

# VSH Users Guide

---

Release 1.0

*for x86 and PowerPC Environments*

**Thomas Besemer Consulting**

<http://www.tbcorp.com>

---

Product names mentioned in the *VSH Users Guide* are trademarks of their respective manufacturers and are used here only for identification purposes.

Copyright ©2000-2003, Thomas E. Besemer. All rights reserved.

Printed in the United States of America.

All rights reserved. No part of the *VSH Users Guide* may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photographic, magnetic, or otherwise, without the prior written permission of Thomas E. Besemer.

Thomas E. Besemer makes no representations, express or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither shall Thomas E. Besemer be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights which vary from state to state within the United States of America.)

---

# Contents

---

<b>PREFACE</b>	<b>V</b>	
For More Information	v	
Typographical Conventions	vi	
Special Notes	vii	
Technical Support	vii	
<b>CHAPTER 1</b>	<b>OVERVIEW</b>	<b>1</b>
	Introduction	1
	Key VSH Components	2
	The VSH Device Driver	2
	The VSH Shell	3
	The VSH Agent	3
	VSH Users Guide Examples	4
	VSH Example Kernel Memory Declarations	4
	VSH Example Kernel Procedure	4
	How They are Used	5

---

<b>CHAPTER 2</b>	<b>VSH INSTALLATION.....</b>	<b>7</b>
	Installation of VSH .....	7
	Installing the VSH Source .....	7
	Configuration VSH .....	8
	Target Specific Configuration .....	9
	Building VSH .....	11
	Installation .....	11

---

<b>CHAPTER 3</b>	<b>THE VSH SHELL .....</b>	<b>13</b>
	Introduction .....	13
	Running the VSH Shell .....	13
	The <code>.vshrc</code> File .....	14
	Command Line Editing .....	15
	Linking with the VSH Shell .....	16
	The VSH Shell Local Symbol Table .....	16
	Extending the VSH Shell Parser .....	17
	General Notes .....	17
	VSH Shell Commands .....	17
	Call Subroutine .....	18
	Display Memory .....	18
	Help.....	19
	Remaping Memory .....	20
	Remove Mapped Memory Entry.....	21
	Display Mapped Memory .....	22
	Lookup Symbol.....	22
	Log to File.....	23
	Run Script File.....	23
	Set Memory.....	24
	Sleep.....	25

---

<b>CHAPTER 4</b>	<b>THE VSH AGENT.....</b>	<b>27</b>
	Introduction .....	27
	Example Script .....	28
	Running the VSH Agent .....	30

---

VSH Agent Interface .....	31
Opening a Session.....	32
Closing a Session.....	33
Symbol Lookup.....	33
Remaping Memory .....	34
Fetching Memory.....	34
Setting Memory.....	35
Terminate Agent .....	36



---

## *Preface*

The *VSH Users Guide* contains installation and usage information for the VSH Environment. This manual assumes that you have a basic understanding of Linux. It is intended primarily for system developers, device driver developers, and application developers working with on an embedded Linux project. Certain tasks in this manual require `root` privilege or other information that typically falls in the system administration domain.

---

## **For More Information**

For more information on the VSH Environment, please refer to the following documentation shipped with VSH.

- *VSH Release Notes*  
This document contains late-breaking information about the release.
- *VSH Manual Pages*  
Contains online reference for the VSH Environment.

## Typographical Conventions

Typefaces used throughout this guide emphasize important concepts. All references to file names and commands are case sensitive and should be typed exactly as shown. These conventions are summarized below.

Description	Example(s)
Commands (Courier New 9 pts.)	<code>ls</code>
Command options (Courier New 9 pts.)	<code>-l</code>
Example code (Courier New 7 pts.)	<code>Put example code in HERE!!!</code>
Filenames (Courier New 9 pts.)	<code>myprog.c</code>
Pathnames (Courier New 9 pts.)	<code>/dev/null</code>
Environment variables (Verdana 9 pts.)	<code>DISPLAY</code>
Keyboard options, buttons & menu sequences (Verdana Bold 9 pts.)	<b>Enter</b>
Keyboard options that must be performed in sequence (Verdana Bold 9 pts.)	<b>Ctrl-C</b>
Function names (Verdana Italics 9 pts.)	<i>getenv()</i>
Words or characters that you type on your computer (Courier New 9 pts. Bold)	<b>vsh -v -d /dev/vsh</b> <b>vsh_agent -v -s</b>

---

Description	Example(s)
Words that you must replace with a real name or value appear in <i>italics</i> .	<code>cat <i>filename</i></code> <code>mv <i>file1 file2</i></code>
New terms, book titles, and emphasized words appear in Garamond <i>italics</i> .	Refer to the man page <i>vsh_agent</i> .

---

## Special Notes

The following notations highlight key points and cautionary notes in this guide.

---

**NOTE:** *These callouts note important points in the text.*

---



**CAUTION!** *Used for situations that present minor hazards that may interfere with or threaten equipment/performance.*

---

## Technical Support

Technical support is available via email. Please contact [vsh\\_support@tbcorp.com](mailto:vsh_support@tbcorp.com).

