Publication No. REF-806-950-R00

FastApp Utility Reference Manual

797 North Grove Rd, Suite 101
Richardson, TX 75081
Phone: (972) 671-9570
www.redrapids.com

Table of Contents

1.0 Introduction 3
1.1 Contents and Structure 3
1.2 Conventions 3
1.3 Distribution Disk (DSK-806-950-Rxx) 4
1.4 Revision History 4
2.0 FastApp Execution 5
2.1 Alias File (-alias) 5
2.2 Command File (-cmd) 6
2.3 Interrupt Service Routine File (-isr) 7
2.4 Interpreter (-run) 7
2.5 Code Generator (-code) 7
2.6 Verbosity (-v) 7
2.7 Application Examples 8
3.0 FastScript Language 9
3.1 Operators 9
3.1 Data Types 10
3.2 Variables 10
3.3 Instructions 10

List of Figures

Figure 2-1 Typical Alias File 5
Figure 2-2 Typical Command File 6

1.0 Introduction

The FastApp utility is based on a scripting language that simplifies the development of software applications for Red Rapids hardware products. The FastScript language encapsulates low level hardware operations in an intuitive set of commands. This language powers the two primary FastApp tools:

- An interpreter that can execute FastScript commands on target hardware without invoking a compiler or linker.
- A code generator that converts a FastScript command file to a FastCode C language source file that can be by compiled into a standalone application.

The FastApp utility is particularly useful during the development of custom application logic on a Red Rapids FPGA based product. New FPGA functions can be easily tested by just writing a few lines of FastScript commands. There is no longer a need to have a C compiler available to exercise the hardware from the host computer.

1.1 Contents and Structure

This manual describes the operation of the FastApp utility and the features of the FastScript language. This publication supplements other hardware and software documents that are available for specific Red Rapids PCI adapter products.

Section	**Description**
Section 1	Introductory information about the manual.
Section 2	Operating instructions for the FastApp utility, including the interpreter and code generator.
Section 3	Details of the instruction set and operators available in the FastScript language.

The latest product documentation and software is available for download from the Red Rapids web site (www.redrapids.com) by following links on the product page.

1.2 Conventions

This manual uses the following conventions:

- Hexadecimal numbers are prefixed by “0x” (e.g. 0x00058C).
- *Italic* font is used for names of registers.
- Blue font is used for names of directories, files, and OS commands.
- Green font is used to designate source code.

☞	Text in this format highlights useful or important information.

!	Text shown in this format is a warning. It describes a situation that could potentially damage your equipment. Please read each warning carefully.

The following are some of the acronyms and abbreviations used in this manual.

-32	32-bit Operating System
-64	64-bit Operating System
API	Application Program Interface
BAR	Base Address Register
DLL	Dynamic Link Library
DMA	Direct Memory Access
ISR	Interrupt Service Routine
Rxx	Any Revision Number
x86	Intel Processor Architecture

1.3 Distribution Disk (DSK-806-950-Rxx)

The FastApp utility is distributed on Red Rapids disk archive number DSK-806-950-Rxx. The directory structure of the archive is outlined below:

\include

The include subdirectory contains source code files to the FastApp API. These are needed to compile a FastCode application.

\linux_x86-32

The linux_x86-32 subdirectory contains the FastApp executable for all 32-bit Linux distributions hosted on an Intel processor architecture.

\linux_x86-64

The linux_x86-64 subdirectory contains the FastApp executable for all 64-bit Linux distributions hosted on an Intel processor architecture.

\win_x86-32

The win_x86-32 subdirectory contains the FastApp executable for all 32-bit Windows operating systems hosted on an Intel processor architecture.

\win_x86-64

The win_x86-64 subdirectory contains the FastApp executable for all 64-bit Windows operating systems hosted on an Intel processor architecture.

1.4 Revision History

Version	Date	Description
R00	7/5/2011	Initial release.

2.0 FastApp Execution

The FastApp utility is executed from a Windows or Linux command prompt as follows:

fastapp [options]

-alias <filename>	:	Name of the optional alias file (not required).
-cmd <filename>	:	Name of the primary FastScript command file (required).
-isr <filename>	:	Name of the interrupt service routine FastScript file (not required).
-run	:	Execute the FastScript source (not required).
-code	:	Generate FastCode from the FastScript source (not required).
-v <n>	:	Specifies verbosity of reporting during run time (n = [0:5], default = 0).

