Common Host Utilities for VxWorks
Release: 1.13.0

Document Revision: 1.2
12 Jun 2017

	Head Office	US Office
Address:	Suite L4A, 160 Dundee Street, Edinburgh, EH11 1DQ, UK	10822 West Toller Drive, Suite 250 Littleton, CO 80127
Telephone:	+44 131 558 2600	(303) 954 8768
Fax:	+44 131 558 2700	(866) 820 9956 - toll free
email:	sales@alpha-data.com	sales@alpha-data.com
website:	http://www.alpha-data.com	http://www.alpha-data.com

Table Of Contents

List of Tables

List of Figures

1 Introduction

This document describes the Common Host Utilities for VxWorks, for Alpha Data Gen 3 Reconfigurable Computing Hardware. In this context, "common" refers to the fact that these utilities, with some exceptions, can be used with all models in Alpha Data's range of Gen 3 Reconfigurable Computing Hardware:

- Embedded system products:
  - ADM-XRC-6TL
  - ADM-XRC-6T1
  - ADM-XRC-6T-DA1
  - ADM-XRC-6TGE and ADM-XRC-6TGEL
  - ADM-XRC-6T-ADV8
  - ADPE-XRC-6T and ADPE-XRC-6T-L
  - ADM-XRC-7K1
  - ADM-XRC-7V1
  - ADM-VPX3-7V2
  - ADM-XRC-KU1
- Datacenter products:
  - ADM-PCIE-7V3
  - ADM-PCIE-KU3
  - ADM-PCIE-8V3
  - ADM-PCIE-8K5

Table 1 lists the available utilities for VxWorks.

AVR2UTIL	Utility for manipulating microcontroller firmware and related data (for certain models only).
BITSTRIP	Utility for removing the header from a .bit file, leaving only the SelectMap data
DMADUMP	Utility for reading and writing using DMA engines
DUMP	Utility for reading and writing memory windows
FLASH	Utility for programming FPGA bitstream (.BIT) files in user-programmable Flash memory
INFO	Utility for displaying information about a reconfigurable computing device
LOADER	Utility for configuring a target FPGA with a bitstream file
MONITOR	Utility that displays sensor readings
VPD	Utility that allows the Vital Product Data of a reconfigurable computing device to be read or written

Table 1 : Utilities for VxWorks

1.1 Directory structure

The files and folders making up the Common Host Utilities are organized as in Figure 1 below:

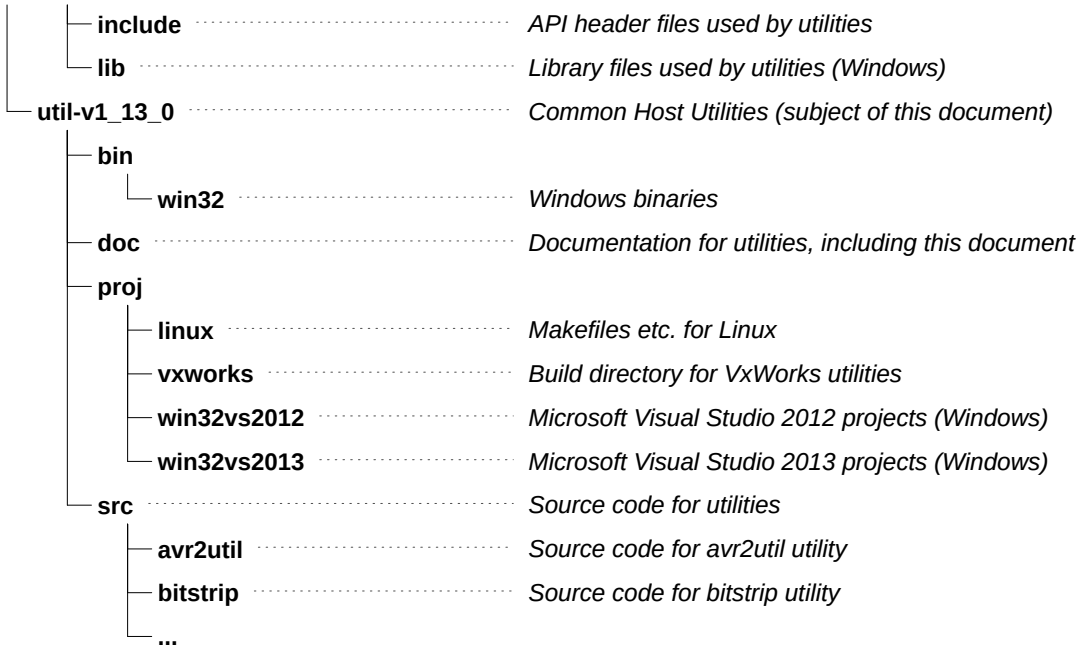

Figure 1 : Directory structure

The root of this package, i.e. the directory which forms the root of tree of directories and files making up this package, is referred to in the remainder of this document as **(root)**.

The base directory of the Common Host Utilties, i.e. **(root)/host/util-v1_13_0/** is referred to in the remainder of this document as **(util)**.

2 Building the Common Host Utilities

2.1 Building the VxWorks utilities on a Windows host

The build system for the host utilities uses the Makefile infrastructure provided by VxWorks.The CPU architecture, tool chain etc. are specified directly on the **make** command line.

In this section, it will be assumed that the target machine is a 64-bit symmetric multiprocessing (SMP) Intel NEHALEM machine, and the tool chain to be used is GNU. In general, the CPU architecture and toolchain that is specified on the **make** command line should match that of the VxWorks kernel of the target machine.

If using a Windows machine for VxWorks hosting and development, follow this procedure:

1. If you do not have write permissions for the host directory, **(root)/host/**, first make a copy of it and use the copy for the remainder of this procedure.
2. Start a **VxWorks Development Shell** via the shortcut on the Windows Start Menu. If you have more than one version of VxWorks installed, use the same VxWorks version as used when building the VxWorks kernel for the target machine. It is important to use this shortcut in order to obtain the correct environment for performing command-line builds using the VxWorks toolchains.
3. Change directory to the **(root)/host/** directory.
4. Issue the following **make** command:

```
make CPU=NEHALEM VXBUILD="LP64 SMP" QUIET=true clean default
```

Assuming that the above command was executed successfully, the binary downloadable module **hostUtils.out**, containing all of the example VxWorks applications, is located in the current directory.

2.2 Building the VxWorks utilities on a Linux host

TBA

2.3 MAKE options for the example VxWorks applications

2.3.1 MAKE targets

The Makefile for the VxWorks examples defines the following top-level targets for the **make** command line:

- **default** or **all**

  This is a **.PHONY** target used to build the host utilities binary.

- **clean**

  This is a **.PHONY** target used to delete the host utilities binary and intermediate build files (**.o**, **.a** etc.).

2.3.2 MAKE variables

The Makefile for the VxWorks examples accepts a number of variables which are passed on the **make** command line. These are:

- **CPU=*<architecture>***

  Specifies the target CPU, e.g. **PENTIUM4**, **PPC604**, **NEHALEM** etc. The default is **PPC604**.

- **QUIET=*<false|true>***

  Specifies whether or not to suppress the display of build commands. A value of **true** can be helpful in order to avoid missing warnings during build, whereas a value of **false** is useful for verifying that the expected compiler and linker options are used.

- **TOOL=*<diab|gnu|icc>***

  Selects the toolchain for building. The default is **gnu**.

- **VXBUILD=*<variant>***

Specifies the VxWorks library variant to use, e.g. **VXBUILD="LP64 SMP"** if building to run under a 64-bit SMP VxWorks kernel. The default is to use the regular libraries, which are 32-bit and uniprocessor for most embedded CPU architectures.

This variable can be specified only for VxWorks 6.6 or later. In the case of VxWorks 6.6, the only valid possibilities are **VXBUILD="SMP"** or to leave VXBUILD unspecified.

3 Common Host Utilties

3.1 AVR2UTIL utility

VxWorks kernel shell entry points

```
int avr2utilHelp              ()
int avr2utilBuildInfo         ()
int avr2utilProductID         (indexOrSerial, flags)
int avr2utilVersion           (indexOrSerial, flags)
int avr2utilEnterServiceMode  (indexOrSerial, flags)
int avr2utilExitServiceMode   (indexOrSerial, flags)
int avr2utilGetClk            (indexOrSerial, flags, clockIndex)
int avr2utilSetClk            (indexOrSerial, flags, clockIndex, frequencyHz)
int avr2utilGetClkNV          (indexOrSerial, flags, clockIndex)
int avr2utilSetClkNV          (indexOrSerial, flags, clockIndex, frequencyHz)
int avr2utilI2cReadToFile     (indexOrSerial, flags, bus, device, address, count,
                               pOutFilename)
int avr2utilI2cVerifyFromFile(indexOrSerial, flags, bus, device, address,
                               pInFilename)
int avr2utilI2cWriteFromFile  (indexOrSerial, flags, bus, device, address,
                               pInFilename)
int avr2utilI2cRead           (indexOrSerial, flags, bus, device, address, count)
int avr2utilI2cWrite          (indexOrSerial, flags, bus, device, address, count,
                               ... /* write bytes */)
int avr2utilUpdateBrdCfg      (indexOrSerial, flags, pBrdCfgFilename)
int avr2utilVerifyBrdCfg      (indexOrSerial, flags, pBrdCfgFilename)
int avr2utilSaveBrdCfg        (indexOrSerial, flags, pBrdCfgSaveFilename)
int avr2utilUpdateFirmware    (indexOrSerial, flags, pFirmwareFilename)
int avr2utilVerifyFirmware    (indexOrSerial, flags, pFirmwareFilename)
int avr2utilSaveFirmware      (indexOrSerial, flags, pFirmwareSaveFilename)
int avr2utilUpdateVPD         (indexOrSerial, flags, pVPDFilename)
int avr2utilVerifyVPD         (indexOrSerial, flags, pVPDFilename)
int avr2utilSaveVPD           (indexOrSerial, flags, pVPDSaveFilename)
int avr2utilDisplayVPD        (indexOrSerial, flags)
int avr2utilDisplayVPDRaw     (indexOrSerial, flags)
int avr2utilDisplaySensors    (indexOrSerial, flags)
int avr2utilDisplaySensorsRaw(indexOrSerial, flags)
int avr2utilOverrideSensor    (indexOrSerial, flags, sensorIndex, overrideValue)
int avr2utilReleaseSensor     (indexOrSerial, flags, sensorIndex)
int avr2utilSpiInfo           (indexOrSerial, flags, chipIndex)
int avr2utilSpiRaw            (indexOrSerial, flags, chipIndex, readCount,
                               writeCount, ... /* write bytes */)
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int clockIndex	Index of a clock generator output whose override frequency is to be programmed or queried.

unsigned int frequencyHz	The override frequency to be programmed for a given clock generator output; the values 0 or 0xFFFFFFFF have special meaning (see descriptions of **avr2utilGetClkNV** and **avr2utilSetClkNV** below).
unsigned int bus	The index of the I2C bus on which the I2C device of interest resides.
unsigned int device	The device number within a given I2C bus, which identifies the I2C device of interest.
unsigned int address	An address within the I2C device of interest.
unsigned int count	A byte count, used in certain I2C-related entry points.
unsigned int sensorIndex	Specifies the index of a sensor.
unsigned int overrideValue	The unscaled value that is to be injected into a sensor, overriding its natural value.
unsigned int chipIndex	Specifies the index of a SPI Flash chip.
unsigned int readCount	A count of bytes to read, used in certain entry points related to SPI Flash access.
unsigned int writeCount	A count of bytes to write, used in certain entry points related to SPI Flash access.
const char* pInFilename	The name of a binary file (**.bin** extension) containing data to be written to an I2C device or used to verify the contents of an I2C device.
const char* pOutFilename	The name of a binary file (**.bin** extension) into which data is written after being read from an I2C device.
const char* pBrdCfgFilename	The name of a binary file (**.bin** extension) containing board configuration information.
const char* pBrdCfgSaveFilename	The name of a binary file (**.bin** extension) into which board configuration information read from a card is to be saved.
const char* pFirmwareFilename	The name of a binary file (**.bin** extension) containing microcontroller firmware.
const char* pFirmwareSaveFilename	The name of a binary file (**.bin** extension) into which microcontroller firmware read from a card is to be saved.
const char* pVPDFilename	The name of a binary file (**.bin** extension) containing Vital Product Data (VPD).
const char* pVPDSaveFilename	The name of a binary file (**.bin** extension) into which Vital Product Data (VPD) read from a card is to be saved.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x2	**FLAG_VERBOSE**	Displays commands sent to the microcontroller and its responses, for debug purposes.

Return value

When **AVR2UTIL** successfully performs the requested operation, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_INVALID_INDEX	3	Index value out of range or not a valid number.
EXIT_INVALID_FREQUENCY	4	Frequency value out of range or not a valid number.
EXIT_READ_FIRMWARE_FAILED	5	Failed to read firmware file.
EXIT_FIRMWARE_TOO_LARGE	6	Firmware file is too large for uC.
EXIT_ALLOCATION_FAILED	7	Failed to allocate buffer for firmware or board config. data.
EXIT_VERIFY_FAILED	8	Errors found when verifying updated firmware or board config. data.
EXIT_UNSUPPORTED_MODEL	9	Attempting to use this utility on an unsupported model.
EXIT_USB_NOT_SUPPORTED	11	Access to AVR2 uC via USB is currently not supported for this OS.
EXIT_MODE_CHANGE_FAILED	12	AVR2 uC did not enter or exit Service Mode as requested.
EXIT_WRONG_MODE	13	Device is in the wrong mode (Service Mode vs. non-Service Mode) for the requested command.
EXIT_UNRECOGNIZED_PRODUCTID	14	Product ID not recognized; aborting as a precaution against firmware corruption.
EXIT_INVALID_I2C_BUS	15	I2C bus number is not valid.
EXIT_INVALID_I2C_DEVICE	16	I2C device number is not valid.
EXIT_INVALID_I2C_ADDRESS	17	I2C address is not valid.
EXIT_I2C_READ_FILE_FAILED	18	Failed to read data for I2C write from file.
EXIT_I2C_WRITE_FILE_FAILED	19	Failed to write data from I2C read to file.
EXIT_INVALID_I2C_COUNT	20	I2C data byte count is not valid.
EXIT_INVALID_I2C_BYTEVAL	21	I2C data byte value is not valid.
EXIT_WRITE_FIRMWARE_FAILED	22	Failed to write firmware/VPD/board config to a file.
EXIT_INVALID_OVERRIDE	23	Sensor override value out of range or not a valid number.
EXIT_INVALID_SPI_INDEX	24	SPI chip index is not valid.
EXIT_INVALID_SPI_READ_COUNT	25	SPI read byte count is not valid.
EXIT_INVALID_SPI_WRITE_COUNT	26	SPI write byte count is not valid.
EXIT_INVALID_SPI_BYTEVAL	27	SPI data byte value is not valid.
EXIT_DEVICE_OPEN_ERROR	100	Failed to open device.
EXIT_AVR2_STATUS_ERROR	101	Failed to get AVR2 uC status.
EXIT_AVR2_COMMAND_ERROR	102	Failed to send command to AVR2 uC.
EXIT_AVR2_BAD_STATUS	103	AVR2 uC returned nonzero status for operation.

Table 2 : Return values for AVR2UTIL utility (continued on next page)

Symbolic name	Value	Meaning
EXIT_AVR2_SHORT_RESPONSE	104	AVR2 uC's response was too short (< 2 bytes) to be valid.
EXIT_LIBRARY_NOT_FOUND	105	Could not find AVR2 or ADB3 shared library/DLL.

Table 2 : Return values for AVR2UTIL utility

Summary

This utility performs maintenance functions on the firmware of the microcontroller and associated data on the following models in Alpha Data's range of reconfigurable computing hardware:

- ADM-XRC-KU1
- ADM-PCIE-8V3
- ADM-PCIE-8K5

AVR2UTIL supports the following use-cases:

- Checking the version of the microcontroller firmware.
- Programming the clock generator on a board in a nonvolatile manner, so that it powers up providing a user-specified frequency at a given clock input on the FPGA.
- Upgrading the microcontroller firmware and its associated data.

Description

In VxWorks, **AVR2UTIL** communicates with the microcontroller on supported models in Alpha Data's range of reconfigurable computing hardware using the ADB3 Driver. This requires the ADB3 Driver to be installed and running.

In the current version of AVR2UTIL for VxWorks, USB communication is not currently supported.

The entry points making up the **AVR2UTIL** utility can be invoked in the VxWorks shell in a number of ways:

- **avr2utilHelp**

  Displays brief help, including the list of entry points.
- **avr2utilBuildInfo**

  Displays the version of **AVR2UTIL** itself and information about how it was built.
- **avr2utilVersion** *indexOrSerial*, *flags*

  Displays the version of the microcontroller firmware in a given device.
- **avr2utilProductID** *indexOrSerial*, *flags*

  Displays the Product ID of the microcontroller firmware in a given device.
- **avr2utilEnterServiceMode** *indexOrSerial*, *flags*

  Commands the microcontroller in a given device to enter Service Mode.
- **avr2utilExitServiceMode** *indexOrSerial*, *flags*

  Commands the microcontroller in a given device to exit Service Mode.
- **avr2utilGetClk** *indexOrSerial*, *flags*, *clockIndex*

  Gets the (volatile) current frequency of the particular clock generator selected by *clockIndex*.
- **avr2utilSetClk** *indexOrSerial*, *flags*, *clockIndex*, *frequencyHz*

  Sets the (volatile) current frequency of the particular clock generator selected by *clockIndex* to *frequencyHz* Hz.
- **avr2utilGetClkNV** *indexOrSerial*, *flags*, *clockIndex*

  Gets the nonvolatile override frequency for the particular clock generator selected by *clockIndex*.

- **avr2utilSetClkNV** *indexOrSerial*, *flags*, *clockIndex*, *frequencyHz*

  Sets the nonvolatile override frequency for the particular clock generator selected by *clockIndex* to *frequencyHz* Hz.

- **avr2utilI2cReadToFile** *indexOrSerial*, *flags*, *bus*, *device*, *address*, *count*, *pOutFilename*

  Performs multiple single-byte reads of an I2C device and saves the data into a file.

- **avr2utilI2cVerifyWithFile** *indexOrSerial*, *flags*, *bus*, *device*, *address*, *pInFilename*

  Performs multiple single-byte reads of an I2C device and compares the data with the contents of a file.

- **avr2utilI2cWriteFromFile** *indexOrSerial*, *flags*, *bus*, *device*, *address*, *pInFilename*

  Writes the contents of a file to an I2C device using multiple single-byte writes.

  NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt board control devices.

- **avr2utilI2cRead** *indexOrSerial*, *flags*, *bus*, *device*, *address*, *count*

  Performs an individual I2C read of one or more bytes from an I2C device, displaying the data read.

- **avr2utilI2cWrite** *indexOrSerial*, *flags*, *bus*, *device*, *address*, *count*, ...

  Performs an individual I2C write of one or more bytes to an I2C device, obtaining the bytes to write from additional values following the *count* argument.

  NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt board control devices.

- **avr2utilUpdateBrdCfg** *indexOrSerial*, *flags*, *pBrdCfgFilename*

  Writes the board-specific configuration area used by the microcontroller firmware with the contents of the file *pBrdCfgFilename*.

  NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt data required by the microcontroller's firmware.

- **avr2utilVerifyBrdCfg** *indexOrSerial*, *flags*, *pBrdCfgFilename*

  Verifies the board-specific configuration area used by the microcontroller firmware against the contents of the file *pBrdCfgFilename*.

- **avr2utilSaveBrdCfg** *indexOrSerial*, *flags*, *pBrdCfgSaveFilename*

  Reads the board-specific configuration area used by the microcontroller firmware, from the nonvolatile memory device in which it resides on a board, and saves it into the file *pBrdCfgSaveFilename*.

- **avr2utilUpdateFirmware** *indexOrSerial*, *flags*, *pFirmwareFilename*

  Writes the firmware of the microcontroller with the contents of the file *pFirmwareFilename*.

  NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt the microcontroller's firmware.

- **avr2utilVerifyFirmware** *indexOrSerial*, *flags*, *pFirmwareFilename*

  Verifies the firmware of the microcontroller against the contents of the file *pFirmwareFilename*.

- **avr2utilSaveFirmware** *indexOrSerial*, *flags*, *pFirmwareSaveFilename*

  Reads the the firmware of the microcontroller, from the nonvolatile memory device in which it resides on a board, and saves it into the file *pFirmwareSaveFilename*.

- **avr2utilUpdateVPD** *indexOrSerial*, *flags*, *pVPDFilename*

  Writes the Vital Product Data (VPD) for the board with the contents of the file *pVPDFilename*.

  NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt the board's VPD.

- **avr2utilVerifyVPD** *indexOrSerial*, *flags*, *pVPDFilename*

  Verifies the Vital Product Data (VPD) for the board against the contents of the file *pVPDFilename*.

- **avr2utilSaveVPD** *indexOrSerial*, *flags*, *pVPDSaveFilename*

Reads the Vital Product Data (VPD), from the nonvolatile memory device in which it resides on a board, and saves it into the file *pVPDSaveFilename*.

- **avr2utilDisplayVPD** *indexOrSerial*, *flags*

  Reads the Vital Product Data (VPD) from the board and displays it in human-readable form.

- **avr2utilDisplayVPDRaw** *indexOrSerial*, *flags*

  Reads the Vital Product Data (VPD) from the board and displays it as raw bytes.

- **avr2utilDisplaySensors** *indexOrSerial*, *flags*

  Reads the Sensor Page from the board and displays it in human-readable form.

- **avr2utilDisplaySensorsRaw** *indexOrSerial*, *flags*

  Reads the Sensor Page from the board and displays it as raw bytes.

- **avr2utilOverrideSensor** *indexOrSerial*, *flags*, *<sensorIndex>*, *<overrideValue>*

  Used by Alpha Data for firmware testing; permits a particular reading to be injected into a sensor, overriding its natural value.

- **avr2utilReleaseSensor** *indexOrSerial*, *flags*, *<sensorIndex>*

  Used by Alpha Data for firmware testing; undoes the **avr2utilReleaseSensor** function, returning a sensor to normal operation.

3.1.1 Available entry points

3.1.1.1 avr2utilHelp entry point

The **avr2utilHelp** entry point displays a brief help message, listing the available entry points and their parameters.

Usage example:

