the handle of an open output device, a FORMATER structure address that holds information about the current formatter job, the name of the report template file, a pointer to the output initialization function, and a time-out value that initializes the output device. It also opens the template file, then calls the output initialization function. The result is returned to the application.

NOTE



The output initialization function fills in the FORMATER structure with pointers to the output drivers print and close functions.

If the output device is a direct interface to a communications port, the maximum number of slots is retrieved and saved so that it can control the maximum number of buffers used during output. Examples of indirect interfaces are spoolers and display drivers.

Prototype

```
int formater_open(int handle, FORMATER *formdata, char *template, int
(*output_initializer)(), int time_out);
```

Parameters

handle	Handle of an open communications device.
formdata	Pointer to a FORMATER structure which is filled in by the <code>formater_open()</code> function and used when calling other formatter functions.
template	Name of the template file.
*output_initializer	Pointer to a function that initializes the output driver, (for example, <code>p150dir</code>).
time_out	Time-out value used when initializing an output device. The initialized output can elect to ignore this parameter.

Return Values

Success: 0
Failure: -1: Template file cannot be opened.

The `formater_open()` function checks the return value from the user-written function passed in `*output_initializer`.

Any value less than zero is considered an error. This function stops execution and returns this error value to its calling routine.

The user-defined function should not return a value of -1, as the user function cannot tell whether a general function error has occurred or if the function is returning a "Template file cannot be opened" error.

Dependencies

Verix eVo SDK

formater_set_flags()

Builds the condition flag from user application-specific variables. It sets a 32-bit unsigned long bitmap flag to be used by the formatter to format any conditional lines of the report template.

NOTE



-1 always terminates the parameter list.

Prototype

```
unsigned long formater_set_flags(int va_alist, ...);
```

Parameters

`va_alist` Either 0 or non-zero; maximum of 32 (values > 32 shifts bits off the end).

Return Values

Unsigned long representation of the bit pattern.

NOTE



- The first value passed as the parameter to the function will be ignored, for example: 1,0,1,0,1,0 will be treated as 0,1,0,1,0.
- All bits are right justified in the return value by position.

Example

The linked example code file demonstrates use of `formater_set_flags()`.

Dependencies

Verix eVo SDK

formater_template()

Specifies which template file to use.

Prototype

```
int formater_template(FORMATER *formdata, char *template);
```

Parameters

`formdata` Pointer to a FORMATER structure.

`template` Name of the template file.

Return Values

Success: 0

Failure: -1: Template file cannot be opened.

Dependencies

Verix eVo SDK

Database Library Function Calls

This section provides descriptions of the following database library function calls:

- `db_close()`
- `db_delete()`
- `db_get_key()`
- `db_open()`
- `db_read()`
- `db_remove()`
- `db_seek_key()`
- `db_seek_record()`
- `db_write()`

Database Library Examples

The following linked files illustrate the use of the function calls provided by the Database library, with examples.

TBatch Key Function

In the [TBatch Key Function](#) example, a batch implementation for a typical application is included. The structure `TBatchRec` is the definition of the batch record and `TBatchKey` is the definition of the key structure to access the batch records.

Example

The linked example code file demonstrates use of `TBatchKey()`.

db_close()

Closes the database and index files.

Prototype

```
#include <dbmgr.h>
short db_close(DB_FILE *db_ptr);
```

Parameters

`db_ptr` Pointer to the database file structure.

Return Values

Success: Positive: Successful operation.

Failure: `DB_INVALID_PARAM`: `db_ptr` is NULL.

-1: System error. `errno` contains the specific error code.

db_delete()

Deletes one or more records from the database and index files.

NOTE



The delete operation slows the application down, so the offsets can be updated.

If the start record plus the number of records to delete is greater than the actual number of records in the file, all records in the file starting from the specified record number are deleted. There is no error message. The argument `rec_cnt` includes the start record number (`rec_num`). If `rec_num = 1` and `rec_cnt = 0`, no record is deleted and 0 is returned.

Prototype

```
#include <dbmgr.h>
short db_delete(DB_FILE *db_ptr, long rec_num, long rec_cnt);
```

Parameters

<code>db_ptr</code>	Pointer to database file structure.
<code>rec_num</code>	The record number to start deleting from.
<code>rec_cnt</code>	The number of consecutive records to delete.

Return Values

Success:	Positive (≥ 0): Successful operation. The number returned indicates the number of records deleted.
Failure:	DB_INVALID_PARAM: <code>db_ptr</code> is NULL. Negative: One of the following errors: <ul style="list-style-type: none">• System error. <code>errno</code> contains the specific error code.• DB_INVALID_REC_NUM: Starting record is out of range.• DB_INVALID_REC_CNT: Record count invalid; negative not allowed.

db_get_key()

Finds the specified key value.

Prototype

```
#include <dbmgr.h>

long db_get_key(DB_FILE *db_ptr, void *key_struct, void *Matchstr,
               long action, char *databuf, int dlen, long *rec_num);
```

Parameters

<code>db_ptr</code>	Pointer to the database file structure.
<code>key_struct</code>	The buffer to contain the key structure.
<code>Matchstr</code>	See Match Structure .
<code>action</code>	Find criteria. Valid values are: <ul style="list-style-type: none"> • <code>DB_FIRST</code>: Start from the beginning of file and search forward to find the first key match. • <code>DB_LAST</code>: Start from the end of file and search backward to find the first key match. • <code>DB_PREVIOUS</code>: Find the previous key from the present position. • <code>DB_NEXT</code>: Find the next key from the present position. • <code>n</code>: nth occurrence of the key relative to the current position. If <code>n > 0</code>, then search forward.
<code>databuf</code>	Pointer to the data buffer.
<code>dlen</code>	Length of the data record.
<code>rec_num</code>	Address of the <code>rec_num</code> that is updated when a successful comparison occurs.

Match Structure

This structure must be created by the programmer. The first element in this structure must be a pointer to a compare function written by the programmer. The other elements of the key structure are values to compare against. This comparison function must return 0 if a match occurs. When [db_seek_key\(\)](#) calls the comparison function, it passes the address of the `key_struct` buffer (this buffer contains the key structure read by the Database library) and the Match structure.

[db_get_key\(\)](#) calls [db_seek_key\(\)](#) internally to obtain the record number of the specified database record and reads the record into the data buffer for subsequent processing.

NOTE

If the function returns a value greater than 0, then the `rec_num` and `key_struct` are updated. If you search for the 5th occurrence but only 1 key exists, then `rec_num` and `key_struct` are updated anyway. In this case, `db_seek_key()` returns 1 and not 5.

The prototype of the compare function is:

```
short (*comp_func)(void *,void *);
```

where, the first argument is the key structure and the second argument is the Match structure.

Return Values

- Success: 1: The requested number of occurrences were found. (`rec_num` and `key_str` are updated).
- Failure: 0: No key was found or the requested number of occurrences were not found.
- DB_INVALID_PARAM: `db_ptr` is NULL.
- DB_INVALID_BUF_SIZE: The length of the buffer size (`dlen`) is zero.
- DB_INVALID_RANGE: `dlen` is not between `DB_MAX_INT` and `DB_MIN_INT` (defined in `dbmgr.h`).
- 1: System error. `errno` contains the specific error code.

NOTE

The compare function that compares the specified search key to the corresponding key in the index file record must be provided by the programmer. If this function is not provided, `db_seek_key()` and `db_get_key()` fail.

Example

The linked example code file demonstrates use of `db_get_key()`.

db_open()

Creates a database file and the corresponding index file and updates the database file structure. If a database file is already created, `db_open()` opens the existing file.

NOTE



- Do *not* provide a filename extension. This function does not check for a filename extension. The filename should not be more than `DB_MAX_FILENAME_SIZE` (= 26) characters.
- In flash, `db_open()` API allows only one file to open in the `O_RDWR` mode.
- Refer to the *Verix eVo OS Programmers Manual* for more details about the restrictions on the usage of flash file.

