IEEE Draft Std P1275.3/D8
Standard for Boot (Initialization Configuration) Firmware
Supplement for IEEE 1014-1987 (VME) Bus

December 15, 1994

DRAFT EDITION for SPONSOR BALLOT

Sponsored by the IEEE Bus Architecture Standards Committee
Prepared by the P1275 Working Group

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331 USA

Foreword by the Chairman of the Working Group

(This foreword is not a part of IEEE Draft Std P1275.3, *Standard for Boot (Initialization Configuration) Firmware, Supplement for IEEE 1014-1987 (VME) Bus*

Introduction

Firmware is the ROM-based software that controls a computer between the time it is turned on and the time the primary operating system takes control of the machine. Firmware's responsibilities include testing and initializing the hardware, determining the hardware configuration, loading (or booting) the operating system, and providing interactive debugging facilities in case of faulty hardware or software.

Open Firmware is the firmware architecture defined by IEEE Std 1275-1994, *Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices.* That standard is bus-independent, instruction-set-independent, and system-independent.

The core requirements and practices specified by IEEE Std 1275-1994 must be supplemented by system-specific requirements to form a complete specification for the firmware for a particular system. This standard establishes such additional requirements pertaining to the bus architecture defined by *ANSI/IEEE 1014-1987, VME Standard*.

Working Group Members

The following individuals were members of the Working Group at the time this document was produced:

William M. (Mitch) Bradley, Chairman, FirmWorks
David M. Kahn, Vice Chairman, Sun Microsystems, Inc.
David L. Paktor, Draft Editor, FORCE COMPUTERS Inc.
Luan D. Nguyen, Secretary, IBM Corporation
Mike Tuciarone, FirmWorks
Martin Walsh, Sun Microsystems, Inc.
Ron Hochsprung, Apple Computer, Inc.
Michael Saari, Sun Microsystems, Inc.
John Rible
Yongjae Rim, IBM Corporation
Ilan Rabinowitz

Contributors

The following individuals have contributed to this document:

Will Crowder, SHL Systemhouse, Inc.
Mike Williams, SHL Systemhouse, Inc.
Paul Thomas, Sun Microsystems, Inc.
Shawn Morrissey

Sponsor Balloting Body

The following individuals were members of the Sponsor Balloting Body:

This page intentionally left blank.

This page intentionally left blank.

1. Overview

This standard describes the application of IEEE Std 1275-1994, *Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices*, to the computer expansion bus defined by *ANSI/IEEE 1014-1987, VME Standard*. This standard addresses both VME bus controllers and devices intended to plug into a VME bus. This standard will address only systems conforming to the 32-bit *IEEE* standard named above. The newer VME64 bus standard, *VITA Std 1-1994*, which specifies a maximum address space of 64 bits, is outside the scope of this standard. All references to `VME` or to `VMEbus` within this standard refer specifically to the *IEEE-1014* specification.

2. References

[1] IEEE Std 1275-1994, *Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices*

[2] *ANSI/IEEE Std 1014-1987*, *IEEE Standard for a Versatile Backplane Bus: VMEbus*

3. Terms

This standard uses technical terms as they are defined in the documents cited in clause 2, plus the following terms:

3.1 bus controller. A hardware device that implements a VME bus.

3.2 bus node. In the device tree, a descendant node that represents a VME bus controller. Note that an Open Firmware device node is not in itself a physical hardware device; rather, it is a software abstraction that describes a hardware device as the interface, or "bridge," between a VME bus and its parent (which could be another bus). In cases where a node represents a bridge between one VME bus and another, the node is both a bus node relative to the bus it controls, and a child node of its parent bus.

3.3 child device. A hardware device that connects to or "plugs in" to a VME bus. Some such hardware devices may have multiple logically-independent parts, or functions, while many devices have only one function per device.

3.4 child node. In the device tree, a descendant node that represents a hardware device plugged into a VME bus. Devices that have multiple functions may represent the overall device as a single child of the bus node, and the separate functions as node-children of the device's node.

3.5 Core, core specification. Synonyms for IEEE Std 1275-1994, *Standard for Boot (Initialization Configuration) Firmware, Core Requirements and Practices*, i.e., the standard that specifies the system-independent and bus-independent requirements for Open Firmware.