```
avr2utilHelp
```

3.1.1.2 avr2utilBuildInfo entry point

The **avr2utilBuildInfo** entry point returns the version number of **AVR2UTIL** itself along with some information about how it was built:

- Whether dynamically or statically linked to the ADB3 API library (used when communicating with the uC via the PCIe host interface of a reconfigurable computing card), and the version of the ADB3 API header files.

The general form of this entry point has no arguments and is:

```
avr2utilBuildInfo
```

3.1.1.3 avr2utilVersion entry point

The **avr2utilVersion** entry point displays the version of the microcontroller firmware in the form *a.b.c.d*. This can be used as a further check in order to verify that a previous operation to update firmware was successful.

If the microcontroller is in Service Mode, this gives the version of the Boot Manager II (BootMan2) firmware; otherwise, it gives the version of the Board Manager II (BoardMan2) firmware.

The general form of this entry point is:

```
avr2utilVersion indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - The following three invocations are all equivalent and display the microcontroller firmware version for the first device in the system:

```
avr2utilVersion
avr2utilVersion 0
```

```
avr2utilVersion 0,0
```

Usage example 2 - Display the the microcontroller firmware version for the device with serial number 100:

```
avr2utilVersion 100,1
```

3.1.1.4 avr2utilProductID entry point

Displays the Product ID of the firmware. The following known Product IDs exist:

- 1320 (0x528), 720 (0x2D0) or 184320 (0x2D000) => Boot Manager II (BootMan2)
- 1321 (0x529), 721 (0x2D1) or 184321 (0x2D001) => Board Manager II (BoardMan2)

This provides a way to determine whether or not the microcontroller is in Service Mode; if the Product ID corresponds to BootMan2, the microcontroller is in Service Mode.

The general form of this entry point is:

```
avr2utilProductID indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - The following three invocations are all equivalent and display microcontroller firmware's Product ID for the first device in the system:

```
avr2utilProductID
avr2utilProductID 0
avr2utilProductID 0,0
```

Usage example 2 - Display the the microcontroller firmware's Product ID for the device with serial number 100:

```
avr2utilProductID 100,1
```

3.1.1.5 avr2utilEnterServiceMode entry point

Commands the microcontroller to enter service mode; if already in Service Mode, this entry point does nothing. The microcontroller remains in Service Mode until commanded to exit Service Mode or until a power cycle occurs.

The general form of this entry point is:

```
avr2utilEnterServiceMode indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Command the first device in the system to enter Service Mode:

```
avr2utilEnterServiceMode
```

Usage example 2 - Command the the device with serial number 100 to enter Service Mode:

```
avr2utilEnterServiceMode 100,1
```

3.1.1.6 avr2utilExitServiceMode entry point

Commands the microcontroller to exit service mode; if not in Service Mode, this entry point does nothing.

The general form of this entry point is:

```
avr2utilExitServiceMode indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Command the first device in the system to exit Service Mode:

```
avr2utilExitServiceMode
```

Usage example 2 - Command the the device with serial number 100 to exit Service Mode:

```
avr2utilExitServiceMode 100,1
```

3.1.1.7 avr2utilGetClk entry point

The **avr2utilGetClk** entry point returns the current frequency, in Hz, of a particular clock generator.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilGetClk indexOrSerial,flags,clockIndex
```

where *indexOrSerial* and *flags* are parameters as described above.

clockIndex is the index of an output in the device's clock generator. For the correspondence of *clockIndex* to physical clock nets, refer to Appendix A.

Usage example 1 - Display the current frequency of clock output 1, for the first device in the system:

```
avr2utilGetClk 0,0,1
```

Usage example 2 - Display the current frequency of clock output 1, for the device with serial number 100:

```
avr2utilGetClk 100,1,1
```

3.1.1.8 avr2utilSetClk entry point

The **avr2utilSetClk** entry point sets the current frequency, in Hz, for a particular clock generator, effective immediately. The operation performed is volatile, and a power cycle returns all clock generators to their default frequencies.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilSetClk indexOrSerial,flags,clockIndex,frequencyHz
```

where *indexOrSerial* and *flags* are parameters as described above. Unlike the **avr2utilSetClkNV** entry point, the **avr2utilSetClk** entry point can reprogram all clock outputs (albeit in a volatile way), even those whose nonvolatile frequency cannot be overridden.

clockIndex is the index of an output in the device's clock generator. For the correspondence of *clockIndex* to physical clock nets, refer to Appendix A.

frequencyHz is the new frequency, in Hz.

Usage example 1 - Set the current frequency of clock output 1, for the first device in the system, to 250 MHz:

```
avr2utilSetClk 0,0,1,250000000
```

Usage example 2 - Set the current frequency of clock output 1, for the device with serial number 100, to 250 MHz:

```
avr2utilSetClk 100,1,1,250000000
```

3.1.1.9 avr2utilGetClkNV entry point

The **avr2utilGetClkNV** entry point returns the current nonvolatile override frequency, in Hz, for a particular clock generator.

At power-on, the microcontroller inspects each clock generator's nonvolatile override frequency in turn. If set to a value other than 0 or 4294967295 (0xFFFFFFFF), it programs the clock generator to output a clock of that frequency. Otherwise, the clock generator remains at its factory default frequency.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilGetClkNV indexOrSerial,flags,clockIndex
```

where *indexOrSerial* and *flags* are parameters as described above.

clockIndex is the index of an output in the device's clock generator. For the correspondence of *clockIndex* to physical clock nets, refer to Appendix A.

Usage example 1 - Display the override frequency of clock output 1 for the first device in the system:

```
avr2utilGetClkNV 0,0,1
```

Usage example 2 - Display the override frequency of clock output 1 for the device with serial number 100:

```
avr2utilGetClkNV 100,1,1
```

3.1.1.10 avr2utilSetClkNV entry point

The **avr2utilSetClkNV** entry point sets the nonvolatile override frequency, in Hz, for a particular clock generator. This entry point does **not** cause the specified clock generator's actual output frequency to change immediately.

At power-on, the microcontroller inspects each clock generator's nonvolatile override frequency in turn. If set to a value other than 0 or 4294967295 (0xFFFFFFFF), it programs the clock generator to output a clock of that frequency. Otherwise, the clock generator remains at its factory default frequency.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilSetClkNV indexOrSerial,flags,clockIndex,frequencyHz
```

where *indexOrSerial* and *flags* are parameters as described above.

clockIndex is the index of an output in the device's clock generator. For the correspondence of *clockIndex* to physical clock nets, refer to Appendix A.

frequencyHz is the override frequency, in Hz. To unset the nonvolatile override frequency for a particular clock generator, use a value of 0 or 0xFFFFFFFF.

Usage example 1 - Set the override frequency of clock output 1, for the first device in the system, to 250 MHz:

```
avr2utilSetClkNV 0,0,1,250000000
```

Usage example 2 - Set the override frequency of clock output 1 for the device with serial number 100, to 250 MHz:

```
avr2utilSetClkNV 100,1,1,250000000
```

Usage example 3 - Unset (remove) the override frequency of clock output 2, for the first device in the system:

```
avr2utilSetClkNV 0,0,2,0xFFFFFFFF
```

3.1.1.11 avr2utilI2cReadToFile entry point

The **avr2utilI2cReadToFile** entry point is primarily for in-house use by Alpha Data, but may also be used by end users under guidance from Alpha Data support personnel. This entry point performs multiple single-byte reads of an I2C device (usually a PROM), saving the data read into a file (usually with a **.bin** extension).

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilI2cReadToFile indexOrSerial,flags,bus,device,address,count,pSaveFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

bus and *device* identify the I2C bus number and I2C device (on that bus), respectively. *address* is the address within the I2C device at which to begin reading. *count* is the number of consecutively-addressed bytes to read,

and *pSaveFilename* is the filename into which to save the data.

Usage example 1 - For the first device in the system, read 256 bytes from I2C bus 0, device 0x57 starting at address 0 within the device and save the data into the file **save_file.bin**:

```
avr2utilI2cReadToFile 0,0,0,0x57,0,256,"host:/path/to/save_file.bin"
```

3.1.1.12 avr2utilI2cVerifyWithFile entry point

The **avr2utilI2cVerifyWithFile** entry point is primarily for in-house use by Alpha Data, but may also be used by end users under guidance from Alpha Data support personnel. This entry point performs multiple single-byte reads of an I2C device (usually a PROM), and verifies that the data read matches the contents of a file (usually with a **.bin** extension).

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilI2cVerifyWithFile indexOrSerial,flags,bus,device,address,pVerifyFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

bus and *device* identify the I2C bus number and I2C device (on that bus), respectively. *address* is the address within the I2C device at which to begin reading, and *pVerifyFilename* is the name of the file against which the data read is compared.

Usage example 1 - For the first device in the system, verify that the data in I2C bus 0, device 0x57 starting at address 0 within the device matches the contents of the file **verify_file.bin**:

```
avr2utilI2cVerifyWithFile 0,0,0,0x57,0,"host:/path/to/verify_file.bin"
```

3.1.1.13 avr2utilI2cWriteFromFile entry point

The **avr2utilI2cWriteFromFile** entry point is primarily for in-house use by Alpha Data, but may also be used by end users under guidance from Alpha Data support personnel. This entry point performs multiple single-byte writes to an I2C device (usually a PROM), obtaining the data to be written from a file (usually with a **.bin** extension).

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilI2cWriteFromFile indexOrSerial,flags,bus,device,address,pDataFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

bus and *device* identify the I2C bus number and I2C device (on that bus), respectively. *address* is the address within the I2C device at which to begin writing, and *pDataFilename* is the name of the file containing the data to be written.

Usage example 1 - For the first device in the system (according to system-defined PCIe enumeration order), write I2C device 0x57 on bus 0, starting at address 0 within the device, with the contents of the file **data_file.bin**:

```
avr2utilI2cWriteFromFile 0,0,0,0x57,0,"host:/path/to/data_file.bin"
```

3.1.1.14 avr2utilI2cRead entry point

The **avr2utilI2cRead** entry point is primarily for in-house use by Alpha Data, but may also be used by end users under guidance from Alpha Data support personnel. This entry point performs an individual I2C read of one or more bytes from an I2C device, displaying the bytes read.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilI2cRead indexOrSerial,flags,bus,device,address,count
```

where *indexOrSerial* and *flags* are parameters as described above.

bus and *device* identify the I2C bus number and I2C device (on that bus), respectively. *address* is the address within the I2C device at which to begin reading and *count* is the length of the I2C read, in bytes.

