## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Building Applications</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Defining Compiler Options</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Compiling Modules for GDB</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Interface Variations</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Small Data Area</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>HI and HIADJ Macros</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Memory Management Unit</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>AltiVec Support</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Architecture Considerations</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Divide-by-Zero Handling</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Processor Mode</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>26-bit Addressing</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Byte Order</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Hardware Breakpoints</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>PowerPC Register Usage</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>Caches</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>Floating-Point Support</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>VxMP Support for Motorola PowerPC Boards</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Exceptions and Interrupts</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Memory Layout</td>
<td>40</td>
</tr>
</tbody>
</table>
5 Reference Materials .................................................................................................. 42

General PowerPC Documents ................................................................. 42
Motorola PowerPC MPC74xx and AltiVec
Development Reference Documents ................................ 42
PPC405GP Development Reference Documents ......................... 42
PPC440GP Development Reference Documents ......................... 43
1. Introduction

This document provides information specific to VxWorks development on PowerPC targets. It includes the following topics:

- **Building Applications.** How to compile modules for your target architecture.
- **Interface Changes.** Information on changes or additions to particular VxWorks features to support the PowerPC processors.
- **Architecture Considerations.** Special features and limitations of the PowerPC processors, including a figure showing the VxWorks memory layout for these processors.
- **Reference Material.** Sources for current information regarding development on the PowerPC architecture.

For general information on the Tornado development environment’s cross-development tools, see the *Tornado User’s Guide.*
2. Building Applications

The Tornado project facility is correctly preconfigured for building BSPs supplied by Wind River. However, if you choose not to use the project facility or if you need to customize your build, you may need the information in the following sections. This includes a configuration constant, an environment variable, and compiler options that together specify the information the Diab or GNU toolkits require to compile correctly for PowerPC targets.

NOTE: The compiler for PowerPC conforms to the Embedded Application Binary Interface (EABI) protocol. Therefore, type checking is more rigorous than for other architectures.

Defining Compiler Options

PowerPC devices use two preprocessor constants, CPU and TOOL, to define compiler options for a specific device. The CPU variable ensures that VxWorks and your applications are compiled with the appropriate architecture-specific features enabled. The TOOL variable defines the toolchain to use for compiling and linking modules. Specifying CPU and TOOL is usually sufficient to build a module.

NOTE: Modules built with either gnu or diab can be linked together in any combination, except for modules that require C++ support. Cross-linking of C++ modules is not supported in this release. For more information, see the Tornado Migration Guide.

The CPU and TOOL variables should be set to one combination of the values shown in Table 1, depending on the processor you are using.

NOTE: For makefiles having command-line options that differ from those in Table 1, always rely on the options set in the makefile. The exact compiler options for each setting are listed in installDir/target/h/tool/$TOOL/make.$CPU.$TOOL.
<table>
<thead>
<tr>
<th>CPU</th>
<th>TOOL</th>
<th>Supported PowerPC CPU Types</th>
<th>Command-Line Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPC403</td>
<td>diab</td>
<td>PPC403GA, PPC403GC, PPC403GCX</td>
<td>-tPPC403FS:vxworks55 -DCPU=PPC403</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=403 -DCPU=PPC403</td>
</tr>
<tr>
<td>PPC405</td>
<td>diab</td>
<td>PPC405GP</td>
<td>-tPPC405FS:vxworks55 -DCPU=PPC405</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=405 -DCPU=PPC405</td>
</tr>
<tr>
<td>PPC440</td>
<td>diab</td>
<td>PPC440GP</td>
<td>-tPPC440FS:vxworks55 -DCPU=PPC440</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=440 -DCPU=PPC440</td>
</tr>
<tr>
<td>PPC603</td>
<td>diab</td>
<td>PPC603, PPC824x, PPC825x, PPC826x</td>
<td>-tPPC603FH:vxworks55 -DCPU=PPC603</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=603 -DCPU=PPC603</td>
</tr>
<tr>
<td>PPC604</td>
<td>diab</td>
<td>PPC604, PPC604e, MPC745, PPC750, PPC750CX, PPC750CXe, MPC755, MPC7400, (\text{MPC7410, MPC7445, MPC7450, MPC7455} )</td>
<td>-tPPC604FH:vxworks55 -DCPU=PPC604</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=604 -DCPU=PPC604</td>
</tr>
<tr>
<td>PPC604</td>
<td>diab</td>
<td>Altivec-enabled MPC7400, MPC7410, MPC7445, MPC7450, MPC7455 modules</td>
<td>-tPPC7400FV:vxworks55 -Xkeywords=0x0 -DCPU=PPC604</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-fvec -G 0 -fno-builtin -mcpu=604 -DCPU=PPC604</td>
</tr>
<tr>
<td>PPC860</td>
<td>diab</td>
<td>MPC821, MPC823, MPC823e, MPC850, MPC850SAR, MPC855, MPC855T, PPC860</td>
<td>-tPPC860FS:vxworks55 -DCPU=PPC860</td>
</tr>
<tr>
<td></td>
<td>gnu</td>
<td></td>
<td>-G 0 -fno-builtin -mcpu=860 -DCPU=PPC860</td>
</tr>
</tbody>
</table>

* Motorola PowerPC MPC74xx CPUs are treated as a variation of the PowerPC 604 CPU type. Altivec support in the MPC74xx processors is in addition to the existing PPC604 functionality. Modules that make use of Altivec instructions must be compiled with compiler-specific options as described in Table 1, but can be linked with modules that do not use the Altivec compile options. See *Altivec Support*, p. 15, for details.
For example, to specify CPU for a PowerPC 603 on the compiler command line, use the following command-line option when you invoke the compiler:

```
-DCPU=PPC603
```

To provide the same information in a header or source file, include the following line in the file:

```
#define CPU PPC603
```

All VxWorks makefiles pass along the definition of this variable to the compiler. You can define CPU or TOOL on the make command line as follows:

```
% make CPU=PPC603 TOOL=diab
```

You can also set the definition directly in the makefile using the following line:

```
CPU=PPC603
```

### Compiling Modules for GDB

To compile C and C++ modules for debugging in GDB, you must use the -g flag to generate DWARF debug information. An example command line is as follows:

```
% ccppc -mcpu=603 -I/installDir/target/h -fno-builtin \\
-DCPU=PPC603 -c -g test.cpp
```

In this example, `installDir` is the location of your Tornado tree and -DCPU specifies the CPU type.

> **NOTE:** Debugging code compiled with optimization is likely to produce unexpected behavior, such as breakpoints that are never hit or an inability to set breakpoints at some locations. This is because the compiler may re-order instructions, expand loops, replace function calls with in-line code, and perform other code modifications during optimization, making it difficult to correlate a given source line to a particular point in the object code. Users are advised to be aware of these possibilities when attempting to debug optimized code. Alternatively, users may choose to debug applications without using compiler optimization (That is, by compiling without a -O option, or with -O0).
3. Interface Variations

This section describes particular routines and tools that are specific to PowerPC targets in any of the following ways:

- available only for PowerPC targets
- parameters specific to PowerPC targets
- special restrictions or characteristics on PowerPC targets

For complete documentation, see the reference entries for the libraries, subroutines, and tools discussed in the following sections.

Small Data Area

The GNU compiler supports the small data area. However, for this release of Tornado for PowerPC, VxWorks does not support the small data area. Therefore, the `-msdata` compiler flag must not be used. The `-G 0` option is recommended on the command line as well.

HI and HIADJ Macros

The HI and HIADJ macros are used in PowerPC assembly code to facilitate the loading of immediate operands larger than 16 bits. The macro HI(x) is the simple high-order 16 bits of the value x. The macro HIADJ(x) is the high-order 16 bits adjusted by bit 15. If bit 15 is set, the value is adjusted by adding 1.

The macro HIADJ(x) must be used whenever the low-order 16 bits are to be used in an instruction that interprets them as a signed quantity (for instance, addi or lwz). If the low-order bits are used in an instruction that interprets them as an unsigned quantity (for instance, ori), the proper macro is HI, not HIADJ.

For example, addi uses a SIGNED quantity, so HIADJ is the proper macro:

```
   lis rx, HIADJ(VALUE)
   addi rx, rx, LO(VALUE)
```

However, ori uses an UNSIGNED quantity, so HI is the proper macro:

```
   lis rx, HI(VALUE)
   ori rx, rx, LO(VALUE)
```
Memory Management Unit

VxWorks provides two levels of virtual memory support: the basic level bundled with VxWorks, and the full level that requires the optional product VxVMI. Currently, VxVMI is supported on PowerPC 860 family processors only.

For detailed information on VxWorks MMU support, see the VxWorks Programmer’s Guide: Virtual Memory Interface. The following subsections augment the information in that chapter.

Instruction and Data MMU

The PowerPC MMU introduces a distinction between instruction and data MMU and allows them to be separately enabled or disabled. Two parameters, USER_I_MMU_ENABLE and USER_D_MMU_ENABLE, are enabled by default in the Params tab of the Properties window under SELECT_MMU. To disable one or both MMUs, select the corresponding parameter and set the value to FALSE.

