

ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Curtiss-Wright / Synergy Microsystems VGMD* at our website: ***Click HERE***

Synergy Microsystems

VxWorks Board Support Package
Software Guide
for Power PC CPU Boards
Tornado 2.0 / VxWorks 5.4

Revision 1.0b April 6, 2001



PRELIMINARY RELEASE

01-0088/SG-VXWK-01b

Table of Contents

Synergy Microsystems....1
Table of Contents....2
About This Document....5
Searching Text....5
Use of Different Fonts....5
Installing VxWorks....6
Revision Level Information....7
Important Changes in the Current Release....7
General....8
Bootrom Revision Level....10
Directories and Files....11
Known Bugs....12
Notes on Multi-Processor Boards....13
General....13
Dual Processor Board Serial Ports....15
Single Processor Software on Dual Processor Boards....16
Dual Processor Board Exception Handling....17
Board Reset....18
Boot Parameters for Ethernet....19
General....19
Multi-Processor Board Boot Parameters....21
Tornado vs. Target Shell....24
Bus Errors and Machine Checks....25
CTRL_X and sysToMonitor()....26
Memory....27
Error Correction Code (ECC) Memory Support....27
High Radiation Environments....29
The CPU's Memory Map....30
Memory Mapping Considerations: Page Tables....32
Dual Processor Board Onboard Memory Allocation and Cache....34
Quad Processor Board Onboard Memory Allocation....37
Wind River/Gnu Memory Coding Restrictions....38
General....38
Notes on Using Method 1....41
Notes on Using Method 2....42

Notes on Using Method 4 43
For Multi-Processor Boards 44
Compiling Application Programs 45
Altivec Support 45
PMC Modules and PCI Bus Configuration 47
PEX3 Support 50
Internal and External (L2) Caches 51
VME Bus Interfaces 53
General 53
VME Slave Port Addressing through the Alma/Universe 55
VME Slave Address Mapping Via The PCI-VME Bridge Chip 56
Size and Address Restrictions 57
Changing the VME Base Slave Address 58
VME Slave Port Addressing through the Direct VME Interface 59
VME and PCI Master Port Definition 60
The uMaster[ ] Array 62
The dMaster[ ] Array 65
Direct VME Master on Dual Processor Boards: sysDirMasterPortSet( ) 67
Universe VME Initialization Defaults 68
sysBusToLocalAdrs and sysLocalToBusAdrs 69
PCI Bus Interfaces 70
Defining Offboard PCI Bus Master/Slave Windows 70
sysBusToLocalAdrsPci and sysLocalToBusAdrsPci 71
Writes to CPU Control Registers (LEDs) 72
Timers 73
The VxWorks Tick Timer 73
The Decrementer Register and Timestamp Driver 74
Interrupts 76
VME Interrupts 77
The VMEbus SYSFAIL Interrupt 79
Interrupts from PCI Devices 80
Multi-Processor Board Interrupts 82
Mailboxes on VGM1 (rev. D and later) and all VGM, VSS and KGM Boards 83
About Mailbox Length 84
Mailbox Software Support 85
Making Bootroms and Kernels 86
Project vs. Non-Project 86
Creating a BSP Project 88
Making A New Bootrom 90

Making A New Kernel 91
Multi-Processor Board Bootroms 93
Using the Boot Flash ROM 94
Using The System Or Expansion Board Flash ROMs 96
Note for Multi-Processor Boards 102
NVRAM (Non-Volatile RAM) 103
General 103
Clock/Calendar 105
SCSI Support 106
General 106
Multi-Processor Boot from Local SCSI 109
Built-in Self Test Capabilities 110
General 110
Multi-Processor Boards 112
PU32 Support 113
Toubleshooting 115
VxWorks LED Diagnostic Information 115
LED Reference Guide 120

About This Document

This document is best viewed with Adobe Acrobat Reader 4 or greater. If the bookmark navigation feature does not appear automatically upon opening the document, please select the `Show/Hide Navigation Pane` icon on the navigation bar at the top of the document, or choose "`Show Bookmarks`" from the "`Window`" menu.

Searching Text

To search for particular words or phrases in this document, please use the "`find`…" option in the "`Edit`" menu of Adobe Acrobat Reader. This document is not accompanied by an index, so the "`search`…" option is not available.

Use of Different Fonts

This document uses different fonts and styles when referring to file names, and specific values within the text. Here is a list of these style definitions:

The names of files, directory locations, commands and screen output are displayed in `Courier 11` typeface, as shown in the example below:

- Edit the `config.h` so that `INCLUDE_FFS_DRV` and `INCLUDE_DOSFS` are defined, and do a `make`. (For projects see the "`Synergy Flash File RAM disk`" component.)

Specific values referred to in the text are displayed in **Techno 11** typeface, as shown in the following example:

- If you were to increase `PCI_M_SIZE` to **0x2000_0000**, for instance, you would have to switch `PCI_M_METHOD` to **PAGE**.

All other document body text is in Times 12 typeface.

Installing VxWorks

If you are using an IBM PC host, run the setup.exe program on the CDROM. Install the software into the `target/` directory.

If you are using a Unix host, just `untar` the `synBSP1_21g.tar` file at the `target/` level.

After this you can remove directories for Synergy CPU models which you are not using. See the "Directories and Files" section in this document to figure out which directories you need. Dual processor boards require both "`x`" and "`y`" directories. Quad processor boards require "`x`", "`y`", "`z`", and "`w`" directories. Do not remove the `target/config/synergy/` directory!

You will also note a directory called "`bootroms/`" on the CDROM. This is an extra "bootroms-only" directory. The bootrom.hex files are also found within the various board directories, but this allows you access them without re-installing the whole BSP. If you want a copy of the bootrom your board was shipped with (assuming it was shipped with rev. 1.2/1h), just copy the appropriate bootrom.hex file from the `bootroms/` directory.

Revision Level Information

This document is for Wind River release 5.4 of VxWorks and 2.0 of Tornado. Wind River Board Support Package (BSP) version is 1.2. It begins with revision "/01a". This was developed from the earlier Tornado 1.0.1 compatible BSP. It supports the Gemini series of Power PC VMEbus CPU boards and the Synergy Compact PCI CPU boards.

Important Changes in the Current Release

Projects

The Synergy BSP now supports compiling the kernel as a Tornado Project. This now allows you to compile your Project as a "bootable application". The BSP still retains the ability to be compiled in the "traditional" way by doing a command line "`make`" on Unix host or a "`Standard BSP Builds`" on the PC host. See the sections of this document entitled "Project versus Non-Project" and "Creating a BSP Project".

End Ethernet Driver

The Synergy ethernet driver is now a full END driver. This is the default ethernet driver that you get whenever you make a kernel. If you want to use the old driver (pre-1.21h), you have to change the "`make`".

Bootrom Incompatibilities - Dual CPU Boards Only

Unfortunately, making the BSP compilable as a project required altering the memory allocation used for dual processors. The new allocation is actually more sensible than the old one, but the old bootroms are incompatible with the new kernels. So, bootroms with rev. 1.21g or earlier cannot load dual processor kernels of rev. 1.21h or later. There is no incompatibility with single processor or quad processor boards. You can make new bootroms - see the bootroms section later in this document. The bootrom.hex files delivered with the BSP can be used unless you require some non-standard configuration. Most users don't need to generate new bootrom.hex files.

Windview Bug Fixed

The bug in previous revs that caused Windview not to work correctly on multi-processor boards has been fixed.

General

Wind River	Synergy	Date
1.2	/1a	10aug99

First release. Supports VGM1, VGM2, VGM5, KGM5, dual and single processor boards, VGMD and the quad processor VSS4 board. BSP itself is not a Tornado "project" yet. For Unix host systems it has to be built using make files in the traditional way. Using Tornado, it requires selecting certain options. See the "Installation Notes" sections above and "Making a New Kernel or Bootrom" below. The Wind River PCI autoconfiguration software for Tornado 2.0 is implemented. See notes on PCI bus below.

1.2 /1b 21aug99

Fixed bug in PC-host makefiles. Fixed bug in `sysLocalToBusAdrsPci()`. Fixed bug in `sysBusTas()` so that `SM_TAS_HARD` works properly now. The 21554 chip is now unconditionally in the PCI exclusion list. `synPpcDecTimer.c` sets the timer rollover limit to maximum (257 sec.) if Windview is not in use. If Windview is in use, it rolls over in 1/60 sec.

1.2 /1c 05jan00

Added support for PReP map (Grackle Mem. Map A). Onboard PCI peripheral devices (SCSI, ether, etc.) are now auto-configured at system startup. New Flash support. Bug fixed in SCSI driver. ECC memory support added. `synMpicIntHandler()` updated for T2 Windview interrupt logging. The BSP rev. & Version string have been added to the bootrom at offset **0x118** from the ROM FWA.

1.2 /1d 10feb00

Fixed bug in PCI PEX3 initialization. Got ECC error reporting working. Fixed bug in VME slave port initialization. Bus error now prints access address. Changed PEX3 IO space allocation to not use 16-bit ISA space.

1.2 /1e 04may00

Fixed a bug in PCI IO space auto-allocation. Added Altivec support for G4/7400 processors. Added L2 sizing to `bootInit.c` for G4's. Added L2 support for G4 processors. Added documentation to `README` file. Fixed a bug in `sysBusTasClearPci()` in `sysBusPci.c`. Added `sysWatchdogEnable()` to `sysLib.c` (VSS4 only!).

1.2 /1f 15jun00

Eliminated the `MAKE_BOOT_ROM` macro from `config.h`. Fixed a Y-processor LED bug in `bootInit.c`. Bug in `pciAutoConfigLib.c` fixed which affected auto-config. on the cPCI bus. Default period for timestamp software changed. Default PCI master/slave window setup in `config.h` changed. Made bootrom and kernel banners show L2 cache ratio and processor clock speed.

1.2 /1g 02oct00

Modified to allow SMon to boot VxWorks. Fixed a bug in `romInit.s` relating to initialization of 512Mb memory boards. Updated the PU32 driver to allow PU32 to be auto-configured. Added new PU32 section to `README` file. Updated PCI auto-configure software to use the latest release from Wind River, which fixes a PEX3 configuration bug. Added code to `usrConfig.c` to clear Universe Master registers as well as the slave registers. Added `INCLUDE_RPC` to default `config.h`. Added error messages to indicate when PCI auto-configure cannot satisfy a request for PCI I/O or Memory space. Fixed a bug on multi-processor boards mapping 1Gb or more of address space (the system was allowing too much space for page tables). Fixed a PCI autoconfiguration bug that incorrectly initialized transparent PCI to PCI bridges residing on the cPCI backplane. Changed the makefile in the `synergy/` directory so it will run correctly with the standard distribution (as opposed to the Synergy BSP development environment).

1.2 /1h 27Dec00

Added a full END driver to the BSP as the default ethernet driver. The BSP can now be compiled as a Tornado 2.0 project, so bootable applications can be built. Changed the dual CPU's memory allocation scheme. Altered `sysMemTop()`. See sections below about "Projects" and dual processors. Fixed bug in memory usage when `INCLUDE_32MB_CODE_LIMIT` is not defined. The Windview bug that afflicted multi-processor boards has been fixed.

Bootrom Revision Level

Up until rev. 1.21b, it has generally been possible to boot a VxWorks kernel using a bootrom of earlier revision than the kernel. With rev. 1.21c and later this is still possible with single processor boards, but not with dual or quad processor boards. Single processor kernels of rev. 1.21c or later can be loaded using earlier bootroms. Dual or Quad processor kernels of rev. 1.21c or later will hang during the boot process if a bootrom of earlier rev. is used. There are bootrom.hex files on the CDROM for all processors. See below for instructions on how to make bootroms for multi-processor boards. It is expected that rev. 1.21c and later bootroms will work for all later revisions of the kernels.

Directories and Files

The standard Synergy Microsystems BSP distribution contains the following directories:

```
synergy/   Files common to all Synergy board models.
svgm2_04/  For 604e VGM2 with 1 processor installed.
svgm2_04x/ For X processor on 604e VGM2 with 2 processors installed.
svgm2_04y/ For Y processor on 604e VGM2 with 2 processors installed.
svgm5/     For VGM5 with PPC 750 with one processor installed.
svgm5x/    For X processor on VGM5 with 2 processors installed.
svgm5y/    For Y processor on VGM5 with 2 processors installed.
svgmd/     For the VGMD board.
svss4x/    For X processor on VSS4 quad processor board.
svss4y/    For Y processor on VSS4 quad processor board.
svss4z/    For Z processor on VSS4 quad processor board.
svss4w/    For W processor on VSS4 quad processor board.
kgm5/      For the KGM5 Compact PCI board.
kgm5x/     For the X processor on the KGM5 with 2 processors.
kgm5y/     For the Y processor on the KGM5 with 2 processors.
```

Note that Synergy does not use the Wind River `config/all/` directory. The `configAll.h`, `bootInit.c`, `bootConfig.c` and `usrConfig.c` have been placed in the `synergy/` directory and slightly altered.

You can use a dual processor CPU board as a single processor board by installing a single processor bootrom and using a single processor kernel. See section on "Bootroms" below.

Known Bugs

- If you select the PReP map (Grackle "Map A") and attempt to use the shared memory driver ("`sm`") with a KGM5, the system will hang during kernel load. `sm` works fine for all board types including KGM5 if the CHRP map is used ("Grackle map B"). `sm` also works with PReP memory map (Map A) for all VME boards. The KGM5 family is the only CPU model that has a problem with `sm`, and then only under Map A. The large majority of users use Map B, so this does not impact many users. This bug is being researched and should be fixed in the next release of the BSP.

- Restart (`Control-X`) is still not reliable for the VSS4, though it has been improved. See the section below on `^X` (`sysToMonitor`). If you `^X` all processors and do the X processor last, it will work. Processor X has to be restarted after the other 3 have started their countdown but before any of them have finished it.

- When compiling for Y, Z, or W processors as projects, you may see some compiler warning messages reporting that the function `sysMailAddress()` is defined with conflicting types. This is an artifact of the `make` process for the multi-processor boards. It is harmless. We are still looking for a solution.

Notes on Multi-Processor Boards

General

IMPORTANT: The descriptions regarding multi-processor boards in this document are largely for dual processor boards, but the same concepts apply to quad processor boards (unless otherwise specified).

If you have a dual processor board, do not just ignore the Y processor. Either boot it fully, or put a single processor bootrom in the board and run it as a single processor board. Leaving the Y processor in its initial state will degrade system performance and possibly produce some odd side effects.

If you are compiling the kernels as projects, it is advisable to make one workspace for the board and to have it contain a separate BSP project for each processor on the board. If you make each one a separate workspace, it becomes quite awkward to make changes since only one workspace can be open at a time.

The multi-processor versions of the Gemini CPU board are treated by VxWorks very much like 2 or 4 different CPU boards. Since VxWorks already handles multiple processors in a single VME board cage, it doesn't require a lot of modifications to support multiple CPUs on the same board. So, two (or 4) separate VxWorks kernels are loaded onto the board - one for each processor. For dual processor boards there is a separate board directory (`svgm2_04x/` for example) for the X processor kernel and another (`svgm2_04y/`) for the Y processor. For quad processor boards there are corresponding `Z` and `W` directories. You configure them independently and do a separate "`make`" or "`build`" operation for each.

The various resources of the board are generally split among the processors. The onboard memory space is split up between the two processors. The front panel serial ports are divided between them. The board has one mailbox for each processor. The X processor runs the ethernet or SCSI (if it is used at all) and handles initialization of the VMEbus master and slave ports. VME interrupts are all directed to the X processor (See the "Mailboxes" section below for a way to change that). Since all VME interrupts are funneled into the one PCI-VME bridge interrupt and each interrupt must be steered to only one of the two processors, only one processor can be interrupted directly when a VME interrupt occurs. X can in turn interrupt Y via Y's mailbox, however (or Z or W via their mailboxes).

In general a single peripheral device is handled entirely by one of the processors. Certain resources are assigned to each processor rather than have the software arbitrate for ownership of things dynamically. However, there are a few exceptions to this rule. Both processors need to be able do VME block transfer,

and there is only one VME BLT circuit on the board, so this is shared via a semaphore. Also, the function of gaining ownership of the VMEbus is performed by one circuit and only one processor can use it at a time, so that is arbitrated for also. (This doesn't affect normal VMEbus read/write, which can be performed by either processor at any time. It relates only to the software actually requesting ownership of the bus - a relatively rare occurrence. It is done in `sysBusTas`.)

As explained in the sections below on Flash ROM, the X processor is the only one that has the Flash primitives on it. It is the only one that can reprogram or erase bootflash, user flash or socketted flash memory.

Dual Processor Board Serial Ports

The two front panel serial ports are split between the two processors. See the definition of `NUM_TTY` in `config.h`. Processor X uses the first serial port and processor Y uses the second.

Single Processor Software on Dual Processor Boards

Single processor bootroms may be used on dual processor boards. The single processor ROMs cause the second processor to hang executing a short no-op loop. Since the ROM enables instruction cache, the Y processor effectively vanishes. It makes no external memory accesses, and makes no use of any peripheral chips (serial, SCSI, timer, etc.). The X processor can then load a single processor kernel.

Do not load a single processor kernel on a dual processor board that has a dual processor ROM in it. The dual processor ROM will cause the Y processor to copy the Y bootrom image into RAM and the Y processor will interfere with the X processor's operation.

Dual Processor Board Exception Handling

The file `synExcArchLib.oo` in each board directory is a slightly altered version of the Wind River exception library. Since both PPC processors expect their exception tables to be at the same place - **0x100** - slightly non-standard front-end exception code is provided. At each exception location there is a short sequence of instructions that determines which processor is being interrupted and switches to another exception table for that processor. As a result, there are 3 exception tables: the real one shared by both processors, and one each for the X and Y copies of VxWorks.

The `synExcArchLib.oo` is a normal .o file whose name has been altered so it won't be deleted by a "`make clean`". This file is provided in elf format only because the source text is proprietary to Wind River and not a normal part of the Wind River software distribution.

Board Reset

To reset the board, push both toggle switches to the right. The VxWorks bootrom does not support a hard reset of an individual processor.

Your software can reset the board by calling `sysReset()`. It takes no arguments.

Boot Parameters for Ethernet

General

For the typical situation of booting VxWorks over ethernet from a host, one would use something like the parameters shown below. Naturally, you have to fill in the proper IP addresses for your network. See the boot parameters section of chapter 3 of the Wind River "User's Guide."

What is shown below is for a single processor board. See the section below for dual and quad CPU boards if you have a multi-processor CPU. If you have a multi-processor board, it is unwise to ignore the other processors (that is, to not set up their boot parameters). Set up their boot parameters and let them load so that they are in a known state. You can run a multi-processor board as a single processor CPU if desired just by using a single processor bootrom and kernel. See the section on multi-processor boards earlier in this document.

```
boot device            : esyf
Unit Number            : 0
processor number       : 0
host name              : YourHost
file name              : /yourPath/wind/target/svgm5/vxWorks
inet on ethernet (e)   : 123.123.123.123
host inet (h)          : 123.123.123.124
user (u)               : yourUserName
flags (f)              : 0x0
target name (tn)       : fooBaz
```

"`esyf`" is the name of one of the ethernet devices that can be used on the Gemini boards. The `esyf` boot device automatically negotiates for 10BaseT or 100BaseT communications, choosing 100BaseT if it is available. There are two other ethernet boot devices that can be chosen, however:

- "`esyft`" will disable the auto-negotiation and force communication in 10BaseT mode.
- "`esyfh`" will disable the auto-negotiation and force communication in 100BaseT mode.

The IP addresses are shown as **123.123.123.123** because those are dependent on details of your local network. Your system administrator should be able to supply you with IP addresses appropriate for your system.

The board's ethernet address is stored in NVRAM `at NV_ENET_ADRS` (normally defined as **0xFFE9E778** in `config.h`) as a three byte number. You can change this with the "`m`" command (bytewise access only). Normally this is set at the factory to be the board's serial number. The digits from the decimal serial number are entered as is with no conversion to Hex. The other three bytes of the ethernet address are the portion that identifies Synergy Microsystems and those are specified by `ENET_DEFAULT` in `svgm1.h`. If you have a duplicate ethernet address conflict on your network, modify the three bytes at **0xFFE9E778**. When the NVRAM fails because the Lithium battery is exhausted, it may manifest as an ethernet address problem. You can manually reset the address using the ROM monitor '`m`' command (bytewise access only!) when the board is powered up until the "snap-on" battery is replaced. See also the section on NVRAM below. For writes, NVRAM allows bytewise access only.

Other boot devices described later in this document are SCSI and a "RAM disk" in flash file. You may also use the Wind River "shared memory" backplane driver called "`sm`". It works as described in the WIND RIVER documentation.

Multi-Processor Board Boot Parameters

The boot parameters for a multi-processor board are filled in very much as they are in a multi- board VxWorks system. For the X processor, if it is the anchor for the system or there is only one board in the system, set the boot device as you normally would (that's `esyf`, normally). The Y processor on the anchor board (or if its the only board in the system) should use **sm=0x4100** as its boot device.

Normally one uses a terminal connected to each processor's serial port to set that processor's boot parameters. However, the command 'y' has been added to the bootrom to allow you to set the Y processor's boot parameters from the X processor terminal. On quad processor boards, the commands 'z' and 'w' allow setting the other processor's boot parameters. These are all equivalent to the 'c' command.

For a dual processor board that is not the anchor board, both X and Y processors should have their boot device set as required by your system configuration. For the default Synergy VMEbus addressing (as described in `synMmuVme.h`) the value is **sm=0x90004100** for both processors. (This assumes using the VME interface of the PCI-VME bridge chip as opposed to the direct port.) Your bootrom must be configured so that the specified VME master address window is defined. You may need to make a new bootrom to accommodate this.