Usage example 1 - For the first device in the system, perform a 4-byte read from I2C bus 1, device 0x30 at address 0x10 within the device and display the data:

```
avr2utilI2cRead 0,0,1,0x30,0x10,4
```

3.1.1.15 avr2utilI2cWrite entry point

The **avr2utilI2cWrite** entry point is primarily for in-house use by Alpha Data, but may also be used by end users under guidance from Alpha Data support personnel. This entry point performs an individual I2C write of one or more bytes from an I2C device, displaying the bytes read.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilI2cWrite indexOrSerial,flags,bus,device,address,count,...
```

where *indexOrSerial* and *flags* are parameters as described above.

bus and *device* identify the I2C bus number and I2C device (on that bus), respectively. *address* is the address within the I2C device at which to begin reading and *count* is the length of the I2C write, in bytes.

... represents one or more comma-separated bytes of data to be written; the number of such additional arguments **must** be equal to the value passed for *count*.

Usage example 1 - For the first device in the system, perform a 4-byte write to I2C bus 1, device 0x30 at address 0x10 within the device, where the individual bytes written are 0x12, 0x34, 0x56 and 0x78:

```
avr2utilI2cWrite 0,0,1,0x30,0x10,4,0x12,0x34,0x56,0x78
```

3.1.1.16 avr2utilUpdateBrdCfg entry point

NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt data required by the microcontroller's firmware.

The **avr2utilUpdateBrdCfg** entry point writes the block of data in the selected device, which contains configuration information used by the BoardMan2 firmware, with the contents of the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilUpdateBrdCfg indexOrSerial,flags,pBrdCfgFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pBrdCfgFilename is a nul-terminated string which specifies the file on the VxWorks host containing the board configuration information.

Usage example 1 - Update the board configuration area for the first device in the system:

```
avr2utilUpdateBrdCfg 0,0,"host:/path/to/boardman2_cfg_admxrcku1.bin"
```

Usage example 2 - Update the board configuration area for the device with serial number 100:

```
avr2utilUpdateBrdCfg 100,1,"host:/path/to/boardman2_cfg_admxrcku1.bin"
```

3.1.1.17 avr2utilVerifyBrdCfg entry point

The **avr2utilVerifyBrdCfg** entry point verifies that the block of data in the selected device, which contains configuration information used by the BoardMan2 firmware, matches the contents of the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilVerifyBrdCfg indexOrSerial,flags,pBrdCfgFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pBrdCfgFilename is a nul-terminated string which specifies the file on the VxWorks host containing the board configuration information to be used for verification.

Usage example 1 - Verify the board configuration area for the first device in the system:

```
avr2utilVerifyBrdCfg 0,0,"host:/path/to/boardman2_cfg_admxrcku1.bin"
```

Usage example 2 - Verify the board configuration area for the device with serial number 100:

```
avr2utilVerifyBrdCfg 100,1,"host:/path/to/boardman2_cfg_admxrcku1.bin"
```

3.1.1.18 avr2utilSaveBrdCfg entry point

The **avr2utilSaveBrdCfg** entry point reads the block of data in the selected device, which contains configuration information used by the BoardMan2 firmware, and saves it into the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilSaveBrdCfg indexOrSerial,flags,pBrdCfgSaveFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pBrdCfgSaveFilename is a nul-terminated string which specifies the name of a file on the VxWorks host to be created and written with the board configuration information.

Usage example 1 - Save the board configuration area for the first device in the system:

```
avr2utilSaveBrdCfg 0,0,"host:/path/to/boardman2_cfg_admxrcku1_saved.bin"
```

Usage example 2 - Save the board configuration area for the device with serial number 100:

```
avr2utilSaveBrdCfg 100,1,"host:/path/to/boardman2_cfg_admxrcku1_saved.bin"
```

3.1.1.19 avr2utilUpdateFirmware entry point

NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt the microcontroller's firmware.

The **avr2utilUpdateFirmware** entry point writes the BoardMan2 firmware of the microcontroller with the contents of the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilUpdateFirmware indexOrSerial,flags,pFirmwareFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pFirmwareFilename is a nul-terminated string which specifies the file on the VxWorks host containing the BoardMan2 firmware to be programmed into the device.

Usage example 1 - Program the BoardMan2 firmware for the first device in the system:

```
avr2utilUpdateFirmware 0,0,"host:/path/to/boardman2_a.b.c.d.bin"
```

Usage example 2 - Program the BoardMan2 firmware for the device with serial number 100:

```
avr2utilUpdateFirmware 100,1,"host:/path/to/boardman2_a.b.c.d.bin"
```

3.1.1.20 avr2utilVerifyFirmware entry point

The **avr2utilVerifyFirmware** entry point verifes the BoardMan2 firmware of the microcontroller against the contents of the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilVerifyFirmware indexOrSerial,flags,pFirmwareFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pFirmwareFilename is a nul-terminated string which specifies the file on the VxWorks host which is to be used to verify the BoardMan2 firmware in the selected device.

Usage example 1 - Verify the BoardMan2 firmware for the first device in the system:

```
avr2utilVerifyFirmware 0,0,"host:/path/to/boardman2_a.b.c.d.bin"
```

Usage example 2 - Verify the BoardMan2 firmware for the device with serial number 100:

```
avr2utilVerifyFirmware 100,1,"host:/path/to/boardman2_a.b.c.d.bin"
```

3.1.1.21 avr2utilSaveFirmware entry point

The **avr2utilSaveFirmware** entry point reads the BoardMan2 firmware of the microcontroller and saves it into the specified file (usually with a **.bin** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilSaveFirmware indexOrSerial,flags,pFirmwareSaveFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pFirmwareSaveFilename is a nul-terminated string which specifies the name of a file on the VxWorks host to be created and written with the BoardMan2 firmware in the selected device.

NOTE: The entire region of nonvolatile memory allocated for firmware is saved, regardless of the actual code size of the firmware. This means that files containing saved firmware (i.e. read out of a board using this entry point) are in general larger than files used to update firmware.

Usage example 1 - Save the BoardMan2 firmware for the first device in the system:

```
avr2utilSaveFirmware 0,0,"host:/path/to/boardman2_a.b.c.d_saved.bin"
```

Usage example 2 - Save the BoardMan2 firmware for the device with serial number 100:

```
avr2utilSaveFirmware 100,1,"host:/path/to/boardman2_a.b.c.d_saved.bin"
```

3.1.1.22 avr2utilUpdateVPD entry point

NOTE: This entry point should be used only under guidance from Alpha Data, because incorrect usage can corrupt the board's VPD.

The **avr2utilUpdateVPD** entry point writes the Vital Product Data (VPD) with the contents of the specified file (usually with a **.bin** or **.seg** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilUpdateVPD indexOrSerial,flags,pVPDFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pVPDFilename is a nul-terminated string which specifies the file on the VxWorks host containing the VPD to be programmed into the device.

Usage example 1 - Update the VPD for the first device in the system:

```
avr2utilUpdateVPD 0,0,"host:/path/to/vpd.seg"
```

Usage example 2 - Update the VPD for the device with serial number 100:

```
avr2utilUpdateVPD 100,1,"host:/path/to/vpd.seg"
```

3.1.1.23 avr2utilVerifyVPD entry point

The **avr2utilVerifyVPD** entry point verifies the Vital Product Data (VPD) in the selected device against the contents of the specified file (usually with a **.bin** or **.seg** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilVerifyVPD indexOrSerial,flags,pVPDFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pVPDFilename is a nul-terminated string which specifies the file on the VxWorks host which is to be used to verify the VPD in the selected device.

Usage example 1 - Verify the VPD for the first device in the system:

```
avr2utilVerifyVPD 0,0,"host:/path/to/vpd.seg"
```

Usage example 2 - Verify the VPD for the device with serial number 100:

```
avr2utilVerifyVPD 100,1,"host:/path/to/vpd.seg"
```

3.1.1.24 avr2utilSaveVPD entry point

The **avr2utilSaveVPD** entry point reads the Vital Product Data (VPD) in the selected device and saves it into the specified file (usually with a **.bin** or **.seg** extension).

This entry point requires the microcontroller to be in Service Mode, at least temporarily. See 3.1.3 below.

The general form of this entry point is:

```
avr2utilSaveVPD indexOrSerial,flags,pVPDSaveFilename
```

where *indexOrSerial* and *flags* are parameters as described above.

pVPDSaveFilename is a nul-terminated string which specifies the name of a file on the VxWorks host to be created and written with the VPD in the selected device.

Usage example 1 - Save the VPD for the first device in the system:

```
avr2utilSaveVPD 0,0,"host:/path/to/vpd_saved.seg"
```

Usage example 2 - Save the VPD for the device with serial number 100:

```
avr2utilSaveVPD 100,1,"host:/path/to/vpd_saved.seg"
```

3.1.1.25 avr2utilDisplayVPD entry point

The **avr2utilDisplayVPD** entry point reads the Vital Product Data (VPD) in the selected device, from the nonvolatile memory in a board in which it resides, and displays it in human-readable form.

This entry point works whether or not the microcontroller is in Service Mode.

The general form of this entry point is:

```
avr2utilDisplayVPD indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Display the VPD in human-readable form for the first device in the system:

```
avr2utilDisplayVPD
```

Usage example 2 - Display the VPD in human-readable form for the device with serial number 100:

```
avr2utilDisplayVPD 100,1
```

3.1.1.26 avr2utilDisplayVPDRaw entry point

The **avr2utilDisplayVPDRaw** entry point reads the Vital Product Data (VPD) in the selected device, from the nonvolatile memory in a board in which it resides, and displays it as raw bytes.

This entry point works whether or not the microcontroller is in Service Mode.

The general form of this entry point is:

```
avr2utilDisplayVPDRaw indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Display the VPD in raw form for the first device in the system:

```
avr2utilDisplayVPDRaw
```

Usage example 2 - Display the VPD in raw form for the device with serial number 100:

```
avr2utilDisplayVPDRaw 100,1
```

3.1.1.27 avr2utilDisplaySensors entry point

The **avr2utilDisplaySensors** entry point reads the Sensor Page in the selected device and displays it in human-readable form.

This entry point requires the microcontroller not to be in Service Mode, at least temporarily, in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilDisplaySensors indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Display the Sensor Page in human-readable form for the first device in the system:

```
avr2utilDisplaySensors
```

Usage example 2 - Display the Sensor Page in human-readable form for the device with serial number 100:

```
avr2utilDisplaySensors 100,1
```

3.1.1.28 avr2utilDisplaySensorsRaw entry point

The **avr2utilDisplaySensorsRaw** entry point reads the Sensor Page in the selected device and displays it in human-readable form.

This entry point requires the microcontroller not to be in Service Mode, at least temporarily, in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilDisplaySensorsRaw indexOrSerial,flags
```

where *indexOrSerial* and *flags* are parameters as described above.

Usage example 1 - Display the Sensor Page in raw form for the first device in the system:

```
avr2utilDisplaySensorsRaw
```

Usage example 2 - Display the Sensor Page in raw form for the device with serial number 100:

```
avr2utilDisplaySensorsRaw 100,1
```

3.1.1.29 avr2utilOverrideSensor entry point

The **avr2utilOverrideSensor** entry point facilitates firmware testing by Alpha Data. It injects a value into a particular sensor, overriding its natural value. The sensor remains overridden until at least one of the following occurs:

- A power cycle, including removal and reapplication of standby power.
- The microcontroller enters and exits Service Mode, either using the **enter-service-mode** and **exit-service-mode** entry points or by toggling the physical Service Mode switch.
- The **avr2utilReleaseSensor** entry point is used to explicitly release the sensor.

This entry point requires the microcontroller not to be in Service Mode, at least temporarily, in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilOverrideSensor indexOrSerial,flags,sensorIndex,overrideValue
```

where *indexOrSerial* and *flags* are parameters as described above. The *sensorIndex* parameter identifies the sensor to be overridden, and *overrideValue* is the sensor-specific value injected into the sensor.

Usage example 1 - Override sensor 11 of the first device in the system with the value 1000:

```
avr2utilOverrideSensor 0,0,11,1000
```

Usage example 2 - Override sensor 3 of the device with serial number 100 with the value 0:

```
avr2utilOverrideSensor 100,1,3,0
```

3.1.1.30 avr2utilReleaseSensor entry point

The **avr2utilOverrideSensor** entry point facilitates firmware testing by Alpha Data. It undoes the effect of a call to **avr2utilOverrideSensor**, returning the sensor to normal operation.

This entry point requires the microcontroller not to be in Service Mode, at least temporarily, in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilReleaseSensor indexOrSerial,flags,sensorIndex
```

where *indexOrSerial* and *flags* are parameters as described above. The *sensorIndex* parameter identifies the sensor to be returned to nornmal operation.

Usage example 1 - Return sensor 11 of the first device in the system to normal operation:

```
avr2utilReleaseSensor 0,0,11
```

Usage example 2 - Return sensor 3 of the device with serial number 100 to normal operation:

```
avr2utilReleaseSensor 100,1,3
```

3.1.1.31 avr2utilSPIInfo entry point

The **avr2utilSPIInfo** entry point reads the Serial Flash Discoverable Parameters (SFPD) information from a SPI Flash chip that is accessible from the microcontroller, and displays it in human-readable form.

This entry point requires the microcontroller not to be in Service Mode, at least temporarily, in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilSPIInfo indexOrSerial,flags,chipIndex
```

where *indexOrSerial* and *flags* are parameters as described above. The *chipIndex* parameter identifies particular SPI Flash chip whose SFDP information is to be displayed.

Usage example 1 - Display the SFDP information for the 2nd SPI Flash chip (index 1) of the first device in the system:

```
avr2utilSPIInfo 0,0,1
```

3.1.1.32 avr2utilSPIRaw entry point

The **avr2utilSPIRaw** entry point is primarily for in-house use by Alpha Data for test purposes. This entry point performs a transaction consisting of zero or more bytes written to an SPI Flash chip followed by zero or more bytes read from the same SPI Flash chip, with the bytes to be written obtained from variable argument list (denoted by ...). The bytes read are displayed in raw form, i.e. as individual byte values.

The first argument is the zero-based index of the SPI Flash chip. The second argument is the number of bytes read, and the third and later arguments are the bytes written to the SPI Flash chip.

This entry point requires the microcontroller not to be in Service Mode in order to succeed. See 3.1.2 below.

The general form of this entry point is:

```
avr2utilSPIRaw indexOrSerial,flags,chipIndex,readCount,writeCount,...
```

where *indexOrSerial* and *flags* are parameters as described above. The *chipIndex* parameter identifies particular SPI Flash chip whose SFDP information is to be displayed.

readCount and *writeCount* respectively are the number of bytes to read & display and the number of bytes to write.

Following *writeCount*, the variable number of bytes to write (passed in VxWorks shell as normal int-sized values) must be passed. The number of byte values passed must be exactly equal to *writeCount*.

Usage example 1 - For the first device in the system, read and display (in the form of raw bytes) the 8-byte SFDP header from the first SPI Flash chip (index 0). JEDEC document **JESD216B** describes the SFDP mechanism in SPI Flash chips.