3.6 Open Firmware. The firmware architecture defined by IEEE Std 1275-1994 and its applicable supplements or, when used as an adjective, a software component compliant with such an architecture.

4. Bus Characteristics

VMEbus defines an address domain of 16, 24 and 32 bit segments and a data domain of 16 and 32 bit segments. Board addresses are independent of their physical placement within the backplane; usually, board addresses are assigned by means of some physical configuration device, such as DIP or rotary-hex switches.

The VMEbus standard specifies the physical layout of 16 or 32 data lines, and 32 address lines, plus additional addressing lines, including six *Address Modifier* (AM) lines and a line called *Lword**.

VMEbus data transactions take place within a number of distinct address spaces represented by a subset of the 64 possible values encoded by the six Address Modifier lines. The two data-widths are encoded by the *Lword** line.

The six Address Modifier lines represent a combination of one of three address widths (32-bit, 24-bit or 16-bit) with additional information, such as whether transactions are configured to come singly or in bursts, or the type of access, (supervisor vs. user, and program vs. data). The *Lword** line represents 32-bit data-width transactions when active, or 16-bit data-width transactions when inactive.

Additional lines provide support for byte accesses, transparently to the software. Consult Reference [2] for details.

Not all combinations of address width, access type, and data width are valid. A discussion of the valid combinations is beyond the scope of this standard. Consult Reference [2] for details.

VMEbus defines seven prioritized Interrupt Request lines, numbered 1 through 7.

VMEbus uses vectored interrupts. When a device requests an interrupt, the bus-controller initiates an Interrupt-Acknowledge cycle, during which the interrupting device writes, and the bus-controller reads, a Status/ID number, or "vector", that identifies the device. This information is then used to efficiently call the appropriate interrupt handler.

VMEbus defines three additional lines that may signal an interrupt: Bus-Error, SysFail and an optional AC Fail.

The VMEbus standard allows backplane configurations to be built with either 16 or 32 data lines.

The VMEbus standard allows backplane configurations built with 16 data lines to perform 32-bit processor accesses by splitting the transactions into two 16-bit data transfers. However, if hardware support for splitting a 32-bit transfer into two 16-bit transfers is not provided on a backplane built with 16 data lines, the VMEbus standard does not specify the behavior of the bus controller when a 32-bit data transfer is attempted. Under these circumstances, it is the common and recommended (but not universal) practice to have the bus controller signal a Bus-Error.

The VMEbus standard provides for an Arbitration subsystem, which schedules requests from multiple bus masters, transfers control of the bus among them in an orderly manner, and guarantees that only one controls the bus at any given time. The Arbitration Bus consists of six bussed lines called Bus Busy, Bus Clear, and Bus Request 0 through 3, along with four daisy-chained lines called Bus Grant 0 through 3. Three types of arbitration scheduling algorithms are described, named: Prioritized (PRI), Round-Robin-Select (RRS), and Single-level (SGL), and permission is given to use other scheduling algorithms. For devices that are "plugged in" to the VMEbus (i.e., child devices), three types of bus requesting algorithms are described, called: Release-When-Done (RWD), Release-On-Request (ROR), and a FAIR requester. Consult Section 3 of Reference [2] for details.

4.1. Physical Address Formats and Representations

4.1.1. Numerical Representation

The numerical representation of a VMEbus physical address consists of two cells denoted "phys.hi" and "phys.lo". Additionally, one cell is used to encode the size field of a child's **`reg`** format property.

"Phys.hi" denotes the VMEbus logical address space; it represents the six VMEbus Address Modifier lines and the *Lword** line. AM0 is mapped onto the least significant bit, AM1 onto the next higher significant bit, and so on, and *Lword** is mapped onto the bit whose binary weight is 64. A '1' value in a "phys.lo" bit denotes the active state of the corresponding VMEbus line.

"Phys.lo" corresponds to the VMEbus address lines.

Note -- Since the numeric representation of a VMEbus physical address consists of 2 address cells, and the number of cells used to encode the size field of a child's **`reg`** property is 1, explicit implementations of the standard properties **`#address-cells`** and **`#size-cells`** need not be present.

4.1.2. Text Representation

The text string representation of a VMEbus physical address is: *AML,VME-address* . *AML* corresponds to the "phys.hi" cell of the numerical representation and *VME-address* corresponds to the "phys.lo" cell.