The processor number of X on the anchor board should be **0** and the Y processor should be number **1**. On the second dual board, X is **2** and Y is **3**, and so on. Each dual board uses two processor numbers.

Only processor **0** should have it's "`inet on ethernet`" entry filled in. This field should be blank on all other processors. Fill in "`host inet`" as appropriate and fill in "`inet on backplane`" as appropriate for your network subnet (unless you are using proxy ARP). Typically processor **0**'s would end in **1** and all other processors have backplane IPs ending in a digit equal to their processor number plus 1. (This can be set automatically, too. See `INCLUDE_SM_SEQ_ADDR` in VxWorks Programmer's Guide chapter 5.) All non-anchor processors have their gateway address set equal to the backplane address of the anchor processor. (Note: All of the above is true for systems with multiple single processor boards, too.)

See the VxWorks Programmer's Guide manual for information on using proxy ARP. It allows giving each board in the system it's own IP address, with all of the IP's appearing to be on the same network.

An example of the way boot parameters might be set up for the two processors where processor X is the anchor and using a backplane subnet of **209.0.0.0** is shown below:

****** For the X processor *********

```
boot device             : esyf
processor number        : 0
host name               : phong
```

```
      file name               : /Tornado/wind/target/config/svgm5x/vxWorks
      inet on ethernet (e)    : 197.42.42.235
      inet on backplane(b)    : 209.0.0.1
      host inet (h)           : 197.42.42.19
      user (u)                : vw
      flags (f)               : 0x0
      target name (tn)        : Synergy0
```

******* For the Y processor *******

```
      boot device             : sm=0x4100
      processor number        : 1
      host name               : phong
      file name               : /Tornado/wind/target/config/svgm5y/vxWorks
      inet on backplane(b)    : 209.0.0.2
      host inet (h)           : 197.42.42.19
      gateway inet (g)        : 209.0.0.1
      user (u)                : vw
      flags (f)               : 0x0
      target name (tn)        : Synergy1
```

In order for this example to work, the host system's routing tables must be configured to associate the **209.0.0.0** subnet with the **197.42.42.235** gateway. The rules are the same as those for any VxWorks multi-board system using the `sm` network. The only difference is that if the X processor is number **0**, the Y's processor's access to the shared memory is onboard rather than offboard, so it uses an onboard address (**sm=0x4100**).

If the X processor is not the anchor board, configure the X and Y processors exactly as you would any non-anchor processors in a multi-board VxWorks system, with the additional constraint that Y's processor number = X processor number +1.

For a quad processor board, some setting values increment accordingly. The rest is the same as for the Y processor.

******* For the Z processor *******

```
      boot device             : sm=0x4100
      processor number        : 2
      host name               : phong
      file name               : /Tornado/wind/target/config/svgm5z/vxWorks
      inet on backplane(b)    : 209.0.0.3
      host inet (h)           : 197.42.42.19
      gateway inet (g)        : 209.0.0.1
      user (u)                : vw
      flags (f)               : 0x0
      target name (tn)        : Synergy2
```

******* For the W processor *******

```
      boot device             : sm=0x4100
      processor number        : 3
      host name               : phong
      file name               : /Tornado/wind/target/config/svgm5w/vxWorks
      inet on backplane(b)    : 209.0.0.4
      host inet (h)           : 197.42.42.19
      gateway inet (g)        : 209.0.0.1
      user (u)                : vw
      flags (f)               : 0x0
      target name (tn)        : Synergy3
```

If the multi-processor board is not the anchor, then fill in the boot parameters for each processor as though they were separate VME boards accessing the anchor over the VME bus.

Tornado vs. Target Shell

If you compile the BSP as a project, select the "`VxWorks`" tab in the workspace window and open up the project component folders. The folder labeled "`Development Tool Components`" contains parameters controlling target shell and Tornado Windshell. If you want to use target shell, you must enable the "`target shell components`" and select "`downloaded symbol table`".

If you are compiling the kernel the old way - not as a project - the `config.h` in each board directory is default configured to compile for Tornado rather than the target shell. If you want to compile for the target shell, edit `config.h` in the appropriate board directory. There is a line near the beginning that says:

```
#undef INCLUDE_CONFIGURATION_5_2
```

If you change this to:

```
#define INCLUDE_CONFIGURATION_5_2
```

and then re-make the kernel, the new kernel will come up using the target shell via the board's front panel serial ports. No Tornado communications software will be included in the kernel.

Bus Errors and Machine Checks

When using the target shell, some bus errors and machine checks (usually caused by access to unpopulated address space) will appear to make VxWorks not return to the terminal prompt after printing the error message. If this happens, type `^C` (CTRL-C). The shell needs this to restart itself properly under some circumstances.

The `vxMemProbe()` function does not give correct results when a write to the VMEbus is performed. You should do reads to probe VME memory if possible. `sysVxMemProbe()` has been provided (only for those CPUs using the Universe PCI to VME bridge chip) as a substitute for `vxMemProbe()`. It takes exactly the same arguments as `vxMemProbe()` and returns valid results for all addresses onboard and offboard.

CTRL_X and sysToMonitor()

Single processor boards will perform a restart after ^X (CTRL-X) is typed, as specified by VxWorks. VxWorks will reload, but the board will not reset.

On multi-processor boards, ^X can present a problem. On the VGM2 dual processor boards, both X and Y processors are restarted. This is necessary because of the relationship between the 2 processors and the shared L2 cache hardware.

CPUs using PPC 750 or PPC 7400 processors have a separate L2 cache for each processor, so it is possible to ^X one without disturbing the other. However, often the X processor of the multi-processor board is also the system anchor. In this case doing a ^X on the X processor will leave the other processors without access to the backplane driver (general access to VMEbus is not affected). The restarted X processor has forgotten that the other processors exist. One solution is to ^X all processors in turn, making sure to do the X processor last.

An alternative to the ^X on any VMEbus CPU is executing `sysReset()`. This performs the equivalent of a front panel reset. Of course, if the board is the system controller, then all boards in the cage will see a reset. This may have undesirable side effects depending on the configuration of your system.

On the VSS4 (rev. C or later), you can also enable the watchdog and just let it time out. Just call `sysWatchdogEnable` and, 250ms later, the board will reset.

Memory

Error Correction Code (ECC) Memory Support

The initialization of the ECC/Parity hardware is performed in the bootrom. If ECC/Parity is enabled, the only difference in the kernel is that it contains a couple of extra functions (search `sysLib.c` for `INCLUDE_RAM_PAR_ECC`). So, in general if you change the state of the ECC/Parity feature, you need to remake both the kernel and the bootrom.

The ECC features of an appropriately equipped CPU board are enabled (for non-project kernels) by defining `INCLUDE_RAM_PAR_ECC` in `config.h` and re-making the bootrom. The default is that ECC is enabled. The same macro controls the few ECC functions in the kernel. Since the bootrom can't be built as a project, edit `config.h` and follow the section below on making bootroms.

For a non-project kernel, just make the kernel after making the bootrom. For Unix host, just type "`make`". For PC host use the `Standard BSP Builds` menu.

For a project kernel, look in "`Synergy BSP Config`" project folder for the "`Enable RAM Parity and ECC`".

To disable parity/ECC, disable the `INCLUDE_RAM_PAR_ECC` and make a new kernel and bootrom.

When ECC is enabled that means that ECC bits in RAM are valid and that fetches from RAM will be checked and corrected if possible. Single bit errors are always correctable. Multibit errors are detectable but not correctable. Do not confuse enabling ECC via `INCLUDE_RAM_PAR_ECC` with enabling ECC error reporting which is described below. The two are separate.

The initialization of ECC is handled in the bootrom, not the kernel. Defining `INCLUDE_RAM_PAR_ECC` will cause the bootrom to initialize all of RAM upon power-on or `RESET`, by doing a read and write to each memory location throughout all of memory. LED 4 (on the front panel) will blink while it is doing this. This should take only a few seconds even for large amounts (64 - 128Mb) of RAM. Thereafter each fetch from RAM is checked for ECC correctable errors or uncorrectable errors. Enabling ECC checking in this way does result in somewhat reduced system performance. A bootrom with ECC enabled will print out a message saying so when the bootrom starts up. A bootrom compiled with `INCLUDE_RAM_PAR_ECC` defined will also work for non-ECC memory boards.

For multi-processor boards, `INCLUDE_RAM_PAR_ECC` must be defined in the same state for all processors. That is, for a dual processor board, if it is defined it should be defined in `config.h` for BOTH X and Y processors. If it is undefined for one, it should be undefined for both.

You can control whether detected ECC errors are reported by causing "machine check" exceptions to occur. A machine check is a fatal error exception. Any task that gets a machine check is suspended by VxWorks after it prints an error message identifying the problem. If one of the VxWorks system tasks gets a machine check, it will also be suspended, which generally crashes the system after printing the error message. So, since ECC errors can happen to any task, enabling machine check makes it possible to suspend any application task and possibly crash the system. Since VxWorks itself usually uses only a few percent of the machine cycles, the probability of crashing VxWorks in the event of an ECC error rather than suspending an application task is about 3%.

You must first decide what should happen when a single bit (corrected) ECC error is detected or a multi-bit error is detected. See the "High Radiation Environments" section below. It may affect how you decide to configure the ECC reporting.

Machine check in response to a multi-bit ECC error is controlled by `ECC_MULTIBIT_MCP_ENABLE` that is defined in `config.h` and in the project component. Since multibit errors are rare and very serious, the default for this should be defined as **TRUE**. If you choose to disable machine check on multi-bit errors, you may periodically check the status to see if one has been detected by calling `sysEccMultiBit()`. If an error has been detected it returns **TRUE** and then clears the MPC106 status bit so that a subsequent multibit failure can be detected. Be advised that if multibit errors are occurring, the behavior of the computer is completely unpredictable.

If you have machine check enabled for multibit errors, the exception routine maintains a count of the total number of multibit machine check exceptions that have occurred. That counter is kept at a location whose address is expressed by the macro `ECC_MULTI_BIT_ERRORS_PTR`. This is defined in `svgm1.h` (so you can see what the address is) and behaves syntactically as a pointer to it.

Single bit ECC errors are not as important since they (usually) represent correctable errors. (Be advised that massive multibit errors can fool the hardware and appear as single bit errors.) The MPC106 keeps an 8-bit count of single bit ECC errors. If the error count gets to **0xff**, it sticks at that value. It does not wrap to 0. You can specify a "trigger value" such that if the error count ever reaches that value then a machine check will be generated. The macro `ECC_TRIGGER_VALUE` is defined in `config.h` and its value is the same one used for the trigger. Its default value is **32**.

You can control what happens when the counter reaches the trigger value via the macro `ECC_1BIT_MCP_ENABLE` defined in `config.h` and in the project component. If it is defined as **TRUE**,

then a machine check will be generated when the counter reaches the trigger value. At any time you can call `sysEcc1BitCount()` to obtain the current value of the 8-bit counter register. This is valid regardless of whether machine check is enabled or not. If machine check is enabled, when a machine check happens, the exception routine reads the counter, adds it to a 32-bit single bit error counter in RAM and then zeroes the 8-bit counter. The total 32-bit count of single bit errors is maintained by the machine check exception routine at an address that is the value of the macro `ECC_1_BIT_ERRORS_PTR`. This is defined in `synergy/svgm1.h`. It is used syntactically as a pointer to it.

You can choose to disable machine check on "single bit error trigger exceeded", and instead periodically check the `sysEcc1BitCount()` function to see how bad the error situation is getting. If the number of errors is large, you may choose to stop your application and restart the computer. Or, you can set the trigger value to a high value, leave machine check enabled and also periodically monitor the value.

Remember that the ECC error configuration is determined by the bootrom and the kernel. If you change the configuration, you must regenerate both the bootrom AND the kernel.

NOTE: For multi-processor CPU boards, make sure that the `config.h` files for all processors have the same ECC configuration!

High Radiation Environments

The BSP does not provide a "memory scrubber" task, although they are easily written. Applications running in high radiation environments (such as an aircraft at high altitude) that produce frequent single bit memory failures should have a low priority task that runs infrequently and just reads & writes the next few Kb of memory every time it runs. It should be set up so that it cycles through all of RAM within one or a few hours. That way single bit errors will likely be found before they become multiple bit errors and so they will all be corrected. If you use a scrubber task, you should probably have it clear the single bit error count periodically to prevent single bit error interrupts from ever occurring.

The CPU's Memory Map

The VxWorks BSP for the VGM1, VGM2, VGM5, VGMD, and VSS4 all use Grackle memory map B by default. This complies with the CHRP standard (Common Hardware Reference Platform) and is the memory map described in the Synergy Microsystems CPU manual. The KGM5 & VGM5 family of boards can use memory map A or memory map B. Memory map A complies with the PReP (PPC Reference Platform Specification) standard. It is summarized briefly in the table below. See the CPU manual for a description of memory map B. Memory Map A is not fully supported by the Motorola MPC107, so future CPU boards may not be able to use it. Map B is less confusing and is recommended.

MEMORY MAP A

Description of Address Space	Processor's Address	Address on PCI Bus	PCI master's view of Address
system memory space	0 to 0x3FFF_FFFF	<none>	0x8000_0000 to 0xFFFF_FFFF
reserved	0x4000_0000 to 0x7FFF_FFFF	<none>	<none>
PCI/ISA I/O space	0x8000_0000 to 0x807F_FFFF	0x0000_0000 to 0x007F_FFFF	0x0 to 0x0000_FFFF
PCI I/O space	0x8100_0000 to 0xB7FF_FFFF	0x0100_0000 to 0x3F7F_FFFF	0x0080_0000 to 0x3F7F_FFFF
PCI/ISA Interrupt acknowledge	0xBFFF_FFF0 to 0xBFFF_FFFF	0x3FFF_FFF0 to 0x3FFF_FFFF	<none>
PCI memory space	0xC000_0000 to 0xFEFF_FFFF	0x0000_0000 to 0x3EFF_FFFF	0x0100_0000 to 0x7EFF_FFFF
ROM space	0xFF00_0000 to 0xFFFF_FFFF	<none>	<none>

Memory Map B is not described here because it is described exhaustively in the CPU manual. The chief difference between the two is that the PReP standard translates nearly every type of CPU access into a different address value on the PCI bus; the CPU and the PCI bus have very different views of the address space. The CHRP map does almost no translation. That is, the address that appears on the PCI bus is the same address that the PPC sees, so, CHRP is less confusing. PReP is historically more like what has been used in PCs.

The default memory map that is used for all Synergy boards is Map B. You can change this separately for each board type by editing `svgm1.h`. At about line 100 is a section that has a small group of defines for each of the Synergy board types. To select Map A for one of these board types, just insert these 2 lines into the appropriate section:

```
#undef  PPC_BRIDGE_MAP
#define PPC_BRIDGE_MAP  MPC106_MAP_A
```

NOTE: If you change this, you must regenerate both the bootrom and the kernel in order for it to take effect. A Map A bootrom can't load a Map B kernel, and vice versa.

Memory Mapping Considerations: Page Tables

The discussion in the sections below on PCIbus and VMEbus configuration refer to memory mapping and explain that there are 2 different ways to map memory: Page tables and DBAT registers. For spaces larger than 256Mb, (or if all 4 DBATs are already in use) page tables must be used. Pay attention to how much space you are mapping with page tables. Otherwise, you could run into problems finding a place to put them.

After your system loads, you can find out how much space is mapped with page tables by executing `sysPhysDump()`.

Two important facts about page tables are that 1) the RAM allocated for them is always a power of 2 in length, and 2) The page table must begin at an address that is a multiple of its length. So a 4Mb page table needs 4Mb of space starting at an address which is a multiple of 4Mb.

The amount of space needed for page tables can become rather large. Mapping more than 1 Gb of space requires 16Mb of page table space! The space mapped does not have to be contiguous space to count in this respect. For example, if you define a VME window that's 512 Mb and a PCI space window that's 512Mb, the few other small patches of mapped space (bootrom, etc.) will push you over the 1 Gb figure and 16Mb of table space will be required. In multi-processor systems, the Synergy BSP tries to allocate space for the page tables at the physical end of memory, which is why it is advisable have more than 32Mb of RAM for such systems. So, if you have a multi-processor board with more that 32Mb of RAM, be sure to call `sysEndOfUsableRam()` to find out what is the highest address you can use. (Overwriting the page tables will crash the system hard.)

In single processor systems or in any system that is using the `-mlongcall` patch (that is, a system where `INCLUDE_32MB_CODE_LIMIT` is not defined) the page tables are kept within the VxWorks system itself rather than at the end of RAM. If you are operating with the 32Mb code limit, which is the default, it is impossible for the system to find 16Mb of aligned RAM space below 32Mb. Since VxWorks starts at **0** and `malloc()` allocates memory space from the end of the 32Mb downward, the need for 16Mb of aligned space cannot be met and the system will crash while it is loading, typically with a **0x4** displayed in the board's LEDs.

Given the way the PCI autoconfiguration scheme works (i.e., it's rather wasteful of address space), it is fairly easy to map enough space to get into this condition, especially if you have PCI cards which require large amounts of address space.

There are a few things you can do to fix this problem. First, you should never map more space than you need. For instance, For your VME windows, use the DBATs if possible (this is the default). If you have two cards on the VMEbus which you need to access and they are 1Gb apart in address space, make two small VME master windows instead of one big one. You can have up to 8 VME master windows. Review the section below on VME windows for information on how to do this. See also `synMmuVme.h` for further information.

If you have a single processor board and are running with the 32Mb code limit, and you cannot avoid mapping a huge amount of address space, you have to define `RELOCATE_PAGE_TABLES` in your `config.h` file. If you are using the project facility, this parameter is accessible in the "`32Mb Code Space Limitation`" component. In the `config.h` it is already there, just change it from `#undef` to `#define`. In the project GUI, change it from **FALSE** to **TRUE**.

If you are compiling as a project, you will have to add `$(WIND_BASE)/target/config/synergy/mmuPpcLib.oo` to the `EXTRA_MODULES` macro in the "`Build`" tab. If compiling as a non-project, then you will have to add the file name "`mmuPpcLib.oo`" to the `MACH_EXTRA` line in the makefile in the board directory. This will cause the single processor BSP to place the page table at the end of physical RAM, just as it does for multi-processor systems. Of course, you must start with more than 32Mb of RAM. Note also that you will then have to call `sysEndOfUsableRam()` in order to find out what is the highest RAM address that you can use. If you have 64Mb of RAM (**0x0400_0000**), the last 16Mb, for instance, might now be used for MMU page tables, so `sysEndOfUsableRam()` will return **0x0300_0000**. Prior to relocating the page tables it would have returned **0x0400_0000**. Do not use the area starting at the address that is returned by this function. Overwriting the MMU page tables will crash the system. If you do not have `RELOCATE_PAGE_TABLES` defined, then `sysEndOfUsableRam()` will always just return the physical end of memory.

Dual Processor Board Onboard Memory Allocation and Cache

When the board is reset, each processor starts out using a fixed amount of RAM memory. Nominally this is 8Mb for each processor. It is determined by parameters defined in the makefile in the `x/` and `y/` directories, `synergy/svgm1.h` and the exception library. X occupies the area from **0** to **0x007FFFFF**. Y occupies the area from **0x00800000** to **0x00FFFFFF**.

For the default configuration (MMU enabled, `INCLUDE_32MB_CODE_LIMIT` defined), then an area at the end of physical memory is used to contain the MMU page tables - one set for each processor. Your application cannot use this region which begins at the address returned by `sysendOfUsableRam()`. How much is used for page tables depends on how much address space has been mapped via page tables (as opposed to DBATs). These tables describe all onboard RAM plus whatever VME and/or address PCI space the user may have added (see `synMmuVme.h` and the section on VME master/slave ports below). Typically 4 or 8 Mb is used to contain MMU page tables. The tables are created and filled in automatically by the BSP when it loads. Note that if `INCLUDE_32MB_CODE_LIMIT` is not defined, or there is not more than 32Mb on the board, the page tables are located within the kernel space itself and need not concern the user.

However, when the page tables are located at the end of RAM, your software must not use that area. Stepping on the page tables crashes the system immediately. There is a function in `sysLib.c` called `sysEndOfUsableRam()` which will return the highest RAM address that can be used by your application. Do not use RAM addresses at or above that address.

If `INCLUDE_32MB_CODE_LIMIT` is defined, then after the processors finish loading, only memory below 32Mb has been placed into the system's managed "pool" of memory. It remains for the user to add more memory to the allocation pool (see `memAddToPool` in the reference manual), or not as desired. If `INCLUDE_32MB_CODE_LIMIT` is not defined, then each processor has (by default) taken an equal portion of the RAM above 32Mb and placed it in its managed pool of memory. Use "`memShow(1)`" to see how much RAM each processor thinks it has.

Regardless of the state of `INCLUDE_32MB_CODE_LIMIT`, by the time all the processors have loaded, all of the memory on the board has been described to all the processors' MMUs. All processors can access all of RAM.