```
avr2utilSPIRaw 0,0,0,8,5,0x5A,0,0,0,0
```

3.1.2 Entry points requiring non-Service Mode

The following entry points can only perform their main operation while **not** in Service Mode:

- **avr2utilGetClk, avr2utilGetClkNV, avr2utilSetClk, avr2utilSetClkNV**
- **avr2utilI2cReadToFile, avr2utilI2cVerifyWithFile, avr2utilI2cWriteFromFile**
- **avr2utilI2cRead, avr2utilI2cWrite**
- **avr2utilDisplayVPD, avr2utilDisplayVPDRaw**
- **avr2utilDisplaySensors, avr2utilDisplaySensorsRaw**
- **avr2utilOverrideSensor, avr2utilReleaseSensor**
- **avr2utilSPIInfo, avr2utilSPIRaw**

If the microcontroller is in Service Mode, these entry points temporarily switch the microcontroller out of Service

Mode, perform the necessary operations, and then switch the microcontroller back into Service Mode.

3.1.3 Entry points requiring Service Mode

The following entry points can only perform their main operation while in Service Mode:

- **avr2utilUpdateBrdCfg, avr2utilSaveBrdCfg, avr2utilVerifyBrdCfg**
- **avr2utilUpdateFirmware, avr2utilSaveFirmware, avr2utilVerifyFirmware**
- **avr2utilUpdateVPD, avr2utilSaveVPD, avr2utilVerifyVPD**

If the microcontroller in PCIe mode is not in Service Mode, these entry points temporarily switch the microcontroller into Service Mode, perform the necessary operations and then switch the microcontroller out of Service Mode.

3.2 BITSTRIP utility

VxWorks kernel shell entry points

```
int admxrc3BitstripHelp()
int admxrc3Bitstrip(pInputFilename[, pOutputFilename])
```

where the parameters are:

const char* pInputFilename	Specifies the input bitstream (**.bit**) filename.
const char* pOutputFilename	Optionally specifies out filename (e.g. **.bin** extension).

Summary

Reads an FPGA bitstream (**.bit**) file, displays certain information from the header, and optionally writes the SelectMap data (without the header) to another file.

Description

To simply display information from a **.bit** file's header, use

admxrc3Bitstrip *pInputFilename*

To display information from a **.bit** file's header and write the SelectMap data to another file, use

admxrc3Bitstrip *pInputFilename*,*pOutputFilename*

The data written to *pOutputFilename* is suitable for sending to a target FPGA using ADMXRC3_ConfigureFromBuffer.

Return values

When **BITSTRIP** runs successfully, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_INSUFFICIENT_ARGS	2	Not enough positional arguments.
EXIT_LOAD_BIT_ERROR	3	Failed to read the input **.bit** file.
EXIT_SAVE_BIN_ERROR	4	Failed to write the output file.

Table 3 : Return values for BITSTRIP utility

3.3 DMADUMP utility

VxWorks kernel shell entry points

```
int admxrc3DmaDumpHelp()
int admxrc3DmaFB (indexOrSerial, flags, channel, address,      count, val8)
int admxrc3DmaXFB(indexOrSerial, flags, channel, address(ULL), count, val8)
int admxrc3DmaFW (indexOrSerial, flags, channel, address,      count, val16)
int admxrc3DmaXFW(indexOrSerial, flags, channel, address(ULL), count, val16)
int admxrc3DmaFD (indexOrSerial, flags, channel, address,      count, val32)
int admxrc3DmaXFD(indexOrSerial, flags, channel, address(ULL), count, val32)
int admxrc3DmaFQ (indexOrSerial, flags, channel, address,      count, val64(ULL))
int admxrc3DmaXFQ(indexOrSerial, flags, channel, address(ULL), count, val64(ULL))
int admxrc3DmaRB (indexOrSerial, flags, channel, address,      count)
int admxrc3DmaXRB(indexOrSerial, flags, channel, address(ULL), count)
int admxrc3DmaRW (indexOrSerial, flags, channel, address,      count)
int admxrc3DmaXRW(indexOrSerial, flags, channel, address(ULL), count)
int admxrc3DmaRD (indexOrSerial, flags, channel, address,      count)
int admxrc3DmaXRD(indexOrSerial, flags, channel, address(ULL), count)
int admxrc3DmaRQ (indexOrSerial, flags, channel, address,      count)
int admxrc3DmaXRQ(indexOrSerial, flags, channel, address(ULL), count)
int admxrc3DmaWB (indexOrSerial, flags, channel, address,      count, ...)
int admxrc3DmaXWB(indexOrSerial, flags, channel, address(ULL), count, ...)
int admxrc3DmaWW (indexOrSerial, flags, channel, address,      count, ...)
int admxrc3DmaXWW(indexOrSerial, flags, channel, address(ULL), count, ...)
int admxrc3DmaWD (indexOrSerial, flags, channel, address,      count, ...)
int admxrc3DmaXWD(indexOrSerial, flags, channel, address(ULL), count, ...)
int admxrc3DmaWQ (indexOrSerial, flags, channel, address,      count, ...(ULL))
int admxrc3DmaXWQ(indexOrSerial, flags, channel, address(ULL), count, ...(ULL))
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int channel	Index of the DMA channel / engine to be used for the DMA transfer.
unsigned int address	The starting OCP/AXI address within the FPGA to be used for the DMA transfer.
unsigned long long address(ULL)	The starting OCP/AXI address within the FPGA to be used for the DMA transfer. This is **unsigned long long** in order to allow for 64-bit addresses.
unsigned int count	The number of bytes of data to transfer.
unsigned int val8	Fill value which is interpreted as being 8 bits wide.
unsigned int val16	Fill value which is interpreted as being 16 bits wide.
unsigned int val32	Fill value which is interpreted as being 32 bits wide.
unsigned long long val64	Fill value which is interpreted as being 64 bits wide.

...	A variable number of **unsigned int** values, which are interpreted as 8-bit / 16-bit / 32-bit write values depending on which **admxrc3DmaW*** entry point is used.
...(ULL)	A variable number of **unsigned long long** write values, which are interpreted as 64-bit write values.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x20	**FLAG_BIGENDIAN**	Big-endian byte order is used, as opposed to little-endian.

Summary

Displays data read from a target FPGA using a DMA engine, or writes data to a target FPGA using a DMA engine.

Description

The **DMADUMP** utility operates in one of three modes:

- Reading data from a target FPGA using a DMA engine and displaying it; for this mode, use the **admxrc3DmaR*** or **admxrc3DmaXR*** entry points.
- Writing data to a target FPGA using a DMA engine; for this mode, use the **admxrc3DmaW*** or **admxrc3DmaXW*** entry points.
- Filling an address region in a target FPGA using a DMA engine with a particular value; for this mode, use the **admxrc3DmaF*** or **admxrc3DmaXF*** commands.

The entry point **admxrc3DmaDumpHelp** displays a brief help message, listing the available entry points and arguments.

If *flags* contains **FLAG_BIGENDIAN**, the **DMADUMP** utility reads or writes numeric values in big-endian byte ordering convention as opposed to little-endian (the default).

Read entry points

The particular read entry point used implies the word width for displaying data:

- **admxrc3DmaRB** *indexOrSerial*, *flags*, *channel*, *address*, *count*

  admxrc3DmaXRB *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*

  Read a block of data and display it as bytes.
- **admxrc3DmaRW** *indexOrSerial*, *flags*, *channel*, *address*, *count*

  admxrc3DmaXRW *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*

  Read a block of data and display it as words (16-bit).
- **admxrc3DmaRD** *indexOrSerial*, *flags*, *channel*, *address*, *count*

  admxrc3DmaXRD *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*

  Read a block of data and display it as doublewords (32-bit).
- **admxrc3DmaRQ** *indexOrSerial*, *flags*, *channel*, *address*, *count*

  admxrc3DmaXRQ *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*

  Read a block of data and display it as quadwords (64-bit).

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *channel* parameter is the index of the DMA channel/engine to be used for the DMA transfer.

The *address* parameter is the OCP/AXI address, in the selected DMA channel/engine's address space, at which to begin reading data. If one of the **X** entry points such as **admxrc3DmaXRB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to read.

Example 1 - Using the first device in the system (index 0), read and display 256 bytes from address 0x1000 via DMA channel 1:

```
admxrc3DmaRB 0,0,1,0x1000,0x100
```

Example 2 - Using the device with serial number 100, read 768 bytes displayed as quadwords from address 0xF_00000000 via DMA channel 2:

```
admxrc3DmaXRQ 100,1,2,(long long)0xF00000000,768
```

Example 3 - Using the second device in the system (index 1), read 96 bytes displayed as big-endian doublewords from address 0x1000 via DMA channel 1:

```
admxrc3DmaRD 1,2,1,0x1000,96
```

Write entry points

The particular write entry point used implies the word width for writing data:

- **admxrc3DmaWB** *indexOrSerial*, *flags*, *channel*, *address*, *count*, ...

  admxrc3DmaXWB *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as bytes (8-bit).
- **admxrc3DmaWW** *indexOrSerial*, *flags*, *channel*, *address*, *count*, ...

  admxrc3DmaXWW *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as words (16-bit).
- **admxrc3DmaWD** *indexOrSerial*, *flags*, *channel*, *address*, *count*, ...

  admxrc3DmaXWD *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as doublewords (32-bit).
- **admxrc3DmaWQ** *indexOrSerial*, *flags*, *channel*, *address*, *count*, ...

  admxrc3DmaXWQ *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned long long** values, and written as quadwords (64-bit).

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *channel* parameter is the index of the DMA channel/engine to be used for the DMA transfer.

The *address* parameter is the OCP/AXI address, in the selected DMA channel/engine's address space, at which to begin writing data. If one of the **X** entry points such as **admxrc3DmaXWB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to write.

The values to be written must then be provided as additional parameters, and must be sufficient additional parameters to satisfy the byte count specified by the *count* parameter. In the case of the **admxrc3DmaWQ** and **admxrc3DmaXWQ** entry points, the 64-bit write values must be passed using a cast; see usage example 2 below.

Note that if too few values are provided to satisfy the byte count, some VxWorks platforms may generate an exception due to the stack bounds being breached, and this may cause the VxWorks shell task to be killed and restarted.

Example 1 - Using the first device in the system (index 0), write the sequence of seven bytes { 1, 2, 3, 4, 5, 6, 7 } at address 0x1000 via DMA channel 1:

```
admxrc3DmaWB 0,0,1,0x1000,7,1,2,3,4,5,6,7
```

Example 2 - Using the device with serial number 100, write two quadwords { 0x01234567_89ABCDEF, 0x5A5A5A5A_5A5A5A5A } at address 0xF_00000000 via DMA channel 2:

```
admxrc3DmaXWQ 100,1,2,(long long)0xF00000000,16,(long long)0x0123456789ABCDEF,(long long)0x5A5A5A5A5A5A5A5A
```

Example 3 - Using the second device in the system (index 1), write four big-endian doublewords at address 0x1000 via DMA channel 1:

```
admxrc3DmaWD 1,2,1,0x1000,16,0x12345678,0x23456789,0x3456789a,0x456789ab
```

Fill entry points

The particular write entry point used implies the word width of the fill value:

- **admxrc3DmaFB** *indexOrSerial*, *flags*, *channel*, *address*, *count*, val8

  admxrc3DmaXFB *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, val8

  Fills a block of memory (in the FPGA) with a particular byte value.
- **admxrc3DmaFW** *indexOrSerial*, *flags*, *channel*, *address*, *count*, val16

  admxrc3DmaXFW *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, val16

  Fills a block of memory (in the FPGA) with a particular word (16-bit) value.
- **admxrc3DmaFD** *indexOrSerial*, *flags*, *channel*, *address*, *count*, val32

  admxrc3DmaXFD *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, val32

  Fills a block of memory (in the FPGA) with a particular doubleword (32-bit) value.
- **admxrc3DmaFQ** *indexOrSerial*, *flags*, *channel*, *address*, *count*, val64(ULL)

  admxrc3DmaXFQ *indexOrSerial*, *flags*, *channel*, *address(ULL)*, *count*, val64(ULL)

  Fills a block of memory (in the FPGA) with a particular quadword (64-bit) value.

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *channel* parameter is the index of the DMA channel/engine to be used for the DMA transfer.

The *address* parameter is the OCP/AXI address, in the selected DMA channel/engine's address space, at which to begin filling. If one of the **X** entry points such as **admxrc3DmaXFB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to fill.

The final parameter is the fill value. This is interpreted as a byte, word, doubleword or quadword depending on which entry point is used. In the case of the **admxrc3DmaFQ** and **admxrc3DmaXFQ** entry points, the 64-bit fill value must be passed using a cast; see usage example 2 below.

Example 1 - Using the first device in the system (index 0), fill 17 bytes with the value 0xCD starting at address 0x100D via DMA channel 1:

```
admxrc3DmaFB 0,0,1,0x100D,17,0xCD
```

Example 2 - Using the device with serial number 100, fill a megabyte starting at address 0xF_00000000 with the quadword value 0xDEADBEEFCAFEFACE via DMA channel 2:

```
admxrc3DmaXFQ 100,1,2,(long long)0xF00000000,0x100000,(long long)0xDEADBEEFCAFEFACE
```

Example 3 - Using the second device in the system (index 1), fill 16 bytes using a big-endian fill value of 0x12345678 starting at address 0x1000 via DMA channel 1:

```
admxrc3DmaFD 1,2,1,0x1000,16,0x12345678
```

Example session

Assuming that the target FPGA is currently configured with an FPGA bitstream which has a RAM-like region in the OCP/AXI address space of DMA channel 0 at address 0x80000, an example session looks like this:

```
-> admxrc3DmaFB 0,0,0,0x80000,0x20,0xee
value = 0 = 0x0
-> admxrc3DmaRD 0,0,0,0x80000,0x40
Dump of memory at 0x00000000_00080000 + 64(0x40) bytes:
                            00       04       08       0C
0x00000000_00080000: EEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE ................
0x00000000_00080010: EEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE ................
0x00000000_00080020: 00000000 00000000 00000000 00000000 ................
0x00000000_00080030: 00000000 00000000 00000000 00000000 ................
value = 0 = 0x0
-> admxrc3DmaWD 0,0,0,0x80004,0x4,0x12345678
value = 0 = 0x0
-> admxrc3DmaRD 0,0,0,0x80000,0x40
Dump of memory at 0x00000000_00080000 + 64(0x40) bytes:
                            00       04       08       0C
0x00000000_00080000: EEEEEEEE 12345678 EEEEEEEE EEEEEEEE ....xV4.........
0x00000000_00080010: EEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE ................
0x00000000_00080020: 00000000 00000000 00000000 00000000 ................
0x00000000_00080030: 00000000 00000000 00000000 00000000 ................
value = 0 = 0x0
```

Remarks

Each DMA engine has its own address space. This means that in general, unless an FPGA design explicitly makes a shared resource available to multiple DMA engines, writing data using one DMA engine and then attempting to read it back using a different DMA engine will not return the data just written.

Memory access windows have a separate address space from that of each DMA engine. This means that in general, writing data using the **DMADUMP** utility and attempting to read it back via the DUMP utility will not return the same data. However, it is possible to create an FPGA design which explicitly makes a shared resource available via both a memory access window and a DMA channel. In that case, data written by one window/DMA channel can be read back via another window/DMA channel.

Return value

When **DMADUMP** successfully executes the requested function, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_ALLOCATION_FAILURE	3	A memory allocation failed.
EXIT_DEVICE_OPEN_ERROR	4	Failed to open ADMXRC3 device.
EXIT_WRITEDMA_ERROR	5	A DMA write call failed.
EXIT_READDMA_ERROR	6	A DMA read call failed.
ERROR_COUNT_TOO_LARGE	7	Requested byte count is too large.