Prototype

```
#include <dbmgr.h>
short db_open(DB_FILE *db_ptr, char *filename, int key_len, int mode);
```

Parameters

<code>db_ptr</code>	Pointer to database file structure.
<code>filename</code>	The filename with <i>no</i> extension.
<code>key_len</code>	The length of key structure in bytes.
<code>mode</code>	The file creation mode as described in <code>fnctl.h</code> (the standard C/C++ library include file). Valid values are: <ul style="list-style-type: none"> • <code>O_RDWR</code> • <code>O_EXCL</code> • <code>O_TRUNC</code> • <code>O_CREAT</code>

NOTE



- When creating a new file, use the `O_CREAT | O_RDWR` mode.
- `O_RDONLY`, `O_WRONLY` and `O_APPEND` are not supported.

Return Values

Success:	Positive: Handle to the index file.
Failure:	<p><code>DB_INVALID_PARAM</code>: <code>db_ptr</code> is NULL.</p> <p><code>DB_INVALID_RANGE</code>: <code>key_len</code> or <code>mode</code> is not between <code>DB_MAX_INT</code> and <code>DB_MIN_INT</code> (defined in <code>dbmgr.h</code>).</p> <p><code>DB_INVALID_REC_NUM</code>: <code>key_len</code> is negative.</p> <p><code>DB_INVALID_BUF_SIZE</code>: <code>key_len</code> is 0.</p> <p><code>DB_INVALID_FILENAME</code>: Length of the filename is greater than <code>DB_MAX_FILENAME_SIZE</code> (defined in <code>dbmgr.h</code>).</p> <p><code>DB_INVALID_FILE_MODE</code>: Invalid file mode.</p> <p>Negative: System error. <code>errno</code> contains the specific error code.</p>

NOTE

If the file name is given with an extension, the API adds .dat extension along with the file name given. For example, test.txt is created as test.txt.dat.

db_read()

Retrieves a specified record from the database file.

Prototype

```
#include <dbmgr.h>

short db_read(DB_FILE *db_ptr, void *key_struct, char *data_buf, int dlen,
              long rec_num);
```

Parameters

db_ptr	Pointer to the database file structure.
key_struct	Pointer to the key structure.
data_buf	Pointer to the data buffer.
dlen	Length of the data buffer.
rec_num	Use the following parameters: <ul style="list-style-type: none"> • n: Record number to be read. • DB_CURRENT: Reads current record; does not move file pointer. • DB_FORWARD: Reads current record and moves file pointer to the next record. • DB_BACKWARD: Reads previous record and leaves file pointer in front of the record that was read.

Note: db_read() also takes rec_num as a zero-based index.

Return Values

Positive:	Positive: Number of bytes read.
Failure:	DB_INVALID_HANDLE: File handle not greater than zero. DB_INVALID_REC_NUM: rec_num is negative or greater than the number of records in the database. DB_INVALID_PARM: db_ptr is NULL. DB_INVALID_RANGE: dlen is greater than DB_MAX_INT. DB_INVALID_BUF_SIZE: dlen is equal to 0. Negative: System error. errno contains the specific error code.

db_remove()

Deletes the database file and corresponding index files.

NOTE

Do *not* provide a filename extension. This function does not check for a filename extension. The filename should not be more than `DB_MAX_FILENAME_SIZE` (= 26) characters.

Prototype

```
#include <dbmgr.h>
short db_remove(char *filename);
```

Parameters

`filename` Name of the file to remove.

Return Values

Success: 0: File successfully removed.

Failure: `DB_INVALID_FILENAME`: Length of the filename is greater than `DB_MAX_FILENAME_SIZE` (defined in `dbmgr.h`).

Negative: System error. `errno` contains the specific error code.

db_seek_key()

Searches the index file records for the specified key value. The key value can be an element of a structure or a text string.

Prototype

```
#include <dbmgr.h>

long db_seek_key(DB_FILE *db_ptr, void *key_struct, void *Matchstr,
                long action, long *rec_num);
```

Parameters

db_ptr	Pointer to the database file structure.
key_struct	Pointer to the key structure.
Matchstr	This structure must be created by the programmer. The first element in this structure must be a pointer to a compare function written by the programmer. See db_get_key() for details.
action	Find criteria, as follows: <ul style="list-style-type: none"> • DB_FIRST: Start from the beginning of file and search forward to find the first key match. • DB_LAST: Start from the end of file and search backward to find the first key match. • DB_PREVIOUS: Find the previous key from the present position. • DB_NEXT: Find the next key from the present position. • n: nth occurrence of the key relative to the current position. If n>0, search forward.
rec_num	Address of the record number that updates when the key is found.

Return Values

Success:	Positive (>0): The key was found specified number of times. <code>rec_num</code> is updated with the last record number found.
Failure:	0: No key was found. DB_INVALID_REC_NUM: <code>rec_num</code> is negative or greater than the number of records in the database. Negative: System error. <code>errno</code> contains the specific error code.

NOTE



If the function returns a value greater than 0 (zero), then `rec_num` and `key_struct` are updated. Searching for a 5th occurrence of a key when there is only one occurrence still updates `rec_num` and `key_struct`. In this case, `db_seek_key()` returns 1 and not 5.

`db_read()` must be called to get the entire key and data structure.

The first element of the Match structure must be initialized to the compare function. The prototype of the compare function is:

```
short (*comp_func)(void *,void *)
```

where, the first argument is the key structure and the second argument is the Match structure.

Example

The linked example code file demonstrates use of `db_seek_key()`.

db_seek_record()

Searches for a specified record in an index file and modifies the file pointer position accordingly.

NOTE



`db_seek_record()` takes record number as a zero-based index.

Prototype

```
#include <dbmgr.h>
long db_seek_record(DB_FILE *db_ptr, long rec_num, int origin);
```

Parameters

- | | |
|----------------------|--|
| <code>db_ptr</code> | Pointer to the database file structure. |
| <code>rec_num</code> | Number of the record to seek, relative to the origin. |
| <code>origin</code> | Record number of the starting point. Valid values are: <ul style="list-style-type: none"> • <code>SEEK_SET</code>: Start from the beginning of the file. • <code>SEEK_CUR</code>: Start from current position of the file pointer. • <code>SEEK_END</code>: Start from end of the file towards the beginning. |

NOTE



- While searching the records backwards, the `rec_num` variable should be specified with a negative sign. If the negative value of `rec_num` is greater than the number of records or greater than the records from the `nth` position, then the `db_seek_record` fails.
- While searching the records, the starting file value, `SEEK_SET` variable cannot have a negative value.

Return Values

- | | |
|----------|---|
| Success: | Positive: Record number in the database file. |
| Failure: | <p><code>DB_INVALID_PARAM</code>: <code>db_ptr</code> is NULL.</p> <p><code>DB_INVALID_HANDLE</code>: File handle not greater than zero.</p> <p><code>DB_INVALID_REC_NUM</code>: Origin is not <code>SEEK_SET</code>, <code>SEEK_CUR</code>, or <code>SEEK_END</code>.</p> <p><code>DB_SEEK_ERROR</code>: Error in seeking the record.</p> <p>Negative: System error. <code>errno</code> contains the specific error code. See <code>errno.h</code> for error descriptions.</p> |

Example

The linked example code file demonstrates use of `db_seek_record()`.

db_write()

Writes or updates a record in both the database and index files.

Prototype

```
#include <dbmgr.h>

short db_write(DB_FILE *db_ptr, void *key_struct, char *data_buf,
               unsigned int data_len, long rec_num);
```

Parameters

db_ptr	Pointer to the database file structure.
key_struct	Pointer to the key structure.
data_buf	Pointer to the data buffer.
data_len	Data length.
rec_num	File entry index or pointer. Valid values are: <ul style="list-style-type: none">• DB_APPEND: Writes to end of file.• DB_CURRENT: Writes to current file position; does not advance file pointer.• DB_FORWARD: Writes to current file position and advances the file pointer.• DB_BACKWARD: Overwrites the previous record. File pointer is placed in front of written record.