If `INCLUDE_MMU_BASIC` is defined (practically speaking, it always is), the memory maps of the two processors are described by `sysPhysMemDesc[]`. Remember, each processor has it's own `sysPhysMemDesc[]` and both describe all of the onboard RAM, most of which is cache enabled. The

processors depend on snooping to maintain cache coherency. You can print out the `sysPhysMemDesc[]` contents by executing the function `sysPhysDump()`. This function will show you how much has been used in each section.

sysPhysMemDesc[] for X processor

Element	Contents
0	Describes the first 5 pages (one page = 4096 bytes) of RAM as uncached.
1	The 50Mb address range from 0xFCF80000 to the end of the address space. This is onboard I/O, CPU control registers, etc. It is marked as uncached. (Same for X and Y.)
2	This describes the X CPU's RAM up to 16Mb.
3	This is the RAM from 16Mb to 32Mb. Normally where the Y kernel resides.
4	Describes first half of RAM above 32Mb (less the amount at the end of the memory used for page tables).
5	Describes the second half of high RAM (less the amount at the end of the memory used for page tables).
6	Describes the page table space at the physical end of RAM. If your board has only 32Mb onboard, the page tables will instead be located in the initial 8Mb allocation for the kernel and this will not be used.

sysPhysMemDesc[] for Y processor (dual)

Element	Contents
0	Describes the first 5 pages (one page = 4096 bytes) of RAM as uncached (for both processors)
1	The 50Mb address range from 0xFCF80000 to the end of the address space. This is onboard I/O, CPU control registers, etc. It is marked as uncached. (Same for X and Y.)
2	This describes the Y CPU's RAM from 16Mb to 32Mb.
3	This is the X processor's RAM from 0 to 16Mb.
4	Describes first half of RAM above 32Mb (less the amount at the end of the memory used for page tables).
5	Describes the second half of high RAM (less the amount at the end of the memory used for page tables).
6	Describes the page table space at the physical end of RAM. If your board has only 32Mb onboard, the page tables will instead be located in the initial 8Mb allocation for the kernel and this will not be used.

This allocation describes memory so that each processor sees all of RAM. All RAM except for the first 5 pages (=5 * 4096) is (initially) marked as cached. The address range above **0xFCF80000** is also marked as uncached for both processors.

Referring to parts of the region above 32Mb as "the X processor's area" or "the Y processor's area" is a bit misleading. Both processors can access both areas. You could for instance, place the entire RAM (exclusive of the page table space!) into X's managed memory pool, if you needed to have large block of RAM in the pool. The entries in `sysPhysMemDesc[]` were provided in case a user wanted to have some different properties ascribed to some areas for some processors.

Quad Processor Board Onboard Memory Allocation

The quad processor board uses the `sysPhysMemDesc[]` array in a different, simpler way. If `INCLUDE_32MB_CODE_LIMIT` is defined (in `config.h`) and there is more than 32Mb of RAM on the board, it initially allocates 8Mb below 32Mb to each processor. It also places the MMU page tables at the high end of memory. This can amount to 16Mb of RAM for the 4 sets of Page tables. The page tables must not be overwritten! Your program can obtain the highest address that it can use - the word immediately in front of the page tables - by calling `sysEndOfUsableRam()`. If there is only 32Mb of RAM on the board, then `sysEndOfUsableRam()` returns the same result as `sysLocalMemTop()`. That is the physical end of RAM, since in that configuration the page tables are contained within the 8Mb system area of each processor's system.

If `INCLUDE_32MB_CODE_LIMIT` is undefined, then the region above 32Mb will be allocated among the various processors. That is, `malloc()` and `free()` will used it. The page tables will be contained within each processor's VxWorks system instead of at the end of RAM.

The region between 32Mb and the page tables is nominally split up into 4 pieces - one for each processor. The percentages for each processor are described by macros in `config.h` called:

```
AFTER_32MB_X_PCNT
AFTER_32MB_Y_PCNT
AFTER_32MB_Z_PCNT
AFTER_32MB_W_PCNT
```

As in the dual processor case described above, this nominal allocation is an artificial distinction. All processors have the same view (by default) of RAM and the user could, for instance, decide to have one processor use this entire region or any other combination. The extra entries in `sysPhysMemDesc[]` were provided in case a user wanted to have some different properties ascribed to some areas for some processors.

You can use `sysPhysDump()` to see how the RAM has been described and how much space was used by page tables.

Wind River/Gnu Memory Coding Restrictions

General

There is a coding restriction that must be observed when writing software for the PPC processor using the Gnu tool chain with VxWorks. If you have 32Mb or less total onboard RAM, then this restriction is invisible and you can ignore this section. If you have more than 32 Mb, you must be aware of it. If you are not using the Gnu tool chain you need to find out if the compiler you are using has any restriction like this and make appropriate changes in `config.h` and/or your application makefiles.

The problem is the standard Gnu compiler uses a 25-bit addressing mode to do function calls. This addressing mode is fast, but it can only access the first 32 Mb of RAM. This means that executable code must reside in the first 32 Mb of RAM, that is, below address **0x02000000**. You can eliminate this restriction by compiling your application using the `-mlongcall` switch. Since Tornado 2.0, this feature is included in the standard Wind River distribution. The `-mlongcall` causes the compiler to generate 32-bit accesses to functions and, therefore, they may be located anywhere in onboard RAM. There is a speed penalty for this change. (See notes at the end of this section.)

To understand the discussion below you must be clear on the distinction between mapping memory and placing memory into the "managed memory pool". All onboard memory gets "mapped", that is, described to the MMU (Memory Management Unit). It must be mapped to be accessible at all. But when the term "managed" is used, it refers to the pool of memory from which `malloc()` allocates RAM and to which `free()` returns it. Blocks of memory are placed into the managed pool by a call to `memAddToPool()`. Some memory is added to the pool automatically when the system is starting up. Other parts are left to be added to the pool by the user if desired. In many applications where you need a few very large RAM regions, you might not want to have that RAM managed.

When you execute the VxWorks "`ld`" command to download an application program, it calls `malloc()` to get a piece of memory into which to place the program. `malloc()` always allocates RAM from the block that was most recently added to the managed pool. If you place memory above 32Mb into the managed pool and then use `ld` to download a program it will be placed in high RAM. If you have not compiled the application with `-mlongcall` as mentioned above, this program cannot execute. It will generate error messages at the time it is loaded.

The Gemini BSP is not dependent on the user having the compiler patch, but it can accommodate it if you wish. In fact, it is probably better not to use the compiler patch because your program will execute faster.

However, the memory usage becomes a bit more complicated. There are 3 different ways to deal with this problem:

- If you choose not to get the compiler patch, then you can operate under the following restriction if you use the standard Gnu compiler. While loading programs, only the first 32Mb of RAM must be in the managed memory pool. After all programs are loaded, then any additional memory can be added to the pool by calling `memAddToPool()` (assuming you want VxWorks to manage that additional memory space). If you do add memory to the pool, then you cannot do any more software downloads after that point. `ld` commands done after that will place the code above 32Mb and it won't execute up there. This is the BSP's default method as specified by the definition of `INCLUDE_32MB_CODE_LIMIT` and `LOCAL_MEM_AUTOSIZE` in `config.h`. For kernels compiled as projects, see the "`32Mb Code Space Limitation`" component. This is the mode in which VxWorks is most commonly used.

- Make sure that the make files that you use to compile your application programs specify `-mlongcall` when they invoke the compiler. In this case undefine the `INCLUDE_32MB_CODE_LIMIT` macro and define the `LOCAL_MEM_AUTOSIZE` or `LOCAL_MEM_SIZE` macros in `config.h` as described by Wind River. If the "`MACH_EXTRA`" line in the makefile lists a file called `mmuPpcLib.oo`, then delete that file from the `MACH_EXTRA` list. If you switch back to defining `INCLUDE_32MB_CODE_LIMIT`, you'll have to put it back.

  For kernels compiled as projects, access the "`32Mb Code Space Limitation`" component to define/undefine `INCLUDE_32MB_CODE_LIMIT`. See the Wind River "`BSP Memory Configuration`" component for `LOCAL_MEM_AUTOSIZE`. The `mmuPpcLib.oo` file is controlled through the `EXTRA_MODULES` macro in the `Builds` tab.

- Compile your applications with `-mlongcall`, and leave `INCLUDE_32MB_CODE_LIMIT` defined. After the kernel loads, have you standard initialization (a script is good for this) place some or all of the memory above 32Mb into management. (If you have a multi-processor board, 2 processors obviously cannot manage the same or overlapping regions!)

- Tell VxWorks how much memory to use by setting `LOCAL_MEM_SIZE` to that value and undefine `LOCAL_MEM_AUTOSIZE` and `INCLUDE_32MB_CODE_LIMIT`. If `LOCAL_MEM_SIZE` is <= **32Mb** it doesn't matter if you have the compiler patch or not. Then write your application code to use the area above `LOCAL_MEM_SIZE` in whatever way is appropriate for you. That area will not be managed by `malloc()`. This is a simple, practical method for many applications. It only works, however, for single processor boards.

As delivered, the Gemini BSP makes the default assumption that you have chosen method 1 above. In detail this means that `sysMemTop()` will never return a result larger than **32Mb** no matter how much RAM there is. This is different from its behavior as defined by Wind River. This behavior of `sysMemTop()` is controlled by a macro in `config.h` called `INCLUDE_32MB_CODE_LIMIT`. If this is defined, then `sysMemTop()` never returns more than **0x02000000**. This is true regardless of the values of `LOCAL_MEM_SIZE` and `LOCAL_MEM_AUTOSIZE`. If `INCLUDE_32MB_CODE_LIMIT` is not defined (or if you have no more than 32Mb of RAM), then `sysMemTop()` works as specified by Wind River.

`sysPhysMemTop()` will always behave as documented in the Wind River reference manual and in Wind River Technical Note #42. Also, the actual total memory size is always available by calling `sysLocalMemTop()` and is also always stored at `(int *)NV_MEMSIZE`. This is a location in NVRAM that is set by `romInit.s` when the system powers up (by reading it from a configuration register on the memory board - it incurs no execution speed penalty). See `svgm1.h`.

Note that if the MMU is enabled (which it normally is), then the Gemini BSP always maps all of the physical memory on the board. All memory is visible to all processors on the board at all times regardless of the state of the macros described in this section.

The file `mmuPpcLib.oo` is a very slightly modified version of the Wind River `mmuPpcLib.o` file. It allows the processor to relocate the MMU page tables to high RAM, which is outside the VxWorks kernel memory area. For multi-processor boards, you must use the Synergy version if `INCLUDE_32MB_CODE_LIMIT` is defined. For single processor boards, you need it only if `RELOCATE_PAGE_TABLES` is defined (or for kernels compiled as projects, it is marked as `TRUE` in the appropriate component). If you are compiling the kernel as a non-project, then you include this file by listing it in the `MACH_EXTRA` line of the make file. If the kernel is a project, then you list `$(WIND_BASE)/target/config/synergy/mmuPpcLib.oo` in the `EXTRA_MODULES` macro in the "`Builds`" tab.

Notes on Using Method 1

This BSP by default has `INCLUDE_32MB_CODE_LIMIT` and `LOCAL_MEM_AUTOSIZE` defined, so it starts with no more than **32Mb** allocated to the managed memory pool. If there's less than 32Mb, it autosizes memory. To use the system, start VxWorks, and then do all necessary `ld` commands to load application programs. After those are completed, call the `memAddToPool()` function to add any additional RAM to the managed pool (so it can be used by `malloc()` and `free()`). After the `addMemToPool()` is executed, no further `ld` commands can be executed.

Also, when using method 1, note that if your programs use large data arrays, these should not be included as part of your program because they will make your program larger than necessary. For instance, instead of:

```
char data[0x01000000];
```

use:

```
char *data;
```

and later in an initialization function, `malloc()` the 16Mb of RAM needed for the "data" array and initialize the "data" pointer to point to it. In this way, the data space will be assigned in the high RAM area and not use up code space.

Notes on Using Method 2

Undefine `INCLUDE_32MB_CODE_LIMIT` in `config.h`. You should decide if you want to specify `LOCAL_MEM_SIZE` or if you want the system to auto-size RAM by defining `LOCAL_MEM_AUTOSIZE`. Install the Gnu compiler patch described above. Then re-make VxWorks and your applications. You can then do `ld` commands at any time; application code will run at any address. The relocation of your data space to high RAM mentioned above is unnecessary. You do not need to re-make the bootroms.

Notes on Using Method 4

Undefine `INCLUDE_32MB_CODE_LIMIT` and `LOCAL_MEM_AUTOSIZE` in `config.h` (or the project GUI). Set `LOCAL_MEM_SIZE` to the desired size (probably 32Mb). Write your application to manage the region above 32Mb for itself. Note also that in this case the above advice about relocating large data arrays to memory not used by VxWorks is still advisable in order to conserve the space in the first 32Mb of RAM for code. Delete `mmuPpcLib.oo` from the `MACH_EXTRA` line in the makefile if you are compiling in the old way. If you are compiling as a project, delete the path to `mmuPpcLib.oo` that was specified in the `EXTRA_MODULES` macro in the "`Build`" tab.

For Multi-Processor Boards

On dual processor boards, `config.h` files (or project component definitions) for both X and Y processors must have the same memory configuration. That is, `LOCAL_MEM_SIZE`, `LOCAL_MEM_AUTOSIZE`, and `INCLUDE_32MB_CODE_LIMIT` must be in the same state for both kernels. If you `#undef INCLUDE_32MB_CODE_LIMIT`, then you must also remove `mmuPpcLib.oo` from the `MACH_EXTRA` line in the makefile or the `EXTRA_MODULES` macro in the Project facility. These two always go together.

On a dual processor board, `LOCAL_MEM_SIZE` describes the RAM used by both processors. So, if `LOCAL_MEM_SIZE` is 20Mb, then the 20Mb will be split between the 2 processors (see `MEM_ALLOC_X_PCNT`). If you set `USER_RESERVED_MEM` to 1Mb, for example, then 1 Mb is removed from use by both processors. `sysMemTop()` will return the address of that end of memory assigned to that processor's kernel. For example, with `LOCAL_MEM_SIZE` set to 20Mb and `USER_RESERVED_MEM` set to 1 Mb, `sysMemTop()` will return 19 Mb for processor Y and 19 Mb for processor X.

Compiling Application Programs

As well as adding (using the command `#include`) the various Wind River header files in your application program, you may wish to `#include synBsp.h`. This header file has a prototype for all the Synergy-specific functions in `sysLib.c`.

A sample makefile for compiling generic applications (non-project) is provided in `sysLib.c`. It is called "`makefile.std`". You can invoke it as:

```
make -f makefile.std myApp.o
```

You will have to edit it to create a target for your application. It is full of examples. If you compile your application as a project, it is of no use to you.

Altivec Support

For those users with G4 (=PPC7400) processors who want to use the Altivec facility of the G4 processor, you must use the VxWorks '`taskSpawn()`' function to spawn a task using the Altivec. You cannot use '`sp()`'. Furthermore, you must turn on the Altivec bit in the 'options' argument to `taskSpawn`. That is done by using the command `OR` in the `VX_ALTIVEC_TASK` macro. The latter is defined in `synBsp.h`, so you'll need to `#include` that too. Note that you should probably `OR` on the `VX_FP_TASK` bit also since your application will probably use the normal floating point unit as well as Altivec. An example would be:

```
taskSpawn "myTask", 100, VX_FP_TASK |VX_ALYIVEC_TASK, 0x5000, start,
0,0,0, 0,0,0, 0,0,0, 0
```

To create a task which uses the G4 processor's Altivec processor, you must compile your application with the special version of the Gnu compiler that supports Altivec instructions and/or use the Altivec-compatible version of the Synergy DSP library.

You can get a copy of the compiler from **www.altivec.org**. When you compile your application, specify the `-fvec` flag to the compiler.

Your application must call `sysAltivecEnable()` before doing any Altivec operations. The VxWorks kernel (rev 1.2) turns the Altivec unit off by default.

When you spawn your Altivec task, be sure to have the floating point bit and the Altivec bit set in the 'options' argument that you supply to `taskSpawn()`. The '`options`' argument should include

`(VX_FP_TASK | VX_ALTIVEC_TASK)`. `VX_ALTIVEC_TASK` is defined in `synergy/synBsp.h`. Your application should probably use `#include synBsp.h` anyway, since it contains prototypes for many useful functions in the Synergy BSP.

NOTE: An Altivec task cannot be started using the "`sp`" command. The `sp` command does not allow you to specify the options word. The Wind River default options word used by `sp` only specifies floating-point support, not Altivec support.

See the `taskSpawn()` function in the Wind River Reference Manual.

PMC Modules and PCI Bus Configuration

In order to install a PMC module, the socketted EPROM must in general be removed. If you have boot code in a 32 pin DIP EPROM/Flash ROM, it can be copied into the surface mount boot Flash ROM soldered on the motherboard using the "`romToFlash`" command (see the "Using the Boot Flash" section below). A bootrom.hex file can also be copied over the network into the boot flash using the `loadRom.o` program. See the section on "Bootroms" below.

The default location in PCI space for the onboard PMC board(s) is determined by the macros in the `config.h` file whose names all begin with "`PCI_M`". See the comments in that file. For kernels compiled as projects, these are also accessible from the Project GUI in the "`PCI Address Space Configuration`" component in the "`Synergy BSP Configuration`" folder.

In earlier BSPs, use of PCI auto-configuration was optional. With rev. 1.21c and later the PCI autoconfiguration software must be used (`INCLUDE_PCI` in `config.h` or the project facility must be defined). This software is used to configure the onboard PCI devices such as SCSI and Ethernet.

The macros defined in `config.h` following the `#define INCLUDE_PCI` (and in the "`PCI Address Space Configuration`" component) are used to configure the slave & master PCI ports for the offboard PCI bus. That is, the PCI bus on the other side of the 21554 PCI to PCI non-transparent bridge. It also controls how much onboard PCI address space there is for PMC modules and the PEX3 PMC carrier.

At the moment these macros are used instead of the Wind River macros in the "`PCI configuration`" component in the "`busses`" folder. The names will be brought into line with Wind River specification soon.

The other file that controls most PCI auto-configure operations is `synergy/sysBusPci.h`. Users will probably not have to modify the parameters defined in this file. The default set up has the processor auto-configure only the devices on the onboard PCI bus or an attached PEX3 expansion module. It does not probe across the 21554 bridge onto the P0 PCI bus (on VME boards) or the Compact PCI bus (on the cPCI series of boards) unless it is told to do so. This is done (for a non-project kernel) by changing the definition of:

```
#define PCI_GLOBAL_CONFIG 0   /* Default value is 0 */
```

in `sysBusPci.h` to be:

```
#define PCI_GLOBAL_CONFIG 1
```

For kernels compiled as projects this parameter is accessed in the "`PCI Address Space Configuration`" component.

A kernel compiled with this macro set to **1** will probe across the 21554 bridge and attempt to auto-configure any device (except another non-transparent bridge) which it encounters.

If you want to access configuration space directly, you may want to use the Wind River-supplied functions such as `pciConfigInByte()`, `pciConfigInLong()`, `pciConfigOutWord()`, etc. See also the functions defined in `target/src/drv/pci/pciConfigShow.c`.

Also see the Synergy-supplied functions `writeMPC()`, `readMPC()` `writeMPC16()` and `readMPC16()` for accessing the PCI bus configuration space (something you probably won't need to do if you use the autoconfiguration software).

There are 2 different methods of expressing a configuration space address. The Wind River method specifies the bus, device, and function as separate numbers. The other method (used by `writeMPC/readMPC`) masks all that information into a single 32-bit quantity. The `MSB` (=bit 31) is always **1**, the bus number is in bits **23-16**, the device is in **15-11**, the function in bits **10-8** and the register offset in the low byte. So the Synergy PMC module begins at **0x80006800**. That is bus **0**, device **0xd**, function **0**, and offset **0**. See `PMC_A_CFG_ADRS` in `target/config/synergy/svgm1.h`. For a VGMD CPU that has 2 PMC connectors, the second PMC device is at **0x80007000** in configuration space. (See `PMC_A1_CFG_ADRS` for further details.)

Access to byte-reversed PCI memory space is handled by `readPCI()`, `writePCI()`, `readPCI8()`, `writePCI8()`, `readPCI16()`, and `writePCI16()`. You can access PCI memory space using all the same functions you use to access RAM, but some portions of it (depending on the device being accessed) may be byte-reversed. These functions perform the conversion to Big Endian format. All of these functions are defined in `target/config/synergy/synMpic.c`. See comments in that file for more information.

The onboard PMC connector is accessed in configuration space at **0x80006800**. See `PMC_A_CFG_ADRS` in `target/config/synergy/svgm1.h`. For a VGMD CPU that has 2 PMC connectors, the second PMC device is at **0x80007000** in configuration space. (See `PMC_A1_CFG_ADRS`.)

Again, using the auto-configure software eliminates the need to know these details.

In general you want your PMC module (or modules) and your PEX3 to be configured by this software. Your PMC driver can then use "geographical" addressing to locate the PMC card, initialize it and use it.

See the `target/src/drv/pci/pciConfigShow.c` file for the various "Show" words available to tell you about PCI devices.

PEX3 Support

The PEX3 is a 6U daughterboard that expands the CPU's PMC slot to 3 PMC slots on the PEX3. There is a DEC 21154 PCI to PCI bridge chip on the PEX3 that supports a secondary PCI bus on the PEX3. The autoconfiguration software should initialize the Base Address Registers of any PMC modules on the PEX3 when the system comes up (except for any that might be listed in the Exclusion list in `target/config/synergy/sysBusPci.h`).

The 3 PMC connectors on the PEX3 module are accessed at **0x80010000**, **0x80010800** and **0x80011000** in configuration space (see `PMC_B_CFG_ADRS`, `PMC_C_CFG_ADR`, and `PMC_D_CFG_ADRSS` in `svgm1.h`). See `readMPC()`, `writeMPC()` and associated functions in `synergy/synMpic.c` for access to configuration space. You can also use the Wind River functions

```
pciConfigInLong(), pciConfigOutLong()
pciConfigInWord(), pciConfigOutWord()
pciConfigInByte(), pciConfigOutByte()
```

to access PEX3 devices in configuration space. The arguments to these Wind River functions include bus number, device number and function number. For a PMC module on the PEX3, the bus number is always **1**. The device number for the 3 PMC modules is **0**, **1** and **2**.

The autoconfiguration software generally makes configuration space accesses unnecessary. If you have trouble configuring a PMC module, use the Wind River `pciHeaderShow()` command to inspect the devices configuration space registers after configuration. See `target/src/drv/pci/pciConfigShow.c` for further details.

See the section below on "Interrupts from PCI Devices" for information on servicing interrupts from PEX3 devices.