Table 4 : Return values for DMADUMP utility

3.4 DUMP utility

VxWorks kernel shell entry points

```
int admxrc3DumpHelp()
int admxrc3FB (indexOrSerial, flags, window, address,      count, val8)
int admxrc3XFB(indexOrSerial, flags, window, address(ULL), count, val8)
int admxrc3FW (indexOrSerial, flags, window, address,      count, val16)
int admxrc3XFW(indexOrSerial, flags, window, address(ULL), count, val16)
int admxrc3FD (indexOrSerial, flags, window, address,      count, val32)
int admxrc3XFD(indexOrSerial, flags, window, address(ULL), count, val32)
int admxrc3FQ (indexOrSerial, flags, window, address,      count, val64(ULL))
int admxrc3XFQ(indexOrSerial, flags, window, address(ULL), count, val64(ULL))
int admxrc3RB (indexOrSerial, flags, window, address,      count)
int admxrc3XRB(indexOrSerial, flags, window, address(ULL), count)
int admxrc3RW (indexOrSerial, flags, window, address,      count)
int admxrc3XRW(indexOrSerial, flags, window, address(ULL), count)
int admxrc3RD (indexOrSerial, flags, window, address,      count)
int admxrc3XRD(indexOrSerial, flags, window, address(ULL), count)
int admxrc3RQ (indexOrSerial, flags, window, address,      count)
int admxrc3XRQ(indexOrSerial, flags, window, address(ULL), count)
int admxrc3WB (indexOrSerial, flags, window, address,      count, ...)
int admxrc3XWB(indexOrSerial, flags, window, address(ULL), count, ...)
int admxrc3WW (indexOrSerial, flags, window, address,      count, ...)
int admxrc3XWW(indexOrSerial, flags, window, address(ULL), count, ...)
int admxrc3WD (indexOrSerial, flags, window, address,      count, ...)
int admxrc3XWD(indexOrSerial, flags, window, address(ULL), count, ...)
int admxrc3WQ (indexOrSerial, flags, window, address,      count, ...(ULL))
int admxrc3XWQ(indexOrSerial, flags, window, address(ULL), count, ...(ULL))
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int window	Index of the memory access window to be used for the data transfer.
unsigned int address	The starting OCP/AXI address within the FPGA to be used for the data transfer.
unsigned long long address(ULL)	The starting OCP/AXI address within the FPGA to be used for the data transfer. This is **unsigned long long** in order to allow for 64-bit addresses.
unsigned int count	The number of bytes of data to transfer.
unsigned int val8	Fill value which is interpreted as being 8 bits wide.
unsigned int val16	Fill value which is interpreted as being 16 bits wide.
unsigned int val32	Fill value which is interpreted as being 32 bits wide.
unsigned long long val64	Fill value which is interpreted as being 64 bits wide.

...	A variable number of **unsigned int** values, which are interpreted as 8-bit / 16-bit / 32-bit write values depending on which **admxrc3W*** entry point is used.
...(ULL)	A variable number of **unsigned long long** write values, which are interpreted as 64-bit write values.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x20	**FLAG_BIGENDIAN**	Big-endian byte order is used, as opposed to little-endian.

Summary

Displays data read from a target FPGA using a memory access window (CPU-initiated reads and writes), or writes data to a target FPGA using a memory access window.

Description

The **DUMP** utility operates in one of three modes:

- Reading data from a target FPGA and displaying it; for this mode, use the **admxrc3R*** or **admxrc3XR*** entry points.
- Writing data to a target FPGA; for this mode, use the **admxrc3W*** or **admxrc3XW*** entry points.
- Filling an address region in a target FPGA with a particular value; for this mode, use the **admxrc3F*** or **admxrc3XF*** commands.

The entry point **admxrc3DumpHelp** displays a brief help message, listing the available entry points and arguments.

If *flags* contains **FLAG_BIGENDIAN**, the **DUMP** utility reads or writes numeric values in big-endian byte ordering convention as opposed to little-endian (the default).

Read entry points

The particular read entry point used implies the word width for displaying data:

- **admxrc3RB** *indexOrSerial*, *flags*, *window*, *address*, *count*

  admxrc3XRB *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*

  Read a block of data and display it as bytes.
- **admxrc3RW** *indexOrSerial*, *flags*, *window*, *address*, *count*

  admxrc3XRW *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*

  Read a block of data and display it as words (16-bit).
- **admxrc3RD** *indexOrSerial*, *flags*, *window*, *address*, *count*

  admxrc3XRD *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*

  Read a block of data and display it as doublewords (32-bit).
- **admxrc3RQ** *indexOrSerial*, *flags*, *window*, *address*, *count*

  admxrc3XRQ *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*

  Read a block of data and display it as quadwords (64-bit).

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *window* parameter is the index of memory access window to be used for reading data.

The *address* parameter is the OCP/AXI address, in the selected window's address space, at which to begin reading data. If one of the **X** entry points such as **admxrc3XRB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to read.

Example 1 - Using the first device in the system (index 0), read and display 256 bytes from address 0x1000 via window 1:

```
admxrc3RB 0,0,1,0x1000,0x100
```

Example 2 - Using the device with serial number 100, read 768 bytes displayed as quadwords from address 0xF_00000000 via window 2:

```
admxrc3XRQ 100,1,2,(long long)0xF00000000,768
```

Example 3 - Using the second device in the system (index 1), read 96 bytes displayed as big-endian doublewords from address 0x1000 via window 0:

```
admxrc3RD 1,2,0,0x1000,96
```

Write entry points

The particular write entry point used implies the word width for writing data:

- **admxrc3WB** *indexOrSerial*, *flags*, *window*, *address*, *count*, ...

  admxrc3XWB *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as bytes (8-bit).
- **admxrc3WW** *indexOrSerial*, *flags*, *window*, *address*, *count*, ...

  admxrc3XWW *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as words (16-bit).
- **admxrc3WD** *indexOrSerial*, *flags*, *window*, *address*, *count*, ...

  admxrc3XWD *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned int** values, and written as doublewords (32-bit).
- **admxrc3WQ** *indexOrSerial*, *flags*, *window*, *address*, *count*, ...

  admxrc3XWQ *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, ...

  Write values are supplied as **unsigned long long** values, and written as quadwords (64-bit).

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *window* parameter is the index of the memory access window to be used for writing data.

The *address* parameter is the OCP/AXI address, in the selected windows's address space, at which to begin writing data. If one of the **X** entry points such as **admxrc3XWB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to write.

The values to be written must then be provided as additional parameters, and must be sufficient additional parameters to satisfy the byte count specified by the *count* parameter. In the case of the **admxrc3WQ** and **admxrc3XWQ** entry points, the 64-bit write values must be passed using a cast; see usage example 2 below.

Note that if too few values are provided to satisfy the byte count, some VxWorks platforms may generate an exception due to the stack bounds being breached, and this may cause the VxWorks shell task to be killed and restarted.

Example 1 - Using the first device in the system (index 0), write the sequence of seven bytes { 1, 2, 3, 4, 5, 6, 7 } at address 0x1000 via window 1:

```
admxrc3WB 0,0,1,0x1000,7,1,2,3,4,5,6,7
```

Example 2 - Using the device with serial number 100, write two quadwords { 0x01234567_89ABCDEF, 0x5A5A5A5A_5A5A5A5A } at address 0xF_00000000 via window 0:

```
admxrc3XWQ 100,1,0,(long long)0xF00000000,16,(long long)0x0123456789ABCDEF,(long l
ong)0x5A5A5A5A5A5A5A5A
```

Example 3 - Using the second device in the system (index 1), write four big-endian doublewords at address 0x1000 via window 0:

```
admxrc3WD 1,2,0,0x1000,16,0x12345678,0x23456789,0x3456789a,0x456789ab
```

Fill entry points

The particular write entry point used implies the word width of the fill value:

- **admxrc3FB** *indexOrSerial*, *flags*, *window*, *address*, *count*, val8

  admxrc3XFB *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, val8

  Fills a block of memory (in the FPGA) with a particular byte value.
- **admxrc3FW** *indexOrSerial*, *flags*, *window*, *address*, *count*, val16

  admxrc3XFW *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, val16

  Fills a block of memory (in the FPGA) with a particular word (16-bit) value.
- **admxrc3FD** *indexOrSerial*, *flags*, *window*, *address*, *count*, val32

  admxrc3XFD *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, val32

  Fills a block of memory (in the FPGA) with a particular doubleword (32-bit) value.
- **admxrc3FQ** *indexOrSerial*, *flags*, *window*, *address*, *count*, val64(ULL)

  admxrc3XFQ *indexOrSerial*, *flags*, *window*, *address(ULL)*, *count*, val64(ULL)

  Fills a block of memory (in the FPGA) with a particular quadword (64-bit) value.

The first two parameters, *indexOrSerial* and *flags* are common to most entry points and are as described above.

The *window* parameter is the index of the window to be used for writing data.

The *address* parameter is the OCP/AXI address, in the selected window's address space, at which to begin filling. If one of the **X** entry points such as **admxrc3XFB** is used, an **unsigned long long** value is required for the address, which permits a 64-bit OCP/AXI address to be specified. Note that passing an **unsigned long long** parameter in the VxWorks shell requires a cast; see usage example 2 below.

The *count* parameter is the number of bytes to fill.

The final parameter is the fill value. This is interpreted as a byte, word, doubleword or quadword depending on which entry point is used. In the case of the **admxrc3FQ** and **admxrc3XFQ** entry points, the 64-bit fill value must be passed using a cast; see usage example 2 below.

Example 1 - Using the first device in the system (index 0), fill 17 bytes with the value 0xCD starting at address 0x100D via window 1:

```
admxrc3FB 0,0,1,0x100D,17,0xCD
```

Example 2 - Using the device with serial number 100, fill a megabyte starting at address 0xF_00000000 with the quadword value 0xDEADBEEFCAFEFACE via window 0:

```
admxrc3XFQ 100,1,0,(long long)0xF00000000,0x100000,(long long)0xDEADBEEFCAFEFACE
```

Example 3 - Using the second device in the system (index 1), fill 16 bytes using a big-endian fill value of

0x12345678 starting at address 0x1000 via window 0:

```
admxrc3FD 1,2,0,0x1000,16,0x12345678
```

Example session

Assuming that the target FPGA is currently configured with an FPGA bitstream which has a RAM-like region in the OCP/AXI address space of window 0 at address 0x80000, an example session looks like this:

```
-> admxrc3RD 0,0,0,0x80000,0x40
Dump of memory at 0x00000000_00080000 + 64(0x40) bytes:
                           00       04       08       0C
0x00000000_00080000: 00000000 00000000 00000000 00000000 ................
0x00000000_00080010: 00000000 00000000 00000000 00000000 ................
0x00000000_00080020: 00000000 00000000 00000000 00000000 ................
0x00000000_00080030: 00000000 00000000 00000000 00000000 ................
value = 0 = 0x0
-> admxrc3FW 0,0,0,0x80000,0x20,0x1234
value = 0 = 0x0
-> admxrc3RD 0,0,0,0x80000,0x40
Dump of memory at 0x00000000_00080000 + 64(0x40) bytes:
                           00       04       08       0C
0x00000000_00080000: 12341234 12341234 12341234 12341234 4.4.4.4.4.4.4.4.
0x00000000_00080010: 12341234 12341234 12341234 12341234 4.4.4.4.4.4.4.4.
0x00000000_00080020: 00000000 00000000 00000000 00000000 ................
0x00000000_00080030: 00000000 00000000 00000000 00000000 ................
value = 0 = 0x0
-> admxrc3WD 0,0,0,0x80004,0x8,0xdeadbeef,0xcafeface
value = 0 = 0x0
-> admxrc3RD 0,0,0,0x80000,0x40
Dump of memory at 0x00000000_00080000 + 64(0x40) bytes:
                           00       04       08       0C
0x00000000_00080000: 12341234 DEADBEEF CAFEFACE 12341234 4.4.........4.4.
0x00000000_00080010: 12341234 12341234 12341234 12341234 4.4.4.4.4.4.4.4.
0x00000000_00080020: 00000000 00000000 00000000 00000000 ................
0x00000000_00080030: 00000000 00000000 00000000 00000000 ................
value = 0 = 0x0
```

Remarks

One most models, memory access window 0 permits the CPU to generate AXI/OCP reads and writes in the target FPGA. The addres space for a memory access window is separate from the address space of a DMA engine. This means that in general, writing data using the **DUMP** utility and attempting to read it back via the DMADUMP utility will not return the same data. However, it is possible to create an FPGA design which explicitly makes a shared resource available in both the address space of a memory access window and the address space of a DMA engine. In that case, data written by one window/DMA channel can be read back via another window/DMA channel.

Return values

When **DUMP** successfully executes the requested function, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_ALLOCATION_FAILURE	3	A memory allocation failed.
EXIT_DEVICE_OPEN_ERROR	4	Failed to open ADMXRC3 device.
EXIT_MAPWINDOW_ERROR	5	Failed to map window into process' virtual address space.

Table 5 : Return values for DUMP utility

3.5 FLASH utility

VxWorks kernel shell entry points

```
int admxrc3FlashHelp           ()
int admxrc3FlashInfo           (indexOrSerial, flags, targetIndex)
int admxrc3FlashChkblankRegion(indexOrSerial, flags, targetIndex, regionIndex)
int admxrc3FlashChkblank       (indexOrSerial, flags, targetIndex[, regionStart,
                                regionLength])
int admxrc3FlashEraseRegion    (indexOrSerial, flags, targetIndex, regionIndex)
int admxrc3FlashErase          (indexOrSerial, flags, targetIndex[, regionStart,
                                regionLength])
int admxrc3FlashProgramRegion (indexOrSerial, flags, targetIndex, pBitFilename,
                                regionIndex)
int admxrc3FlashProgram        (indexOrSerial, flags, targetIndex, pBitFilename[,
                                regionStart, regionLength])
int admxrc3FlashVerifyRegion   (indexOrSerial, flags, targetIndex, pBitFilename,
                                regionIndex)
int admxrc3FlashVerify         (indexOrSerial, flags, targetIndex, pBitFilename[,
                                regionStart, regionLength])
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int targetIndex	Index of the target FPGA whose configuration Flash memory is to be operated upon.
unsigned int regionStart	The starting byte address of a user-specified region within the Flash memory.
unsigned int regionLength	The length
unsigned int regionIndex	The index of a predefined region within the Flash memory.
const char* pBitFilename	The filename of a bitstream (**.bit**) file to be used for a program or verify operation.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x10	**FLAG_FORCE**	Causes the FLASH utility to ignore a mismatch between the FPGA identification string embedded in the **.bit** file specified by **pBitFilename** and the actual FPGA fitted to the reconfigurable computing card
0x100	**FLAG_FAILSAFE**	Causes the FLASH utility to target the "failsafe" region of Flash when using the **admxrc3FlashChkblank**

Summary

Manipulates the Flash memory which holds configuration bitstreams for a particular target FPGA in a reconfigurable computing device.

Description

The **FLASH** utility has five operations:

- **Blank check** using the **admxrc3FlashChkblank** or **admxrc3FlashChkblankRegion** entry point.

  Verifies that a region is blank, i.e. all bytes are 0xFF.
- **Erase** using the **admxrc3FlashErase** or **admxrc3FlashEraseRegion** entry point.

  Erases a region so that it becomes blank, i.e. all bytes are 0xFF.
- **Information** using the **admxrc3FlashInfo** entry point.

  Displays information about the Flash memory that holds a region.
- **Program** using the **admxrc3FlashProgram** or **admxrc3FlashProgramRegion** entry point.

  Programs the specified bitstream (**.bit**) file into a region so that the target FPGA is configured from a particular region at power-on or reset.
- **Verify** using the **admxrc3FlashVerify** or **admxrc3FlashVerifyRegion** entry point.

  Verifies that a region contains the specified bitstream (**.bit**) file.