Note: rec_num accepts integer values.

Return Values

Success:	Positive: Number of bytes written.
Failure:	DB_INVALID_PARAM: db_ptr is NULL. DB_INVALID_REC_NUM: rec_num is negative or greater than the number of records in the database. Negative: System error. errno contains the specific error code.

NOTE



If a record exists at the current file pointer position, it is overwritten.

Printing Monochrome Bitmap

This feature enables printing of monochrome bitmaps independent of the printers used in Verix eVo terminals.

print_image()

Prints the monochrome bitmap. The bitmap is printed if the following conditions are satisfied:

- valid monochrome bitmap type.

- offset should be of the range 0-60.
- width of the bitmap should be within the printable area from the offset position.

Prototype print_image(int offset, char * filename);

parameters

offset	0-60.
	Note: If the value is less than 0, then 0 is taken as the offset value. If the value is greater than 60, then 60 is taken as the offset value.
filename	Valid bmp file.

Return Values

Success	1	
Failure	PRINTER_FAIL	Unable to open the printer.
	OPEN_FAIL	Unable to open the file.
	READ_FAIL	Unable to read the file.
	NOT_MONOCHROME	Not a monochrome bitmap.
	NO_DATA	There is no value in the bitmap.
	NO_MEMORY	Dynamic memory allocation fails.
	WRITE_FAIL	Unable to write to printer.

Print Pak 3700 Function Calls

Each of the stand-alone driver functions can be called directly by the application. Each function includes a parameter that specifies the opened port handle. This handle is an identifier generated by the operating system when the port is opened. The application is responsible for both opening the opened port to obtain the handle and closing the opened port after print operations complete.

This section provides descriptions of the following Print Pak 3700 driver function calls:

- p3700_close()
- p3700_dnld_char()
- p3700_dnld_font_file()
- p3700_dnld_graphic_file()
- p3700_id()
- p3700_init()
- p3700_mode()
- p3700_print()
- p3700_print_graphic()
- p3700_select_font()
- p3700_sel_tbl_dnld_char()
- p3700_status()

NOTE



Print Pak 3700 function calls are supported in Vx680 and Vx520 terminals.

p3700_close()

Waits for all pending data to transmit to the printer, then returns.

CAUTION



p3700_close() performs a device-level close operation.

Prototype

```
#include <printer.h>
int p3700_close(short h_comm_port);
```

Parameters

h_comm_port Handle for the opened communication port.

Return Values

Success: 0: No error.
Failure: -1: Printer did not respond.

p3700_dnld_char()

Transmits a single character to the printer. The first byte in the passed buffer must be the character to replace, followed by either 14 bytes (8 x 14 font), 32 bytes (16 x 16 font), or any other font size supported by the printer. These bytes represent the bitmap for the new font character.

Prototype

```
#include <printer.h>

short p3700_dnld_char(short h_comm_port, unsigned char *rd_buf,
                     unsigned short write_bytes);
```

Parameters

h_comm_port	Handle for the opened communication port.
rd_buf	Buffer containing the bytes to download to the printer.
write_bytes	Number of characters to download (including the char to replace). For example, an 8 x 14 font requires 15 bytes.

Return Values

Success:	> 0: Number of bytes written.
	0: No error.
Failure:	-11: Download failed.

p3700_dnld_font_file()

Downloads a font file set to the printer. The font file is composed of an 8-byte header record. The definition of this record is as follows:

- 1: (N) p3700 printer ID code
- 2: ESC
- 3: l (lower-case L)
- 4: font_size
- 5: font_table
- 6 & 7: number of bytes per character (for example, for a 16 x 16 font, bytes 6 and 7 are 0x00 and 0x20).
- 8: number of characters to download

Following the header record is a series of font images that compose each new font character. The first byte in the font image is the ASCII character to replace in the selected font table. The remaining bytes in the font image are the bytes that form the new font character.

Prototype

```
#include <printer.h>

short p3700_dnld_font_file(short handle, short h_font_file,
                           short font_table);
```

Parameters

handle	Handle for the opened communication port.
h_font_file	Handle for the font file.
font_table	Font table to select.

Return Values

Success: > 0: Number of font characters downloaded.

Failure: 0: No characters downloaded (error).

-8: Header error occurred.

-9: File error occurred.

-11: Font download failed.

-22: Invalid printer.

NOTE



- Verix eVo ACT printer library checks for the cancel key operation during the download of graphic file and font file to the printer.
- If the DISABLECANKEYF config.sys entry is set to 1, then Verix eVo ACT library ignores the cancel key pressed during the font file download.
- If the DISABLECANKEYF config.sys entry is set to 0 or any other value, or not found, then Verix eVo ACT library checks for the cancel key press and aborts the font file download if the cancel key is pressed.

p3700_dnld_graphic_file()

Downloads a graphic logo file to the printer. The graphic file is composed of a 16-byte header record. The definition of this record is:

- 1: "N" (printer ID code of P3700)
- 2: ESC
- 3: "G"
- 4: "M"
- 5: Image format (ASCII)
- 6: ",",
- 7: Reserved ("0") Image ID (ASCII)
- 8: ",",
- 9, 10, 11: Image Width (ASCII) (for example, the image width is "240" for a 240 column image).
- 12: ",",
- 13,14,15: Image height (ASCII) (for example, the image height is "120" for a 120 column height image).
- 16: ",",

Prototype

```
#include <printer.h>
short p3700_dnld_graphic_file(short h_comm_port, short h_graphic_file);
```

Parameters

`h_comm_port` Handle for the opened communication port.
`h_graphic_file` Handle for the graphic file.

Return Values

Success: 0: No error.
 Failure: -8: Header error occurred.
 -9: File error occurred.
 -39: Wrong file.

NOTE



- Verix eVo ACT printer library checks for the cancel key operation during the download of graphic file and font file to the printer.
- If the `DISABLECANKEYG config.sys` entry is set to 1, then Verix eVo ACT library ignores the cancel key pressed during the graphic file download.
- If the `DISABLECANKEYG config.sys` entry is set to 0 or any other value, or not found, then Verix eVo ACT library checks for the cancel key press and aborts the graphic file download if the cancel key is pressed.

p3700_id()

Sends the <ESC>i command (request printer ID) to the ITP, and waits for a response. If the ITP responds within the time-out period, the ID is compared to the valid ID for an ITP.

Prototype

```
int p3700_id(short h_comm_port, short id_time_out);
```

Parameters

`h_comm_port` Handle to the opened communications port.

`id_time_out` Time-out period, in increments of approximately 500 ms.

Return Values

Success: 0: No errors detected.

Failure: -1: Printer did not respond.

 -2: Invalid printer ID.

 -22: Invalid printer type.

 -23: Escape sequence not found.

 -24: Printer not initialized.

p3700_init()

Sets the printer to native mode by sending the <GS><ESC>c<GS> command. If the printer responds within the time-out period, the ID is compared to the valid ID for an ITP (P).

Prototype

```
#include <printer.h>
int p3700_init(short h_comm_port, short time_out);
```

Parameters

<code>h_comm_port</code>	Handle for the opened communication port.
<code>time_out</code>	Time out, in 1-sec increments.

Return Values

Success:	0: No errors detected.
Failure:	-1: No response from the printer.
	-2: Invalid printer ID.
	-22: Invalid printer type.
	-23: Escape sequence not found.
	-24: Printer not initialized.
	-29: Printer not available.

p3700_mode()

Converts special print characters to a valid printer command sequence. This sequence is appended to the buffer passed as the second argument. Returns an updated pointer to the print buffer.

Prototype

```
#include <printer.h>
unsigned char *p3700_mode(unsigned char mode, unsigned char *buf);
```

Parameters

mode Special print character.

buf Pointer to the print buffer.