Internal and External (L2) Caches

The PPC 604e and 603e processors feature 32Kb of internal instruction cache and 32Kb of internal data cache. Additionally, Synergy provides the user the option of 256Kb or more of external, or "L2" cache. The PPC 750 processor used on the VGM5 typically has even larger, faster L2 cache.

For non-project kernels, the use of these caches and the MMU are controlled by macros defined in the `config.h` file for each board. The section is reproduced below:

```
/*************************************************************************/
/* MMU and Cache options */
#undef     INCLUDE_MMU_FULL
#define    INCLUDE_MMU_BASIC           /* bundled mmu support */

#if defined(INCLUDE_MMU_BASIC)
#define    USER_D_CACHE_ENABLE   /* Enable data cache,copyback w/snooping*/
#define    USER_I_CACHE_ENABLE         /* Enable instruction cache */
#define    INCLUDE_CACHE_SUPPORT
#undef     INCLUDE_L2_CACHE      /* Define this to enable L2 cache. */
/* only ppc 750 gives you a choice of write-thru or copyback on L2. */
#define    L2_CACHE_MODE           (CACHE_WRITETHROUGH)

#    if (PROCS_INSTALLED == 2)
#define    USER_D_CACHE_MODE     (CACHE_COPYBACK)
#    else
#define    USER_D_CACHE_MODE     (CACHE_COPYBACK | CACHE_SNOOP_ENABLE)
#    endif
#define    USER_I_CACHE_MODE     (CACHE_WRITETHROUGH)

#else      /* defined(INCLUDE_MMU_BASIC) */
#undef     INCLUDE_MMU_BASIC
#undef     USER_D_CACHE_ENABLE
#undef     USER_I_CACHE_ENABLE
#undef     INCLUDE_L2_CACHE
#undef     USER_D_CACHE_MODE
#defin     USER_D_CACHE_MODE     CACHE_DISABLED
#defin     USER_I_CACHE_MODE     CACHE_DISABLED
#endif     /* defined(INCLUDE_MMU_BASIC)  */
```

To make a kernel with no cache or MMU, undefine `INCLUDE_MMU_BASIC`.

If you are compiling the kernel as a project, then in the "`Synergy BSP Configuration`" folder in the project GUI, see "`Enable/configure L2 Cache`". Also see the Wind River "`hardware/memory`" folders.

NOTE: Any board with a PPC 750 and an MPC 106 chip (KGM5, VGMD, VGM5, and VSS4) needs to have the MMU enabled in order to run at all. If you compile a kernel for such a board with `INCLUDE_MMU_BASIC` undefined, the MMU will be enabled anyway (cache will be off). The reason is the PPC 750 bootrom unconditionally enables the MMU - it must in order to allow write access to the MPIC (Multi-Processor Interrupt Controller), serial ports and CPU registers. If you then load a kernel which was generated with `INCLUDE_MMU_BASIC` undefined, the kernel will not attempt to do anything with the MMU, so the MMU will remain in the state in which the bootrom left it. Practically speaking this means you can make a kernel for a PPC 750 board which has no cache enabled by undefining `INCLUDE_MMU_BASIC`. However, the boards with PPC 750 and MPC 106 will always have the MMU enabled. Only DBAT (Data Block Address Translation) registers are used to describe MMU address translation (which is 1 to 1) and cache characteristics. Page tables are not used. Since there are only four DBAT registers and the system has to use two of them to run itself, the system is not capable of accessing the whole 4 Gb of memory space in this condition. Turning the cache off also seriously degrades performance. So, it is not recommended to run this type of CPU in this configuration because its performance is significantly restricted.

To make a kernel with MMU and internal cache enabled, define `MMU_BASIC`. If your Gemini board is equipped with L2 cache, then also define the `INCLUDE_L2_CACHE` macro.

Note that with `MMU_BASIC` defined, the Memory mapper is enabled, but it performs only a one to one translation of memory addresses. The MMU is really just used to control caching on a page by page basis (one page = 4096 bytes). (See `cacheDmaMalloc()` in the Wind River Reference manual.) The `L2_CACHE_MODE` is only relevant for PPC 750 CPUs. The 750 supports both `CACHE_WRITETHROUGH` and `CACHE_COPYBACK` modes. The default is copyback because it gives better performance.

NOTE: On certain versions of the PPC 750, the processor chip has a bug which can cause cache failures if L2 cache is in copyback mode and if there is another processor on the same board. So if you have a multi-processor PPC 750 board, you may find it will refuse to let its L2 cache go into copyback mode. It will be in write-through regardless of whether or not you specify `COPY`. This is the software (`sysCache.c`) protecting you from the bug. If the processor is a late enough version to have this bug fixed, then the software will not restrict the configuration.

VME Bus Interfaces

General

Synergy provides two different VMEbus interface mechanisms for the VGM series of boards. With some restrictions they can both be used at the same time. One interface method is via the PCI-VME bridge chip. The other method is the Synergy "Direct VMEbus" interface. The PCI-VME bridge chip is a complete VME interface supporting interrupts and block transfers in addition to the standard A32, A24, A16 supervisor and user data and program accesses. The Synergy Direct interface is less flexible, but faster for non-sequential single read/write operations. It also supports read/modify/write. The VGM1 series of boards uses the IBM Alma PCI-VME bridge. The VGM2, VGM5, VSS4 and VGMD use the Universe II PCI-VME bridge made by Tundra Semiconductor Corp.

The PCI-VME bridge chip can be used to define multiple slave ports. Since each port is limited in the AM (Address Modifier) codes that it will support, you may want to define more than one of them at the same address with different properties.

NOTE: Take care not to define PCI-VME bridge and Direct VME slave ports that overlap.

Before getting into the details of specifying VME master and slave ports in following sections, please look at the default VME port definition. You may not need to change much and so may not need to spend much time figuring out the address translation mechanisms described below. To check your default VME master and slave port definition, get VxWorks to load and run, and then type `sysMasterPortsShow()` and `sysSlavePortsShow()`. The kernel delivered with the BSP should work fine for this purpose.

`sysSlavePortsShow()` will print something like this:

```
VMEadrs          Size       LocalAdrs        (Universe Ports)
0: 20000000      02000000   00000000         prg+dat Usr+Sup A32

Direct slave port at VME adrs= 0xc2000000 is Enabled.
Direct slave port size = 0x02000000

Mailbox VME slave address is at 0x20000080
```

indicating that one PCI-VME bridge slave port is defined. It begins at VME address **0x20000000**. It is 32Mb long and begins onboard at **0**. The direct VME port is responding to **0xC2000000** and is also 32Mb

long. Small changes to macros at the beginning of `synMmuVme.h` will change the location and size of this window. See below.

`sysMasterPortsShow()` will print something like:

```
-> sysMasterPortsShow

    Local        VMEadrs      Size         AM
0:  0x80000000   0x00000000   0x00800000   0x2d
1:  0x81000000   0x00000000   0x01000000   0x3d
2:  0x90000000   0x20000000   0x10000000   0xd
3:  0xa0000000   0x30000000   0x10000000   0xd

Direct Master A32: Local =0x40000000 vmeAdrs =0xc0000000 size=0x10000000
Direct master A16: Local =0x7fff0000 vmeAdrs =0x00000000 size=0x00010000
Direct Master A24: Local =0x7b000000 vmeAdrs =0x00000000 size=0x01000000
```

There are 4 master ports defined by the Universe/Alma PCI-VME bridge chip. The first is A16 space (AM code **0x2d**) and is accessed at **0x80000000** onboard. It translates onboard accesses for **0x80000000** to VMEbus address **0**. It uses 8Mb of onboard address space even though A16 space is only 64Kb long (Alma has a minimum 8Mb window size). An A24 port (AM code **0x3d**) begins at **0x81000000** as seen from onboard and maps onto the bus at **0** also. Starting at **0x90000000** is a VME A32 (AM code **0xd**) window. References to **0x90000000** will appear on the bus at VME address **0x20000000**. The window is 256Mb long. A second 256Mb window into A32 space starts at **0xA0000000**. It maps onto the bus at **0x30000000**. Ports **0** and **1** are controlled by entries in arrays defined in `synMmuVme.h` (see below).

VME Slave Port Addressing through the Alma/Universe

The Synergy BSP follows the Wind River prescription for describing the first VME slave ports. By "first" this means one A32 port, one A16 port and one A24 port. They are defined by the macros:

```
VME_A16_SLV_SIZE VME_A16_SLV_BUS  VME_A16_SLV_LOCAL
VME_A24_SLV_SIZE VME_A24_SLV_BUS  VME_A24_SLV_LOCAL
VME_A32_SLV_SIZE VME_A32_SLV_BUS  VME_A32_SLV_LOCAL
```

which are in the board's `config.h` file. If you are compiling the kernel as a project, then these are found in the "`hardware/busses`" folder in the project facility.

The default condition has the slave A16 and slave A24 ports disabled by having `VME_A16_SLV_SIZE` and `VME_A24_SLV_SIZE = 0`. See notes below before changing these!

These macros behave as defined in the Wind River documentation. See comments in `config.h`, also. Note that CPU number also affects the A32 slave VME address as explained below.

VME Slave Address Mapping Via The PCI-VME Bridge Chip

This is the one VME slave port you will have if you define `DEFAULT_SLAVE_PORT0` to be **TRUE** in `sysMmuVme.h`. There's a separate copy of `synMmuVme.h` for each board directory in the Synergy BSP. For multi-processor boards the X processor directory has the `synMmuVme.h` file. The `Y` (or `Z` or `W`) directories just have a symbolic link to the copy in the `X` directory. This guarantees they all use the same configuration, which is essential.

The VME address of the default VME slave port is determined by the board processor number (which is specified in the Monitor ROM boot parameters) and the `#define SLAVE_PORT_SIZE` and `SLAVE_PORT_SEP` in `synMmuVme.h`. For projects, `SLAVE_PORT_SEP` is also accessible in the "`Synergy BSP Configuration`" folder as the "`VME slave Port Separation`" component. The default slave VME address is computed as shown below:

```
VME slave address = (processorNumber * SLAVE_WINDOW_SEP) + VME_A32_SLV_BUS
```

In this way, if you have several Synergy boards in the same VME chassis, their VME slave ports will be automatically configured to be non- conflicting by merit of their unique CPU numbers. The size of each default slave port is `SLAVE_PORT_SIZE`. The slave ports begin in VME space at `VME_A32_SLV_BUS`. The onboard address that the VME slave address maps into is normally **0**, but you can modify this via the `DEAFULT_SLAVE_PORT0_ONBOARD` macro. You can alter these four parameters as required for your system, but `SLAVE_PORT_SEP` must always be >= `SLAVE_PORT_SIZE` or the boards' slave ports will overlap. The definition of the default port takes place in `sysProcNumSet()` in `sysLib.c` automatically when the system is initializing itself.

Since the PCI-VME bridge chip can support up to 8 slave ports simultaneously, there is also provision for defining other slave ports in addition to or instead of the default port described above.

The extra slave ports are described by an array called `slavePorts[]` which is defined in `sysMmuVme.h`. It has 8 elements. Each element has 5 numbers describing one slave port. `sysProcNumSet()` in `sysLib.c` reads through this list and defines a slave port for each entry which does not specify `slaveSize` = **0**.

If `DEFAULT_SLAVE_PORT0` is not **TRUE**, then initialization of the default port will be skipped. If the `DEAULT_SLAVE_PORT0` is **TRUE**, then only the first 7 of the 8 elements in `slavePorts[]` will be examined by `sysHwInit()`.

Size and Address Restrictions

Please note that there are size and address restrictions on these slave ports. The minimum slave port size is **1Mb** for the Alma, or **64Kb** for the Universe (actually, the Universe allows **4Kb** for ports **0** and **4** and **64Kb** for all other). The slave port address must be a multiple of the slave port size.

Slave ports are normally aligned so that a VME access to their first word will access the onboard RAM at address **0**. For the default slave port, the onboard starting address is defined by the `DEFAULT_SLAVE_PORT0_ONBOARD` macro in `synMmuVme.h`. The `slavePorts[]` array has a parameter for each of the slave ports it describes, which is the starting onboard address. See the comments in `synmmuVme.h`.

IMPORTANT: The command `sysSlavePortsShow()` will print out a listing of the board's current slave VME port configuration.

Changing the VME Base Slave Address

If you want to relocate your board's VME base slave address to some address other than the default of **0x2000_0000**, you simply edit `config.h` to alter the value of (for instance) `VME_A32_SLV_BUS` and recompile the kernel.

IMPORTANT: If you have a multi-processor board, then when you change the value of any slave port, you must also modify and regenerate the bootrom. Because different processors are in different phases of loading simultaneously and there is only 1 Universe chip, it must be initialized in a way that is usable to all processors. For single processor boards you can get away with regenerating only the kernel. The same is true for the direct VME interface described below.

VME Slave Port Addressing through the Direct VME Interface

The direct VME interface is simpler and faster than the PCI-VME bridge - nearly twice as fast for non-sequential single VME bus cycles. It supports only A32 supervisory or user data and program accesses.

There are default initialization parameters for the direct slave port at the beginning of `synMmuVme.h`. They are:

```
#define DIR_SLAVE_PORT_ENABLE  /* undef this to disable direct slave port*/
#define DIR_VME_SLAVE_FWA (0xC0000000 + (sysProcNum * SLAVE_PORT_SEP))
#define DIR_VME_SLAVE_SIZE 0x02000000      /* 16, 32, 64, 128, 256 Mb only! */
```

You can edit these to change the default direct slave port initialization that takes effect when the system starts. To disable the port entirely, `#undef` the `DIR_SLAVE_PORT_ENABLE` macro.

Altering the location on the VME bus of the Direct Slave interface at run time is easy. Simply call `sysDirSlavePortSet()`, specifying the VME address and the port size.

```
sysDirSlavePortSet(0x03000000, 0x01000000)
```

The above call will set the port to start at **0x03000000** on the VMEbus and be **16Mb** in size. Take care not to overlap it with any PCI-VME bridge slave ports!

The only legal sizes for the direct slave port are **16**, **32**, **64**, **128**, and **256Mb**. The starting address of the port must be a multiple of the port size.

A VME access to the first word of the direct slave port will access the onboard RAM at onboard address **0**. This cannot be configured in any other way.

IMPORTANT: The command `sysSlavePortsShow()` will print out a listing of the current slave VME port configuration.

If you have a multi-processor CPU board and are using the direct interface to load over the VME bus (via the `sm` backplane driver), then to change the direct VME slave address you must regenerate both the kernel and the bootrom. This is the same restriction described above for the Universe VMEbus slave address.

VME and PCI Master Port Definition

This BSP follows the Wind River prescription for describing the first VME master ports. Here "first" refers to one A32 port, one A16 port and one A24 port. They are defined by the macros:

```
VME_A16_MSTR_SIZE      VME_A16_MSTR_BUS VME_A16_MSTR_LOCAL
VME_A24_MSTR_SIZE      VME_A24_MSTR_BUS VME_A24_MSTR_LOCAL
VME_A32_MSTR_SIZE      VME_A32_MSTR_BUS VME_A32_MSTR_LOCAL
```

These are defined in the `config.h` file. By default A32 and A24 ports are defined and A16 is not. See comments in `config.h` and also Wind River documentation. These are defined via the PCI-VME bridge chip.

The PCI-VME bridge VME master ports are defined by the contents of two arrays in the `synMmuVme.h` file. The `uMaster[]` array defines VME master ports via the Universe/Alma, while the `dMaster[]` array defines VME master ports via the Direct VME interface.

You edit the contents of these arrays to alter these windows. These arrays also set up the necessary MMU registers/page tables to enable access.

The PCI space is defined by macros in `config.h`. These are also accessible from the project facility in the "`Synergy BSP Configuration`" folder's "`PCI Address Space Configuration`" component. These are shown with their default values below:

```
#define PCI_M_ONBOARD_ADRS      0xc0000000
#define PCI_M_OFFBOARD_ADRS     0x20000000
#define PCI_M_SIZE              0x10000000
#define PCI_MEM_LOCAL_SIZE      0x08000000 /*Subpart of it that's onboard.*/
#define PCI_M_METHOD            DBAT3 /*Must use PAGE if it's > 256Mb */
#define PCI_M_CACHE_MODE        VM_STATE_CACHEABLE_NOT
```

The PCI memory space as seen by the processor starts at `PCI_M_ONBOARD_ADRS`. The total size is `PCI_M_SIZE`. The VGM5, VSS4, and KGM5 boards can support an offboard PCI bus in addition to the usual onboard PCI bus. This area consists of two adjacent windows - a portion that's mapped onto the offboard PCI bus and a portion that addresses onboard PCI memory space. The window to the offboard PCI bus begins at `PCI_M_ONBOARD_ADRS`. The last `PCI_MEM_LOCAL_SIZE` bytes of the total window are onboard PCI memory space. The onboard PCI memory space portion begins at:

```
(PCI_M_ONBOARD +PCI_M_SIZE -PCI_MEM_LOCAL_SIZE)
```

Note that since both areas must be a power of two in size, the only practical division is to split it in half or make the offboard window of **0** length. If you don't need access to the offboard PCI bus (many VME boards do not even have a P0 connector installed), then set `PCI_MEM_LOCAL_SIZE` to the same value as `PCI_M_SIZE`. This leaves **0** space for the offboard window. This may be especially important if you have a PMC module that requests a large amount of space.

NOTE: If you change the offboard window to **0** length, you will need to regenerate both the kernel and the bootrom with these new parameters. Leaving a 50/50 split and just changing the size of the window does not normally require making a new bootrom.

The `PCI_M_METHOD` indicates the method that should be used by the MMU to map this region of memory space. The MMU "method" is described in detail in `synMmuVme.h` and below in the sections on "`uMaster[]`" and "`dMaster[]`". Basically, it is either **PAGE** to indicate mapping via page tables, or **DBAT0**, **DBAT1**, **DBAT2**, or **DBAT3** to indicate one of the Data Block Address Translation registers.

NOTE: If a DBAT register is already used for the Universe master or the Direct master (neither of which are present on the cPCI boards) then it cannot be used again to map the PCI space. DBATs can only be used once. DBATs can map no more than **0x1000_0000** bytes. If you were to increase `PCI_M_SIZE` to **0x2000_0000**, for instance, you would have to switch `PCI_M_METHOD` to **PAGE**.

If you increase `PCI_M_SIZE` beyond a value of **256Mb**, then you must change `PCI_METHOD` to **PAGE**, because a DBAT can't map more than **256Mb** of RAM.

Configuration of the offboard PCI bus (P0 or cPCI) is discussed in `config.h`. Look for macro names beginning with "`PCI_IO`".

The uMaster[ ] Array

Each entry in `uMaster[]` defines one VME bus master port window to the Universe/Alma PCI to VME bridge. You must have exactly 8 entries. A VMEbus window effects an "address translation" of an onboard address (as seen by the PPC) into a VMEbus address (that is, the address that appears on the VMEbus). The Universe/Alma chip responds to onboard accesses to addresses in the range **0x8000_0000** to **0xC000_0000** and translates those into other addresses on the VMEbus.

For instance, a VMEbus board might have a VME slave address of **0x0100_0000** (16Mb) and occupy **16Mb** of VME address space. To address this board, the Universe/Alma could have a 16Mb window defined to start anywhere within the **0x8000_0000** to **0xC000_0000** range and to translate that address to **0x0100_0000** on the VMEbus. For example, a 16Mb window defined at **0x9000_0000** would translate to an onboard access to **0x9000_0000**, then into an access to **0x0100_0000** on the VMEbus.

Since VxWorks is nearly always configured to use the memory mapper, you must, in this example, also describe the 16Mb address space at **0x9000_0000** to the MMU. When the MMU is enabled, no memory anywhere can be accessed unless it has been described to the MMU.

The purpose of the `uMaster[]` array below is to describe the onboard address range of the VME window to the MMU and to specify how you want that address translated into a VMEbus address by the Universe/Alma chip.

Six parameters must be specified for each Universe/Alma VME master window:

1. "onboard address" is the beginning address of the VME window as seen from onboard (that is, the address your program accesses to use this window).

2. "VME Address" is the VME Address into which the onboard address is translated. In other words, a read from the onboard address shows up on the VMEbus as a read from this VME address.

3. "am code" is the VME address modifier code. It is a 6 bit code specifying if the access is supervisor mode, user mode, a 32-bit address, a 16-bit address, etc. Wind River has defined standard names for these in `target/h/vme.h`.

4. "Size" is the size of the VME window in bytes. See rules about size below.

5. "Method" specifies which MMU method you want to use for this block of address space. There are 2 possible methods - DBAT registers, and page tables. DBATs are preferable but have some restrictions, chiefly that there are only 4 DBAT registers in the PPC (see below). You can have any number of spaces described by page tables. Spaces described by page tables must begin on 4kb boundaries and have lengths that are multiples of 4kb. The ratio of address space size to page table size is **128**; a 128Mb address space described by page tables consumes 1Mb of RAM for the table. Initializing these tables takes time when the system starts up. Using a DBAT to describe address space consumes no RAM since it uses no tables. But DBATs can't describe less than **128Kb** or more than **256Mb**. It is, therefore, best to use DBATs to describe very large addresses spaces. Once all 4 DBATs are used, you must use page tables to map any additional blocks of memory.

6. "cache mode" notes whether you want this space cached or not. Normally it is a bad idea to cache VME space so you would specify `VME_STATE_CACHEABLE_NOT`. To cache an area specify `VME_STATE_CACHEABLE` (for copyback cache) or `VME_STATE_CACHEABLE_WRITE_THRU` for write-through cache.