Informational entry points

The **admxrc3FlashInfo** entry point displays information about the bank of configuration Flash memory for a given target FPGA:

- **admxrc3FlashInfo** *indexOrSerial*, *flags*, *targetIndex*

The first two parameters are as described above, and the third parameter, *targetIndex*, specifies which target FPGA's Flash memory is the subject of the blank check. This parameter is required because each target FPGA, if there are more than one, may have a separate bank of Flash memory.

Output is of the following form:

```
Target FPGA is device 7vx690tffg1761
Flash type is Numonyx Axcell P30 (Symm bl), 65536(0x10000) kiB
Useable region is 0x1200000-0x3FFFFFF
```

Blank check entry points

There are two entry points which perform a blank check:

- **admxrc3FlashChkblank** *indexOrSerial*, *flags*, *targetIndex*, *regionStart*, *regionLength*
- **admxrc3FlashChkblankRegion** *indexOrSerial*, *flags*, *targetIndex*, *regionIndex*

For both entry points, the first two parameters are as described above, and the third parameter, *targetIndex*, specifies which target FPGA's Flash memory is the subject of the blank check. This parameter is required because each target FPGA, if there are more than one, may have a separate bank of Flash memory.

In most use cases, **admxrc3FlashChkblank** is recommended and *regionStart* & *regionLength* are omitted (so that they are given values of 0). When *regionLength* is zero, the FLASH utility determines the correct region of Flash to blank-check using model-specific information. If *flags* omits **FLAG_FAILSAFE** (0x100), then the "default" region of Flash is blank-checked. This is the region from which the target FPGA is normally configured. If *flags* includes **FLAG_FAILSAFE** (0x100), then the "failsafe" region of Flash is blank-checked. Not all models have a "failsafe" region; please refer to the User Manual for your reconfigurable computing hardware in order to determine if it has a failsafe region.

Specifiying *regionStart* and *regionLength* allows the user to explicitly specify an address range to blank-check, which overrides the normal behaviour of FLASH in which it determines range of addresses to blank-check based on model-specific information. This is useful when storing multiple compressed bitstreams within a single region, to be used in conjunction with IPROG reconfiguration.

FLAG_FAILSAFE	regionLength	Behavior
omitted from *flags*	omitted or zero	Blank checks default region
included in *flags*	omitted or zero	Blank checks failsafe region
N/A	nonzero	Blank checks user-specified address range

Table 6 : Summary of admxrc3FlashChkblank entry point behavior

The second entry point, **admxrc3FlashChkblankRegion** requires the index of a predefined region (whose meaning model-specific) to be passed as the fourth parameter, *regionIndex*. Here, the Flash utility converts *regionIndex* into an address range using model-specific information. This is useful for certain models which have 4 or more predefined regions, of which the "failsafe" and "default" regions are only two. The **FLAG_FAILSAFE** (0x100) flag has no effect.

Example 1 - Using the first device in the system (index 0), the following calls all blank check the default region of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashChkblank
admxrc3FlashChkblank 0,0,0
admxrc3FlashChkblank 0,0,0,0,0
```

Example 2 - Using the device with serial number 100, blank check the failsafe region of the configuration Flash memory for target FPGA 1:

```
admxrc3FlashChkblank 100,0x101,1
```

Example 3 - Using the second device in the system (index 1), blank check byte address range 0x300000 - 0x39FFFFF inclusive of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashChkblank 1,0,0,0x3000000,0xA00000
```

Example 4 - Using the device with serial number 100, blank-check region 3 of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashChkblankRegion 100,1,0,3
```

Erase entry points

There are two entry points which perform an erase operation:

- **admxrc3FlashErase** *indexOrSerial*, *flags*, *targetIndex*, *regionStart*, *regionLength*
- **admxrc3FlashEraseRegion** *indexOrSerial*, *flags*, *targetIndex*, *regionIndex*

For both entry points, the first two parameters are as described above, and the third parameter, *targetIndex*, specifies which target FPGA's Flash memory is the subject of the erase operation. This parameter is required because each target FPGA, if there are more than one, may have a separate bank of Flash memory.

In most use cases, **admxrc3FlashErase** is recommended and *regionStart* & *regionLength* are omitted (so that they are given values of 0). When *regionLength* is zero, the FLASH utility determines the correct region of Flash to erase using model-specific information. If *flags* omits **FLAG_FAILSAFE** (0x100), then the "default" region of Flash is erased. This is the region from which the target FPGA is normally configured. If *flags* includes **FLAG_FAILSAFE** (0x100), then the "failsafe" region of Flash is erased. Not all models have a "failsafe" region; please refer to the User Manual for your reconfigurable computing hardware in order to determine if it has a failsafe region.

Specifiying *regionStart* and *regionLength* allows the user to explicitly specify an address range to erase, which overrides the normal behaviour of FLASH in which it determines range of addresses to erased based on model-specific information. This is useful when storing multiple compressed bitstreams within a single region, to be used in conjunction with IPROG reconfiguration.

FLAG_FAILSAFE	regionLength	Behavior
omitted from *flags*	omitted or zero	Erases default region
included in *flags*	omitted or zero	Erases failsafe region
N/A	nonzero	Erases user-specified address range

Table 7 : Summary of admxrc3FlashErase entry point behavior

The second entry point, **admxrc3FlashEraseRegion** requires the index of a predefined region (whose meaning model-specific) to be passed as the fourth parameter, *regionIndex*. Here, the Flash utility converts *regionIndex* into an address range using model-specific information. This is useful for certain models which have 4 or more predefined regions, of which the "failsafe" and "default" regions are only two. The **FLAG_FAILSAFE** (0x100) flag has no effect.

Regardless of which entry point is used, the erase operation is performed as two distinct phases:

1. The address range in the selected bank of Flash memory, that has either been explicitly specified as parameters or have been automatically computed by the **FLASH** utility, is erased.
2. A blank check is performed on the address range that has just been erased, in order to confirm that the erase operation was successful.

Example 1 - Using the first device in the system (index 0), the following calls all erase the default region of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashErase
admxrc3FlashErase 0,0,0
admxrc3FlashErase 0,0,0,0,0
```

Example 2 - Using the device with serial number 100, erase the failsafe region of the configuration Flash memory for target FPGA 1:

```
admxrc3FlashErase 100,0x101,1
```

Example 3 - Using the second device in the system (index 1), erase byte address range 0x300000 - 0x39FFFF inclusive of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashErase 1,0,0,0x3000000,0xA00000
```

Example 4 - Using the device with serial number 100, erase region 3 of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashEraseRegion 100,1,0,3
```

Program entry points

There are two entry points which perform a program operation:

- **admxrc3FlashProgram** *indexOrSerial*, *flags*, *targetIndex*, *pBitFilename*, *regionStart*, *regionLength*
- **admxrc3FlashProgramRegion** *indexOrSerial*, *flags*, *targetIndex*, *pBitFilename*, *regionIndex*

For both entry points, the first two parameters are as described above, and the third parameter, *targetIndex*, specifies which target FPGA's Flash memory is the subject of the programming operation. This parameter is required because each target FPGA, if there are more than one, may have a separate bank of Flash memory.

The fourth parameter is the bitstream filename or path (**.bit** extension), which may be on a filesystem local to the VxWorks machine or on the VxWorks host. The **FLASH** utility reads this file into memory and writes the SelectMap data in it into the Flash.

In most use cases, **admxrc3FlashProgram** is recommended and *regionStart* & *regionLength* are omitted (so that they are given values of 0). When *regionLength* is zero, the FLASH utility determines the correct region of Flash to program using model-specific information. If *flags* omits **FLAG_FAILSAFE** (0x100), then the "default" region of Flash is programmed. This is the region from which the target FPGA is normally configured. If *flags*

includes **FLAG_FAILSAFE** (0x100), then the "failsafe" region of Flash is programmed. Not all models have a "failsafe" region; please refer to the User Manual for your reconfigurable computing hardware in order to determine if it has a failsafe region.

Specifiying *regionStart* and *regionLength* allows the user to explicitly specify an address range to program, which overrides the normal behaviour of FLASH in which it determines range of addresses to programmed based on model-specific information. This is useful when storing multiple compressed bitstreams within a single region, to be used in conjunction with IPROG reconfiguration.

FLAG_FAILSAFE	regionLength	Behavior
omitted from *flags*	omitted or zero	Program default region
included in *flags*	omitted or zero	Program failsafe region
N/A	nonzero	Programs user-specified address range

Table 8 : Summary of admxrc3FlashProgram entry point behavior

The second entry point, **admxrc3FlashProgramRegion** requires the index of a predefined region (whose meaning model-specific) to be passed as the fourth parameter, *regionIndex*. Here, the Flash utility converts *regionIndex* into an address range using model-specific information. This is useful for certain models which have 4 or more predefined regions, of which the "failsafe" and "default" regions are only two. The **FLAG_FAILSAFE** (0x100) flag has no effect.

Regardless of which entry point is used, the programming operation is performed as three distinct phases:

1. The address range in the selected bank of Flash memory, that has either been explicitly specified as parameters or have been automatically computed by the **FLASH** utility, is first erased.
2. The SelectMap data from the **.bit** file, specified by *pBitFilename*, is written into the Flash memory. This typically does not cover the entire address range that was erased in phase 1.
3. The data in the Flash memory that was written in phase 2 file is verified against the specified **.bit** file, to confirm that the programming operation was successful.

Example 1 - Using the first device in the system (index 0), the following calls both program the default region of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashProgram 0,0,0,"host:/path/to/my.bit"
admxrc3FlashProgram 0,0,0,"host:/path/to/my.bit",0,0
```

Example 2 - Using the device with serial number 100, program the failsafe region of the configuration Flash memory for target FPGA 1:

```
admxrc3FlashProgram 100,0x101,1,"host:/path/to/my.bit"
```

Example 3 - Using the second device in the system (index 1), program byte address range 0x300000 - 0x39FFFFF inclusive of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashProgram 1,0,0,"host:/path/to/my.bit",0x3000000,0xA00000
```

Example 4 - Using the device with serial number 100, program region 3 of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashProgramRegion 100,1,0,"host:/path/to/my.bit",3
```

Verify entry points

There are two entry points which perform a verify operation:

- **admxrc3FlashVerify** *indexOrSerial*, *flags*, *targetIndex*, *pBitFilename*, *regionStart*, *regionLength*
- **admxrc3FlashVerifyRegion** *indexOrSerial*, *flags*, *targetIndex*, *pBitFilename*, *regionIndex*

For both entry points, the first two parameters are as described above, and the third parameter, *targetIndex*, specifies which target FPGA's Flash memory is the subject of the verify operation. This parameter is required

because each target FPGA, if there are more than one, may have a separate bank of Flash memory.

The fourth parameter is the bitstream filename or path (**.bit** extension), which may be on a filesystem local to the VxWorks machine or on the VxWorks host. The **FLASH** utility reads this file into memory and uses it to verify the SelectMap data in the Flash.

In most use cases, **admxrc3FlashVerify** is recommended and *regionStart* & *regionLength* are omitted (so that they are given values of 0). When *regionLength* is zero, the FLASH utility determines the correct region of Flash to verify using model-specific information. If *flags* omits **FLAG_FAILSAFE** (0x100), then the "default" region of Flash is verified. This is the region from which the target FPGA is normally configured. If *flags* includes **FLAG_FAILSAFE** (0x100), then the "failsafe" region of Flash is verified. Not all models have a "failsafe" region; please refer to the User Manual for your reconfigurable computing hardware in order to determine if it has a failsafe region.

Specifiying *regionStart* and *regionLength* allows the user to explicitly specify an address range to verify, which overrides the normal behaviour of FLASH in which it determines range of addresses to verified based on model-specific information. This is useful when storing multiple compressed bitstreams within a single region, to be used in conjunction with IPROG reconfiguration.

FLAG_FAILSAFE	regionLength	Behavior
omitted from *flags*	omitted or zero	Verify default region
included in *flags*	omitted or zero	Verify failsafe region
N/A	nonzero	Verify user-specified address range

Table 9 : Summary of admxrc3FlashVerify entry point behavior

The second entry point, **admxrc3FlashVerifyRegion** requires the index of a predefined region (whose meaning model-specific) to be passed as the fourth parameter, *regionIndex*. Here, the Flash utility converts *regionIndex* into an address range using model-specific information. This is useful for certain models which have 4 or more predefined regions, of which the "failsafe" and "default" regions are only two. The **FLAG_FAILSAFE** (0x100) flag has no effect.

Example 1 - Using the first device in the system (index 0), the following calls both verify the default region of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashVerify 0,0,0,"host:/path/to/my.bit"
admxrc3FlashVerify 0,0,0,"host:/path/to/my.bit",0,0
```

Example 2 - Using the device with serial number 100, verify the failsafe region of the configuration Flash memory for target FPGA 1:

```
admxrc3FlashVerify 100,0x101,1,"host:/path/to/my.bit"
```

Example 3 - Using the second device in the system (index 1), verify byte address range 0x300000 - 0x39FFFFF inclusive of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashVerify 1,0,0,"host:/path/to/my.bit",0x3000000,0xA00000
```

Example 4 - Using the device with serial number 100, verify region 3 of the configuration Flash memory for target FPGA 0:

```
admxrc3FlashVerifyRegion 100,1,0,"host:/path/to/my.bit",3
```

Return values

When **FLASH** successfully executes the requested function, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_BAD_RANGE	2	Explicitly-specified address range is not valid.
EXIT_INSUFFICIENT_ARGS	4	Too few positional arguments following command.
EXIT_ILLEGAL_FLAGS	5	An illegal combination of flags was passed.
EXIT_INVALID_RANGE	6	A Flash address range was invalid.
EXIT_ALLOCATION_FAILURE	7	A memory allocation failed.
EXIT_BLOCK_QUERY_FAILURE	8	Failed to query a Flash block (ADMXRC3_GetFlashBlockInfo failed).
EXIT_CREATE_FILE_ERROR	9	Failed to open file for writing readback data.
EXIT_WRITE_FILE_ERROR	10	Failed to write to readback data file.
EXIT_READ_FLASH_ERROR	11	Failed to read from Flash (ADMXRC3_ReadFlash failed).
EXIT_WRITE_FLASH_ERROR	12	Failed to write to Flash (ADMXRC3_WriteFlash failed).
EXIT_ERASE_FLASH_ERROR	13	Failed to erase Flash (ADMXRC3_EraseFlash failed).
EXIT_READ_BIT_ERROR	14	Failed to read .bit file.
EXIT_SANITY_ERROR	15	A sanity check failed.
EXIT_FPGA_MISMATCH	16	The FPGA identifier in the .bit file does not match the FPGA in the device.
EXIT_BOOT_FLAG_VERIFY_ERROR	17	Boot flag read back has incorrect value.
EXIT_DEVICE_OPEN_ERROR	18	Failed to open ADMXRC3 device.
EXIT_CARDINFO_ERROR	19	Failed to get information about ADMXRC3 device.
EXIT_FAILSAFE_NOT_SUPPORTED	20	Writing to failsafe region not supported for this ADMXRC3 device.
EXIT_ILLEGAL_TARGET_INDEX	21	Target FPGA index illegal for this ADMXRC3 device.
EXIT_UNSUPPORTED_MODEL	22	The ADMXRC3 device is an unsupported model.
EXIT_FPGAINFO_ERROR	23	Failed to get information about target FPGA.
EXIT_FLASHINFO_ERROR	24	Failed to get information about Flash memory.
EXIT_ILLEGAL_REGION	25	Region index passed is illegal for this ADMXRC3 device.
EXIT_SET_CLOCK_FAILED	26	Failed to set LCLK clock generator to nominal frequency.
EXIT_NOT_BLANK	27	Blank check failed; region is not blank.
EXIT_VERIFY_FAILED	28	Verification failed; data read back did not match data written.
EXIT_NULL_FILENAME	29	Illegal NULL string passed for a filename.

Table 10 : Return values for FLASH utility

3.5.1 Region to address range mapping

WARNING
The **FLAG_FAILSAFE** flag, and the *regionIndex*, *regionStart* & *regionLength* parameters must be used with care on models that feature a Virtex-6 target FPGA, because they can be used to overwrite the **failsafe region** of the Flash memory. The failsafe region is factory-programmed with a bitstream that protects again sub-micron effects that might otherwise degrade the performance of the target FPGA over time.

Xilinx answer record 35055 elaborates on protecting Virtex-6 GTX transceivers from performance degradation over time.

Alpha Data recommends that the failsafe region should not be erased or overwritten. If overwritten on a model that features a Virtex-6 target FPGA, the user must ensure that it is written with a valid, known-good bitstream that satisfies the requirements for protecting the target FPGA from sub-micron effects.

Most of Alpha Data's reconfigurable computing cards have Flash memory capable of storing multiple **.bit** files, and are divided into two or more regions. The address map for each Flash memory bank, including information about regions, is presented in the ADMXRC3 API Hardware Addendum.

The following guidelines are recommended when using the **FLASH** utility:

- For models that have a failsafe region, such as the ADM-XRC-6T1, use the **admxrc3FlashErase** and **admxrc3FlashProgram** entry points where possible. Do not pass **FLAG_FAILSAFE** in *flags* unless the intention is definitely to erase or program the failsafe region with a known-good **.bit** file. Omit or use values of 0 for the *regionStart* and *regionLength* parameters, unless the intention is to program an address range that is not one of the predefined regions (i.e. it corresponds to neither the default nor failsafe region).
- For models that have a failsafe region, such as the ADM-XRC-6T1, when using the **admxrc3FlashEraseRegion** and **admxrc3FlashProgramRegion** entry points, take care to verify that the *regionIndex* parameter is correct. *regionIndex* must not correspond to the failsafe region unless the intention is definitely to overwrite the failsafe bitstream.
- For models that do not have a failsafe region, such as the ADM-XRC-KU1, use any of the available entry points for the **FLASH** utility. For such models, the **FLASH** utility rejects the **FLAG_FAILSAFE** flag with an error.

Some examples of safe usage of the **FLASH** utility follows:

- The following are all equivalent on the ADM-XRC-6T1, and perform a blank-check on the failsafe region (1), which should fail assuming that the factory-programmed failsafe bitstream is still present:

