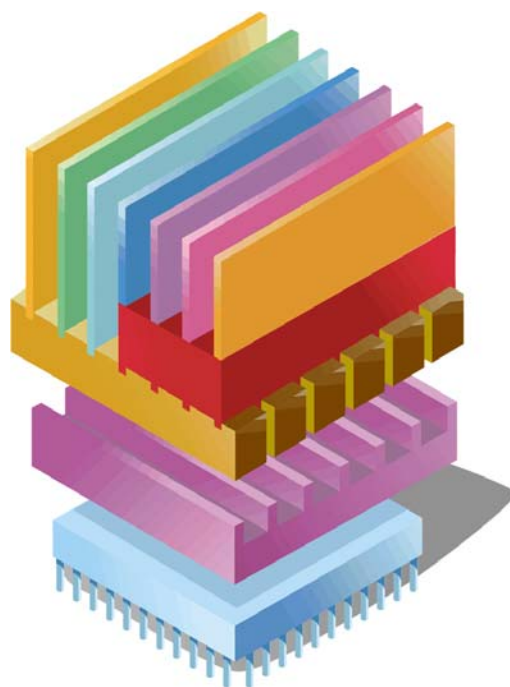


Verix® V Operating System

Programming Tools Reference Manual

For V^x Solutions



VERIX® V Operating System Programming Tools Reference Manual
© 2009 VeriFone, Inc.

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

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

VeriFone, the VeriFone logo, Omni, VeriCentre, Verix, and ZonTalk are registered trademarks of VeriFone. Other brand names or trademarks associated with VeriFone's 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

www.verifone.com

VeriFone Part Number 23231, Revision J



CONTENTS

PREFACE	7
Assumptions About the Reader	7
For More Information	7
Conventions and Acronyms	7
Conventions	7
Acronyms	9
 CHAPTER 1	
Your First Verix Application	
What's In the SDK?	12
How to Build Verix ARM Applications	14
Installation	14
Example Program	14
Looking Ahead	16
Programming Documentation	16
 CHAPTER 2	
C Programming Concepts for Verix Terminals	
C Data Types	17
Program Arguments	17
Stack and Heap	18
Version Number	18
C Library	19
<assert.h>—Diagnostics	19
<locale.h>—Localization	20
struct lconv *localeconv()	21
Char *setlocale()	22
<ctype.h>—Character Handling	23
<math.h>—Mathematics	23
<setjmp.h>—Non-local Jumps	24
<signal.h>—Signal Handling	24
<stdarg.h>—Variable Arguments	24
<stdio.h>—Input/Output	24
.....	26
FILE *tmpfile()	27
<stdlib.h>—General Utilities	28
<string.h>—String Handling	30
<time.h>—Date and Time	30
TZRULE.TXT File	34
Time Zone and DST Application Support	35
getTZdata()	36
getTZdata()	49
read_RTC ()	51

CHAPTER 3		
Compiling and Linking		
	File Naming Conventions	53
	ARM Tools Installation	53
	Environment Variables	54
	Locating the Tools	54
	Creating Libraries	55
	Options	55
	General Options	55
	Compile Options	56
	Assembler Options	57
	Link Options	57
	Header Options.	58
	Library Options	58
CHAPTER 4		
DDL – Direct Download Utility		
	Direct Downloads with DDL	59
	Invoke DDL	59
	Communications Options	60
	File Download Options	60
	Removing Files	60
	Miscellaneous Options	61
	Configuration Settings.	61
	Environment Variable	61
	File Authentication and DDL	62
CHAPTER 5		
Creating and Using Libraries		
	Differences Between Verix ARM and Verix 68K Shared Libraries	66
	Linked vs. Shared Libraries	66
	Absolute vs. Position-Independent	67
	ARM vs. Thumb	67
	Archive Libraries.	68
	Names.	68
	Creating Archive Libraries.	68
	Listing Contents	68
	Shared Libraries	68
	Programming Considerations	68
	Creating a Shared Library.	74
	Calling Shared Libraries	77
	Updating Shared Libraries	78
	Debugging Shared Libraries	78
	Managing Shared Libraries.	79
CHAPTER 6		
Miscellaneous Tools		
	VRXHDR Utility.	81
	VRXHDR Examples	82
	VLR Utility.	82
	VLR Error Messages.	83
CHAPTER 7		
Debugging		
	Install the RealView Debugger	86
	Build a Debuggable Program	86
	VRXDB	86

	Running VRXDB	87
	Properties	88
	Commands	88
	Miscellaneous Commands	93
	Debugging Procedures	94
	Automatic Startup	94
	Manual Startup	95
	Ending a Debug Session	96
	Debugging in Different Groups	96
	Debugging an Existing Task	96
	Using VRXDB Without the Debugger	96
	Multitask Debugging	96
	*DBMON Abort Codes	99
	Tips, Pitfalls, and Annoyances	99
	Tools	101
	fromelf	101
CHAPTER 8		
Understanding the Verix Error Log		
	Crash Kit Tools	103
	findsrc	103
	findlib	104
	appmap	104
	fromelf	106
	OS Debug Trace	106
	Debugging With the Verix ARM Error Log	108
	Reading the Crash Log	108
	Example Error Screen	108
	Verix ARM Internal Components	109
	Crash Analysis	111
CHAPTER 9		
Porting Verix 68K Applications to Verix ARM		
	C Programming Issues	115
	Compiler Errors and Warnings	115
	Data Representation	116
	#include Paths	118
	Determining the Platform (#ifdefs)	119
	Verix Interface Issues	119
	Const-Correct Prototypes	119
	Narrow Arguments	120
	Unimplemented and Deprecated Functions	120
	Changed Functions	121
	Additional Resources	122
CHAPTER 10		
Installing the Verix USB Client RS232 Driver		
	Procedure	123
	INDEX	127



This manual supports the Verix V Development Toolkit (VVDTK), which assists with the development of applications for the Verix V transaction terminal. This manual:

- Introduces the VVDTK;
- Describes the programming environment; and,
- Provides instructions for using the development tools supplied in the VVDTK, such as the compiler and debugger.

NOTE

Although this manual contains some operating instructions, please refer to the reference manual of your terminal for complete operating instructions. Refer to the *Verix V Operating System Programmers Manual*, VPN - 23230 for further discussion of the Verix V OS and descriptions of the function calls.

Assumptions About the Reader

To use this manual you should have an understanding of the following:

- C programming concepts and terminology.
- Windows 98, Windows 2000, or Windows NT operating systems.

For More Information

Detailed operating information can be found in the reference manual of your terminal. For equipment connection information, refer to the reference manual or installation guides.

NOTE

Refer to the Verix V Operating System Programmers Manual for further discussion of the Verix V OS (operating system) and descriptions of the function calls.

Conventions and Acronyms

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

Conventions

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

- The `courier` typeface is used for code entries, filenames and extensions, and anything that requires typing at the DOS prompt or from the terminal keypad.
- Terminal displays are shown all uppercase in the `ARIAL` typeface.

- Text references in [blue](#) are links in online documentation. Click on the text to jump to the topic.

NOTE



Notes point out interesting and useful information.

CAUTION



Cautions point out potential programming problems.

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

Table 1 **Conventions**

Abbreviation	Definition
A	ampere
b	binary
bps	bits per second
dB	decibel
dBm	decibel meter
h	hexadecimal
Hz	hertz
KB	kilobytes
kbps	kilobits per second
kHz	kilohertz
LSB	least-significant bit
mA	milliampere
MAX	maximum (value)
MB	megabytes
MHz	megahertz
MIN	minimum (value)
ms	milliseconds
MSB	most-significant bit
pps	pulse per second
s	seconds
TYP	typical (value)
V	volts

Acronyms

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

Table 2 **Acronyms**

Acronym	Definition
ANSI	American National Standards Institute
APDU	Application Protocol Data Units
API	Application Program Interface
ASCII	American Standard Code For Information Interchange
APACS	Association For Payment Clearing Services: Standards Setting Committee; A Member Of The European Committee For Banking Standards (Ecbs)
ATR	Answer To Reset
BCD	Binary Coded Decimal
BRK	Break
BWT	Block Waiting Time
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
CVLR	Compressed Variable-length Record
CWT	Character Waiting Time
DDL	Direct Download Utility
DLL	Dynamic Link Library
DTK	Development Toolkit. See <i>Vvdtk</i> .
DTR	Data Terminal Ready
EOF	End-of-file
EPP	External Pin Pad
FIFO	First In, First Out
FIQ	Fast Interrupt Request
ICC	Integrated Circuit Card; Smart Card
IDE	Integrated Development Environment
IEEE	Institute Of Electrical And Electronics Engineers
IFD	Smart Card Interface Device
IFSC	Information Field Size Card
IFSD	Information Field Size Reader
IPP	Internal Pin Pad
IRQ	Interrupt Request
LAN	Local Area Network
LCD	Liquid Crystal Display
LRC	Longitudinal Redundancy Check
MAC	Message Authentication Code
MIB	Management Information Block
MMU	Memory Management Unit
MUX	Multiplexor
NMI	Nonmaskable Interrupt

Table 2 **Acronyms** (continued)

Acronym	Definition
OS	Operating System
PIN	Personal Identification Number
PTS	Protocol Type Selection
RAM	Random Access Memory
ROM	Read-only Memory
RTS	Request To Send
Rx	Receive
SCC	Serial Communication Control
SCC buffer	Storage Connecting Circuit Buffer
SDLC	Synchronous Data Link Control
SMS	Small Message Service
SRAM	Static Random-access Memory
TCB	Task Control Block
Tx	Transmit
UART	Universal Asynchronous Receiver Transmitter
VVDTK	Verix V Development Toolkit
VLR	Variable-length Record
WTX	Workstation Technology Extended
WWAN	Wireless Wide Area Network

Your First Verix Application

The succeeding sections describes how to create, compile, download, and run a very simple Verix V application. This practice should help you with future application construction.

Refer to the Getting Started with VVDTK, VPN 22474 included in the VVDTK CD for software installation and setup information.



These examples are written to allow you to progress through them rapidly. Descriptions of utilities used during the procedures and details of terminal operation are therefore omitted.

Figure 1 shows the general features of a Verix terminal.

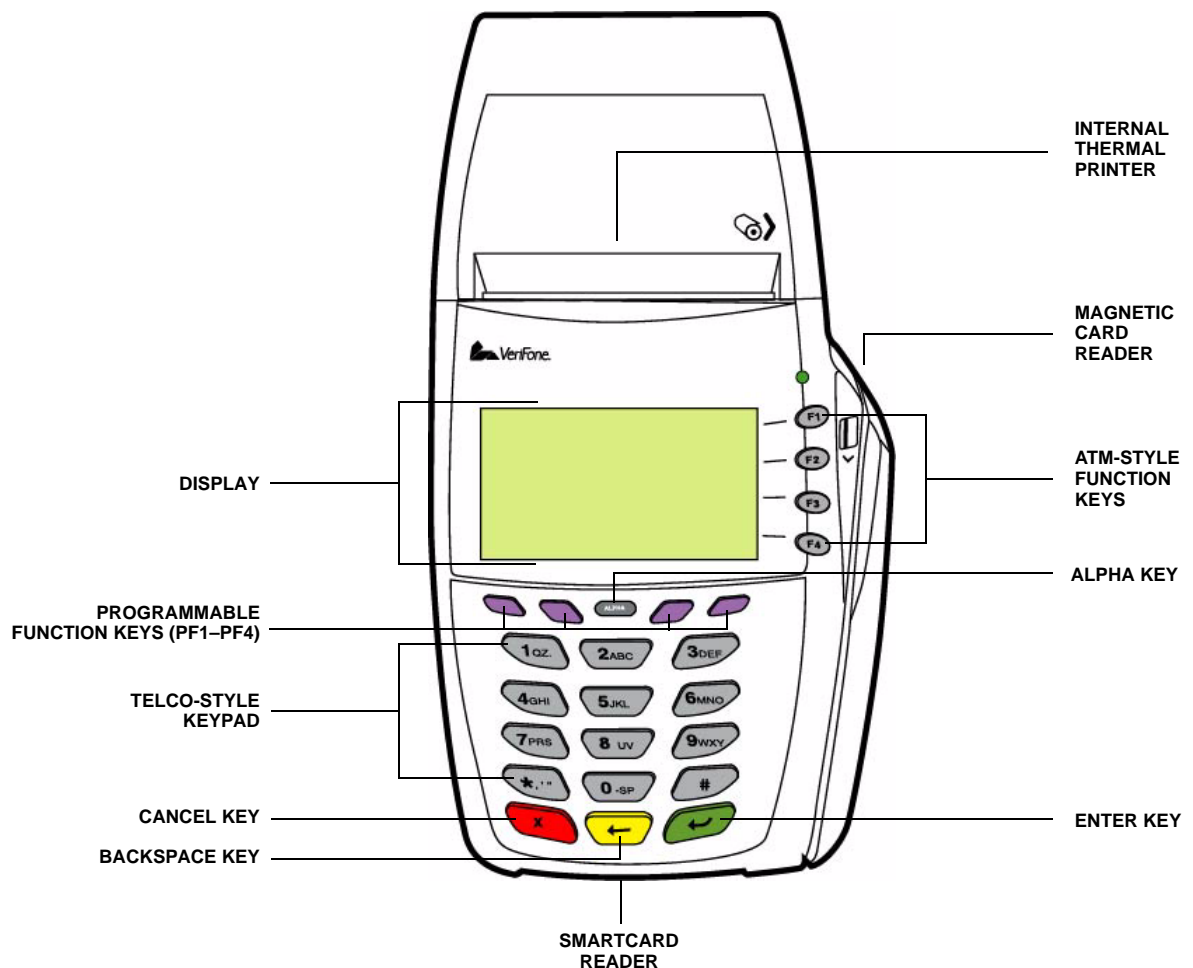


Figure 1 Verix Terminal: Front View

What's In the SDK?

The directory structure of VVSDK contains the directories and files listed in Table 3.

Table 3 Verix SDK Directory Structure \bin

Executable	Description
bin\	
vrxcc.exe	Verix build tool. See Compiling and Linking .
vrxhr.exe	Verix tool used to set stack and heap sizes.
vrplib.exe	Shared library generator. Creating and using shared libraries is described in Creating a Shared Library .
ddl.exe	Download tool. See DDL – Direct Download Utility .
vlr.exe	Variable-length record conversion utility.
dbmon.out	Debug monitor. Debugging with the ARM RealView debugger is described in Debugging .
vrxdb.exe	Verix debug controller. Helper program needed to use the RealView debugger.
vssconv.exe	Security script tool. See VeriShield Security Scripts and documentation installed with the VeriShield tool.
vrxdb_RVD4 appmap.exe	Displays the virtual memory map for an application and the shared libraries it uses.
findsrc.exe	Source code locating tool.
findlib.out	Converts an address in the system
lib\	
verix.lib	Main system library. This file is a copy of either verixs.lib or verixu.lib, if using the shared system library contained in the Verix firmware. Most applications should do this, so by default verix.lib is the same as verixs.lib.
voy.lib	Voyager library.
clibpi.a	
clibpi11.a	
verix11.lib	
verixn.lib	
voy11.lib	
voy11s.lib	
voy11s.o	
voyn.lib	
voyns.lib	
voyns.o	
voys.o	
include\	
ctype.h	Character handling header file.
errno.h	Verix error definition header file.

Table 3 Verix SDK Directory Structure \bin (continued)

Executable	Description
float.h	Floating point limits header file.
libvoy.h	Voyager library smart card interface header file.
limits.h	Integer limits header file.
math.h	Mathematics header file.
setjmp.h	Non-local jumps header file: For bypassing the normal function call and return discipline (useful for dealing with unusual conditions encountered in a low-level function of a program).
stdarg.h	Variable arguments header file.
stddef.h	Common definitions header file.
stdio.h	Input/output header file.
stdlib.h	General utilities header file.
string.h	String handling header file.
svc.h	Verix kernel services and system library interface header file.
svc_sec.h	Verix security services interface header file.
svc_swi.h	Software interrupt definition header file.
svctxo.h	Note: svctxo.h has been merged into svc.h, but a dummy svctxo.h is provided for backward compatibility. svc_swi.h is an internal file that should not be included directly by applications.
assert.h	Macro assert header file.
locale.h	Localization header file.
sdkver.h	SDK version header file.
time.h	Time and date header file.
config\	Configuration files. Contains several files required by the debugger.
examples\	
app.mak	Example makefile. Note that programs with a small number of source files may not need a makefile – a single VRXCC command can build the application.
tools.mak	Definitions of ARM tools and options that can be included in your makefiles.
lib.mak	Library makefile.
doc\	Current documentation files.

How to Build Verix ARM Applications

Installation

This section discusses how to build a Verix ARM application using the RealView Compilation Tools (RVCT) from the RealView Development Suite (RVDS).

Example Program

Use the following procedure to create an example program.

1 Create the program source file.

Use your preferred editor to create the file `HELLO.C` program source file which contains the following text (or in the online version of this manual, you can open a `.txt` file, copy and compile the text by clicking [Example](#)).

```
#include <string.h>
#include <svc.h>
char Greeting[] = "Hello Omni 5100";
void main (void)
{
    int display = open(DEV_CONSOLE, O_WRONLY);
    write(display, Greeting, strlen(Greeting));
    normal_tone();
    close(display);
}
```

NOTE



This is a non-standard declaration of `main` as used in C. The standard `int main` form can also be used, but the returned value is ignored.

2 Compile the program.

To compile and link a simple application:

a At the DOS command prompt, type the following:

```
vrxcc hello.c
```





This produces an executable file named `hello.out`. VRXCC works similar to other “cc” tools, that is, creates a list of source and/or object files. See [Compiling and Linking](#) for details.

3 Authenticate the `.OUT` file using the VeriShield file signing tool (a `.p7s` file is produced). See the documentation in the VeriShield help system for instructions on file signing.

4 Download the executable file and corresponding signature file.

The following instructions assume original factory settings and passwords. The procedure may need to be adjusted if terminal settings or passwords have been altered.

a Using an appropriate cable (supplied with the VVDTK), connect the COM1 port on the terminal to either the COM1 or COM2 serial port on your PC.

- b** Power up the terminal, and allow to go through the startup sequence.
- c** Do one of the following:
 - If `DOWNLOAD NEEDED` appears, begin the downloading process (go to [step h](#) on the PC, start [step d](#)).
 - If the display shows `SYSTEM ERROR`, press the system mode function key, F3, to enter system mode. Then go to [step d](#), enter the system mode password and enter the system password.
 - If the display reads `FAILED` or a previously-loaded application starts, unplug the terminal from power source, and plug it back in. While the copyright notice is being displayed, enter system mode by pressing F2 and F4 simultaneously.
- d** Enter the system mode password (the default password is Z66831 — press 1-ALPHA-ALPHA to enter the letter “Z”, refer to Alpha Key Support in the Verix V Operating System Programmers Manual, VPN 23230 for information on entering alpha characters):
 - Press ENTER () to accept the password. System mode menu 1 appears.
 - Press either the down arrow function key (under the display), or ENTER () to advance to Menu 2.
 - Press F2 to select `DOWNLOAD`.
- e** Press ENTER () to accept the default file Group 1. The screen displays `GROUP PASSWORD`.
 - Enter the password for file Group 1. The default password is Z66831, the same as the default system mode password and entered in the same way.
 - Press ENTER () to accept the password.
- f** Press F3 to select a *full* download.
- g** Press F3 to select COM1 (COM1 refers to the terminal port, not the PC port).

The message `DOWNLOADING NOW` appears.

- h** On the PC, start the download utility by typing one of the following commands:

```
ddl hello.out *go=hello.out hello.out.p7s
```

if using COM1 on PC:

```
ddl -p2 hello.out *go=hello.out hello.p7s
```

NOTE



The `ddl` commands *must* include the `hello.p7s` file produced in [step 3](#). Verix file authentication will fail if the `.p7s` file is not included in the download.

The naming convention for VeriShield-generated signature files is `.out.p7s`.

if using COM2 on the PC:

The download proceeds, displaying a progress bar on the terminal and a byte count on the PC. When it is complete, the message `DOWNLOAD DONE` appears (briefly) on the terminal screen and the PC reports `DOWNLOAD SUCCEEDED`.

5 Run the application.

Following the download, the terminal automatically resets, authenticates the file, then executes the application. You should see the greeting message displayed and hear the terminal beep.

To rerun the application, restart the unit.

Looking Ahead

To successfully develop Verix applications, you must understand the following:

- The C programming language. Familiarity with standard ANSI C is assumed. [Chapter 2](#) describes specific required aspects of C programming.
- The hardware and operating system environment in which Verix application programs run. The Application Programming Environment chapter in the Verix V Operating System Programmers Manual, VPN 23230 contains a high-level overview of this topic.
- How to use the tools in the SDK to write, compile, link, secure, download, and debug your programs. These tools are discussed in the succeeding chapters.

Programming Documentation

The goal of Verix on ARM is to be as compatible as possible with Verix 68K platforms. Therefore, most of the application and user-level Verix 68K documentation is still relevant.

[Porting Verix 68K Applications to Verix ARM](#) describes differences between the platforms and development environments.

Refer to the *Verix V Operating System Programmers Manual - VPN 23230* for descriptions of the APIs.



C Programming Concepts for Verix Terminals

This chapter describes aspects of C programming specific to Verix-based Omni series terminals. These include data representations, passing arguments to programs, managing memory, version number, and limitations of the standard C library implementation.

C Data Types

The standard C data types have the following internal representations:

Table 4 C Data Types

Type	Alignment	Bits	MIN	MAX
unsigned char	1	8	0	255
unsigned short	2 (halfword)	16	0	65535
int	4 (word)	32	-2147483648	2147483647
unsigned int	4	32	0	4294967295
long	4	32	-2147483648	2147483647
long long	8 (doubleword)	64	0	4294967295
float	4 (word)	32	1.755E-38	3.403E+38
double	8 (doubleword)	64	225E-308	1.798E+308
long double	8 (doubleword)	64	225E-308	1.798E+308
All pointers	4 (word)	32		

All data is big-endian that is, the address of a variable is the address of its most-significant byte.

See the *RealView Compilation Tools Compiler and Libraries Guide* for more information on C and C++ data types.

Program Arguments

The `run()` function (to start a new task) allows you to pass arguments to programs, for example:

```
run("test.out", "-o april.dat")
```

The argument string is split into separate arguments which are passed to `main()` through its `argc` and `argv` parameters. `argv[0]` contains the program name. In the example, `main()` receives the following arguments:

```
argc = 4
argv[0] = "TEST.OUT"
argv[1] = "-o"
argv[2] = "APRIL.DAT"
argv[3] = NULL
```

To pass arguments to a program started from system mode, set the `CONFIG.SYS` variable `*ARG` to the argument string. For example:

```
DDL test.out *GO=test.out *ARG="-o april.dat"
```

Note that the argument string is quoted because it includes an embedded blank.

Since `*ARG` is a `CONFIG.SYS` variable it is restricted to the limited CVLR character set. In particular, all letters are converted to uppercase. It is a good idea for applications to treat program arguments as case *insensitive*.

Stack and Heap

The amount of memory allocated for the stack and heap are defined in the executable (`.OUT`) file header and can be displayed and/or changed by the `VRXHDR` utility (see the compile and link instructions in [Chapter 3](#)). The default heap size is minimal because many programs do not use the heap.

Be careful about shrinking the heap size to zero. Standard I/O streams allocate buffer space from the heap by default. The function `get_env()` also allocates heap space for its result.

Stack and heap sizes are available to the program through the global variables:

```
long _stacksize;
```

```
long _heapsize;
```

The two library functions `long _stack_max (void)` and `long _heap_max (void)` return the amount of stack and heap space actually being used by the program. These can be useful during development for sizing the areas. The result of `_stack_max()` is an estimate that can understate the actual use by a few words. These variables and functions are declared in `<stdlib.h>`.

Version Number

The executable (`.out`) file header contains a version number that can be set and displayed by the `VRXHDR` utility. A program can access its version number through the global variable `const unsigned short _version` declared in `<stdlib.h>`. The high and low bytes contain the major and minor parts of the version number.

For example, Version 2.7 would be represented by the hexadecimal value `0x0207`. The two fields can be extracted by shifting and masking or by a cast, as shown below.

```
unsigned char *ver = (unsigned char *)&_version;

/* Now use ver[0] and ver[1] */
```

C Library

The Verix system library includes most of the standard ANSI C library. Some headers and functions that are inappropriate or not useful in the Verix environment were excluded, even if a technically-compliant implementation was possible. And a few implemented functions do not fully comply with the ANSI standard.

A few extensions to the standard library are included. Their definitions can be removed by defining the macro `_ANSI_STRICT` before including the headers.

The tables in the following sections lists all functions in the standard ANSI C library, plus Verix extensions, organized by header file. Missing, non-ANSI-compliant, and extension functions are noted and explained. Detailed descriptions of the standard functions can be found in any ANSI C reference.

The current Verix V system library implements a subset of the standard C library. Application developers re-use C code from open source distributions or other VeriFone product families. In many cases, versions of missing ANSI C APIs are implemented. The Verix V SDK is enhanced with the following ANSI C headers: [<assert.h>–Diagnostics](#), [<locale.h>–Localization](#), and [<time.h>–Date and Time](#).