Multiple switches are allowed on the command line. The FastApp utility can be executed without the –run or –code switch to perform syntax error checking on all of the FastScript source files (alias, cmd, isr). Each filename may contain up to sixty characters including any extension. A detailed discussion of each switch is provided in the following sections.

2.1 Alias File (-alias)

The FastApp alias file provides a convenient option to make FastScript code easier to read and comprehend. The alias file is simply a list of symbolic constants that can be used as substitutes for numerical values throughout the FastScript language. For instance, many FastScript commands require a device number to identify the specific target hardware for a command. This number can be expressed as an integer, but it can be more intuitive to use a symbol such as DEV0 for device number zero.

The format of the alias file is very simple. It consists of the symbol name followed by the associated numerical value. The white space between these two values can be any number of spaces or tabs. Leading white spaces are also allowed. Comments can be included in the file by using a "#" delimiter as the first character in the line. An example of a typical alias file is shown in Figure 2-1.

```
# Define symbols for up to four hardware devices:

DEV0  0
DEV1  1
DEV2  2
DEV3  3

# Define symbols for two base address registers:

BAR0  0
BAR1  1

# Define a symbol for the address offset of the firmware revision register:

REVISION   0x0800
```

Figure 2-1 Typical Alias File

The numerical values can be expressed as either decimal or hexadecimal quantities. Hexadecimal values are distinguished from decimal values by a “0x” prefix preceding the number. For instance, the value 161 can be expressed as follows:

Decimal: 161

Hexadecimal: 0xA1

Leading zeros do not have to be included in hexadecimal numbers, but it may help to clarify the specific data type that the value represents. For instance, the hexadecimal representation of the value 161 can also be written as follows:

32-bit: 0x000000A1

64-bit: 0x00000000000000A1

It is important to remember that a symbol can only be used to represent one data type. It will assume the data type of the first occurrence of the symbol in the FastScript code. Consider a case where the hexadecimal value 0xA1 is assigned to a symbol called HEX161. The symbol HEX161 can be used to represent either a 32-bit or 64-bit data type in the code, but not both. Similarly, the hexadecimal value 0x00000000000000A1 can only be used to represent a 64-bit data type since it contains too many digits for a 32-bit representation.

A fatal error will be flagged if a type conflict is detected for a symbol.

2.2 Command File (-cmd)

The FastApp command file is the only required input to execute the utility. The command file contains the sequence of FastScript commands that will be executed by the interpreter or converted to C code by the code generator. It is roughly equivalent to “main” required by most high level programming languages.

The command file consists of a sequence of FastScript commands that will be executed sequentially. Command arguments can be separate by any number of white spaces, which may consist of spaces or tabs. Leading white spaces are also allowed. Comments can be included in the file by using a “#” delimiter as the first character in the line. An example of a typical command file is shown in Figure 2-2.

```
# Open the device.
Bus O DEV0 BAR0 0x88 0x0
If $S = 0
    Print Device open successful.
    # Close the device.
    Bus C DEV0
Else
    Print Device open failed.
End
```

Figure 2-2 Typical Command File

A complete description of the FastScript language can be found in Section 3.0 of this manual.

2.3 Interrupt Service Routine File (-isr)

The FastApp interrupt service routine (ISR) file is very similar to the command file. It contains a sequence of FastScript commands that will be executed when the interrupt handler is invoked by the hardware driver. The format of this file is identical to the command file.

The ISR file can reference any symbol defined in the alias file.

2.4 Interpreter (-run)

The FastApp interpreter will execute the FastScript command and ISR files if the –run switch is present. The response from the hardware will be identical to executing the standalone application that can be compiled with the code generator output.

There is typically very little performance sacrificed by running the interpreter since timing critical tasks such as data movement are manage entirely by the hardware.

Any files generated by the interpreter will be created in the current working directory.

2.5 Code Generator (-code)

The FastApp code generator will convert the FastScript alias, command, and ISR files to FastCode C language source files if the –code switch is present. The code generator will create three files in the current working directory:

*cmdfilename*_FastCode.h	:	Header file that will include definitions for all of the symbols listed in the FastScript alias file.
*cmdfilename*_FastCode.c	:	Main source code based on the FastScript command file.
RRAdapter_ISR.c	:	Interrupt service routine code based on the FastScript ISR file. This output file is always required to compile the application, even if no FastScript ISR file was specified.