### World Wide Web

<http://www.tbcorp.com>

### Thomas Besemer Consulting

[tbesemer@tbcorp.com](mailto:tbesemer@tbcorp.com)  
Phone: (408) 307-1674



---

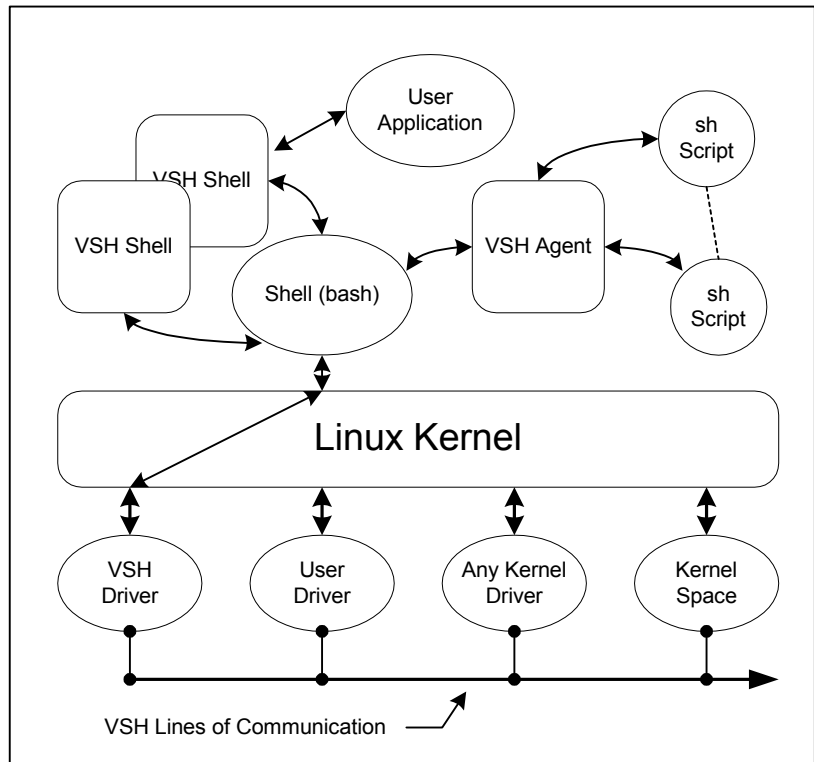
## Introduction

VSH is a interactive shell and environment that provides embedded Linux developers a robust set of function for interacting with kernel level modules and devices from the application level. For developers familiar with the VxWorks environment from Windriver, they will find that VSH is very similar to the VxWorks shell. The primary services supported with the VSH environment are:

- The ability to remap memory space in the kernel from application space.
- The ability to set and display memory in the kernel from application space.
- The ability to look up symbols by name and address in kernel space from application space.
- The ability to link VSH with the users application, and perform symbol lookups in the users application, and do set and display memory operations on them.

All of these operations may be done from within VSH, which executes as a standard linux application, or via standard sh scripts. Additionally, VSH natively supports basic scripting, allowing batch type operations to be invoked to operate on kernel memory space.

Figure 1-1 depicts the relationship between VSH and other Linux resources.



**Figure 1-1: VSH and Linux**

As shown in figure 1-1, VSH is a set of resources that, at its most basic level, allow application level code or `sh` script to communicate with kernel level resources.

The most viable usage of the VSH environment is to aid in development and debug of device drivers. Modern devices are often complicated sub-systems with large register sets; through VSH, the developer can rapidly configure and read back registers in these devices, and interact with the devices during the development of custom device drivers. In many instances, the developer will find that they do not need to write a large and complicated device driver, but rather, simply use scripting with VSH to manage their device.

Another key advantage of the VSH environment is to debug new or existing drivers. During normal runtime of their embedded Linux environment, they are

able to lookup up symbols by name, remap physical addresses and then perform basic display and set operations on this kernel memory space. This facility provides an easy way to dump buffers, dump register sets and change the configuration of kernel based sub-systems during normal runtime.

---

## Key VSH Components

The VSH Environment is comprised of three major components:

- The VSH Device Driver.
- The VSH Shell.
- The VSH Agent.

All of these components are supplied in source code format, and may easily be extended and customized by the developer.

### The VSH Device Driver

The VSH Device Driver is a standard Linux character device driver. It installs as a module, and provides services to the application level code for the following operations:

- Remapping of memory using the kernel *ioremap()* system call.
- Mapping of “raw” addresses to be operated on.
- Setting and displaying of kernel memory space.

It is interacted with using standard *open()*, *close()*, *read()* and *ioctl()* calls. A standard device file, typically called `/dev/vsh`, exists in the `/dev` directory to support these system calls.

The driver is constructed in such a way that it allows multiple, independent sessions (opens) to exist simultaneously. Additionally, an interface library is provided with the driver which may be linked with the users application, providing a set of API's to interact with the driver.

### The VSH Shell

The VSH Shell is an application program invoked from a standard bash prompt. It provides the following key services:

- Symbol lookup by name or address for kernel symbols.
- Symbol lookup by name or address for application code symbols, if the developer links VSH with their application.
- The ability to either remap or allow raw kernel addresses for subsequent operations.
- The ability to set and display memory in kernel space.
- The ability to set and display memory in application space, if the developer links VSH with their application.
- The ability to log all transactions to a log file.
- The ability to run batch operations through simple scripts which contain VSH commands.
- Support for basic command line editing, using simple ‘vi’ type commands.

The VSH Shell is based on a flex and bison based parser, and may easily be extended with new commands. Additionally, provisions exist for linking the shell with the users application, allowing interaction with it. In this configuration, developers are able to extend the VSH command set, providing a robust CLI for their own application.

## The VSH Agent

The VSH Agent is a daemon and a set of support binaries that allows the developer to interact with the VSH Device Driver through standard `sh` script. Typically, it is started during normal system start-up, and persists through the uptime of the system. Through the VSH Agent, using standard `sh` script, the developer is able to:

- Remap kernel addresses, as well as allow raw kernel addresses to be operated on.
- Lookup kernel symbols by name or address.
- Set and display memory in kernel space.

Although the VSH Shell provides basic batch type operations of VSH Shell commands, it does not provide the more robust conditional constructs supported by native `sh` scripting. Through the VSH Agent, the developer is able to construct complex scripts to configure devices and/or memory in kernel space, and evaluate status using all native `sh` type operations or other tools such as `awk`. With the complexity of modern devices, this capability allows the

developer to rapidly prototype up device configurations, and evaluate the status of their devices during normal runtime.

The VSH Agent supports the concept of opening and closing the VSH Device Driver, thus supporting the ability for multiple scripts to execute simultaneously.

---

## VSH Users Guide Examples

Throughout this guide, numerous examples are provided on how to use the various functions of VSH. In order to provide consistency in the guide, the VSH Device Driver contains two memory locations, and one procedure, which are used in the examples. Since these examples exist in the VSH Device Driver, they are part of kernel space.

### VSH Example Kernel Memory Declarations

```
int vshScratchBuffer[ 0x1000 ];
char *vshTestString = "this is a test";
```

These are used in examples which set and display memory in kernel space.

### VSH Example Kernel Procedure

```
int vsh_diag()
{
    printk( "vsh_diag(): called, vshTestString 0x%x\n"
           (int)vshTestString );
}
```

This is used in the examples which invoke procedures in the kernel from application space.

