Freescale Semiconductor
Engineering Bulletin

EB687
Rev. 0, 01/2008

Creating VLE Applications

For the MPC5500 Family

by: Bill Terry
32-Bit Automotive Applications

Table of Contents

1 Overview

Some processor cores used in the MPC5500 family of microprocessors incorporate the variable length encoding (VLE) extension of the Power Architecture instruction set. VLE allows greater code density with minimal or no loss of system performance by using a mix of 32-bit and 16-bit instructions. This application note provides an overview of VLE, and specific details for creating a new VLE application, or for porting an existing Power Architecture application to a VLE implementation. Certain members of the MPC5500 family are dual core processors that support VLE. While many of the principles in this document apply to dual core VLE operation, only a single core implementation is discussed.

1.1 Variable Length Encoding (VLE)

VLE allows Power Architecture Book E implementations to support a more efficient binary representation of an application. This can be particularly important for the embedded processor spaces where code density plays a major role in affecting overall system cost. VLE is a supplemental feature that may be applied to a portion of an application, or to the entire application. VLE code is typically generated by setting a compiler switch. Most compilers allow for conditional VLE compilation, selectable on a file by file or sub-project basis. See the documentation for the VLE compiler that you use for specific details.

See Table 1 for a list of the current MPC5500 devices that support VLE.

Table 1. MPC5500 VLE Support

Device	Core	VLE Support
MPC5514 MPC5516 MPC5517[1]	e200z0	Yes[2]
	e200z1	Yes
MPC5533	e200z3	Yes
MPC5534	e200z3	Yes
MPC5553	e200z6	No
MPC5554	e200z6	No
MPC5561	e200z6	Yes
MPC5565	e200z6	Yes
MPC5566	e200z6	Yes
MPC5567	e200z6	Yes

NOTES:
[1] MPC5514, MPC5516 and MPC5517 are dual core devices
[2] z0 core supports VLE only. It does not support the classic Power Architecture instruction set.

1.2 16-bit vs. 32-bit Instructions

The VLE extension uses the same semantics as traditional Book E, but due to the limited instruction encoding formats, VLE instructions typically support reduced immediate fields and displacements. VLE instructions are encoded in either a 16-bit or 32-bit format, and these may be intermixed. Note that some, but not all, Power Architecture instructions assemble in a VLE image, with identical syntax as used in a normal non-VLE Power Architecture application. The signal processing engine (SPE) instructions available as part of the classic Power Architecture are also available for use within VLE code.

VLE instructions are 16-bit aligned, however 16-bit VLE instructions may be freely intermixed with 32-bit VLE instructions without penalty. VLE and non-VLE code spaces are completely compatible and intercallable, with the restriction that VLE function or routine start addresses must be 32-bit aligned to be callable and linkable from non-VLE code space.

Complete information on the implementation of the VLE instruction set and programming Book E processors may be found in the following references available at www.freescale.com.

- Variable-Length Encoding (VLE) Extension Programming Interface Manual (VLEPIM)
- *Addendum to the e200z6 Power Architecture™ Core Reference Manual, Rev. 0 e200z6 with VLE*
- EREF: A Programmer's Reference Manual for Freescale Book E Processors
- *e200z6 Power Architecture™ Core Reference Manual*

- *e200z3 Power Architecture™ Core Reference Manual*

2 VLE Configuration Requirements

The VLE extension may be used globally within an application, or applied only to specific sections of the application. The e200z0 core is an exception, it supports only the VLE instruction set. Additionally, pre-compiled Power Architecture libraries or object files may be linked in with VLE applications as most compiler vendors provide VLE versions of the standard ANSI C libraries. The following sections explain how the application memory space must be planned, and the general device configuration procedure that must be followed when using VLE in an application. This application note assumes internal boot mode, with code executing from the flash memory. External and serial boot modes are not discussed.