Return Values

Success: 0: No errors detected.

Failure: -1: No response from the printer.

 -2: Invalid printer ID.

 -22: Invalid printer type.

 -23: Escape sequence not found.

 -24: Printer not initialized.

 -29: Printer not available.

p3700_print()

Sends a text string to the ITP.

Prototype

```
#include <printer.h>
int p3700_print(short h_comm_port, unsigned char *buf);
```

Parameters

h_comm_port Handle to the opened communications port.

buf Pointer to the string to print (maximum 1000 bytes).

Return Values

Failure: < 0: Port write error.

 1: No send attempted; returns 1 for formatter engine; or

Success: ≥ 1: Actual number of characters sent to printer.

p3700_print_graphic()

Prints a graphic file already in printer memory.

Prototype

```
#include <printer.h>
short p3700_print_graphic(short h_comm_port, short imageId,
                          short offset);
```

Parameters

h_comm_port	Handle for the opened communication port.
imageId	The index of the image ID must be 0.
offset	The left margin for the image.

Return Values

Success: > 0: No error.
Failure: ≤ 0: Error occurred.

p3700_select_font()

Selects the font table to use for printing or downloading.

Prototype

```
#include <printer.h>
short p3700_select_font(short h_comm_port, short font_size,
                        short font_table);
```

Parameters

h_comm_port	Handle for the opened communication port.
font_size	Size of the font.
font_table	Font table to select.

Return Values

Success: > 0: No error.
Failure: < 0: Error selecting the font.

p3700_sel_tbl_dnld_char()

Selects the font table, then downloads a single font character to the printer.

Prototype

```
#include <printer.h>

short p3700_sel_tbl_dnld_char(short h_comm_port, unsigned char *font_buf,
                             short font_table, short font_size, short
                             font_bytes);
```

Parameters

h_comm_port	Handle for the opened communication port.
font_buf	Buffer containing the bytes to download to the printer.
font_table	Font table to select.
font_size	Size of the font (valid values determined by printer).
font_bytes	Number of characters to download (including the char to replace). For example, an 8 x 14 font requires 15 bytes.

Return Values

Success: 0: No error.

Failure: -11: Download failed.

p3700_status()

Sends the <ESC>d command to the ITP and waits for it to respond with its status byte. The time-out value is specified in increments of 1000 ms (1 second). Specifying a time-out value of 6 allows the printer approximately 6 seconds to respond.

Prototype

```
#include <printer.h>
int p3700_status (short h_comm_port, short stat_time_out);
```

Parameters

<code>h_comm_port</code>	Handle to the opened communications port.
<code>stat_time_out</code>	Time-out period, in increments of approximately 500 ms.

Return Values

Success:	0: No error.
Failure:	-4: No status.
	-5: Paper low.
	-10: RAM error.
	-20: Printer failure.
	-21: Paper out.
	-23: Escape sequence not found.
	-24: Printer not initialized.
	-27: Firmware corrupt.



Example Programs

This chapter provides example code for the following:

- [EXFIELD.C](#) displays the prompt, ACL FIELD FUNCS, then displays the prompt, FIELD CNT EXAMPLE
- [EXTMOUT.C](#) defines 11 time outs, ranging from 1 second to 3 minutes
- [EXTRANS.C](#) prompts for an account and waits for either a card swipe or keyboard input
- [Data Capture Engine Example Program](#) demonstrates the functionality of DCE.
- [Modem Engine Example Program](#) demonstrates the functionality of modem engine.
- [Report Formatter Example Program](#) demonstrates the functionality of report formatter.
- [P3700 Example Program](#) demonstrates the use of all P3700 printer functions.

Further description of each application is included as code comments.

EXFIELD.C

```

/*****
* Application: EXFIELD
* Release Date:7/12/99
* Version: 1.0
*
* Purpose: Demonstrates usage of Verix eVo ACT Library functions:
* fieldcnt , fieldfix , fieldray and fieldvar
* Other ACL functions used :
* prompt , get_char , view_buffer , display
* Description: EXFIELD displays the prompt ACL FIELD
* FUNCS
* It then displays the prompt FIELDCNT EXAMPLE.
* The buffer used as input to the fld_cnt() functions is
* displayed. The user may scroll through the buffer using
* the # and keys.
* Keys CLEAR or ENTER terminate the display. Next each
* field in the buffer is displayed.
* Next , the prompt FIELDFIX is displayed. The buffer
* used as input to the fld_fix() function is displayed
* for review , followed by each field in the buffer.
* EXFIELD then displays the FIELDRAY EXAMPLE prompt. The
* buffer used as input to the fld_ray() function is
* displayed for review , followed by each field in the
* buffer.
* EXAMPLE OF VARIABLE FIELDS DELIMITED BY SPACES.
* Lastly , the FIELDVAR EXAMPLE prompt is displayed. The
* buffer used as input to the fld_var() function is
* displayed for review, followed by each field in the
* buffer , EXAMPLE OF VARIBALE FIELDS DELIMITED BY SPACES
*
*****/

#include <aclconio.h>
#include <aclstr.h>
#include <string.h>
#include <svc.h>

char example1[] =
{ "EXAMPLE OF VARIBALE FIELDS DELIMITED BY SPACES" };
int ex1_fld = 7; /* 7 fields delimited by spaces */
char fld_fix[] = {"122333444455555" };
int num_fix = 5; char fld_cnt[] = {"\0021\00312\004123\0051234\00612345"
};
int num_cnt = 5;
char field[17];
char buff[15];

int main( int argc, char * argv[])
{
int console, h_clock=1;

```

```

int i,j;

console = open(DEV_CONSOLE,0);
prompt(h_clock,"ACL FIELD FUNCS",100,CLR_EOL);

/* fieldcnt() example : Starting at the beginning of the
* fld_cnt buffer(offset 0) ,
* show counted fields 1 through 5 */

prompt(h_clock,"FIELD CNT EXAMPLE",100,CLR_EOL);
view_buffer(fld_cnt,1,KM_CR | KM_ESC);

for(i=1; i<=num_cnt; ++i)
{
j = fieldcnt(fld_cnt,0,i,field);
prompt(h_clock,field,50,CLR_EOL);
get_char();
}

/* fieldfix() example :
* Show fixed fields of lengths 1 through 5 from fld_fix
* buffer.
* Offsets to start field at are based on the return value
* ( field size ) from the previous fieldfix() call */

prompt(h_clock,"FIELD FIX EXAMPLE",1000,CLR_EOL);
view_buffer(fld_fix,1,KM_CR | KM_ESC);

for(i=1 , j=0; i<= num_fix; ++i)
{
j+= fieldfix(fld_fix , j, i, field);
prompt(h_clock,field,500,CLR_EOL);
get_char();
}

/* fieldray() example :
* Show ray fields, each terminated by space, from
* example Offsets to start field at are based on the
* return value (field size) from the previous fieldfix()
* call */

prompt(h_clock,"FIELD RAY EXAMPLE",1000,CLR_EOL);
view_buffer(example1,1,KM_CR | KM_ESC);

for(j = 0; j< strlen(example1); ++j)
{
j+= fieldray(example1 , j, ' ',field);
}

```

```
prompt(h_clock,field,500,CLR_EOL);
get_char();
}

/* fieldvar() example :
 * Show variable fields 1 through 5 in example1 buffer.
 * Each field is terminated by space.
 */

prompt(h_clock,"FIELDVAR EXAMPLE",1000,CLR_EOL);
view_buffer(example1,1,KM_CR | KM_ESC);

for(i=1; i<= ex1_fld; ++i)
{
j = fieldvar(example1, i, ' ', field);
prompt(h_clock,field, 500, CLR_EOL);
get_char();
}
display("\f END EXFIELD");
return 1;
}
```