### How They are Used

An example of how VSH is used to display the contents of `vshTestString` is shown below:

```
vsh -> lkup vshTestString
  Address      Name
-----
0xC887AF88,  vshTestString  [vsh_module]

vsh -> allowraw 0xc887af88 0x100
allowraw: Physical 0xC887AF88 allowed
```

```
vsh -> dm 0xc887af88
C887AF88: C887A8FB 00000000 00000000 00000000 .....
C887AF98: 00000000 00000000 00000004 C887AF88 .....
C887AFA8: 00000000 C887AF88 C887AF88 00000100 .....
C887AFB8: 00000001 00000000 00000000 00000000 .....

vsh -> allowraw 0xc887a8fb 0x100
allowraw: Physical 0xC887A8FB allowed

vsh -> dmb 0xc887a8fb
C887A8FB: 74 68 69 73 20 69 73 20 this is
C887A903: 61 20 74 65 73 74 00 00 a test..
```

Allow the specific commands used above are detailed in Chapter 2, the basic operations were invoked in the above example are:

- The VSH Device Driver symbol `vshTestString` address is looked up in the symbol table, and is mapped in as a “raw” address (meaning that it can be read and written using it’s actual value).
- The contents of this symbol are displayed. Since the symbol is a pointer, the first element of the display is the value of the pointer.
- The address of the pointer is mapped in as a “raw” address, and then displayed, showing the string contents.

Note that these operations were done in the VSH Shell, which runs as a user application. Although these diagnostic declarations in the VSH Device Driver are incredibly simple, they serve well to show the power of VSH; from the VSH Shell, kernel level information is displayed without the need for complex debugger configurations. And just as easily, `sh` script could have been employed in conjunction with the VSH Agent to accomplish the same task.

In this particular case, the interactions were logged to a file using the VSH Shell logging facility, and imported into this users guide.

---

## Installation of VSH

VSH is supplied in source code format on a CD-ROM. Contained on this CD-ROM are the following primary components:

- The source code for all of VSH.
- A `Makefile` which is used to build VSH.
- This Users Guide, in PDF format.
- A set of man pages for all of the VSH services.

In order to properly install VSH, you will need **root** privileges, as the installation requires that a new device file be created in `/dev`, the manual pages be installed in their correct location, and the support binaries be installed in `/usr/local/bin` (this location may be changed by editing the `Makefile`). Once installed, the developer does not need require further **root** privileges.

### Installing the VSH Source

The CD-ROM must first be mounted, typically as follows:

```
mount /dev/cdrom
```

Then, the sources must be copied into a user specific directory. From that directory, enter the following;

```
cp -Rp /mnt/cdrom .
```

Note that if your CD-ROM mounts at a different location than `/mnt/cdrom`, you will need to use that location instead.

## Configuration VSH

There are two key elements to configuring VSH:

- General, processor independent configuration.
- Processor specific information.

## General Configuration

The general configuration is maintained in the file `<~src/vsh_config.h>`. In most cases, this file will not require changes. Below shows the contents of this file:

```
/*
 * vsh - Linux Kernel and Application Debug Shell
 *
 * Copyright (c) 2000-2003 Thomas E. Besemer.
 * All rights reserved.
 *
 * This file, and all files associated with vsh, may not be
 * redistributed in any form without prior written consent
 * of Thomas E. Besemer.
 *
 * tbesemer@tbcorp.com
 * http://www.tbcorp.com
 */

#ifndef VSH_CONFIG_H
#define VSH_CONFIG_H

#undef VSH_DEBUG
#define VSH_DEVICE_NAME "/dev/vsh"
#define VSH_VSHRC_NAME ".vshrc"
#define VSH_MAX_SESSIONS 4
#define VSH_IOREMAP_MAX_ENTRY 32

#endif
```

Table 2-1 describes each of the entries in this configuration file.

**Table 2-1: VSH Basic Configuration Variables**

Configuration Entry	Explanation
VSH_DEBUG	Either can be defined, or undefined. When defined, the VSH Driver is compiled to operate very verbosely. Typically only used when making changes to the driver.
VSH_DEVICE_NAME	The default name of the entry in /dev. If you change this, you must also edit the Makefile.
VSH_VSHRC_NAME	The name of the file read when the VSH Shell is invoked.
VSH_MAX_SESSIONS	The maximum number of open sessions with the VSH Device Driver.
VSH_IOREMAP_ENTRY	The total number of active memory addresses which can exist at any time.

Under most conditions, these values are viable as shipped.

## Target Specific Configuration

Most typical embedded Linux applications are cross compiled from a native x86 host, to a target processor, such as the PowerPC. Depending upon your environment, you will need to update the `Makefile`, contained in the parent VSH directory. The key elements that require customization are:

- Cross tool specification.
- Driver level flags for the target kernel. These flags need to point to the source code for the target kernel.

## Cross Tool Specification

Shown below is the entry in the Makefile for specifying the tools which are used to build VSH:

```
# Leave blank for x86, else supply name of tools. ie, "powerpc-linux-"
#
CROSS=
AS_MOD=$(CROSS)as
```

```
CC_MOD=$(CROSS) gcc
LD_MOD=$(CROSS) ld

CC=$(CROSS) gcc
LD=$(CROSS) ld
```

Typically, the only variable that needs configuration is CROSS. However, as seen in the subsequent lines, the capability exists to user a different compiler for the VSH Device Driver module versus the VSH application code.

## VSH Device Driver Flags

These are used to compile the VSH Device Driver, which is compiled as a Linux module. Shown below are two examples, one for x86, and one for PowerPC.

```
# PowerPC Flags
#
KERNEL_BASE=/projects/elrg/src/yellowdog/linux-2.4.19
GCC_INC=/projects/elrg/local/lib/gcc-lib/powerpc-linux/2.95.3/include
CPU_FLAGS="-m405"
ARCH=ppc
MOD_CFLAGS = -D_KERNEL -I$(KERNEL_BASE)/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -fomit-
frame-pointer -I$(KERNEL_BASE)/arch/$(ARCH) -fsigned-char -msoft-float -
pipe -ffixed-r2 -Wno-uninitialized -mmultiple -mstring -Wa, $(CPU_FLAGS)
-DMODULE -nostdinc -I $(GCC_INC) -DKBUILD_BASENAME=vsh -DEXPORT_SYMTAB

# x86 Flags
#
KERNEL_BASE=/projects/src/linux-2.4.18
GCC_INC=/usr/lib/gcc-lib/i386-redhat-linux/2.96/include
CPU_FLAGS=
ARCH=i386
MOD_CFLAGS = -D_KERNEL -I$(KERNEL_BASE)/include -Wall -Wstrict-
prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -fomit-
frame-pointer -I$(KERNEL_BASE)/arch/$(ARCH) -fsigned-char -msoft-float -
pipe -Wno-uninitialized -Wa, $(CPU_FLAGS) -DMODULE -nostdinc -I
$(GCC_INC) -DKBUILD_BASENAME=vsh -DEXPORT_SYMTAB
```

The key configuration changes the developer need to worry about are shown below, in Table 2-2:

**Table 2-2: VSH Target Specific Flags**

Variable	Description
KERNEL_BASE	Points to the base for the source code of the target kernel.
GCC_INC	The include directory for the cross tool.
CPU_FLAGS	Any processor specific flags.