AML may take either of two forms: (1) the ASCII hexadecimal representation of "phys.hi ", or (2) a text string matching the name of one of the constants listed in section **5.2.2.1**.

For display, *AML* is the ASCII hexadecimal representation of "phys.hi".

VME-address is the ASCII hexadecimal representation of "phys.lo".

Conversion of hexadecimal numbers from text representation to numeric representation shall be case-insensitive, and leading zeroes shall be permitted but not required. Conversion from numeric representation to text representation shall use the lower-case forms of the hexadecimal digits in the range {a..f}, suppressing leading zeroes.

4.2. Bus-specific Configuration Variables

This standard does not define the means by which the VMEbus characteristics, (e.g., the type of bus arbitration, or the selection of interrupt levels to be enabled), are configured.

4.2.1. Format of a Probe List

VMEbus does not specify any sort of geographical or slot oriented addressing; some mechanism must be provided on a processor board or host adapter to allow the user to specify a probing order for VMEbus devices.

The order of probing with respect to other busses that might be connected to the device is determined by the processor board's main bus probe list, and any bus adapter probe lists for busses that are hierarchically higher in the device tree than the VMEbus.

This standard does not define the means by which the order of probing of the FCode drivers of devices plugged-in to the VMEbus is specified. The recommended method is to initiate the ordered probing of the FCode drivers by means of the *script*.

4.3. Interrupt-Specification Format

For the standard **`interrupts`** property the seven VMEbus Interrupt Request levels, numbered 1 through 7, are represented by their corresponding numbers, 1 through 7. The vector (Status/ID number) that a child-device uses to identify itself during the Interrupt-Acknowledge cycle is likewise represented by its corresponding number.

4.4. FCode Interpretation Semantics

The VMEbus Standard makes no provision for FCode. Therefore, manufacturers of host systems that supply access to the VMEbus, or to which a VMEbus host adapter can be attached, must provide some mechanism to allow a system integrator to specify the location of FCode drivers for VMEbus host adapters and devices.

The recommended practice for the location of an FCode driver for a VMEbus plug-in device is to supply the driver on a ROM physically incorporated into that device. Where that is impractical, a device manufacturer may supply the FCode driver in some other form such that a system integrator can incorporate the driver into the host system.

The **`probe-vme`** user interface command, described in section **7.3**, can be used in the *script* to interpret FCode programs for VME devices.

5. Bus Nodes

Systems that supply access to the VMEbus shall define a device-node for the VMEbus as specified below:

5.1. Properties

5.1.1. Core-specified Properties for Bus Nodes

The following Open Firmware standard properties have special interpretations for VMEbus.

"**`device_type`**" S

Standard *property-name* to specify the implemented interface.

A Standard Package conforming to this specification and corresponding to a device that implements a VMEbus controller shall implement this property with the string value "**vme**".

"**`reg`**" S

Standard *prop-name* to define the package's registers.

"**`ranges`**" S

Standard *prop-name* to specify child-to-parent address mapping.

The format of child physical addresses is as specified in section **4.1.1** of this supplement.

Note -- A VMEbus controller device that is "plugged in" to another bus might be required, by the bindings of the parent-bus, to publish an "**`interrupts`**" property, encoded in the form specified by the bindings of the parent-bus. Similarly, a VMEbus controller that is built-in to the same hardware with its CPU and is capable of generating direct CPU-level interrupts, might publish an "**`interrupts`**" property, encoded in the form used for specifying CPU-level interrupts.

5.1.2. Bus-specific Properties for Bus Nodes

"**`interrupt-translations`**" S

Bus-specific *property-name*, specifies the mapping of VMEbus interrupt levels to the parent's interrupts.

prop-encoded-array: One through seven ordered sets, of the form (*VMEbus_level,Parent_Specifier*). The *VMEbus_level* is an integer in the range {1 .. 7}, representing a VMEbus Interrupt Request level, encoded as with **`encode-int`** ; the *Parent_Specifier* is a bus-specific interrupt-specifier, encoded as specified for the parent bus, that represents the interrupt passed by the VMEbus controller to its parent bus when the VMEbus Interrupt Level corresponding to *VMEbus_level* is detected. If a given VMEbus interrupt level is not translated to the parent bus, then the entry for that VMEbus interrupt level shall not be present.