2.1 Planning the Application

As discussed previously, the application image may be either classic Power Architecture, VLE, or a combination of the two. As in standard classic Power Architecture applications (where the core provides an MMU), the MMU is used to configure memory regions for both VLE and non-VLE code space via the TLB entries written through the MMU assist registers (MAS0-4 and MAS6). On most VLE enabled cores, VLE code space has the same configurable attributes as non-VLE code space, example: page size, physical and virtual page addresses, and cache attributes to mention a few. However, in VLE enabled cores, there is a new bit added to the MAS2 register that identifies the page defined by the MMU TLB entry as either VLE or non-VLE code space.

The necessary memory layout must be determined so that the appropriate VLE and non-VLE memory spaces are established by MMU configuration during initialization (if necessary). Modification to the linker map as defined by a linker descriptor file or in a makefile may also be required. This is discussed in later sections.

2.2 General Considerations

In summary, the main software design criteria that must be considered when creating a VLE application are:

- Is the application VLE only, or a combination of VLE and classic Power Architecture?
- Are there any existing libraries or object files to be linked in that are non-VLE code?
- If the application is mixed VLE/non-VLE, will boot code be VLE or classic Power Architecture?
- What functions or routines may require minimal execution time?[1]

The following sections detail the setup required for both VLE and mixed VLE non-VLE applications.

1. While compiling as VLE code may have a slight adverse affect on execution time in some cases, VLE code may actually execute in less time under certain conditions.

3 VLE Only Applications

In some cases, the entire application is compiled with VLE, and is linked with classic Power Architecture libraries also compiled with the VLE option. Because all code is VLE in this case, device initialization is straightforward. The following steps generalize the sequence:

- Configure the Reset Config Halfword (RCHW) to enable VLE mode.
- Depending on the application memory partitioning and layout, modify MMU if required.
- Execute all other initialization code, FMPLL setup, and any other low level tasks.
- Go to main()

3.1 RCHW and Boot Assist Module (BAM)

The RCHW for VLE enabled cores has a new VLE bit defined. The RCHW VLE bit definition is shown in Figure 1.

ADDRESS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0000_0000	Reserved					WTE	PS0	VLE	Boot ID							

Figure 1. Reset Configuration Half Word

At reset the BAM executes and searches for a valid RCHW at the lowest 32-bit address in certain defined memory spaces. For a full explanation of BAM operation see the MPC55xx Reference Manual for your device. If the BAM locates a valid RCHW as indicated by a Boot ID value of 0x5A, the WTE, PS0, and VLE bits are read to determine further conditional BAM program operation.

Under a normal non-VLE boot, the BAM configures the MMU to a default configuration that is suitable for use by many applications without modification. If the RCHW VLE bit is read as 1 by the BAM, the default MMU configuration is changed to support a VLE application. The default MMU configuration with and without the VLE bit set is illustrated in Table 2. TLB entries 1, 2, and 3 are defined as VLE code space when the RCHW[VLE] bit is set.

Table 2. MMU Settings by BAM at Reset

	RCHW[VLE] = 0					RCHW[VLE] = 1				
Region	**TLB**	**VADDR**	**PADDR**	**VLE**	**SIZE**	**TLB**	**VADDR**	**PADDR**	**VLE**	**SIZE**
Peripheral Bridge B and BAM	0	0xFFF0_0000	0xFFF0_0000	no	1MB	0	0xFFF0_0000	0xFFF0_0000	no	1MB
Internal flash	1	0x0000_0000	0x0000_0000	no	16MB	1	0x0000_0000	0x0000_0000	yes	16MB
EBI	2	0x2000_0000	0x0000_0000	no	16MB	2	0x2000_0000	0x0000_0000	yes	16MB
Internal SRAM	3	0x4000_0000	0x4000_0000	no	256K	3	0x4000_0000	0x4000_0000	yes	256K
Peripheral Bridge A	4	0xC3F0_0000	0xC3F0_0000	no	1MB	4	0xC3F0_0000	0xC3F0_0000	no	1MB

NOTES

Attempting to execute VLE code in memory that is configured by the MMU as a non-VLE page causes the core to generate an exception. Similarly, attempting to execute non-VLE code in memory that is configured by the MMU as a VLE page also causes the core to generate an exception. Peripheral memory space is never defined as VLE memory.