The Rules for defining VME master ports for the PCI-VME bridge Chip are:

1. The "Method" parameter below must be either **DBAT0**, **DBAT1**, **DBAT2**, **DBAT3**, or **PAGE**.
2. DBATs should be used for large regions of address space. A DBAT can't describe less than **128Kb** (**0x00020000**) or more than **256Mb** (**0x10000000**) of address space.
3. The size for a DBAT region must be a power of two.
4. The size for a PAGE region must be a multiple of 4096 bytes.
5. There must be 8 elements in the `uMaster[]` array. Unused elements should be all zeroes.
6. The "onboard" address of a VME window using PAGE method must be between **0x80000000** and **0xbfffffff**. The last word address+1 computed as onboard + size, must be between **0x80000000** and **0xc0000000**.
7. Any entry in `uMaster[]` with **size = 0** is ignored.
8. VME Windows should normally be marked as not cached! Caching offboard RAM space can easily create cache incoherency bugs.
9. VME AM codes (address modifiers) can be found in `target/h/vme.h`. Most commonly used are **0x2d** = A16 supervisor data, **0xd** =A32 supervisor data, and **0x3d** =A24 supervisor data.
10. The "`VME_A32_MSTR_LOCAL`", "`VME_A24_MSTR_BUS`", etc. parameters in the default `uMaster[]` array below are macros defined in `config.h`. They are required by Wind River to be part of the BSP. You may choose to ignore them and replace them with your own values. Wind River only anticipated one window of each type and the board can support up to 8, so there's a mismatch here. Note that the A32 window is expected to be large, so it is defined to use **DBAT0**. The smaller A16 and A24 windows are defined with PAGE method.
11. The Alma bridge chip (on VGM1) requires each window be at least **8Mb** in size.

Below is the default set up for the `uMaster[]` array. This corresponds to the sample printout from `sysMasterPortsShow()` listed earlier in this document.

```
UNIV_VME_MASTER uMaster[8] = {
/* onboard adrs, VME addrs, am code,   size, method,    cacheMode */
{ 0xa0000000, 0x30000000, VME_AM_EXT_DATA, 0x10000000, DBAT1,
VM_STATE_CACHEABLE_NOT},

{ VME_A32_MSTR_LOCAL, VME_A32_MSTR_BUS, VME_AM_EXT_DATA , VME_A32_MSTR_SIZE,
DBAT0,  VM_STATE_CACHEABLE_NOT},

{ VME_A16_MSTR_LOCAL, VME_A16_MSTR_BUS, VME_AM_SUP_SHORT_IO,
VME_A16_MSTR_SIZE, PAGE, VM_STATE_CACHEABLE_NOT},

{ VME_A24_MSTR_LOCAL, VME_A24_MSTR_BUS, VME_AM_STD_SUP_DATA,
VME_A24_MSTR_SIZE, PAGE,  VM_STATE_CACHEABLE_NOT},

{     0,    0,    0,    0,    0,    VM_STATE_CACHEABLE_NOT},
{     0,    0,    0,    0,    0,    VM_STATE_CACHEABLE_NOT},
{     0,    0,    0,    0,    0,    VM_STATE_CACHEABLE_NOT},
{     0,    0,    0,    0,    0,    VM_STATE_CACHEABLE_NOT}
     };  /* Must have exactly 8 entries! */
```

The dMaster[ ] Array

The `dMaster[]` array below is used to describe master VME windows for the direct VME interface. Like the Universe/Alma VME interface, the direct interface performs an address translation that converts an onboard address to a VMEbus address. The onboard address regions that are used must also be described to the MMU. Unlike the Universe/Alma VME windows, the direct VME interface has few programmable parameters. There are just 3 direct VME windows possible. Their addresses as seen from onboard are:

```
1- The A32 window from 0x4000_0000 to 0x7800_0000 (869Mb)
2- The A24 window from 0x7B00_0000 to 0x7BFF_FFFF ( 16Mb)
3- The A16 window from 0x7FFF_0000 to 0x7FFF_FFFF ( 64Kb)
```

Note that the A16 and A24 windows describe the entire VME A16 and A24 space. These windows cannot be disabled, though you can choose not to describe them to the MMU or not to describe their full size to the MMU. The translation to VME address for A16 and A24 space is fixed by the hardware:

```
Access to 0x7B00_0000 translates onto the VMEbus as A24 address 0.
Access to 0x7FFF_0000 translates onto the VMEbus as A16 address 0.
```

This translation cannot be changed. The only thing you can specify is how (or if) you want these windows to be mapped by the MMU. The A16 and A24 spaces are small and so best described by the PAGE method.

The 869 Mb A32 window begins at **0x4000_0000** as seen from onboard. The address translation to VME address converts **0x4000_0000** to `DIR_VME_MASTER_FWA`, which is a macro defined near the beginning of `synMmuVme.h`. There are only 4 possible values for `DIR_VME_MASTER_FWA`. They are:

```
0x0000_0000
0x4000_0000
0x8000_0000
0xC000_0000
```

However, the whole 859Mb space need not all be mapped by the MMU since you probably don't need to access all of it.

For instance, if you have a VME slave at a VME address of **0xA000_0000** you would access it through the direct interface by setting `DIR_VME_MASTER_FWA` to **0x8000_0000**. Then accessing **0x6000_0000** onboard would translate into **0xA000_0000** on the VME. **0x6000_0000** is **0x2000_0000** from the start of the onboard VME window so it gets translated to **0x8000_0000 + 0x2000_0000**, which is VME address **0xA000_0000**. Since you would not be using the first **0x2000_0000** bytes of the onboard VME window, it need not be described to the MMU at all.

Because of all this, the `dMaster[]` array contains no address translation information, only address mapping information for the MMU. The address translation is completely described by your selection of the value for `DIR_VME_MASTER_FWA` (which is in `synMmuVme.h`). (See next section for information on dual processor boards.)

RULES:

1. Any DBAT used above in the `uMaster[]` table cannot be used in the direct table. DBATs can only be used once!
2. There must be 8 entries in the `dMaster[]` array. Any entry with size specified as **0** is ignored.

```
DIR_VME_MASTER dMaster[] = {
/* onboard       size,     method       cacheMode */
{ 0x40000000, 0x10000000, DBAT2, VM_STATE_CACHEABLE_NOT},/* A32 */
{ 0x7FFF0000, 0x00010000, PAGE , VM_STATE_CACHEABLE_NOT},/* A16 */
{ 0x7B000000, 0x01000000, PAGE , VM_STATE_CACHEABLE_NOT},/* A24 */
{         0,          0,      0, VM_STATE_CACHEABLE_NOT},

{         0,          0,      0, VM_STATE_CACHEABLE_NOT},
{         0,          0,      0, VM_STATE_CACHEABLE_NOT},
{         0,          0,      0, VM_STATE_CACHEABLE_NOT},
{         0,          0,      0, VM_STATE_CACHEABLE_NOT}
};
```

Direct VME Master on Dual Processor Boards: sysDirMasterPortSet( )

The 8-bit CPU register at **0xffef_ff18** controls the relationship between the onboard access address and the address that appears on the VMEbus - in other words, the address translation that is done by the direct interface. As explained above, an access onboard to **0x4000_0000** generates a VMEbus access at address **DIR_VME_MASTER_FWA**. (See preceding section for further details.)

On dual processor CPU boards, each processor has its own direct VME address translation control. Initially the X and Y processors both set their Direct Master interface translation to access `DIR_VME_MASTER_FWA`. However, you may want the X and Y processors to have different address translations. Or you may want to dynamically change the address translation of the direct master interface. This can be done by calling `sysDirMasterPortSet`. It takes one argument and that is the new translation value.

So, for example, to change the direct VME master port so that an access to **0x4000_0000** will generate a VME master access to **0x8000_0000**, one would execute:

```
sysDirMasterPortSet(0x80000000)
```

After doing this, executing `sysMasterPortsShow()` will show that the address translation for the direct port has changed.

Universe VME Initialization Defaults

The Universe VME master is initialized to use "`Release on Request`", and "`fair`" modes. Write posting is enabled. In general this gives the most efficient performance for most users. There are many adjustable parameters, however. See the Universe II manual from Tundra Semiconductor Corp. and the Synergy Gemini CPU manual for further details. Virtually all of the initialization is performed in `sysHwInit()` in `sysLib.c` and a few functions called by `sysHwInit()`. Some changes you might want to make are obvious (like changing to `Release When Done`, or disabling write posting) and easy to do. Others are more obscure and require a detailed knowledge of the part and the VMEbus. The header file for the Universe chip is `synergy/universe.h`. The `synMmuVme.h` file in each board directory also contains some configuration parameters, as mentioned above.

sysBusToLocalAdrs and sysLocalToBusAdrs

Providing these two standard BSP routines presents a problem because the Gemini series has two VMEbus interfaces. In many cases an address can be accessed through both of these, so the function is not single valued. The solution is to provide two sets of these conversion routines. The functions `sysBusToLocalAdrsDirect()` and `sysLocalToBusAdrsDirect()` return information valid only for the direct VME interface.

The Wind River standard names `sysBusToLocalAdrs()` and `sysLocalToBusAdrs()` check the VME ports supported through the PCI-VME bridge chip first. If that search is unsuccessful, they then call `sysBusToLocalAdrsDirect()` and `sysLocalToBusAdrsDirect()` to see if the address can be found through the direct port. So, the standard routines give priority to the PCI-VME bridge interface. If you want to check the direct interface only, call `sysBusToLocalAdrsDirect()` and `sysLocalToBusAdrsDirect()` directly.

Because Synergy boards can have VME, offboard PCI and onboard PCI buses, the address translation performed by these functions can get pretty involved. This is simplified by the use of "pseudo AM codes" for PCI bus. See the section below on "`sysBusToLocalAdrsPci() and sysLocalToBusAdrsPci()`".

PCI Bus Interfaces

Defining Offboard PCI Bus Master/Slave Windows

The Offboard PCI bus is present on the Compact PCI CPUs and on those VGM5 VMEbus CPUs that have a P0 connector installed between P1 and P2. It can have master and slave windows defined on it just as the VMEbus can. These are also controlled by arrays in the `synMmuVme.h` file.

The bootrom and kernel use the Wind River auto-configuration software to configure the standard board I/O (ethernet, SCSI, etc.) as well as any PMC boards on the motherboard or on the PEX3 extender card.

If you have a VMEbus CPU with a P0 connector installed between the VME P1 and P2 connectors, that P0 connector offers a connection to an offboard PCI bus (provided for in the VME64 standard). The board connects to that bus via a DEC 21554 non-transparent PCI to PCI bridge. For VME CPUs the PCI auto-configuration software is set to ignore the 21554 bridge. Since in most VME applications the P0 connector is not connected to a PCI bus, it is best not to initialize the chip.

If you have a Compact PCI CPU, then the Compact PCI bus is your offboard PCI bus.

The offboard PCI bus appears to the PPC CPU as a piece of memory. It is described in the VME section above.

Two macros control the probing action of the auto-configuration software over the offboard PCIbus. They are `PCI_GLOBAL_CONFIG` and `PCI_UNIVERSAL_CONFIG`. Their default values are **0**. If `PCI_GLOBAL_CONFIG` is **1**, auto-configure will attempt to discover and configure PCI devices past the onboard non-transparent bridge (that is, PCI devices on the offboard PCI bus). If `PCI_UNIVERSAL_CONFIG` is **1**, auto-configure will attempt to discover and configure every PCI device it can find, traversing every non-transparent bridge it discovers.

Synergy has provided two "Show" words to print out the current state of the offboard PCIbus master and slave windows. They are called:

```
sysMasterPortsShowPci()
```

and:

```
sysSlavePortsShowPci()
```

sysBusToLocalAdrsPci and sysLocalToBusAdrsPci

These functions are provided for the PCI bus and are directly analogous to their VMEbus counterparts. Their arguments are the same. Synergy has defined PCI bus "pseudo AM codes" to facilitate handling PCI bus addresses through the existing `sysBusToLocalAdrs()` and `sysLocalToBusAdrs()` functions.

The PCI pseudo AM codes are defined in `sysBusPci.h` and are:

```
#define PCI_AM_MEM_LOCAL    0x100 /* Local PCI bus memory space */
#define PCI_AM_MEM_SYS      0x101 /* P0-PCI or cPCI bus memory space */
#define PCI_AM_IO_LOCAL     0x110 /* Local PCI bus I/O space */
#define PCI_AM_IO_SYS 0x111 /* P0-PCI or cPCI bus I/O space */
```

Since all VME AM codes are below **0x40**, if `sysBusToLocalAdrs()` and `sysLocalToBusAdrs()` are called with an AM code greater than **0x40**, they will in turn call `sysBusToLocalAdrsPci()` and `sysLocalToBusAdrsPci()` to attempt translation.

Writes to CPU Control Registers (LEDs)

The Grackle chip on the Gemini board performs all memory accesses to onboard RAM and to the "ROM space" - addresses above **0xFF000000**. This includes CPU control registers, LED registers, etc. When a write is performed, the Grackle gives priority to a write to ROM space at the expense of servicing other pending memory requests from onboard DMA, PCI or slave accesses from some VMEbus master. If a program does a large number of consecutive writes to ROM space, it can cause the Grackle to ignore other pending memory access requests for too long a time. In an extreme case, it can cause a bus error to VME master attempting to access onboard RAM. The fix for this problem is to intersperse ROM space reads between the ROM space writes.

Fortunately, there is little need to perform repeated writes to the CPU control registers. Most control items (like direct VME slave address) are just written once at system start-up and not bothered with again. The front panel LEDs are probably the only register to which an application program might well perform many frequent writes. The BSP's `sysCodeLedSet()`, `sysLedOn()` and `sysLedOff()` functions have already been altered to deal with this problem. So, if your program uses these functions, rather than writing directly to the LED control register, then you will have no problems. However, if for some reason you write code that performs frequent writes to CPU control registers, be sure to protect them with a ROM space read just before or after the write. (See `sysCodeLedSet()` in `sysLib.c` for a code example.)

Timers

The VxWorks Tick Timer

The VxWorks tick timer is by default running at **60 Hz**, so calling `taskDelay(60)` will produce a **1** second delay. This frequency is determined by `SYS_CLK_RATE` in `synergy/svgm1.h`. It may be increased for higher time resolution. Be advised that there is overhead associated with each tick of the clock. It is unwise to set it to values larger than **1000** or **2000**. If you need higher resolution, use the timestamp driver (see below). The timer software is in `synergy/synMpicTimer.c`.

The timers are physically located in the MPIC chip. There are 4 of them. The usage is as follows:

Board type	**T0**	**T1**	**T2**	**T3**
single processor	VxWorks Tick	unused	Aux. Timer	unused
dual Processor	X's Tick	Y's Tick	X Aux. Timer	Y Aux. Timer
Quad Processor	X's Tick	Y's Tick	Z's Tick	W's Tick

The basic frequency that the timers divide down is **PCI_CLOCK/8**. This is hardwired on the board and cannot be changed or read by the software. The MPIC timer chips have a "Frequency reporting register" which shows only what the software writes into it. It has no connection to the hardware. After rev. 1.21f, the BSP was changed to write to the MPIC software reporting register at the time the timer is enabled. Prior to this, the contents of this register were non-initialized. Of course, if the software has the base frequency wrong, the frequency reporting register will also be wrong. It only shows the value that the software is assuming for the frequency.

The Decrementer Register and Timestamp Driver

The PPC family supports a decrementer register internal to the microprocessor itself. The decrementer register is used as the timer for the VxWorks time stamp driver. This is a 32 bit down counter. It is loaded with a 32-bit value and counts down at a rate equal to a supplied clock signal. When it underflows, an interrupt is generated and the counter is automatically reset to the initial value.

To configure the timestamp into the kernel, define `INCLUDE_TIMESTAMP` in `config.h` and re-make the kernel. In the project facility it is found in "`hardware/peripherals/clocks/high res. timestamp`" folder. The timestamp commands work as described by Wind River. See the comment headers for functions in `synergy/synPpcDecTimer.c`.

If you are not using Windview, the "rollover" time of the decrementer is set by the value of `decCountVal` in `synergy/synPpcDecTimer.c`. Its default is **0x7fff_ffff**. (See `DECREMENTER_DEFAULT` in `synergy/svgm1.h`.) It is a 32-bit value whose MSB (Most Significant Bit) must be **0**, so **0x7fff_ffff** is the largest allowed value. The decrementer counts this number down at a rate of **BUS_CLOCK/4**, which is **66666666/4 = 16.667MHz**. When it underflows to **-1** an interrupt is generated. Then the counter is automatically reset to `decCountVal` and the process begins again. The decrementer register is read by calling `vxDecGet()` and written to by calling `vxDecSet()`.

If you do not define `INCLUDE_TIMESTAMP`, then the `synPpcDcTimer.c` code is not included in the kernel. The default code that services the decrementer in that case is in `sysLib.c`. The interrupt routine is called `sysDecInt()`. Each time it is entered, it just increments a global variable called `decRegRollovers`. You can read the value of the decrementer register at any time by calling `vxDecGet()`. You can change it by calling `vxDecSet()`. The default value that is loaded into the decrementer is `DECREMENTER_DEFAULT` and is defined in `synergy/svgm1.h`. So the units of `decRegRollovers` is `DECREMENTER_DEFAULT/16666667`. Setting it to **16,666,667** gives a period of **1** second. The value of **0x7fffffff** gives it a rollover period of **128.8** seconds. At any given moment the fractional part of the current rollover period is obtained by computing X - `vxDecGet()`, where `X` is the value that was initially loaded into the decrementer. So, if one sets `DECREMENTER_DEFAULT` to **16666667**, the basic time resolution of the timer is **60** nanoseconds (=1/16666667). In this case, at any instant the elapsed time (in seconds) since the computer was started would be:

```
decRegRollovers + (16666667 - vxDecGet())/16666667
```

If you leave it set to **0x7fffffff**, then the units of `decRegRollovers` is **128.849** seconds and the time (in seconds) at any given moment is:

```
(decRegRollovers + (0x7fffffff - vxDecGet())/0x7fffffff) * 128.849
```

This is a very a very high degree of time resolution, but it's absolute accuracy is only as good as the 66.667 MHz crystal oscillator generating the board's basic clock. That is about one part in 10,000 or 100,000. If you use this as a clock, it will be off by as much as a few seconds per day.

Interrupts

All interrupting devices that may be used in the system need to have an entry in a table in `target/src/drv/intrCtl/synMpic.c` which is named `intVecTable[]`. Normally users need not concern themselves with this table.

This table records the interrupt vector and other information for a device and describes both VMEbus interrupters and onboard interrupting devices. All the descriptions of the onboard interrupting devices are already done for you, but any additional VME interrupting devices may be filled in by the user, or `intConnect()` can do it automatically. Interrupt vectors are one byte numbers, so the largest possible interrupt vector is **255**.

Even if a device is not installed in your system, it is permissable to put a description of it in `intVectable[]`. The information for a particular device in this array is not used unless a call to `intConnect()` is made, so it is harmless to describe any and all devices which may ever end up in your VME system. (See extensive comments in `synMpic.c` which explain the `intVecTable[]` array.)

VME Interrupts

The system makes the assumption that interrupt vectors above `MAX_ONBOARD_VEC` are for VME interrupters. The default value of `MAX_ONBOARD_VEC` is **0x30**. Vectors less than this are for onboard devices (that is, devices run by the system). User interrupt vectors should be chosen to be above this value. If you have a VME interrupter whose vector is less than `MAX_ONBOARD_VEC` and you cannot change it, and there is already an onboard device using this vector number, then you may have to change the vector number of the onboard device to a higher number by editing `intVecTable[]` and `synergyInt.h`. The vector numbers are all arbitrary assignments anyway. All that really matters is that they are all unique. See `synMpic.c` for details.

To enable a VME interrupting device you need to do three things:

1. Pick a vector number larger than `MAX_ONBOARD_VEC`, if possible. If that's not possible, then see comments above and in `synMpic.c`. You may have to modify `intVecTable[]`.
2. Call `intConnect()` specifying the vector and your interrupt service routine's address.
3. Call `intEnable()` specifying the VME interrupt level (IRQ) of the VME device.

See the VxWorks Reference manual for information on `intConnect()`, `intEnable()` and `intDisable()`. When you compile your application make sure it has `#include intLib.h` in it.

`intConnect()` will search through the `intVecTable[]` array to find an entry describing the specified vector number. If it finds one it will make an entry in another table (`sysIntTbl[]`), which is used by the front-end interrupt routine to steer the interrupt to the specified interrupt routine function. If `intConnect()` can't find the vector in the `intVecTable[]`, and the vector is less than `MAX_ONBOARD_VEC`, it will return **ERROR** (which is **-1**). If the vector is above `MAX_ONBOARD_VEC`, it will automatically create an entry for the device in both tables (`intVecTable` and `sysIntTbl`) and the interrupt is then ready to enable.

The VxWorks `intEnable()` and `intDisable()` functions take interrupt level as their argument. Interrupt levels **1** through **7** are understood to refer to VMEbus interrupt levels **1** through **7**. Interrupt levels or priorities (assigned by the MPIC interrupt controller) for onboard devices are actually **0-0xF**, but since 7 of those conflict with the VME levels they are referred to in calls to `intEnable()` as **0x100** through **0x10F**. In other words, there is a bias of **0x100** added to the onboard interrupt levels to allow the software to distinguish those from the VMEbus levels. There are device interrupt level macros defined in `target/config/synergy/synergyInt.h` for all onboard devices (SCSI, serial, ethernet, etc), which you can use if necessary. Since the BSP already takes care of initializing and controlling these devices, you shouldn't need to deal with them. Most users will only need to enable and disable VME interrupts.

Note that `synMpic.c` also contains the functions `sysMpicIntEnable()` and `sysMpicIntDisable()`. These perform the same functions as `intEnable()` and `intDisable()`, except that they take vector number as their input argument instead of interrupt level. These functions are unique to the Synergy Microsystems' BSP.