Note: cmdfilename refers to the name of the FastScript command file.

The FastCode source can be compiled into a standalone application using commercial C language tools.

2.6 Verbosity (-v)

The FastApp verbosity setting determines how much status information is displayed to the standard output when the utility is executed. The scale ranges from zero (default) to five, with the number of messages increasing with the value. Each value will also include messages that would be generated with all of the lower settings combined. Error messages will always be displayed, regardless of the verbosity setting.

FastApp Verbosity Settings	
Verbosity	**Description**
0	The utility will execute silently. Only print statements called from within the FastScript language will be displayed.
1	A summary of command line options is displayed. Messages will also indicate the start and completion of the file parsers that scan the alias and command files for syntax errors.
2	Messages will display the start and completion of the interpreter and code generator tools if they are selected with command line switches.
3	Every line of valid FastScript code that is processed by the parser will be displayed using the abbreviated command format. This can be a helpful debugging tool to locate syntax errors that may be causing error messages.
4	Every FastScript instruction that is processed by the interpreter will be displayed as it executes. This can be a helpful debugging tool to trace the code, but it can also produce a large output. The instructions executed by the interrupt service routine will also be displayed.
5	Reserved setting currently equivalent to 4.

2.7 Application Examples

The FastApp utility offers a general purpose capability that can be applied across several Red Rapids product families. Application examples can be found with the documentation and software distribution that is provided for specific products.

3.0 FastScript Language

The FastScript language is focused on the task of commanding a device to perform a specific function based on hardware register settings. For instance, a Red Rapids signal acquisition product may have several configuration settings that select clock rates, filter characteristics, output format, etc. These products also produce status information that is only available by reading on-board registers.

3.1 Operators

The FastScript language uses a custom set of relational operators to evaluate conditional statements. These operators simplify the manipulation of binary values to extract information from the entire word, or just specific bits.

FastScript Relational Operators

Operator	Symbol	Description
Equality	=	Returns true if the operands are equal.
Inequality	!	Returns true if the operands are not equal.
Less Than	<	Returns true if the first operand is less than the second.
Less Than Or Equal	<=	Returns true if the first operand is less than or equal to the second.
Greater Than	>	Returns true if the first operand is greater than the second.
Greater Than Or Equal	>=	Returns true if the first operand is greater than or equal to the second.
Partial High Bit Comparison	+	Returns true if any high “1” bit in the first operand matches the corresponding bit in the second.
Full High Bit Comparison	+=	Returns true if all high “1” bits in the first operand match all of the corresponding bits in the second.
Partial Low Bit Comparison	-	Returns true if any low “0” bit in the first operand matches the corresponding bit in the second.
Full Low Bit Comparison	-=	Returns true if all low “0” bits in the first operand match all of the corresponding bits in the second.

The FastScript language includes bitwise operators to modify binary values. These are primarily applied to traditional read-modify-write transaction sequences.

FastScript Bitwise Operators

Operator	Symbol	Description
AND	&	Perform a logical AND on each pair of corresponding bits.
OR	|	Perform a logical OR on each pair of corresponding bits.

3.1 Data Types

The FastScript language primarily operates on 32-bit and 64-bit unsigned integer data types. Values can be expressed as either decimal or hexadecimal quantities interchangeably throughout the code. Hexadecimal values are distinguished from decimal values by a “0x” prefix preceding the number. For instance, the value 161 can be expressed as follows:

Decimal: 161

Hexadecimal: 0xA1

Leading zeros do not have to be included in hexadecimal numbers, but it may help to clarify the specific data type that the value represents. For instance, the hexadecimal representation of the value 161 can also be written as follows:

32-bit: 0x000000A1

64-bit: 0x00000000000000A1

Keep in mind that all bits will be evaluated when using a FastScript relational operator. So it will be necessary to include all of the leading ones if you are using the Partial Low Bit Comparison relational operator (-) to search for a zero. For instance, you can search for a zero in bit number 3 of a binary value using the operand 0xFFFFFFFFFFFFFFFB. The value you are searching may not occupy all 64 bits, but the operand must mask the unused bits with a one in this case.