A Standard Package corresponding to a VMEbus controller that has a fixed translation between VMEbus Interrupt Levels and the interrupts of its parent shall implement this property, otherwise this property shall not be present.

"**`berr-interrupt`**" S

Bus-specific *property-name*, specifies the interrupt passed to the parent when the VMEbus BERR signal is detected.

prop-encoded-array: One bus-specific interrupt-specifier, encoded as specified for the parent bus, representing the interrupt passed by the VMEbus controller to its parent bus when a VMEbus Bus-Error signal is detected.

A Standard Package corresponding to a VMEbus controller that has a fixed translation between the VMEbus BERR signal and the interrupts of its parent shall implement this property, otherwise this property shall not be present.

"**`sysfail-interrupt`**" S

Bus-specific *property-name*, specifies the interrupt passed to the parent when the VMEbus SysFail signal is detected.

prop-encoded-array: One bus-specific interrupt-specifier, encoded as specified for the parent bus, representing the interrupt passed by the VMEbus controller to its parent bus when a VMEbus SysFail signal is detected.

A Standard Package corresponding to a VMEbus controller that has a fixed translation between the VMEbus SysFail signal and the interrupts of its parent shall implement this property, otherwise this property shall not be present.

"**`acfail-interrupt`**" S

Bus-specific *property-name*, specifies the interrupt passed to the parent when the VMEbus AC Fail signal is detected.

prop-encoded-array: One bus-specific interrupt-specifier, encoded as specified for the parent bus, representing the interrupt passed by the VMEbus controller to its parent bus when a VMEbus AC Fail signal is detected.

A Standard Package corresponding to a VMEbus controller that has a fixed translation between the VMEbus AC Fail signal and the interrupts of its parent shall implement this property, otherwise this property shall not be present.

Note -- For a VMEbus controller that is built-in to the same hardware with its CPU and is capable of generating direct CPU-level interrupts, the interrupt-specifiers may be encoded in the form used for specifying CPU-level interrupts.

5.2. Methods

5.2.1. Core-specified Methods for Bus Nodes

A Standard Package implementing the "**`vme`**" device type shall implement the following Open Firmware standard methods with physical address representations as specified in section **4.1.1** of this standard.

`open`	(-- okay?)	Prepare this device for subsequent use.
`close`	(--)	Close this previously-**`open`**ed device.
`map-in`	(phys.lo phys.hi size -- virt)	Map the specified region, return a virtual address.
`map-out`	(virt size --)	Destroy mapping from previous **`map-in`**.
`dma-alloc`	(size -- virt)	Allocate a memory region for later use.
`dma-free`	(virt size --)	Free memory allocated with **`dma-alloc`**.
`dma-map-in`	(virt size cacheable? -- devaddr)	Convert virtual address to VME bus DMA address.
`dma-map-out`	(virt devaddr size --)	Free DMA mapping set up with **`dma-map-in`**.
`dma-sync`	(virt devaddr size --)	Synchronize (flush) DMA memory caches.
`probe-self`	(arg-str arg-len reg-str reg-len fcode-str fcode-len --)	Interpret FCode, as a child of this node.
`decode-unit`	(reg-str reg-len -- phys.lo phys.hi)	Convert text unit-string to physical address.
`encode-unit`	(phys.lo phys.hi -- reg-str reg-len)	Convert physical address to text unit-string.

5.2.1.1. Text String Input to the *decode-unit* method

The text string representation of a VMEbus physical address as described in section **4.1.2** is used as the input to the **`decode-unit`** method, and, by extension, as the *reg-string* and *fcode-string* inputs to the **`probe-self`** method or to the **`probe-vme`** User Interface Command (see section **7.3**), and as the *reg-string* input to **`add-vme-device`** or to **`begin-package`** when applied to the VMEbus node. The **`decode-unit`** method, et al, shall interpret the *AML* portion of the string as described in section **4.1.2**. Permission is hereby explicitly granted to include additional implementation-dependent strings that represent addresses outside the VMEbus space, which may also be interpreted as the *fcode-string* input to the **`probe-self`** method or the **`probe-vme`** command. This is a change from what the Core specifies.