<assert.h>– Diagnostics

The header `.assert` prints to `stderr`, which is often inappropriate in the Verix environment. Because unavailable calls abort, more useful customized replacements may be implemented.

not implemented `void assert (int expression)`

The macro `assert` is used to enforce assertions at critical places within the program. If `NDEBUG` is defined at the time `<assert.h>` is included, the `assert` macro does not generate any code.

It is highly recommended that `NDEBUG` is defined and possible `assert` failure conditions are handled for release builds. Even though the `assert` macro only expands to 1 line of C code, many asserts increases the application's Read Only (RO) constant data area and could also increase code size. If an assertion proves to be untrue, `assert` writes a message to the `CONFIG.SYS` variable `*ASSERT` in the current group and then causes a data abort (TYPE 1).

The message format is:

`"filename":"line number" "expression that caused the assertion failure"`

Example

`test.c:35 h>0`

This means file `test.c` at line 35, expression `h>0`, assertion failed.

The data abort fault address has the format: `0xYYYYZBAD`, where `YYYY` is the line number and `Z` is the group number. Thus, an abort TYPE 1 with a fault address `xxxxxBAD` indicates an assertion failure.

The sample source code below causes the screen crash on [Figure 2](#)

```
int main(int argc, char *argv[])
{
    int h;
    h = open(DEV_CONSOLE,0);
    assert (h>0);
    if ( h<=0 )
        return (-1);
    // The rest of the app...
}
```

[Figure 2](#) shows the assertion failure crash screen.

```

SYS MODE ERROR LOG
TYPE  1
TASK  2
TIME  070427093307
CPSR  000000030
PC    704201B2
LR    7042018B
ADDR  00231BAD
```

Figure 2 **Crash Screen**

The error log means that there is an assertion failure at line number 0x23, in group 1, task 2. The *ASSERT key in CONFIG.SYS in group 1 has the value of: ASSERTTT.C:35 h>0 which tells that the assertion failure is at file ASSERTTT.C, line number 35, expression h>0.

<locale.h>— Localization

Only the “C” locale is implemented and the “” locale is treated the same as the “C” locale. Five categories are defined in `locale.h` but are not implemented.

struct lconv *localeconv()

Returns a pointer to a structure describing the current locale. In Verix V, this is always the “C” locale.

Prototype `struct lconv *localeconv(void);`

Return Values

Success	Pointer to the current locale, in Verix V, always the “C” locale.
---------	---

Char *setlocale()

Sets the C locale.

Prototype `Char *setlocale(int category, const char *locale);`

Parameter

`*locale` The current locale, C.

`category` A defined category in `locale.h`.

Return Values The current implementation of this function only accepts a NULL pointer, C, or "" for the parameter `*locale`; anything else results in a return value of NULL.

**<ctype.h>—
Character
Handling**`int isalnum (int c)``int isalpha (int c)``int iscntrl (int c)``int isdigit (int c)``int isgraph (int c)``int islower (int c)``int isprint (int c)``int ispunct (int c)``int isspace (int c)``int isupper (int c)``int isxdigit (int c)``int tolower (int c)``int toupper (int c)`

extension `int isascii (int c)`
 Tests if `c` is in the ASCII character range (0–0x7F).

extension `int toascii (int c)`
 Forces `c` to ASCII character range by clearing most-significant bit
 (`c = c & 0x7F`).

**<math.h>—
Mathematics**`double acos (double x)``double asin (double x)``double atan (double x)``double atan2 (double x)``double cos (double x)``double sin (double x)``double tan (double x)``double cosh (double x)``double sinh (double x)``double tanh (double x)``double exp (double x)``double frexp (double value, int *exp)``double ldexp (double x, int exp)``double log (double x)``double log10 (double x)``double modf (double value, double *iptr)`

```
double pow (double x, double y)
```

```
double sqrt (double x)
```

```
double ceil (double x)
```

```
double fabs (double x)
```

```
double floor (double x)
```

```
double fmod (double x, double y)
```

```
extension double cotan (double x)
Trigonometric cotangent.
```

```
extension double hypot (double x, double y)
Computes sqrt(x*x + y*y) using an algorithm that avoids overflow for large
argument values.
```

<setjmp.h>— Non-local Jumps

```
int setjmp (jmp_buf env)
```

```
void longjmp (jmp_buf env, int val)
```

<signal.h>—Signal Handling

The header is not included. The Verix operating system does not support Unix-style signals on which the ANSI C signal model is based.

```
not implemented void (*signal(int sig, void (*func)(int))) (int)
```

```
not implemented int raise (int sig)
```

<stdarg.h>— Variable Arguments

```
void va_start (va_list ap, parm)
```

```
type va_arg (va_list ap, type)
```

```
void va_end (va_list ap)
```

<stdio.h>—Input/ Output

On Verix-based Omni series terminals, `stdin` reads from the keyboard while `stdout` and `stderr` write to the display (default). `stdout` and `stderr` are identical and unbuffered. The console is opened on the first use of any of the streams. The display is erased and the keyboard buffer is cleared at this time (even if the console was already open.) Only one task can have the console open at any time. Applications cannot assume that the standard streams are automatically available.

Automatic cleanup at program termination, such as flushing buffers and closing open files, is not provided.

Because the standard input/output library adds an additional layer of overhead to the Verix operating system I/O interface, most applications prefer to bypass it and use direct calls to `open()`, `read()`, `write()`, and so on.

```
int remove (const char *filename)
```

```
int rename (const char *old, const char *new)
```


not implemented	<p><code>FILE* tmpfile (void)</code> Automatic deletion of files on program exit is complex, and most applications never terminate.</p> <p><code>char* tmpnam (char *s)</code></p> <p><code>int fclose (FILE *stream)</code></p> <p><code>int fflush (FILE *stream)</code></p> <p><code>FILE* fopen (const char *filename, const char *mode)</code></p> <p><code>FILE* freopen (const char *filename, const char *mode, FILE *stream)</code></p> <p><code>void setbuf (FILE *stream, char *buf)</code></p>
non-compliant	<p><code>void setvbuf (FILE *stream, char *buf, int mode, size_t size)</code> Line buffering is not implemented.</p> <p><code>int fprintf (FILE *stream, const char *format, ...)</code> Even though long long data type is supported in Verix V, the formatting of variables of long long data type is not supported. This applies to all the printf and scanf family of functions.</p> <p><code>int fscanf (FILE *stream, const char *format, ...)</code></p> <p><code>int printf (const char *format, ...)</code></p> <p><code>int scanf (const char *format, ...)</code></p> <p><code>int sprintf (char *s, const char *format, ...)</code></p> <p><code>int sscanf (char *s, const char *format, ...)</code></p> <p><code>int vfprintf (FILE *stream, const char *format, va_list arg)</code></p> <p><code>int vprintf (const char *format, va_list arg)</code></p> <p><code>int vsprintf (char *s, const char *format, va_list arg)</code></p> <p><code>int fgetc (FILE *stream)</code></p>
non-compliant	<p><code>char* fgets (char *s, int n, FILE *stream)</code> Treats either '\n' or '\r' as end-of-line.</p> <p><code>int fputc (int c, FILE *stream)</code></p> <p><code>int fputs (const char *s, FILE *stream)</code></p> <p><code>int getc (FILE *stream)</code></p> <p><code>int getchar (void)</code></p>
not implemented	<p><code>char* gets (char *s)</code> Dangerous because there is no reliable way to prevent buffer overflow. Detecting an end-of-line is also problematic.</p> <p><code>int putc (int c, FILE *stream)</code></p> <p><code>int putchar (int c)</code></p> <p><code>int puts (const char *s)</code></p> <p><code>int ungetc (int c, FILE *stream)</code></p>

```

size_t fread (void *ptr, size_t size, size_t nmemb, FILE
              *stream)

size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE
              *stream)

int fgetpos (FILE *stream, fpos_t *pos)

int fseek (FILE *stream, long offset, int whence)

int fsetpos (FILE *stream, const fpos_t *pos)

long ftell (FILE *stream)

void rewind (FILE *stream)

void clearerr (FILE *stream)

int feof (FILE *stream)

int ferror (FILE *stream)

```

not implemented	<pre>void perror (const char *s)</pre> <p>Printing to stderr is often inappropriate in the Verix environment. <code>strerror</code> provides a more flexible alternative.</p>
extension	<pre>int fileno (FILE *stream)</pre> <p>Returns operating system handle associated with <code>stream</code>.</p>
extension	<pre>FILE *fdopen (int handle, const char *type)</pre> <p>Opens a stream associated with an operating system handle.</p>

stdio.h uses the function below:

FILE *tmpfile()

Creates a temporary file of mode “wb+” that is automatically removed when closed or when the program terminates

Prototype FILE *tmpfile(void);

Return Values Returns a stream, or NULL if it could not create the file. The most likely causes of returning NULL are:

- The application called tmpfile() more than TMP_MAX number of times within the same second in the same group,
- The application does not have enough heap memory.

This function is implemented in two parts.

- 1 Verix V SDK - verix.lib and verixn.lib. This part implements the tmpfile() function and removes the files it created when fclose() is called for these temporary files or when the program exits normally.
- 2 Verix V Operating System - In case of a power fail or when entering System Mode before the program gets a chance to call fclose() or exit(), the temporary files created by tmpfile() remains in the file system. The OS scans through the file system for special file names with the following pattern during OS start up, and removes these files:

<DELETE ME>T\$yymdddhmmssx

The file name pattern is selected to minimize the possibility of removing a valid file created by an application where:

- < and > characters are illegal characters in Windows file names,
- x is from A to Z, and
- yyymmdddhmmss is the time the name was generated.

Example: The file <DELETE ME>T\$070503032923B is removed by the OS during start up.

**<stdlib.h>—
General Utilities**

The `wchar_t` (wide character) is identical to `char`. The functions that manipulate wide and multi-byte characters are not supported. In addition to the following extended functions, `<stdlib.h>` defines several non-standard global variables:

```
const long _stacksize      Allocated size of task's stack (bytes)
const long _heapsize       Allocated size of task's heap (bytes)
const unsigned short _version  Code file version number
```

```
double atof (const char *nptr)
```

```
int atoi (const char *nptr)
```

```
long atol (const char *nptr)
```

```
double strtod (const char *nptr, char **endptr)
```

```
long strtol (const char *nptr, char **endptr, int base)
```

```
unsigned long strtoul (const char *nptr, char **endptr, int base)
```

```
int rand (void)
```

non-compliant

```
void srand (unsigned int seed)
```

The default rand seed is 0, instead of 1.

```
void* calloc (size_t nmemb, size_t size)
```

```
void free (void *ptr)
```

```
void* malloc (size_t size)
```

```
void* realloc (void *ptr, size_t size)
```

not implemented

```
void abort (void)
```

Depends on signals, which are not available.

not implemented

```
int atexit (void (*func)(void))
```

Requires static storage space, if used. V*670 applications generally do not terminate.

non-compliant

```
void exit (int status)
```

This is the same as Verix V API `_exit()`, please refer to Verix V Operating System Programmers Manual for details.

```
char* getenv (const char *name)
```

not implemented

```
int system (const char *string)
```

No command processor is available.

```
void* bsearch (const void *key, const void *base, size_t nmemb,
               size_t size, int (*compar)(const void *, const
               void *))
```

```
void qsort (void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *))
```

```
int abs (int j)
```

```
div_t div (int numer, int denom)
```

```
long labs (long j)
```

```
ldiv_t ldiv (long numer, long denom)
```

not implemented

```
int mblen (const char *s, size_t n)
```

not implemented	int mbtowc (wchar_t *pwc, const char *s, size_t n)
not implemented	int wctomb (char *s, wchar_t wchar)
not implemented	size_t mbstowcs (wchar_t *pwcs, const char *s, size_t n)
not implemented	size_t wcstombs (char *s, const wchar_t *pwcs, size_t n)
extension	unsigned short _syslib_version (void) Returns system library version.
extension	long _heap_max (void) Returns calling task's maximum heap use to date.
extension	long _stack_max (void) Returns calling task's maximum stack usage to date. This is an estimate that may slightly underestimate actual usage.
extension	int _debugging (void) Returns a non-zero value if the calling task is being debugged; zero if not.
extension	void* mallocl (long size) Same as malloc except size is long.
extension	void* reallocl (void *ptr, long size) Same as realloc except size is long.
extension	void *malloc_all (long *size) Returns the last contiguous block of memory in the heap. The purpose of this routine is to allow applications to grab the whole heap before allocating any memory so that the applications can manage the heap themselves if necessary.
extension	int check_heap (void**addr, unsigned long *data) Checks the integrity of heap. This function scans the heap in search for bad block headers. Returns the following values: <ul style="list-style-type: none"> • 0, no errors. • 1, bad area header. • 2, bad block header. • 3, zero size block. • 4, block extends past end of heap. If an error is found, *addr is set to the address of the offending block. It sends *data to its header if *addr and data are not NULL.

Example:

```
void *addr
unsigned long data
int status;

status = check_heap(&addr, &data)
if (status)
{
    //something wrong with the heap
    dbprintf("bad heap at address: 0x%x, data: 0x%x\n", addr,
data);
}
```

<string.h>—String Handling

	<code>void* memcpy (void * s1, const void *s2, size_t n)</code>
	<code>void* memmove (void *s1, const void *s2, size_t n)</code>
	<code>char* strcpy (char * s1, const char *s2)</code>
	<code>char* strncpy (char *s1, const char *s2, size_t n)</code>
	<code>char* strcat (char *s1, const char *s2)</code>
	<code>char* strncat (char *s1, const char *s2, size_t n)</code>
	<code>int memcmp (const void *s1, const void *s2, size_t n)</code>
	<code>int strcmp (const char *s1, const char * s2)</code>
	<code>int strncmp (const char *s1, const char *s2, size_t n)</code>
not implemented	<code>int strcoll (const char *st, const char *s2)</code> Locale related. See <locale.h>— Localization .
not implemented	<code>size_t strxfrm (char *s1, const char *s2, size_t n)</code> Locale related. See <locale.h>— Localization .
	<code>void* memchr (const void *s, int c, size_t n)</code>
	<code>char* strchr (const char *s, int c)</code>
	<code>size_t strcspn (const char *s1, const char *s2)</code>
	<code>char* strpbrk (const char *s1, const char *s2)</code>
	<code>char* strrchr (const char *s, int c)</code>
	<code>size_t strspn (const char *s1, const char *s2)</code>
	<code>char* strstr (const char *s1, const char *s2)</code>
	<code>char* strtok (char *s1, const char *s2)</code>
	<code>void* memset (void *s, int c, size_t n)</code>
	<code>char* strerror (int errnum)</code>
	<code>size_t strlen (const char *s)</code>
extension	<code>void* memccpy (void *s1, const void *s2, int c, size_t n)</code> Copies characters from s2 to s1 until either character c or n other characters have been copied.

<time.h>—Date and Time

The header is not included. The Verix real-time clock returns a result already broken down into month, day, hour, minute, and so on. Converting this to the arithmetic `time_t` would be costly and unnecessary.

not implemented	<code>clock_t clock (void)</code>
not implemented	<code>double difftime (time_t t1, time_t t2)</code>
not implemented	<code>time_t mktime (struct tm *timeptr)</code>
not implemented	<code>time_t time (time_t *timer)</code>
not implemented	<code>char* asctime (const struct tm *timeptr)</code>

```

not implemented  char* ctime (const struct tm *timeptr)
not implemented  char* gmtime (const struct tm *timeptr)
not implemented  char* localtime (const struct tm *timeptr)
not implemented  size_t strftime (char *s, size_t maxsize, const char *format,
                                const struct tm *timeptr)

```

ANSI C represents calendar time as seconds since January 1, 1900. Verix V uses January 1, 1980 as a starting point. The constant `seconds_offset_1900` is defined as the seconds between January 1, 1900 and January 1, 1980. Verix V supports time zone and daylight saving, and automatically adjusts DST based on regional policy.

Time Manipulation Functions

Following are the functions used in time manipulation:

- `clock_t clock(void)`
This calls the Verix V API `read_ticks()`, when divided by `CLOCKS_PER_SEC`, returns the seconds elapsed since the terminal was powered up or reset.
- `time_t time(time_t *pt)`
This calls the Verix V API `read_clock()` and adds `seconds_offset_1900` to return the ANSI C calendar time. If `pt` is not `NULL`, the return value is also assigned to `*pt`.
- `double difftime(time_t t1, time_t t0)`
Computes the difference between two times `t1` and `t0`. The result is positive if `t1` is a later time than `t0`.
- `time_t mktime(struct tm *ptm)`
Converts the local time in the structure pointed to by `ptm` into a calendar time.

Time Conversion Functions

Following are the functions used for time conversion:

- `char *asctime(const struct tm *timeptr)`
- `char *ctime(const time_t *timer)`
- `struct tm *gmtime(const time_t *timer)`
- `struct tm *localtime(const time_t *timer)`
- `size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr)`

The return values for `*gmtime` and `*localtime` is a structure statistically allocated and shared by these two functions. Each time either of these functions is called, the contents of the structure are overwritten. It is safe to copy the return values into a local structure to avoid being overwritten by any subsequent calls of these functions. If time zone is not set in the OS, `gmtime` returns the current system time. The `%Z` parameter on `strftime()` returns zero.

Timezone and DST Implementation

The environment variables `*TZ` and `*TZRULE` need to be downloaded in the OS and set to GID1 to process the time zone. When set to other GIDs, `*TZ` and `*TZRULE` are ignored. These variables hold the parameter for setting, changing, adding, and deleting time zone regional rules and definitions. When time zone is defined, the OS clock is updated based on the current time zone value.

Download `*TZ` and `*TZRULE` using the following methods:

- DDL on PC command line.
- USB Flash where `CONFIG. $$$` file in `VERIFONE.ZIP`.
- DDL or download() API using `CONFIG. $$$` file.
- VeriCentre and modem downloads.
- System Mode Edit menu via `CONFIG.SYS` restart.

`*TZ` and `*TZRULE` parameters are processed at the end of the download, and they must be downloaded separately.

Example:

```
DDL -p1 *TZ=PHT
DDL -p1 *TZRULE=DEL, PHT
```

For the Split RTC to function, the correct clock and time zone for the region must be downloaded when upgrading to an OS that supports time zone.

*TZ

This holds the parameter for time zone code, and sets the time zone in the OS. When set, the OS searches for the offset time in the file `TZULE.TXT`.

Example

```
*TZ=EST
```

This means that the time zone is UTC-05 (Eastern Standard Time). The time zone automatically switches from standard time to daylight saving once it approaches the DST regional policy date also stored in the `TZULE.TXT` file.

```
*TZ=CDT-MEXICO
```

Central Daylight Time is used in the summer of some US and Mexican states, and Canadian provinces, all with different time zone implementation. There are time zone codes used by more than one region having different start and end dates. To specify the correct regional policy, append “-<country>” in the parameter.

***TZRULE**

This environment variable is used in adding, changing, and deleting time zone offset times and DST regional policies.

Changing/Adding a DST Regional Policy

To change/add a DST regional policy, use the format below with no spaces:

```
*TZRULE=STD,OFFSET1,DST,OFFSET2,START[ /TIME ],END[ /TIME ]
```

where:

- STD - is the abbreviation of the standard time zone.
- OFFSET1 - is the offset time (HH:MM:SS) with respect to UTC of the standard time zone.
- DST - is the abbreviation of the time zone during daylight saving time.
- OFFSET2 - is the offset time (HH:MM:SS) with respect to UTC of the DST time zone.
- START/
END - are the dates when DST takes effect and expires, respectively. It takes the format:
Mm.w.d {month=1-12, week=1-5 (5 is always last), day=0-6}
 - month=1 means January, month=12 means December
 - week is always start of the first day of the month
 - day=0 means Sunday, day=6 means Saturday

Example *TZRULE=EST,UTC-05,EDT,UTC-04,MO3.2.0/02,M11.1.0/02,

This means that the user is changing/adding the information on EDT/EST region. The DST in this case starts on Sunday, March 9, 2008 at 2:00AM local standard time, and ends on Sunday, November 2, 2008 at 2am daylight time.

If more than one region uses the time zone code, the settings should be:

```
*TZRULE=CST-USA,UTC-06,CDT-USA,UTC-05,M03.2.0/02,M11.1.0/02,
```

The parameter must be complete, otherwise, the setting is ignored. If the user needs to use the old values and change some of the information, include 0 to retain the old information. It is also required to use the standard time zone code so that the OS knows exactly what to change.

If the time zone code already exists, the user is trying to change what is already defined on TZULE.TXT files. Otherwise, it means that the user is adding a new time zone code.

Deleting a DST Regional Policy

Deleting a time zone means that the user is removing the time zone from the time zone regional rules file TZRULE.TXT. The time zone currently set is not deleted.

To delete a time zone code, set the DEL keyword before the time zone abbreviation on *TZRULE variable.

Example *TZRULE=DEL,EST

This means that all information stored on EST and DST are deleted. If the time zone abbreviation does not exist on the file, the settings are ignored.

The user can revert back to the original time zone rules file or to the default file by deleting all the files including GID0.

TZRET

This holds the return values for *TZ and *TZRULE and indicates if they are successfully set. TZRET is accessed on the System Mode CONFIG.SYS.

If TZRET=1 *TZ and *TZRULE are successfully set.

If TZRET=2 *TZ and *TZRULE failed to set.

TZRULE.TXT File

The TZRULE.TXT is located in the OS drive S:, which is hidden from the user. This file is downloaded once an OS with timezone implementation is downloaded. If a time zone is changed or added, the OS copies the TZRULE.TXT file on the flash drive F: as a protected file #tzrules.txt.

The OS always checks if #tzrule.txt on drive F already exists, if so, the OS uses it, otherwise, the OS uses S:/tzrule.txt.

NOTE



The tzrule.txt on drive S: is always the default file if the terminal does not update tzrule.txt or it does not exist on drive F:.

If the default values need to be retained, clear the memory including GID0. To clear GID0:

- 1 Go to SYS MODE>MEMORY FUNCTIONS-CLEAR MEM.
- 2 Enter the correct password.
- 3 Press the PF Key 3 (third purple key below the display).
- 4 Press the F4 key.
- 5 Restart the terminal.

Time Zone and DST Application Support

To implement the time zone and automatic DST change using the enhanced SDK <time.h>:

- 1 Determine if the currently loaded OS and SDK versions support time zone. Use `_syslib_version()` and `_SYS_VERSION` in `stdlib.h` and `TZ_SYSVERSION` in `time.h` of the SDK.

`TZ_SYSVERSION 0x100` is the `sys.lib` version that implements time zone.

- 2 Determine the current time zone offset time.

The offset time is determined by obtaining the difference of the return value of `time()` and `gmtime()`.

- 3 Determine the current DST status.

The `tm_isdst` flag on `struct tm` in `time.h` determines if the current time zone is on DST.

- 4 Determine the current time zone setting.

The OS provides an API that retrieves the time zone currently set.

getTZdata()

Retrieves the currently set time zone.

Prototype `int getTZdata(struct tz *mytz);`

Parameters The struct tz returns the following 'null' terminated fields.

<code>char *tz_std(20)</code>	:	Standard time zone definition.
<code>char *offset1(15)</code>	:	Standard offset time with respect to UTC.
<code>char *tz_dst(20)</code>	:	Daylight time zone definition.
<code>char *offset2(15)</code>	:	Daylight offset time with respect to UTC.
<code>char *start_date(20)</code>	:	Start date of DST.
<code>char *end_date(20)</code>	:	End date of DST.

Return Values

0	Success
-1	Fail

NOTE



If there is no DST definition on the current time zone set, the field that refers to DST information returns NULL.