You can cause your Synergy VME board to generate interrupts on the VMEbus by calling `sysBusIntGen()`. The VME interrupt is generated by the Universe PCI-VME bridge and it has a restriction on the vector number. The vector number is required to be an even number. If `sysBusIntGen()` is called with an odd vector number (on a board using the Universe chip), then it will return **ERROR**. Otherwise it returns **OK**. Also note that for boards with the Universe bridge, if this function is called twice in quick succession, the first VME interrupt may not have been acknowledged yet at the time of the second call. In this case, the function will add on a semaphore until the interrupt is responded to. If the previous interrupt is never acknowledged, this function will hang indefinitely because the `semTake()` call specifies `WAIT_FOREVER`. Depending on your environment, you may want to disable this feature or protect calls to `sysBusIntGen()` with takes and gives of a Mutex semaphore (to prevent priority inversion bugs). In general some sort of synchronization like this is necessary.

The VMEbus SYSFAIL Interrupt

For Gemini boards using the Universe PCI to VME bridge chip (not the VGM1 board) the default SYSFAIL software is set to respond to VME SYSFAIL by generating an interrupt. Functions controlling this operation are defined in `synMpic.c` and are called `sysFailIntEnable` and `sysFailIntDisable`. They take no arguments and, as their names imply, enable and disable the SYSFAIL interrupt.

When a SYSFAIL interrupt occurs, the `universeInt5()` function in `synMpic.c` traps the interrupt. It calls `sysAbortInt()` in `sysLib.c`. `sysAbortInt()` just prints a message (via `logMsg`) and disables further SYSFAIL interrupts. This is because SYSFAIL is often held true by various VME boards during reset, sometimes for long periods of time. If you need to re-enable the SYSFAIL interrupt, do so by calling `sysFailIntEnable()`.

You can modify `universeInt5` or `sysAbortInt` if your environment requires other action in response to a VME SYSFAIL.

Interrupts from PCI Devices

The PCI bus only supports 4 different interrupt lines: **A**, **B**, **C**, and **D**. PMC modules are generally made so that they all assert PCI interrupt **A**, although in general a PMC module could assert any or all of the 4 interrupts. The PCI slots are wired so that each slot connects PCI interrupt **A** to one of the 4 interrupt lines on the PCI bus. If there is more than one PCI slot, then the interrupt connection rotates one pin over for each slot. For example a VGM5 or KGM5 has only one PMC slot so that slot connects the 4 interrupts straight across. The second slot rotates the connection wiring by one pin as shown below:

VGMD PMC slot 0 (The only PMC slot on the VGM2, VGM5, and KGM5)

PMC BOARD	CPU PCI Bus	
A	A	So the 1st PMC slot is wired straight across.
B	B	Interrupt A on the PMC card is A on the PCI bus.
C	C	
D	D	

VGMD PMC slot 1

PMC BOARD	CPU PCI Bus	
A	B	The 2nd PMC slot is wired connects A on the
B	C	PMC card to B on the PCI bus.
C	D	
D	A	

On the PEX3 the wiring is as follows:

PEX3 PMC slot 0

PMC Board	CPU PCI bus	
A	B	So the 1st PMC slot is rotated by 1 wire.
B	C	
C	D	
D	A	

VGMD PMC slot 1

PMC Board	CPU PCI bus
A	C

B	D
C	A
D	B

VGMD PMC slot
2

PMC Board	CPU PCI bus
A	D
B	A
C	B
D	C

So, once a PMC board's slot is discovered (just read the configuration space registers), your software can compute to which interrupt the PMC module is actually connected.

Once you know which PCI interrupt a board is going to use, you can treat it like any other interrupting device. That is, do an `intConnect()` and an `intEnable()` to turn it on. If there is more than one device on the same interrupt, then the interrupt routine will have to sort out which device is interrupting.

The file `synergy/synergyInt.h` defines the vector numbers and interrupt levels for all the devices on the board. The vectors for PCI interrupts are:

```
PCI_A_INT_VEC
PCI_B_INT_VEC
PCI_C_INT_VEC
PCI_D_INT_VEC
```

The levels are:

```
PCI_A_INT_LEVEL
PCI_B_INT_LEVEL
PCI_C_INT_LEVEL
PCI_D_INT_LEVEL
```

So, the sequence:

```
intConnect(PCI_A_INT_VEC, myIsrFcn, 0);
intEnable (PCI_A_INT_LEVEL);
```

would get PCI interrupt A connected to `myIsrFcn()` and enable it.

Multi-Processor Board Interrupts

Interrupts can be individually assigned to either the X or Y processor. When a call to `intConnect()` is made (see `target/src/drv/intrCtl/synMpic.c`) it takes note of which processor is making the call and sets up the MPIC interrupt controller to direct that interrupt to the correct processor.

However, since some interrupt groups are bundled together, there are some complications. The PCI to VME bridge chip groups all the VME interrupts onto a single interrupt. This means that all the VME interrupts must be handled by one processor (the X processor). The same is true for BLT/DMA done interrupts, etc. If Y is required to handle some VME interrupts, it is best to implement them through the Y mailbox. This was done for the BLT/DMA interrupt and may be used as a model. (See the description of the FIFO mailbox below.) The code is in `synMpic.c` in `almaInt()` for boards with Alma PCI-VME bridge, and in `universeInt5()` for models using the Universe bridge.

The method is to have the X processor receive and clear the interrupt, and then have it write a code to the Y processor's mailbox. You must set up the Y processor beforehand (by a call to `sysRegisterMail` as described above) to associate that mailbox interrupt code to a service routine appropriate to the VME interrupt. Such changes are made in `almaInt()` or `universeInt5()` in `synMpic.c`.

Mailboxes on VGM1 (rev. D and later) and all VGM, VSS and KGM Boards

The VGM1 rev. D and later boards (VGM2, VGM5, VSS4) have a simple, very robust mailbox circuit. Any write to byte at onboard address **0x80** causes the data written to memory to also be entered into a hardware FIFO (First In First Out). The non-empty condition of the FIFO causes a mailbox interrupt. On Dual processor versions of the Gemini CPU the mailbox at **0x80** is for processor X and another at **0xA0** is for processor Y (the region in between is unused). Each processor has its own hardware FIFO. The write to the mailbox may come from any source: the onboard processor(s) themselves, a VMEbus master, PCIbus master, etc. Normally one writes byte-wise to the mailbox.

This scheme virtually guarantees that no mailbox writes will be missed. All the masters in the system can attempt writes to the same mailbox simultaneously and they will all get captured and recorded and will eventually be serviced in the order in which they were actually performed. Of course, there is a chance that the FIFO could overflow. The FIFO is 256 bytes long and so can record 255 mailbox interrupts. In practice, experience has shown this gives plenty of capacity for mailbox activity. If the CPU is so busy that it can't keep up with mailbox interrupts, a longer FIFO will only delay failure, not avoid it. If it can keep up, then you only need enough capacity to absorb temporary bursts of activity.

About Mailbox Length

The Mailbox is a single byte. If you do a 32-bit write to the mailbox address, the mailbox circuitry will see the data value in the high 8 bits of that 32-bit word, not the data value in the low order byte. That high byte is usually **0**, so you will get an error message about an "`unregistered mailbox code of 0`". To avoid the problem, just do bytewise writes to the mailbox address. The mailbox ignores the bytes that follow after the mailbox byte. You might plan to do 32-bit writes (or even longer data writes) and use the bytes following after the mailbox for other application-specific information for the mailbox function. However, the byte that lands in the mailbox byte must be the one that has the mailbox code in it.

Mailbox Software Support

The BSP presently provides support for 256 different mailbox routines. (This is determined by `NUM_MAIL_ROUTINES` in `sysLib.c`.) The system presently uses 3 of these. The function `sysRegisterMail()` (in `sysLib.c`) allows the user to define additional mailbox routines. `sysRegisterMail` is called with a data value (0-255), a function address and a constant argument. It uses the data value as an index into an array (`mailRoutine[]`) where this information is stored. When a mailbox interrupt occurs, the mailbox interrupt routine reads the data value from the Mailbox FIFO and uses it as an index into this array. The function (whose address is stored there) is then called with the supplied constant argument. Thus the data byte written to the mailbox selects the function that will be executed. The VxWorks shared memory software already uses the value of **1**. (See `VXMAILSTOP` defined in `config.h`.) On dual processor boards the value **2** is used by the BLT (BLock Transfer) interrupt (see `Y_BLT_MAIL_CODE` in `config.h`). If the value is above **10** you should be safe from conflict with future system enhancements.

The mailbox interrupt is enabled via a call to `sysMailboxEnable()`. It can only be called (that is, it only has an effect) after at least one mailbox routine has been registered via a call to the `sysRegisterMail()`. On dual and quad processor boards, this is done by the operating system when it's starting. On single processor boards which are not using shared memory, no mailbox routines are defined by the system, so the user must call `sysMailboxEnable()` after at least one mailbox routine has been registered.

The user may extend this logic to include a larger number of possible interrupt routines. The front-end mailbox interrupt routine ignores mailbox writes with data values that have not previously been associated with a function via a call to `sysRegisterMail()`. It clears the interrupt, but if no function has been associated with that data value, it does nothing further. There is a global variable called `sysMailCtr` which is incremented upon each mailbox interrupt whether it is caused by a legal data value (i.e., a "registered" data value) being written to the mailbox or not.

To summarize, in order to add a mailbox function, call `sysRegisterMail` to associate your function with a particular mailbox code. (On single processor boards, also do a `sysMailboxEnable()` call after registering the first mailbox routine. Only one call is necessary.) For example, a call to `sysRegisterMail` like this:

```
sysRegisterMail(11, foo, 23)
```

would cause the function `foo()` to be called with an argument of **23** every time an **11** is written to the mailbox byte. See `sysRegisterMail()` in `sysLib.c` for further details.

Making Bootroms and Kernels

Project vs. Non-Project

See the Wind River User's Manual for basic information on projects and workspaces.

This document describes compilation and configuration of the kernel both as a project and as a non-project (the old way of compilation). You may use either method. The only restriction you face when you compile your application as a project and the BSP as a non-project is that you cannot make a bootable application. That is, you can't link your project application in with a non-project kernel to produce a single executable file. You can of course, always load it with the "`ld`" command.

The way one configures the kernel for compilation as a non-project (with a Unix host) is nearly always to edit something in the `config.h` file and then type "`make`" in the board directory. See the section "`Making A New Kernel`" below to learn how to compile a non-project BSP with a PC host. Project configuration, on the other hand, requires finding the appropriate element in the project component tree that is presented to you by the project facility and enabling or disabling it.

If you are compiling the BSP as a project, any configuration feature which is not covered in the project component tree ("`VxWorks`" tab) is still handled the old way. That is, you edit the appropriate .h file and then re-build the project. The vast majority of users should not need to do this, since virtually all parameters are accessible through the project GUI.

All the commonly used parameters are available through the "`VxWorks`" tab in the project facility. There may be a few that were overlooked. Please let us know if you come across one.

If this document refers to defining "`INCLUDE_FOO`" to enable a particular feature, you can find the project component that controls this by pulling down the "`Project`" menu and selecting "`Find Object`". Pull down the "`Type`" menu. You will nearly always be looking for a component "`INCLUDE_`" or a "`Parameter`". After selecting the appropriate type of object, open the "`Object`" menu and scroll through looking for the name. When you find it, select it. Then click the "`Find`" button. It will locate the desired component.

Note also that while the project facility provides Synergy with enough capability to add to VxWorks to build a kernel, Wind River does not provide any project capability to configure the code for a bootrom.

Therefore, the Synergy bootroms cannot be built as projects. They have to be made in the non-project way. See the section on making bootroms below. Most users will never need to make bootroms.

Creating a BSP Project

The directory tree that is installed by the normal install process is for the non-project directory and kernel. If you aren't interested in using the project facility, you do not need to read this section.

For a single processor CPU board (see below for multi-processor boards) or a dual processor board that you are running as a single processor, create a workspace and base it on the appropriate BSP directory (the `svgm2/`, `svgm5/`, `svgmd/`, `kgm5/` or `svcmd/` directory).

If you have a dual processor board and want to run it as a dual processor board, then you have to create one workspace containing the 2 BSP projects - one for each processor. For a VGM5 dual, for instance, you might make a workspace called "`svgm5dual`". Within that would be the `svgm5x project` and the `svgm5y project` and they would be based on the `svgm5x/` and `svgm5y/` directories respectively. For the VSS4, you need 4 projects based on the four VSS4 directories: `svss4x/`, `svss4y/`, `svss4z`, and `svss4w/`.

Once the project (or projects) is created, there is an initialization you have to do for each one. Click on the "`Builds`" tab and then click on each BSP project to open it (you'll see a triangle symbol). Right-click the triangle and select "`properties`" from the pop-up menu.

In the "`macros`" tab of the dialog box that appears, select the `EXTRA_MODULES` macro. For its "`value`" list the whole path to the `synergyLib.a` file. Put it in terms of `$(WIND_BASE)`. Enter it as:

```
$(WIND_BASE)/target/config/synergy/synergyLib.a
```

If you have a single processor board and `RELOCATE_PAGE_TABLES` is defined (the default is that is not defined) then enter a blank after the `synergyLib.a` string and add the following:

```
$(WIND_BASE)/target/config/synergy/mmuPpcLib.oo
```

Click "`add/set`" and then click "`OK`".

If you have a multi-processor board, which you are using as a multi-processor board, then you must add the `mmuPpcLib.oo` path as described above. You must also add the path:

```
$(WIND_BASE)/target/config/synergy/synExcArchLib.oo
```

Next, regardless of what kind of board you have, select the "`C/C++ Compiler`" tab. You must either de-select the "`include debug info`" box or, type in a "**2**" in the "`Optimization Level`" box. If you want optimization level 2 and debug info, it will give you a warning message, but let you do it. If you run this way, the debugger may behave a little strangely because the source text is not so closely related to the actual code. The optimizer changes things around a bit. Do not set the optimization level to "**3**" or to "**none**".

If you have a multi-processor board, which you are using as a multi-processor board, the above steps must be performed for each of the CPU projects.

Next click on the "`VxWorks`" tab and open each project. Within each one check the "`Development Tool Components`" folder. If you want to use target shell, you'll need to enable that and also the "`symbol table download`". You may want to turn on various show routines, as well.

Making A New Bootrom

In general a bootrom cannot be built as a project. Wind River did not provide for configuring the bootrom via the project facility. Some small changes (especially for the multi-processor boards) need to be made to the standard bootrom code in order for it to work on the Synergy board. So, for the moment, the bootroms must be compiled the old, non-project way.

A new bootrom is made (on a Unix host) by typing "`make bootrom.hex`" in the relevant board directory. On a PC host it is done by configuring the Tornado `Build` menu to give you the old method of compilation as an option. To do this select "`options...`" under the "`Tools`" pull-down menu. Then click on the "`Project`" Tab. In that dialog box, check the "`Show Tornado 1.0.1 menus`" option. After that is done, a new entry called "`Standard BSP Builds`" will appear in the "`Build`" menu. Do not be misled by the reference to Tornado 1.0. This will allow you to make Tornado 2.0 compatible bootroms.

Physically, the process is to generate a bootrom.hex file. (Your BSP was delivered with standard bootrom.hex files already compiled for you. They get deleted when you do a "`make clean`", though.) The bootrom.hex file is an S-record file which can either be programmed directly into the boot flash (see "`loadRom.o`" and "`romToFlash`" in the "Using the Boot Flash ROM" section below), or it can be downloaded into an EPROM programmer and then programmed into an EPROM or 32 pin DIP flash device.

In each board directory there is a file called `configBootrom.h`. This is included by `bootConfig.c` and is used to alter configuration of the bootrom. It normally contains a few lines to specify no L2 cache and to disable some common options (like Windview) to conserve ROM space. You may want to edit this to add or subtract some features to/from the bootrom code. Most users don't have any need to alter this file.

Note that the PPC bootroms do not use the first **0x100** bytes of the ROM. The code begins at **0x100** relative to the beginning of the ROM. When a PPC is reset, it sets its program counter to this address. The bootrom.hex file that is generated by "make bootrom.hex" has **0**s in the first **0x100** bytes and the executable code begins at byte **0x100**. The first **0x100** bytes are unused by the system.

At relative address **0x118** from the beginning of the bootrom is a null terminated ASCII string which contains the BSP revision level. This is the same revision information printed when the system starts. It is in the usual format of "VxWorks kernel rev/BSP rev". So for a Tornado 2 system (VxWorks kernel rev 1.2) and Synergy BSP rev "1e", for instance, it would be "`1.2/1e`".

Note for Multi-Processor Boards: The software that manipulates the Flash ROM resides only on the X processor.

Making A New Kernel

NON-PROJECT COMPILATION:

Using a Unix host, generating a new kernel is done in the traditional way by changing directories to the board directory (such as `#cd svgm5/`) and typing "`make`". To make a bootrom you type "`make bootrom.hex`". (In older versions of the BSP there was a macro called `MAKE_BOOT_ROM` that had to be defined in `config.h`. That is no longer needed.)

Using a PC host, you must first configure Tornado to compile a non-project BSP. To do this select `"options..."` under the `"Tools"` pull-down menu. Then click on the `"Project"` Tab. In that dialog box, check the `"Show Tornado 1.0.1 menus"` option. (Do not be misled by the reference to Tornado 1.0. This will allow you to make Tornado 2.0 compatible kernels.) After that is done, a new entry called `"Standard BSP Builds"` will appear in the `"Build"` menu.

To generate a kernel or bootrom, pull down the "`Build`" menu and select `"Standard BSP Builds"`. This allows you to select a Synergy board directory and a target such as `"vxWorks"`, `"vxWorks.st"`, etc. A bootrom is built using the `"Build Bootrom"` menu entry.

PROJECT COMPILATION:

To compile a kernel as a project, you have to make the workspace and project first (See Tornado 2.0 User's Manual). After it is created, open the workspace and click on the "`VxWorks`" tab in the workspace window. Then open the BSP configuration tree. The folder named "`Synergy BSP Configuration`" allows you to configure all the Synergy-specific features. You can also look through the `config.h` file for your CPU. The comments there on many of these parameters are informative.

Open the Synergy folder and look at the various features. You may want to change some. If you have a PU32 PMC module, see the "PU32 Support" section later in this document. The various Wind River components are described in the Wind River documentation.

After you have things set the way you want, just pull down the "`Build`" menu and choose "`Build`" or "`Build All`". The latter does a "`clean`" first and rebuilds everything.

If you have a multi-processor board (and are using all processors), then you will have to do this for each processor. See the "`Creating a BSP Project`" section above.

WARNING: Do not attempt to make a bootrom this way. While it can be made to work for Synergy single processor boards, it does not allow selecting many options and it does not work at all for multi-processor bootroms.

Multi-Processor Board Bootroms

The dual processor board needs to have one bootrom for each processor, but only has one EPROM socket. So, both bootrom images are burned into the same EPROM. Typically a "040" type device is used (27C040 EPROM or 29C040 flash). The X bootrom image is placed at the beginning of the ROM and the Y bootrom image is placed at **0x40000** from the ROM beginning. (This offset can be changed in `romInit.s` - search for "`ycontinue`".)

First, go to the X directory, make any bootrom changes you require in `configBootrom.h`, and do a `make bootrom.hex`. Go to the Y directory and repeat the process. (See the "Making a New Bootrom" section above.)

Once the two bootrom.hex files exist, you can either concatenate them (using the `dualRom.c` program described below), or burn them directly into boot flash using the `loadRom.c` application program.

You should concatenate the two files if you need to burn them into a 32-pin DIP device (like a 27c040). Otherwise you can just copy them into onboard flash, or you can do both. The last option allows the user to keep the EPROM some place safe as a backup.

The program `synergy/dualRom.c` is used to join the two S-record files together. This copies the X processor bootrom.hex to a new file and concatenates the Y processor bootrom.hex onto it, with all the address fields of the S-records adjusted to place the Y bootrom at an offset of **0x40000**. This single S-record file may then be downloaded into your PROM programmer normally. The `dualRom` program would be invoked from your Unix/PC host development system (from the `synergy/` directory for example) as:

```
dualRom ../svgm2_04x/bootrom.hex ../svgm2_04y/bootrom.hex dual.hex
```

To program the bootrom.hex files into the onboard boot flash, use the `loadRom.c` program (`loadRom.o` is also provided to save you the trouble of compiling it). This is the easiest method. It is described below in the following section "Using the Boot Flash ROM". It is also described in the detailed comments at the beginning of the `loadRom.c` file.

For quad processor boards, the process is the same. You must make bootrom.hex files for the Z and W processors. The program `synergy/dualRom.c` will then combine all four bootrom.hex files.

Using the Boot Flash ROM

You can program the boot flash on the CPU either by copying the contents of the socketted bootrom into the boot flash or by downloading a bootrom.hex file over the network and programming it directly into the boot flash.

The Flash ROM software described in this section will not be compiled into the kernel unless the `INCLUDE_FLASH_FILE` macro is defined in `config.h` (which it is by default). If you are compiling the kernel as a project, then make sure that the "`Synergy Flash Primitives`" component is enabled in the "`Synergy BSP Configuration`" folder in the project facility.

The command `sysFlashShow` will print out a report on the types and amounts of flash memory present on the CPU board.

The command "`romToFlash`" (see below) is included in VxWorks to copy the contents of the EPROM/FlashROM device in the 32 pin DIP socket into the boot flash ROM on the motherboard (not to be confused with the "user-flash" flash ROM on the motherboard). The jumper from pin **1** to **2** on **JL01** (see the manual that came with your Synergy board for pin location) should be in place to make the system use the socketted device as the boot device as opposed to using the boot Flash (which is soldered to the motherboard).

To copy an EPROM (or other 32 pin DIP socketted device) into the boot flash, boot a VxWorks kernel that was configured as described above and execute the command `romToFlash`. It will print progress information as it runs. If you are using Tornado, you won't see any of this printout until it is finished, so it will appear busy for 30 seconds or a minute. After it is done, you can remove the jumper and the EPROM and then the board will boot from the boot flash ROM. Note that in general the EPROM socket has to be empty in order for a PMC module to be installed on the board, although some PMC modules may fit with an EPROM installed. This depends on the height of the components installed on the PMC module.