5.2.2. Bus-specific Methods for Bus Nodes

A Standard Package implementing the "**vme**" device type shall implement the following bus-specific methods:

5.2.2.1. *AML* Constants

In order to make the numeric representations of the six VMEbus *AML* values in their single-transaction form accessible to child-nodes, the following six constants shall be defined as methods within the bus-node:

`vmea32d32` (-- phys.hi) M
Returns VMEbus phys.hi to access 32-bit address space with 32 bit Supervisory data transfers.

`vmea24d32` (-- phys.hi) M
Returns VMEbus phys.hi to access 24-bit address space with 32 bit Supervisory data transfers.

`vmea16d32` (-- phys.hi) M
Returns VMEbus phys.hi to access 16-bit address space with 32 bit Supervisory data transfers.

`vmea32d16` (-- phys.hi) M
Returns VMEbus phys.hi to access 32-bit address space with 16 bit Supervisory data transfers.

`vmea24d16` (-- phys.hi) M
Returns VMEbus phys.hi to access 24-bit address space with 16 bit Supervisory data transfers.

`vmea16d16` (-- phys.hi) M
Returns VMEbus phys.hi to access 16-bit address space with 16 bit Supervisory data transfers.

5.2.2.2. *AML* Modifiers

The following words shall be defined as methods within the bus-node; they convert the numeric representation of any of the VMEbus *AML* values from one form to another, as specified in the individual descriptions:

`burst` (phys.hi-single -- phys.hi-burst) M
Convert the numeric representation of any VMEbus *AML* value in single-transaction form to its burst-transaction form.

`vme-user` (phys.hi-supvr -- phys.hi-user) M
Convert the numeric representation of any VMEbus *AML* value in Supervisory form to its non-privileged (User-mode) form.

`vme-program` (phys.hi-pgm -- phys.hi-data) M
Convert the numeric representation of any VMEbus *AML* value in data-transfer form to its program-transfer form.

The above *AML* Modifier methods shall yield correct results when used in any combination and in any order that is meaningful, in terms of the VMEbus Address Modifiers as defined in Reference [2], and when supplied a correct input parameter. For example, the combinations: "**`burst vme-user`**" and "**`vme-user vme-program`**" shall yield a correct result when applied to one of the six *AML* Constants; the combination "**`burst vme-program`**", since it is not meaningful, will yield undefined results.

Note--The above Space Modifier methods are expected to be used by a device driver to modify the value to which **`my-space`** was set at the time **`begin-package`** was invoked. Typically, **`my-space`** would be set to one of the six named *AML* Constants, which supply Supervisory-mode single-transaction data-transfer forms of the VMEbus address spaces. Therefore, there is no need to define methods for the inverse transformations.

5.2.2.3. *AML*-Specific DMA-mapping Methods

`dma-space-map-in` (phys.hi virt size cacheable? -- devaddr) M

Convert the given virtual address to a VMEbus DMA address suitable for the given *phys.hi*.

`dma-space-map-out` (phys.hi virt devaddr size --) M

Free DMA mapping set up with **`dma-space-map-in`** using the same *phys.hi* as that given.

Note--Although the core specification allows additional parameters for **`dma-map-in`** and **`dma-map-out`** which could be used for the *phys.hi* parameters in the above words, the above methods are introduced here as a concession to existing FCode programs whose drivers may make use of the Core-defined words without additional parameters

5.2.2.4. Core-Specified DMA-mapping Methods

`dma-map-in` (virt size cacheable? -- devaddr) M

Convert the given virtual address to a VMEbus DMA address suitable for the *phys.hi* given by **`my-space`** . The semantics of this method are equivalent to: **`my-space`** *virt size cacheable?* **`dma-space-map-in`**

`dma-map-out` (virt devaddr size --) M

Free DMA mapping set up with **`dma-map-in`** using the *phys.hi* given by **`my-space`** . The semantics of this method are equivalent to: **`my-space`** *virt devaddr size* **`dma-space-map-out`**

Note--The preceding methods are included here as a concession to existing FCode programs. Their use in new FCode programs is discouraged. New VMEbus host adapters and new host systems that supply access to the VMEbus need to implement these methods in order to accommodate older devices whose drivers may make use of them.

5.2.2.5. *AML*-Specific DMA-memory Allocation Methods

`dma-space-alloc` (phys.hi size -- virt) M

Allocate a region of memory suitable for later use for DMA-mapping at the given *phys.hi*.