3.2 Source Code

There are no changes required to any C files when compiling an application for VLE. However, if the application includes any Power Architecture assembler code, that code will likely not assemble with the VLE option enabled. There are two possible solutions to this problem. Either the assembler files must be ported to use the VLE compatible instruction set, or a separate non-VLE memory region for location of this code must be created by modifying the MMU. The assembler files may then be assembled without the VLE option enabled and relocated to the non-VLE region. This mixed VLE and non-VLE application is completely described in Section 4, on page 6.

3.3 Linker and Alignment Requirements

Linking VLE code requires the definition of a new section in the linker command file or script. This new section name may vary depending on the compiler that is used. In most cases, the compiler has command line switches or GUI configurable compile options. Consult the documentation for your compiler and set the VLE compile option for all files in the application. This causes VLE code to be generated for all C source files, and generally causes the linker to link in VLE versions of any required ANSI C libraries.

Unless a change to the default MMU settings that are configured by the BAM is required, your linker file or script must define a section for the VLE code that starts at 0x0000_0000. In the case of the Green Hills MULTI tools, that section is called `.vletext` by default. Consult the documentation for your tools to determine what the VLE text section is called. Optionally, you can modify the source code using `#pragma` or `.org` compiler/assembler directives to force the linker to place it in a user defined section at 0x0000_0000.

A simple example linker file that uses the default MMU settings as configured by the BAM is shown below. In this example, the normal `.text` section has been replaced by `.vletext` for use with the Green Hills compiler. This example does not necessarily include all linker directives that may be necessary for actual application code.

```
MEMORY
{
     /* 2M Internal Flash */
     flash_rchw     : org = 0x00000000, LENGTH = 0x8
     int_flash      : org = 0x00000008, LENGTH = 1M - 0x8

     /* 64K Internal SRAM */
     int_sram       : org = 0x40000000, LENGTH = 64k
}
SECTIONS
{
     /* Flash data */
     .rchw          : { *(.rchw) } > flash_rchw
     .vletext       : {} > int_flash
     .rodata        : {} > int_flash
```

```
    .ctors          : {} > int_flash
    .dtors          : {} > int_flash
    .syscall        : > int_flash
    .fixaddr        : > int_flash
    .fixtype        : > int_flash

    /* SRAM data */
    .data           : {} > int_sram
    .sdata          : {} > int_sram
    .sbss           : {} > int_sram
    .sdata2         : {} > int_sram
    .sbss2          : {} > int_sram
    .heap           : {} > int_sram
    .bss            : {} > int_sram
}
```

In the case of a VLE only application, there are no special code alignments required, other than those that are always required for Book E processors, such as interrupt vectors.

4 Mixed VLE and Non-VLE Applications

In some applications it may be desirable to mix non-VLE compiled code with VLE compiled code. In this case, the default MMU configuration shown in Table 2 must be modified. The basic initialization code must do all the necessary device configuration, including modifying the MMU TLB entries before the application begins actual execution. The following steps generalize the sequence:

- If the boot code space is VLE, configure the Reset Config Halfword (RCHW) to enable VLE mode. If the boot code space is non-VLE, configure the RCHW with the RCHW[VLE] bit cleared.
- Modify or add MMU TLB entries to support the application memory partitioning and layout. Specifically, set up memory pages for both VLE and non-VLE code.
- Execute all other initialization code, FMPLL setup and any other low level tasks.
- Go to main()

4.1 RCHW and BAM

The RCHW must be configured appropriately, as described in Section 3.1, on page 4. In the case of mixed VLE and non-VLE code, the RCHW may be located in what is configured as either a VLE or non-VLE memory region. The restriction is that the code located at the address specified immediately following the RCHW must be appropriate for the defined page. Example: VLE code on a VLE page or non-VLE code on a non-VLE page.