Included with your BSP distribution is a file in the `synergy/` directory called `loadRom.c` (and `loadRom.o`). This is an application program that can be loaded into the target VxWorks with the '`ld`' command, or included in your kernel by adding `loadRom.o` to the `MACH_EXTRA` list in the makefile. It will read a bootrom.hex file over the network and program it into the boot flash. See detailed documentation in `synergy/loadRom.c`. First you generate the bootrom.hex file or files that you need. (Multi-processor CPUs require one "`.hex`" file for each processor.) Then you download `loadRom.o` and invoke "`loadRom()`" with an argument that is the file path to the directory that contains the bootrom.hex file for the processor (or for the X processor on multi-processor boards). An example would be:

```
loadRom("/usr/wind/target/config/svgm5")    /* single processor */
```

Or, for a dual (or quad) processor board:

```
loadRom("/usr/wind/target/config/svgm5x")    /* multi-processor */
```

For multi-processor boards, `loadRom` will automatically find the `Y` (and `Z` and `W`) directory and read the bootrom.hex files in those directories. After this process is completed, you should be able to reboot from the boot flash using the new bootrom code. If you were previously booting from the 32 pin DIP socketted device, you will have to remove the **JL01** pin **1-2** jumper to enable booting from the boot flash.

NOTE: if there is something wrong with the bootrom.hex file which is downloaded into the boot flash (i.e., a configuration mistake), the board may be unable to boot from it. If so, it will then be necessary to have an EPROM that you can boot from in order to get the system going again, and then repeat the process. See the comments at the beginning of `loadRom.c` for further information.

The User flash ROM can be tested by typing `sysFlashTest`. This will erase and write to every byte of User flash (not boot flash) and verify that the correct data is read back.

Using The System Or Expansion Board Flash ROMs

The system flash ROM consists of one, two or four devices (depending on your CPU model) which provide 1, 2, 4, or 8 Mb of "user flash" (or "system flash") memory (not to be confused with the "Boot Flash" ROM, which can contain the VxWorks bootrom code).

A PEX3 PMC expansion board may also contain additional optional flash memory. Options are controlled by three defines in `config.h`. `INCLUDE_FLASH_FILE` causes flash control commands to be included in the kernel. If you also define `INCLUDE_FFS_DRV`, then software supporting a RAM disk in the system flash will be included in VxWorks. If you are compiling the kernel as a project, see the components called "`Synergy Flash File RAM disk`" and "`Synergy Flash Primitives`" in the "`Synergy BSP Configuration`" project folder.

`INCLUDE_PEX3` causes the PEX3 flash and DRAM initialization code to be included. That is defined in `config.h` and, for projects, in "`Include Synergy PEX3`".

Basic high-level system flash control functions (exclusive of using a RAM disk) are declared in `synergy/flash/flash.h` and described in `synergy/flash/flash_hi.c`.

The function `flashDiag()` will perform various read/write tests of flash memory. It takes one argument, a sum of the following values:

```
#define TEST_SOCK_FLASH              0x00000001
#define TEST_BOOT_FLASH              0x00000002
#define TEST_USER_FLASH              0x00000004
#define TEST_PEX3_FLASH              0x00000008
#define TEST_PFSH_FLASH              0x00000010
#define TEST_VERBOSE                 0x00000100
#define COPY_BOOT_FLASH_TO_SOCK      0x00010000
#define COPY_SOCK_FLASH_TO_BOOT      0x00020000
#define SHOW_ALL_VERIFY_ERRORS       0x01000000
```

Many of the flash primitive routines take a flash area (type) number. The following areas have been defined:

```
#define SOCK_FLASH    0x0
#define BOOT_FLASH    0x1
#define USER_FLASH    0x2
#define PEX3_FLASH    0x3
#define PFSH_FLASH    0x4
```

To configure a flash ROM area as a RAM disk, define `INCLUDE_FFS_DRV`, `INCLUDE_FLASH_FILE`, and `INCLUDE_DOSFS`, and optionally `INCLUDE_PEX3`, in `config.h` and do a `make`. This causes `synFfsDrv.c` to be loaded, allowing VxWorks DOS file system facilities to run a file system in the flash devices. After this it is not necessary to use any of the block erase, read or write functions mentioned above. The flash ROM can be read and written to like a local disk drive.

There is an important restriction when using a RAM disk in the flash ROM. The flash can be read like RAM, but writing to it is quite different. Flash ROMs have a limited lifetime for writes. The flash ROMs can only withstand about 100,000 write operations before failing. There is no limit on the number of reads that can be performed. This means you must use the flash RAM disk as a "read many times, write very infrequently" type of device. The most common usage is to write VxWorks and/or your application .o files into the flash RAM disk and then bootstrap load the system from the RAM disk. This is much faster than loading over a network and is essential if the target application does not have a network connection available.

In the following example, an **8 Mbyte** FFS disk is created in the user flash (area **2**) with **16,384** 512-byte blocks, and offset **0** blocks from the start of the flash file device. The device is then initialized for use with `dosFs` and assigned the name "`/ffs/`". This device name is recommended because it is the form the bootrom code recognizes as a bootable flash file device. If you have more or less than **8Mb** of flash ROM, adjust these numbers accordingly. The example is given as if compiled in a program rather than typed in at the keyboard. (NOTE: If you are powering up a CPU that already has a FFS with data files on it, do not do this sequence. See below!)

```
BLK_DEV *pBlkDev;
DOS_VOL_DESC *pVolDesc;

flashWriteEnable(2);
pBlkDev = ffsDevCreate (512,  16384, 0, 2);
pVolDesc = dosFsMkfs ("/ffs/", pBlkDev);

/* Here, write all files to the Flash disk */

ffsFlush();      /* make sure RAM cache gets flushed to Flash ROM!!*/
```

(You can type this from the keyboard. Just use any variable name in place of `pBlkDev` and `pVolDesc` - they don't have to be defined with the right type except when being referenced from within a C program. See example below.) If you forget to write enable the flash before doing the copy operation it will proceed for a while (filling the cache space) and then give you a flash memory error message when it actually tries to write.

The `dosFsMkfs()` routine calls `dosFsDevInit()` with default parameters and initializes the file system on the disk by calling `ioctl()` with the `FIODISKINIT` function. After executing these 3 commands, the `/ffs/` device should appear in the output from the `devs` command.

Note that if the RAM disk was previously defined and loaded (that is, previous to the last time the CPU board was turned off), the `dosFsMkfs()` call will wipe out its previous contents. Note that the third parameter in `ffsDevCreate` (**0** in the example) should agree with the `FFS_BLK_OFFSET` macro in `config.h`.

If the FFS disk memory already contains a earlier disk image that you want to preserve, then call `dosFsDevInit()` instead of `dosfsMkfs`. The formatting parameters given to `dosFsDevinit` (<bytesPerBlk>, <nBlocks>, and <blkOffset>) must be identical to those used when the disk image was created. For example (again, for a board with 8Mb of Flash):

```
pBlkDev = ffsDevCreate (512, 16384, 0, 2);
pVolDesc = dosFsDevInit ("/ffs/", pBlkDev, 0 );
```

(If this two line sequence is compiled in a program, the third argument to `dosFsDevInit` should, technically, be a **NULL** rather than **0**, but from the keyboard you must specify **0**.) The previous contents of the FFS disk will then be preserved (see `synFfsDrv.c`). This implies that if you have created a flash file system and you want the kernel that loads over ethernet (or SCSI, which is not from FFS) to be able to use the FFS device, then each time that kernel downloads via ethernet it must do the `ffsDevCreate()` and the `dosFsDevInit()` calls before it can use the FFS device.

After the `/ffs/` device exists, you can use "`copy`" to copy `VxWorks` and `vxWorks.sym` to the flash RAM disk. You can use the VxWorks `ls` command to verify that the files are on the RAM disk. You must `execute ffsFlush()` to make sure the data (which is buffered in RAM cache by `synFfsDrv.c`) is actually written to the flash ROM. To make VxWorks boot from this RAM disk, you must make a bootrom that contains the flash RAM disk software. This bootrom can then load VxWorks from the flash RAM disk. (To do this just go through the normal "`make bootrom.hex`" operation described earlier in this document, but make sure `INCLUDE_FLASH_FILE` and `INCLUDE_FFS_DRV` are defined in `config.h` - or the project facility - when you perform the `make`.)

If your application program is going to create a RAM disk, write to it and then use it (without resetting the CPU), then it must call `flashWriteProtect()` after doing the last write (that is, the last call to `ffsFlush`), so that no further writes can occur.

IMPORTANT: The `ffsFlush()` function must be called at appropriate times to make sure that cached data is written out to all flash file RAM disks before the system is turned off. Writing to flash ROMs is a slow process. That is why flash RAM disk I/O is cached in RAM (`NUM_CACHE_BLKS` in `synFsDrv.h`). When your program is done writing to flash, do an `ffsFlush()` to make sure the data is written out

before a power failure or system crash causes the cache contents to be lost. Since the number of writes to Flash File chips is limited, do not cause any more writes to occur than are necessary. The flash file chips can be expected to function for about 100,000 writes.

NOTE also that if you are using Tornado Wind Shell, as opposed to target shell, functions like `ls` will behave differently (see section 5.4 of Tornado 1.0 User's Guide or section 6.3.12 of the Tornado 2.0 User's Guide). This is true of all WindSH Primitives. Since the file system is defined on the target, `ls` in the Wind Shell won't refer to it. It will try to access file systems local to the host on which Tornado runs. The solution is to include `ls` in the target kernel and then indicate to the Wind Shell that it should execute the target's version of `ls`, rather than the host version. This is done in 2 steps. First (for a non-project kernel), in the `config.h` define these macros:

```
#define INCLUDE_SHELL  /* To access local file structure via Tornado */
#define INCLUDE_LD
#define INCLUDE_NET_SYM_TABLE
#define INCLUDE_SHOW_ROUTINES
```

(If they are undefined there, change the "`#undef`" to "`#define`". If they are not there at all, add them.) Then re-make the kernel.

For a kernel compiled as a project, these are found in the Wind River "`Development Tool Components`" folder of the project facility.

Then in the Wind Shell, to execute the target version of `ls`, prefix an `@` symbol on the command name, as in:

```
@ls "/ffs/",1
```

The printout from this `ls` will go to the board's serial port rather than to the Wind Shell. To divert the target's print out to the wind shell use the `ioGlobalStdSet()` function. An example of how to do this is given in `target/synergy/divert`, and you can load this file as a script to effect the diversion of the print out.

For kernels compiled as a project, see the Wind River "`Development`" folder in the Project facility GUI.

To summarize, the full sequence of events to (manually) create a Flash RAM disk, copy VxWorks onto it and then boot VxWorks from it, is shown below. This is written assuming that you are using target shell rather than Tornado. (Compiling for target shell is easy - just `#define INCLUDE_CONFIGURATION_5_2` in `config.h`. Or, for projects, go into "`target Shell Components`" and enable what you need.)

1. Edit the `config.h` so that `INCLUDE_FFS_DRV` and `INCLUDE_DOSFS` are defined, and do a `make`. (For Projects see "`Synergy Flash File RAM disk`" component.)
2. Download the new kernel over ethernet and (if you have 8Mb of flash) type:

```
-> flashWriteEnable 2
value = 0 = 0x0
-> pb = ffsDevCreate(512,16384,0,2)
new symbol "pb" added to symbol table.
pb = 0x1ffe730: value = 33548096 = 0x1ffe740 = pb + 0x10
-> pv = dosFsMkfs("/ffs/",pb)
```

   If you execute the `devs` command now you should see `/ffs/` in the list.
3. Type a `cd` command to change your NFS directory to your board directory. This would be something like:

```
cd "/usr/wind/target/config/svgm1_04d/"
```

4. Copy the file from you host system to RAM disk, by typing:

```
copy "vxWorks", "/ffs/vxWorks"
copy "vxWorks.sym", "/ffs/vxWorks.sym"
```

   Doing an `ls "/ffs/"` should show you the two files on the RAM disk.

```
-> ls "/ffs/",1
size        date             time        name
870132      JAN-01-1980      00:00:00    VXWORKS
121922      JAN-01-1980      00:00:00    VXWORKS.SYM
```

5. Make sure the cache buffers get written to Flash by typing:

```
ffsFlush
```

6. Return to your development system and type:

```
make bootrom.hex
```

7. Burn the new bootrom.hex into an EPROM (see section above entitled "Making a New Bootrom") and install the EPROM on the board and install the EPROM jumper (**JL01** pin **1-2**). Or download the .hex file into the boot flash using the `loadRom.o` utility.
8. Turn on the board and set the boot parameters as shown below:

```
boot device             : ffs
processor number        : 0
host name               : host
file name               : /ffs/vxWorks
inet on ethernet (e)    :
inet on backplane(b)    :
host inet (h)           :
user (u)                : your_user_name
flags (f)               : 0x0
target name (tn)        : targetName
Other (o)               :
```

9. Type `@` to start the system loading and you should see it come up quickly from the RAM disk.

If you want to also have the ethernet interface present, list `esyf` as the "other" and fill in the `host inet` and `inet on ethernet` as appropriate for your system.

Note for Multi-Processor Boards

There is a complication when doing this process on a dual processor board. First, the Y processor should be configured to load using `sm` as usual, but it should specify **FTP** as the load method. Secondly, if target shell is being used (as opposed to Tornado), it must load `vxWorks.st` rather than vxWorks. Alternately, an FTP patch must be applied to `usrConfig.c`. The problem is that the Y processor kernel is otherwise unable to get the FTP server on X to switch clients from the bootrom to the newly loaded kernel. A more permanent fix is going to be worked out shortly.

The boot parameters for the other processor(s) on the board should be set up as follows (assuming the Y processor kernel was called `vxWorksy.st`).

```
boot device             : sm=0x4100
processor number        : 1
host name               : host
file name               : /ffs/vxWorksy.st
inet on ethernet (e)    :
inet on backplane(b)    : 209.0.0.2
host inet (h)           : 209.0.0.1
user (u)                : your_user_name
flags (f)               : 0x0
target name (tn)        : targetName
```

In the above example it was assumed that the backplane subnet was **209.0.0.0**, the X processor is **209.0.0.1** and the Y processor is **209.0.0.2**. The subnet IP numbers will be different in your case.

NVRAM (Non-Volatile RAM)

General

The Gemini series NVRAM is powered by a snap-on battery pack with an expected lifetime of several years. Either 32Kb or 128Kb of NVRAM may be installed on the board.

NVRAM begins at address **0xFFE98000** for boards with **32Kb** (**0x2000**) NVRAM and it begins at **0xFFE80000** for boards with **128Kb** (**0x8000**) installed. The VxWorks BSP (for reasons of compatibility with earlier versions) behaves as though the NVRAM begins at **0xFFE9E000** (defined as `S_BBRAM` in `svgm1.h`) regardless of how much is installed. This is **8Kb** from the end of the NVRAM space. The BSP uses only a small part of the NVRAM space. The region from **S_BBRAM + 0x500** through to **S_BBRAM + 0x800** is used. You may alter the region used by the BSP by changing `S_BBRAM` in `svgm1.h`. However, this will require regenerating both the bootrom and the kernel. When first powering up the board you will have to reset the ethernet address to its new location.

The BSP's detailed used of NVRAM referenced to the base address of **0xffe9e000** is shown below:

Offset	**Description**
0 - 02ff	Unused.
0x300 - 0x3ff	W processor boot parameters (on quad boards). Unused on single and dual processor boards.
0x400 - 0x4ff	Z processor boot parameters (on quad boards). Unused on single and dual processor boards.
0x500 - 0x5ff	Y processor boot parameters (on dual boards). Unused on single processor boards.
0x600 - 0x6ff	A copy of the boot parameters (on dual boards this is the X processor's boot parameters).
0x700 - 0x76b	Unused.
0x76C - 0x76F	Size of L2 cache (for 750 and 7400 processors only).
0x770 - 0x773	The memory size. This is filled in by the bootrom when the board is reset.
0x774 - 0x777	Ethernet address for Y processor.
0x778 - 0x77b	Ethernet address for board (or for X processor on dual boards). This is the serial number entered with the "m" command in byte-wise mode, two digits per byte. SN of 845622 is entered as 3 bytes: 88, 56, 22. They actually are taken as hex, but that doesn't matter (they're still unique and

	that's all that counts). The leading 3 digits of the ethernet address that identify synergy are compiled into the ethernet code as a macro and so, aren't stored here.
0x077c - 0x77c	MPIC init counter. 0 Means the once-only part of the MPIC done by sysMpicInit() hasn't been done yet. (1 byte)
0x077d - 0x77f	Unused.
0x0780 - 0x783	Flag used for Smon booting VxWorks on dual board.
0x0784 - 0x1fef	Unused.
0x1ff0 - 0x1fff	Clock/calendar registers.

The NVRAM can be read as 8, 16 or 32 bit data, but it can only be written bytewise. So, when using the `m` command to alter memory, always specify bytewise access. When writing data to NVRAM use `bcopyBytes()` or a byte-wise pointer, etc. For reads, it can be accessed like any other memory.

See `synNvRam.c` for `sysNvRamGet()` and `sysNvRamSet()` functions which transfer data to and from NVRAM. These access NVRAM relative to the boot line, however, so are primarily used by the BSP to change and display the boot parameters.

Clock/Calendar

The calendar/clock registers are accessed in the **last 16 bytes** of the NVRAM space. Since the larger NVRAM begins at a smaller address the end of the NVRAM is always at the same address. See `synCalLib.c` for functions to set the clock/calendar time and date, to read the time and date, and print out the time and date.

SCSI Support

General

The Gemini board's Symbios 53C885 chip provides both Ethernet and SCSI I/O. This part is a descendant of the NCR 53Cxxx series SCSI chips. Support for wide and narrow SCSI-II is provided. SCSI-I is not supported. To add SCSI capability to your system, find `INCLUDE_SCSI` in `config.h` and define it (its default condition is undefined). Set the `SCSI_HARDDISK_ID` to be equal to the SCSI ID of your disk. Read the comments in the SCSI section of the `config.h` file.

Using a local file system from a Tornado host is a bit more complicated than using it from the target shell. See Wind River's "Tornado User's Guide" Chapter 7 section on "C-Primitives". See the Wind River "Programmer's Guide" Chapter 4 for more details of using SCSI in VxWorks.

To copy a VxWorks kernel out to disk and have the system load from disk, do the following:

1. Enable SCSI options in `config.h` (including `SCSI_AUTO_CONFIG`) and do a `make`. Or, for kernels compiled as projects, enable the "`Synergy SCSI initialization`" component in the "`Synergy BSP Configuration`" folder. You should (if you are using SCSI2) define the `INCLUDE_SCSI2` parameter there as `TRUE`. The other parameters are set at their nominal defaults and probably don't need to change.

2. Load that kernel via ethernet. After the kernel loads, initialize a DOS file system on the SCSI disk.
   To make a new file system type:
   ```
   scsiShow
   ```
   This will print the address of `pScsiPhysDev`. Next type:
   ```
   pb = scsiBlkDevCreate(0xPPPPPPPP,0,0)
   ```
   where the **`0xPPPPPPPP`** is the value for `pScsiPhysDev` that was printed by `scsiShow` above. Next, to create a new file structure (destroying any earlier one!), type:
   ```
   dosFsMkfs("/sd0/",pb,0)
   ```
   This will initialize the file structure and create the `/sd0/` device which you should now see when you type `devs`.
   If there is already a DOS file system on the disk that you wish to use, then instead of the `dosFsMkfs()` call above, type:
   ```
   dosFsDevInit("/sd0/",pb,0)
   ```

3. If you don't want to boot VxWorks off the SCSI disk, you can stop at this point. You can now access and create files on the disk, read, write, etc. If you want to copy VxWorks (or `vxWorks.st`, or `vxWorks` and `vxWorks.sym`) from your host onto the SCSI disk, then proceed. This presumes you have re-generated the kernel with SCSI options defined. The copy is done with the `copy` command. (The `copy` command may not be present in Tornado. You may have to use target shell to do this, or write a simple application, etc.) To use `copy` (from the target shell), for example, assuming the path to VxWorks on your host is `/usr/wind/`, type:

   ```
   copy "/usr/wind/target/config/svgm1_04d/vxWorks","/sd0/vxworks"
   ```

   Copy any other files you need in the same manner (such as `vxWorks.sym`). The `ls` command can be used to see that the files are on the `/sd0` device. Type:

   ```
   ls "/sd0/"
   ```

   See '`ls`' in the Wind River Tornado reference manual. Note that if you are using Tornado, `ls` will be understood by Tornado to refer to the file system on your host, not on the target. Preface the command with an `@` sign to force it to use the version on the target system. See Wind River Tornado User's Guide Chapter 7.

   NOTE: If you have done this initialization process once, and if you have `SCSI_AUTO_CONFIG` defined in `config.h`, then the next time you load the kernel it will automatically find the disk and initialize the file structure. If you type `devs` right after the kernel loads, you should see the `/sd0/` device already listed.

4. Generate a bootrom in the fashion described in the bootrom section above. Make sure that the SCSI parameters in `config.h` are defined, including `INCLUDE_SCSI_BOOT` in `configBootrom.h`.

5. Install the bootrom EPROM/Flash ROM in the CPU card. If you specify `boot device = esyf`, this bootrom will still allow you to boot over ethernet (as long as you didn't change the defaults in your `config.h`). To boot from SCSI disk, one would use boot parameters something like these:

   ```
   boot device             : scsi=4,0
   processor number        : 0
   host name               : yourHostName
   file name               : /sd0/vxWorks
   inet on ethernet (e)    : 123.123.123.123
   host inet (h)           : 123.123.123.124
   user (u)                : yourUserName
   flags (f)               : 0x0
   target name (tn)        : fooBaz
   ```

   The **4** in the "**scsi=4,0**" is the SCSI ID of the disk. Substitute the correct ID for your disk drive. The **0** is the logical unit number. Fill in IP addresses appropriate for your network. (See `SCSI_HARDDISK_ID` in `config.h`.)