`dma-space-free` (phys.hi virt size --) M

Free memory allocated by **`dma-space-alloc`** with the given *phys.hi*.

Note--Although the core specification allows additional parameters for **`dma-alloc`** and **`dma-free`** which could be used for the *phys.hi* parameters in the above words, the above methods are introduced here as a concession to existing FCode programs whose drivers may make use of the Core-defined words without additional parameters

5.2.2.6. Core-Specified DMA-memory Allocation Methods

`dma-alloc` (size -- virt) M

Allocate a region of memory suitable for later use for DMA-mapping at the *phys.hi* given by **`my-space`** . The semantics of this method are equivalent to: **`my-space`** *size* **`dma-space-alloc`**

`dma-free` (virt size --) M

Free memory allocated by **`dma-space-alloc`** with the *phys.hi* given by **`my-space`** . The semantics of this method are equivalent to: **`my-space`** *virt size* **`dma-space-free`**

Note--The preceding methods are included here as a concession to existing FCode programs. Their use in new FCode programs is discouraged. New VMEbus host adapters and new host systems that supply access to the VMEbus need to implement these methods in order to accommodate older devices whose drivers may make use of them.

6. Child Nodes

6.1. Properties

6.1.1. Core-Specified Properties for Child Nodes

The following Open Firmware standard properties have special meanings or interpretations for VMEbus.

"**reg**" S

Standard *property-name*, defines device addresses.

6.1.2. Bus-specific Properties for Child Nodes

"**interrupts**" S

property-name, identifies the VMEbus interrupt levels and vectors (Status/ID numbers) this device can generate.

prop-encoded-array: One (*VMEbus_level,vector*) integer-pair, encoded as with **encode-int**, for each interrupt the device is capable of generating. The value of each *VMEbus_level* integer is in the range {1 .. 7}, representing a VMEbus Interrupt Request level; the corresponding *vector* integer represents the vector (Status/ID number) the device transmits during the Interrupt Acknowledge cycle following its assertion of the VMEbus Interrupt Request level. (For example, the pair 3,c0 would mean that the device generates VMEbus Interrupt Request level 3 and identifies itself with the vector c0 during the Interrupt Acknowledge cycle.)

A Standard Package corresponding to a VMEbus child device that is capable of generating VMEbus interrupts shall implement this property; if the device is not capable of generating VMEbus interrupts, this property shall be omitted.

"**requests**" S

property-name, identifies the type(s) of VMEbus arbitration requests this device can generate on each Bus Request line.

prop-encoded-array: An ordered set of four integers, one for each Bus Request line, each encoded as with **encode-int**. The first integer corresponds to Bus Request 0, the second integer corresponds to Bus Request 1, and so on. The value of each integer shall reflect the type of arbitration request(s), as described in Section 3.3.2 of Reference [2], that this device is capable of generating on the corresponding Bus Request line, using the following encoding: 0 if no requests are generated on this line, -1 if the device cannot determine the type of requests it generates on this line, or the sum of as many of the following as are applicable: 1 = FAIR, 2 = ROR, 4 = RWD.

A Standard Package corresponding to a VMEbus child device that is capable of generating arbitration requests shall implement this property; if the device is not capable of generating arbitration requests, this property shall not be present.

"**generates-sysfail**" S

property-name to denote that the device is capable of generating a SysFail signal to the VMEbus.

prop-encoded-array: <none> The information is conveyed by the presence or absence of this property.

A Standard Package corresponding to a VMEbus child device that is capable of generating the SysFail signal, or that cannot determine whether it is capable of generating the SysFail signal, to the VMEbus shall implement this property; if the device is not capable of generating SysFail, this property shall not be present.

"**bbsy-max**" S

property-name, identifies the maximum amount of time the device requires to generate the Bus-Busy signal.

prop-encoded-array: One integer, encoded as with **encode-int**, representing the maximum amount of time the device requires to generate the Bus-Busy signal, calibrated in nanoseconds.

"**bg-max**" S

property-name, identifies the maximum amount of time the device requires to propogate a Bus-Grant signal.

prop-encoded-array: One integer, encoded as with **encode-int**, representing the maximum amount of time the device requires to propogate a Bus-Grant-Out signal from a Bus-Grant-In signal, calibrated in nanoseconds. If the maxima differ among the four Bus-Grant lines, then the maximum of the four shall be used.