Below is the time zone rules files table showing the default values stored on TZRULE.TXT file. If the user needs to change the values in the file, download *TZRULE with parameters having the format:

```
*TZRULE=ACST, UTC+09:30, ACDT, UTC+10:30,M10.1.0/02,M04.1.0/02,
```

Table 5 Time Zone Rules File Table

STD	Offset	DST	Offset	DST Start	DST End
ACST	UTC+09:30	ACDT	UTC+10:30	M10.1.0/02	M04.1.0/02
AST	UTC-04	ADT	UTC-03	M03.2.0/02	M11.1.0/02
AEST	UTC+10	AEDT	UTC+11	M10.1.0/02	M04.1.0/02
AFT	UTC+04:30	None	None	None	None
AKST	UTC-09	AKDT	UTC-08	M03.2.0/02	M11.1.0/02
AMT	UTC+04	AMST	UTC+05	M03.5.0/02	M10.4.0/03
ANAT	UTC+12	ANAST	UTC+13	M03.5.0/02	M10.4.0/03
ART	UTC-03	None	None	None	None
AST-ARABIA	UTC+03	None	None	None	None
AST-BRAZIL	UTC-05	None	None	None	None
AT	UTC-01	None	None	None	None
AWST	UTC+08	AWDT	UTC+09	M10.4.0/03	M03.5.0/02
AZOT	UTC-01	AZOST	UTC	M03.5.6/00	M10.4.0/01
AZT	UTC+04	AZST	UTC+05	M03.5.0/04	M10.4.0/05
BAT	UTC+03	BADT	UTC+04	M04.1.2/03	M10.1.3/04
BDT	UTC+06	None	None	None	None
BET	UTC-11	None	None	None	None
BNT	UTC+08	None	None	None	None
BORT	UTC+08	None	None	None	None
BOT	UTC-04	None	None	None	None
BRA	UTC-03	None	None	None	None
BTT	UTC+06	None	None	None	None
CAT	UTC+02	None	None	None	None
CCT	UTC+06:30	None	None	None	None
CST-USA	UTC-06	CDT-USA	UTC-05	M03.2.0/02	M11.1.0/02
CST-MEXICO	UTC-06	CDT-MEXICO	UTC-05	M04.1.0/02	M10.4.0/02
CST-CANADA	UTC-06	CDT-CANADA	UTC-05	M03.2.0/02	M11.1.0/02
CET	UTC+01	CEST	UTC+02	M03.5.0/04	M10.4.0/05
CHAST	UTC+12:45	CHADT	UTC+13:45	M09.4.0/02:45	M04.1.0/03:45
CKT	UTC-10	None	None	None	None
CLT	UTC-04	CLST	UTC-03	M10.2.6/00	M03.2.6/00
COT	UTC-05	None	None	None	None
CST-CHINA	UTC+08	None	None	None	None
CUBT	UTC-05	CST-CUBA	UTC-04	M03.2.6/00	M10.4.0/01
CUT	UTC	None	None	None	None

Table 5 Time Zone Rules File Table (continued)

STD	Offset	DST	Offset	DST Start	DST End
CVT	UTC-01	None	None	None	None
CWT	UTC+08:45	None	None	None	None
CXT	UTC+07	None	None	None	None
DAVT	UTC+07	None	None	None	None
DDUT	UTC+10	None	None	None	None
DNT	UTC+01	DST	UTC+02	None	None
EAST	UTC-06	EASST	UTC-05	M10.2.6/10	M03.2.6/10
EAT	UTC+03	None	None	None	None
ECT	UTC-04	None	None	None	None
ECT-ECUADOR	UTC-05	None	None	None	None
EST	UTC-05	EDT	UTC-04	M03.2.0/02	M11.1.0/0
EET	UTC+02	EEST	UTC+03	M03.5.0/03	M10.4.0/04
EMT	UTC+01	None	None	None	None
EST-BRAZIL	UTC-03	None	None	None	None
FST	UTC-02	FDT	UTC-01	None	None
FJT	UTC+12	FJST	UTC+13	None	None
FKT	UTC-04	FKST	UTC-03	M09.1.0/02	M04.3.0/02
FWT	UTC+01	FST-FRENCH	UTC+02	M03.5.0/02	M10.4.0/03
GALT	UTC-06	None	None	None	None
GAMT	UTC-09	None	None	None	None
GET	UTC+04	GEST	UTC+05	None	None
GFT	UTC-03	None	None	None	None
GILT	UTC+12	None	None	None	None
GMT-BSDT	UTC	BSDT	UTC+02	M03.5.0/01	M10.4.0/02
GMT-BST	UTC	BST	UTC+01	M03.5.0/01	M10.4.0/02
GMT	UTC	IST	UTC+01	M03.5.0/01	M10.4.0/02
GST-GUAM	UTC+10	None	None	None	None
GST-GULF	UTC+04	None	None	None	None
GST-GREENLAND	UTC-03	None	None	None	None
GST-SOUTHGEORGIA	UTC-02	None	None	None	None
GYT	UTC-04	None	UTC-09	None	None
HAST	UTC-10	HADT	UTC-09	M03.2.0/02	M11.1.0/02
HKT	UTC+08	0	0	0	0
HOE	UTC+01	0	0	0	0
HST	UTC-10	0	0	0	0
ICT	UTC+07	0	0	0	0
IDLE	UTC+12	0	0	0	0
IDLW	UTC-12	0	0	0	0
IST-ISRAEL	UTC+02	IDT-ISRAEL	UTC+03	M03.4.5/02	M10.1.0/02
IST-INDIA	UTC+05:30	IDT-INDIA	UTC+06:30	0	0

Table 5 Time Zone Rules File Table (continued)

STD	Offset	DST	Offset	DST Start	DST End
IRT	UTC+03:30	IRDT	UTC+04:30	M03.3.4/00	M09.3.5/00
IOT	UTC+05	0	0	0	0
IRKT	UTC+08	IRKST	UTC+09	M03.5.0/02	M10.4.0/03
ITA	UTC+01	0	0	0	0
JAVA	UTC+07	0	0	0	0
JAYT	UTC+09	0	0	0	0
JST	UTC+09	0	0	0	0
KST	UTC+09	KDT	UTC+10	0	0
KGST	UTC+05	KGST	UTC+06	0	0
KOST	UTC+12	0	0	0	0
KRAT	UTC+07	KRAST	UTC+08	M03.5.0/02	M10.4.0/03
LHST	UTC+10:30	LHDT	UTC+11	M03.5.0/02	M10.4.0/02
LIGT	UTC+10	0	0	0	0
LINT	UTC+14	0	0	0	0
LKT	UTC+06	0	0	0	0
MAGT	UTC+11	MAGST	UTC+12	M03.5.0/02	M10.4.0/03
MAL	UTC+08	0	0	0	0
MART	UTC+09:30	0	0	0	0
MAWT	UTC+06	0	0	0	0
MST-USA	UTC-07	MDT-USA	UTC-06	0	0
MST-MEXICO	UTC-07	MDT-MEXICO	UTC-06	M04.1.0/02	M10.4.0/02
MEX	UTC-06	0	0	0	0
MHT	UTC+12	0	0	0	0
MMT	UTC+06:30	0	0	0	0
MPT	UTC+10	0	0	0	0
MSK	UTC+03	MSD	UTC+04	M03.5.0/02	M10.4.0/03
MT	UTC+08:30	0	0	0	0
MUT	UTC+04	0	0	0	0
MVT	UTC+05	0	0	0	0
MYT	UTC+08	0	0	0	0
NCT	UTC+11	0	0	0	0
NST	UTC-03:30	NDT	UTC-02:30	0	0
NFT	UTC+11:30	0	0	0	0
NOR	UTC+01	0	0	0	0
NOVT	UTC+06	NOVST	UTC+07	M03.5.0/02	M10.4.0/03
NPT	UTC+05:45	0	0	0	0
NRT	UTC+12	0	0	0	0
NSUT	UTC+06:30	0	0	0	0
NUT	UTC-11	0	0	0	0
NZST	UTC+12	NZDT	UTC+13	M09.4.0/02	M04.1.0/03

Table 5 Time Zone Rules File Table (continued)

STD	Offset	DST	Offset	DST Start	DST End
OMST	UTC+06	OMSST	UTC+07	M03.5.0/02	M10.4.0/03
PST-USA	UTC-08	PDT-USA	UTC-07	M03.2.0/02	M11.1.0/02
PST-CANADA	UTC-08	PDT-CANADA	UTC-07	M03.2.0/02	M11.1.0/02
PST-MEXICO	UTC-08	PDT-MEXICO	UTC-07	M04.1.0/02	M10.4.0/02
PET	UTC-05	0	0	0	0
PETT	UTC+12	PETST	UTC+13	M03.5.0/02	M10.4.0/02
PHOT	UTC+13	0	0	0	0
PHT	UTC+08	0	0	0	0
PKT	UTC+05	0	0	0	0
PMT	UTC-03	PMDT	UTC-02	M03.2.0/02	M11.1.0/02
PNT	UTC-08:30	0	0	0	0
PONT	UTC+11	0	0	0	0
PWT	UTC+09	0	0	0	0
PYT	UTC-04	PYST	UTC-03	M10.3.5/00	M03.2.5/00
R1T	UTC+02	0	0	0	0
R2T	UTC+03	0	0	0	0
RET	UTC+04	0	0	0	0
SAST	UTC+02	0	0	0	0
SBT	UTC+11	0	0	0	0
SCT	UTC+04	0	0	0	0
SET	UTC+01	0	0	0	0
SGT	UTC+08	0	0	0	0
SRT	UTC+03	0	0	0	0
SWT	UTC+01	SST-SWEDEN	UTC+02	M03.5.0/02	M10.4.0/03
SST-SAMOA	UTC-11	0	0	0	0
SST-SOUTHSAMOA	UTC+07	0	0	0	0
TFT	UTC+05	0	0	0	0
THA	UTC+07	0	0	0	0
THAT	UTC-10	0	0	0	0
TJT	UTC+05	0	0	0	0
TKT	UTC-10	0	0	0	0
TMT	UTC+05	0	0	0	0
TOT	UTC+13	0	0	0	0
TRUK	UTC+10	0	0	0	0
TST	UTC+03	0	0	0	0
TVT	UTC+12	0	0	0	0
ULAT	UTC+08	ULAST	UTC+09	0	0
USZ1	UTC+02	USZ1S	UTC+03	M03.5.0/02	M10.4.0/03
UTC	UTC	0	0	0	0
UTZ	UTC-03	0	0	0	0

Table 5 Time Zone Rules File Table (continued)

STD	Offset	DST	Offset	DST Start	DST End
UYR	UTC-03	0	0	0	0
UZT	UTC+05	0	0	0	0
VET	UTC-04:30	0	0	0	0
VLAT	UTC+10	0	0	0	0
VUT	UTC-11	VLAST	UTC+11	M03.5.0/02	M10.4.0/03
WAKT	UTC+12	0	0	0	0
WAT	UTC+01	0	0	0	0
WET	UTC	WAST	UTC+02	M09.1.0/02	M04.1.0/03
WFT	UTC+12	WEST	UTC+01	M03.5.0/02	M10.4.0/03
WGT	UTC-03	0	0	0	0
WIB	UTC+07	WGST	UTC-02	M03.5.5/10	M10.4.5/11
WITA	UTC+08	0	0	0	0
WIT	UTC+09	0	0	0	0
WST-BRAZIL	UTC+04	0	0	0	0
WST-SAMOA	UTC-11	0	0	0	0
WUT	UTC+01	0	0	0	0
YAKT	UTC+09	YAKST	UTC+10	M03.5.0/03	M10.4.0/04
YAPT	UTC+10	0	0	0	0
YST	UTC-09	YDT	UTC-08	M03.2.0/02	M11.1.0/02
YEKT	UTC+05	YEKST	UTC+06	M03.5.0/02	M10.4.0/03
Z	UTC	0	0	0	0

Table 6 below defines the time zone codes.

Table 6 Time Zone Code Definition

Code	Definition
ACDT	Australian Central Daylight Time
ACST	Australian Central Standard Time
ADT	Atlantic Daylight Time
AEDT	Australian Eastern Daylight Time
AEST	Australian Eastern Standard Time
AFT	Afghanistan Time
AKDT	Alaska Daylight Time
AKST	Alaska Standard Time
AMST	Armenia Summer Time
AMT	Armenia Time
ANAST	Anadyr Summer Time
ANAT	Anadyr Time
ART	Argentina Time
AST	Atlantic Standard Time
AST-Arabia	Arabia Standard Time

Table 6 Time Zone Code Definition (continued)

Code	Definition
AST-Brazil	Acre Standard Time (Brazil)
AT	Azores Time
AWDT	Australian Western Daylight Time
AWST	Australian Western Standard Time
AZOST	Azores Summer Time
AZOT	Azores Time
AZST	Azerbaijan Summer Time
AZT	Azerbaijan Time
BADT	Baghdad Daylight Time
BAT	Baghdad Time
BDST	British Double Summer Time
BDT	Bangladesh Time
BET	Bering Standard Time
BNT	Brunei Darussalam Time
BORT	Borneo Time (Indonesia)
BOT	Bolivia Time
BRA	Brazil Time
BST	British Summer Time
BTT	Bhutan Time
CAT	Central Africa Time
CCT	Cocos Islands Time (Indian Ocean)
CDT-USA	Central Daylight Time (USA)
CDT-Mexico	Central Daylight Time (Mexico)
CDT-Canada	Central Daylight Time (Canada)
CEST	Central Europe Summer Time
CET	Central Europe Time
CHADT	Chatham Daylight Time (New Zealand)
CHAST	Chatham Standard Time (New Zealand)
CKT	Cook Islands Time
CLST	Chile Summer Time
CLT	Chile Time
COT	Colombia Time
CST	Central Standard Time (USA, Canada, Mexico)
CST-China	China Time
CST-Cuba	Cuba Summer Time
CUBT	Cuba Time
CUT	Coordinated Universal Time
CVT	Cape Verde Time
CWT	Central West Time (Australia)
CXT	Christmas Island Time (Indian Ocean)

Table 6 Time Zone Code Definition (continued)

Code	Definition
DAVT	Davis Time (Antarctica)
DDUT	Dumont-d'Urville Time (Antarctica)
DNT	Dansk Normal
DST	Dansk Summer
EASST	Easter Island Summer Time (Chile)
EAST	Easter Island Time (Chile)
EAT	East Africa Time
ECT	Eastern Caribbean Time
ECT-Ecuador	Ecuador Time
EDT	Eastern Daylight Time (USA)
EEST	Eastern Europe Summer Time
EET	Eastern Europe Time
EMT	Norway Time
EST	Eastern Standard Time (USA)
EST-Brazil	Eastern Brazil Standard Time
FDT	Fernando de Noronha Daylight Time
FJST	Fiji Summer Time
FJT	Fiji Time
FKST	Falkland Islands Summer Time
FKT	Falkland Islands Time
FST	Fernando de Noronha Standard Time (Brazil)
FST-French	French Summer Time
FWT	French Winter Time
GALT	Galapagos Time
GAMT	Gambier Time
GEST	Georgia Summer Time
GET	Georgia Time
GFT	French Guiana Time
GILT	Gilbert Islands Time
GMT	Greenwich Mean Time
GST-Guam	Guam Standard Time
GST-Gulf	Gulf Standard Time
GST-Greenland	Greenland Standard Time
GST-SouthGeorgia	South GeorgiaTime
GYT	Guyana Time
HADT	Hawaii-Aleutian Daylight Time (USA)
HAST	Hawaii-Aleutian Standard Time (USA)
HKT	Hong Kong Time
HOE	Spain Time
HST	Hawaiian Standard Time

Table 6 Time Zone Code Definition (continued)

Code	Definition
ICT	Indochina Time
IDLE	International Date Line East
IDLW	International Date Line West
IDT-Israel	Israeli Daylight Time
IDT-India	Indian Daylight Time
IRDT	Iran Daylight Time
IOT	British Indian Ocean Territory (Chagos)
IRKST	Irkutsk Summer Time
IRKT	Irkutsk Time
IRT	Iran Time
IST-Israel	Israeli Standard Time
IST-India	Indian Standard Time
IST	Irish Summer Time
ITA	Italy Time
JAVT	Java Time
JAYT	Jayapura Time (Indonesia)
JST	Japan Standard Time
KDT	Korean Daylight Time
KGST	Kyrgyzstan Summer Time
KGT	Kyrgyzstan Time
KOST	Kosrae Time
KRAST	Krasnoyarsk Summer Time
KRAT	Krasnoyarsk Time
KST	Korean Standard Time
LHDT	Lord Howe Daylight Time (Australia)
LHST	Lord Howe Standard Time (Australia)
LIGT	Melbourne, Australia
LINT	Line Islands Time (Kiribati)
LKT	Lanka Time
MAGST	Magadan Summer Time
MAGT	Magadan Time
MAL	Malaysia Time
MART	Marquesas Time
MART	Mawson Time (Antarctica)
MAWT	Mountain Daylight Time (USA)
MDT-USA	Mountain Daylight Time (Mexico)
MDT-Mexico	Mexico Time
MEX	Marshall Islands Time
MHT	Myanmar Time
MMT	North Mariana Islands Time

Table 6 Time Zone Code Definition (continued)

Code	Definition
MPT	Moscow Summer Time
MSD	Moscow Time
MST	Mountain Standard Time
MT	Moluccas
MUT	Mauritius Time
MVT	Maldives Time
MYT	Malaysia Time
NCT	New Caledonia Time
NDT	Newfoundland Daylight Time
NFT	Norfolk Time (Australia)
NOR	Norway Time
NOVST	Novosibirsk Summer Time (Russia)
NOVT	Novosibirsk Time
NPT	Nepal Time
NRT	Nauru Time
NST	Newfoundland Time
NSUT	North Sumatra Time
NUT	Niue Time
NZDT	New Zealand Daylight Time
NZST	New Zealand Standard Time
OMSST	Omsk Summer Time
OMST	Omsk Time
OMST	Pacific Daylight Time (USA)
PDT-USA	Pacific Daylight Time (Canada)
PDT-Canada	Pacific Daylight Time (Canada)
PDT-Mexico	Pacific Daylight Time (Mexico)
PET	Peru Time
PETST	Petropavlovsk-Kamchatski Summer Time
PETT	Petropavlovsk-Kamchatski Time
PGT	Papua New Guinea Time
PHOT	Phoenix Islands Time (Kiribati)
PHT	Philippine Time
PKT	Pakistan Time
PMDT	Pierre & Miquelon Daylight Time
PMT	Pierre & Miquelon Standard Time
PNT	Pitcairn Time
PONT	Ponape Time (Micronesia)
PST	Pacific Standard Time
PWT	Palau Time
PYST	Paraguay Summer Time

Table 6 Time Zone Code Definition (continued)

Code	Definition
PYT	Paraguay Time
R1T	Russia Zone 1
R2T	Russia Zone 2
RET	Reunion Time
SAST	South Africa Standard Time
SBT	Solomon Islands Time
SCT	Seychelles Time
SET	Prague, Vienna Time
SGT	Singapore Standard Time
SRT	Suriname Time
SST-Sweden	Swedish Summer
SST-Samoa	Samoa Standard Time
SST-SouthSumatran	South Sumatran Time
SWT	Swedish Winter
TFT	TF Time (French Southern & Antarctic Lands), Kerguelen
THA	Thailand Standard Time
THAT	Tahiti Time
TJT	Tajikistan Time
TKT	Tokelau Time
TMT	Turkmenistan Time
TOT	Tonga Time
TRUK	Truk Time
TST	Turkish Standard Time
TVT	Tuvalu Time
ULAST	Ulan Bator Summer Time
ULAT	Ulan Bator Time
USZ1	Kaliningrad Time (Russia)
USZ1S	Kaliningrad Summer Time (Russia)
UTC	Coordinated Universal Time
UTZ	Greenland Western Standard Time
UYR	Uruguay Time
UZT	Uzbekistan Time
VET	Venezuela Time
VLAST	Vladivostok Summer Time
VLAT	Vladivostok Time
VUT	Vanuata Time
WAKT	Wake Time
WAST	West Africa Summer Time
WAT	West Africa Time
WEST	Western Europe Summer Time

Table 6 Time Zone Code Definition (continued)

Code	Definition
WET	Western Europe Time
WFT	Wallis and Futuna Time
WGST	West Greenland Summer Time
WGT	West Greenland Time
WIB	Western Indonesia Standard Time
WITA	Waktu Indonesia Tengah
WIT	Waktu Indonesia Timur
WST-Brazil	Western Brazil Standard Time
WST-Samoa	West Samoa Time
WTZ	Greenland Eastern Daylight Time
WUT	Austria Time
YAKST	Yakutsk Summer Time
YAKT	Yakutsk Time
YAPT	Yap Time (Micronesia)
YDT	Yukon Daylight Time
YEKST	Yekaterinburg Summer Time
YEKT	Yekaterinburg Time
YST	Yukon Standard Time
Z	ZULU

Timezone and DST Application Support

Below are the routine on how to implement the functions using the enhanced SDK `<time.h>`.

- Determine the current time zone. This returns the current time zone offset time.

The offset time is determined by obtaining the difference of the return value of `time()` and `gmtime()`.

- Determine the current DST status.

This returns whether DST is enabled or disabled. The `tm_isdst` flag on `struct tm` in `time.h` determines if the current time zone is on DST. This is set if DST comes into place.

- Determine the current time zone setting.

The OS provides an API that retrieves the current time zone set.

getTZdata()

Retrieves the currently set time zone.

Prototype `int getTZdata(struct tz *mytz);`

Parameters The `struct tz` returns the following 'null' terminated fields.

<code>char *tz_std(20)</code>	:	Standard time zone definition.
<code>char *offset1(15)</code>	:	Standard offset time with respect to UTC.
<code>char *tz_dst(20)</code>	:	Daylight time zone definition.
<code>char *offset2(15)</code>	:	Daylight offset time with respect to UTC.
<code>char *start_date(20)</code>	:	Start date of DST.
<code>char *end_date(20)</code>	:	End date of DST.

Return Values

0	Success
-1	Fail



If there is no DST definition on the current time zone set, the field that refers to DST information returns `NULL`.

Split RTC

The "split RTC" pertains to the storing of the local current time as the sum of two values—the one present in the RTC (Real Time Clock) hardware plus a system variable representing a delta value between the current time and the RTC time.

How the Split RTC Works

How the Split RTC Works

- OS versions with Split RTC or time zone implemented:

The OS versions with Split RTC record the delta time. It does not write to the RTC clock whenever the user calls `clk_write()`. RTC is written only once, and this is during the download of a valid `*TZ` string on OS versions with the time zone feature implemented.

NOTE



Delta time is lost during OS downgrades. Thus, the user must reset the unit's clock (via System Mode or `ddl-c`) when an OS version with time zone or Split RTC feature is downgraded to an older OS version to prevent losing the Delta time.

If the user upgrades the OS to a version with Split RTC or time zone, the clock must likewise be reset.

- Set the RTC clock once for the lifetime of the terminal:

The OS versions that implement the Time Zone feature allow the user set the RTC clock once for the lifetime of the terminal, through DDL or `download()` API using `Config.$$$` file.

When a valid `*TZ` string is downloaded the first time, the OS calculates the UTC time based on the `*TZ` information and the current local time (the current local time can be downloaded at the same command line as the `*TZ`, if not, the OS uses the values in the RTC plus delta time as the current local time). The OS then writes the UTC time to the RTC clock and sets a flag to indicate that the RTC clock is set. Once the RTC clock is set, the OS never writes to the RTC clock again; however, the OS still accepts new `*TZ` strings.

NOTE



The `*TZ` strings are not saved to the `CONFIG.SYS` file.

- Delta Time

Delta time is a system variable maintained by the OS with Split RTC or Time Zone feature. The OS updates delta time whenever the user wants to update the time (through DDL, System Mode, or API calls) and returns the sum of the RTC clock time plus the Delta time when the user retrieves the time.

read_RTC ()

Returns the time kept in the RTC clock. This is similar to `read_clock`, however, `read_RTC` requires the calling task to own the clock device. For OS versions with Time Zone, this is the UTC time.

Prototype `read_RTC (int hdl, char *yyyymmddhhmmssw);`

Parameter

<code>*yyyymmddhhmmssw</code>	The time kept in the RTC clock.
-------------------------------	---------------------------------

Return Values

Success 0

Failure: -1 and `errno` is set as follows:

- `EACCES`, invalid `yyyymmddhhmmssw` pointer.
- `EBADF`, invalid handle.

NOTE



Do not use the clock device's `_control()` function as it is used for setting the RTC clock, which is internal to the OS.

Benefits of Split RTC Architecture

For terminals using the Maxim RTC clock chip (Vx510, Vx570, and Vx610 units), the wait (which was added to overcome a hardware problem with the Maxim chip) is now eliminated from the clock-write routine.

For all Vx platforms, this allows the preservation of the RTC value independent of user updates, specially when the user writes an incorrect time to the clock.



Compiling and Linking

This chapter describes the tools used to compile and link Verix applications. The VRXCC utility is used to build Verix programs and static libraries. Based on the input parameters, VRXCC compiles, links, archives, and/or assigns the Verix executable file header information. This chapter describes the VRXCC utility. Alternatively, the ARM tools can be called directly (see the RealView manuals for details).

File Naming Conventions

VRXCC recognizes the following file extensions:

- ***.c**: C language source file. Compiled with `armcc` or `tcc`.
- ***.cpp**: C++ language source file. Compiled with `armcpp` or `tcpp`.
- ***.o**: Intermediate object file; valid only as an output file.
- ***.out**: Executable code file; valid only as an output file.
- ***.axf**: Debugger file.
- ***.a**: ARM static library file.
- ***.lib**: Shared library file.

Creating and using shared libraries is documented in [Chapter 5](#).

ARM Tools Installation

RealView Development Suite (RVDS) CDs consist of RealView Compilation Tools (RVCT), RealView Debugger (RVD), and other utilities. Multiple versions of RVCTs can be installed on the same computer. It has been observed that problems occur when multiple versions of RVDs are installed on the same computer. The application development environment may need multiple versions of RVCT but only one version of RVD is needed on the same computer.

NOTE



It is strongly recommended that before installing a new version of RVD, uninstall the existing RVD first.