3.2 Variables

The FastScript language includes three built-in variables:

(1) $V: Working variable that stores the most recent value read from an address.

(2) $S: Stores the return status of the most recent command executed.

(3) $I: Stores the device number that generated the current interrupt.

The $V variable is used extensively throughout the code. For example, it would be used three times in a traditional read-modify-write transaction sequence. First it would be used to store the value resulting from the read operation. Next it would be manipulated with a mask in the modify operation. Finally, it would become the source of the write operation.

The FastScript language does not currently support user defined variables.

3.3 Instructions

The FastScript language uses a custom set of instructions to simplify communication between the host computer and the Red Rapids device. Most FastScript commands take the following form:

instruction <*dev*> <*bar*> <*offset*> [*options*]

The instruction may be entered as a complete word, or a one letter abbreviation. The *dev* field identifies the number of the device on the bus. Zero will be assigned to the first card found by the driver, additional devices will be numbered sequentially. The *bar* field refers to the base address register that will be used to reference the register or memory location of interest. The *offset* specifies the specific address of the register or memory location relative to *bar*.

Most Red Rapids products contain two base address registers; BAR0 and BAR2. BAR0 is typically assigned to the command/status registers that control the various hardware functions on the device while BAR2 is typically assigned to application specific settings. Both spaces use 64-bit aligned addressing, so there is no BAR1 or BAR3 space available. This approach enables portability of code between 32-bit and 64-bit operating systems.

The available *options* are specific to each instruction.

FastScript Instructions		
Instruction	**ABBR**	**Description**
#	#	Delimiter to indicate the start of a comment line.
Bus	B	Open or close a device on the bus.
DMA	D	Configure buffers assigned to the device DMA channels.
Hold	H	Pause execution for a specified time or keyboard action.
If-Else-End	I - E - N	Conditional statement that branches based on a Boolean result.
Modify	M	Modify a variable with the specified mask.
Print	P	Print text and an optional variable to the standard output.
Query-Else-End	Q - E - N	Query an address for a condition and branch based on the result.
Read	R	Read the contents of the specified address.
Serial	S	Program a serial bus interface on the device.
Write	W	Write a value to the contents of the specified address.

The following pages summarize each of the FastScript instructions. Command arguments shown between angled brackets < > are required, optional arguments are shown between square brackets [ ].

Instruction	#
Abbreviation	#
Syntax (close)	# [*comment string*]
Status Return	None.
Description	Delimiter to indicate the start of a comment line.

Argument	**Range**	**Description**
comment string	N/A	Any combination of text characters.

Examples

# This line will not be executed.

The comment line can be placed anywhere in a FastScript file.

#This line will not be executed.

No space is required after the delimiter.

Instruction	Bus
Abbreviation	B
Syntax (close) **Syntax (open)**	Bus C *<dev>* Bus O *<dev> <int mask bar> <int mask offset> <int mask disable>*
Status Return	Zero if successful, error code otherwise.
Description	Open or close a device on the bus. The bus open operation requires information about the global interrupt mask register assigned to the device. This is needed for the interrupt handler to reduce redundant traffic.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
int mask bar	0, 2, 4	Base address register of the global interrupt mask register.
int mask offset	32-bit	Address offset from bar of the global interrupt mask register.
int mask disable	64-bit	Binary value that should be written to the global interrupt mask register to disable all interrupts from the device.

Examples

Bus O 0 0 0x0088 0x0

Open device number 0 and set the global interrupt mask register to address offset 0x0088 from BAR0. A global interrupt mask value of 0x0 will disable all interrupts.

B C 0

Close device number 0.

Instruction	DMA
Abbreviation	D
Syntax (allocate)	DMA A *<dev> <bar> <offset> <channel> <burst size> <burst count> <page count> <pages/int>*
Syntax (free)	DMA F *<dev> <channel>*
Syntax (save)	DMA S *<dev> <channel> <file name> <number of bits>*
Status Return	Zero if successful, error code otherwise.
Description	Configure buffers assigned to the device DMA channels. This instruction is also used to dump the contents of a DMA channel to a file.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
bar	0, 2, 4	Base address register for this transaction.
offset	32-bit	Address offset from bar where the 64-bit DMA page addresses are stored.
channel	1 to 4	DMA channel number assigned to the buffer.
burst size	32-bit	Size of each individual DMA burst transaction in bytes.
burst count	32-bit	Number of bursts to store in each DMA page.
page count	1 to 1k	Number of individual DMA pages to allocate.
pages/int	1 to 1k	Specifies the number of DMA pages the channel will service before the hardware issues an update interrupt.
file name	N/A	Name of the file to store DMA buffer contents.
number of bits	8 or 16	Word size of the data stored in the DMA buffer.