6.2. Methods

There are no special requirements on the methods for VMEbus Child Nodes beyond those specified in the Core.

7. User Interface Extensions

There are no special requirements for commands, beyond those specified in the Core, that are only applicable to VMEbus child devices.

7.1.Bus-specific Interpretations of Standard Commands

There are no requirements for commands specified in the Core to have special meaning when used with VMEbus child devices. Except as noted below, there are no requirements for commands specified in the Core to have special meaning when used with VMEbus controllers.

The requirements listed in this section, relating to the 32-bit register-access words **`rl@`** and **`rl!`** , are only applicable to those bus nodes associated with VMEbus controllers whose backplane configurations are built with 16 data lines and whose hardware does not provide support for splitting a 32-bit transfer into two 16-bit transfers. The essence of these requirements is that such support will be provided by the firmware of the bus node.

`rl@` (qaddr -- quad) F

Fetch a quadlet from the device register at *qaddr* in two 16-bit transfers, if the bus hardware does not support 32-bit transfers.

The device-node associated with a VMEbus controller whose hardware does not provide support for 32-bit transfers shall substitute (as with **`set-token`**) the standard FCode token for **`rl@`** with a bus-specific implementation that will read 32 bits of data by splitting the transaction into two 16-bit transfers. This is a change from what the Core specifies.

`rl!` (quad qaddr --) F

Store a quadlet to the device register at *qaddr* in two 16-bit transfers, if the bus hardware does not support 32-bit transfers.

The device-node associated with a VMEbus controller whose hardware does not provide support for 32-bit transfers shall substitute (as with **`set-token`**) the standard FCode token for **`rl!`** with a bus-specific implementation that will store 32 bits of data by splitting the transaction into two 16-bit transfers. This is a change from what the Core specifies.

The substitution of the FCode tokens for **`rl@`** and **`rl!`** shall take place immediately before the interpretation of the FCode of a child-device, as by **`probe-self`** and **`probe-vme`** . Upon completion of interpretation of the child-device's FCode, the FCode tokens for **`rl@`** and **`rl!`** shall be restored to their previous values.

Example implementation: The following example shows a possible implementation:

```
: my-rl@ ( qaddr -- quad )  dup 2+ rw@ swap rw@ wljoin ;
: my-rl! ( quad qaddr -- )  swap lwsplit -rot over 2+ rw! rw! ;
```

Bus nodes associated with VMEbus controllers that <u>do</u> provide hardware support for 32-bit transfers shall implement the register-access words as defined in the Core, not the special variants listed in this section.

7.2.Bus-specific FCodes

There are no special requirements for FCodes beyond those specified in the Core.

7.3.Bus-specific User Interface Commands

The following VMEbus-specific user interface commands are introduced to facilitate the probing of child-devices. Systems conforming to this specification, that supply access to a VMEbus device, shall implement these commands:

`probe-vme` (arg-str arg-len reg-str reg-len csr-offset fcode-str fcode-len --) M

Interpret FCode for a new VMEbus child-device.

The *reg-string* and *fcode-string* inputs follow the form of the text string representation of a VMEbus physical address as described in section **5.2.1.1**; the *arg-string* input follows the form of the *arg-string* input to the User Interface Command, **`begin-package`** , and *csr-offset* is an integer. A *csr-offset* value of -1 means the attempt to read the CSR is to be bypassed.

First, check to see that the device is plugged in by testing a Command-and-Status Register of the device at the location given by the sum of the *reg-string* and the *csr-offset*. Decode the *reg-string* as with the **`decode-unit`** method of the VMEbus node, resulting in a *phys.lo* and *phys.hi* . Add the value of the *csr-offset* to the *phys.lo* , resulting in the physical address of the CSR to test. Attempt to read one byte from that address, (perhaps by mapping the register, as with the **`map-in`** method of the VMEbus node, using **`cprobe`** to ensure the device is present, and then undoing any previous mapping, as with the **`map-out`** method of the VMEbus node), or (if the CSR is appropriately aligned) one doublet, or (if the CSR is appropriately aligned and if the *phys.hi* specifies a 32-bit data access) one quadlet.