The RVDS installation script sets/modifies the following environment variables on the target machine:

- **ARMROOT**: Path to the root directory of ARM installation.
- **RVDEBUG_INSTALL**: Path to RVD. The SDK Tool `vrxdb` uses this variable to find which RVD to launch.
- **RVCT20INC**: Path to the INCLUDE directory of RVCT2.0.
- **RVCT20LIB**: Path to the LIB directory of RVCT2.0.

- RVCT22BIN: Path to the armcc.exe of RVCT2.2.
- RVCT22INC: Path to the INCLUDE directory of RVCT2.2.
- RVCT22LIB: Path to the LIB directory of RVCT2.2.
- PATH: Pre-pended with RVCT and RVD executable directories.
- ARMLMD_LICENSE_FILE: Path to the license file, set by the license wizard or by the user.
- ARMHOME: Path to the directory that has the CONFIG folder is same as ARMROOT. If the license wizard does not create this, the user needs to create this manually.

Environment Variables

VRXCC uses the following environment variables:

- **VRXSDK** - Root directory of Verix V Software development kit. This variable is set during the installation of the Verix V Development Toolkit (VVDTK).
- **RVCTDIR** - Directory containing the RVCT tools (only in SDK version 1.4 and later), for example:

```
set RVCTDIR=  
C:\Program Files\ARM\RVCT\Programs\2.2\503\win_32-pentium
```

Locating the Tools

VRXCC searches the ARM compilation tools in the directories that are listed in the PATH environment variable. If there are multiple versions of ARM compilation tools installed, the first in the PATH will be picked. Use the `-v` option of `vrxcc` to see which version of ARM compilation tools is used.

The RVDS installation script automatically pre-pends the PATH environment variable with the ARM tools executable directories. With typical installations, VRXCC locates the latest installed ARM tools through the PATH environment variable. If multiple versions of ARM tools are installed on the same computer, RVCTDIR can be used to pick the version of ARM tools to use.

For example, the following line sets RVCTDIR to use RVDS2.2:

```
set RVCTDIR=  
C:\Program Files\ARM\RVCT\Programs\2.2\503\win_32-pentium
```

The following line sets RVCTDIR to use RVDS2.0:

```
set RVCTDIR=  
C:\Program Files\ARM\RVCT\Programs\2.0.1\277\win_32-  
pentium
```

Use the `-v` option of `vrxcc` to see which version of ARM compilation tools is used. The help page of `vrxcc` also shows which ARM tool is used:

```
vrxcc-Compile and link Verix/ARM applications (Version 1.1,
Build 12)
```

```
Using RVDS2.0.1 or earlier version
```

Or:

```
vrxcc - Compile and link Verix/ARM applications (Version
1.1, Build 12)
```

```
Using RVDS2.2 or later version
```

The Verix V SDK tools are located using the `VRXSDK` environment variable.

Creating Libraries

If a library name is specified as the output file (for example, `-o utils.lib`), then the object files are placed in the library rather than linked. This may involve creating a new library or updating an existing one, depending on the options used.

Options

VRXCC recognizes common compiler and linker options and passes them to the appropriate tools. For example, `-I` options are passed to the compiler, while `-map` are passed to the linker. Some options, such as `-g`, affect both compiling and linking. Any option can be passed to the tools using `-armcc`, `-armlink`, and so on. Options specified this way are not checked for legality or compatibility with Verix.

General Options

- `-help`: Displays a usage summary and exit. The following synonym can also be used:
 - `-?`
- `-F file`: Reads additional arguments from the named file.
- `-v[v]`: Verbose mode.
- `-v`: Displays the ARM tool commands as they are invoked.
- `-vv`: adds additional progress and status messages.
- `-n`: Does not run the tools, just shows the commands that run. Implies `-v`.
- `-c`: Compile only; do not link.
- `-o file`: Output filename.
 - If the build includes a link, this is the final `.out` file.
 - If compiling only (`-c` option), it is the object file.
 - If file specifies a directory, listing and map files and the `.axf` file are placed in the same directory.

- `-e file`: Redirect error output (`stderr`) from invoked tools to the named file. A file name of `-` redirects `stderr` to `stdout`. `-e` does not redirect error messages generated by VRXCC itself.
- `-k`: Keep temporary files instead of deleting them at the end of a build. Object files are not considered temporary and are never automatically deleted. The `.axf` file is in the same directory as the `.out` file; other temporary files are generated in the current working directory. The following temporary files can be generated:
 - `<name>.axf`: ELF format linker output file, prior to conversion to Verix executable format. This file is used for debugging and is automatically kept when using the `-g` option.
 - `_vrxcc.via`: Linker via file containing list of files to be linked. `vxacc` generates this if there are many files to link.
 - `_vrxcc.sct`: Linker scatter file.

Compile Options

VRXCC always passes the following options to the C and C++ compilers:

```
--cpu ARM920T --bigend -J<vrxsdk>\include
```

In addition, starting from Verix V SDK version 1.4, the `Ono_fp_formats` and `Ono_memcpy` options are passed to the RVCT compiler to disable the ARM built-in optimization for the `printf` and `memcpy` family of functions. For Verix V SDK version 1.3 and earlier, the `--apcs/adsabi` and `-apcs/shlclient` options are passed to the RVCT compiler to request code that is compatible with ADS.

VRXCC does not distinguish between C and C++ options. Therefore, it is not recommended to compile C and C++ source files in the same VRXCC command. See the compiler documentation for details on the options passed to the compiler.

- `-a`: Generate ARM code. Thumb is the default.
- `-cpp`: Use the C++ compiler for `.c` files. It is used automatically for `.cpp` files.
- `-inter`: Enable ARM-Thumb inter-working. This is automatically enabled by `-shlabs` and `-shlpic`. Passes `--apcs/inter` to the compiler.
- `-I dir`: Add `dir` to the include file search path. Passed to the compiler.
- `-D name[=value]`: Define a macro. Passed to the compiler.
- `-U name`: Undefine a macro. Passed to the compiler.
- `-g`: Generates debugging information. Passed to the compiler. Note that this also disables full optimization unless you provide an explicit `-O` option.
- `-shlabs`: Compile code with the appropriate options for use in an absolute shared library. Passes `-apcs=/inter` to the ARM compiler RVCT version 2.2 and later and Passes `--apcs/rwpi/shl/inter` to the ARM compiler RVCT version 2.0.1 and earlier.

- `-shlpic`: Compile code with the appropriate options for use in a position-independent shared library. Passes `-apcs=/fPIC/inter` to the ARM compiler RVCT version 2.2 and later and Passes `--apcs/ropi/rwpi/shl/inter` to the ARM compiler version RVCT2.0.1 and earlier.
- `-O[parm]`: Optimization options. Passed to the compiler.
- `-armcc,opt[,opt...]`: Pass the specified options to `armcc`. Commas are replaced by spaces. If commas are part of the `opt`, use the escape character `\` to pass in commas (e.g. `vrxcc -armcc, "--diag_suppress 167\,550"` to pass in `-diag_suppress 167,550`). `vrxcc` does not check the options for legality or compatibility with Verix.

Assembler Options

VRXCC always passes the following options to the assembler:

`-cpu ARM920T -bigend`

Note that the `-a` option, or the lack of it, has no effect on assembler code. Thumb code should be marked with a `CODE16` directive in the source file. See the assembler documentation for details of command line options passed through to the assembler.

- `-inter`: Assert code is compatible with ARM-Thumb inter-working. Passes `-apcs/inter` to the assembler.
- `-g`: Generate debugging information. Passed to the assembler.
- `-i dir`: Add `dir` to the include file search path. Passed to the assembler.
- `-list`: Generate a list in `intputfile.lst`. Note that a different filename cannot be specified. To do that or specify listing control options, use `-armasm -list,filename,...`.
- `-armasm,opt[,opt...]`: Pass the specified options to `armasm`. Commas are replaced by spaces. VRXCC does not check the options for legality or compatibility with Verix.

Link Options

Generally, the only link options appropriate to specify are those that control debug information (`-g`) or request extra output (`-map`, `-info`, and so on). See the linker documentation for details of options passed directly to the assembler.

- `-a`: Main program is ARM (rather than Thumb) code. Note that applications can mix ARM and Thumb code. This option is required if—and only if—the main function is ARM code.
- `-g`: Include debugging information in the linker output (`.axf`) file and do not delete the `.axf` file after the `.out` file is generated. `-g` does not affect the `.out` file, but note that it may affect compiler optimization.
- `-verbose`: Output a verbose description of the link process. This is written to the `stderr` stream. Use `-e` to redirect it to a file. Passed to linker.
- `-map`: Output load map to file `name.map`, where `name` is the base filename of the `.out` file being generated. Passes `-map -list name.map` to the linker.

- `-symbols`: Output symbol list to file `name.map`. The output from `-map`, `-symbol`, and `-xref` is written to the same file.
 - `-xref`: Generate symbol cross-reference `name.map`. The output from `-map`, `-symbol`, and `-xref` is written to the same file.
- `-armlink, opt[, opt...]`: Pass the specified options to `armlink`. Commas are replaced by spaces. The commas avoid ambiguity between options belonging to `-armcc` and options for VRXCC. VRXCC does not check the options for legality or compatibility with Verix.

Header Options

VRXCC supports the following subset of the VRXHDR options. In addition, if `-g` has been specified, it passes the `-d` option to enable debugging of the executable file.

- `-s size`: Set stack size to `size`. Verix allocates the stack in 1024 byte increments. If `size` is not a multiple of 1024, it is incremented to the next multiple of 1024.
- `-h size`: Set heap size to `size`.
- `-vrxhdr, opt[, opt...]`: Pass the specified options to VRXHDR. Commas are replaced by spaces. VRXCC does not check the options for legality.

Library Options

VRXCC supports only basic library creation and update. The `-r` option is always passed to `armar`. Use `armar` directly to extract, list, and so on, library objects.

- `-create`: Create a new library, replacing any existing file. Passed directly to `armar`.
- `-u`: Replace object in library only if the given file is newer. Passed directly to `armar`.
- `-armar, opt[, opt...]`: Pass the specified options to `armar`. Commas are replaced by spaces. VRXCC does not check the options for legality or compatibility with Verix.



DDL – Direct Download Utility

The three download methods for Verix-based Omni series terminals are *direct*, *remote*, and *back-to-back*. This chapter explains how to perform direct downloads using the Direct Download utility (DDL). DDL is part of the VDTK. Direct downloads use a cable between the development PC and the terminal. See Download Operations in the Verix V Operating System Programmers Manual, VPN 23230 or the reference manual of your terminal for information on download procedures, error messages, file authentication, king with compressed files, and download result messages.

Direct Downloads with DDL

Direct downloads use a cable RS-232 communications link between the development PC and the target terminal. Starting from Verix V SDK version 1.5, DDL also supports direct download through USB cable connection between the development PC and the target terminal. For more information, see [Installing the Verix USB Client RS232 Driver](#). DDL (DDL.EXE) controls the download. Downloads can be initiated from the DOS prompt on the development PC. The [Invoke DDL](#) section describes the download process using DDL.

DDL supports full or partial downloads and CONFIG.SYS file manipulation. The date, time, and system passwords can be set through the download.

Invoke DDL

DDL downloads files from a PC to a Verix-based Omni series terminal and allows the clock, password, and CONFIG.SYS variables to be set. DDL is invoked from a command line as:

```
DDL -p port -b baud -t timeout -d file -e file -i file  
      -c [offset] -x password -f file file -r files -z . . .  
      key=value. . .
```

All arguments and flags are optional. Flag letters (for example, -b) are not case sensitive.

Arguments that contain embedded spaces must be placed inside quotation marks. Spaces are allowed in configuration setting values and password strings.

A download operation can be interrupted by pressing Ctrl-C or Ctrl-Break on the keyboard. The state of the terminal may not be well defined following an interrupted download.

Communications Options

The following options set the communications parameters.

- `-p port` Host port number. 0 means download through USB; default = 1 (COM1).
- `-b baud` Baud rate; acceptable values are 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200; default = 115200; ignored when port is 0.
- `-t timeout` Communications time-out, in seconds; default = 1.

File Download Options

The following options set download parameters.

- `-d file` Downloads file as data file.
- `-e file` Downloads file as executable (code) file.
- `-i file` Downloads file as code if it has a `.out` or `.LIB` suffix; otherwise download as data (same as filename alone).

Filenames not preceded with the `-d`, `-e`, or `-i` option flags are treated as if they were prefixed with `-i`. Directory and drive information is removed from the filename sent to the terminal. Note that for readability, options can include space before the value (for example, `-f ddlargs` instead of `-fddlargs`).

NOTE



The `-d`, `-e`, and `-i` options are provided primarily for backward compatibility. They are rarely needed in practice because the default is most always correct.

Removing Files

The `-r` option deletes specific files or groups of files during a download. Its syntax is:

`-r [drive:][group/][file]` drive is "I" (RAM), "F" (FLASH), or "*" (both). group can be GID1–15, "*" to designate all groups, "." to designate the current group, or empty (implying GID15).

file is a filename. If omitted, all files in the specified drive(s) and group(s) are deleted.

The following are code line examples:

- `-r foo.out` deletes `foo.out` from the current drive and group.
- `-r F:2/foo.out` deletes `foo.out` from GID2 in flash.
- `-r /foo.out` deletes `foo.out` from GID15 on current drive.
- `-r ./` deletes all files from current group and drive.
- `-r F:2/` deletes all GID2 flash files.
- `-r *:*.` deletes all files in all groups.

The position of `-r` arguments in the DDL parameter list is significant. For example,

```
ddl SETDRIVE.F newapp.out -r oldapp.out
```

first downloads `newapp.out`, then deletes `oldapp.out`.

In case of an insufficient memory, the order of the arguments can be reversed to first delete the old file.

It is not an error if a specified file does not exist. The usual restrictions on group access apply; in particular, wild cards specify that all groups can be used only in GID1 downloads.

Miscellaneous Options

The following options set date, time and password parameters, and arguments from a referenced file.

- `-c [offset]` Sets terminal date and time to host clock values, optionally offset by the specified number of hours (–23 to +23). The clock does not reset until all files are successfully downloaded.
- `-x password` Sets file group password. If files are downloaded to more than one group, the password for the last specified group is changed, regardless of the position of the `-x` option on the command line. The password is not changed until all files are successfully downloaded. Therefore, if the download fails for any reason, the previous password remains in effect.

File group passwords can also be changed with the configuration setting “*PW=password.” This takes effect immediately, setting the password for the current group. Using *PW is easier and safer than using `-x`. Note that *PW is not actually stored in `CONFIG.SYS`.
- `-f file` Read more options and arguments from file. It can contain multiple lines, with multiple arguments on each line.

Configuration Settings

Arguments of the form `key=value` set entries in `CONFIG.SYS`. If `value` contains spaces, it must be enclosed in quotation marks.

Environment Variable

The environment variable `DDL` can specify additional options (see [Direct Downloads with DDL](#)). If an option is specified in both the environment variable and on the command line, the command line takes precedence.

Examples

```
DDL test.out *go=test.out test.p7s
```

Downloads the files `test.out` and `test.p7s`, and sets the *GO configuration variable so that these files execute on terminal reset. The default port (COM1) and default baud rate (19200) are used.

```
DDL -p 2 -b 115200 test.out -c *go=test.out test.p7s
```

Similar to previous example, except downloads over port 2 at 115200 baud and sets terminal clock to host clock.

```
DDL test2.out alpha.dat beta.dat *go=test2.out test2.p7s
*arg="ALPHA BETA"
```

Downloads an executable file, the signed file, and two data files. Also, sets the *ARG variable so that the program receives ALPHA and BETA in its `argv` array.

```
DDL -f ddlargs
```

Retrieves arguments from `ddlargs` file.

File Authentication and DDL

Download the signature (.p7s) file and the file it signs. The signature file must always be loaded into RAM, even if the file it signs is in flash. Both files *must* be loaded to the same file group.

NOTE



If a file fails authentication, a message displays. A key must be pressed to clear the message.

If you use a zip file to download, include the signature files in it. It is not necessary to sign the zip file.

Examples

1 Download to RAM:

```
ddl hello.out hello.p7s *go=hello.out
```

2 Download a flash file, see [Direct Downloads with DDL](#):

```
ddl SETDRIVE.F hello.out SETDRIVE.I hello.p7s *go=f:hello.out
```

or, more briefly, since RAM is the default:

```
ddl hello.p7s SETDRIVE.F hello.out *go=f:hello.out
```

3 Download to a different file group:

```
ddl SETGROUP.3 hello.p7s SETDRIVE.F hello.out
```

Actual authentication is done on terminal restart. Note that the following messages appear on screen during the authentication process:

```
**** Verifying Files ****
```

```
System Certificate
```

```
K2PART.CRT
```

```
** Authentic **
```

Initial loading of system certificates occurs. This screen appears for approximately one second after authentication completes.

```
**** Verifying Files ****
```

```
Check Certificate
```

```
OWNR.CRT
```

```
** Authentic **
```

All other certificates are processed with this message scheme. If the authentication does not succeed, the bottom line reads — FAILED —.

```
**** Verifying Files ****
```

```
Compare Signature
```

```
MYFILE.P7S
```

```
MYFILE.OUT
```

```
** Authentic **
```

When signature files are processed, the name of the target file also appears. As with certificates, upon successful verification, this text line appears for approximately one second. Failure is indicated by displaying — FAILED — on the screen.

NOTE



If the file signature verification fails, a message indicating the failure is displayed. The file will not be executed. Press any key to clear the message.



Creating and Using Libraries

The built-in system library contains standard C library (functions such as `strlen`, `printf`, and so on) and the Verix API (application program interface) library (functions such as `read()`, `SVC_WAIT`, and so on). Support for specialized terminal features such as, smart cards and security modules, is provided in additional libraries, some of which are included in the VVDTK. Some applications may require special versions. (Special versions can be obtained through your VeriFone representative.)

Application programs use library functions by including the appropriate header files and linking with the library files. Most of the API function calls are declared in the header file, `svc.h`.

NOTE

C++ interfaces cannot be exported from shared libraries. C++ can be used to implement a library but the exported functions must be extern "C."

In addition to using the supplied system libraries, application programmers can create *libraries*. This is a good way to organize groups of common functions, especially if they are used by more than one application.

Verix *shared* libraries are stored in `.LIB` files which are downloaded to the terminal in the same manner as program `.OUT` files (including the file authentication requirement). Applications link with an interface stub file that contains the code required to find and call the library functions at run time.

On ARM platforms, a maximum of 32 shared libraries can be used by a single task. One of these is the system library that contains operating system functions and the standard C library. All programs automatically use this library. The remaining 31 libraries can be defined by the application programmer.

Creating a shared library involves compiling its code with some special options, preparing a description file, and running the VRXLIB tool. The description file assigns an identification number to the library and to each function. If a library or function number changes, applications that call the library must be relinked. This cancels out many of the benefits of using shared libraries and should be avoided.

This chapter describes how to create and use application libraries in both *linked* and *shared* formats.

Differences Between Verix ARM and Verix 68K Shared Libraries

This section provides programmers familiar with the Verix 68K programming environment a quick list of differences in the Verix ARM programming environment.

- Verix ARM supports 31 user shared libraries, while Verix 68K platforms only supports 2.
- There are two kinds of Verix ARM library: absolute and position-independent. Verix 68K libraries are similar to ARM absolute libraries.
- The run-time location of Verix 68K libraries are determined by the library ID. The Verix ARM library developer must specify addresses for absolute libraries and ensure that libraries used by a single task do not conflict.
- Verix 68K libraries have a 32KB limit on their static data size; Verix ARM libraries do not have this limitation.
- A main function is required for Verix 68K libraries, but optional in Verix ARM. In the Verix 68K, task-startup is aborted if a library main returns a non-zero value. In Verix ARM, the return value is ignored (that is, it can call `_exit` to abort).

Linked vs. Shared Libraries

The main advantages of shared libraries are:

- Reduced memory requirements, since multiple tasks can share the same library code file.
- Ability to update libraries without rebuilding applications.
- Ability to customize applications by running them with different library versions.

However, there are trade-offs:

- The memory space advantage of shared libraries obviously depends on the assumption that there are multiple tasks to actually share them. This may or may not be true in a particular environment. Further, all shared library functions are present in memory whether any application uses them. With a linked library, only functions actually called are included in the executable. A shared library that contains unused functions could actually increase memory usage.
- The ability to update a library independently of applications may not always be an advantage. With linked-in libraries, applications are self-contained.
- There is a modest amount of memory and execution time overhead for shared libraries.
- Shared libraries are subject to a few additional programming restrictions, as detailed below.
- The shared library number/function number scheme and memory location requires planning and coordination.

Shared and linkable libraries are not mutually exclusive. The Verix system library for example, is implemented as a mixture of linked-in and shared functions. Infrequently used functions (such as math routines) are linked-in if needed. Common functions (such as the string utilities) are shared. The `LIBSYS.A` library, where applications are linked, contains the actual code for the linked-in functions plus the interface code for the shared portion. It is possible to make the same library available in both linkable and shared forms.

Absolute vs. Position- Independent

Verix ARM supports two types of shared libraries: *absolute* (ABS) and *position-independent* (PIC). Absolute libraries are loaded at a predefined memory address. Position-independent libraries can be loaded anywhere, but this flexibility comes at the cost of higher overhead and some additional restrictions.

Information on assigning the location of an absolute shared library is presented in the [Shared Libraries](#), [Programming Considerations](#), and [Library Placement](#) sections.

The pros and cons of absolute and position-independent libraries are summarized in [Table 7](#).

Table 7 Absolute vs. Position-Independent Libraries

Absolute Libraries		Position-Independent Libraries	
–	The developer must reserve the address range.	+	The developer need not worry about addresses.
+	Can have initialized static data (except for pointers to data).	–	No initialized static data.
+	Smaller.	–	Larger. (Typically 16 extra bytes per function, plus about 0.5% code size increase due to position-independence).
+	Less function call overhead.	–	More function call overhead (typically four instructions).

ARM vs. Thumb

Shared library functions can be compiled to ARM or Thumb object code. The usual speed vs. the space trade-offs apply. A library can contain both types of functions. Nothing special (other than recompiling) must be done to change a mode of a function.

A library function can be called from ARM or Thumb application code, regardless of the mode of the function itself. However, the call mechanism can be optimized for ARM callers or for Thumb callers. The variant is specified when the library is built. If library callers are known to be predominantly Thumb, then Thumb optimization slightly reduces memory requirements and call overhead. However, the penalty for calling a Thumb-optimized library from ARM code is larger than the opposite case. So, if the caller mode is unknown or if both modes will be commonly used, the ARM-friendly variant is recommended. Both versions work correctly with either type of caller—this is strictly an optimization issue.

Archive Libraries

Conventional libraries (often called *archive* or *static* libraries) contain object code copied into the application executable file.

Names

The conventional form of an archive library name is `LIB<name>.A` where, `<name>` is replaced by a short descriptive name. For example, the Integrated Chip Card library is named `LIBICC.A`. Library names are not required to use this form, but the linker allows names to be abbreviated for convenience.

Creating Archive Libraries

The RealView `armar` utility can create and update library files. To create a library you would first compile all source files, then combine the object files into a library with a command such as:

```
armar -create libprint.a *.o
```

where, `-create` directs `armar` to create a new library (`libprint.a`) instead of updating an existing one. The `*.o` wild card specifies that all object files in the current directory be added to the library.

Listing Contents

To list the contents of a library use a command such as:

```
armar -t libprint.a
```

The `armar` utility has many options to manage library file contents (type `armar -help` for a summary or refer to the RealView documentation for a complete description).

Shared Libraries

Verix also supports shared (also called *dynamic*) libraries, in which the object code is in a separate file from the application. Applications connect to shared library functions at run time rather than when the program is linked. The concept is similar to a Dynamic Link Library (DLL) in Microsoft Windows.

Programming Considerations

For the most part, designing and coding a shared library is similar to an ordinary library, however, the following sections discuss some additional restrictions and special considerations.

Library Placement

The developer of an absolute library must specify its memory location. Libraries used by a task must not overlap each other or the main program. (Libraries in different tasks do not conflict.) Assigning addresses requires some familiarity with the run-time layout of Verix applications.

Figure 3 shows the layout of the virtual memory seen by each task.