**Table 2-2: VSH Target Specific Flags**

Variable	Description
ARCH	Defines the processor architecture.
MOD_CFLAGS	Defines any specific options for the particular architecture of the target processor.

The current release of VSH has been tested under x86 and PowerPC. However, there are no specific ties in the code to either processor architecture.

Often, a good way to find out what specific flags you need for your particular architecture is to build your kernel, and then execute the following:

```
make modules
```

This will cause the modules to be built for your target kernel and processor, and shows clearly the variables that need to be used in the above configurations.

## Building VSH

If properly configured, the user should be able to build all of the VSH components simply by typing:

```
make
```

At the root install directory.

Optionally, the user may wish to link the VSH Shell with their own application (see Chapter 3 - The VSH Shell). To do this:

```
make -f Makefile -DVSH_EXTRA=user_object_file.o
```

The file <user\_object\_file.o> must be an incrementally linked object, containing all of the users application. It must not have *main()* declared in it.

## Installation

Once all the VSH files have been built, VSH may be installed on the local host by typing:

```
make install
```

This operation will do the following:

- Test to see if `/dev/vsh` exists; if not, it will be created.
- Installation of the VSH Agent and VSH Shell in `/usr/local/bin` (if you desire an alternate location, visit the Makefile and change the variable `VSH_INSTALL_LOCATION`).

Once this is done, VSH is fully operational on the local host. This operation assumes self-hosting.

For cross-development, the files will need to be installed slightly differently. For cross-targets which mount their root filesystem via NFS, you may install in this fashion:

```
make install -DVSH_INSTALL_DIR=<your target NFS root>
```

The target binaries are placed in the correct directories, based on the parameter `VSH_INSTALL_DIR`.

For ROM based images, the developer must copy the binary files located in `<~bin>` over to the location where their ROM based images are built from.

## Man Page Utilization

A set of reference man pages exist for VSH. For a typical installation, the developer must modify their environment to allow the `man` command to find them. For Redhat 7.3, the following addition must be made to `/etc/man.config`:

```
MANPATH <vsh_install_root/man>
```

Please refer to the man page on `man` for correct configuration.

Once your environment is correctly configured, you may type `'man vsh'` to get the list of available man pages.

---

## Introduction

The VSH Shell is a Linux application that is invoked from a standard linux bash prompt. When it runs, it opens the VSH Device Driver in preparation for user interaction with memory within the kernel, and when linked with the users application, with user memory.

Multiple VSH Shell sessions may exist simultaneously. Each session maintains it's own state, in the VSH Device Driver.

A configuration file may exist in the directory which VSH is invoked from. This file, `<.vshrc>`, can contain any of the standard VSH Shell commands, and is executed when the VSH Shell is first started.

---

## Running the VSH Shell

When using a standard configuration, the VSH Shell is started by entering the command **vsh**. Optionally, a `'-h'` flag may be supplied, which will cause the VSH Shell to display the start options, as shown below:

```
[tbesemer@icfv-server vsh]$ vsh -h
usge: vsh -fhdv
```

```
-f filename # Override .vshrc for startup
-d filename # Override /dev/vsh for Device File
-a         # Run VSH even if Device File open fails
-v         # Operate Verbose
-h         # Display startup options
```

```
[tbesemer@icfv-server vsh]$
```

Note that the user has the option of overriding the default `/dev/vsh` file `/dev/vsh` with the `-d filename` parameter, as well as overriding the default `<.vshrc>` file with the `-f filename` parameter.

Once running, the user is free to use all available VSH Shell commands. As shown below, the shell is started, the `help` command is invoked, and then the session is terminated through the `quit` command.

```
[tbesemer@icfv-server vsh]$ vsh
vsh: can't stat .vshrc
vsh -> help

dm[b|w] address count      - Display Memory
sm[b|w] address value     - Set Memory
ioremap address length    - Remap a Physical Address (ioremap)
iounmap address          - Unmap a Physical Address (iounmap)
ioshowmap <optional address> - Display vsh ioremap'd Addresses
allowraw address         - Allow a "RAW" address, such as
                          a Kernel Address
sleep seconds            - Sleep for <seconds>
lkup [local] symbol      - Lookup a symbol
call [local] symbol [params] - Call a subroutine for execution
run filename             - Execute script <filename>
quit                    - Exit vsh
help                    - Shows this list

- Note that hex numbers are in the form 0x<value>.
- After doing an ioremap, the original physical address may be used.

vsh -> quit
[tbesemer@icfv-server vsh]$
```

In this example, a `<.vshrc>` file did not exist, and diagnostic output simply noted this.

## The `.vshrc` File

The `<.vshrc>` file is a text file which is read during start-up. It may contain a list of standard VSH Shell commands that will be executed in a linear, batch type operation. This capability allows the developer to cause the VSH Shell to start-up with a pre-configured configuration.

Note that using the `-f filename` option when invoking the VSH Shell will cause the file `<filename>` to be read, instead of the standard `<.vshrc>` file. This is useful when the developer has several different configurations the wish to use.

## Command Line Editing

The VSH Shell provide simple, `vi` like command line editing, similar to what bash supports when the `set -o vi` option is used. Command line editing mode is entered just as you would in `vi`, through hitting the escape key. Table 3-1 provides a summary of the standard command line editing commands supported in the VSH Shell.

**Table 3-1: Editing Command Summary**

Command	Operation
<code>esc</code>	Enter command line edit mode.
<code>k</code>	Scroll back one line.
<code>j</code>	Scroll forward one line.
<code>w</code>	Space forward one word in line.
<code>b</code>	Space back one word in line.
<code>cw</code>	Change word, entering standard command entry mode.

Entering a carriage return at any time causes the currently displayed command to be executed. Entering **CTL-C** at any time terminates the operation, returning the user to the VSH Shell prompt.

## Linking with the VSH Shell

The VSH Build environment easily facilitates linking of the developers application code with the VSH Shell. There are two advantages in linking the developers application with the shell:

- The VSH Shell memory operations can operate on the developers application. For example, subroutines in the developers application may be invoked from the VSH Shell prompt, or memory locations may be set or modified.
- The VSH Shell parser can be extended to provide a custom CLI for the developers application.