Examine the result of the attempts to read the given CSR. If none of the attempts to read the CSR succeeded, then presume that the device is not plugged in, and return without performing any further steps.

If the value of the *csr-offset* was given as -1, then skip the preceding steps, presume that the device is plugged in, and proceed with the following steps.

Decode the *fcode-string* as described in section **5.2.1.1**. Check to ensure that there is a valid FCode program at the location indicated (perhaps by mapping the location and using **cpeek** to ensure that the location is present and that the first byte is a valid FCode byte). If so, then perform the function of **new-device** to create a new device-node under the VMEbus node. Perform the function of **set-args** , using the *arg-string* and *reg-string* parameters. Next, interpret the FCode program (perhaps by mapping in the *phys.lo* and *phys.hi* that resulted from decoding the *fcode-string* , with a range sufficiently large to contain the FCode located there, and then executing **byte-load**) at the given location, and then perform the function of **finish-device** . Undo any mapping in of the address range containing the FCode.

If a valid FCode program cannot be located at the indicated location, do not create the new device node, and do not attempt to interpret FCode.

Note -- see the note under **probe-self** in the Core document.

Example usage: In the following example, the device-specific FCode driver has been stored in a location 0x300 bytes past the start of a user programmable, non-volatile storage area, whose location is given by an *fcode-string* with an upper portion that represents addresses outside the VMEbus space, as permitted in section **5.2.1.1**:

```
probe-all
   0 0  " vmea32d32,57000"
   100                          \ The CSR to test is at offset 100 from the base.
   " flash,300"                 \ Physical address of the non-volatile storage area, plus 300
      probe-vme
install-console  banner
```

add-vme-device (name-str name-len reg-str reg-len size csr-offset --) M

Add a device-node for a new VMEbus child-device for which FCode is not available.

The *reg-string* input follows the form of the text string representation of a VMEbus physical address as described in section **4.1.2**; *size* and *csr-offset* are integers. A *csr-offset* value of -1 means the attempt to read the CSR is to be bypassed.

First, check to see that the device is plugged in by testing a Command-and-Status Register of the device at the location given by the sum of the *reg-string* and the *csr-offset*. Decode the *reg-string* as with the **decode-unit** method of the VMEbus node, resulting in a *phys.lo* and *phys.hi* . Add the value of the *csr-offset* to the *phys.lo* , resulting in the physical address of the CSR to test. Attempt to read one byte from that address, (perhaps by mapping the register, as with the **map-in** method of the VMEbus node, using **cprobe** to ensure the device is present, and then undoing any previous mapping, as with the **map-out** method of the VMEbus node), or (if the CSR is appropriately aligned) one doublet, or (if the CSR is appropriately aligned and if the *phys.hi* specifies a 32-bit data access) one quadlet.

Examine the result of the attempts to read the given CSR. If none of the attempts to read the CSR succeeded, then presume that the device is not plugged in, and return without performing any further steps.

If the value of the *csr-offset* was given as -1, then skip the preceding steps, presume that the device is plugged in, and proceed with the following steps.

Perform the function of **new-device** to create a new device-node under the VMEbus node. Use the *size* , along with the values of *phys.lo* and *phys.hi* that resulted from decoding the *reg-string*, to create a **reg** property for the new device-node. Perform the function of **set-args** , using the an *arg-string* of zero length, along with the *reg-string* parameter. Use the given *name-string* to create a name property for the new device-node, and then perform the function of **finish-device** .

Example usage: The following example shows not only how this word might be used in the *script* to add a new device-node, but also how additional properties might be created in that device-node:

```
probe-all
   " foo-dev" " vmea32d32,100000" 400
      100                       \ The CSR to test is at offset 100 from the base.
      add-vme-device            \ Create the device node for foo-dev , if it's plugged in.
                                \ Now, add the bbsy-max and bg-max properties:
   " /xyz-vme/foo-dev" ['] find-device catch 0= if
                                \ Skip all this if the device node wasn't created
```

```
        d# 500 encode-int " bbsy-max" property
                                        \ Max time to assert Bus-Busy is 500 nanoseconds

        d# 100 encode-int " bg-max" property
                                        \ Max time to propogate Bus-Grant is 100 nanoseconds

        device-end

    then

install-console  banner
```