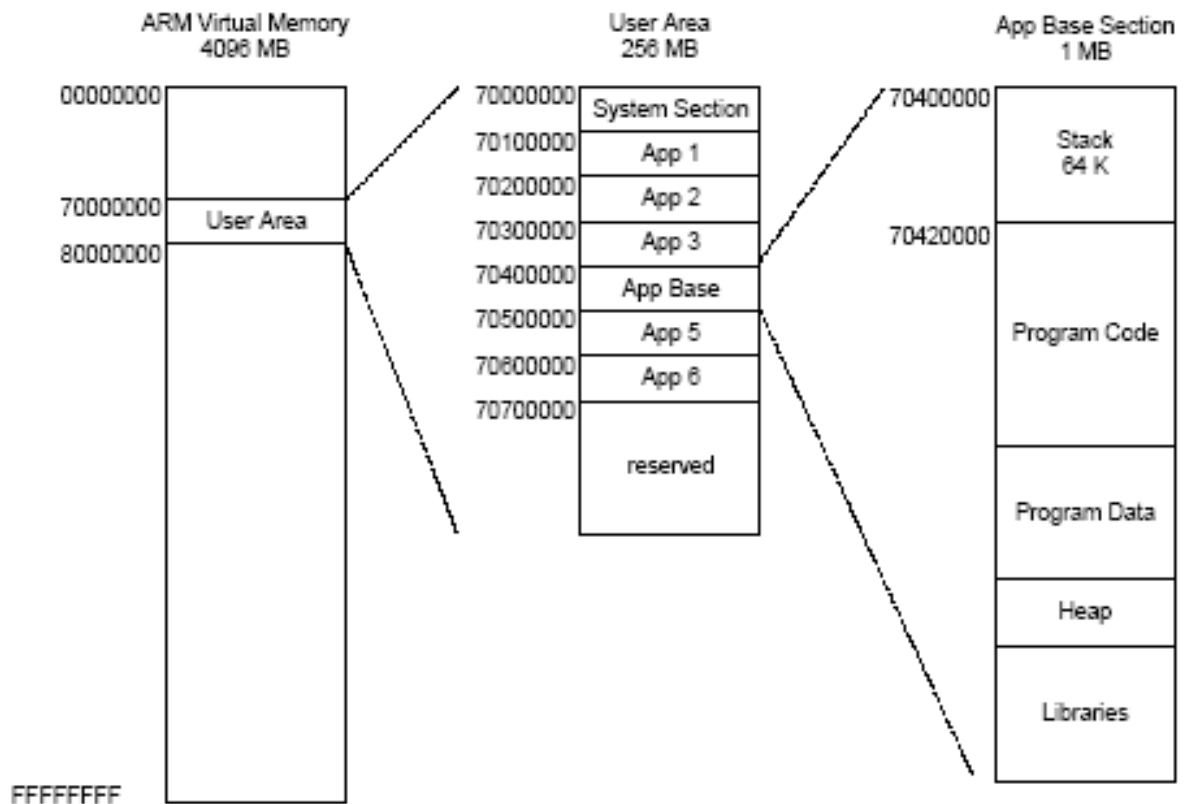


Figure 3 Virtual Memory

The 256-MB region from address 70000000 through 7FFFFFFF is reserved for user-space code and data. Currently, Verix permits applications to only use the first seven megabytes. In ARM terminology, each 1-MB block is known as a *section*. The first section, at 70000000, is reserved for system use. The main application program is loaded 128 KB from the start of the application base section, at address 70420000. A very large application could spill over into the next two megabytes (App 5 and App 6) assuming enough physical memory is available. The stack precedes the code, ending at address 70420000. If the stack is bigger than 128 KB, it can spill over into the preceding section (App 3).

The space available for libraries is from 70100000 to the start of the stack, and from the end of the main program's heap through 706FFFFFF. Of course a library developer may not know how big the client applications will be. It is tempting to put libraries in the App 1, 2, or 6 sections that are extremely unlikely to be required by a main program. However, each section used by a task requires an OS page table, which occupies 4 KB of RAM. The most efficient use of RAM occurs if all libraries used by a task can be loaded in the application base section along with the main program.

Many applications do not require more than approximately 10 KB of stack space, so about 100 KB is available at the start of the base section (70400000). If the application code and data occupy say, 500 KB, then another 400 KB will be available at the end of the section (starting at 7049C000). Of course there may be multiple libraries competing for the space.

To avoid this whole issue, use position-independent libraries, which Verix can place at any convenient location. If space is available in the base section, Verix puts them there to avoid any requirement for additional page tables.

Global Data

Each application using a shared library has its own copy of the library data, just as it would if the library were linked in. (Shared libraries do not, therefore, provide a means for tasks to share data.)

Position-independent libraries cannot have initialized static data. For example,

```
int seed = 1;
```

is not allowed as a global variable.

Absolute libraries can have initialized static data with one exception—pointer variables cannot be statically initialized to the address of data (because the address is not known at link time). For example:

```
char buffer[128];  
char *buf_ptr = buffer ERROR - address of buffer is unknown
```

Pointers can be initialized to the address of constant data, including strings. The following is acceptable:

```
char *name = "Waldo";
```

Note that the above restrictions affect only global variables; there are no restrictions on local data initialized at run-time. The library main function provides an alternate way to initialize data.

Global variables defined in a shared library cannot be directly accessed by library users. The library must provide functions to access such data. If necessary, a library function could return a pointer to a global variable, giving callers direct access through the pointer.

Similarly, library functions cannot directly access global variables defined by the main application.

Function Calls from Libraries

Most stdio library functions cannot be called from a shared library. The exceptions are `sprintf`, `vsprintf`, `sscanf`, `rename`, and `remove`. Calls to other functions results in a link error of the form “name cannot be assigned to PI Exec region.” However, when using RVDS4.0, `stdio.h` functions can be used. See `readme.txt` file in the SDK for detailed instructions.

Callbacks

Shared library code cannot directly call functions defined in application code. They can, however, call application functions using function pointers (*callbacks*) passed to the library.

main() Function

A shared library can have a `main()` function that is automatically called at application launch, before the application `main()` executes. This can be used to perform initialization. It has the usual prototype:

```
int main (int argc, char **argv);
```

`argc` and `argv` are the same arguments passed to the application. The `main()` return value is ignored.

When more than one shared library is used, the order in which their `main()` functions are called is not defined. Therefore, it is unsafe for the `main()` function of one library to call another function in a shared library, since the other library may not have been initialized. (This does not apply to the system library; it can always be safely called.)

C++

C++ interfaces cannot be exported from shared libraries. C++ can be used to implement a library, but the exported functions must be extern "C". This means that shared libraries do not support class libraries or overloaded functions.

In absolute libraries, C++ can be used for internal implementation, with one restriction—static object destructors are not called when the program exits (libraries do not know when the program exits).

NOTE



RV 2.0.1 Compiler Advisory: In position independent libraries built with RV 2.0.1 compiler, C++ is much more restricted. Features such as virtual functions, pointers to members, and run-time type identification depend on statically-initialized pointers, which are not allowed in PIC code. It is recommended that C++ code in PIC libraries be limited to "a better C," not use classes.

These limitations do not apply to position independent libraries built with the RV 2.2, or later, compiler.

_SHARED_LIB Macro

When a code is compiled for a shared library, the macro `_SHARED_LIB` must be defined prior to including any standard header files. This can be done with the compiler option `-D_SHARED_LIB`. Application programs can test this macro in code that needs to be compiled differently depending on whether or not it is part of a shared library. VRXCC defines this automatically when the `-shlibs` or `-shlpic` option is used (see [Compiling and Linking](#)).

Version Numbers

A version number in the form *n.m* is specified when a shared library is created and is recorded in both the library file and the interface file linked with the application.

In the `.LIB` file, the version is part of the file header and works just like a program version number. That is, it can be inspected and changed by the `VRXHDR` utility and library functions can read it through the global variable `_version`.

Application code may be concerned with two library version numbers: the version of the interface with which the application was linked, and the version of the library actually used at run time. The interface version is available as a global variable:

```
extern const unsigned short _LIBNAME_VERSION;
```

where, `LIBNAME` is the library name (uppercase).

The version of the library used at run time is not directly readable by the application. If it is needed, the library must provide a function to return it.

Example

```
unsigned short mylib_version (void) { return _version; }
```

NOTE



If the application references `_version` directly, it retrieves its own version number, not version of the library.

`<stdlib.h>` contains the following declarations for the shared system library:

```
extern const unsigned short _SYS_VERSION;
```

Returns the interface version.

```
unsigned short _syslib_version (void);
```

Returns the library version.

It is recommended that application libraries include similar declarations in their header files. A program can check that the run-time library matches the interface with which it was built using code similar to the following:

```
if (_syslib_version() != _SYS_VERSION) error(...)
```

Intra-Library Calls

Shared libraries can call other shared libraries. However when a program is loaded for execution, Verix does not “deep search” for libraries called only from other libraries. Therefore, the main program must reference all the shared libraries it requires, whether it calls them directly or not. One way to reference a library without calling any of its functions is to use its interface version constant.

For example, suppose a library used by a program needs the library `UTIL.LIB`, but the program does not call it itself. Somewhere in the program (for example, at the start of `main()`) include the following code

```
extern volatile unsigned short _UTIL_VERSION;
(void)_UTIL_VERSION;
```

The `volatile` parameter prevents the compiler from optimizing away the seemingly unused reference. The `(void)` cast quiets a compiler warning.

Dynamic Loading

Dynamic loading of libraries is implemented using the system call:

```
int load_named_DLL(char * dllFileName);
```

The file name specifies a previously authenticated code file formatted as a shared library similar to that of the existing shared libraries. If the call succeeds, the caller receives a “handle” to the newly-linked shared library. Otherwise, a return code of `-1` plus the value in `errno` indicates the reason for failure – lack of memory, file not authenticated, or file not found.

The Verix V SDK builds shared libraries with an interface section (dispatch table) near the beginning of the library. Each public function has an interface routine. Access to function `N` consists of branching to the corresponding interface routine; the interface routine then branches to the code for routine `N`. The “handle” returned by the `load_named_DLL()` function is the address of the interface routine for the first public function.

Each published routine in a shared library has an associated number indicating its relative position in the dispatch table. The address of the routine `N` in the DLL with handle `H` is provided by the macro:

```
DLL_function_address(H, N)
```

This is defined in the `SVC.H` header file, `errno` is not affected.

Variable Library Specifier

Each library name is specified at the time the program is linked and recorded in the program header. Any library name beginning with the “@” character is treated as an indirect reference rather than the actual name of the library. The OS loader searches the `CONFIG.SYS` file to find the variable and its contents, and then indicate the actual name of the library. For instance, at build time, the library name is specified as `@COMLIB`. During deployment, the user sets the `CONFIG.SYS` variable, `COMLIB`, to `F:SERIAL.LIB` at download. When the program is run, the OS detects the “@” prefix, searches the `CONFIG.SYS`, and links to the actual library `F:SERIAL.LIB` if present.

When swapping between shared libraries at program load time, the shared libraries must have the same ID, function names, and numbers exported in the `.lid` file. If the shared libraries are absolute, they must specify the same address in the `.lid` file.

Weak Library Specifier

Any library name prefixed with “?” is treated as optional. The program still runs even if the library name (excluding the “?” prefix) is missing at run time.

WARNING



Any runtime reference to this library results in a program failure.

Default Library Extension

The user does not need to specify the full name of each shared library when creating the program header. The loader allows the omission of the `.LIB` extension from the library names in the program header.

Creating a Shared Library

The steps for creating a shared library are as follows:

1 Name the library.

Various suffixes are applied to this name when files are generated. Library file names on the terminal are limited to 12 characters including the `.LIB` suffix and any necessary drive prefix such as `F:`. Therefore, the maximum length of a library name is 8 characters if no drive prefix is necessary, and 6 characters if a prefix is required. To avoid file name conflicts, the name should be different from any of the component source files.

2 Assign a library ID number.

On ARM platforms, a maximum of 32 shared libraries can be used by a single task. One of these is the system library that contains operating system functions and the standard C library. All programs automatically use this library. The remaining 31 libraries can be defined by the application programmer and must be assigned an ID number.

3 Code and debug library functions.

It is recommended to test and debug library functions by linking them with the application or a test driver, and converting to a shared library after you are confident the code is working in the simpler environment. Debugging code in shared libraries is more difficult than debugging linked-in code.

4 Compile the library source code with shared library options.

Compile the library source code with the following options in addition to the standard program compile options:

<code>--apcs/shl/rwpi/inter</code>	absolute library using RVCT2.0.1 and earlier versions.
<code>--apcs=/inter</code>	absolute library using RVCT2.2 and later versions.
<code>--apcs/shl/ropi/rwpi/inter</code>	position-independent library using RVCT2.0.1 and earlier versions.

```
--apcs=/fPIC/inter      position-independent library using
                        RVCT2.2 and later versions.

-D_SHARED_LIB           all libraries
```

If using the VRXCC tool, the `-shlibs` and `-shlpic` options automatically pass the appropriate options to the compiler (see [Compiling and Linking](#)).

1 Create a library archive.

Use the `armar` tool to create an archive file containing all the library object files. This should be named `LIBNAME.A`, where `LIBNAME` is the library name, as shown in the following example:

```
> armar -create -r calendar.a *.o
```

If using the VRXCC tool, you can combine the compile with the archive creation as shown in the following example:

```
> vrxcc -shlpic -o calendar.a dayofweek.c isleapyear.c ...
```

2 Create a library description file.

The library description file specifies the library name, number, and version, and lists the functions exported by the library. It must be named `LIBNAME.LID`, where `LIBNAME` is the name of the library.

Example

```
# Description file for shared library CALENDAR
CALENDAR id=5 addr=70a80000 ver=1.3 thumb
1 day_of_week
2 is_leap_year
4 julian_day
...
```

`#` begins comments. (Blank lines and comment lines can be used anywhere. Comments can also follow other text on a line.)

The first non-comment line contains mandatory two fields:

<code>name</code>	Maximum 8 characters (case-insensitive), <code>CALENDAR</code> in the previous example.
<code>id</code>	Library ID number (1–31).

The name field must be first; all other fields can appear in any order. Field names are case sensitive.

The following are optional fields:

<code>version</code>	<code>n.m</code> format. If this field is omitted, it is set to <code>0.0</code> . The keyword can be abbreviated to <code>ver</code> .
<code>address</code>	Library load address, in hex. Zero (default) denotes a position-independent library. The keyword can be abbreviated to <code>addr</code> .
<code>size</code>	Maximum library size. Causes the VRXLIB tool to report an error if the library file exceeds this setting. The library file is still generated and is viable. This can be useful to ensure that an absolute library does not overflow its allotted space.
<code>thumb</code>	Optimize the library for Thumb callers. This does not prevent it from being called from ARM code – it is just less efficient. The keyword <code>arm</code> is also accepted, but since it is the default it has no effect.

The remaining lines list the names of each library function and assigns a function number to each. Function numbers start with 1 (not 0). Functions must be listed in increasing numerical order. Gaps in the numeric sequence are permitted. Since changing function numbers is undesirable, it may be convenient to organize the functions in groups, with a few unused numbers at the end of each group for future expansion. Place holder lines with a number and no function name are also allowed. Unused function numbers occupy space in the library so large gaps in function numbers should be avoided.

In a PIC library, functions with numbers >255 incur slightly more overhead. If creating a large PIC library, assign large function numbers to less frequently called functions. The difference is small compared to the difference between absolute and PIC libraries.

If function 0 is specified it is used as a default for unused function numbers. An unused function number could potentially be called as a result of version mismatch between an application and library. Normally this will cause a program abort. Specifying a default function to be called in such cases can allow the application to fail gracefully. Note however that this works only for function numbers included in the descriptor file, not those past the end of the range. The default function cannot be called directly (unless it is also specified as a callable function elsewhere.) In the following example function unimplemented will be called for function numbers 3, 5, and 6:

Example

```
CALENDAR id=5 addr=70a80000 ver=1.3 thumb
# The function "unimplemented" will be called if
# the application attempts to call unused function
# numbers 3, 5 or 6.
0 unimplemented
1 day_of_week
2 is_leap_year
4 julian_day
# The following line inserts unused function
# entries for function numbers 5 and 6.
6
```

3 Run VRXLIB.

The VRXLIB tool reads the library archive and description files and produces the downloadable library file (`LIBNAME.LIB`), the interface file (`LIBNAME.O`), and a symbol file (`LIBNAME.AXF`) for the debugger. VRXLIB is called from the DOS command line as:

```
VRXLIB [-U] [-u] [-q] [-k] [-s] [-l linkopt] libname
```

Arguments

`libname` Library name.

Files `libname.lid` and `libname.a` must exist in the current directory.

Options

<code>-U</code>	Displays a use summary and exits.
<code>-u</code>	Suppresses warnings for unused function numbers.
<code>-q</code>	Invokes quiet mode; suppresses progress messages.
<code>-k</code>	Retains intermediate files, such as generated assembler source files.
<code>-s</code>	Generates source files only; does not assemble or link them.
<code>-l opt</code>	Specifies additional linker option or file. Only one argument can be specified, but it can be <code>-Ffile</code> , which directs the linker to read additional options from file.

Calling Shared Libraries

To use a shared library, link the application with the interface file `LIBNAME.O` produced by VRXLIB. Download the library file to the terminal if it is not already there.

Program files contain a table of the libraries needed, so Verix can load them along with the main program. This is created automatically when the program links with the library interface file. However the name of the library file on the terminal can be different from the name used when the interface file was created. The most common reason for this is that the library is downloaded to Flash, so its terminal name requires an `F:` prefix. The VRXHDR tool can be used to change library file names in the program file. Example:

```
> vrxhdr -l ether.lib=F:ether.lib
```

See the VRXHDR documentation for details.

Programs using shared libraries (other than the built-in system library) must be compiled with the `--apcs/shlclient` option when using RVCT2.0.1 and earlier versions. If using the VRXCC tool, this option is supplied automatically (see [Compiling and Linking](#)).

Updating Shared Libraries

A shared library can be modified without recompiling or relinking client applications, provided that the function numbers (as defined in the library description file) and the function interfaces (number and types of arguments) do not change. Applications automatically update after a new version of the library is downloaded.

If new functions are added to the library, they must be assigned unused function numbers. This generally means the end of the function list, unless slots were left open for this purpose.

If a function is deleted, its number can be reused. There should be no reference to deleted functions in applications. If an application calls an unused function number, a bus error occurs.

If the ID number of an existing function is changed, all client applications that call it must be relinked with the new interface file. If the number or type of function arguments is changed, callers must generally be recompiled.

Debugging Shared Libraries

Absolute shared libraries can be debugged by simply loading the library symbol information in the RVD debugger. For general information on debug procedures, see [Debugging](#). To debug an absolute shared library,

- 1 Build library with debug information.

In VRXCC use the `-g` option to save the debugging information.

- 2 Follow normal debug procedures for downloading and starting the download session on the application.

- 3 Load the debug information for the library in RVD:

- a Select **File > Load Image**.

- b Select the `.axf` file for the library in the scroll list and check the **Symbols Only** box.

RVD is now set up to debug the shared library.

NOTE



Due to limitations in the current version of RVD, the most reliable way to step into a function is through the low-level (instead of the high level) step into function.

Managing Shared Libraries

When a file is loaded into a terminal, it replaces any existing file that has the same name. This is a convenient feature for updating shared libraries through a simple download of a new revision with the same filename as the old version.

CAUTION



Avoid replacing a newer version of a library with an older one. Programs designed to run with a new revision may not function properly with a previous version. This is especially important in the Verix multi-application environment, where one application may work fine when referencing an older library, but another may fail when function calls it expects to reference in the library are not available in an older revision.

Also, if each application downloads a shared library into Group 15 (see File Management in the Verix V Operating System Programmers Manual; GID15 is open and referenced by all file groups), the last one loaded may not be the newest version and can cause other programs to fail.

Use the following guidelines to avoid this type of problem with shared libraries:

- Each application should always check the library version to prevent using an obsolete revision. See the discussions in [Version Numbers](#). Some applications may also want to prevent use of newer versions, depending on certification criteria. In this case, a shared library may not be the best solution.
- Shared libraries should always be backwards compatible so that new versions can be used by existing applications.
- If multiple applications are separately loading the same library, ensure that the newest revision is the last one loaded into a terminal.



Miscellaneous Tools

This section presents the tools available with the Verix OS.

VRXHDR Utility

Executable (.out and .lib) files begin with a header that contains the information required to load and run the program. The header can be inspected and parts of it changed using the VRXHDR utility. VRXHDR is the Verix equivalent of the Verix OUTHDR utility.

If VRXCC is used to build an application, VRXHDR is automatically called with the appropriate options. When using VRXHDR to modify the header of an executable file, ensure that the corresponding signature file is re-generated and downloads with the executable into the Verix-based Omni series terminal.

The VRXHDR usage is:

```
VRXHDR [-U] [-d|-n] [-t|-c] [-v n.m] [-s n] [-h n] [-l old=new] [-f n]
[-q] file
```

where, *file* is the executable or library file to list or modify. .OUT is assumed when no suffix is provided. The options are:

- U Print usage message and exit.
- d Mark executable as debuggable.
- n Mark executable as non-debuggable. By default, files are marked as debuggable. use -n option to prevent a file from being debugged.
- v n.m Set version to *n.m*. By default, files have a version of 1.0. The version information for an application can be checked using the `stdlib.h _version` variable. The version number is treated as two integers rather than a decimal number, therefore, 1.02 is equivalent to 1.2.
The allowed range is 0.0 to 255.255.
- s n Set stack size to *n* bytes. The default stack size is 4096. Verix allocates stack in 1024 byte increments.
If *n* is not a multiple of 1024, VRXHDR rounds up to the next 1024 multiple.
- h n Set heap size to *n* bytes. The default heap size is 4096.
- l old=new Rename the shared library reference. Useful if a library file is stored on the terminal under a different name than used to build it. The most common use of this is to prefix `F:` to the library name to indicate that it is stored in flash memory.
Library names are restricted to 12 characters, including any prefix and the .lib extension.

- f n Set flags byte to n (0–255). This option allows direct access to the bits in the flag byte and requires knowledge of the internal format of the header. This is not normally required and is provided for future expansion of flag bits.
- q Quiet; do not list file header.

VRXHDR Examples

Display a file header, VRXHDR sales.out:

```

Magic          0xA3 (program)
Flags          0x02 (Thumb)
Version        1.0
Code Addr      0x70420040
Code Size      8332 (0x208C)
Data Addr      0x70422400
Data Size      52 (0x34)
Heap Size      4096 (0x1000)
Stack Addr     0x7041F000
Stack Size     4096 (0x1000)
Entry          0x70420075 (Thumb code)
Library        .SYS.LIB

```

Set the stacksize for a program to 5K

```
Vrxhdr -s 5000 sales.out
```

Set version of shared library file PRINT.LIB to 2.7.

```
vrxhdr -v2.7 print.lib
```

Change the reference to shared library vfc.lib to indicate that it is stored in flash memory (F:).

```
vrxhdr -l vfc.lib=F:vfc.lib sales.out
```

VLR Utility

VLR.EXE converts text files into Verix variable-length record (VLR) and compressed variable-length record (CVLR) files. These files can be downloaded to a Verix-based Omni series terminal and read by the `read_vlr` or `read_cvlr` functions. The program can also perform reverse conversions.

Use

```
VLR [-U] [-a] [-c] in_file out_file
```

Arguments

```

in_file      Input file.
out_file     Output file.

```

Options

- U Displays VLR use summary and exits.
- a Reverse conversion: Convert VLR or CVLR to text.
- c Compress or decompress.

By default, the program converts an ASCII text file to a VLR file with one line per record. The `-a` option reverses the process, converting each record of a VLR input file to a line of ASCII text. If you are creating a record of Key Value pair format (like the `config.sys` file that the Verix V OS uses), each Key and Value are a separate line, make sure that you have an even number of lines in your input text file. `-c` enables compression (ASCII to CVLR) or decompression (CVLR to ASCII), depending on the presence or absence of the `-a` option.

See the documentation of `read/write_vlr` for more information about the VLR and CVLR formats. The VLR tool cannot handle all uses. For example, VLR records can store binary data, not just ASCII text. Also, since VLR and CVLR are record formats not file formats, different record types can be mixed in the same file. The VLR tool cannot handle mixed format files.

Example

```
vlr hotcards.txt hotcards.dat
```

Converts text file `hotcards.txt` to a VLR file `hotcards.dat`.

```
vlr -c hotcards.txt hotcards.dat
```

Converts text file `hotcards.txt` to a CVLR file `hotcards.dat`.

```
vlr -a hotcards.dat hotcards.txt
```

Converts VLR file `hotcards.dat` to ASCII text file `hotcards.txt`.

VLR Error Messages

Error messages for the VLR utility are described in [Table 8](#).

Table 8 VLR Utility Error Messages

Error Message	Description
No input filename specified	No filenames provided on the command line.
No output filename specified	Only one filename provided on the command line.
Can't open "file" - errno description	Attempt to open file failed. The string associated with the returned <code>errno</code> message displays.
Line n longer than 255 characters	Line number <i>n</i> of the input text file is too long. In VLR records, lines are limited to 255 characters.
Bad VLR format in file	Input file not valid VLR format file. The program reports this when it encounters an unexpected end of file.

Debugging

Full-featured symbolic debugging for Verix programs is provided by the RealView Debugger. Its capabilities and user interface are documented in the *RealView* online manual. This chapter describes how to use the debugger with Verix-based Omni series terminals; it also describes the use of Verix-specific debugger extensions available using the VRXDB program. Refer to the `readme.txt` file included in the SDK directory or the *Getting Started with VVDTK* guide on the VVDTK CD for current information for the RealView tool.

The debugger runs on a host PC and communicates with a monitor program running on the terminal over a serial communications port. It gets information about source code locations, variable names and types, and so on, from the `.axf` file created when the program is linked.