In order to link with the VSH Shell, the developer must configure their build environment to produce a final, incrementally linked object file. Then, the VSH Makefile is invoked to link the application with the VSH Shell. This linking is done as follows:

```
make -f Makefile vsh -DVSH_EXTRA=user_app.o
```

This causes the final build of the VSH Shell to contain the users application code (in this case, called <user\_app.o>).

### The VSH Shell Local Symbol Table

The VSH Shell contains a local symbol table that is automatically generated during the final build of the shell. This local symbol table contains all public data and procedures in both the VSH Shell, and the users application.

This symbol table is a compiled in array that contains the name of the symbol, and it's address. The shell command **lkup local** may be used to search for these symbols, either by name, or address. The shell command **call** may be used to invoke any of the procedures in the symbol table, with up to 10 arguments.

## Extending the VSH Shell Parser

The VSH Shell parser is constructed using **flex** and **bison**. The logic for these is contained in the files `<vsh_lex.l>` and `<vsh_bison.y>`. Through these two files, the developer may add their own keywords, and then their own syntax for parsing of the keywords. Examples of how commands are invoked are shown in `<vsh_bison.y>`, as well as through their associated handling routines, contained in `<vsh_core.c>`.

## General Notes

The file `<vsh_lex.l>` contains *main()*, the entry point into the VSH Shell. The users application must not have a declaration of *main()* when linking with the shell. Typically, the user will have one or more entry points, such as *user\_entry1()*, which is then called from the VSH Shell command prompt using the call command, or from a command in the `<.vshrc>` file. Any calls to initialize the users application must return to VSH, so that normal VSH Shell operations can occur.

---

## VSH Shell Commands

The section details the usage of all VSH Shell commands. Referring back to Chapter 1, all examples are centered around two data declarations, and one procedure declaration contained in the VSH Device Driver. These declarations serve as public symbols that reside in kernel space, providing clear examples of how to interact with data and procedures in the kernel.

These examples are found at the end of `<vsh_drvr.c>`, and are declared as shown below:

```
int vshScratchBuffer[ 0x1000 ];
char *vshTestString = "this is a test";
```

These are used in examples which set and display memory in kernel space.

```
int vsh_diag()
{
    printk( "vsh_diag(): called, vshTestString 0x%x\n"
           (int)vshTestString );
}
```

This is used as an example of a kernel based procedure that may be called from the VSH Shell.

---

## Call Subroutine

### Syntax

```
call [local] symbol [p0-p9]
```

### Description

Calls a subroutine for execution in the kernel, with up to ten parameters. The parameters p0 through p9 may be decimal, hexadecimal or strings. If strings, the pointer to the string is passed to the subroutine when it is invoked. The return value from the subroutine is printed at the end of invocation. Since any diagnostic output from the subroutine called would generally be done using *printk()*, that output will be displayed on the system console. If the optional specifier **local** is supplied, the symbol lookup is done from the local symbol table. An example of usage:

```
call vsh_diag 0 0
call local user_init1 0x1234 0 "start"
```

---

## Display Memory

### Syntax

```
dm [address] [count]
dmw [address] [count]
dmb [address] [count]
```

### Description

Used to display memory space within the kernel. No checking is made to see if the supplied address is valid. The parameter address may be either a hexadecimal number, or a symbol name contained in the kernel Symbol Table. If a hexadecimal number, it must previously have been mapped using either the **ioremap** or **allowraw** VSH commands. If no address is specified, memory is displayed at the current position (defined from the last invocation of one of the Display Memory commands). Memory may be displayed in standard long word format, word format or byte format.

The optional parameter count is a hexadecimal number which specifies the number of elements to display. The default is 0x10. An example of output from the Display Memory functions:

```

vsh -> lkup vshTestString
  Address      Name
-----
  0xC887AF88,  vshTestString  [vsh_module]

vsh -> allowraw 0xc887af88 0x100
allowraw: Physical 0xC887AF88 allowed

vsh -> dm 0xc887af88 0x01
C887AF88:  C887A8FB  ....

vsh -> allowraw 0xc887a8fb 0x100
allowraw: Physical 0xC887A8FB allowed

vsh -> dmb 0xc887a8fb
C887A8FB:  74 68 69 73 20 69 73 20  this is
C887A903:  61 20 74 65 73 74 00 00  a test..

```

In this example, the following actions were taken:

- The address of the symbol `vshTestString` was looked up using the **lkup** command.
- It's address was added to environment using the **allowraw** command.
- The value contained at this location was displayed using the **dm** command. This value represents a pointer.
- The pointer value was added to the environment using the **allowraw** command, and then it's value was displayed using the **dmb** command.

This very typical of how the user will operate on kernel memory; first, the address is located. Then it is added to the environment, and finally, the contents of it are displayed.

Any kernel address that is public may be operated on in this fashion, including PCI addresses, which are added to the environment using the **ioremap** command, versus the **allowraw** command. Refer to the section on *Remapping Memory* for additional details on adding addresses to the VSH Shell environment.

---

## Help

### Syntax

```
help [topic]
```

## Description

If invoked without options, displays a list of all subjects for which help is available. If a topic is selected, displays detailed information about the topic.

Example output:

```
vsh -> help
dm[b|w] address count      - Display Memory
sm[b|w] address value     - Set Memory
ioremap address length    - Remap a Physical Address (ioremap)
iounmap address           - Unmap a Physical Address (iounmap)
ioshowmap <optional address> - Display vsh ioremap'd Addresses
allowraw address          - Allow a "RAW" address, such as
                           a Kernel Address
sleep seconds             - Sleep for <seconds>
lkup [local] symbol       - Lookup a symbol
call [local] symbol [params] - Call a subroutine for execution
run filename              - Execute script <filename>
quit                      - Exit vsh
help                      - Shows this list

- Note that hex numbers are in the form 0x<value>.
- After doing an ioremap, the original physical address may be used.
```

---

## Remapping Memory

### Syntax

```
ioremap address size
allowraw address size
```

### Description

The commands `ioremap` and `used` are used to re-map any kernel address for subsequent display or modification. Typically, this call is used to map physical addresses, such as PCI addresses, into kernel space. The hexadecimal parameter `address` specifies the base address of the region to re-map. The parameter `size` specifies the total length of the region, in bytes. Once the `ioremap` has been done, the original address may be used for subsequent operations.