- Virtual and Real Equivalence Requirement

In all PowerPC cores except Book E processors, the PowerPC MMU is temporarily disabled by hardware upon interrupts and exceptions. VxWorks also temporarily disables the MMU during certain processor family-specific register changes, cache updates, and page table updates. VxWorks saves the original MMU-enabled state and restores the state as quickly as possible, to maintain memory coherency. As a result, VxWorks requires that exception vectors, handlers, certain update code, and task stacks be mapped such that the virtual and real addresses are identical.

One outcome of the necessity that the virtual and real addresses must be equivalent, is that two copies of the VxWorks run-time (for example, on two processors running their own copies of the VxWorks image) cannot share the same physical address space for running VxWorks.

- MMU Translation Model

The VxWorks PowerPC implementations share a common programming model for mapping 4 KB memory pages. The physical memory address space is described by the data structure sysPhysMemDesc[], defined in sysLib.c. This data structure is made up of configuration constants for each page or group of pages. All of the configuration constants defined in the VxWorks Programmer’s Guide: Virtual Memory Interface are available for PowerPC virtual memory pages.

Use of the VM_STATE_CACHEABLE constant, listed in the VxWorks Programmer’s Guide: Virtual Memory Interface for each page or group of pages, sets the cache to copy-back mode.
In addition to `VM_STATE_CACHEABLE`, the following additional constants are supported:

- `VM_STATE_CACHEABLE_WRITETHROUGH`
- `VM_STATE_CACHEABLE_NOT`
- `VM_STATE_WRITEABLE`
- `VM_STATE_WRITEABLE_NOT`
- `VM_STATE_MEM_COHERENCY`
- `VM_STATE_MEM_COHERENCY_NOT`
- `VM_STATE_GUARDED`
- `VM_STATE_GUARDED_NOT`

**NOTE:** Memory coherency page state is only supported for PPC603 and PPC604.

The first constant sets the page descriptor cache mode field in cacheable write-through mode. Cache coherency and guarded modes are controlled by the other constants. There is no default configuration, because each memory region may have specific requirements; see individual BSPs for examples.

For more information regarding cache modes, see *PowerPC Microprocessor Family: The Programming Environments*.

For more information on memory page states, state flags, and state masks, see the *VxWorks Programmer’s Guide: Virtual Memory Interface*.

### PowerPC 60x Memory Mapping

The PowerPC 603 (including PPC82xx) and 604 (including PPC7xx, PPC74xx, collectively the PPC604 family) MMU supports two models for memory mapping. The first, the Block Address Translation (BAT) model, allows mapping of a memory block ranging in size from 128 KB to 256 MB (or larger, depending on the CPU) into a BAT register. The second, the segment model, gives the ability to map the memory in pages of 4 KB. Tornado for PowerPC supports both memory models.

- **PowerPC 603/604 Block Address Translation Model**

  The Block Address Translation (BAT) model takes precedence over the segment model. However, the BAT model is not supported by the VxWorks `vmLib` or cache libraries. Therefore, functions provided by those libraries are not effective, and no errors are reported, in memory spaces mapped by BAT registers. Typically, in VxWorks, the BATs are only used to map large external regions, or PROM/Flash, where fine grain control is unnecessary; this has the advantage of reducing the size of the PTE table used by the segment model.
All PPC603 and PPC604 family members include 4 BATs: 4 Instruction BATs (IBAT) and 4 Data BATs (DBAT). The BAT registers are always active, and must be initialized during boot. Typically, `romInit()` initializes all (active) BATs to zero, so that they perform no translation. No further work is required if the BATs are not to be used for any address translation at all.

**NOTE:** The original PPC specification defined 4 instruction and 4 data BATs; these BATs are always active in all CPUs with BATs. Note that the additional BATs found on Motorola MPC7x5 and MPC74x5, however, are not enabled by default, and may be left disabled if desired.

Motorola MPC7x5 and MPC74x5 CPUs have an additional 4 IBAT and 4 DBAT registers. These extra BATs can be enabled or disabled (`HID0` or `HID1`, depending on the CPU); they are disabled by hardware reset. Configuring these additional BATs for VxWorks is optional.

The IBM PPC750FX also adds 4 IBAT and 4 DBAT registers, but these are always enabled.\(^1\) In this case, the additional BATs must be configured.

The data structure `sysBatDesc[ ]`, defined in `sysLib.c`, handles the BAT register configuration. All of the configuration constants used to fill `sysBatDesc[ ]` are defined in `installDir/target/h/arch/ppc/mmu603Lib.h` for both the PowerPC 603 and the PowerPC 604. Providing the correct entries in `sysBatDesc[ ]` is sufficient to configure the basic 4 BATs; no additional software configuration is required. See below for configuration of all 8 BAT registers. If `sysBatDesc[ ]` is not defined by the BSP, the BATs are left alone after being configured by `romInit()`.

### Enabling Additional BATs

If the extra BATs are to be used, the following steps must be performed in the BSP:

1. Extend the `sysBatDesc[ ]` array to provide initialization values for the additional BATs.
2. Select or write a BAT initialization routine. The MPC7x5 and MPC74x5 initialization functions are provided with this release. The IBM PPC750FX is not available.
3. Connect the initialization routine to the function pointer provided by the kernel, so that the BATs are initialized at the proper time during MMU initialization.

---

1. The IBM PPC750FX was not available for test when this document was written, but is expected to be included in this Tornado release.
The `sysBatDesc[]` array essentially doubles in size, and the order of the entries is fixed. The initial 16 entries are identical in meaning to the original array, so may remain unchanged. For example (from the sp745x BSP):

```c
UINT32 sysBatDesc [2 * (_MMU_NUM_IBAT + _MMU_NUM_DBAT +
    _MMU_NUM_EXTRA_IBAT + _MMU_NUM_EXTRA_DBAT)] =
{
    /* I BAT 0 */
    ((ROM_BASE_ADRS & _MMU_UBAT_BEPI_MASK) | _MMU_UBAT_BL_1M |
     _MMU_UBAT_VS | _MMU_UBAT_VP),
    ((ROM_BASE_ADRS & _MMU_LBAT_BRPN_MASK) | _MMU_LBAT_PP_RW |
     _MMU_LBAT_CACHE_INHIBIT),
    0,0, /* I BAT 1 */
    0,0, /* I BAT 2 */
    0,0, /* I BAT 3 */
    /* D BAT 0 */
    ((ROM_BASE_ADRS & _MMU_UBAT_BEPI_MASK) | _MMU_UBAT_BL_1M |
     _MMU_UBAT_VS | _MMU_UBAT_VP),
    ((ROM_BASE_ADRS & _MMU_LBAT_BRPN_MASK) | _MMU_LBAT_PP_RW |
     _MMU_LBAT_CACHE_INHIBIT),
    0,0, /* D BAT 1 */
    0,0, /* D BAT 2 */
    0,0, /* D BAT 3 */
    /*
    * These entries are for the the I/D BATs (4-7) on the MPC7455/755.
    * They should be defined in the following order.
    * IBAT4U,IBAT4L,IBAT5U,IBAT5L,IBAT6U,IBAT6L,IBAT7U,IBAT7L,
    * DBAT4U, DBAT4L, DBAT5U, DBAT5L, DBAT6U, DBAT6L, DBAT7U, DBAT7L,
    */
    0,0, /* I BAT 4 */
    0,0, /* I BAT 5 */
    0,0, /* I BAT 6 */
    0,0, /* I BAT 7 */
    0,0, /* D BAT 4 */
    0,0, /* D BAT 5 */
    0,0, /* D BAT 6 */
    0,0 /* D BAT 7 */
};
```

The BAT initialization routine is declared as follows:

```c
(void) myBatInitFunc (int * &sysBatDesc[0])
```

This routine reads `sysBatDesc[]`, initializes the BAT registers, and performs any other required setup, for example, configure HID0 for MPC74x5. See the CPU-specific reference manual for the additional BAT register numbers and
configuration information. The following functions initialize the MPC7x5 and MPC74x5, respectively:

```c
/*
 * mmuPpcBatInitMPC74x5 initializes the standard 4 (0-3) I/D BATs &
 * the additional 4 (4-7) I/D BATs present on the MPC74[45]5.
 */
IMPORT void mmuPpcBatInitMPC74x5 (UINT32 *pSysBatDesc);

/*
 * mmuPpcBatInitMPC7x5 initializes the standard 4 (0-3) I/D BATs &
 * the additional 4 (4-7) I/D BATs present on the MPC7[45]5.
 */
IMPORT void mmuPpcBatInitMPC7x5 (UINT32 *pSysBatDesc);
```

Finally, the BAT initialization routine must be connected to the MMU initialization hook, `_pSysBatInitFunc`, which is NULL by default:

```c
IMPORT FUNCPTR _pSysBatInitFunc;

_pSysBatInitFunc = mmuPpcBatInitMPC7x5;
```

The assignment to `_pSysBatInitFunc` may be made conditional upon the value of the PVR, to allow the same kernel to run on different CPUs.