```
admxrc3FlashChkblank 0,0x100,0
admxrc3FlashChkblank 0,0x100,0,0x2900000,0x1700000
admxrc3FlashChkblankRegion 0,0x100,0,1
```

  The following are all equivalent on the ADM-XRC-6T1, and write a .bit file into the default region (0), which is safe to overwrite:

```
admxrc3FlashProgram 0,0x100,0,"/path/to_my_design.bit",
admxrc3FlashProgram 0,0x100,0,0x1200000,0x1700000
admxrc3FlashProgramRegion 0,0x100,0,"/path/to_my_design.bit",0
```

- The following are all equivalent on the ADM-PCIE-7V3, and write a bitstream into the default region (1):

```
admxrc3FlashProgram 0,0,0,"/path/to/my_design.bit"
admxrc3FlashProgramRegion 0,0,0,"/path/to/my_design.bit",1
admxrc3FlashProgram 0,0,0,"/path/to/my_design.bit",0x2000000,0x2000000
```

- On the ADM-PCIE-7V3, the following programs a small compressed bitstream (must be less than or equal to 10 MiB) in the uppermost 10 MiB of region 3:

```
admxrc3FlashProgram 0,0,0,"/path/to/small.bit",0x7600000,0xA00000
```

3.6 INFO utility

VxWorks kernel shell entry points

```
int admxrc3InfoHelp()
int admxrc3Info(indexOrSerial[, flags])
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x10	**FLAG_SHOWFLASHINFO**	The **INFO** utility displays Flash memory bank information.
0x20	**FLAG_SHOWMODULEINFO**	The **INFO** utility displays I/O module information.
0x40	**FLAG_SHOWSENSORINFO**	The **INFO** utility displays sensor information.

Summary

Displays information about a reconfigurable computing device.

Description

The **admxrc3Info** entry point demonstrates the use of most of the informational functions in the ADMXRC3 API. The output consists of several sections, the first of which is obtained using ADMXRC3_GetVersionInfo:

```
API information
API library version           1.4.17
Driver version                1.4.17
```

The second section shows information obtained using ADMXRC3_GetCardInfoEx, and shows the information in the ADMXRC3_CARD_INFOEX structure:

```
Card information
Model                         ADM-VPX3-7V2
Serial number                 200(0xC8)
Number of programmable clocks 2
Number of DMA channels        4
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             23
Number of I/O module sites    1
Number of memory banks        4
Bank presence bitmap          0xF
```

The third section uses the **NumTargetFpga** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFpgaInfo to enumerate the target FPGAs in the device:

```
Target FPGA information
FPGA 0                        7VX690TFFG1761-2I
```

The fourth section uses the **NumMemoryBank** member of the ADMXRC3_CARD_INFOEX structure and

ADMXRC3_GetBankInfo to enumerate the memory banks (non-Flash) in the device:

```
Memory bank information
Bank 0                          SDRAM, DDR3, 262144(0x40000) kiW x 32 + 0 bits
                                303.0 MHz - 800.0 MHz
                                Connectivity mask 0x1
Bank 1                          SDRAM, DDR3, 262144(0x40000) kiW x 32 + 0 bits
                                303.0 MHz - 800.0 MHz
                                Connectivity mask 0x1
... (other memory banks) ...
```

The fourth section uses the **NumWindow** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetWindowInfo to enumerate the memory access windows in the device:

```
Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xC0400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0x64EC00000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
... (other windows) ...
```

The next section appears if *flags* contains **FLAG_SHOWFLASHINFO**. It uses the **NumFlashBank** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetFlashInfo to enumerate the Flash memory banks in the device:

```
Flash bank information
Bank 0                          Numonyx Axcell P30 (Symm bl), 65536(0x10000) kiB
                                Useable area 0x1200000-0x3FFFFFF
```

The next section appears if *flags* contains **FLAG_SHOWMODULEINFO**. It uses the **NumModuleSite** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetModuleInfo to enumerate the I/O module sites in the device and show what is fitted, if anything:

```
I/O module information
Module 0                        Product FMC-CLINK-MINI
                     Part number FMC-CLINK-MINI
                    Manufacturer Alpha Data
                   Serial number 112
                Manufacture time 8995680 minutes since 00:00 1/1/1996
                     I/O voltage 1.80
                           Flags 0x0
```

The next section appears if *flags* contains **FLAG_SHOWSENSORINFO**. It uses the **NumSensor** member of the ADMXRC3_CARD_INFOEX structure and ADMXRC3_GetSensorInfo to enumerate the sensors in the device:

```
Sensor information
Sensor 0                        12V VPX power rail
                                V, double, exponent 0, error 0.000
Sensor 1                        5V VPX power rail
                                V, double, exponent 0, error 0.000
... (other sensors) ...
```

Return values

When **INFO** runs successfully, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_DEVICE_OPEN_ERROR	2	Failed to open ADMXRC3 device.

Table 11 : Return values for INFO utility

3.7 LOADER utility

VxWorks kernel shell entry points

```
int admxrc3LoaderHelp()
int admxrc3Loader(indexOrSerial, flags, targetIndex, pBitFilename)
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int targetIndex	Index of the target FPGA to be configured.
const char* pBitFilename	Filename or path of bitstream (**.bit**) file to be used to configure the selected target FPGA.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x10	**FLAG_BINARY**	Treats *pBitFilename* as binary SelectMap data, not a .bit file.
0x20	**FLAG_NOLINKCHECK**	Does not check the status of the host link after configuring the FPGA.
0x40	**FLAG_IGNOREMISMATCH**	Ignores any mismatch between the FPGA device string embedded in the **.bit** file and the FPGA in the reconfigurable computing device.

Summary

Configures a target FPGA with a **.bit** file, and then exits.

Description

The **LOADER** utility configures the target FPGA, identified by *targetIndex*, within a reconfigurable computing device with the bitstream file identified by *pBitFilename*, and then exits.

By default, **LOADER** expects *pBitFilename* to name a **.bit** file, i.e. a file generated by the Xilinx ISE or Vivado design tools, and attempts to parse the file accordingly. However, if *flags* includes **FLAG_BINARY**, **LOADER** treats *pBitFilename* as a file containing raw SelectMap data. Such a file can be obtained in a number of ways, including a user-created program or by using the BITSTRIP utility.

Because the **LOADER** utility uses ADMXRC3_ConfigureFromFile or ADMXRC3_ConfigureFromBuffer, it normally checks that communications with the target FPGA have been established before exiting. If *flags* includes **FLAG_NOLINKCHECK**, **LOADER** omits this check. **FLAG_NOLINKCHECK** is appropriate for an FPGA design that has no MPTL host interface, for example stand-alone Ethernet design.

Return values

When **LOADER** runs successfully, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_DEVICE_OPEN_ERROR	2	Failed to open ADMXRC3 device.
EXIT_STAT_FILE_ERROR	3	Failed to stat() .bit file.
EXIT_FILE_TOO_LARGE	4	.bit file too large to load into memory.
EXIT_ALLOCATION_FAILED	5	Failed to allocate buffer for firmware or board config data.
EXIT_FILE_OPEN_ERROR	6	Failed to open .bit file for reading.
EXIT_FILE_READ_ERROR	7	Failed to read .bit file.
EXIT_CONFIGURATION_FAILED	8	ADMXRC3_Configure{FromBuffer,FromFile} failed.

Table 12 : Return values for LOADER utility

3.8 MONITOR utility

VxWorks kernel shell entry points

```
int admxrc3MonitorHelp()
int admxrc3Monitor(indexOrSerial[, flags[, numRepetition[, periodSec]]])
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int numRepetition	Number of times to repeat reading the sensors (0 means repeat for ever).
unsigned int periodSec	Interval between each set of sensor readings

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.

Summary

Displays readings from all sensors.

Description

The **MONITOR** utility repeatedly displays sensor readings in the command shell at the interval specified by the *periodSec* parameter. The number of repetitions to perform before terminating can be specified on the command line using the *numRepetition* parameter, and if zero or omitted, the program runs for ever.

MONITOR makes use of the ADMXRC3_GetSensorInfo and ADMXRC3_ReadSensor functions from the ADMXRC3 API, and can run alongside other reconfigurable computing applications without disturbing them.

The output looks like this:

```
Model:                                  257 (0x101) => ADM-XRC-6TL
Serial number:                          101 (0x65)
Number of sensors:                      10
  Sensor  0               1V supply rail: 0.987000 V
  Sensor  1             1.5V supply rail: 1.509186 V
  Sensor  2             1.8V supply rail: 1.803192 V
  Sensor  3             2.5V supply rail: 2.508896 V
  Sensor  4             3.3V supply rail: 3.268082 V
  Sensor  5               5V supply rail: 5.017990 V
  Sensor  6      XMC variable power rail: 12.000000 V
  Sensor  7              XRM I/O voltage: 2.495712 V
  Sensor  8    LM87 internal temperature: 49.000000 deg C
  Sensor  9      Target FPGA temperature: 57.000000 deg C
```

Return values

When **MONITOR** runs successfully, the return value is 0. When an error occurs, one of the following values is

returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_DEVICE_OPEN_ERROR	2	Failed to open ADMXRC3 device.
EXIT_CARDINFO_ERROR	3	Failed to get information about ADMXRC3 device.
EXIT_NO_SENSORS_FOUND	4	No sensors in the selected device.

Table 13 : Return values for MONITOR utility

3.9 VPD utility

VxWorks kernel shell entry points

```
int admxrc3VpdHelp()
int admxrc3VpdFB(indexOrSerial, flags, offset, count, val8)
int admxrc3VpdFW(indexOrSerial, flags, offset, count, val16)
int admxrc3VpdFD(indexOrSerial, flags, offset, count, val32)
int admxrc3VpdFQ(indexOrSerial, flags, offset, count, val64(ULL))
int admxrc3VpdFS(indexOrSerial, flags, offset, count, pString)
int admxrc3VpdRB(indexOrSerial, flags, offset, count)
int admxrc3VpdRW(indexOrSerial, flags, offset, count)
int admxrc3VpdRD(indexOrSerial, flags, offset, count)
int admxrc3VpdRQ(indexOrSerial, flags, offset, count)
int admxrc3VpdWB(indexOrSerial, flags, offset, count, ...)
int admxrc3VpdWW(indexOrSerial, flags, offset, count, ...)
int admxrc3VpdWD(indexOrSerial, flags, offset, count, ...)
int admxrc3VpdWQ(indexOrSerial, flags, offset, count, ...(ULL))
int admxrc3VpdWS(indexOrSerial, flags, offset, count, pString)
```

where the parameters are:

unsigned int indexOrSerial	Specifies the index (or serial number) of the device to open.
unsigned int flags	Specifies the bitwise OR of zero or more flags that modify the behavior of the utility; see below for details.
unsigned int address	The starting address within VPD Space to be used for the data transfer.
unsigned int count	The number of bytes of data to transfer.
unsigned int val8	Fill value which is interpreted as being 8 bits wide.
unsigned int val16	Fill value which is interpreted as being 16 bits wide.
unsigned int val32	Fill value which is interpreted as being 32 bits wide.
unsigned long long val64	Fill value which is interpreted as being 64 bits wide.
...	A variable number of **unsigned int** values, which are interpreted as 8-bit / 16-bit / 32-bit write values depending on which **admxrc3VpdW*** entry point is used.
...(ULL)	A variable number of **unsigned long long** write values, which are interpreted as 64-bit write values.
const char* pString	A NUL-terminated ASCII string (8-bit characters) to be used as a write value or fill value.

The **flags** parameter is the bitwise-OR of zero or more of the following values:

Value	Symbolic name	Meaning
0x1	**FLAG_BYSERIAL**	**indexOrSerial** is treated as a serial number rather than a device index.
0x20	**FLAG_BIGENDIAN**	Big-endian byte order is used, as opposed to little-endian.

Summary

Displays data read from Vital Product Data (VPD) memory, or writes data to VPD memory. VPD memory contains information about a reconfigurable computing card, such as the type of FPGA fitted, memory bank sizes etc.

Description

Avoiding VPD corruption
The **VPD** utility must be used with care, particularly when using its capability to write or fill VPD memory. Corrupting the VPD of a reconfigurable computing card can impair its functionality until the VPD is restored to its correct values.

To avoid corrupting VPD, please ensure that you are aware of the address map of VPD Space for the particular reconfigurable computing card in use. This information is provided by ADMXRC3 API Hardware Addendum.

Writing to VPD writes requires a software enable to be activated in the ADB3 Driver. Additionally, on certain models in Alpha Data's range of reconfigurable computing hardware, a card must be put into **Service Mode** before VPD memory can be accessed. For further details, refer to Section 3.9.1 below.

The **VPD** utility operates in one of three modes:

- Filling a region of VPD memory with a value or string; for this mode, use the **admxrc3VpdFB**, **admxrc3VpdFW**, **admxrc3VpdFD**, **admxrc3VpdFQ** or **admxrc3VpdFS** entry points.
- Reading data from VPD memory and displaying it; for this mode, use the **admxrc3VpdRB**, **admxrc3VpdRW**, **admxrc3VpdRD** or **admxrc3VpdRQ** entry points.
- Writing numeric or string data to a region of VPD memory; for this mode, use the **admxrc3VpdWB**, **admxrc3VpdWW**, **admxrc3VpdWD**, **admxrc3VpdWQ** or **admxrc3VpdWS** entry points.

The address space accessed by the VPD utility is called **VPD Space**, and the address map of VPD Space is model-specific. Details of address maps of VPD Space for supported models are given in ADMXRC3 API Hardware Addendum.

The entry point **admxrc3VpdHelp** displays a brief help message, listing the available entry points and arguments.

If *flags* contains **FLAG_BIGENDIAN**, the **VPD** utility reads or writes numeric values in big-endian byte ordering convention as opposed to little-endian (the default).