EXTMOUT.C

```

/*****
* Application: EXTMOUT
* Release Date: 07/12/99
* Version: 1.0
* Purpose: Demonstrates usage of Verix eVo ACT Library functions:
* set_itimeout and CHK_TIMEOUT
* Other ACL functions used - display and display_at
*
* Description: EXTMOUT defines 11 timeouts ranging from 1
* second to 3 minutes from current time.
* Each timeout controls what is displayed at a
* corresponding display position.
* EXTMOUT displays a "twinkling" character in each of the
* 16 display positions.
* As each timeout expires , EXTMOUT displays the
* corresponding letter from a program message in the
* display position assigned to the timeout.
*****/

#include <acldev.h>
#include <aclconio.h>
#include <svc.h>

long timeouts[16];

char tw_chars[] = { '|','/','-','\\','|','/','-','\\'};
char end_prompt[] = { "END TIMER DEMO"};
char temp[2];

int main( int argc, char * argv[])
{
    int h_clock = 1;
    int i,dom,tw_index;
    int console;
    int more;

    console = open(DEV_CONSOLE,0);
    display("\f START DEMO");
    get_char();

    /* Set 12 timeouts - timeouts range from 1 second to 3
    * minutes from current time */
    timeouts[11] = set_itimeout(h_clock,3,TM_MINUTES);
    timeouts[10] = set_itimeout(h_clock,2,TM_MINUTES);
    timeouts[9] = set_itimeout(h_clock,1,TM_MINUTES);
    timeouts[8] = set_itimeout(h_clock,50*TICKS_PER_SEC,TM_TICKS);
    timeouts[7] = set_itimeout(h_clock,45,TM_SECONDS);

```

```

    timeouts[6] = set_itimeout(h_clock,30*TICKS_PER_SEC,TM_TICKS);
    timeouts[5] = set_itimeout(h_clock,25,TM_SECONDS);
    timeouts[4] = set_itimeout(h_clock,20*TICKS_PER_SEC,TM_TICKS);
    timeouts[3] = set_itimeout(h_clock,15,TM_SECONDS);
    timeouts[2] = set_itimeout(h_clock,10*TICKS_PER_SEC,TM_TICKS);
    timeouts[1] = set_itimeout(h_clock,5*TICKS_PER_SEC,TM_TICKS);
    timeouts[0] = set_itimeout(h_clock,1*TICKS_PER_SEC,TM_TICKS);

    /* While all timeouts have not expired */

    more = 1;
    tw_index =0;

    while(more)
    {
        more = 0;
        gotoxy(1,1);

        /* Check each timeout if it has not expired , display
        * "twinkle" character otherwise display corresponding
        * letter of END message */

        for(i=0;i<16;++i)
        {
            if(CHK_TIMEOUT(h_clock,timeouts[i]))
            {
                temp[0]=tw_chars[tw_index];
                temp[1]='\0';
                display_at(1,1,temp,CLR_EOL);
                more = 1;
            }
            else
            {
                temp[0]=end_prompt[i];
                temp[1]='\0';
                display(temp);
            }
        }
        tw_index = (++tw_index)%8;
    }
}

```

EXTRANS.C

```

/*****
* Application:  EXTRANS
* Release Date: 07/12/99
* Version: 1.0
* Purpose: Demonstrates usage of Verix eVo ACT Library functions:
* append_char, range, sputf, str2long, display,
* ERR_BEEP, getkbd_entry, KBHIT, keyin_amt_range,
* NORM_BEEP prompt_at , view_buffer , chk_card_rdr ,
* card_parse , MEMCLR
* Description: EXTRANS prompts for an account and waits
* for either a card swipe or keyboard input.
* If keyboard input , EXTRANS reads the account and then
* prompts for and reads an expiration date. If card input
* ,EXTRANS reads the card and parses this card
* information. After the account is read , EXTRANS
* prompts for an amount. It then validates both the
* account the expiration date. Account is validated based
* on range data in config.sys and must be between
*1000-5999
* Expiration data is validated based on the current Opsys
* date. If an error is found, an appropriate message is
* displayed. The user may scroll through it ( using # and * keys ) and exit
the display through the Clear or Enter
* keys.
*****/

#include <aclconio.h>
#include <acldev.h>
#include <aclstr.h>
#include <aclutil.h>
#include <svc.h>
#include <svctxo.h>
#include <string.h>
#include <applidl.h>

char messages[3][16] =
{ "ACL EXAMPLE","1 = CONTINUE","CANCEL = END"};
char error_msg[] = { "INVALID KEY"};
char buff[CARD_SIZE];
char szKeyMap[MAX_ALPNUM_KEYS][CHAR_PER_KEY] = {"0-+%", "1QZ.", "2ABC&",
"3DEF%", "4GHI*", "5JKL", "6MNO~", "7PRS^", "8TUV[", "9WXY]", "*",'\":',
"#=:.$?";

struct
{
int type;
struct TRACK card_info;
char range_data[12];
char amount[15];
}input;

```

```
RANGE_PARMS r_parm;

int get_key_input(int h_clock);
int verify_exp_date(int h_clock);
void AdjustValue(unsigned char* key);

int main( int argc, char * argv[])
{
    int h_clock = 1;
    int h_card;
    int console;
    char* p_prompts[3];
    int i,errflag;
    unsigned char ch;

    window(1,1,16,1);
    h_card = open(DEV_CARD,0);
    h_clock = open(DEV_CLOCK,0);
    console = open(DEV_CONSOLE,0);
    for(i=0;i<=2;++i)
        p_prompts[i] = messages[i];

    r_parm.label = "ACCT";
    r_parm.data_fld = input.range_data;
    r_parm.start = 1;
    r_parm.file_name = "CONFIG.SYS";

    while(1)
    {
        clrscr();
        display_at(1,1,p_prompts[0],CLR_EOL);
        display_at(1,2,p_prompts[1],CLR_EOL);
        display_at(1,3,p_prompts[2],CLR_EOL);
        ch = get_char();
        AdjustValue(&ch);
        if(ch == '1')
        {
            /* Continue selection wait for either card swipe or keyboard entry*/

            r_parm.start = 1;
            errflag = 0;
            MEMCLR(&input,sizeof(input));
            clrscr();
            display_at(1,1,"ACCOUNT ?",CLR_EOL);
            while(!chk_card_rdr(h_card)&&!KBHIT());
        }
    }
}
```

```
/* If card swipe , read card data and parse */

if(chk_card_rdr(h_card))
{
    read(h_card,buff,256);
    input.type = card_parse(buff,&input.card_info,"123");
}
else
/* Else must be keyboard data so get data and store */
input.type = get_key_input(h_clock);
errflag = (input.type >=0) ? 0 : -1;
/* Get amount , valid amounts are between 1.00 and 999999.99 .
* amounts are stored with comma separator and decimal
* point radix in str_amount */

if(errflag >= 0)
{
    clrscr();
    display_at(1,1,"AMOUNT?",CLR_EOL);
    keyin_amt_range(input.amount,RDXPSEPC,99999999L,100L,2);
    /* Verify expiration date and account range */

    if(0 >= ( errflag = verify_exp_date(h_clock)))
    {
        r_parm.acct_num = input.card_info.acct;
        errflag = ( range(&r_parm) <= 0) ?-3:1;
    }
}

/* If invalid account/date , display appropriate error message */

if( errflag < 0)
{
    switch(errflag)
    {
        case(-2) :
            display("\f DATE");
            break;
        case(-3) :
            display("\f ACCT");
            break;
        default :
            display("\f INPUT");
            break;
    }
}
```

```

ERR_BEEP();
prompt_at(h_clock,7,1," ERROR",1000,NO_CLEAR);
}

/* If there is some input , then format data for display
 * and let user review it. Review is terminated with
 * either the Enter or Cancel key */

if(errflag != -1)
{
MEMCLR(buff,sizeof(buff));
if(input.type == 0)
strcpy(buff,"KEY INPUT: ");
else
sprintf(buff,"TRACK %d INPUT:",input.type);
strcat(buff," ACCT=");
strcat(buff,input.card_info.acct);
strcat(buff," EXP DATE=");

strncat(buff,input.card_info.exp,7);
strcat(buff," RANGE= ");
strncat(buff,input.range_data,7);
strcat(buff," AMOUNT=");
append_char(buff,'$');
strcat(buff,input.amount);

NORM_BEEP();
view_buffer(buff,1,KM_CR | KM_ESC);
}
else
if(ch == KEY_CANCEL)
break;
}
display("\f END EXAMPLE");
return 1;
}

/* get account and prompt for and read a yymm expiration date */

int get_key_input(int h_clock)
{
int ret_val;

/* Get account . Wait forever for input. Valid values are
 * numeric with at least 1 character and no more than 16
 * characters. */