### PowerPC 603/604 Segment Model

The segment model allows memory to be mapped in pages of 4 KB. All mapping attributes are defined in the individual page descriptors (write-through/copy-back, cache-inhibited, memory coherent, guarded, execute, and write permissions).

The application programmer interface for the PowerPC 603/604 memory mapping unit is the same as that described previously for MMU translation model.

The page table size depends on the total memory to be mapped. The larger the memory to be mapped, the bigger the page table. The VxWorks implementation of the segment model follows the recommendations given in *PowerPC Microprocessor Family: The Programming Environments*. During MMU library initialization, the total size of the memory to be mapped is computed, allowing dynamic
determination of the page table size. Table 2 shows the correspondence between the total amount of memory to map and the page table size.

Table 2  Page Table Size

<table>
<thead>
<tr>
<th>Total Memory to Map</th>
<th>Page Table Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MB or less</td>
<td>64 KB</td>
</tr>
<tr>
<td>16 MB</td>
<td>128 KB</td>
</tr>
<tr>
<td>32 MB</td>
<td>256 KB</td>
</tr>
<tr>
<td>64 MB</td>
<td>512 KB</td>
</tr>
<tr>
<td>128 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>256 MB</td>
<td>2 MB</td>
</tr>
<tr>
<td>512 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td>1 GB</td>
<td>8 MB</td>
</tr>
<tr>
<td>2 GB</td>
<td>16 MB</td>
</tr>
<tr>
<td>4 GB</td>
<td>32 MB</td>
</tr>
</tbody>
</table>

**PowerPC 403 Memory Mapping**

Although some PPC403 family processors have hardware memory mapping units, VxWorks does not support this feature.

**PowerPC 405 Memory Mapping**

The PPC405 memory mapping model allows memory to be mapped in pages of 4 KB. The translation table is organized into two levels: the top level consists of an array of 1,024 Level 1 (L1) table descriptors; each of these descriptors can point to an array of 1,024 Level 2 (L2) table descriptors. All mapping attributes are defined in L2 descriptors (write-through/copy-back, cache-inhibited, guarded, execute, and write permissions).

The translation table size depends on the total memory to be mapped. The larger the memory to be mapped, the bigger the table.

**NOTE:** VxWorks allocates page-aligned descriptor arrays from the heap at virtual memory initialization time. This results in a small amount of initial memory fragmentation.
The application programmer interface for the PowerPC 405 memory mapping unit is the same as that described previously for the MMU translation model.

- **PowerPC 405 Performance**
  
  For optimal performance, the number of TLB entries for data access should be maximized. To eliminate instruction MMU contention for TLB entries, leave `USER_I_MMU_ENABLE` undefined.

**PowerPC 440 Memory Mapping**

The PPC440 core provides a 36-bit physical address space and a 32-bit program (virtual) address space. The mapping is accomplished with Translation Lookaside Buffers (TLBs) which are managed by software.

The PPC440 is an implementation of the Book E processor specification. The MMU is always active and all program addresses are translated by the TLBs. The MSR_{IS} and MSR_{DS} bits are used to extend the virtual address space so that TLB lookups can happen from two different address spaces for either instruction or data references. This easily allows for a static map to be used for boot and basic operation when MSR_{IS,DS} = (0,0) (VxWorks regards this as MMU “disabled”), and enables dynamic 4 KB page mapping (MMU “enabled”) when MSR_{IS} = 1 or MSR_{DS} = 1.

- **Boot Sequencing**

  After a processor reset, the board support package sets up a temporary static memory model. The following steps are included in the BSP `romInit.s` module:

  1. The processor receives a reset exception.
  2. The processor hardware maps a single 4 KB page of memory at the top of the 32-bit program address space and branches to the reset vector (located in the last word of the program address space).
  3. The reset vector contains a branch instruction to `resetEntry()` (located in the last 4 KB of the program address space).
  4. The `resetEntry()` routine initializes the TLB entries to map the entire program address space to physical address space devices and memory, using large size (256 MB) translation blocks. Unused TLBs are marked as invalid. The MSR_{IS} and MSR_{DS} fields are set to zero, and execution continues with an `rti` to the `romInit()` routine.
Run-Time Support

This release of the VxWorks kernel provides support for the PowerPC 440 memory management unit (MMU). To include this support, configure INCLUDE_MMU_BASIC.

Tornado supports two cooperating models for memory mapping. The first, the static model, allows mapping of memory blocks ranging from 1 KB to 256 MB in size by dedicating an individual processor TLB entry to each block. The second, the dynamic model, provides the ability to map physical memory in 4 KB pages using the remaining available TLB entries in a round-robin fashion.

PPC440 Static Model:

The data structure sysStaticTlbDesc[], defined in sysLib.c, describes the static TLB entry configuration. The number of static mappings is variable, depending on the size of the table, but should be kept to a minimum to allow the remaining TLB entries on the chip to be used for the dynamic model.

The static TLB entry registers are set by the initialization software in the MMU library.

Entry descriptions in sysStaticTlbDesc[] that set the _MMU_TLB_TS_0 attribute are used when VxWorks has the MMU “disabled” (that is, MSR[IS,DS] = (0,0)). Note that the VxWorks virtual memory library cannot represent physical addresses larger than the lowest 4 GB, and several of the PowerPC 440GP devices are located at higher physical addresses. To provide access to these devices when VxWorks has the MMU “enabled” (that is, MSR_GS = 1 or MSR_DS = 1), some entry descriptions in sysStaticTlbDesc[] set attribute _MMU_TLB_TS_1.

All of the configuration constants used to fill sysStaticTlbDesc[] are defined in installDir/target/h/arch/ppc/mmu440Lib.h.

PPC440 Dynamic Model:

The PPC440 dynamic mapping model allows memory to be mapped in pages of 4 KB. The translation table is organized into two levels: the top level consists of an array of 1,024 Level 1 (L1) table descriptors; each of these descriptors can point to an array of 1,024 Level 2 (L2) table descriptors. All mapping attributes are defined in L2 descriptors (write-through/copy-back, cache-inhibited, guarded, execute, and write permissions).
The translation table size depends on the total memory to be mapped. The larger the memory to be mapped, the bigger the table.

**NOTE: VxWorks allocates page-aligned descriptor arrays from the heap at virtual memory initialization time. This results in a small amount of initial memory fragmentation.**

The application programmer interface for the PowerPC 440 dynamic model is identical to the MMU translation model described previously.

- **PowerPC 440 Performance**

For optimal performance, the number of TLB entries for data access should be maximized as follows:

1. Minimize the number of static entries defined in `sysStaticTlbDesc[]`.
2. Leave `USER_I_MMU_ENABLE` undefined, eliminating Instruction MMU contention for dynamic TLB entries.

- **PowerPC 8xx Memory Mapping**

The PowerPC 8xx memory mapping model allows you to map memory in 4 KB pages. The translation table is organized into two levels: the top level consists of an array of 1,024 Level 1 (L1) table descriptors; each of these descriptors can point to an array of 1,024 Level 2 (L2) table descriptors. Three mapping attributes are defined in the L1 descriptors (copy-back, write-through, and guarded cache modes), the others (cache off and all access permission attributes) are defined in the L2 descriptors. This affects granularity. For example, if one 4 KB page is mapped in copy-back mode, all pages within the corresponding 4 MB block (1,024 x 4 KB pages) are mapped in copy-back mode, except for any pages having cache off defined. That is, the cache mode setting of a single page can affect the cache mode setting of all mapped pages in the block.

The application programmer interface for the PowerPC 8xx memory mapping unit is described previously for the MMU translation model. PowerPC 8xx processors that implement hardware memory coherency typically do not support the use of the `VM_STATE_MEM_COHERENCY` attribute; the state `VM_STATE_CACHEABLE_NOT` identifies a page as memory coherent.
AltiVec Support

AltiVec is a vector coprocessor and PowerPC instruction set extension introduced on the Motorola PowerPC 74xx family of processors. VxWorks treats AltiVec as an extension to the PowerPC 604 core: A PPC604 binary image can run without modification on any AltiVec part, but does not provide access to, or control of, the AltiVec unit itself. This section describes the VxWorks implementation of AltiVec support, including:

- VxWorks run-time support for AltiVec
- Enabling AltiVec support
- C language extensions for vector types and formatted I/O
- Compiling modules that use the AltiVec unit
- Debugging and GDB extensions for AltiVec
- Tornado tool support; WTX and WDB extensions for AltiVec
- Known problems with C++ mixed linking of AltiVec and non-AltiVec modules

VxWorks Run-Time Support for AltiVec

The following features are supported for the AltiVec unit by the VxWorks kernel.