Examples

DMA A 3 0 0x2000 1 1024 4 8 2

Allocate a DMA buffer to channel number 1 on device number 3 using address offset 0x2000 from BAR0 to store the configuration information. Create a total of 8 DMA pages for this channel, each 4 kbytes (1024 x 4) large. Configure the device to issue an interrupt on completion of every 2 pages.

D F 3 1

Free the DMA buffer assigned to channel number 1 on device number 3.

DMA S 3 1 chAout.txt 16

Save the contents of the DMA buffer assigned to channel number 1 on device number 3 in a file named chAout.txt. Set the data word size to 16 bits for the samples extracted from the buffer.

Instruction	Hold
Abbreviation	H
Syntax (keybrd) **Syntax (timer)**	Hold K Hold T <*delay*>
Status Return	None.
Description	Pause execution for a specified time or keyboard action. The keyboard option will send the following message to standard output: Press **ENTER** from the keyboard to continue...

Argument	**Range**	**Description**
delay	32-bit	Length of the pause in microseconds.

Examples

Hold K

Pause execution until ENTER is hit on the keyboard.

H T 10

Pause execution for 10 microseconds.

Instruction	If-Else-End
Abbreviation	I - E - N
Syntax	If <*variable*> <*operator*> <*operand*> [*FastApp script*] Else [*FastApp script*] End
Status Return	N/A
Description	Conditional statement that branches based on a Boolean result. The Else branch is not required, but the End statement must always terminate the conditional.

Argument	**Range**	**Description**
variable	64-bit	Any FastApp built-in variable ($I, $V, $S).
operator	N/A	Any of the available FastApp operators.
operand	64-bit	A 64-bit binary constant.
FastApp script	N/A	Any number of FastApp script commands.

Examples

```
If $S = 0
  Print Status variable indicates successful execution of previous command.
Else
  Print Status variable returned an error code for the previous command.
End
```

The conditional checks the value of the built-in status variable ($S) to determine whether the previous command returned an error code. This particular structure is very useful in most applications. For instance, it can be used to check the status return from the Bus open command to immediately exit the application if there was an error.

```
I $V - 0xFFFFFFFFFFFFFFFA
  P At least one low bit detected.
N
```

The conditional will return true if either bit 0 or bit 2 of the built-in variable $V is low. There is no Else branch included in this example. The Print statement will execute if the conditional is true, otherwise it is skipped and execution proceeds after the End statement.

Instruction	Modify
Abbreviation	M
Syntax (close)	Modify *<variable> <operator> <mask>*
Status Return	None.
Description	Modify a variable with the specified mask.

Argument	**Range**	**Description**
variable	64-bit	Only the FastApp $V built-in variable is allowed.
operator	N/A	Any of the available FastApp bitwise operators.
mask	64-bit	Binary constant that is used to modify the variable.

Examples

Modify $V & 0x0A0B0C0D0E0F0123

Modify the built-in variable $V by performing a bitwise AND operation with 0x0A0B0C0D0E0F0123.

M $V | 16

Set bit 4 of the built-in variable $V to a high.

Instruction	Print
Abbreviation	P
Syntax (close)	Print [*option*]
Status Return	None.
Description	Print text and an optional variable to the standard output. If requested, the variable will be printed immediately following the text.

Argument	**Range**	**Description**
option	64-bit	Any FastApp built-in variable ($I, $V, $S).

Examples

Print Hello World

Prints “Hello World” to the standard output.

P $V Firmware Revision:

Prints the text “Firmware Revision:“ followed by the value of built-in variable $V. This example implies that the preceding command was a Read operation to a register that stores the revision date.