```

```

ret_val =
getkbd_entry(h_clock, NULL, input.card_info.acct, 0, NUMERIC, szKeyMap, sizeof(
szKeyMap), 16, 1);
if( 0 < ret_val)
{
/* Get expiration date - must be 4 numeric characters. */

ret_val = getkbd_entry(h_clock, "EXP
DATE(YYMM)?", input.card_info.exp, 0, NUMERIC, szKeyMap, sizeof(szKeyMap), 4, 4)
;
}
return((ret_val>0)?0 : ret_val);
}

/* Simple expiration date validation. Expiration date must be valid and of
form yymm. */

int verify_exp_date(int h_clock)
{
int retval = 1;
char cur_date[16], card_date[7];

if(read(h_clock, cur_date, 15) != 15)
retval = -1;

memcpy(card_date+2, input.card_info.exp, 4);
card_date[6] = '\0';
cur_date[6] = '\0';

if(card_date[2] > '6')
memcpy(card_date, "19", 2);
else
memcpy(card_date, "20", 2);
retval = (str2long(card_date) <= str2long(cur_date)) ? 1: -2;
return(retval);
}

void AdjustValue(unsigned char* key)
{
if ( *key > 0xf9)
*key -= 0xf9; // screen keys: fa-fd -> 1-4
else if (*key < 0xef)
*key &= 0x7f; // keypad keys: 0-ef -> 0-7f
}

```

Data Capture Engine Example Program

```

/* SAMPLE PROGRAM TO DEMONSTRATE THE FUNCTIONALITY OF DCE
1. DCE_PUTCFG_C()
2. DCE_GETCFG_C()
3. DCE_PUTCFG_S()
4. DCE_GETCFG_S()
5. dce_key_cvt()
*/

#include <acdce.h>
#include <stdio.h>
main ()
{
    int console_handle;
    int i, rtn_val;
    unsigned int ui;
    DCE_KEY_DATA set_val;
    DCE_KEY_DATA get_val;
    unsigned char str[3], str1[3];
    unsigned char c, c1;

    console_handle = open("/dev/console",0);
    printf("\fdCE DEMO-CONFIG.SYS\n");
    get_char();

    set_val.c_val = &c;
    get_val.c_val = &c1;

    c = 'A';
    // Add CONFIG.SYS key DCE-1 with value 'A'
    rtn_val = DCE_PUTCFG_C("DCE-1", set_val);
    if (rtn_val != 1)
        printf ("WRITE - FAIL %d\n", rtn_val);
    else
        printf ("DCE-1 = %c (CHAR)\n", *set_val.c_val);
    get_char();

    // Read CONFIG.SYS key DCE-1 into char variable c1
    if ((rtn_val = DCE_GETCFG_C("DCE-1", get_val)) != 1)
        printf ("READ - FAIL %d\n", rtn_val);
    else
        printf ("DCE-1 = %c (AGAIN)\n", *get_val.c_val);
    get_char();

    set_val.s_val = str;
    get_val.s_val = str1;

```



```
// Write CONFIG.SYS key DCE-2 with string (value=CC)
memset (str, '\0', sizeof(str));
memset (str, 'C', sizeof(str)-1);
get_val.s_val = str1;
if ((rtn_val = DCE_PUTCFG_S("DCE-2",set_val)) != 1)
    printf ("WRITE - FAIL %d\n", rtn_val);
else
    printf ("DCE-2 = %s (STR)\n", set_val.s_val);
get_char();

if ((rtn_val = DCE_GETCFG_S("DCE-2",get_val)) != 1)
    printf ("READ - FAIL %d\n", rtn_val);
else
    printf ("DCE-2 = %s (AGAIN)\n", get_val.s_val);
get_char();

// Demonstrate the use of the dce_key_cvt function
// Write CONFIG.SYS key DCE-3 with string (value=DD)
memset (str, '\0', sizeof(str) );
memset (str, 'D', sizeof(str)-1);
if ((rtn_val = dce_key_cvt(DCE_WRITE, "config.sys", "DCE-3", DCE_S,
set_val)) != 1)
    printf ("READ - FAIL %d\n", rtn_val);
else
    printf ("DCE-3 = %s (STR)\n", set_val.s_val);
get_char();

if ((rtn_val = dce_key_cvt(DCE_READ, "config.sys", "DCE-3",DCE_S,
get_val)) != 1)
    printf ("READ - FAIL %d\n", rtn_val);
else
    printf ("DCE-3 = %s (AGAIN)\n", get_val.s_val);
get_char();
// If you check CONFIG.SYS, DCE- 1 = A,DCE - 2= CC
printf("END OF DCE DEMO");
close(console_handle);
}
```

Modem Engine Example Program

/* To use the test program: You require 2 terminals

1. Uncomment the call MdmCalTest() and comment the call MdmRecTest() in main()
2. Change the phone number to the appropriate number to which the host terminal will be connected and build the workspace
3. Download the output file (.out) to one terminal. This acts as the caller
4. Uncomment the call MdmRecTest() and comment the call MdmCalTest() in main() and build the workspace
5. Download the output file (.out) to the second terminal. This acts as the host. */

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <aclconio.h>
#include <svc.h>
#include <xmodem.h>

void MdmCalTest();
void MdmRecTest();

unsigned int main(void)
{
    int conHandle;
    conHandle = open("/dev/console", 0);

    MdmCalTest(); // Caller Module
    //MdmRecTest(); // receiver Module
    return 0;
}

void msg_wait( char *sz_msg )
{
    write_at( sz_msg, strlen(sz_msg), 1,2);
    SVC_WAIT(1000);
}

int get_zp_phonenum( char *sz_phonebuff )
{
    char szptr[50];
    int retVal;
    retVal = (short)get_env("*ZP", (char *)szptr, 5);
    if(szptr != NULL)
    {
        strncpy( sz_phonebuff, szptr, 5 );
    }
}
```

```

else
{
    msg_wait( "\\f*ZP IS BAD. FIX." );
    strcpy(sz_phonebuff, "\\0");
}
szptr[5] = '\\0';
sz_phonebuff[5] = '\\0';
if(szptr != NULL)
return (0);
return (-1);
}

void MdmCalTest()
{
    int retVal;
    int handle = -1;
    char dlStr[20];
    char dataTx[10];
    char dataRx[10];
    int iwrite, i;
    char test[20];
    write_at("Modem TEST", 10, 1,1);
    retVal = xmdm_open(&handle, NULL, -1,6,Rt_1200,Fmt_A7E1);
    sprintf(test, "opening, %d", retVal);
    write_at(test, 11, 1,2);
    retVal = xmdm_flush(handle);
    sprintf(test, "Clearing, %d", retVal);
    write_at(test, 11, 1,3);

    if ((retVal = get_zp_phonenum(dlStr)) != 0)
    return;
    retVal = xmdm_get_line_dial( handle, dlStr, &iwrite, -1, 15);
    sprintf(test, "Dialed, %d", retVal);
    write_at(test, 10, 1,4);

    if((retVal == CONNECT_300) || (retVal == CONNECT_1200)
    || (retVal == CONNECT_2400))
    {
        // send and recieve data
        write_at("connect scs", 11, 1,5);
        write_at("Recd :", 6,1,8);
        write_at("Sent :", 6, 1,7);
        SVC_WAIT(100);
        for(i = 0; i<10; i++)
        {
            memset(dataTx, 0, sizeof(dataTx));

```