- AltiVec unit initialization is performed by the `altivecInit()` function. Typically, this is called by the BSP `sysHwInit()` routine if `INCLUDE_ALTIVEC` is defined.
- Run-time detection of the AltiVec unit is possible using the `altivecProbe()` function. This function is used internally by VxWorks to prevent attempts to enable AltiVec for a CPU that lacks such a unit. This allows a single build of a VxWorks kernel to run on boards that support both AltiVec and non-AltiVec parts, for example, the mv5100 family of boards can be configured with either an MPC750/755 or an MPC7400/7410 CPU.
- Tasks that use the AltiVec unit must be spawned with the `VX_ALTIVEC_TASK` option flag set.
- Tasks created without the `VX_ALTIVEC_TASK` option that use AltiVec instructions incur an “AltiVec Unavailable Exception” error, and the task is suspended.
Tasks cannot be spawned with vector parameters. Only integer-sized parameters can be passed to \texttt{taskSpawn()}.  

The MPC74xx processor’s AltiVec registers are saved and restored as part of the task context. The VxWorks kernel saves and restores all 32 AltiVec registers when switching between AltiVec contexts. The value of the VRSAVE register is preserved, but not used, by the context switch code.  

The \texttt{altivecTaskRegsShow()} function displays values of AltiVec registers in the shell.  

The stack frame is correctly aligned to a 16-byte boundary by AltiVec-specific code in each function prologue when the correct compiler option is specified (below). Aligning the stack frame in this way allows AltiVec-enabled functions to share a run-time stack with other PowerPC functions that align their frames on 8-byte boundaries. This stack alignment is performed dynamically by the function prologue code; see Figure 1.  

The \texttt{altivecSave()} and \texttt{altivecRestore()} functions save and restore AltiVec register contents from memory. These functions can be called from interrupt handlers. Before calling these functions, the programmer must ensure that memory has been allocated to store the values, and that the memory is aligned on a 16-byte boundary.  

The AltiVec-specific functions shown in Table 3 have been added to VxWorks.  

<table>
<thead>
<tr>
<th>Routine</th>
<th>Command Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{altivecInit()}</td>
<td></td>
<td>Initializes AltiVec coprocessor support.</td>
</tr>
<tr>
<td>\texttt{altivecTaskRegsShow()}</td>
<td>\texttt{task}</td>
<td>Prints the contents of the AltiVec registers of a task.</td>
</tr>
<tr>
<td>\texttt{altivecTaskRegsSet()}</td>
<td>\texttt{task, ALTIVECREG_SET *}</td>
<td>Sets the AltiVec registers of a task.</td>
</tr>
<tr>
<td>\texttt{altivecTaskRegsGet()}</td>
<td>\texttt{task, ALTIVECREG_SET *}</td>
<td>Gets the AltiVec registers from a task TCB.</td>
</tr>
</tbody>
</table>
Interface Variations

Layout of the AltiVec EABI Stack Frame

The stack frame for functions using the AltiVec registers adds the following areas to the standard EABI frame:

- vector register save area (32 * 128 bytes)
- alignment padding (always zero bytes because the frame is always 16-byte aligned)
- saved VRSAVE register (4 bytes)

Table 3  AltiVec-Specific Routines  (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Command Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>altivecProbe()</td>
<td></td>
<td>Probes for the presence of an AltiVec unit.</td>
</tr>
<tr>
<td>altivecSave()</td>
<td>[ALTIVEC_CONTEXT *]</td>
<td>Saves vector registers to memory.</td>
</tr>
<tr>
<td>altivecRestore()</td>
<td>[ALTIVEC_CONTEXT *]</td>
<td>Restores vector registers from memory.</td>
</tr>
<tr>
<td>vec_malloc()</td>
<td>size_t</td>
<td>Returns a 16-byte aligned pointer for an object of a given size.</td>
</tr>
<tr>
<td>vec_calloc()</td>
<td>size_t nObj, size_t size</td>
<td>Returns a 16-byte aligned pointer for an array of nObj objects each of size size, initialized to 0.</td>
</tr>
<tr>
<td>vec_realloc()</td>
<td>void *p, size_t nbytes</td>
<td>Increases the size of a 16-byte aligned buffer to nbytes.</td>
</tr>
<tr>
<td>vec_free()</td>
<td>void *p</td>
<td>Deallocates the memory area pointed to by p.</td>
</tr>
</tbody>
</table>

NOTE: Memory allocation in VxWorks 5.5 for PowerPC604 is always 16-byte aligned; vec_malloc, vec_calloc, and vec_realloc are aliases for alloc.
The stack frame layout for functions using the AltiVec registers is shown in Figure 1. Non-AltiVec stack frames are unchanged from prior VxWorks releases.

### Alignment Constraints for AltiVec Stack Frames

The required alignment for the SVR4 EABI specification is 16 bytes. This allows an extra 8-byte padding to be introduced in the AltiVec stack frame, and guarantees compatibility with the old 8-byte-aligned environment. AltiVec routines can be called from any other PowerPC EABI-compliant code that assumes an 8-byte alignment for stack boundary.

### C Language Extensions for Vector Types

The AltiVec specification adds a new family of vector data types to the C language. Vector types are 128 bits long, and are used to manipulate values in AltiVec.
registers. Under control of a compiler option, \texttt{vector} is now a keyword in the C and C++ languages. The Altivec programming model introduces five new keywords as simple type-specifiers: \texttt{vector}, \texttt{__vector}, \texttt{pixel}, \texttt{__pixel}, and \texttt{bool}.

⚠️ CAUTION: \texttt{vector} is used as both a C++ class name and a C variable name in the VxWorks header files and some BSP source files, and conflicts with the \texttt{vector} keyword. Where possible, use \texttt{__vector} rather than \texttt{vector} in VxWorks code as a precaution.

### Formatted Input and Output of Vector Types

The \textit{Altivec Technology Programming Interface Manual} also specifies vector conversions for formatted I/O. VxWorks supports the new formatted input and output of \texttt{vector} data types using the \texttt{printf()} and \texttt{scanf()} class functions shown in Table 4.

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument Type; Converted To</th>
</tr>
</thead>
<tbody>
<tr>
<td>%vc</td>
<td>vector unsigned char</td>
</tr>
<tr>
<td>%vd</td>
<td>vector signed int</td>
</tr>
<tr>
<td>%vhd</td>
<td>vector signed short</td>
</tr>
<tr>
<td>%vf</td>
<td>vector float</td>
</tr>
<tr>
<td>%vu</td>
<td>vector unsigned int</td>
</tr>
<tr>
<td>%vs</td>
<td>null-terminated character string</td>
</tr>
</tbody>
</table>

For a comprehensive discussion on the new format specifications, see the \textit{Altivec Technology Programming Interface Manual}. The following example program illustrates the input and output of sample \texttt{vector} values as well as several formatting variations.

```c
void testFormattedIO()
{
    __vector unsigned char s;
    __vector signed int I;
    __vector signed short SI;
    __vector __pixel P;
    __vector float F;
```
s = (__vector unsigned char)
    (\'0\', \'1\', \'2\', \'3\', \'4\', \'5\', \'6\', \'7\', \'8\', \'9\', \'A\', \'B\', \'C\', \'D\', \'E\', \'F\');

I = (__vector signed int) (99, 88, -34, 0);
SI = (__vector signed short) (1, 2, -1, -2, 0, 3, 4, 5);
P = (__vector _pixel) (50, 51, 52, 53, 54, 55, 56, 57);
F = (__vector float) (-3.1415926, 3.1415926, 9.8, 0.000);

printf("s = (%vc), (%,vc)\n\n", s, s);
printf("I = (%,vd), (%,2vld), (%,_3lvi)\n\n", I, I, I);
printf("I = (%,#vd), (%,vlx), (%_1vX), (%vo)\n\n", I, I, I, I);
printf("I = (%,#vd), (%,#vlp), (%,_lvP), (%#vo)\n\n", I, I, I, I);
printf("SI = (%_vhd), (%:hvd), (%;vhi)\n\n", SI, SI, SI);
printf("VECTOR STRING: (%vs)\n\n", "GOOD !!");
printf("VECTOR PIXEL (%+:5hvi)\n\n", P);
printf("VECTOR FLOAT *e5.6*: (%,5.6ve)\n", F);
printf("VECTOR FLOAT *E5.6*: (%:5.6vE)\n", F);
printf("VECTOR FLOAT *g5.6*: (%;5.6vg)\n", F);
printf("VECTOR FLOAT *G5.6*: (%5.6vG)\n", F);
printf("VECTOR FLOAT *f.7* : (%_.7vf)\n", F);
printf("VECTOR FLOAT *e* : (%ve)\n", F);
}

This program generates the following output:

-> testFormattedIO
s = (0123456789ABCDEF), (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
I = (99,88,-34,0), (99,88,-34, 0) (99_88_-34_ 0)
I = (99,88,-34,0), (63,58,ffffffffde,0), (63_58_FFFFFFDE_0), (143 130 37777777736 0)
I = (99,88,-34,0), (0x63,0x58,0xffffffffde,0x0), (0x63_0x58_0xffffffffde_0x0), (0143 0130 037777777736 0)
SI = (1_2_-1_-2_0_3_4_5), (1:2:-1:-2:0:3:4:5), (1;2;-1;-2;0;3;4;5)
VECTOR STRING: (GOOD !!)
VECTOR FLOAT *e5.6*: (-3.141593e+00,3.141593e+00,9.800000e+00,0.000000e+00)
VECTOR FLOAT *E5.6*: (-3.141593E+00,3.141593E+00,9.800000E+00,0.000000E+00)
VECTOR FLOAT *g5.6*: (-3.14159;3.14159; 9.8; 0)
VECTOR FLOAT *G5.6*: (-3.14159 3.14159 9.8 0)
VECTOR FLOAT *f.7* : (-3.1415925_3.1415925_9.80000002_0.0000000)
VECTOR FLOAT *e* : (-3.141593e+00 3.141593e+00 9.800000e+00 0.000000e+00)
value = 76 = 0x4c = \'L\'
->
Compiling Modules with Diab to Use the AltiVec Unit

Modules that use the AltiVec registers and instructions must be compiled with the Diab compiler option: `-tPPC7400FV:vxworks55` (see Table 1). Use of this flag always enables the AltiVec keywords `__vector`, `__pixel`, and `__bool`.