The slightly different command, `allowraw`, simply inserts a raw address into the internal VSH Device Driver table, allowing it to be operated on with its “raw” value. Typically, this is used for kernel memory addresses.

### Using `ioremap`

Below is an example of mapping a PCI address into the environment. The first fragment shows the output from the native Linux command `lspci`, using the verbose (`-v`) option (note that output was truncated to show only one entry). The second fragment shows the usage of the `ioremap` command, and then is followed by examples of displaying the current mapping, and displaying the contents of the address.

## Finding the PCI Device Address:

```
[root@icfv-server vsh]# lspci -v
.....
01:09.0 Ethernet controller: Lite-On Communications Inc LNE100TX (rev 20)
  Subsystem: Netgear FA310TX
  Flags: bus master, medium devsel, latency 32, IRQ 11
  I/O ports at d800 [size=256]
  Memory at ff8ffc00 (32-bit, non-prefetchable) [size=256]
  Expansion ROM at ff880000 [disabled] [size=256K]
```

## Remapping and Displaying Contents:

```
vsh -> ioremap 0xff8ffc00 0x100
ioremap: Physical 0xFF8FFC00 mapped to 0xC8881C00 Virtual

vsh -> ioshowmap
  Physical      Virtual      Length
-----
0xC887AF88    0xC887AF88  0x00000100
0xC887A8FB    0xC887A8FB  0x00000100
0xC887B800    0xC887B800  0x00000100
0xFF8FFC00    0xC8881C00  0x00000100

vsh -> dm 0xff8ffc00
FF8FFC00:  00008000 00008000 01FF0000 01FF0000 .....
FF8FFC10:  00000000 00000000 0616E000 0616E000 .....
FF8FFC20:  0616E200 0616E200 02660010 02660010 .....f...f.
FF8FFC30:  810C2002 810C2002 0001FBFF 0001FBFF . . . . .
```

## Using allowraw

This example shows how to map in a raw kernel address (in this case, the example string `vshTestString`), and display the contents.

```
vsh -> lkup vshTestString
  Address      Name
-----
0xC887AF88,  vshTestString [vsh_module]

vsh -> allowraw 0xc887af88 0x100
allowraw: Physical 0xC887AF88 allowed

vsh -> dm 0xc887af88 0x01
C887AF88:  C887A8FB ....
```

## Remove Mapped Memory Entry

### Syntax

```
iounmap address
```

### Description

Used to un-map memory previously mapped with one of the IO Re-map functions. Used simply to free internal VSH resources. The parameter `address` must correspond to the original parameters supplied during the mapping.

## Display Mapped Memory

### Syntax

```
ioshowmap
```

### Description

Lists all memory addresses currently mapped in VSH. Example of usage:

```
vsh -> ioshowmap
Physical      Virtual      Length
-----
0xC887AF88   0xC887AF88   0x00000100
0xC887A8FB   0xC887A8FB   0x00000100
0xC887B800   0xC887B800   0x00000100
0xFF8FFC00   0xC8881C00   0x00000100
```

## Lookup Symbol

### Syntax

```
lkup [local] symbol_name
```

```
lkup [local] symbol_address
```

### Description

Searches the Kernel Symbol Table for either the string specified by **symbol\_name**, or all symbols near the value **symbol\_address**, and reports the results. The optional parameter **local** causes the VSH Shell to look in the local (application) symbol table. Lookup does not occur on an “exact match” of the supplied symbol name; rather, it returns results based on any string that contains the supplied **symbol\_name**.

For example, looking for the symbol ‘vsh’ finds the following:

```
vsh -> lkup vsh
Address      Name
-----
0xC887A1B0,  vsh_iounmap [vsh_module]
0xC887A6C4,  vsh_diag [vsh_module]
0xC887A170,  vsh_iounmap_all [vsh_module]
0xC887B800,  vshScratchBuffer [vsh_module]
0xC887A060,  vsh_get_kAddr [vsh_module]
0xC887A11C,  vsh_insert_raw [vsh_module]
0xC887A060,  __insmod_vsh_module_S.text_L1664 [vsh_module]
0xC887AF88,  vshTestString [vsh_module]
0xC887AF20,  __insmod_vsh_module_S.data_L128 [vsh_module]
0xC887AFA0,  __insmod_vsh_module_S.bss_L18528 [vsh_module]
0xC887A000,  __insmod_vsh_module_O/projects/src/vsh/vsh_module.o_M3E404B03_V132114
[vsh_module]
0xC887A24C,  vsh_showmap [vsh_module]
0xC887A1FC,  vsh_setmem [vsh_module]
0xC887A0B8,  vsh_ioremap [vsh_module]
```

In this example, all the public VSH symbols that exist in the VSH Device Driver are returned.

The symbol 'vshTestString' is located by having a fully qualified symbol name (exact match):

```
vsh -> lkup vshTestString
  Address      Name
-----
0xC887AF88,  vshTestString  [vsh_module]
```

A address may be supplied, and symbols near that address are displayed:

```
vsh -> lkup 0xc887af88
  Address      Name
-----
0xC887AF88,  vshTestString  [vsh_module]
0xC887AF20,  __insmod_vsh_module_S.data_L128  [vsh_module]
0xC887AFA0,  __insmod_vsh_module_S.bss_L18528  [vsh_module]
```

Finally, an example of looking up a local symbol (this symbol, yyerror, exists in <vsh\_lex.l>):

```
vsh -> lkup local yyerror
  Address      Name
-----
0x08049F20,  yyerror
```

---

## Log to File

### Syntax

```
log filename
```

### Description

Logs all VSH console activity to the file specified by filename.

---

## Run Script File

### Syntax

```
run filename
```

### Description

Causes all VSH commands in the file specified by filename to be executed. An example file:

Blank lines are ignored. The '#' character is treated as a comment escape sequence.

Example usage:

## Notes

Due to limitations in the VSH parser, script files may not invoke script files (nesting not allowed).