```

        sprintf(dataTx, "PING %d", i);
        retVal = xmdm_send_data( handle, dataTx, strlen(dataTx),10);
        write_at(dataTx, 6, 8,7);
        SVC_WAIT(400);
        memset(dataRx, 0, sizeof(dataRx));
        do{
            retVal = xmdm_receive_data(handle, dataRx, 0, 6,10);
        } while(retVal != 6);
        write_at(dataRx, 6, 8,8);
        SVC_WAIT(400);
    }

}

SVC_WAIT(1000);
//disconnect
retVal = xmdm_hangup( handle, -1, 0);
sprintf(test, "disconnect, %d", retVal);
write_at(test, 13, 1,5);

retVal = xmdm_close(handle,0,0);
sprintf(test, "closing, %d", retVal);
write_at(test, 11, 1,6);

}

void MdmRecTest()
{
    int retVal;
    int handle = -1;
    char dlStr[20];
    char dataTx[10];
    char dataRx[10];
    int i;
    char test[20];

    write_at("Modem TEST", 10, 1,1);

    retVal = xmdm_open( &handle, NULL, -1,6,Rt_1200, Fmt_A7E1);
    sprintf(test, "opening, %d", retVal);
    write_at(test, 11, 1,2);
    SVC_WAIT(1000);
    strcpy(dlStr, "S0=1"); // wait for one dial ring only
    retVal = xhayes_send_cmd( handle, dlStr);

    retVal = xmdm_flush(handle);
    sprintf(test, "Clearing, %d", retVal);

```

```

write_at(test, 11, 1,4);

write_at("WAITING 1200 CALL", 17, 1,4);

do {
    retVal = xhayes_response(handle, -1, 0);
    sprintf(test,"Resp %d",retVal);
    write_at(test,(short)strlen(test),1,5);
} while (retVal != CONNECT_1200);

write_at("CONNECTED", 9, 1,6);

write_at("Recd :", 6, 1,8);
write_at("Sent :", 6, 1,7);

for(i = 0; i<10; i++)
{
    memset(dataRx, 0, sizeof(dataRx));
    do{
        retVal = xmdm_receive_data(handle, dataRx, 0, 6,10);
    } while (retVal != 6);
    write_at(dataRx, 6, 8,8);
    SVC_WAIT(400);
    memset(dataTx, 0, sizeof(dataTx));
    sprintf(dataTx, "PONG %d", i);
    retVal = xmdm_send_data( handle, dataTx, strlen(dataTx),10);
    write_at(dataTx, 6, 8,7);
    SVC_WAIT(400);
}
SVC_WAIT(1000);
while(xmdm_output_pending(handle) == X_OUTPUT_PEND);
//disconnect
retVal = xmdm_hangup( handle, -1, 0);
sprintf(test, "disconnect, %d", retVal);
write_at(test, 13, 1,5);

retVal = xmdm_close(handle,0,0);
sprintf(test, "closing, %d", retVal);
write_at(test, 10, 1,6);
}

```

Report Formatter Example Program

```
#include<stdio.h>
#include<svc.h>
#include<string.h>
#include<printer.h>
#include<formater.h>

char term_id[11];
char date[9];
char v_time[6];
char bonus;
long int_ttl;
long cum_ttl;
long int_bil;
long cum_bil;
long int_crd;
long cum_crd;
char name[5];
int test;

#ifndef G_C
#define G_C 1
#define G_S 2
#define G_I 3
#define G_L 4
typedef struct
{
void *addr;
int type;
}g_data;
#endif

g_data gvardata[] = {
/* 0 */ { (void *)term_id, G_S },
/* 1 */ { (void *)date, G_S },
/* 2 */ { (void *)v_time, G_S },
/* 3 */ { (void *)&bonus, G_C },
/* 4 */ { (void *)&int_ttl, G_L },
/* 5 */ { (void *)&cum_ttl, G_L },
/* 6 */ { (void *)&int_bil, G_L },
/* 7 */ { (void *)&cum_bil, G_L },
/* 8 */ { (void *)&int_crd, G_L },
/* 9 */ { (void *)&cum_crd, G_L },
/* 10 */ { (void *)name, G_S },
/* 11 */ { (void *)&test, G_I }
};
```

```

FORMATER p3300;

int p3300_direct (FORMATER* formater, int init_time_out )
{
    formater->output_close = p3300_close;
    formater->output_print = p3300_print;
    formater->output_mode = p3300_mode;
    formater->direct = 1;
    formater->max_buffers = 2;
    return p3300_init(formater->h_comm_port, init_time_out);
}

void main(void)
{
    int i, errline, start, h_comm_port;
    unsigned long condition;
    int handle_logo;
    char tempBuff[30];
    open_block_t parm;

    open("/dev/console", 0);
    // Initialize the Global variables
    memset(term_id, '\0', 11);
    strcat(term_id, "1234567890");
    memset(date, '\0', 9);
    strcat(date, "09/06/91");
    memset(v_time, '\0', 6);
    strcat(v_time, "10:45");
    bonus = 'K';
    int_ttl = 23456;
    cum_ttl = 2123450;
    int_bil = 11111;
    cum_bil = 77777;
    int_crd = 8888;
    cum_crd = 12377;
    memset(name, '\0', 5);
    strcpy(name, "wind");
    test = 12345;

    printf("Formater testing\n\n");
    h_comm_port = open("/dev/com4", 0);
    memset(&parm, 0, sizeof(parm));
    parm.rate = Rt_19200;
    parm.format = Fmt_A8N1 | Fmt_auto;
    parm.protocol = P_char_mode;

```

```
parm.parameter = 0;
set_opn_blk( h_comm_port, &parm);
SVC_WAIT(1000);
printf("Downloading Logo File\n");
p3300_init(h_comm_port, 6);
SVC_WAIT(200);
handle_logo = open("act.lgo", O_RDONLY);
// open a valid logo file
P3300_dnld_graphic_file (h_comm_port, handle_logo);

printf("Initializing\n");
formater_main(gvardata) ;
formater_open(h_comm_port, &p3300, "test3300.frm", p3300_direct, 12);

condition = 0L;
start = 1;
printf("Printing...\n");
while( formater_line(&p3300, start, TO_END, 1, condition, &errline) < 0 )
{
printf(tempBuff,"Printer err: %d",i);
start = errline;
}
printf("Closing Printer\n");
formater_close(&p3300);
printf("\nFormatter End\n");
}
```


P3700 Example Program

This example program demonstrates the use of all P3700 printer functions. Return values from the called functions are not generally processed as this is for demonstration purposes only. VeriFone strongly recommends that you process the return values in your application using this library.