Diab also enables the AltiVec keywords `vector`, `pixel`, `bool` (and `vec_step`) by default if the `-tPPC7400FV` option is used. However, each keyword can be individually enabled or disabled with the Diab compiler (dcc) option `-Xkeywords=<mask>`, where `mask` is a logical OR of the values in Table 5.

**Table 5  Diab -Xkeywords Mask**

<table>
<thead>
<tr>
<th>Mask</th>
<th>Keyword Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>extended</td>
</tr>
<tr>
<td>0x02</td>
<td>pascal</td>
</tr>
<tr>
<td>0x04</td>
<td>inline</td>
</tr>
<tr>
<td>0x08</td>
<td>packed</td>
</tr>
<tr>
<td>0x10</td>
<td>interrupt</td>
</tr>
<tr>
<td>0x20</td>
<td>vector</td>
</tr>
<tr>
<td>0x40</td>
<td>pixel</td>
</tr>
<tr>
<td>0x80</td>
<td>bool</td>
</tr>
<tr>
<td>0x100</td>
<td>vec_step</td>
</tr>
</tbody>
</table>

**NOTE:** Many non-AltiVec-specific keywords are also controlled by `-Xkeywords`.

For example, the following command-line sequence enables `bool` and `vec_step`, but disables `vector` and `pixel` (and also all of the non-AltiVec keywords in Table 5). See the Diab Release Notes for details.

```
dcc -tPPC7400FV:vxworks55 -Xkeywords=0x180 -DCPU=PPC604 -DTOOL_FAMILY=diab -DTOOL=diab -c fioTest.c
```

**CAUTION:** `vector` is used as both a C++ class name and a C variable in the VxWorks header and source files, and conflicts with the `vector` keyword.

The version of the Diab compiler included with this VxWorks release is fully compliant with the Motorola AltiVec EABI document.
Compiling Modules with GNU to Use the AltiVec Unit

Modules that use the AltiVec registers and instructions must be compiled with either the `-fvec` or `-fvec-eabi` flags (see Table 1). The `-fvec` flag disables the use of the new `vector` and `pixel` keywords; `__vector` and `__pixel` are available for these new types. The `-fvec-eabi` flag enables all five keywords as a new family of types: `bool`, `vector`, `__vector`, `pixel`, and `__pixel`.

⚠️ CAUTION: `vector` is used as both a C++ class name and a C variable in the VxWorks header and source files, and conflicts with the `vector` keyword enabled by means of the `-fvec-eabi` flag. Use `-fvec` to compile modules.

The version of the GNU compiler included with this VxWorks release is fully compliant with the Motorola AltiVec EABI document.

Parameter Passing Issues

Code compiled with the GNU toolchain `-fvec` compiler option automatically adjusts its stack pointer dynamically at run-time to align itself on a 16-byte boundary. This feature enables AltiVec code to share the same run-time stack with regular non-AltiVec code which is aligned on 8-byte boundaries. However, this feature can also cause complications when calling AltiVec-enabled functions from non-AltiVec code, with either more than eight integer-class parameters or more than eight floating-point parameters. (Integer-class parameters are `char`, `short`, `int`, pointer types, and so forth.)

The initial eight parameters of each class are passed in registers. The remaining parameters are passed through a special area on the caller’s stack called the `parameter save area`. The callee’s code refers directly to the caller’s stack frame to access these parameters. However, with AltiVec-enabled functions, there may be an 8-byte padding boundary between the two stack frames to fulfill the 16-byte AltiVec alignment constraint. Under these conditions, the callee cannot correctly access its parameters on the stack. The normal parameter passing machinery is broken for such cases.

⚠️ CAUTION: Examples of commonly used AltiVec-enabled functions are the `printf` and `scanf` family of functions. Applications calling these functions with more than eight integer-class or more than eight floating-point arguments may behave unpredictably.
Extensions to the WDB and WTX Protocols for AltiVec Support

The presence and state of the AltiVec unit must also be communicated to the Tornado host tools, such as GDB. The following new WTX and WDB API functions have been added for AltiVec support.

Table 6  WTX API Functions for AltiVec Support

<table>
<thead>
<tr>
<th>Routine</th>
<th>Command Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wtxTargetHasAltivecGet()</td>
<td>hWtx</td>
<td>Returns TRUE if the target has an AltiVec unit.</td>
</tr>
</tbody>
</table>

Table 7  WDB API Functions for AltiVec Support.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Command Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wdbAltivecSave()</td>
<td>void</td>
<td>Saves the AltiVec registers into a buffer.</td>
</tr>
<tr>
<td>wdbAltivecRestore()</td>
<td>void</td>
<td>Restores the AltiVec register values from a buffer.</td>
</tr>
<tr>
<td>wdbAltivecGet()</td>
<td>ppRegs</td>
<td>Gets a pointer to the AltiVec context.</td>
</tr>
<tr>
<td>wdbAltivecSet()</td>
<td>pRegs</td>
<td>Sets the AltiVec context from a buffer.</td>
</tr>
</tbody>
</table>

GDB Extensions

In this release, GDB features a `setaltivec` command that allows users to set a particular value in a given vector register. Typical usage scenarios for the `setaltivec` command are as follows:

```
(gdb) help setaltivec
setaltivec <regname> 0x<hex>_<hex>_<hex>_<hex>
Sets the value of the specific AltiVec register.
```

To set a given value in an AltiVec register using `setaltivec`, use the following:

```
(gdb) setaltivec v4 0x45454545_12345678_12_5A7
```

Vector register contents can be printed using the `print` command:

```
(gdb) print $v4
0x454545451234567800000012000005A7
```
C++ Exception Handling and AltiVec Support

Throwing C++ exceptions between modules compiled with different compiler flags may result in unexpected behavior. C++ exceptions save register state. Modules compiled with AltiVec support (using either -fvec or -fvec-eabi) save all non-volatile AltiVec registers, but modules compiled without AltiVec support do not save any AltiVec registers. If a C++ exception is thrown from an AltiVec-enabled module, caught by a non-AltiVec enabled handler, and then thrown from there to an AltiVec-enabled handler that alters the AltiVec registers, it is possible to corrupt the saved AltiVec state. In particular, the non-volatile vector registers (v20 through v31) may be corrupted.

The following example illustrates the above scenario. It consists of a program composed of two files, file1.cpp and file2.cpp. Because file2 is compiled with the -fvec option, we call it AltiVec code. file1 is compiled without the -fvec option, so it is considered non-AltiVec code.

The example takes program flow across the two modules. It is also contrived to make intelligent guesses about the compiler register allocation strategy. The output is incorrect when one of the files is compiled without the -fvec option.

- Listing For file1.cpp
  
  ```c
  extern "C" int printf (const char *fmp, ...);
  extern void bar ();

  void foo ()
  {
    try
    {
      bar ();
    }
    catch (...)
    {
    }
  }
  ```

- Listing For file2.cpp
  
  ```c
  typedef __vector signed long T;

  void bar ()
  {
    // use a non-volatile vector register
    asm ( "vsplitisw 24,0" ); // v24 <- (0,0,0,0)
  }
  ```
```cpp
void Start ()
{
    // use a non-volatile vector register v24
    T local = (__vector signed long) (-1, -1, -1, -1);
    asm ( "vsplitsisw 24,15" ); // v24 <- (15, 15, 15, 15)
    foo ();
    // continue using the non-volatile vector registers
    asm ( "addi 9, 31, 32" ); // local <- v24
    asm ( "stvx 24, 0, 9" );
    printf ("Finally, local = (%vld), local");
}
```

### Reproduce the Problem

To produce a partially linked object `file2.o`, compile the two files with the following commands:

```
% ccppc -mcpu=604 -c file1.cpp
% ccppc -mcpu=604 -nostdlib -fvec -r file1.o file2.cpp
```

Download `file2.o` to a target, and execute the `Start` function.

```
- > Start
Finally, local = (0,0,0,0)
- >
```

Function `foo` in `file1.cpp` is non-AltiVec code. Therefore, the `try...catch` block in `foo` does not save and restore the AltiVec context. Within the `try...catch` block, the call to `bar` alters the value of vector register v24. Because `file1.cpp` does not save AltiVec context, the value 0 in v24 assigned by `bar` remains unchanged when program flow returns to `Start`. The original value 15, assigned before the call to `bar`, is now corrupted. Hence, the incorrect output, `local = (0,0,0,0)`.