Once the BAM finds a valid RCHW, it sets up the MMU depending on the RCHW[VLE] bit and ultimately branches to the address specified by the next value above the RCHW in memory. Typically, that address is the beginning of the application initialization code. If the RCHW[VLE] bit is set, that branch to the initialization code must be to a VLE enabled memory page, or if the VLE bit is cleared, that branch must be to a non-VLE enabled memory page.

An example of how the BAM handles VLE and non-VLE boot code is shown in Figure 2 and Figure 3.

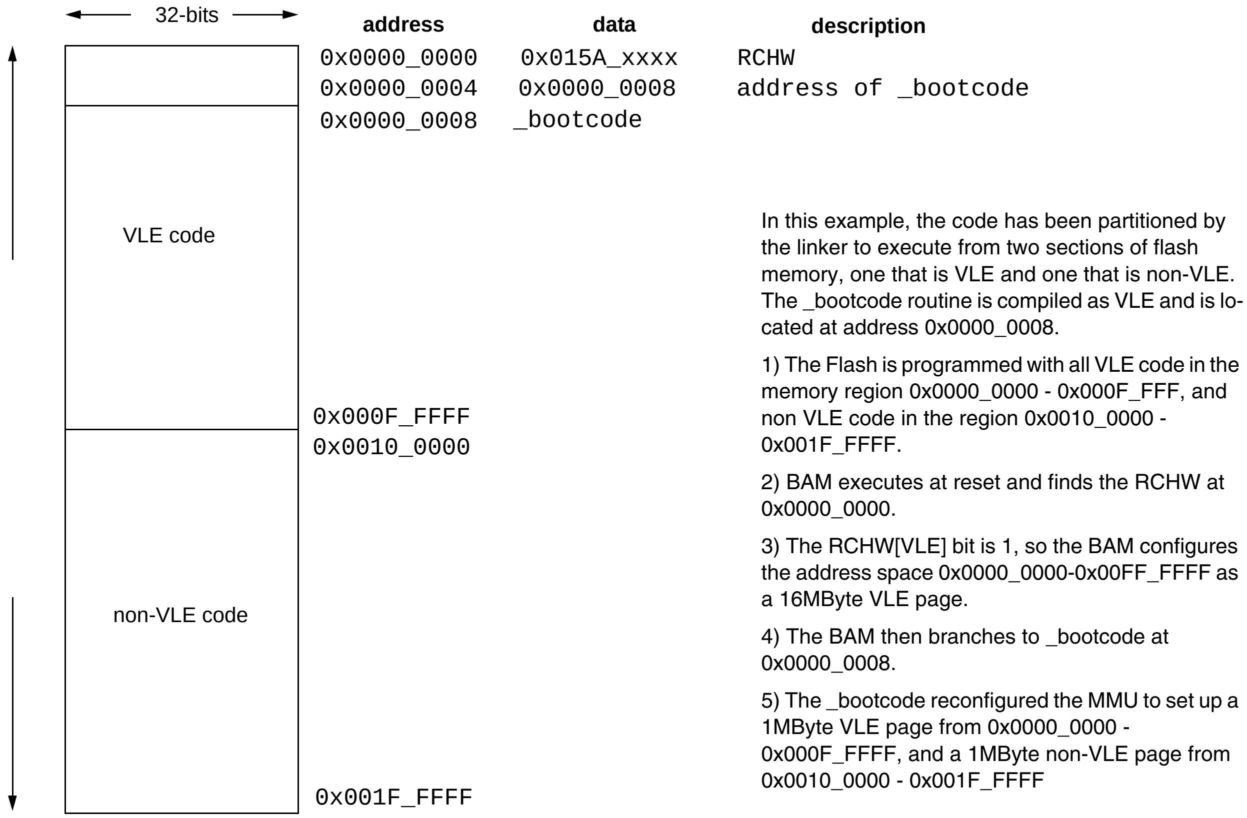

Figure 2. BAM and RCHW Operation — VLE Boot Code