```
#include <svc.h>
#include <stdio.h>
#include <string.h>
#include <printer.h>

void main(void)
{
    int handle; // file handle for the Printer
    int conHandle; // file handle for the display
    open_block_t_parm;
    // structure to fill comm parameters for com port
    int h_font_file; // handle to the font file
    int retVal;
    // capture the return value from called functions
    unsigned char ucCharValue;
    unsigned short usOffset;
    unsigned char printBuf[256 + 1];
    unsigned char s_buf[10];

    // Open the Display Terminal
    conHandle = open("/dev/console", 0);

    // PRINTING on the terminal
    write_at("P3700 Print Sample",18,1,1);
    handle = open("/dev/com4", 0);

    //Set the Comm Parameters
    memset(&parm,0,sizeof(parm));
    parm.rate = Rt_19200;
    // ITP is always set to 19200 baud
    parm.format = Fmt_A8N1 | Fmt_auto | Fmt_RTS;
    // ITP is always set at 8N1
    parm.protocol = P_char_mode;
    parm.parameter = 0;
    set_opn_blk(handle, &parm);
    SVC_WAIT(200);
    p3700_init(handle, 6);
    SVC_WAIT(100);

    // Fill the buffer with the range for printable
    // characters(0x20 to 0x7f)
```

```

ucCharValue = 0x20;
for(ucCharValue=0x20; ucCharValue<= 0x7f; ucCharValue++)
printBuf[ucCharValue - 0x20]= ucCharValue;
printBuf[ucCharValue++ - 0x20]='\n';
printBuf[ucCharValue++ - 0x20]='\0';

// Starting with the default font which 8x14 at 42
// column mode
retVal = p3700_print(handle,(unsigned char *)" \n\n8x14, 42 columns\n\n");
retVal = p3700_print(handle, printBuf);

// Set the printer default font to 8x14 at 32 column
// mode
retVal = p3700_select_font(handle, 0x03, 0);
retVal = p3700_print(handle, (unsigned char *)" \n\n8x14, 32
columns\n\n");
retVal = p3700_print(handle, printBuf);

// Set the printer default font to 5x8 at 24 column
// mode 5x8 fonts are by default printed in double
// width and height
retVal = p3700_select_font(handle, 0x02, 0);
retVal = p3700_print(handle, (unsigned char *)" \n\n5x8, 24 columns\n\n");
retVal = p3700_print(handle, printBuf);

// Set the printer default font back to 8x14 at 42
// column mode
retVal = p3700_select_font(handle, 0x04, 0);
retVal = p3700_print(handle, (unsigned char *)" \n\nBack to 8x14, 42
columns\n\n");
retVal = p3700_print(handle, printBuf);

// printer in NORMAL font
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = PRINT_NORM;
strcpy((char *)&printBuf[1], (const char *)" \n\nNORMAL PRINT\n\n");
retVal = p3700_print(handle, printBuf);
// Print in Double Height
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = DBL_HEIGHT;
strcpy((char *)&printBuf[1], (const char *)"DOUBLE HEIGHT\n\n");
retVal = p3700_print(handle, printBuf);

// printer in NORMAL font
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = PRINT_NORM;
strcpy((char *)&printBuf[1], (const char *)"NORMAL PRINT\n\n");

```

```

retVal = p3700_print(handle, printBuf);

// Print in Double Width
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = DBL_WIDTH;
strcpy((char *)&printBuf[1], (const char *)"DOUBLE WIDTH\n\n");
retVal = p3700_print(handle, printBuf);

// Print in Double Height and Double Width
// Please note that Printing Attributes can be clubbed
// meaningfully
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = PRINT_NORM;
printBuf[1] = DBL_HEIGHT;
printBuf[2] = DBL_WIDTH;
strcpy((char *)&printBuf[3], (const char *)"DOUBLE HEIGHT & WIDTH\n\n");
retVal = p3700_print(handle, printBuf);

// Print in Inverse Style
memset(printBuf, 0, sizeof(printBuf));
printBuf[0] = PRINT_NORM;
printBuf[1] = INVERSE;
strcpy((char *)&printBuf[2], (const char *)"INVERSE PRINTING\n\n");
retVal = p3700_print(handle, printBuf);

// Graphics Download & print
strcpy((char *)printBuf, (const char *)"Downloading Logo File");
retVal = strlen((const char *)printBuf);
write_at((char *)printBuf, retVal, 1, 1);
h_font_file = open("vmaclogo.lgo", O_RDONLY);
// open a valid logo file
retVal = p3700_dnld_graphic_file (handle, h_font_file);

strcpy((char *)printBuf, (const char *)"Logo at Offset 0\n");
retVal = p3700_print(handle, printBuf);
usOffset = 0; // Set the offset at 0 dots from the Left
retVal = p3700_print_graphic(handle, 0, usOffset);
strcpy((char *)printBuf, (const char *)"Logo at Offset 100\n");
retVal = p3700_print(handle, printBuf);
usOffset = 100;
// Set the offset at 100 dots from the Left
retVal = p3700_print_graphic(handle, 0, usOffset);

// Down load font file 16 X 16
strcpy((char *)printBuf, (const char *)"Downloading 16x16 Font");
retVal = strlen((const char *)printBuf);

```

```

write_at((char *)printBuf,retVal,1,1);
// 16x16 printer font file containing 128 chars from
// offset 0 to 127
h_font_file = open("16x16.pft", O_RDONLY);
// download the printer font file at font table 1
retVal = p3700_dnld_font_file (handle, h_font_file, 1);
strcpy((char *)printBuf,(const char *)"Printing 16x16 Font\n\n");
retVal = p3700_print(handle, printBuf);
retVal = p3700_select_font(handle, 0x01, 1);
// 0x01 corresponds to 16x16 font size

// Hexa Dump of 32 Chars downloaded from offset 0 to
// 31
strcpy((char *)printBuf,(const char *)"Hexa Dump Printing\n\n");
retVal = p3700_print(handle, printBuf);
memset(printBuf,0,sizeof(printBuf));
printBuf[0] = SEL_HEX; // set to Hexa Dump Mode
// In Hex Dump Mode each character is specified by a
// two byte offset
// For example Character at offset 8 is specified as
// 08 and offset 15 as 0e
// Hexa Dump output buffer should terminate with a
// semicolon character
// Fill the buffer with the range for non printable
// characters(0x00 to 0x1f)
strcpy((char *)&printBuf[1],(const char *)
"000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f;\n\n");
retVal = p3700_print(handle, printBuf);
strcpy((char *)printBuf,(const char *)"Printing ASCII Chars\n\n");
retVal = p3700_print(handle, printBuf);
// Hexa Dump not required for characters in printable
// range
// Fill the buffer with the range for printable
// characters(0x20 to 0x7f)
ucCharValue = 0x20;
for(ucCharValue=0x20; ucCharValue<= 0x7f; ucCharValue++)
printBuf[ucCharValue - 0x20]= ucCharValue;
printBuf[ucCharValue++ - 0x20]='\n';
printBuf[ucCharValue++ - 0x20]='\0';
retVal = p3700_print(handle, printBuf);

// Create the buffer for a single character download
// of size 5X8
s_buf[0] = 'A';
// To be Downloaded at offset 0x41 ('A') in 5 X 8 size
s_buf[1] = 31;// 00011111
s_buf[2] = 17;// 00010001
s_buf[3] = 17;// 00010001

```

```

s_buf[4] = 17;// 00010001
s_buf[5] = 31;// 00011111
s_buf[6] = 17;// 00010001
s_buf[7] = 17;// 00010001
s_buf[8] = 17;// 00010001

// Select the font table 10 for a 5x8 character to
// store at offset 0x41
retVal = p3700_sel_tbl_dnld_char(handle,s_buf,10,0x02,9);
if (retVal < 0)
write_at("DNLD FAIL",9,1,1);
else
retVal = p3700_print(handle, (unsigned char *)"A\n");
s_buf[0]='Z';
// Copy the pattern A in offset for Z also
retVal = p3700_dnld_char(handle, s_buf, 9);
if (retVal < 0)
write_at("DNLD FAIL",9,1,1);
else
retVal = p3700_print(handle, (unsigned char *)"AZ\n");
// this will print AA

// switch to font table 1 for 16x16 characters
retVal = p3700_select_font(handle, 0x01, 1);
retVal = p3700_print(handle, (unsigned char *)"AZ\n");
// this will print AZ

// switch to font table 10 for 5X8 characters
retVal = p3700_select_font(handle, 0x02, 10);
retVal = p3700_print(handle, (unsigned char *)"AZ\n");
// this will print AA

SVC_WAIT(1000);
retVal = p3700_close(handle);
write_at((char *)"Printer Demo Complete",21,1,1);

}

```



VeriFone, Inc.
2099 Gateway Place, Suite 600
San Jose, CA, 95110 USA
(800) VeriFone (837-4366)
www.verifone.com

Verix eVo ACT

Programmers Guide