Read commands

The read command implies the word width used for displaying the data:

- **admxrc3VpdRB** *indexOrSerial*, *flags*, *address*, *count*

  Byte (8-bit) reads; data is displayed as bytes.
- **admxrc3VpdRW** *indexOrSerial*, *flags*, *address*, *count*

  Word (16-bit) reads; data is displayed as words.
- **admxrc3VpdRD** *indexOrSerial*, *flags*, *address*, *count*

  Doubleword (32-bit) reads; data is displayed as doublewords.
- **admxrc3VpdRQ** *indexOrSerial*, *flags*, *address*, *count*

  Quadword (64-bit) reads; data is displayed as quadwords.

In all cases, the first four arguments passed must be:

(a) *indexOrSerial* - identifies the reconfigurable computing device to be used. The first reconfigurable computing device in the system, in the kernel's enumeration order, has index 0.

(b) *flags* - the bitwise-OR of zero or more flags that modify the behavior of the **VPD** utility. See flag definitions above.

(c) *address* - the byte address within VPD memory at which to begin reading.

(d) *count* - byte count; the number of bytes of VPD memory to read and display.

Some usage examples follow.

Example 1 - Using the first card in the system (index 0), dump 256 bytes of VPD memory at address 0x1FF00:

```
admxrc3VpdRB 0,0,0x1ff00,0x100
```

Example 2 - Using the card with serial number 100, dump 32 doublewords of VPD memory at address 0x0:

```
admxrc3VpdRD 100,1,0x0,0x80
```

Write commands

The write command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the word width of the data. The available write commands are:

- **admxrc3VpdWB** *indexOrSerial*, *flags*, *address*, *count*, *values...*

  Write values are supplied as **unsigned int** values, and written as bytes (8-bit).

- **admxrc3VpdWW** *indexOrSerial*, *flags*, *address*, *count*, *values...*

  Write values are supplied as **unsigned int** values, and written as words (16-bit).

- **admxrc3VpdWD** *indexOrSerial*, *flags*, *address*, *count*, *values...*

  Write values are supplied as **unsigned int** values, and written as doublewords (32-bit).

- **admxrc3VpdWQ** *indexOrSerial*, *flags*, *address*, *count*, *values...*

  Write values are supplied as **unsigned long long values**, and written as quadwords (64-bit).

- **admxrc3VpdWS** *indexOrSerial*, *flags*, *address*, *count*, *pString*

  Data is supplied as an ASCII string (8-bit characters).

In all cases, the first four arguments passed must be:

(a) *indexOrSerial* - identifies the reconfigurable computing device to be used. The first reconfigurable computing device in the system, in the kernel's enumeration order, has index 0.

(b) *flags* - the bitwise-OR of zero or more flags that modify the behavior of the **VPD** utility. See flag definitions above.

(c) *address* - the byte address within VPD memory at which to begin writing.

(d) *count* - byte count; the number of bytes to write to VPD memory.

In the case of **admxrc3VpdWB**, **admxrc3VpdWW** & **admxrc3VpdWD**, the fifth and subsequent arguments are values to write to VPD memory. These are expressed as **unsigned int** values, but are cast to the appropriate word size (**uint8_t**, **uint16_t** or **uint32_t**) when written to VPD memory. Sufficient write values must be passed in order to satisfy the byte count, *count*. For example, if the **admxrc3VpdWW** entry point is used and *count* is 8, four write values values must be passed (as the 5th to 8th parameters).

In the case of **admxrc3VpdWQ**, the fifth and subsequent arguments are values to write to VPD memory. These are expressed as **unsigned long long values**, and therefore require a cast in the VxWorks shell (see usage example 2 below). The values are cast to **uint64_t** when written to VPD memory. Sufficient write values must be passed in order to satisfy the byte count, *count*. For example, if *count* is 24, three write values values must be passed as the 5th to 7th parameters.

In the case of **admxrc3VpdWS**, the fifth argument is a NUL-terminated string to copy to VPD memory. If this string (including the NUL-terminator) is not long enough to satisfy the byte count specified by *count*, the remaining portion of the region of VPD memory determined by*address* and *count* is not modified. NOTE: The copy of the string written to VPD memory is NUL-terminated if shorter than *count* bytes, but if longer, it is truncated and not NUL-terminated.

Some usage examples follow.

Example 1 - Using the first card in the system (index 0), write 4 words 0x0123, 0x4567, 0x89AB & 0xCDEF at VPD address 0x1FF80:

```
admxrc3VpdWW 0,0,0x1ff80,8,0x0123,0x4567,0x89ab,0xcdef
```

Example 2 - Using the card with serial number 100, write 2 quadwords 0xDEADBEEFCAFEFACE & 0x0123456789ABCDEF at VPD address 0x100000:

```
admxrc3VpdWQ 100,1,0x100000,16,(long long)0xdeadbeefcafeface,(long long)0x01234567
89abcdef
```

Example 3 - Using second card in the system (index 1), write the NUL-terminated string "Hello World!" at VPD address 0x100011:

```
admxrc3VpdWS 1,0,0x100011,13,"Hello World!"
```

Fill entry points

When filling a region of VPD memory with data, the entry point used determines whether the fill value is numeric or a string. In the case of numeric data, the entry point used also implies the word width of the data. The available fill entry points are:

- **admxrc3VpdFB** *indexOrSerial*, *flags*, *address*, *count*, *val8*

  Fill value is supplied as an **unsigned int** value, and written as a byte (8-bit).
- **admxrc3VpdFW** *indexOrSerial*, *flags*, *address*, *count*, *val16*

  Fill value is supplied as an **unsigned int** value, and written as a word (16-bit).
- **admxrc3VpdFD** *indexOrSerial*, *flags*, *address*, *count*, *val32*

  Fill value is supplied as an **unsigned int** value, and written as a doubleword (32-bit).
- **admxrc3VpdFQ** *indexOrSerial*, *flags*, *address*, *count*, *val64(ULL)*

  Fill value is supplied as an **unsigned long long** value, and written as a quadword (64-bit).
- **admxrc3VpdFS** *indexOrSerial*, *flags*, *address*, *count*, *pString*

  Fill value is a NUL-terminated ASCII string (8-bit characters).

In all cases, the first four arguments passed must be:

(a) *indexOrSerial* - identifies the reconfigurable computing device to be used. The first reconfigurable computing device in the system, in the kernel's enumeration order, has index 0.

(b) *flags* - the bitwise-OR of zero or more flags that modify the behavior of the **VPD** utility. See flag definitions above.

(c) *address* - the byte address within VPD memory at which to begin filling.

(d) *count* - byte count; the number of bytes of VPD memory to fill.

The fifth parameter is the fill value, which is either numeric or a string, depending on which entry point is used.

If the entry point is **admxrc3VpdFS** and *pString* is a string shorter than *count* characters, the string is repeated until the byte count is satisfied. If the string is longer than *count*, only the first *count* characters are used. If a string contains spaces, it must be quoted on the command line so that it is not interpreted by the shell as two or more separate arguments.

For the numeric fill commands **admxrc3VpdFB**, **admxrc3VpdFW**, **admxrc3VpdFD** and **admxrc3VpdFQ**, the numeric value is repeated until the byte count is satisfied. Note that passing an **unsigned long long** parameter in the VxWorks shell for the **admxrc3VpdFQ** entry point requires a cast; see usage example 2 below.

Some usage examples follow.

Example 1 - Using the first card in the system (index 0), fill VPD addresss 0x1FF80 to 0x1FFFF (0x80 bytes) with the byte value 0xFF (for many types of nonvolatile memory, 0xFF is the value which is returned after erasing the memory):

```
admxrc3VpdFB 0,0,0x1ff80,0x80,0xFF
```

Example 2 - Using the card with serial number 100, fill VPD addresses 0x100080 to 0x1000BF (64 bytes) with the quadword value 0x0123456789ABCDEF:

```
admxrc3VpdFQ 100,1,0x100080,64,(long long)0x0123456789abcdef
```

Example 3 - Using second card in the system (index 1), fill VPD addresses 0x1FF80 to 0x1FFFF with copies of the NUL-terminated string "Hello World!". The final copy of the string will not be NUL-terminated because 0x80 is not an integer multiple of the string length, including NUL (13).

```
admxrc3VpdFS 1,0,0x1FF80,0x80,"Hello World!"
```

Example session

The following session was captured in VxWorks using an ADM-XRC-6T1. The base address 0x100000 is used because that is the VPD-space address of the user-definable area of VPD memory in the ADM-XRC-6T1.

```
-> admxrc3VpdRB 0,0,0x100000,0x60
Dump of VPD at 0x00100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x00100000: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100020: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100030: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100040: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100050: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
value = 0 = 0x0
-> admxrc3VpdFS 0,0,0x100008,20,"hello world!"
value = 0 = 0x0
-> admxrc3VpdWD 0,0,0x100020,12,0xdeadbeef,0xcafeface,0x12345678
value = 0 = 0x0
-> admxrc3VpdFW 0,0,0x100031,10,0xa55a
value = 0 = 0x0
-> admxrc3VpdRB 0,0,0x100000,0x60
Dump of VPD at 0x00100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x00100000: FF FF FF FF FF FF FF FF 68 65 6C 6C 6F 20 77 6F ........hello wo
0x00100010: 72 6C 64 21 00 68 65 6C 6C 6F 20 77 FF FF FF FF rld!.hello w....
0x00100020: EF BE AD DE CE FA FE CA 78 56 34 12 FF FF FF FF ........xV4.....
0x00100030: FF 5A A5 5A A5 5A A5 5A A5 5A A5 FF FF FF FF FF .Z.Z.Z.Z.Z......
0x00100040: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0x00100050: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
value = 0 = 0x0
```

Return values

When **VPD** successfully executes the requested function, the return value is 0. When an error occurs, one of the following values is returned:

Symbolic name	Value	Meaning
EXIT_OK	0	Success.
EXIT_ALLOCATION_FAILURE	3	A memory allocation failed.
EXIT_DEVICE_OPEN_ERROR	4	Failed to open ADMXRC3 device.
EXIT_READVPD_ERROR	5	A call to ADMXRC3_ReadVPD failed.

Table 14 : Return values for VPD utility (continued on next page)

Symbolic name	Value	Meaning
EXIT_WRITEVPD_ERROR	6	A call to ADMXRC3_WriteVPD failed.

Table 14 : Return values for VPD utility

3.9.1 VPD write-protection mechanisms

In order to be able to write to VPD memory, a software protection mechanism must be disabled:

- In VxWorks, an integer value **adb3DrvEnableVpdWrite** must be set to 1, which can be done using the VxWorks kernel shell. For example:

  ```
  -> adb3DrvEnableVpdWrite=1
  ```

  A change to this value takes immediate effect, and it is **not** necessary to reboot the machine after changing it.

  The **adb3DrvEnableVpdWrite** value is embedded within the ABD3 Driver binary. If the ADB3 Driver is built as a VxWorks downloadable kernel module, it is therefore set to 0 when the ADB3 Driver is downloaded to the VxWorks target. If the ADB3 Driver is built into the VxWorks kernel image, its initial value on boot is 0.

As well as the software protection mechanism, some models in Alpha Data's range of reconfigurable computing hardware must be in **Service Mode** in order for any access to VPD memory (whether reading or writing) to succeed. This applies to **all** models that have a switch setting for **Service Mode**. Please contact the User Manual for your reconfigurable computing hardware in order to determine whether or not this applies.

Appendix A: AVR2UTIL clock generator indices

A.1 ADM-XRC-KU1

In the ADM-XRC-KU1, the frequencies of clock generators with indices 1 and 3 may be overridden using the **setclknv** command, whereas the clock generators with indices 0 and 2 may not (because their frequencies must be fixed in order for the board to function correctly).

clockgen-index	Net(s) [1]	Purpose	Factory default (MHz)	ADMXRC3 API index [2]	Note
0	REFCLK250M_N0 REFCLK250M_N1	MPTL reference clock	250	N/A	[3]
1	PROGCLK_N0 PROGCLK_N1 PROGCLK_N2	Reference clock for user-definable MGTs	156.25	1	
2	REFCLK300M_N0 REFCLK300M_N1 REFCLK300M_N2 REFCLK300M_N3	Reference clock for DDR4 SDRAM and other logic	300	N/A	[3]
3	FABRIC_CLK_N	General purpose clock	300	2	

Table 15 : AVR2UTIL clock generator indices (ADM-XRC-KU1)

Note:

[1] For differential clocks, only the negative side of a differential pair is listed.

[2] This is the clock generator index used in calls such as **ADMXRC3_SetClockFrequency**.

[3] Not user-programmable. Attempting to set an override frequency using **AVR2UTIL** will fail with exit code 103. Not exposed by ADMXRC3 API.

A.2 ADM-PCIE-8V3

In the ADM-PCIE-8V3, the frequencies of clock generators with indices 0, 1 and 2 may be overridden using the **setclknv** command, whereas the clock generator with index 3 may not (because its frequency must be fixed in order for the board to function correctly).

clockgen-index	Net(s) [1]	Purpose	Factory default (MHz)	ADMXRC3 API index [2]	Note
0	GTY_CLK_0B_N GTY_CLK_0C_N	QSFP+ 0 reference clock QSFP+ 1 reference clock	161.1328125	0	
1	GTY_CLK_1B_N GTY_CLK_1C_N	FireFly 0 reference clock FireFly 1 reference clock	161.1328125	1	
2	MEM_CLK_0_N MEM_CLK_1_N	Reference clock for DDR4 SDRAM	300	2	
3	FABRIC_CLK_N	General purpose clock	300	N/A	[3]

Table 16 : AVR2UTIL clock generator indices (ADM-PCIE-8V3)

Note:

[1] For differential clocks, only the negative side of a differential pair is listed.

[2] This is the clock generator index used in calls such as **ADMXRC3_SetClockFrequency**.

[3] Not user-programmable. Attempting to set an override frequency using **AVR2UTIL** will fail with exit code 103. Not exposed by ADMXRC3 API.

A.3 ADM-PCIE-8K5

In the ADM-PCIE-8K5, the frequencies of all four clock generators, with indices 0 to 3, may be overridden using the **setclknv** command.

clockgen-index	Net(s) [1]	Purpose	Factory default (MHz)	ADMXRC3 API index [2]	Note
0	GTY_CLK_0_N	SFP+ 0 reference clock	156.25	0	
1	GTY_CLK_1_N	SFP+ 1 reference clock	156.25	1	
2	MEM_CLK_0_N MEM_CLK_1_N	Reference clock for DDR4 SDRAM	300	2	
3	GTH_CLK_2_N	FireFly 1 reference clock	156.25	3	

Table 17 : AVR2UTIL clock generator indices (ADM-PCIE-8K5)

Note:

[1] For differential clocks, only the negative side of a differential pair is listed.

[2] This is the clock generator index used in calls such as **ADMXRC3_SetClockFrequency**.

Revision History

Date	Revision	Nature of change
30 Aug 2016	1.0	Initial version.
3 Mar 2017	1.1	Documented new commands in AVR2UTIL 2.5.0: i2c-read-to-file, i2c-verify-from-file, i2c-write-from-file, i2c-read, i2c-write, save-brdcfg, save-firmware, save-vpd, display-vpd, display-vpd-raw, display-sensors, display-sensors-raw, override-sensor, release-sensor
12 Jun 2017	1.2	Documented new commands in AVR2UTIL 2.7.1: getclk, setclk, spi-info, spi-raw

Address: Suite L4A, 160 Dundee Street, Edinburgh, EH11 1DQ, UK
Telephone: +44 131 558 2600
Fax: +44 131 558 2700
email: sales@alpha-data.com
website: http://www.alpha-data.com

Address: 10822 West Toller Drive, Suite 250 Littleton, CO 80127
Telephone: (303) 954 8768
Fax: (866) 820 9956 - toll free
email: sales@alpha-data.com
website: http://www.alpha-data.com