Verix ARM application debugging involves the following software components:

- RealView Debugger (RVD): Provides the user interface and main debugging engine, and is seen running on PC. The main component of RVD is called the Target Vehicle Server (TVS), referenced in debugger error messages.
- VRXDB (Verix Debug Controller): VRXDB executes RVD and serves as a go-between for it and the debug monitor running on the terminal. VRXDB normally runs on the same PC as RVD and communicates with it through TCP. It talks to the monitor using the serial or USB port. In addition to assisting debugger operations, VRXDB provides its own command interface that supports a number of Verix utility commands, such as listing terminal files. See [VRXDB](#) for details.
- Target Debug Monitor (dbmon.out): This program runs on the terminal that talks to the proxy and controls the target task. It uses the Verix API to access the target task's memory, start and stop execution, and so on.

Figure 4 illustrates the debugging process.

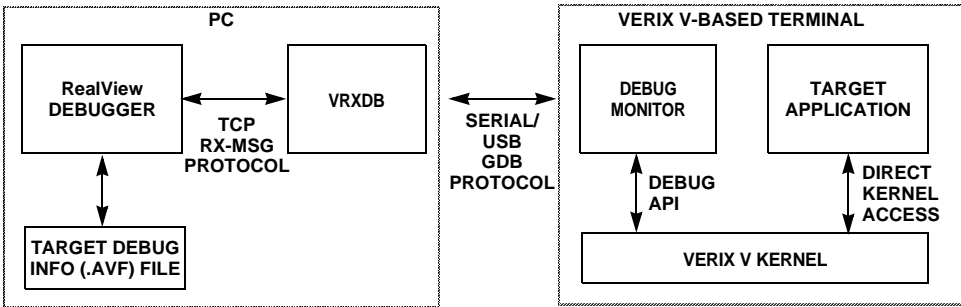


Figure 4 Debugging

Install the RealView Debugger

The RealView debugger is installed as part of the RealView Developer Suite installation.

NOTE



RealView ICE (RVI) software is not required anymore with RealView debugger (RVD) version 1.7 and later, but, is required for earlier versions of RVD. RVI is not installed as part of the RVDS installation, it is provided on a separate CD if needed.

Build a Debuggable Program

To enable source-level debugging you must request debug information when you compile and link your application. In VRXCC, use the `-g` option (for both compiling and linking—if done in separate steps.) This

- Instructs the compiler and linker to generate debug information in their output files.
- Preserves the `.axf` file produced by the linker.

Normally this file is deleted after VRXCC generates the Verix `.out` file. The debugger requires the `.axf` file as this is where all the debug information is stored.

- Sets the “debuggable” flag in the Verix `.out` file.

You can see (and change) this flag using the VRXHDR tool. Verix does not allow you to debug a file that does not have this flag set. VeriFone recommends turning this flag off in released software to prevent user access. Note that the file must be authenticated again after any modification.

When requesting debug information, the compiler normally reduces the optimization level from 2 to 1. Fully optimized code can be difficult to debug as object instructions may be reordered with respect to the source code, variables may be optimized out of existence, and so on. However, some problems may only be exposed only at higher optimization levels, leading frustration when an existing the problem goes away when using the debugger. To get both full optimization and debug information, use the `-O2` option with the `-g` option.

VRXDB

VRXDB is a helper program for debugging Verix applications. It has three principal functions:

- Launches the ARM RealView debugger (RVD).
- Mediates communication between RVD and dbmon, the debug monitor running on the terminal. VRXDB communicates with RVD through a TCP connection, using the ARM RealView Message protocol. It talks to the monitor over a serial or USB link, using an extended version of the GNU debugger remote protocol.
- Implements Verix-specific commands, for example listing files. A command line interface is provided for this purpose.

Running VRXDB

VRXDB is a console application run from a DOS command window, the Windows Start/Run menu, or a Windows shortcut icon. The syntax is:

```
vrxdb -? -n -d -l -i init-file -x command name=value out-file axf-file
```

The following lists options. All options and arguments are optional.

-?	Help. Print usage summary and exit.
-n	Do not read commands from <code>stdin</code> . After running commands from the initialization file and <code>-x</code> options, VRXDB exits unless starting a debug session.
-d	Launch the debugger. This is implied if a <code>out-file</code> argument is present. Used alone, it is equivalent to entering a debug command with no arguments.
-l (lowercase L)	Log activity to <code>vrxdb.log</code> . Equivalent to setting <code>log.level</code> to 1.
-i file	Initialization file. This file contains the VRXDB commands that run at startup. If this option is not present, VRXDB looks for a file named <code>vrxdb.ini</code> in the current directory. An error displays if it does not exist.
-x cmd	Execute the specified command. The command string must be quoted if it contains spaces. Multiple <code>-x</code> options are allowed. <code>-x</code> commands run after the initialization file.
name=val	Set a property (see Properties). Multiple property setting arguments are allowed. This must be a single argument; spaces around the equal sign (=) are not allowed.
out-file	Target program <code>.out</code> file (on the terminal). If specified, VRXDB automatically starts the debugger. This, and the <code>axf-file</code> argument, are equivalent to entering the command “debug out-file axf-file.”
axf-file	Target program debug information file (on the host PC).

At startup, VRXDB performs the following:

- 1 Sets properties defined on the command line.
- 2 Runs commands from the initialization file.
- 3 Runs commands specified in `-x` options.
- 4 Launches the debugger if a target program is specified.
- 5 Prompts for user-command input (unless `-n` is specified).

VRXDB terminates when a quit command executes or when no more commands remain and no debug session is in progress. To confirm that no commands remain, VRXDB must find the `-n` option, or an end-of-file on standard input.

Properties

Properties are name-value pairs similar to environment variables. They can be set from the VRXDB command line or with the `set` command. The following properties are defined:

<code>dbg.name</code>	Debugger to use. The value must be RVD.
<code>dbg.port</code>	TCP/IP port number for debugger communication. Default is 3010. DO not change this for RVD.
<code>dbg.tasks</code>	Control debugging of new tasks. The value can be: <ul style="list-style-type: none"> • <code>none</code>: Do not debug new tasks • <code>child</code>: Debug tasks created by tasks being debugged The default is <code>child</code> .
<code>tgt.port</code>	Set the PC serial port for target terminal communication. Port 0 denotes USB (default). Changes to <code>tgt.port</code> and <code>tgt.baud</code> must be made before running any command that talks to the target. Once target communication starts, changes are ignored.
<code>tgt.baud</code>	Serial baud rate for target communication (default 115200). Ignored for USB.
<code>log.file</code>	Log file name (default <code>vrxdb.log</code>). Changing the filename does not enable logging or affect any logging in progress. See log.level .
<code>log.level</code>	Log detail level. The log contains a record of commands, debugger messages, and so on. A level of zero (default) turns off logging; 1 turns it on; higher numbers add additional detail. The highest level is 3. Changing the log level from zero to a positive value opens the file specified in <code>log.file</code> , clearing it if it already exists. Setting the level to zero closes the log.
<code>cmd.prompt</code>	Command prompt string (default <code>></code>).
<code>cmd.lines</code>	If non-zero, command output pauses after each group of this many lines (default 0). Note: This feature is enabled only for commands that have lengthy output, such as <code>dump</code> .

Commands

Commands can be entered from an initialization file, using `-x` command line options or interactively. If standard input is redirected from a file, commands are read from the file without prompting. In this case, VRXDB exits when the end of file is reached unless a debug session is in progress.

Commands are not processed when the target program is running.

Commands with the potential to produce large amounts of output (for example, `dump`) can be interrupted by pressing any key. See also the [cmd.lines](#) property.

Commands consist of a name followed by white space-separated arguments.

Command names can be abbreviated to any unique prefix. (The help command illustrates the minimum abbreviation.) Comments are denoted by a pound symbol or hash mark (`#`).

Verix file names entered as command arguments have the general form `drive:group/name`. The drive and group prefixes are optional. If omitted, the default is `I:` and the current group. Using a slash that is not preceded by a number denotes GID15.

The following subsections describe the VRXDB commands. Aliases are also listed. To pass multiple args, separate with space and enclose in double quotes.

Debugging Commands

Command `debug [out-file [axf-file [args...]]]`

Description Launches the debugger

Command `free [task-number]`

Description Releases a new task from debugger control. This applies only to newly created tasks where a debug session has started. Once you begin debugging a task, you must use the debugger to free or terminate it.

Arguments

<code>out-file</code>	The Verix <code>.out</code> file that must be present on the terminal. The <code>.out</code> extension can be omitted.
<code>axf-file</code>	The corresponding debug information file on the host PC. If this is not specified, it is assumed to be <code>out-file</code> with the <code>.axf</code> extension. (Note that this works only if the file is in the current directory.)

Arguments following `axf-file` are passed to the target task through `argv`. If arguments are specified, you must explicitly specify the `.axf` file to avoid ambiguity. If no target is specified, the debugger launches, but no debug session starts.

Use the [Manual Startup](#) procedure to start a debug session.

The debug command cannot be used if the debugger is already running. To start a new debug session, terminate the debugger or use the [Manual Startup](#) procedure.

<code>args</code>	Optional debug command argument listing arguments to be passed to target task through <code>argv</code> . If multiple arguments are required, separate by spaces and enclose in double quotes.
<code>task-number</code>	Releases a new task from debugger control. This applies only to newly created tasks where a debug session has started. Once you begin debugging a task, you must use the debugger to <i>free</i> or terminate it.

Verix V Utility Commands

Command `group [group-number]`

Description Displays or changes current group.

Arguments With no argument, the current group number displays. The current group is the group in which `dbmon` is running. This affects commands such as `env`. Changing the debugger's group does not affect target tasks.

Command `dir [file]`
`ls [file]`

Description List Verix files in the local directory.

Arguments `file` can be an individual filename or just a group and/or drive. If no argument is present it defaults to `I:` and all groups. For example:

```
> ls
09/04/2003 18:30:31 v---          35412          1/DBMON.OUT
09/04/2003 18:30:32 v---          1448           1/TEST.OUT
09/04/2003 18:30:31 ----           152           1/CONFIG.SYS
09/04/2003 18:30:33 ---r           496           1/DBMON.P7S
09/04/2003 18:30:33 ---r           492           1/TEST.P7S
```

This listing shows the file time-stamp, attribute flags, size, and name.

`1/` indicates the group number.

The attributes flags are:

- `v`: authenticated
- `c`: no checksum
- `g`: no grow (fixed allocation)
- `r`: read only

Command `type file`
`cat file`

Description Display the contents of the specified text file.

Command `dump file`

Description Dumps the specified file in hex/ASCII format. For example:

```
> dump config.sys
00000000: 07 CA E4 E5 E2 F5 E7 02 2F 06 CA F3 ED E4 EC 02 ...../.....
00000010: 1F 07 CA E4 E2 ED EF EE 03 09 3F 04 CA E7 EF 0C .....?.....
00000020: E4 E2 ED EF EE CD E7 CE EF F5 F4 05 CA F5 F3 E2 .....
00000030: 02 8F 0A CA E4 E2 F4 E1 F2 E7 E5 F4 09 F4 E5 F3 .....
00000040: F4 CE EF F5 F4 06 CA ED E5 F2 F2 02 4F 06 F5 EE .....O...
00000050: FA E9 F0 02 1F .....
```

Command `env [name[=value]]`

Description Display or set Verix CONFIG.SYS variables. This can be used in the following ways:

<code>env</code>	Shows all config.sys variables.
<code>env a</code>	Show value for variable a.
<code>env a=b</code>	Adds or replaces variable a.
<code>env a=</code>	Deletes variable a.

Command `tasks [id]`
`task [id]`
`ps [id]`

Description List the tasks running on the terminal. If no task ID is specified, all tasks are listed. Example output:

ID	Grp	Status	Time	Sched	Data	Stack	File
1	1	ewait	74	67	6068	4096	SYSMODE.OUT
2	1	ready	21	24	6380	4000	DBMON.OUT

where,

ID	The task number.
Grp	The file group to which the task belongs.
Status	The following two values: <ul style="list-style-type: none"> • ready: Running and ready to execute when the CPU is available. • ewait: Waiting for an event. • twait: Waiting for the timer. • exit: terminated. An asterisk preceding Status indicates that the task is under debugger control.
Time	Accumulated execution time in milliseconds.
Sched	The number of times a task has been scheduled for execution.
Data	Size of data region in bytes.
Stack	Size of stack region in bytes.
File	Code file name.

Command `run file [args]`

Description Runs the specified program file as a new task. Not under debugger control. To pass multiple arguments, separate individual arguments with spaces and enclose the complete argument string in double quotes.

Command kill task

Description Terminates the specified task. This command cannot be used on tasks that are being debugged; these must be killed from the debugger.

Command crash

Description Displays the record for the most recent crash. Verix logs a crash diagnostic record whenever a task terminates due to an abort such as, a bus error or divide by zero. For example:

```
> crash
Type      1 (Data abort)
Task      2
Time      09/04/03 10:19:36
PC        704209F2
Addr      00000000
CPSR      00000030
Usr Reg   [ 0]=00000000    [ 1]=7041FDC4    [ 2]=00000000    [ 3]=7042093C
           [ 4]=7041FDC4    [ 5]=00000000    [ 6]=7041FDC4    [ 7]=7041FEC0
           [ 8]=00000000    [ 9]=7041FFF4    [10]=00000000    [11]=00000000
Und Reg   [SR]=80000010    [SP]=4000000C    [LR]=78F4DAF9
Abt Reg   [SR]=00000030    [SP]=4000000C    [LR]=704209F2
Svc Reg   [SR]=00000030    [SP]=10004800    [LR]=10108CB0
Irq Reg   [SR]=60000013    [SP]=40000680    [LR]=60000013
Fiq Reg   [SR]=00000000    [ 8]=00000001    [ 9]=00000002    [10]=00000003
           [11]=00000004    [12]=00000005    [SP]=00000006    [LR]=00000007
```

Command del file
rm file

Description Delete the specified file.

Command download file1 [file2]
dl file1 [file2]

Description Download file1 from the PC to file2 on the terminal. If file2 is not specified it defaults to file1, with drive and directory prefixes removed. Note that this is not a Verix-protocol download, but a simple remote file copy.

Command upload file1 [file2]
ul file1 [file2]

Description Upload file1 from the terminal to file2 on the PC. If file2 is not specified it uses file1, removing any group/drive prefixes. If file2 exists it is overwritten.

Command mem**Description** Displays information on memory and file systems. For example, displays a usage summary of the specified VRXDB command. If no command is specified, a summary of available commands displays.

```

> mem
      RAM      FLASH
Size      1048576  2097152  memory size
Files      44116   7104      memory used by file system
Tasks      20548   0          memory used by running tasks
System     186300  524288     memory used by OS
Recoverable 0       0          memory recoverable by flash coalesce
Free       797612  1565760    available unused memory

```

Miscellaneous Commands

Command help [command]
? [command]**Description** Displays a summary of the usage of the specified VRXDB command. If no command is specified, a summary of available commands displays.**Command** quit**Description** Exits VRXDB. Quitting terminates all target tasks on the terminal, but leaves dbmon running. It does not terminate the debugger.**Command** prop [name [= value]]
set [name [= value]]**Description** Set or display property values. Spaces around the equal sign (=) are optional. If = value is not specified, the current value displays. If name is not specified, all properties are listed. Note that properties can also be set from the VRXDB command line.**Command** ver**Description** Displays VRXDB and Verix versions.**NOTE**

The Verix version displays only during a target connection; this command does not force a connection.

Command batch file**Description** Executes commands from the specified file.**Command** echo text**Description** Print text.

Command `cls`

Description Clears the screen. Similar to the DOS command with the same name.

Debugging Procedures

Use the following procedures to start debugging your Verix application.

Automatic Startup

Use the following procedure for debugging a Verix application.

- 1 Download the application and the debug monitor (`dbmon.out`).

Both require signature files (`*.p7s`) for authentication.

- a Terminal debug communication parameters are determined by `*DBMON`. If not set, `dbmon` uses terminal COM1 at 115.2 baud. Otherwise the format of the `*DBMON` value is `pb` where, `p` is the port (0=USB, 1=COM1, 2=COM2), and `b` is the baud rate. See `<svc.h>` for values. (For USB, this is ignored).

Note that the baud rate must be compatible with the VRXDB `tgt.baud` property.

- b Set `*GO` to run `dbmon`.



Store the `dbmon` in flash so that it does not have to be downloaded every time the application is updated.

- c Restart the terminal.

When `dbmon` starts it beeps twice.

- 2 Locate the directory containing the `.axf` file for the application and run VRXDB using a command such as:

```
vrxdb test42.out
```

After issuing the `vrxdb` command, the debugger starts and usually displays the code for `main()` (but sometimes `src` tab needs to be clicked). RVD does not automatically run target program to `main()`. You must execute two “step over” commands to get through the C library startup code and the `main()`’s function entry code, or set a breakpoint and go.

If the `.axf` file is not in the current directory, execute VRXDB with a command such as:

```
vrxdb test42.out \myapps\test42.axf
```

To do any special setup before starting the debugger (for example, changing the baud rate), do not specify a target program when running VRXDB. Enter any required commands at the VRXDB prompt. When you are ready begin debugging, enter a debug command, for example

```
debug test42
```

If debugging the same program repeatedly, create an initialization file that contains the setup and debug commands. Also, create a Windows shortcut so as to do this by double-clicking a desktop icon.

Manual Startup

Debug sessions can be started from the debugger if no target program is specified on the VRXDB command line or the debug command. The following procedure is required for the following special cases:

- 1 launch the debugger using a debug command with no arguments, if the debugger is not already running.
- 2 Select a program to debug using the RVD Connection Control box.

If it does not pop up automatically, look for a link labeled “Click to Connect to a Target” or select **File>Connection>Connect To Target**.

Under the “Verix-Applications” entry, there is a list of .out files on the terminal. Click the check box for the desired target program. This starts a task running on the terminal and connects the debugger to it. At this point you can examine its memory and registers, step instructions, and so on.

- 3 Locate the .axf file.

To get a source-level view of the program, you must specify where to find its debug information in RVD. This is the .axf file generated when you linked the application. If the main window displays a link labeled “Click to Load Image to Target,” click it. If not, select **File>Load Image** from the RVD menu to display the **Load File to Target** dialog.

- a Check the Symbols Only box.

If the Symbols Only box is not checked, the debugger attempts to download the code to the terminal.



Do not click a link labeled “Click to Load xxx.axf.”

Although RVD remembers the last file, it does not remember the Symbols Only option that must accompany it. Always use the load image dialog box.

Ending a Debug Session

To end a debug session, uncheck its entry in the RVD Connection Control box. The target task is terminated if it is still running. If you prefer to leave the task running, select **File >Connection>Disconnect (Defining Mode)** and select **Free Running**.

When finished debugging, close RVD and use the quit command to exit VRXDB. Do any of the following to start another debug session:

- Leave RVD running and use the [Manual Startup](#) procedure.
- Close RVD, then use another debug command to restart it.
- Close everything and start over.

Normally you do not have to restart dbmon, but it is a good idea to reset the terminal occasionally to clear old tasks.

Debugging in Different Groups

VRXDB recognizes when a target program is in a different group and directs dbmon to switch groups as necessary. dbmon should always run from GID1. This facilitates program development, even if the application will run in a different group.

Debugging an Existing Task

If the target program specified in a debug command or selected in the Connection Control box corresponds to an already running task, the debugger takes control of it instead of starting a new task. The task is stopped at its current point of execution.

If you attach to an existing task using the Connection Control box, one change to the [Manual Startup](#) is necessary:

- You must uncheck the Auto-Set PC and Set PC to Entry Point options in the Load File to Target dialog.

Otherwise the debugger forces the execution address back to the start of the task.

Using VRXDB Without the Debugger

The Verix utility commands can be useful independent of the debugger. Simply connect to the target and enter VRXDB commands to list files, and so on. Use the `-n` and `-x` options to run VRXDB commands in a single step. For example,

```
vrxdb -n -x "dir F:"
```

lists the Verix flash files and returns (dbmon must be running on the terminal).

Multitask Debugging

When running multiple tasks under debug control, the following are important considerations:

- RVD thinks that each Verix target task under debug control is running on a separate CPU. So, for each task under debugger control, an RVD connection must be established and the corresponding Target Debug Info (`.axf` file) must be loaded. When a RVD connection is established, the corresponding task starts on the target terminal. If the task is already running under debug control, the RVD connection does not start another instance of the task.

- Currently, only 1 task under debug control can be active at any time. Other tasks under debug control must be stopped.
- VRXDB commands that require communication to the target terminal cannot be entered when a task under debug control is active.

The following are rules to determine if a task is under debug control

- If the debug monitor in the terminal, dbmon, starts the task, it is under debug control. dbmon starts a task if RVD attempts to establish a connection. This can be the result of any of the following:
 - Starting VRXDB with a debug target task parameter. For example, `vrxdb mytest.out`.
 - Entering the VRXDB `debug` command.
 - Opening the File/Connections dialog in RVD and checking the task name.

In each case, dbmon starts the task unless it is already running under debug control.

- A task started by a task under debug control, is also under debug control. This behavior is controlled by the VRXDB `dbg.tasks` property. By default, tasks created by tasks being debugged are under debug control. To change this behavior, set the property to “none” before the new task starts.
- A task can be attached to the debugger using the VRXDB `attach` command. Currently, this command can only be used on a task with a defined RVD connection that was previously freed.

Multitask Debugging Procedure

For a multi-tasking scenario, take an example of a parent task, `P`, which starts a child task, `C`. Use the following steps to debug these tasks.

1 Start the debugger on the parent task.

Methods to start the debugger on a task are:

The debugger can be started using either the manual or automatic startup procedures described in the previous sections.

2 Debug the parent task as necessary.

When the parent task runs the child task, the child task stops on its first instruction.

To debug the child task:

- Stop the parent task some point after it runs the child task.
- Manually establish a connection to the child task by using the following steps:
 - Select the File/Connection/Connect To Target option from RVD and check `c.out`.

- Load the symbol information (`c.axf`) from the RVD File/Load Image dialog. Ensure that the Symbols Only box is checked.

The child task displays in the RVD source window (the current tab may be at `dsm` instead of `src`).

At this point, the child task is at its first instruction. To get to the C code, press the high-level step several times or set a break point and run. Debug the child task as necessary.

3 Switch tasks.

RVD and VRXDB allow selection of a different debug task/connection anytime the current task/connection is *stopped*. To switch tasks, use the “Change to Next Active Connection” button on the RVD toolbar (the plus + sign with a drop down arrow next to it).

VRXDB *free* and *attach* Commands Example

Though the preceding procedure works, it may be cumbersome for debugging some multiple task scenarios. For example, there may be a server and client task and the server processing for a specific client request needs to be debugged. But that client request may only occur after the server has processed various other client requests. To debug this type of situation, use the VRXDB *free* and *attach* commands.

Use a procedure similar to the steps in [Multitask Debugging Procedure](#) to establish debug control for both the server and client tasks.

While the server task is stopped, temporarily free it from debug control by entering the VRXDB *free* command at the VRXDB command prompt. *free* requires the task ID. To determine the task number for the server task, use the VRXDB *tasks* command. Once the *free* command is entered, the server task is normally scheduled.

Prior to the client sending the server request that needs debugging, stop the client task. At the VRXDB command line, use the *attach* command to put the server task back under debugger control. When the *attach* command is entered, the task stops and is under debugger control.

Use the RVD “Change to Next Active Connection” button as necessary to switch between tasks and debug.

Ideally, it would be preferable to be able to set breakpoints in both tasks and then just start both of them running. But due to limitations in the current RVD and VRXDB versions, this is not supported. The VRXDB *attach* and *free* commands provide a workaround to allow debugging of multiple tasks.

***DBMON Abort Codes**

Table 9 lists common debug abort codes.

Table 9 *DBMON Abort Codes

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

Tips, Pitfalls, and Annoyances

Here are a few things you might find helpful when using the debugger. Always refer to the ARM documentation if experiencing difficulties.

- By default, VRXDB uses the PC COM1 serial port at 115200 baud to communicate with the terminal. On the terminal side, *DBMON defaults to COM1 at 115200 baud.

If a USB connection is preferred, set the VRXDB property, `tgt.port`, to 0, and on terminal, set `*DBMON=0xy` (where *x* is any numeric value and *y* is the desired log level). USB communications require a terminal with a USB port and a USB cable. The first time you run it, you may get a Windows pop up about a new device. There is no special driver. Just click OK or Cancel to close the pop up.
- Messages may display regarding “Target Vehicle Server” errors or another copy of the debugger running. After ensuring that no copies of RVD are running, check for the TVS.exe icon (ARM RealView Debugger Target Vehicle System) in the Windows toolbar. If present, double-click and select Kill Server from the pop-up menu.
- RVD logs the state of your debugging session in a file named `rvdebug.aws` in the directory
`..\Program files\arm\rvd\core\1.6.1\95\win_32-pentium\home\<your-name>`
or similar, depending which version of RVD is running and where it is installed. This is a text file. If start-up errors are a problem, remove any “CONNECT” or “History” entries in this file or delete the file.
- RVD remembers the `.axf` file name between sessions, but does not remember the Symbols Only setting.

Do not click prompts stating, “Click to Load “xxx.axf,” which attempts a full download. Always use the File/Load Image dialog and set the Symbols Only option.