### Correct the Behavior

Compile both files with the `-fvec` option:

```
% ccppc -mcpu=604 -nostdlib -fvec -r file1.cpp file2.cpp -o file2.o
```

Download `file2.o` to a target and execute the `Start` function.

```
- > Start
Finally, local = (15,15,15,15)
- >
```
Because both modules now have AltiVec code (compiled with the -fvec option), the try...catch block in foo now saves and restores the AltiVec context. The value 15 originally assigned in Start is faithfully restored by foo when it returns.

4. Architecture Considerations

This section describes characteristics of the PowerPC architecture that you should keep in mind as you write a VxWorks application. The following topics are addressed:

- divide-by-zero handling
- processor mode
- 26-bit addressing and extended-call exception vector support
- byte order
- hardware breakpoint access types
- PowerPC register usage
- caches
- memory management unit (MMU)
- floating-point support
- VxMP support for Motorola PowerPC boards
- exceptions and interrupts
- debugging
- AltiVec support
- memory layout

For more comprehensive documentation of PowerPC architectures, see the appropriate Motorola microprocessor user’s manual or the IBM user’s manual.

Divide-by-Zero Handling

Integer division by zero produces undefined results. Exception generation and handling are not provided by the compiler or run-time.

Floating-point exceptions are disabled by default during task initialization, causing zero-divide conditions to be ignored. On processors with hardware floating point (PPC603 and PPC604), individual tasks may modify their MSR in order to generate exceptions. On processors without hardware floating point (PPC403, PPC405, PPC440, and PPC860), neither the software floating-point
library nor the compiler provides support for simulating a floating-point exception.

**Processor Mode**

VxWorks 5.5 always runs in supervisor mode on processors in the PowerPC family.

**26-bit Addressing**

VxWorks uses `bl` or `bla` instructions by default for both exception/interrupt handling, and for dynamically downloaded module relocations. By using `bl` or `bla`, the PowerPC architecture is only capable of branching within the limits imposed by a signed 26-bit offset. This limits the available branch range to +/- 32 MB.

**Branching Across Large Address Ranges**

Branches across larger address ranges must be made to an absolute 32-bit address with the help of the LR register. Each absolute 32-bit jump is accomplished with a sequence of at least three instructions (more, if register state must be preserved) that is rarely needed and is expensive in terms of execution speed and code size. Such large branches are typically seen only in very large downloaded modules and very large (greater than 32 MB) system images.

One way of getting around this restriction for downloadable applications is to use the `-mlongcall` compiler option. However, this option may introduce an unacceptable amount of performance penalty and extra code size for some applications. It is for this reason that the VxWorks kernel is not compiled using `-mlongcall`.

Another way to get around this limitation is to increase the size of the WDB memory pool for host tools. By default, the WDB pool size is set to one-sixteenth of the amount of free memory. Memory allocations for host-based tools (such as WindSh and CrossWind) are done out of the WDB pool first, and then out of the general system memory pool. Requests larger than the available amount of WDB pool memory are done directly out of the system memory pool. If an application is anticipated to be located outside of the WDB pool, thus potentially crossing the 32 MB threshold, the size of the WDB memory pool can be increased to ensure the application fits into the required space.
To change the size of the WDB memory pool, redefine the macro `WDB_POOL_SIZE` in your BSP `config.h` file. This macro is defined in `installDir/target/config/all/configAll.h` as follows:

```
#define WDB_POOL_SIZE ((sysMemTop() - FREE_RAM_ADRS)/16)
```

Redefining `WDB_POOL_SIZE` in your BSP local `config.h` file alters the macro for that BSP only.

**Extended-Call Exception Vector Support**

VxWorks 5.5 for PowerPC adds support for extended-call (32-bit addressable) exception vectors.

When exceptions and interrupts occur, PowerPC processors transfer control to a predetermined address, the exception vector, depending on the exception type. After saving volatile task state, the handler function installed for that exception vector is called. This call is made using `bla` instructions that, as described previously, require the handler function to be located within the first 32 MB of memory. Most systems are able to satisfy this 32 MB constraint. However, if a given handler function were to be located above 32 MB, the target address would be unreachable in previous VxWorks releases.

This release adds support for extended-call exception vectors, which can call handler functions located anywhere in the 4 GB address space. Extended-call exception vectors make calls to a 32-bit address in the Link Register (LR) using the `blr!` instructions. Extra work is required for an extended-call exception vector to load a 32-bit address into the LR, and make a call to it. Therefore, using extended-call exception vectors incurs an additional eleven instruction overhead in increased interrupt latency. It is therefore not advisable to use this feature unless absolutely necessary.

This release still maintains the earlier style 26-bit call vectors as the default. Using a single `bl/bla` instruction is much more efficient than the multiple-instruction sequence described previously. It is expected that most targets will continue to use the original relative branch (default) style exception handling.
A new global boolean, `excExtendedVectors`, has been added, that allows users to enable extended-call exception vectors. By default, `excExtendedVectors` is set to `FALSE`. When set to `TRUE`, extended-call vectors are enabled. `excExtendedVectors` must be set to `TRUE` before the exception vectors are initialized in the VxWorks boot sequence (that is, before the call to `excVecInit()`). Setting `excExtendedVectors` after `excVecInit()` does not achieve the desired result, and results in unpredictable system behavior. Selection of extended-call exception vectors is done on a per-BSP basis in order to minimize the impact on those BSPs that do not require this feature.

Enabling Extended-Call Exception Vectors for Command Line BSP Builds

Because `excExtendedVectors` must be set to `TRUE` before the call to `excVecInit()`, users must define the preprocessor define `INCLUDE_SYS_HW_INIT_0`, and also supply a `sysHwInit0()` function that sets `excExtendedVectors` to `TRUE`.

The following example is taken from the ads860 BSP.

Add the following code to `config.h`:

```c
#ifdef INCLUDE_SYS_HW_INIT_0

    /*
     * Perform any BSP-specific initialisation that must be done before
     * cacheLibInit() is called and/or BSS is cleared.
     */

    #ifndef _ASMLANGUAGE
    IMPORT BOOL excExtendedVectors;
    extern void sysHwInit0();
    #endif /* _ASMLANGUAGE */

    #define SYS_HW_INIT_0 sysHwInit0
    #endif /* INCLUDE_SYS_HW_INIT_0 */
```

NOTE: There are no VxWorks API changes to support extended-call exception vectors. In earlier releases, the various connect routines would always return OK. In this release, the connect routines may now return ERROR if the normal vector is selected and the handler function is more than 32 MB away from the vector address. Users are henceforth strongly advised not to ignore the return value from any of the `xxxConnect` family functions - `excConnect()`, `intConnect()`, `excCntConnect()`, `excIntCntConnect()`, and `excIntCntConnectTimer()`. If any of these functions return ERROR, the system may not function as expected, or even fail to boot altogether. Using extended-call exception vectors is the only option left to the user.
Now, add the following code to `sysLib.c`:

```c
#ifdef INCLUDE_SYS_HW_INIT_0

/************************************************************************
* sysHwInit0 - Used here to enable extended exception vector support.
* *
* RETURNS: None.
*/

void sysHwInit0 ()
{
    excExtendedVectors = TRUE;/* enable extended-call exc. vectors */
}
#endif /*INCLUDE_SYS_HW_INIT_0 */
```

### Enabling Extended-Call Exception Vectors for Project Builds

The `INCLUDE_EXC_EXTENDED_VECTORS` component must be enabled for your project. This component sets `excExtendedVectors` to `TRUE` before `excVecInit()` is called during the boot sequence. `INCLUDE_EXC_EXTENDED_VECTORS` is found in the kernel folder.

### Byte Order

The byte order used by VxWorks for the PowerPC family is big-endian.

### Hardware Breakpoints

Not all target architectures support hardware breakpoints, and those that do accept different values for the access type passed to the `bh()` routine. The PowerPC family supports hardware breakpoints, however, the access type of hardware breakpoints allowed depends upon the specific processor.

For each processor family, the number of hardware breakpoints (a hardware limitation), address alignment constraints, and access types are detailed in the following tables. Both instruction and data access must be 4-byte aligned unless otherwise noted.

For more information, see the reference entry for `bh()`.
PowerPC 403 and 405

IBM PPC403 and PPC405 targets have two data breakpoints and two instruction breakpoints.

Address data parameters are 1-byte aligned if width access is 1 byte, 2-bytes aligned if width access is 2 bytes, 4-bytes aligned if width access is 4 bytes, and cache-line-size aligned if access is a data cache line (16 bytes on PPC403, 32 bytes on PPC405). Instruction accesses are always 4-byte aligned.