---

## Set Memory

### Syntax

```
sm address value [count]
smw address value [count]
smb address value [count]
```

### Description

The Set Memory commands are used to set or fill memory addresses in kernel space. The parameter address may be either a hexadecimal number, or a symbol which exists in the kernel Symbol Table. If a hexadecimal number, the address must have been previously mapped using one of the VSH mapping commands (see **ioremap** and **allowraw** commands). The value parameter is a hexadecimal number. The optional count field may be specified, in hexadecimal, to do block fills. An example of usage:

```
vsh -> lkup vshScratchBuffer
      Address      Name
-----
0xC887B800,  vshScratchBuffer  [vsh_module]

vsh -> allowraw 0xc887b800 0x100
allowraw: Physical 0xC887B800 allowed

vsh -> dm 0xc887b800
C887B800:  00000000 00000000 00000000 00000000  .....
C887B810:  00000000 00000000 00000000 00000000  .....
C887B820:  00000000 00000000 00000000 00000000  .....
C887B830:  00000000 00000000 00000000 00000000  .....

vsh -> sm 0xc887b800 0x12345678

vsh -> dm 0xc887b800
C887B800:  12345678 00000000 00000000 00000000  xV4.....
C887B810:  00000000 00000000 00000000 00000000  .....
C887B820:  00000000 00000000 00000000 00000000  .....
C887B830:  00000000 00000000 00000000 00000000  .....
```

In this example, the following was done:

- The symbol `vshScratchBuffer` is looked up using the **lkup** command.
- It's address is added to the environment using the **allowraw** command.
- It's contents are displayed, using the **dm** command.
- It's contents are modified, using the **sm** command.
- It's contents are then displayed again, confirming the change.

---

## Sleep

### Syntax

```
sleep seconds
```

### Description

This command causes the VSH Shell to sleep for the specified number of seconds. Typically, this is used in VSH scripts, allowing for a pause after operations such as setting memory.



---

## Introduction

The VSH Agent provides the developer with the ability to perform all standard VSH type functions from `sh` script. It is comprised of an agent that is typically run in the background, as a daemon, and a set of binary utilities which exist (typically) in `/usr/local/bin`. The binary utilities communicate with the agent using System V IPC mechanisms, and allow scripting to perform the following tasks:

- Lookup symbols, either in the kernel, or locally.
- Remap addresses, using either `ioremap`, or `allowraw` type commands (see Chapter 3, the VSH Shell).
- Set and display memory.

Although the VSH Shell provides simple scripting capability (batch operations of VSH Shell commands), it does not support more complex operations, such as looping or `if/else` constructs. Additionally, since the VSH Agent is best suited for `sh` scripting, the developer may take advantage of other native Linux facilities, such as `awk`, `sed`, arithmetic evaluations and such.

Through this functionality, the developer is able to rapidly prototype up scripting for configuration of complex I/O devices, as well as be able to read these I/O device registers, and perform tests on the results.

## Example Script

Shown below is a simple example of a shell script that performs the following basic operations:

- Launch the VSH Agent.
- Open a session with the VSH Agent.
- Perform a symbol lookup.
- Remap the symbol address to allow operations on it.
- Display and set memory for the symbol.
- Kill the agent

## Script Source

```
#!/bin/sh
#
# vsh - Linux Kernel and Application Debug Shell
#
# Copyright (c) 2000-2003 Thomas E. Besemer.
# All rights reserved.
#
# This file, and all files associated with vsh, may not be
# redistributed in any form without prior written consent
# of Thomas E. Besemer.
#
# tbesemer@tbcorp.com
# http://www.tbcorp.com
#
# Export to local path support (for testing).
#
export PATH=./:$PATH

# Launch the VSH Agent as a daemon
#
vsh_agent -b
if [ $? -ne 0 ]
then
    echo "vsh_agent failed, exiting"
    exit 1
fi

# Open a session with the agent.
#
VSH_FH=`vsh_open`
if [ $? -ne 0 ]
then
    echo "vsh_open failed, exiting"
    exit 1
else
    echo "vsh_open returned $VSH_FH"
fi

# Perform a symbol lookup of vshScratchBuffer, which exists
# in the Driver for testing.
#
VSH_TEST_ARRAY_ADDR=`vsh_lkup vshScratchBuffer`
if [ $? -ne 0 ]
```

```

then
    echo "vsh_lkup failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_lkup returned $VSH_TEST_ARRAY_ADDR"
fi

# Remap this symbol address as a raw address, available for
# subsequent operations.
#
vsh_ioremap -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -s 0x100 -r
if [ $? -ne 0 ]
then
    echo "vsh_ioremap failed, exiting"
    vsh_close $VSH_FH
    exit 1
fi

# Perform reads of byte, short and long words.
#
MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 1`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 2`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

# Change the value of the mapped address, read it back.
#
vsh_sm -v -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4 -s 0x98765432
if [ $? -ne 0 ]
then
    echo "vsh_sm failed, exiting"
    vsh_close $VSH_FH
    exit 1
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

```

```
# Close the session.
#
echo "closing $VSH_FH"
vsh_close $VSH_FH

# Kill the agent.
#
vsh_agent_exit
```

## Script Output

When invoked, this script executes, and displays the following output:

```
[tbesemer@icfv-server vsh]$ sh vsh_exmpl_script.sh
vsh_agent: running as Daemon
vsh_open returned 1
vsh_lkup returned 0xC887B800
vsh_dm returned 0x78
vsh_dm returned 0x5678
vsh_dm returned 0x12345678
Setting 0x98765432 with size 4 to 0xC887B800
vsh_dm returned 0x98765432
closing 1
[tbesemer@icfv-server vsh]$
```

This simple example clearly demonstrates the capability of the VSH Agent, used in conjunction with scripting. Simple text based script files are able to operate on kernel based memory.

---

## Running the VSH Agent

The VSH Agent may run in one of two configurations:

- In the foreground, like any Linux application.
- In the background, as a Linux daemon.

The agent has several command line variables which control it's operation. Running the agent with the **-h** flag dumps these parameters:

```
[tbesemer@icfv-server vsh]$ ./vsh_agent -h
-d /dev/<name> # Override default vsh driver device name.
-s           # When specified, stdio is used instead of syslog.
-b           # Background and run as a daemon
-v           # Run verbose
-h           # Dump this help list.
[tbesemer@icfv-server vsh]$
```

Table 4-1 describes each parameters operation and usage.

**Table 4-1: VSH Agent Command Line Parameters**

Parameter	Action
-d	Override the default device file <code>/dev/vsh</code> with the one specified by name.
-s	Used <code>stdio</code> , instead of <code>syslog</code> , for reporting diagnostic messages.
-b	Background upon start-up, running as a daemon.
-v	Run verbose; with this option, all requests and responses to and from the agent are logged to either <code>stdio</code> or <code>syslog</code> (based on the <code>-s</code> flag).
-h	Dump the help list, and exit.