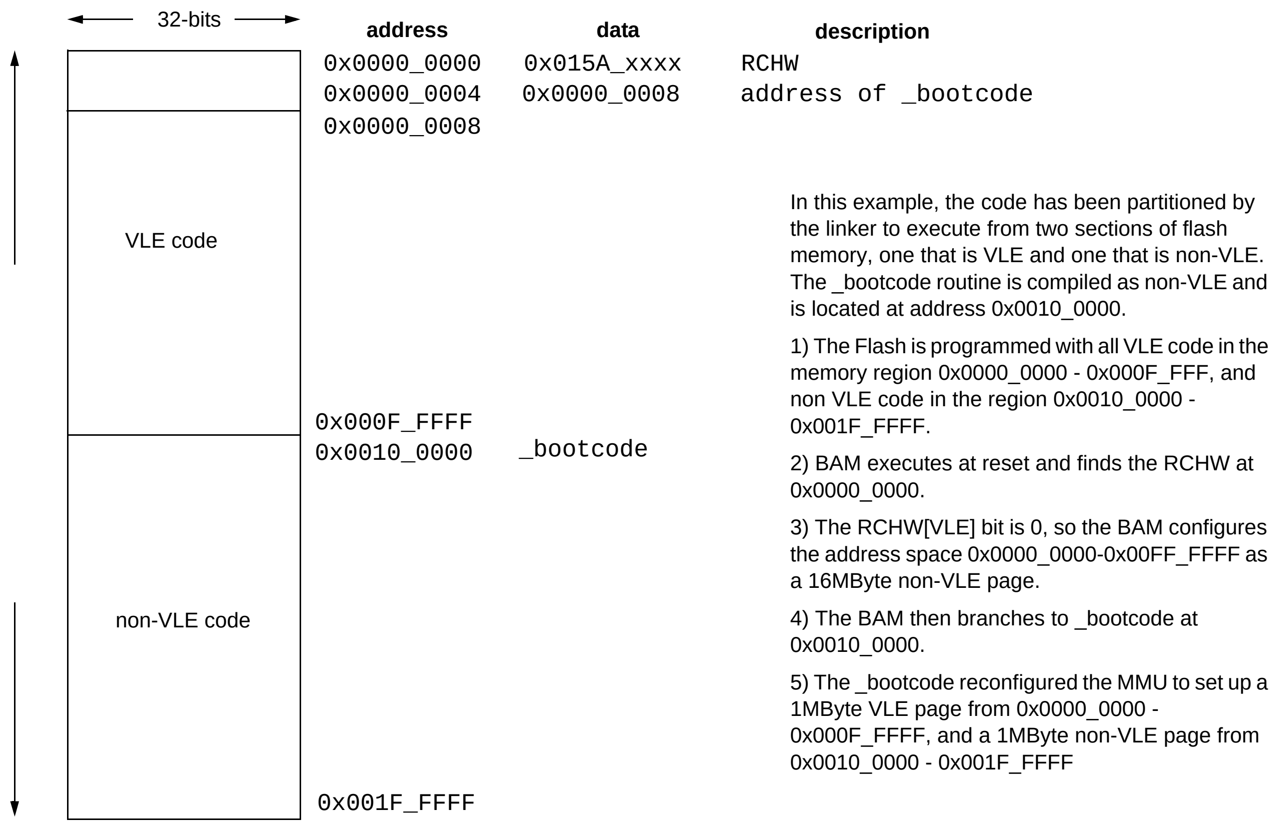

Figure 3. BAM and RCHW Operation — non-VLE Boot Code

In both cases above the partitioning of VLE and non-VLE code, may be configured for multiple memory spaces with sizes determined by the MAS1[TSIZ] bit field.

4.2 Source Code

4.2.1 C Source Files

Source files that are coded in C do not need modification. The compiler is configured to generate VLE code for the parts of the application that are to be programmed into VLE code space. Any C files that contain inline assembler may require modification. That requirement is discussed in Section 4.2.3, on page 9.