- If you exit the debugger with the target still connected, RVD tries to automatically restore that connection on the next start up. This confuses VRXDB. If multiple tasks are connected, RVD attempts to connect to each one, which could start a task prematurely.

Always end your debug session by clearing the check box in the Connection Control dialog before exiting RVD. It may take long (~10 seconds) for RVD to

respond to a disconnect request. It is faster to first exit VRXDB, then click the connection check box.

- Stepping into a function using a function pointer: RVD has four buttons for single stepping. From left-to-right: Hi-level Step Into, Hi-level Step Over, Lo-level Step Into, Lo-level Step Over.

The Hi-level step buttons are for stepping through C code. The Lo-level step buttons are for assembly code. In RVD v1.6.1 Build 159 and earlier, using Hi-level Step Into does not step into a function pointed to by a pointer. However, this can be done using the Lo-level Step Into button. For example,

```
res = (*fp)(2, 3); // Breakpoint here
```

sets a breakpoint for using Hi-Level step until the function call via pointer is reached.

Press the Lo-level Step Into button as required until the source for the called function displays. Because each press of the Lo-level Step Into advances only one assembly instruction at a time, the cursor remains on the C source line until the actual assembly call instruction executes. In the previous example, three Lo-level Step Into button presses were required before the function source code appeared.

- Known RVD bugs:
 - When pausing the cursor over a variable name, its value displays in a pop-up window. Unfortunately, RVD gets the “endianess” backwards, (e.g., a value of 5 displays as 0x05000000).
 - String fields in structs display incorrectly in the Watch and Local panes.
 - Display of local arrays may have off-by-one errors.
- Known VRXDB bugs:
 - If VRXDB cannot connect to DBMON, it exits (it should wait for you to fix the problem).
 - Attaching to an existing task that was not previously freed does not work correctly because RVD resets the PC to the program entry address when it starts the debug session.
- Keep an eye on the log window at the bottom of the debugger screen. Sometimes critical events, such as target program aborts, show up in the logs.
- RVD has a command line interface (documented in a separate manual). When having difficulty doing something in the GUI interface, try the command line equivalent (e.g., the PRINTVALUE command seems to have fewer bugs than the Watch pane displays).
- If the debugger does not exit properly, breakpoints may be left in your code file. When the terminal restarts it detects these as file checksum errors. The only fix is to download the file again.

Tools

The fromelf is a multipurpose ARM tool used for debugging.

fromelf

The tool, fromelf, is included with the compilers and documented in the *Linker and Utilities Guide*. The following options are useful for debugging. fromelf can be run on object files produced by the compiler, or on the .axf file output by the linker (but *not* on .out or .lib files).

- g Dump debugging information. The result will be voluminous and largely obscure, but the line number entries which map addresses to source code locations are easily readable. They can be recognized by an open square bracket ([) character at the end of the line. For example:

```
000065:      SPECIAL(2, 6)      : 30      7042017e: maze.c:52.0 [
000108:      SPECIAL(4, 7)      : 38      70420462: maze.c:155.35 [
```

These entries show that address 7042017E is the start of the object code corresponding to line 52 of file `maze.c`; 70420462 corresponds to line 155, column 35 of the same file.

- c Disassemble code. Ensure that you know what you are doing when using this option; it can reveal useful information about what was happening at the time of a crash.

If debug info is present, the code is annotated with function names.

- s Dump symbols defined and referenced by an object file. This is not specifically tied to crash analysis, but may be a useful thing to know.



Understanding the Verix Error Log

The contents of the error log (also known as the *crash* log) can provide some clues as to what went wrong in the program. This section explains how to interpret common errors reported in the log. It assumes you are comfortable with hexadecimal notation and with the basic processor architecture. Use the log and code listings from the compiler or disassembled code in the debugger to get a more detailed picture of what the program was doing at the time of the crash.

Crash Kit Tools

The Verix V Crash Kit contains tools useful for analyzing crashes.

findsrc

Converts address to source location. It tries to find the source code location that corresponds to an address and requires a map file and an .axf file containing the debug information. Usage is as follows:

<code>findsrc [-d]</code>	Program [address].
<code>program</code>	Name of the program (or library), without a suffix. The file <code>program.map</code> or <code>program.axf</code> or both must exist.
<code>address</code>	Address to decode, in hexadecimal. A "0x" prefix is optional. If no address is given <code>findsrc</code> will prompt for addresses, continuing until you enter "q[uit]".
<code>-d</code>	Dump the entire line number table. <code>address</code> is ignored if specified.

`findsrc` runs the ARM `fromelf` tool to extract debugging information from the `axf` file. The directory containing `fromelf.exe` must either be on your search path or specified in the `RVCTBIN` environment variable.

Example

Assuming that `maze.out` has crashed, the error log shows a PC address of 70420372, and `maze.map` and `maze.axf` are in the current directory:

```
> findsrc maze 70420372
70420372 = maze.o + 570(0x23A)
          @ maze.c:129.20 +4(0x4)
```

The first line of output is based on looking up the address in the map file. It shows that it is 570 bytes from the beginning of object file maze.o. The second line comes from the .axf file and shows that the address corresponds to line 129, column 20 of file maze.c. Column numbers are shown if there is more than one statement per line. The code for this source location started 4 bytes earlier at 7042036E. Since the offset is less than 8 and the PC may be 8 bytes ahead of the actual crash location you should probably also try address 7042036A.

findlib

Converts system library address to function name. This is a Verix program which attempts to convert an address in the system library to the name of the function that contains it. It is usually used following a crash when the error log shows a PC address in the system library code range (7000xxxx).

By default, findlib decodes the PC address from the crash log. A different address can be specified in *ARG (as a hexadecimal number). For instance:

```
ddl findlib.out findlib.p7s *go=findlib.out *arg=70000C28
```

The results displayed in the terminal screen are:

Address	70000C28	address, from *ARG or error log
Function	memcpy	function name
Index	19	shared library function number
Entry	70000138	library entry point address
Code	70000C05	start of actual function code
Offset	36	offset of address from start code (decimal)

NOTE



If the offset in the last line is less than eight (8), the adjusted PC may be in a different function.

findlib's results are not always correct since it does not know about internal library functions. It is usually right for small functions such as those in the string library and is least reliable for complex functions like printf.

appmap

Displays application memory map. This displays the virtual memory map for an application and the shared libraries it uses. If the header of the program file contains Variable Libraries (shared libraries whose name starts with "@") or Weak Libraries (shared libraries whose name start with "?") the utility vrxhr.exe should be used to modify the program header to reflect the libraries that were actually used in the terminal, by either changing the library name (in case of a Variable Library) or by removing the library name (in case of a Weak Library) that was actually used in the terminal. Usage is as follows:

```
appmap    outfile [lib-dir...]
outfile    Program file. The ".out" suffix may be omitted.
```


lib-dir Optional list of directories to search for library files referenced by the program. By default, appmap searches the current directory and the directory containing the program file (if different).

Example:

```
> appmap app.out
```

Start	End	Size	Description
-----	-----	-----	-----
70000000	700077FF	30720	sys.lib (size approximate)
70100000	703FFFFFF	3145728	unused
70400000	7040003F	64	PICLIB.LIB filesystem header
70400040	7040038F	848	PICLIB.LIB
70400390	704003FF	112	PICLIB.LIB padding
70400400	704007FF	1024	PICLIB.LIB data
70400800	7041EFFF	124928	unused
7041F000	7041FFFF	4096	stack
70420000	7042003F	64	app.out filesystem header
70420040	704205CB	1420	app.out
704205CC	704207FF	564	app.out padding
70420800	70420837	56	app.out data
70420838	70421BFF	5064	heap
70421C00	706FFFFFF	3007488	unused

Regions marked “filesystem header” are Verix file system headers, which are mapped into the user address space along with code files. “Padding” results from rounding regions up to a multiple of the 1-KB page size.

The OS has some discretion about where it loads shared libraries. appmap tries to replicate the OS algorithms but these may change in future OS releases.

fromelf This is a multipurpose ARM tool included with the compilers. This may be run on object files produced by the compiler or on the axf file output by the linker (but not on .out or .lib files).

-g Dump debugging information. The result will be voluminous and largely obscure, but the line number entries, which map addresses to source code locations are easily readable. They can be recognized by a “[” character at the end of the line. For example:

```
000065: SPECIAL(2, 6) : 30 7042017e: maze.c:52.0 [
000108: SPECIAL(4, 7) : 38 70420462: maze.c:155.35 [
```

These entries show that address 7042017E is the start if the object code corresponding to line 52 of file maze.c and 70420462 corresponds to line 155, column 35 of the same file.

-c Disassemble code. If you know what you are doing, it can reveal useful information about what was happening at the time of a crash. If debug info is present, the code is annotated with function names.

-s Dump symbols defined and referenced by an object file. This is not specifically tied to crash analysis, but useful to know.

OS Debug Trace For some applications, it is useful to view the OS debug trace. Most of the information logged by the Verix OS is part of the system startup or system error handling.

Viewing the OS Debug Trace

The OS debug trace can be viewed by setting *DEBUG in GID 1.

*DEBUG = < Comm Port>

where <Comm Port> is 1 for COM1 or 2 for COM2. To view the trace, connect a PC running a hyperterminal or a similar program at 115.2 baud, 8n1. When *DEBUG is set, the associated communications part is not available for any other purposes. This includes downloading from System Mode.

Adding to the OS Debug Trace

To debug applications, a trace mechanism such as LOGSYS is often used. Since this type of log feature is implemented at the application layer, trace information is stored in the system buffer pool and the driver pushes the data out of the device in the background. In some situations, it is possible that application trace data is still buffered when the crash occurs.

An application can use the dbprintf() API with *DEBUG to directly log trace information. The dbprintf() is part of the system library but it is not included in any header file. The prototype is:

```
int dbprintf(const char* fmt, ...);
```

Unlike LOGSYS, dbprintf() output will actually be written before dbprint() returns. It disables interrupts affecting other aspects of your application.

System Crash OS Debug Information

When an application crashes, the *DEBUG output will include a complete error log, similar to vrxdb crash command. If the corresponding task's stack pointer is valid, a dump of the top 64 bytes in the stack follows the error log. The figure below provides a section of a sample log.

NOTE



Logging of stack information is not supported in older Verix operating systems. If in doubt, check your OS release notes.

```
*** run (CRASH.OUT, ) *** In G1: T1 -> T2 at 10008840; 1844628 avail
```

```
TCB addr 101CEF58
```

```
TCB_PC = 70420075
```

```
data = 70420800 .. 70421838 ( 5 KB)
```

```
stack = 7041F000 .. 7041FE00 ( 4 KB)
```

```
--Data abort! PC=700014AE CPSR=80000030 (USR,THM) Fault=10 User=T2
```

```
R0=00000010 R1=7041FD9C R2=00000000 R3=00000010
```

```
R4=7041FDD8 R5=42616420 R6=00000080 R7=7041FD98
```

```
R8=00000000 R9=7041FFF4 R10=00000000 R11=00000000
```

```
R12=70001521 SP=7041FD30 LR=70001527 PC=700014AE
```

```
Last SWI (0) ( 00000001, 7041FC40, 00000001, 7041FC40 )
```

```
SPSR_svc=20000030 SP_svc=10004800 LR_svc=00019648
```

```
SPSR_und=30000059 SP_und=4000000C LR_und=63264D49
```

```
SPSR_abt=80000030 SP_abt=4000000C LR_abt=700014AE
```

```
SPSR_irq=20000013 SP_irq=40000680 LR_irq=20000013
```

```
SPSR_fiq=80000013 SP_fiq=40000780 LR_fiq=00056D2C
```

```
7041FD30: 00 00 00 40 00 00 00 10 00 00 00 40 70 42 01 7F ...@.....@pB..
```

```
7041FD40: 00 00 00 80 53 20 37 30 34 31 66 64 39 38 20 44 ....S 7041fd98 D
```

```
7041FD50: 20 31 30 20 53 7A 20 31 32 38 00 5C 70 41 FD 6C 10 Sz 128.\pA.I
```

```
7041FD60: 00 00 00 01 00 00 00 01 70 41 FD EC 00 00 00 20
```

```
***** VERIX initializing *****
```

T1 is Task 1, which is always System Mode. System Mode runs Crash.out, which is Task 2 (T2).

T2 (Crash.Out) has crashed. The PC and LR addresses are in the System Library. Using FindLib, it is determined that memcpy() crashed. But the error log shows no application code addresses.

The Last SWI is the last software interrupt/OS call. In multi-task/app environments, this may or may not be the task that crashed.

The SWI number can be looked up in svc_swi.h. In this case, SWI 0 is write. The first parameter to write is a handle (00000001). Handle 1 is the console (STDIN/STDOUT).

Task 2's (Crash.Out) stack (top 64 bytes). Address 7042017F is listed. Using the FindSrc tool on Crash.axf, this address matches a call to memcpy().

After the crash, the system restarts.

Debugging With the Verix ARM Error Log

When a Verix application crashes, the OS writes the processor registers and related state information to a reserved memory area called the *error log* or *crash log*. A few minutes with the log and the tools described here may save hours of trial-and-error debugging. This section describes how to interpret the most important crash data.

Reading the Crash Log

Following a crash, the terminal performs a fast restart by passing the initial startup screen. System mode provides an option to reading the error log. You can display the most recent log anytime by selecting “Error Log” in system mode. The data is saved until it is overwritten by another crash.

NOTE



Debugger breakpoints also overwrite the log because they appear as crashes to Verix.

The ARM error log display appears as follows:

SYS MODE ERR LOG		
TYPE	1	abort type: 1 = data, 2 = program, 3 = undefined
TASK	2	task number. 1 = system mode, 2 = first user task, ...
TIME	030904092217	time of crash in <i>yymmddhhmmss</i> format
CPSR	400000030	status register
PC	704201A0	execution address
LR	70420140	return address
ADDR	00000000	fault address

This is actually a small subset of the logged data available the using VRXDB's *crash* command. More information about the system mode display.

Example Error Screen

If a Verix application crashes, the OS records the error information and the following screen displays:

```

SYSTEM ERROR
  ERROR LOG->
  SYSTEM MODE->
    RESTART->
    
```

Choose Error Log to display a screen similar to the one shown in [Reading the Crash Log](#).

Verix ARM Internal Components

Interpreting crash data requires some basic knowledge of the ARM processor and Verix memory organization.

NOTE



The ARM processor is fully documented in the ARM Architecture Reference Manual included with the compilers (some features are processor dependent; the V×510/V×610 family of terminals use the ARM 920T processor). The Verix application architecture is described in Application Programming Environment in the Verix V Operating System Programmers Manual.

Registers

As seen by an application program, the ARM has sixteen 32-bit registers, labeled R0–R15. Most of these serve as general registers for working data, but two have special hardware-assigned uses:

- R15 is the *program counter* (PC) that contains the execution address. Because the processor pre-fetches instructions, the PC usually contains an 8-byte address (in ARM mode), or 4 bytes (in Thumb mode), beyond the instruction being executed. Therefore, in most cases the address of the instruction that caused a crash is the PC address shown in the log minus 8 or 4.
- R14 is the *link register* (LR) that contains the return address for function calls. In some cases, such as a crash in a library function, this may be more useful than the PC. Unfortunately you cannot rely on LR always containing the current return address. The compiler may store it on the stack and then use the register for an unrelated purpose. Other optimizations may result in LR containing an older return address, for example to the caller's caller.

In some contexts the low bit of code addresses is set to 1 (making the address odd) to indicate Thumb code.

The *current program status register* (CPSR) (not labeled R0–R15) contains the processor state and condition code. If the last two hex digits are 10, the processor was running in ARM mode at the time of the crash; if 30 it was in Thumb mode. Anything else indicates an OS problem.

If there is a full crash log that shows all registers, it may be useful to know that R0-R3 are used to pass the first four arguments to a function (if there are more, they are passed on the stack). You cannot depend on R0-R3 continuing to hold the original arguments during function execution. R0 returns the function result.

The crash full log also includes registers associated with different processor modes. Normally these are not meaningful for application errors.

Aborts

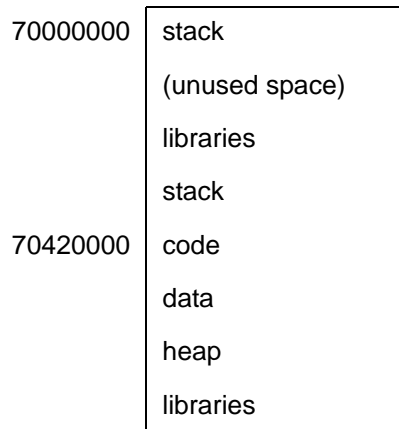
If the processor encounters an error condition, it generates an *abort* interrupt that is handled by the OS. This is what a crash actually is. There are three types of aborts:

- *Data abort*: An attempt to access data at an invalid address.
- *Program abort*: An attempt to execute code at an invalid address.
- *Undef abort*: An attempt to execute an illegal instruction.

For a data abort, the address that caused the problem is stored in a *fault address register* shown in the error log.

Memory

The ARM MMU controls which areas of memory can be accessed by applications (see Memory Management Unit (MMU) for further discussion). In Verix, the 128-MB address range 70000000–7FFFFFFF is reserved for application code and data, but Verix currently restricts them to the first 7 megabytes (70000000-706FFFFFF). Since this is virtual memory, each task has its own copy of this area, which is laid out as follows:



where,

system library	Built-in shared library containing common Verix and C functions. It starts at 70000000 and occupies approximately 20 KB, but the size varies with the OS release.
stack	Local variables and other function call data. The stack grows backwards (from higher-to-lower addresses) from the code region. Its maximum size is specified in the <code>.out</code> file header and can be changed with the VRXHDR tool.
code	Application code. Constant data, such as strings, is also included. This read-only region is actually the <code>.out</code> file mapped into virtual memory. The code region starts at the fixed address 70420000. The first 64 bytes are the Verix file system header. The file data starts at 70420040.
data	Global variables and other static data. Data starts at the next 1-KB page boundary following the code. If the program uses shared libraries with static data of their own it follows the main program's data.
heap	Storage pool for <code>malloc</code> and so on. The heap size is specified in the <code>.out</code> file header and can be changed with VRXHDR. ^a
libraries	Shared library code files. For absolute libraries the address is assigned by the library developer. The OS can load position-independent libraries wherever they fit, before or after the main program. If a library has static data it follows the main program's data, preceding the heap. ^b

- a. Since memory is allocated in 1-KB pages the heap extends to the next end of page. Hence a task usually gets a bit more heap space than it asks for.
- b. Therefore the heap address may vary depending on the libraries used. This is why VRXHDR does not show the heap address.

The VRXHDR tool shows the main region addresses and sizes for a program or library file.

Crash Analysis

So your program crashed and you are looking at the system mode error log. What can it tell you? First look at the abort type.

Data Abort

Data aborts are the most common type of crash. The `ADDR` field contains the illegal address that the application was trying to access.

- 1 If `ADDR` is 00000000, this is probably a null pointer error. A small number might be a null pointer used to access a `struct` field.
- 2 If it is a little bit less than the stack area address (obtained from VRXHDR or `appmap`), it was probably a stack overflow. Try increasing the stack size. "A little bit" is difficult to quantify in general, but if the address is less than the stack start but still of the form 704xxxxx, this is a good guess.
- 3 If it is a little bit greater than 70420000, there is a good chance that an array pointer or index ran off the end of a local array and hit the end of the stack area.

- 4 If it is a little bit greater than or less than the end of the data/heap area, it might be an array overflow involving a global or `malloc`'d variable.
- 5 If it is in the code region, the task may have tried to write to read-only data, for example changing a string constant.
- 6 If it is not even, a 70xxxxxx address it is probably an uninitialized or corrupt pointer. Look for other clues. For example, an address that looks like ASCII text suggests a pointer overwritten by a string, and the characters might provide a hint as to where they may have come from.

Program and Undefined Aborts

Program and undefined instruction aborts indicate a wild control transfer. If a program jumps to an illegal address a program abort results. If the address is legal but does not contain a valid instruction the result is an undef abort. The PC

NOTE



A random bit pattern has a fair chance of being a legal ARM or Thumb instruction. A program that jumps to a non-code memory area may execute garbage for awhile before crashing. Any kind of abort could occur in this case.

indicates the location, but since it is known to be bad, the LR register is often more helpful as the last known valid address. Possible causes of program and undef aborts include:

- 1 A return address saved on the stack was corrupt (perhaps by an array overflow), causing an error when the function returned.
- 2 A bad function pointer.
- 3 Shared library problems such as, a version mismatch between the downloaded library and the interface file to which the caller linked.
- 4 An attempt to run ARM code in Thumb mode or vice-versa. The linker usually catches this kind of problem.

In the last two cases, the PC address probably points into or near to a valid code or library region. If the PC value is not even, a 70xxxxxx address the first two causes are more likely.

Mapping Addresses to Source Locations

If the crash log shows a reasonable looking PC and/or LR address, you might be able to map it to a source code location. This requires a load map and/or an `.axf` file that contains debug information.

If you compile using `-g -O2` and link with `-g -map`, the information is available and there is no effect on the size or performance of the executable.

NOTE



`-g` requests debug information. This normally also reduces the optimization level, but `-O2` restores it to the default. If you intend to run the program under a debugger, omit `-O2` since aggressive optimization can make debugging confusing.

If these files are missing, try rebuilding the application with the desired options. Since `.out` files contain no timestamps or path names, it should be possible to exactly duplicate the original `.out` file.

The load map shows the starting address of the code from each object file. You can determine the source file corresponding to an address using the load map. This is sometimes enough of a clue by itself.

Remember that the PC address in the error log may be 4–8 bytes ahead of the instruction which caused the crash. There is no easy way to know if this is the case so try both.

Advanced users may want to compile the source file involved using the `-asm` and `-fs` options to generate an assembly listing. This provides the maximum amount of information about what the program was doing when it crashed, especially if you have a full crash log showing all the registers.

Crashes in Libraries

If the PC address is in the load map but you do not recognize the object filename, it is probably in either a Verix or ARM library function. The function can often be determined from the object filename. The LR address may point back to where the code called the library. A bug in the library may have been encountered, but it is more likely that the application passed a bad argument.

If the PC address is 7000xxxx, the crash occurred in the system library.

If the PC is in a user shared library, use the library's load map and/or `.axf` file to locate the source code involved. Note however, that addresses for position-independent (PIC) libraries are relative since the linker does not know where the code will be loaded. To decode a PIC library address, subtract the library load address from the address of interest, then look up the offset using the library map.



Porting Verix 68K Applications to Verix ARM

This section describes application programming differences between the Verix 68K environment and Verix ARM. It is intended to be useful to programmers porting existing Verix 68K applications, as well as to those developing new code who want to know what is different in the new environment. This chapter discusses the following topics:

- **C Programming Issues:** Language and tool differences, including internal data representations. This is based on the ARM Developer Suite (ADS) v.2.
- **Verix Interface Issues:** API differences between the two platforms.

C Programming Issues

This section discusses C programming issues.

Compiler Errors and Warnings

Your first attempt to compile Verix 68K code for Verix ARM is likely to result in a flood of warning and error messages. Analysis will usually reveal a much smaller number of distinct problems repeated multiple times.

The ARM C compiler (`armcc`) more strictly enforces ANSI C compliance than the SDS Verix 68K compiler.

More warnings are enabled by default and some Verix 68K compiler warnings are elevated to fatal error status.

Some warnings and errors can be disabled or made non-fatal with compiler options. Ensure that you understand the problems well enough to be convinced they are harmless before starting the compile.

Most new problems are detected by the Verix 68K compiler's `lint` option (`-l`). Start a porting project by first getting an `lint`-free Verix 68K build. Standalone checking tools, such as *PC-Lint* from Gimpel Software, are also available.

Here are a few specific areas where new errors and warnings may display. This is not a comprehensive list.

- **Const-correctness:** The Verix 68K compiler treats assignments to `const` data as only a warning and does not even warn about passing a `const` pointer to a non-`const` function argument. The ARM compiler treats both as fatal errors. C++ code is particularly vulnerable to these errors because in C++ quoted string literals are `const1`.

The Verix ARM version of `<svc.h>` adds *const* modifiers to many Verix API function prototypes. You may have to do the same in your code.

Although C string constants are not modifiable they are not formally *const* because the ANSI C committee realized that making them so would break lots of existing code.

- Pointer compatibility: Implicit conversions between pointers to different types generate a warning on the Verix 68K, but a fatal error on the Verix ARM. A common example is passing an `unsigned char*` pointer to a function which expects a `char*`.
- By default, `armcc` issues warnings for old-style (that is, unprototyped) functions. Watch out in particular for functions with no arguments declared as `foo()` instead of `foo(void)` (the former is a valid prototype in C++, but not in C).

Data Representation

The binary representation of some data types differ between the two platforms. The differences should be transparent to most well-written C code. However the low-level nature of C can expose them in some cases.

Pay special attention to binary data in files and communication messages which are used outside the program. One area where there is *not* an incompatibility is byte order: Verix runs the ARM in big-Endian mode, like the Verix 68K (and unlike most other ARM systems.)