Instruction	Query-Else-End
Abbreviation	Q - E - N
Syntax	Query *<dev> <bar> <offset> <operator> <operand> <max attempts> <delay>* [*FastApp script*] Else [*FastApp script*] End
Status Return	N/A
Description	Query an address for a condition and branch based on the result. This instruction will continuously read the specified address until the conditional returns true or the maximum number of attempts has been reached. The read operations are separated by the specified delay interval to avoid choking the bus. The conditional will branch based on the Boolean result of the last query cycle. The Else branch is not required, but the End statement must always terminate the conditional.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
bar	0, 2, 4	Base address register for this transaction.
offset	32-bit	Address offset from bar that defines a unique 64-bit location.
operator	N/A	Any of the available FastApp operators.
operand	64-bit	A 64-bit binary constant.
max attempts	32-bit	Maximum number of attempted read operations to the address.
delay	32-bit	Delay in microseconds between read operarations.
FastApp script	N/A	Any number of FastApp script Instructions.

Examples

```
Query 0  2 0x0820 - 0xFFFFFFFFFFFFFFFE 5 1
   Print Initialization complete.
Else
   Print Initialization failed.
End
```

Continuously read the 64-bit value from device number 0 at address offset 0x0820 from BAR2 until bit 0 is low. Perform a maximum of 5 read operations separated by 1 microsecond. This particular structure is useful for monitoring a “busy bit” while the hardware initializes.

```
Q 3 0 0x0010 += 0x6 10 50.
   Print Data transfer complete on both channels.
N
```

Continuously read the 64-bit value from device number 3 at address offset 0x0010 from BAR0 until bit 1 and bit 2 are both high. Perform a maximum of 10 read operations separated by 50 microseconds. The Print statement will execute if the conditional finishes true, otherwise it is skipped and execution proceeds after the End statement.

Instruction	Read
Abbreviation	R
Syntax	Read *<dev> <bar> <offset>*
Status Return	Zero if successful, error code otherwise.
Description	Read the contents of the specified address and store the value in variable $V.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
bar	0, 2, 4	Base address register for this transaction.
offset	32-bit	Address offset from bar that defines a unique 64-bit location.

Examples

Read 0 2 0x0800

Read the 64-bit value from device number 0 at address offset 0x0800 from BAR2. The value read will be stored in variable $V and the return status will be stored in variable $S.

R 3 0 512

Read the 64-bit value from device number 3 at address offset 512 from BAR0. The value read will be stored in variable $V and the return status will be stored in variable $S.

Instruction	Serial
Abbreviation	S
Syntax	Serial *<dev> <bar> <offset> <properties> <payload>*
Status Return	Zero if successful, error code otherwise.
Description	Program a serial bus interface on the device. Modern chips frequently use a serial bus to accept configuration instructions or provide status information. This instruction will program a serial bus using a defined set of properties to transmit and receive payload information. The result of a serial read operation is returned through the built-in $V variable.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
bar	0, 2, 4	Base address register for this transaction.
offset	32-bit	Address offset from bar that defines a unique 64-bit location.
properties	32-bit	Code that configures the properties of the serial control port.
payload	32-bit	Concatenated instruction and data word.

Examples

Serial 1 0 0x0130 0x000000A0 0x00005A01

Command the serial port on device number 1 at address offset 0x0130 from BAR0. The behavior of the port is defined by the property code 0x000000A0 and the payload consists of a 16-bit instruction (0x005A) followed by an 8-bit data word (0x01).

S 1 0 0x0130 0x00000160 0x00000ACE

Command the serial port on device number 1 at address offset 0x0130 from BAR0. The behavior of the port is defined by the property code 0x00000160 and the payload consists of a 8-bit instruction (0x0A) followed by an 8-bit data word (0xCE).

Instruction	Write
Abbreviation	W
Syntax	Write <*dev*> <*bar*> <*offset*> [*option*]
Status Return	Zero if successful, error code otherwise.
Description	Write a value to the contents of the specified address. If the option field is left blank, then the variable $V will be written.

Argument	**Range**	**Description**
dev	0 to 7	Number of the device on the bus.
bar	0, 2, 4	Base address register for this transaction.
offset	32-bit	Address offset from bar that defines a unique 64-bit location.
option	64-bit	A 64-bit binary constant to write instead of variable $V.

Examples

Write 0 2 x00800 0x0000000000100010

Write the 64-bit value 0x0000000000100010 to device number 0 at address offset 0x0800 from BAR2. The return status will be stored in variable $S.

W 3 0 512

Write the value of the 64-bit variable $V to device number 3 at address offset 512 from BAR0. The return status will be stored in variable $S.