4.2.2 Assembly Source Files

As in the case of VLE only applications discussed in Section 2, on page 2, any assembly code intended to be programmed into VLE code space must be coded with the VLE instruction set. A short example of how the port to VLE code may look is shown in Figure 4. Typically, there is a corresponding VLE instruction for each Power Architecture instruction.

Assembly Code - Classic Power Architecture	Equivalent Assembly Code - VLE

`lis r3, IRQ_Svc_Count@h`
`ori r3, r3, IRQ_Svc_Count@l`
`lwz r4, 0(r3)`
`addi r4, r4, 1"`
`stw r4, 0(r3)`
	`e_lis r3, IRQ_Svc_Count@h`
	`e_or2i r3, IRQ_Svc_Count@l`
	`e_lwz r4, 0(r3)`
	`se_addi r4, 1`
	`e_stw r4, 0(r3)`

Figure 4. Converting Power Architecture Assembler Code to VLE — Example

Additionally, the assembly source file may require identification for the assembler regarding the type of code on the page and any alignment requirements. As an example, a typical required assembler directive for a Green Hills assembler file is shown below:

```
     .section ".text", "axv"
     .text
     .vle
reset_vector:
     mfmsr r3
     e_or2is r3, 0x0200
     mtmsr r3
     . . .
```

VLE code that is called from a non-VLE function must also be word (32-bit) aligned to avoid an alignment exception condition. Refer to the documentation for your particular compiler/assembler to find the correct assembler directive.

4.2.3 Inline Assembly Code

Many times an application requires inline assembly code for performance reasons, or to facilitate control of low level resources. This is generally done within a C file by use of compiler syntax to designate that the code is assembler. If the C file containing the inline assembly code is to be compiled with the VLE option, the inline assembly must also be coded in the VLE instruction set.

4.3 Linker and Alignment Requirements

Mixing VLE and non-VLE code requires that the code be linked to the appropriate memory areas as determined by the MMU settings in the application initialization code. The following is an example linker directive file that would match the example shown in Figure 4.

```
MEMORY
{
     /*  2M Internal Flash */
     flash_rchw     : org = 0x00000000, LENGTH = 0x8
     int_flash_vle  : org = 0x00000008, LENGTH = 1M - 0x8
     int_flash_novle: org = 0x00100000, LENGTH = 1M

     /*  64K Internal SRAM */
     int_sram       : org = 0x40000000, LENGTH = 64k

SECTIONS
{
     /* ROM data */
     .rchw          : { *(.rchw) } > flash_rcw
     .vletext       : {} > int_flash_vle
     .flash_data    : {} > .
```

```
      .rodata        : {} > .
      .ctors         : {} > .
      .dtors         : {} > .
      .syscall       : > .
      .fixaddr       : > .
      .fixtype       : > .
      .text          : {} > int_flash_novle
}
```

Any VLE coded functions that are called from non-VLE code space must be aligned on a 32-bit boundary to avoid an alignment exception. There are various ways to achieve this required alignment. Refer to the documentation for your compiler/linker.

5 Summary

When creating or modifying an application to use VLE code, the following points must be kept in mind:

- VLE and non-VLE code must be segregated in memory as defined by the memory management unit (MMU).
- The reset config half-word (RCHW) must be configured appropriately, depending on whether the boot/initialization code is in VLE or non-VLE code space (assuming boot mode is from internal Flash).
- VLE code sections must be 16-bit aligned, with the exception that any function that can be called from code executing in non-VLE code space must be 32-bit aligned.
- Any existing assembler source code that is intended to execute in VLE code space may require porting to the VLE instruction set.
- The source code may require specific #pragmas and/or assembler and compiler directives for the toolset to correctly generate the VLE sections of code. Additionally, the linker directive file may require additional sections or modifications to the existing sections. Refer to the documentation for your tools for the correct procedure.

HOW TO REACH US:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

EB687
Rev. 0, 01/2008