Type Sizes

`ints` are 32 bits (4 bytes) on the Verix ARM, versus 16 bits (2 bytes) on the Verix 68K. Other integer sizes are the same: `char` = 8 bits, `short` = 16 bits, `long` = 32 bits. The ARM compiler also provides a 64-bit `long long` type.

Since C generally converts integer sizes as needed and since Verix ARM `ints` are bigger than Verix 68K `ints`, the size difference is transparent in most cases. Of course blatant dependencies such as, hard-coding 2 instead of `sizeof(int)` will cause problems.

C promotes `char` and `short` values to `int` when they are used in expressions. This can occasionally cause surprises. Consider the following:

```
unsigned short word, nibble;
nibble = (word << 4) >> 12;
```

This code probably intended to extract the second 4 bits of `word` by shifting out the high 4 bits, then right justifying the next 4. However, because `word` is expanded to 4 bytes before the first shift, it has no effect.

You may be tempted to change `ints` to `shorts` for Verix 68K compatibility and to save space. However 16-bit values tend to be less efficient than 32 bits on the Verix ARM, as a result of expanding them to `ints` for computation and narrowing them back for storage. Changing an `int` variable to `short` might save 2 bytes of

data space, but for example, may add 8 bytes of code to access it. As a rule of thumb, use `short` only for arrays and other data structures that have a significant impact on memory use, or for cases where the program logic requires exactly 16 bits.

If you are thinking of using portability typedefs, such as `int16`, look at the Verix ARM header file `<stdint.h>` that contains a set of standardized definitions. Be aware that if such definitions are not used thoughtfully they can make matters worse rather than better. For example:

```
INT16 i;           this should probably be int in Verix ARM
INT16 table[500];  this should probably be short in Verix ARM
```

Alignment

The Verix ARM requires 4-byte values (`int`, `long`, `float`) to be naturally aligned in memory, that is, stored at an address that is a multiple of 4. The Verix 68K requires only that such data be stored at an even address. Usually the compiler takes care of this. The following example shows how a problem could arise.

```
short wordbuf[64];           128-byte buffer on an even address
char *buffer = (char*)wordbuf;  access buffer as char
...
*(long*)&buffer[10] = length;  store long length
```

The cast in the final statement lets the program efficiently access buffer bytes 10-14 as a long word. Declaring `wordbuf` as `short` ensures that `buffer[10]` is at an even address, as the Verix 68K requires. But on the Verix ARM it may not be on a 4-byte boundary, causing an abort². A safe but slow way to store the data is

```
memcpy(&buf[10], &length, 4);
```

Other alternatives include forcing the buffer to be aligned at a 4-byte+2 address or defining the buffer as a packed structure. The ARM compiler's `__aligned` keyword may be useful (though non-portable) when dealing with special alignment requirements.

Structures

Both the Verix 68K and Verix ARM compilers may insert padding into structures to properly align fields. But since alignment requirements are more strict on the Verix ARM additional padding may be required. For example:

```
struct op { short opcode; long address; short flags; };
```

This structure occupies 8 bytes on the Verix 68K. The Verix ARM compiler inserts 2 bytes of padding following the first field to align the long value on a 4 byte boundary and add 2 more bytes at the end to make the size a multiple of 4 bytes. Thus the structure size will be 12 bytes.

Use the Verix ARM `__packed` attribute to override the default layout rules, as follows:

```
__packed struct op { short opcode; long address; short flags; };
```

The resulting structure size is now 8 bytes, with the same layout as the Verix 68K. The `long` field is not naturally aligned, but because the compiler knows about it, it generates code to access it safely. (There may be an efficiency penalty for this.) Note that packed Verix ARM structures are not always equivalent to Verix 68K structures. A combination of `__packed` and `__aligned` may be required to define a structure with a specific binary layout.

Bit-field structure members are inherently non-portable. If you are using bit fields and care about how they are arranged in memory, see the detailed documentation in the ARM compiler manual. Since the rules are rather complex, verify your declarations by dumping out sample data.

Signed vs. Unsigned Characters

On the Verix ARM, `char` is unsigned by default. On the Verix 68K it is signed. For most text processing this makes no difference. (Standard ASCII character codes are positive either way.) However, if your program uses character codes with the high-order bit set or uses `char` to store small integer values, it may need to be modified. Portable code should explicitly specify `signed char` or `unsigned char` if it makes a difference.

Note that `char`, `signed char`, and `unsigned char` are considered three distinct types in C, even though two of them are always the same. Therefore, a code such as, `unsigned char name[32]; n = strlen(name);` generates an error for passing an `unsigned char*` to a function expecting a `char*`, even though the types are identical. Worse, it may happen to be properly aligned and pass all tests, then fail when an unrelated change somewhere else in the program causes the run-time location of the data to move.

The Verix ARM `-zc` option changes the default to signed characters. However use of this option is not recommended since it is incompatible with the run-time libraries.

#include Paths

Nested `#includes` can be affected by a difference in the way that the Verix ARM and SDS preprocessors search for files. Both look first in the current directory, then in the directories specified in the `-I` command line options. The difference is how the current directory is defined. The SDS compiler starts with the directory containing the original C file. The ARM compiler uses the directory containing the file doing the include. For example consider the following directory and file structure:

```
src/           directory
test.c         #includes util/cvt.h
util/          directory
cvt.h          #includes "config/arm.h"
config/        directory
arm.h
```

In Verix 68K, `cvt.h` includes `arm.h` from its brother `config` directory. However the `armcc` preprocessor looks for `config/arm.h` in the `util` directory (that contains `cvt.h`) and reports that it cannot be found.

The `-fk` option causes the preprocessor to use the same search rules as SDS. However it is generally best to avoid dependencies on either method by using only simple file names (no directories) in `#include` statements and explicitly listing all necessary directories in `-I` options.

Another common source of `#include` problems is the careless use of bracketed (`<...>`) versus quoted filenames. Only use brackets for standard C library header files (for example, `<string.h>`) and other system files, never for application headers.

Determining the Platform (#ifdefs)

Ideally, you will be able to modify non-portable code so that it works on both platforms. But sometimes you cannot avoid `#ifdefs` to select platform-specific code. Both compilers predefine macros that can be tested to determine the platform. Useful macros defined by the ARM compilers include the following (note the use of two underscores):

- `__arm`: Any ARM/Thumb compiler
- `__thumb`: Thumb compiler
- `__cplusplus`: C++ compiler
- `__sizeof_int`: Defined as 4

The SDS Verix 68K compiler defines:

- `_TARG_68000`
- `_TARG_CPU00`

See the compiler documentation for complete lists.

Verix Interface Issues

This section discusses particular issues between the Verix 68K and Verix programming environments.

Const-Correct Prototypes

Verix 68K function prototypes are changed to use `const` when appropriate. For example, in Verix 68K `dir_get_file_size()` is defined as `long dir_get_file_sz (char *filename)`, but the Verix `<svc.h>` file defines it as `long dir_get_file_sz (const char *filename)`.

Adding `const` to function prototypes should not cause problems except in very unusual cases, such as an application file that declares a library function locally rather than including the proper header.

Narrow Arguments

For reasons of efficiency and consistency some `short` and `char` function arguments have been changed to `int`. The underlined arguments in the following functions were `short` or `char` in the Verix 68K API. (Note that in Verix 68K, `short` and `int` are the same size.)

```
int gotoxy (int x, int y)
int write_at (char *buf, int len, int x, int y);
int setcontrast (int flag);
int set_backlight (int flag);
int set_cursor (int flag);
int put_graphic (char *buf, int len, int x1, int y1, int x2, int y2);
int key_beeps (int flag)
int set_hot_key (int keycode)
```

Unimplemented and Deprecated Functions

Secure Console

The Omni 2650-compatible secure console interface is not implemented in Verix. The following functions have been removed:

- `disable_kbd()`
- `display_blink()`
- `emit_beep()`
- `enable_kbd()`
- `erase()`
- `KD_reset()`
- `get_secure_version()`
- `load_keypad()`
- `put_bitmap()`
- `read_console()`
- `read_cursor()`
- `read_fraud_counter()`
- `request_pin()`
- `reset_display()`
- `reset_key_attributes()`
- `secure_command()`
- `security_status()`
- `set_display()`
- `set_font_id()`
- `set_free_keys()`

- `set_key_attributes()`
- `set_secure_file()`
- `write_console()`
- `write_raw()`

File System Components

The following functions deal with internal details of the Verix 68K file system that are not generally useful to applications. They have been removed from the Verix API `<svc.h>`, although they are still supported in the OS for use by system mode and other internal applications.

- `dir_checkall()`
- `dir_chk_sizes()`
- `dir_get_checksum()`
- `get_file_base()`
- `dir_get_file_base()`
- `get_file_hdr()`
- `dir_get_file_hdr()`
- `dir_get_hdr_base()`

Changed Functions

The following functions differ in some way from their Verix 68K implementations:

- `_exit()`
`_exit()` now takes an argument, like the standard POSIX version.
- `datetime2seconds()`
- `seconds2datetime()`

The seconds argument is defined as a pointer to unsigned long rather than long. The unsigned range is required to represent dates later than January 19, 2048. (The Verix 68K supports these dates if suitable casting is applied. The use of long in the function prototypes was an error.)

- `SVC_INFO_CRASH()`

The `info_crash` structure, which is highly processor-dependent, is redefined.

- `get_component_vars()`

Although no specific changes have been identified, the use of new drivers might require changes to the format and interpretation of the data returned by `get_component_vars()`.

Additional Resources

Coding for portability is discussed in most good C books. Particularly recommended is *C: A Reference Manual* by Harbison and Steele. This is known as the “K&R” method, since it was the one used in old C compilers. The ANSI C standard does not have anything to say about this because it considers directories as an implementation dependency.

The primary reference for the ARM tools is the ARM RealView Developer Suite (RVDS) manuals. Some additional documents available from the ARM web site (www.arm.com) may be of interest to those wanting more information.

Application Note 34, *Writing Efficient C for ARM*, contains a number of useful tips.

Installing the Verix USB Client RS232 Driver

This section describes the step-by-step procedures for installing the Verix USB Client RS232 Driver.

Procedure

To install the driver:

- 1 Set “USBCLIENT” environment variable to “RS232”.
- 2 Connect the Verix USB device to the host PC. The “Found New Hardware Wizard” dialog box will appear.
- 3 Choose “Install from a list or specific location (Advanced)”, then click Next. See [Figure 5](#).

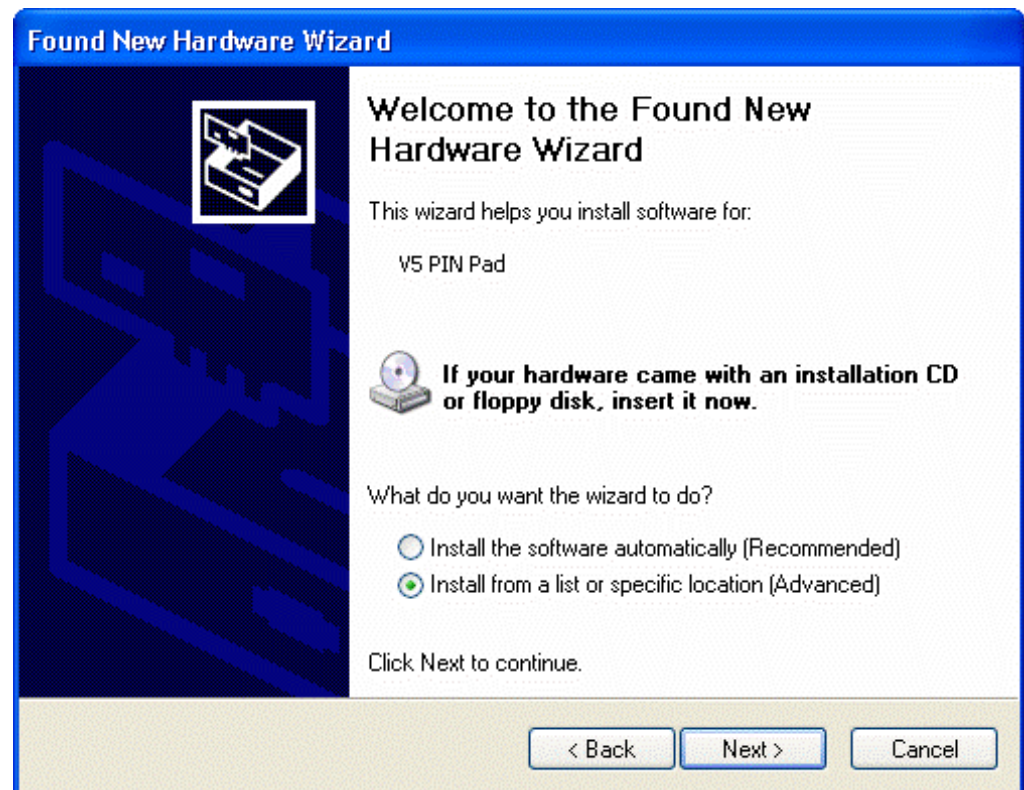


Figure 5 Install from a list or specific location

- 4 Search removable media or browse for the location of “VXUART.inf”, then click “Next”. See Figure 6.

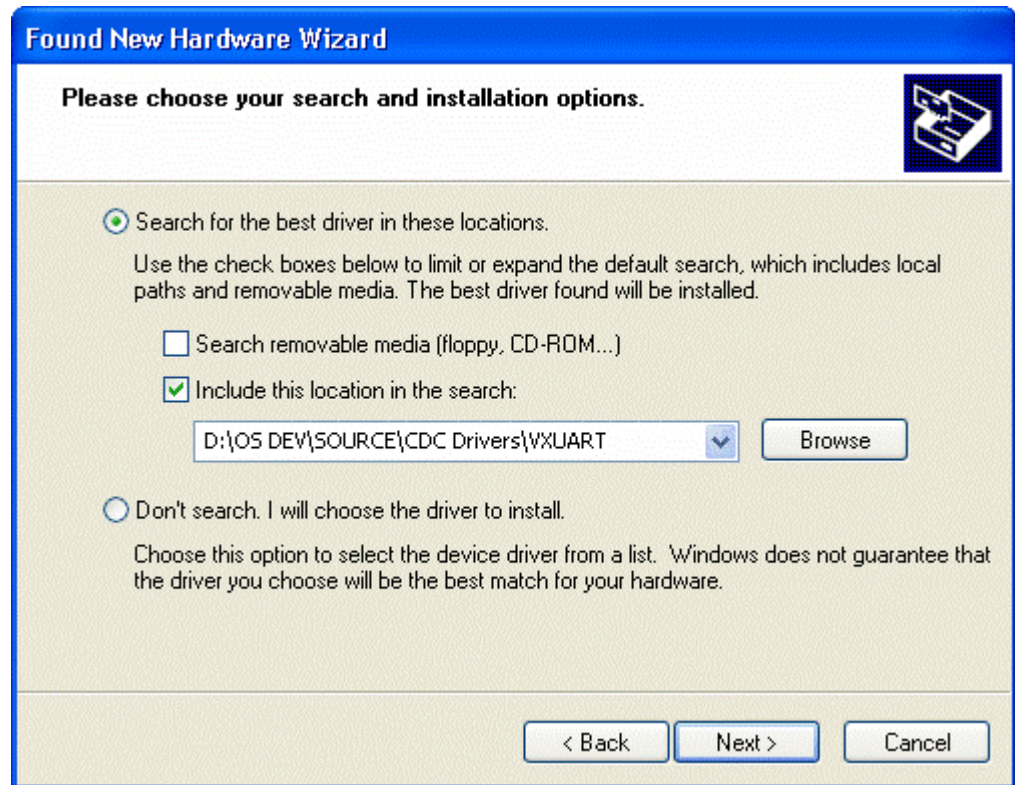


Figure 6 Choosing search and installation options

- 5 The wizard will inform you that the software driver has not passed Windows Logo testing, click “Continue Anyway”. See Figure 7.



Figure 7 Windows Logo testing warning

NOTE



During hardware installation, Windows may or may not detect software that has not passed Windows logo testing, depending on what option is set for Driver Signing.

- If Driver Signing was set to “Ignore”, all device drivers are installed regardless of whether they have or do not have a digital signature. Step 5 in this procedure will not occur.
- If Driver Signing was set to “Warn”, a warning is displayed when attempting to install any device driver without a digital signature, so that you can decide whether to cancel or continue the installation process. Step 5 in this procedure will occur.
- If Driver Signing was set to “Block”, device drivers without a digital signature cannot be installed. You will not be able to complete the driver installation.

- 6 Click “Finish” to complete the installation. See [Figure 8](#).

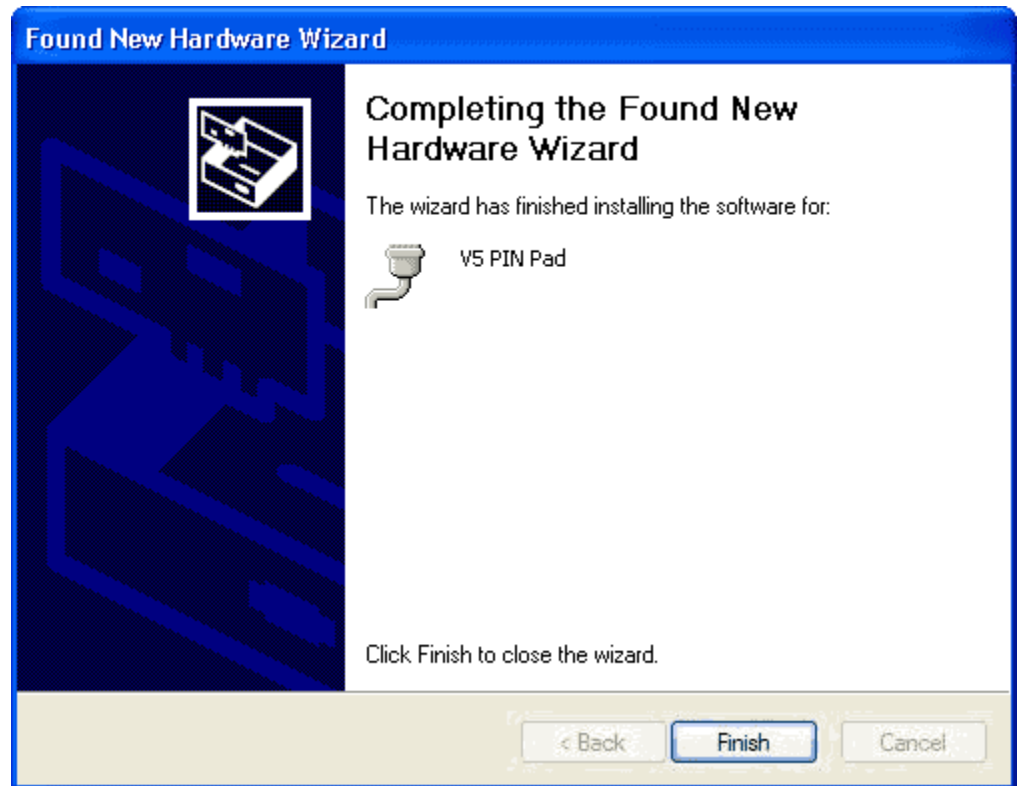


Figure 8 Found New Hardware Wizard completion

Refer to the Windows Device Manager to check if the driver has been successfully installed under Communications Ports.

**A**

abort codes **99**
acronym list **9**
address mapping **112**
ANSI C reference **19**
applications
 porting Verix 68K **115**
armar **58**
assembler options **57**

B

batch file **93**

C

C data types **17**
C header file **19**
 <assert.h> **19**
 <ctype.h> **23**
 <locale.h> **20**
 <math.h> **23**
 <setjmp.h> **24**
 <signal.h> **24**
 <stdarg.h> **24**
 <stdio.h> **24**
 <stdlib.h> **28**
 <string.h> **30**
 <time.h> **30**
C library **19**
calling libraries **77**
cast **18**
child task **97**
commands
 ? **93**
 attach **98**
 batch **93**
 cat **90**
 cls **94**
 crash **92**
 debug **89**
 del **92**
 dir **90**
 dl **92**
 download **92**
 dump **90**
 echo **93**

env **91**
free **89, 98**
help **93**
kill **92**
ls **90**
mem **93**
ps **91**
quit **93**
rm **92**
run **91**
task **91**
tasks **91**
type **90**
ul **92**
upload **92**
utility **90**
ver **93**

compiler options **55, 56**
compiling **14**
compressed variable-length record (CVLR) **82**
CONFIG.SYS
 argument to set variables **61**
CONFIG.SYS environment variables
 *ARG **18**
 _version **72**
CONFIG.SYS variables
 *DBMON **99**
 *GO **94**
conventions
 measurement **8**
crash log **92, 103**
CVLR. See also, *compressed variable-length record*. **82**

D

*DBMON **99**
date setting **61**
dbmon.out **85**
DDL **59**
 communications parameters **60**
 environment variables **61**
 file authentication **62**
 options **60**
debug monitor **94**
debugging **85**
 .axf file **86**

- abort codes **99**
- automatic startup **94**
- commands **88**
- dbmon.out **94**
- ending **96**
- groups **96**
- manual startup **95**
- monitor **94**
- multitask **96**
- options **87**
- procedures **94**
- properties **88**
- RVD **85**
- Target Debug Monitor (dbmon.out) **85**
- tasks **96**
- VRXDB **85**

Direct Download Utility (DDL) **59**

DLL **68**

downloads **14, 92**

- direct **59**

- direct using DDL **59**

- file authentication **62**

Dynamic Link Library **68**

E

environment variables **54**

error log **103**

example program **14**

F

file authentication **14, 62**

- downloads **62**

file signing **14**

files

- .axf **86, 94, 95**

- batch **93**

- crash log **92, 103**

- dbmon.out **85, 94**

- deleting **92**

- display .txt **90**

- downloading **14**

- error log **103**

- execute **91**

- extensions **53**

- header **19**

- library archive **75**

- library description **75**

- list **90**

- log **103**

- program source **14**

- removing **60, 92**

- run **91**

- uploading to host PC **92**

- VRXDB log **87**

functions

- main() **71**

G

global data **70**

global variables

- non-standard **28**

groups **96**

H

header options **58**

I

intra-library calls **72**

L

libraries **55**

- archive **65, 68, 75**

- ARM or Thumb object code **67**

- C++ **71**

- callbacks **71**

- calling **77**

- crashes **113**

- creating **65**

- creating shared **74**

- debugging **78**

- DLL **68**

- dynamic **68**

- intra-library calls **72**

- linked **66**

- linked vs. shared **66**

- list contents of **68**

- location in memory **68**

- macros **71**

- main() function **71**

- managing **79**

- memory **69**

- names **74**

- shared **65, 66, 70**

- shared. See also *DLL*. **68**

- static **68**

- system library **65**

- updating **78**

library description file **75**

library functions **65**

library options **58**

link options **55, 57**

M

- macros **71**
 - _SHARED_LIB **71**
- main() **71**
- measurement conventions **8**
- memory **110**
 - display info on **93**
 - libraries **69**
 - stack and heap **18**
 - virtual **69**
- multitask debugging **96**
- multitask debugging procedure **97**

O

- options **55, 60, 81**
 - debugging **87**
- OUTHDR **81**

P

- files
 - .p7s **14**
- parent task **97**
- passwords **61**
 - system mode default **15**
- pointer variables **70**
- porting applications **115**
- program arguments **17**
- program source file **14**
- properties **88**

R

- RVD **85, 86**
- RVD debugger **78**

S

- shared libraries **65, 68**
 - absolute (ABS) **67**
 - position-independent (PIC) **67**
- shared libraries, creating **74**
- shift and mask **18**
- stack and heap **18**
- strings **30**
- system mode
 - password **15**

T

- Target Debug Monitor **85**
- tasks **91**
 - child **97**

- debugging **96**
- determine if under debug control **97**
- killing **92**
- parent **97**
- switching **98**
- terminating **93**
- time conversion **31**
- time manipulation **31**
- time setting **61**
- timezone and DST Implementation **32**

U

- USB **123**
- utilities
 - ARM
 - fromelf **101**
 - armar **68**
 - DDL **59**
 - downloadutilities
 - DDL **15**
 - OUTHDR **81**
 - VLR **82**
 - error messages **83**
 - VRXCC **14, 53, 77**
 - VRXDB **85, 86**
 - VRXHR **58, 72, 77, 81**
 - VRXLIB **77**
- utility commands **90**

V

- variable-length record (VLR) **82**
- variables
 - DDL **61**
- VeriShield **14**
- Verix 68K
 - porting applications from **115**
- Verix 68K programming environment differences **66**
- version number **18**
- version numbers **72, 93**
- virtual memory **69**
- VLR utility **82**
 - error messages **83**
- VRXCC **14, 77**
- VRXDB **85, 86**
 - commands **88**
 - log file **87**
 - options **87**
 - properties **88**
 - using without the debugger **96**
- VRXHR **58, 77, 81**
 - options **81**

INDEX

V

VRXLIB **77**

VVSDK

 contents **12**

Vx510/Vx610 terminal features **11**



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

VeriX® V Operating System

Programming Tools Reference Manual