Only once instance of the VSH Agent may exist at any time. During start-up, the agent performs a test to see if an agent is currently running, and if so, exits with an error code.

---

## VSH Agent Interface

The VSH Agent is communicated with through a set of binary utilities that typically exist in `/usr/local/bin` (during installation, the developer has the option to edit the `Makefile` and change the installation location). Table 4-2 provides a summary of these utilities:

**Table 4-2: VSH Agent Interface Summary**

Command	Operation
<code>vsh_open</code>	Open a session for subsequent operation. This results in a text “file handle” being generated, which is supplied on all further calls.
<code>vsh_close</code>	Close an open session.

**Table 4-2: VSH Agent Interface Summary**

Command	Operation
<code>vsh_lkup</code>	Lookup a symbol. If successful, the result is printed on stdout as a text string.
<code>vsh_ioremap</code>	Remap an address for subsequent use.
<code>vsh_dm</code>	Display memory.
<code>vsh_sm</code>	Set memory.
<code>vsh_agent_exit</code>	Cause the agent to terminate and exit.

Each of these calls exit with a status code, which indicates the success or failure of the operation. Non-zero return codes represent a failure. An example of how script may evaluate the results of an operation is shown below:

```
VSH_FH=`vsh_open`
if [ $? -ne 0 ]
then
  echo "vsh_open failed, exiting"
  exit 1
else
  echo "vsh_open returned $VSH_FH"
fi
```

Note that the `sh` parameter `?` contains the exit code. All other output, such as correct results, are printed on `stdout`. In this example, using `vsh_open`, the file handle associated with the open is stored in the `sh` parameter `VSH_FH`.

The following sections detail each of the available system calls to the agent, with examples how of they are used (also shown in the example script).

## Opening a Session

Prior to performing any operations with the VSH Agent, a session must be opened. This is similar in operation to a standard `open()` call on a file. If successful, a text based “session handle” is printed on `stdout`. No parameters are required.

### Syntax

```
vsh_open
```

### Example

```
# Open a session with the agent.
#
VSH_FH=`vsh_open`
if [ $? -ne 0 ]
```

```

then
    echo "vsh_open failed, exiting"
    exit 1
else
    echo "vsh_open returned $VSH_FH"
fi

```

The resulting file handle generated is used for all subsequent operations. The operation causes the agent to open the device file for the VSH Device Driver.

---

## Closing a Session

Used to close a session previously opened with **vsh\_open**.

### Syntax

```
vsh_close fh
```

### Example

```

# Close the session.
#
echo "closing $VSH_FH"
vsh_close $VSH_FH

```

This will cause the agent to close the device file for the VSH Device Driver.

---

## Symbol Lookup

This operates similar to the VSH Shell command **lkup**. The primary difference is that it returns the first match (versus the **lkup** command, which returns all symbols that contain the supplied symbol name string).

### Syntax

```
vsh_lkup symbol_name
```

### Example

```

# Perform a symbol lookup of vshScratchBuffer, which exists
# in the Driver for testing.
#
VSH_TEST_ARRAY_ADDR=`vsh_lkup vshScratchBuffer`
if [ $? -ne 0 ]
then
    echo "vsh_lkup failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_lkup returned $VSH_TEST_ARRAY_ADDR"
fi

```

If found, the address of the symbol is printed on stdout. Upon failure, the error code is set.

## Remapping Memory

This operates similar to the VSH Shell commands `ioremap` and `allowraw`. It inserts the address into the environment, allowing subsequent operations to be performed on the address.

### Syntax

```
vsh_ioremap -d handle -r -a addr -s size -v -h
```

### Options

vsh\_ioremap options:

```
-d handle      # Device File Handle from vsh_open
-r            # Do an addition of a raw address
-a addr       # Base address, in hex
-s size       # Region size, in hex
-v           # Run verbose
-h           # Help
```

### Example

```
# Remap this symbol address as a raw address, available for
# subsequent operations.
#
vsh_ioremap -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -s 0x100 -r
if [ $? -ne 0 ]
then
  echo "vsh_ioremap failed, exiting"
  vsh_close $VSH_FH
  exit 1
fi
```

In this case, the `-r` flag was supplied, causing the address simply to be inserted into the VSH environment. If the `-r` flag is omitted, an `ioremap()` occurs.

## Fetching Memory

This operates identical to the VSH Shell command `dm[w|b]`. It causes the contents of the specified address to be displayed on stdout. The address must previously have been added to the environment through `vsh_ioremap`.

### Syntax

```
vsh_dm -d handle -a addr -w [1|2|4] v -h
```

### Options

vsh\_dm options:

```
-d handle      # Device File Handle from vsh_open
-a addr       # Base address, in hex
-w [1|2|4]    # Width of operation
-v           # Run verbose
-h           # Help
```

### Example

```

# Perform reads of byte, short and long words.
#
MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 1`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 2`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4`
if [ $? -ne 0 ]
then
    echo "vsh_dm failed, exiting"
    vsh_close $VSH_FH
    exit 1
else
    echo "vsh_dm returned $MEM_DATA"
fi

```

Note how the **-w** flag controls the width of the fetch operation; byte, short or long word.

## Setting Memory

This operates identical to the VSH Shell command **sm[w|b]**. It causes the contents of the specified address to be set. The address must previously have been added to the environment through **vsh\_ioremap**.

### Syntax

```
vsh_sm -d handle -a addr -w [1|2|4] -s value -v -h
```

### Options

vsh\_sm options:

```

-d handle      # Device File Handle from vsh_open
-a addr        # Base address, in hex
-w [1|2|4]    # Width of operation
-s value       # Value to set, in hex
-v            # Run verbose
-h            # Help

```

### Example

```

# Change the value of the mapped address, read it back.
#
vsh_sm -v -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4 -s 0x98765432

```

```
if [ $? -ne 0 ]
then
  echo "vsh_sm failed, exiting"
  vsh_close $VSH_FH
  exit 1
fi

MEM_DATA=`vsh_dm -d $VSH_FH -a $VSH_TEST_ARRAY_ADDR -w 4`
if [ $? -ne 0 ]
then
  echo "vsh_dm failed, exiting"
  vsh_close $VSH_FH
  exit 1
else
  echo "vsh_dm returned $MEM_DATA"
fi
```

In this example, the value was first set, then read back for confirmation. Like **vsh\_dm**, the **-w** flag controls the width of the operation.

---

## Terminate Agent

This call is used to cause the agent to exit.

### *Syntax*

```
vsh_agent_exit
```