IBM PPC40x processors allow the following access types for hardware breakpoints; note that the access type arguments are slightly different between PPC403 and PPC405. The byte width means break on all accesses between (addr) and (addr + x):

Table 8  PPC403 Access Types

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Breakpoint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Data write byte (one byte width).</td>
</tr>
<tr>
<td>2</td>
<td>Data read byte (one byte width).</td>
</tr>
<tr>
<td>3</td>
<td>Data read/write byte (one byte width).</td>
</tr>
<tr>
<td>5</td>
<td>Data write half-word (two bytes width).</td>
</tr>
<tr>
<td>6</td>
<td>Data read half-word (two bytes width).</td>
</tr>
<tr>
<td>7</td>
<td>Data read/write half-word (two bytes width).</td>
</tr>
<tr>
<td>9</td>
<td>Data write word (four bytes width).</td>
</tr>
<tr>
<td>0xa</td>
<td>Data read word (four bytes width).</td>
</tr>
<tr>
<td>0xb</td>
<td>Data read/write word (four bytes width).</td>
</tr>
<tr>
<td>0xd</td>
<td>Data write cache line (sixteen bytes width), PPC403 only.</td>
</tr>
<tr>
<td>0xe</td>
<td>Data read cache line (sixteen bytes width).</td>
</tr>
<tr>
<td>0xf</td>
<td>Data read/write cache line (sixteen bytes width).</td>
</tr>
</tbody>
</table>
PowerPC 603

The Motorola PPC603 processor has a single instruction breakpoint, and no data breakpoints. PPC603 allows the following access types for hardware breakpoints:

Table 9  PPC405 Access Types

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Breakpoint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Data write byte (one byte width).</td>
</tr>
<tr>
<td>2</td>
<td>Data read byte (one byte width).</td>
</tr>
<tr>
<td>3</td>
<td>Data read/write byte (one byte width).</td>
</tr>
<tr>
<td>4</td>
<td>Data write half-word (two bytes width).</td>
</tr>
<tr>
<td>5</td>
<td>Data read half-word (two bytes width).</td>
</tr>
<tr>
<td>6</td>
<td>Data read/write half-word (two bytes width).</td>
</tr>
<tr>
<td>7</td>
<td>Data write word (four bytes width).</td>
</tr>
<tr>
<td>8</td>
<td>Data read word (four bytes width).</td>
</tr>
<tr>
<td>9</td>
<td>Data read/write word (four bytes width).</td>
</tr>
<tr>
<td>0xa</td>
<td>Data write cache line (32 bytes width).</td>
</tr>
<tr>
<td>0xb</td>
<td>Data read cache line (32 bytes width).</td>
</tr>
<tr>
<td>0xc</td>
<td>Data read/write cache line (32 bytes width).</td>
</tr>
</tbody>
</table>

PowerPC 604 (including 75x, 74xx, 8xx) and PowerPC 440

The Motorola PPC604/75x/74xx/8xx CPUs have one data and one instruction breakpoint. Data and instruction access must be 4-byte aligned.

The PPC8xx and PPC440 have 4 instruction and 2 data breakpoints. Data access is 1-byte aligned on PPC8xx and PPC440 CPUs.
All of these processors allow the following access types for hardware breakpoints:

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Breakpoint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Data read/write.</td>
</tr>
<tr>
<td>2</td>
<td>Data read.</td>
</tr>
<tr>
<td>3</td>
<td>Data write.</td>
</tr>
</tbody>
</table>

**PowerPC Register Usage**

The PowerPC conventions regarding register usage, stack frame formats, parameter passing between routines, and other factors involving code inter-operability, are defined by the ABI (Application Binary Interface) and the EABI (Embedded Application Binary Interface) protocols. The VxWorks implementation for PowerPC follows these protocols. Table 12 shows PowerPC register usage in VxWorks (note that only CPUs with hardware floating-point support have fpr0-31).

**Table 12 PowerPC Registers**

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpr0</td>
<td>Volatile register which may be modified during function linkage.</td>
</tr>
<tr>
<td>gpr1</td>
<td>Stack frame pointer, always valid.</td>
</tr>
<tr>
<td>gpr2</td>
<td>Second small data area pointer register (<em>SDA2_BASE</em>). VxWorks does not support SDA.</td>
</tr>
<tr>
<td>gpr3-gpr4</td>
<td>Volatile registers used for parameter passing and return values.</td>
</tr>
<tr>
<td>gpr5-gpr10</td>
<td>Volatile registers used for parameter passing.</td>
</tr>
<tr>
<td>gpr11-gpr12</td>
<td>Volatile registers that may be modified during function linkage.</td>
</tr>
<tr>
<td>gpr13</td>
<td>Small data area pointer register (<em>SDA_BASE</em>). VxWorks does not support SDA.</td>
</tr>
<tr>
<td>gpr14-gpr30</td>
<td>Non-volatile registers used for local variables.</td>
</tr>
<tr>
<td>gpr31</td>
<td>Used for local variables or “environment pointers.”</td>
</tr>
<tr>
<td>sprg4-sprg7</td>
<td>Book E and PPC4xx special purpose registers; used by VxWorks.</td>
</tr>
<tr>
<td>usprg0</td>
<td>Book E special purpose register; not used by VxWorks.</td>
</tr>
</tbody>
</table>
Caches

The following subsections augment the information in the VxWorks Programmer’s Guide: I/O System.

PowerPC processors contain an instruction cache and a data cache. In the default configuration, VxWorks enables both caches. To disable the instruction cache, highlight the USER_I_CACHE_ENABLE macro in the Params tab under INCLUDE_CACHE_ENABLE and remove the TRUE; to disable the data cache, highlight the USER_D_CACHE_ENABLE macro and remove the TRUE.

For most boards, the cache capabilities must be used with the MMU to resolve cache coherency problems. The page descriptor for each page selects the cache mode. This page descriptor is configured by filling the data structure sysPhysMemDesc[] defined in sysLib.c. (For more information about cache coherency, see the reference entry for cacheLib. For information about the MMU and VxWorks virtual memory, see the VxWorks Programmer’s Guide: Virtual Memory Interface. For MMU information specific to the PowerPC family, see Memory Management Unit, p. 6.)

The state of both data and instruction caches is controlled by the WIMG2 information saved either in the BAT (Block Address Translation) registers or in the segment descriptors. Because a default cache state cannot be supplied, each cache can be enabled separately after the corresponding MMU is turned on. For more information on these cache control bits, see PowerPC Microprocessor Family: The Programming Environments, published jointly by Motorola and IBM.

2. W: the WRITETHROUGH or COPYBACK attribute.
I: the cache-inhibited attribute.
M: the memory coherency required attribute.
G: the guarded memory attribute.
PowerPC 403 and 405

The PPC403, and the PPC405 when not using the MMU, control the W, I, and G attributes using Special Purpose Registers (SPRs). (Because they do not provide any hardware support for memory coherency, these processors always consider the M attribute to be off.)

See the respective processor user’s manual for detailed descriptions of the Data Cache Cacheability Register (DCCR), Data Cache Write-through Register (DCWR), Instruction Cache Cacheability Register (ICCR), and Storage Guarded Register (SGR).

PowerPC 440

The following describes exceptions to the above implementation for PowerPC 440 processors:

The Book E specification and the PPC440 core implementation do not provide a means to set a global cache enable/disable state, nor do they permit independently enabling or disabling the instruction and data caches.

In the default configuration, VxWorks enables both caches. If you disable one cache, you must disable the other. To disable both caches, highlight the USER_I_CACHE_ENABLE and USER_D_CACHE_ENABLE macros in the Params tab under INCLUDE_CACHE_ENABLE and remove the TRUE.

The state of both data and instruction caches is controlled by the WIMG information saved either in the static TLB entry registers or in the dynamic memory mapping descriptors. Because a default cache state cannot be supplied, both caches are enabled after the corresponding MMU is turned on.

If an application requires a different cache mode for instruction versus data access on the same region of memory, #undef USER_I_MMU_ENABLE, #define USER_D_MMU_ENABLE, use sysStaticTlbDesc[ ] to set up the instruction access mode, and sysPhysMemDesc[ ] to set up the data access mode.

The VxWorks cache library interface has changed for the following two calls:

```c
STATUS cacheEnable(CACHE_TYPE cache);
STATUS cacheDisable(CACHE_TYPE cache);
```

The cache argument is ignored and the instruction and data caches are both enabled or disabled together. If called before the MMU library is initialized, cacheEnable returns OK and signals the MMU library to activate the cache after it has completed initialization. If the MMU library is active (that is, MSR_DS = 1), cacheEnable returns ERROR.
Floating-Point Support

PowerPC 403, 405, 440, and 860

The PowerPC 403, 405, 440, and 860 processors do not support hardware floating-point instructions. However, VxWorks provides a floating-point library that emulates these mathematical functions. All ANSI floating-point functions have been optimized using libraries from U. S. Software.

acos() asin() atan() atan2()

ceil() cos() cosh() exp()

fabs() floor() fmod() log()

log10() pow() sin() sinh()

sqrt() tan() tanh()

In addition, the following single-precision functions are also available:

acosf() asinf() atanf() atan2f()
cceilf() cosf() expf() fabsf()

floorf() fmodf() logf() log10f()
powf() sinf() sinhf() sqrtf()
tanf() tanhf()

The following floating-point functions are not available on PowerPC 403, 405, 440, and 860 processors:

cbrtf() infinityf() irintf() iroundf()

log2f() roundf() sincosf() trunff()

trunfcf() log2f() roundf() sincosf()

PowerPC 60x

The following floating-point functions are available for PowerPC 60x processors:

acos() asin() atan() atan2()
cceil() cos() cosh() exp()

fabs() floor() fmod() log()

log10() pow() sin() sinh()

sqrt() tan() tanh()
The following subset of ANSI functions is optimized using libraries from Motorola:

- `acos()`  
- `asin()`  
- `atan()`  
- `atan2()`  
- `cos()`  
- `exp()`  
- `log()`  
- `log10()`  
- `pow()`  
- `sin()`  
- `sqrt()`  

The following floating-point functions are not available on PowerPC 60x processors:

- `cbrt()`  
- `infinity()`  
- `irint()`  
- `iround()`  
- `log2()`  
- `round()`  
- `sincos()`  
- `trunc()`  
- `sin()`  

No single-precision functions are available for PPC60x processors.

Handling of floating-point exceptions is supported for PowerPC 60x processors. By default, the floating-point exceptions are disabled.

To change the default for a task spawned with the `VX_FP_TASK` option, modify the values of the Machine State Register (MSR) and the Floating-Point Status and Control Register (FPSCR) at the beginning of the task code.

- The MSR FE0 and FE1 bits select the floating-point exception mode.
- The FPSCR VE, OE, UE, ZE, XE, NI, and RN bits enable or disable the corresponding floating-point exceptions and rounding mode. (See `archPpc.h` for the macro `PPC_FPSCR_VE` and so forth.)

Register values may be accessed by the routines `vxMsrGet()`, `vxMsrSet()`, `vxFpscrGet()`, and `vxFpscrSet()`.

**VxMP Support for Motorola PowerPC Boards**

VxMP is an optional VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running on separate CPUs. For complete documentation of the optional component VxMP, see the *VxWorks Programmer’s Guide: Shared-Memory Objects*.

Normally, boards that make use of VxMP must support hardware test-and-set (TAS: atomic read-modify-write cycle). Motorola PowerPC boards do not provide atomic (indivisible) TAS as a hardware function. VxMP for PowerPC provides special software routines which allow these Motorola boards to make use of VxMP.
Boards Affected

The current release of VxMP provides a software implementation of a hardware TAS for PowerPC-based VME boards of the 1300, 1600, 2600, and 5100 families manufactured by Motorola. No other PowerPC boards are affected.

NOTE: Some PowerPC board manufacturers, for example Cetia, claim to equip their boards with hardware support for true atomic operations over the VME bus. Such boards do not need the special software written for the Motorola boards.

Implementation

The VxMP product for Motorola PowerPC boards has special software routines which compensate for the lack of atomic TAS operations in the PowerPC and the lack of atomic instruction propagation to and from these boards. This software consists of the routines `sysBusTas()` and `sysBusTasClear()`.

The software implementation uses ownership of the VME bus as a semaphore; in other words, no TAS operation can be performed by a task until that task owns the VME bus. When the TAS operation completes, the VME bus is released. This method is similar to the special read-modify-write cycle on the VME bus in which the bus is owned implicitly by the task issuing a TAS instruction. (This is the hardware implementation employed, for example, with a 68K processor.)

However, the software implementation comes at a price. Execution is slower because, unlike true atomic instructions, `sysBusTas()` and `sysBusTasClear()` require many clock cycles to complete.

Configuring VME Bus TAS

To invoke this feature, set SM_TAS_TYPE to SM_TAS_HARD on the Params tab of the project facility under INCLUDE_SM_OBJ.

Restrictions for Multi-Board Configurations

Systems using multiple VME boards where at least one board is a Motorola PowerPC board must have a Motorola PowerPC board as the board with a processor ID equal to 0 (the board whose memory is allocated and shared). This is because a TAS operation on local memory by, for example, a 68K processor does not involve VME bus ownership and is, therefore, not atomic as seen from a Motorola PowerPC board.
This restriction does not apply to systems that have globally shared memory boards which are used for shared memory operations. Specifying $\text{SM\_OFF\_BOARD}$ as $\text{TRUE}$ on the $\text{Params}$ tab of the properties window for the processor with ID of 0 and setting the associated parameters enables you to assign processor IDs in any configuration. (For more information, see the VxWorks Programmer’s Guide: Shared-Memory Objects.)

**Exceptions and Interrupts**

**PowerPC 403, 405, and 440**

PowerPC 403, 405, and 440 processors support two classes of exceptions and interrupts: normal and critical. This release correctly attaches default handlers to both classes of exception handler. This release supplements the $\text{excConnect( )}$ and $\text{intConnect( )}$ functions by adding the $\text{excCrtConnect( )}$ and $\text{excIntCrtConnect( )}$ functions:

```
STATUS excCrtConnect (VOIDFUNCPTR *vectr, VOIDFUNCPTR routine);
STATUS excIntCrtConnect (VOIDFUNCPTR *vectr, VOIDFUNCPTR routine);
```

The $\text{excCrtConnect( )}$ function connects a C routine to a critical exception vector, in a manner analogous to $\text{excConnect( )}$. The $\text{excIntCrtConnect( )}$ routine performs a similar function for an interrupt (also see $\text{excVecGet( )}$ and $\text{excVecSet( )}$, p.39).

The $\text{excIntConnectTimer( )}$ function, required for PPC403 and PPC405 targets, is not needed for the PPC440.

$\text{excVecGet( )}$ and $\text{excVecSet( )}$

Application code need not use the lower-level interfaces, $\text{excCrtConnect( )}$ and $\text{excIntCrtConnect( )}$ (see PowerPC 403, 405, and 440, p.39); instead, application code should use the simpler $\text{excVecGet( )}$ and $\text{excVecSet( )}$ routines, which automatically handle the different cases of critical and non-critical exceptions:

```
FUNCPTR excVecGet (FUNCPTR *vectr);
void excVecSet (FUNCPTR *vectr, FUNCPTR function);
```
Memory Layout

The VxWorks memory layout is the same for all PowerPC processors. Figure 2 shows the memory layout, labeled as follows:

- **Interrupt Vector Table.** Table of exception/interrupt vectors.
- **SM Anchor.** Anchor for the shared memory network and VxMP shared memory objects (if there is shared memory on the board).
- **Boot Line.** ASCII string of boot parameters.
- **Exception Message.** ASCII string of the fatal exception message.
- **Initial Stack.** Initial stack for `usrInit()`, until `usrRoot()` is allocated a stack.
- **System Image.** The VxWorks system image itself (three sections: text, data, and bss). The entry point for VxWorks is at the start of this region, which is BSP dependent (see BSP-specific documentation).
- **Host Memory Pool.** Memory allocated by host tools. The size depends on the the macro `WDB_POOL_SIZE`. Modify `WDB_POOL_SIZE` under `INCLUDE_WDB`.
- **Interrupt Stack.** Size is defined by `ISR_STACK_SIZE` under `INCLUDE_KERNEL`. Location depends on system image size.
- **System Memory Pool.** Size depends on the size of the system image. The `sysMemTop()` routine returns the address of the end of the free memory pool.

All addresses shown in Figure 2 are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as `LOCAL_MEM_LOCAL_ADRS` under `INCLUDE_MEMORY_CONFIG` for each target.

**NOTE:** The PowerPC architecture supports the placement of the exception vector table in high memory (0xfff00000) by setting the IP bit in the MSR; PPC4xx supports arbitrary placement of the EVT via the EVPR/IVPR. However, VxWorks does not support this placement.
Figure 2  VxWorks System Memory Layout (PowerPC)

Address
+0x0000 LOCAL_MEM_LOCAL_ADRS
+0x3000
+0x4100
+0x4200
+0x4300
+0x4c00

BSP dependent value

KEY

= Available

= Reserved

Interrupt Vector Table
(12KB)

SM Anchor
Boot Line
Exception Message

Initial Stack

System Image

System Memory Pool

Interrupt Stack

Host Memory Pool

_system

_end

sysMemTop()
5. Reference Materials

The information given in this section is current at the time of writing; should you decide to use these documents, you may wish to contact the manufacturer for the most current version.

General PowerPC Documents

- http://mcg.motorola.com/cfm/templates/categories.cfm

Motorola PowerPC MPC74xx and AltiVec Development Reference Documents

- AltiVec technology fact sheets, programming models, and white papers available at http://motorola.com/AltiVec
- AltiVec Technology Programming Environments Manual

PPC405GP Development Reference Documents

- IBM PowerPC 405GP Embedded Processor User’s Manual, IBM Microelectronics, GK10-3118-06, 03/01/01.
**PPC440GP Development Reference Documents**


- IBM technical library documents from:
  