6. Push reset and the board should load from the disk. This is normally much faster than loading over the network.
   NOTE: There is a general problem accessing local file systems on a target from the Tornado Wind Shell. Functions like `ls` and `ld` are defined on the Tornado host and will refer to file systems on that host rather than on the target. This is true of all WindSH Primitives (see section 5.4 of the Wind River "Tornado User's Guide" for more information). To get around this, you need to cause `ls` to be included in the target kernel. You must then indicate to the Wind Shell that it should execute the target's version of `ls` rather than the host version. This is done in 2 steps. First in the `config.h` define these macros:

   ```
   #define INCLUDE_SHELL /* To access local file structure */
   #define INCLUDE_LD
   #define INCLUDE_NET_SYM_TABLE
   #define INCLUDE_SHOW_ROUTINES
   ```

   (For kernels compiled as projects, these parameters are accessed in the "`development tools components`" folder.)

   Then in the wind shell, to execute the target version of `ls` one would prefix an `@` symbol on the command name, as in:

   ```
   @ls "/sd0/",1
   ```

The printout from this `ls` will go to the board's serial port rather than to the wind shell. To divert the target's print out to the wind shell use the `ioGlobalStdSet()` function. An example of how to do this is provided in `target/synergy/divert`, and you can load this file as a script to effect the diversion of the print out.

Multi-Processor Boot from Local SCSI

Since there are no Unix-type remote shell commands supported by the VxWorks kernel on the X processor, the Y processor must use FTP to load its kernel from a local SCSI disk. This means that the X processor kernel must be configured to have its FTP server task running. This is done by defining `INCLUDE_FTP_SERVER` in the X processor `config.h` file at some point after the "`#include configAll.h`" statement. For kernels compiled as projects, find the "`network components/networking protocols/network filesystems`" folder. Or, just search for "`INCLUDE_FTP_SERVER`."

The boot parameters for the Y processor would then be something like this (assuming the dual board processors are **0** and **1**, respectively):

```
boot device             : sm=0x4100
processor number        : 1
host name               : yourHostName
file name               : /sd0/vxWorksy
inet on ethernet (e)    :
inet on backplane(b)    : 209.0.0.2
host inet (h)           : 209.0.0.1
gateway inet (g)        :
user (u)                : yourUserName
ftp password            : foo
flags (f)               : 0x0
target name (tn)        : fooBaz
```

This example assumes the VxWorks kernel for the Y processor is called `vxWorksy` (and that the `.sym` file is `vsWorksy.sym`). The user name and ftp password can be anything, but the ftp password must be specified.

Some versions of VxWorks may have a delay in loading the Y processor. After the kernel is loaded, the FTP session started by the bootrom isn't terminated properly which causes the next session started by the newly loaded Y-kernel to have to wait for a time out period of a minute or so. We are presently working on a fix for this.

Built-in Self Test Capabilities

General

It is possible (prior to starting VxWorks) to use the Synergy SMon monitor ROM to start the Synergy CPU and perform a script that runs various tests. The VxWorks bootrom in this case resides in the user flash rather than in its normal location in the bootrom space at **0xfff0_0000**. Only BSPs rev. 1.2/1g or later (for Tornado 2.0) and 1./01q or later (for Tornado 1.0) support this feature.

The process described below is for a single processor board or for the X processor on a dual processor board. The dual processor is addressed later in this section. If you have a dual processor but intend to use it as a single processor, then just follow the single processor instructions.

You will need to purchase SMon from Synergy Microsystems and use it in the EPROM socket or in the boot flash. The CPU board must also have the user flash option installed. You then have to generate a copy of the VxWorks bootrom compiled to execute at **0xfff8_0100**. This is done by altering the makefile so that `ROM_TEXT_ADRS` is changed from its normal value of **0xfff0_0100** to **0xfff8_0100**. It is convenient to have both versions of `ROM_TEXT_ADRS` defined in the makefile and to just comment out (by placing "`/*`" in front and "`*/`" at the end) one of them at any given time.

Beware that it is an easily made mistake to forget the state of `ROM_TEXT_ADRS` and generate a bootrom with `ROM_TEXT_ADRS` set to **0xFFF8_0100** (for example) and burn that into an EPROM (or vice versa with normal EPROM resident code placed into the user-flash). When the CPU is started with such an EPROM it will not run at all.

When changing the address like this, it is necessary to delete any .o files, which might be present in the directory, that were compiled at the other address, before you generate the new bootrom image. So, delete `romInit.o`, `bootInit.o`, `bootConfig.o` and `bootrom.Z.o` before you do the `make bootrom`. You can do a `make clean` instead. Immediately after that, however, you must perform a `make` (to generate the dependencies file).

Typing `make bootrom` generates the bootrom code.

It is not necessary to make a bootrom.hex. The bootrom file produced is converted to an absolute memory image by typing:

```
powerpc-eabi-objcopy -O binary bootrom raw
```

This produces a file called `raw` (a raw image of the bootrom) that needs to be programmed into the user flash. You must then place the `raw` file into the `/tftpboot/` directory on the host used by SMon.

To program the raw image into user flash, start SMon and then type:

```
load "raw"
```

This will read the raw file into memory at **0x10000**. (Note that for a dual processor board, in which you intend to use both processors, the `load` commands are a bit different. See Below.) Next type:

```
flashwren 1
flashcopy 0 10000 40000 2
```

This programs the raw image into user flash.

When SMon starts, you can cause it to jump to VxWorks by typing:

```
flashadr 0 2
g fff80108
```

You can automate this process by putting these 2 commands in the SMon startup script. In that case it would be:

```
flashadr 0 2;    or    flashadr(0,2);
g fff80108;      or    g(fff80108);
```

You can also put self-test commands in the startup script before the jump to VxWorks.

Multi-Processor Boards

If you have a dual board and you intend to run both processors, then the process is a bit different. First you must edit the makefile in the Y processor directory so it says `ROM_TEXT_ADRS` is **`0xFFFC_0100`** instead of the normal **`0xFFF4_0100`**. Then, as with the X processor described above, remove the old .o files from the Y board directory, and do a `make bootrom`. You then use the `powerpc-eabi-objcopy` command as shown above to generate the Y processor raw image and copy it into the `/tftpboot/` directory. Since both X and Y raw images have to reside in this directory, they need different names, so we'll call one "`rawx`" and the other "`rawy`".

To program the 2 images into the user flash, start SMon and then type:

```
load "rawx"
load "rawy" 50000
```

This command copies both files into RAM in the same relationship to each other that they need to have in the flash. Then program both bootroms into flash by typing:

```
flashwren 1
flashcopy 0 10000 80000 2
```

The VxWorks bootroms are started the same way as described above, that is, by starting SMon and then typing:

```
flashadr 0 2
g fff80108
```

After the bootrom gets started it behaves exactly as though the board had started initially with VxWorks (instead of SMon). The downloaded kernel has no sensitivity to the altered location of the bootrom code except if you reference `ROM_REV_STR_ADRS` (the system BSP revision string stored in the bootrom). If you look at `ROM_REV_STR_ADRS` in `svgm1.h` you will see that it is defined in terms of `ROM_TEXT_ADRS`. If the kernel was compiled after the makefile was altered, then it will be correct. If it is an older kernel, it will reference the wrong address. If you intend to use `ROM_REV_STR_ADRS`, then your should recompile your kernel.

PU32 Support

The PU32 is a PMC module manufactured by Synergy Microsystems. It has up to 32 serial channels implemented with Cirrus CD2401 serial chips. The previous driver (from Tornado 1) was not able to be auto-configured. That driver has been updated a bit (as of Sept. 18, 2000) for Tornado 2 so that the PU32 can reside in the PMC slot on the motherboard or on a PEX3 carrier. The driver will find it and configure itself accordingly. At the present time, the driver can still handle only one PU32 module.

To configure the PU32 driver into the kernel:

NON-PROJECT: Find `INCLUDE_PU32` in the `config.h` file and change it from `#undef` to `#define`. Then remake the kernel.

PROJECT: In the Workspace window, select the "`VxWorks`" tab and then open the "`Synergy BSP Config.`" folder. Use the Project menu to include the element called "`Initialize PU32 channels`". Then right-click this component and check the `NUM_TTY_A` parameter that it has. `NUM_TTY_A` should be set to the number of serial ports on your PU32. There are 4 per cd2401 chip. Do not specify more channels than there actually are!

Note that if you make a kernel (or bootrom) with `INCLUDE_PU32` defined and there is no PU32 installed on the board, the system will hang when it is initializing the serial devices.

When that kernel loads successfully, and if there is a PU32 connected, typing the `devs` command should show you a "`/tyCo/n`" device for each serial channel on the board.

If you have a single processor CPU, then the first 2 serial ports, `/tyCo/0` and `/tyCo/1` are the front panel serial ports. In this case `/tyCo/2` is the first serial port on the PU32, `/tyCo/3` is the second, and so on.

If you have a multi-processor board, then the front panel serial port for each processor is `/tyCo/0`. The first serial port on the PU32 is then `/tyCo/1`. Note that on a multi-processor board, only one processor can have `INCLUDE_PU32` defined at a time.

Once the serial devices are being displayed by the `devs` command, you can start to use them like any other VxWorks serial port. Use an `open` command to open the device and then `read` and `write` commands to send and transmit data. For example:

```
pu0 = open("/tyCo/2",0,0)
write(pu0, "Hello\r\n", 7)
```

will open the first PU32 port and print the word "**Hello**" on the port.

If your PU32-configured kernel hangs with a **0x3** in the LEDs and you compiled the kernel as a project, then the value of `NUM_TTY_A` is wrong. In this case the value of `NUM_TTY_A` must be no greater than the number of physical serial ports actually installed on the board. See the "`Initialize PU32 Channels`" component in the "`Synergy BSP Configuration`" folder. Alter the `NUM_TTY_A` parameter and rebuild the kernel.

Toubleshooting

VxWorks LED Diagnostic Information

VxWorks is an operating system which depends on everything working normally. If some feature is not working, VxWorks is pretty poor at giving any indication other than the point in the load process at which it dies. Wind River does not want VxWorks to output huge amounts of status or configuration information while it downloads.

The 8 user LEDs offer some general clue as to where in the process the problem was encountered. They display a 4 bit count which starts at **0xC** and counts down to **0**. On dual processor boards the X processor uses the low order 4 LEDs (**3 – 0**) and the Y processor uses the high order 4 LEDs (**7 – 4**).

On quad processor boards, there is no reasonable way to share 8 LEDs among 4 processors. The 4 processors all write to the LEDs with no synchronization, so the content of the LEDs isn't very informative.

The start-up process follows these steps:

1. Power-on

- All LEDs are turned on. The processor jumps to the bootrom code at **0xfff0_0000**. The software sets up the `MSR`, `HID0` and `HID1` registers. For PPC 750 and PPC 7400 boards it also must enable the MMU and uses the DBAT registers to map memory.
- The Grackle is initialized.

2. Jumps to `romStart()`

- The very first thing this does is display a **0xC** in the LEDs. The software is still executing out of EPROM/Flash. If the processor is a G4 and the memory card is capable of doing ECC, then the ECC bits throughout memory are initialized. To do this the data cache is briefly enabled and each 32-bit word is read and written. LED **4** blinks rapidly while this is taking place. Then data cache is disabled. On multi-processor boards, only the X processor does the ECC initialization. The other processors are held off and their LEDs will be read at this time.
- L2 cache is not used at all in the bootrom code.
- `romStart()` then reads the compressed VxWorks bootrom software out of a data array in the ROM, decompresses it and copies it into low RAM.

3. Jumps to `usrInit()`

- It jumps to the decompressed bootrom code in low RAM. The very first thing `usrInit` does is write a **0xB** to the LEDs. It then initializes the PCI devices. The Universe chip is set up. Grackle ECC reporting is enabled if the processor and memory can support it. The VxWorks bootrom kernel is started.

4. Jumps to `usrRoot()`

- VxWorks starts the root task. The first thing it does is output a **0xA** to the LEDs.
- Interrupts are enabled and the VxWorks tick timer starts.
- The processor initializes its serial port (or ports).
- The command loop function is started.

5. Jumps to `bootCmdLoop()`

- `booCmdLoop()` outputs a **0x9** to the LEDs. It then prints the bootrom sign-on information. An example would be:

```
VxWorks System Boot
Copyright 1984-1998  Wind River Systems, Inc.
CPU: Synergy vgm5-750 dual
Version: 5.4
BSP version: 1.2/1d
for multiprocessor using Grackle memory map B
Creation date: May  5 2000, 14:34:02

Ethernet: 91 92 45
Shared mem. uses bus locking.
Configured for FIFO mailbox.
PCI autoConfig. code is in bootrom.
SDRAM memory card installed.

Press any key to stop auto-boot...
```

- Then it starts it's 8 second countdown. If no key is struck, it starts loading automatically.

6a. (X processor) Jumps to `bootLoad()`

- It outputs an **0x8** to the LEDs. It prints out the boot parameters which are used during the load. These parameters look something like the following:

```
boot device           : esyf
unit number           : 0
processor number      : 0
host name             : sluggo
file name             : /Soft7/Vxtemp/target/config1.21d/svgm5/vxWorks
inet on ethernet (e)  : 197.42.42.235
inet on backplane(b)  : 209.0.0.1
host inet (h)         : 197.42.42.29
```

```
user (u)              : vw
flags (f)             : 0x0
target name (tn)      : fooBaz
```

- It normally is setup to load via a network, so it now does a lot of network initialization. This may involve the ethernet, backplane driver, Flash ROM RAM disk, or SCSI.
- Then, if it is the X processor, it prints out:

```
Attaching network interface esyf0... done.
Subnet Mask: 0xffffff00
Attaching network interface lo0... done.
```

6b. (For non-X processor) Jumps to `bootLoad()`

- If you are not concerned with the Y Processor, skip to 7- below. At this point Y displays an **0x8** in the LEDs and then prints the boot parameters just as the X processor does. However then, if it is the Y (or Z or W) processor, it prints:

```
Waiting for valid master VME window list...
```

- It's waiting for the X processor's kernel to initialize the Universe chip. The X processor bootrom initialized it once, and after the X kernel starts up, it initializes it again. The VME windows that are defined by the kernel might not be the same as those defined by the bootrom, so the Y processor has to wait for X to signal that it is done accessing the Universe for the last time.
- Next it prints:

```
Backplane anchor at 0x4100... Attaching network interface sm0... done.
```

- It rarely hangs up here, but if it does, it may indicate a problem with the VMEbus. This is because involved in this "sm" process is a use of the Universe chip as an arbitration device to briefly get ownership of the VMEbus. If that attempt to get ownership of the VMEbus fails, Y can hang here indefinitely. If the board in question is a cPCI board, it does a similar thing with the 21554 bridge on the compact PCIbus.
- Next it prints:

```
Attaching network interface lo0... done.
```

7. Jumps to `netLoad()`

- Outputs a **0x7** to LEDs. It then starts reading the VxWorks kernel from what ever device was specified (ethernet, SCSI, etc.). At this point it prints:

```
Loading... 891428
```

- If a processor hangs at this point, it indicates some problem in reading the kernel image. This could be an ethernet problem, if ethernet is being used. If a non-X processor hangs at this point, it indicates that its communication with the X processor is fouled up or that the Unix host that supplies the kernel image file is not responding for some reason. The IP addresses could be wrong. For multi-processor boards, there is an additional pitfall - the Unix host must have a network route defined that identifies the X processor's IP address as the gateway for the `sm` (Shared Memory) backplane subnet. Typing `netstat -nr` on a Unix terminal will show if the network routing is OK. It rarely produces errors.

Usually the only way for that to happen is if the host computer (Skyler) was rebooted and some one forgot to redefine the route. Normally once it gets set up right, it stays that way for months at a time.

- After the kernel image is completely loaded into RAM, the last thing the bootrom code does is print:

```
Starting at 0x30000...
```

and then jumps to the start address of the newly download kernel. (If it is the Y, Z or W processor, it will print a different starting address.) That is the last thing done by the bootrom code. After that anything done by the board is done by the kernel that was just downloaded.

8. New Kernel Starts

- The new Kernel starts up and outputs a **0x5** to the LEDs. Caches and interrupts are disabled and it does virtually all of the initialization (except for the Grackle) over again. This includes the auto-configuration of all the PCI devices.
- A hang with a **0x5** displayed usually indicates something went awry with the PCIbus initialization, although it can be due to some other cause.
- The kernel root task is now started.

9. Kernel Root Task Starts

- A **0x4** is displayed in the LEDs. At this point the processor re-initializes interrupts, starts the tick timer and then fills in the MMU page tables. Filling those tables can take several seconds if there is a lot of onboard memory that needs to be described by the page tables, so the **0x4** stays displayed for a long time.
- L2 cache is enabled at this point (if the kernel was compiled to use L2).
- Serial ports are initialized.
- If local SCSI or flash file RAM disk is being used, it is now initialized.
- Network software is initialized. At this point it will print:

```
Attaching network interface esyf0... done.
Attached TCP/IP interface to esyf unit 0
Initializing backplane net with anchor at 0x4100... done.
Backplane anchor at 0x4100... Attaching network interface sm0... done.
Attaching network interface lo0... done.
```

- This represents a large number of actions and is a vulnerable spot where things often hang up.
- Internal message queue software is initialized.

10. Tries to Download the `vxWorks.sym` File

- It outputs a **0x3** to the LEDs.
- On multi-processor boards, the mailbox interrupts are used for the first time when the non-X processor downloads the `vxWorks.sym` file. If a Y processor hangs displaying the message:

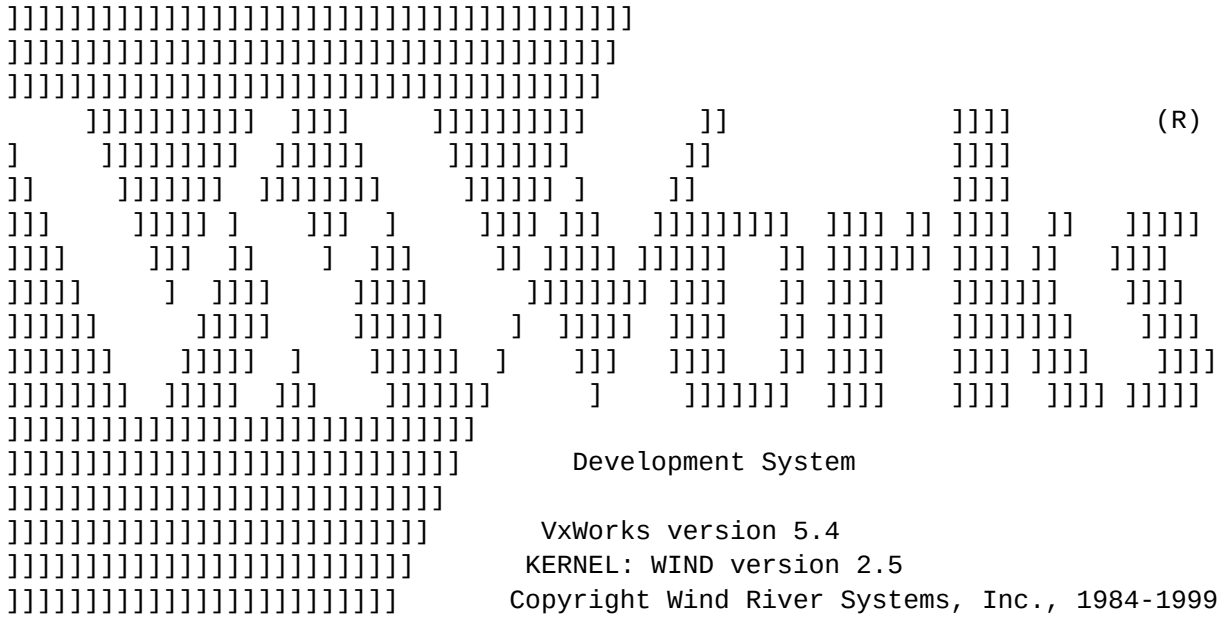
```
Loading symbol table from sluggo:/vxw/VXTEST/svgm5y/vxWorks.sym ...
```

then either there is some networking problem or the mailbox interrupt may not be working. It is most often a network problem of some kind. If it is a single processor board, then it is definitely a network problem because the mailbox interrupts are not used by VxWorks itself and are not enabled.

- It does a bit more network initialization.

11. Prints Out the VxWorks Banner

- Displays a **0x2** in the LEDs and prints the VxWorks banner:



- It then displays the board configuration. This is something like:

```
BSP:1.2/1f 0x02000000/0x04000000 RAM,MULTIcpu
Board: Synergy vgm5-750 dual. Processor #0.
MMU_BASIC, D-Cache=on, I-Cache=on, L2=copyback
Shared memory code uses bus locking.
Compiled May 15 2000 at 18:27:39, memory map B
```

12. Shell and Startup Script

- It next displays a **0x1** in the LEDs and initializes the VxWorks shell and, if a startup script was specified, it attempts to load it an interpret it.
- If the processor is a PPC 7400, the software that saves and restores the Altivec registers is now initialized.

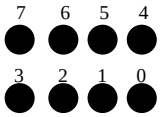
13. Done

- It is done loading, so it outputs a **0** to the LEDs.

LED Reference Guide

Below is a diagram showing the numbering convention for the 8 user LEDs on your on your Synergy PPC CPU board. Also shown is a quick reference guide for converting the hexidecimal code into the patterns shown in the 8 user LEDs.

User LED Configuration



Hex-to-LED Conversion Guide

For the X Processor:

Hex Value	LED Pattern	Hex Value	LED Pattern
0	none	0x8	3
0x1	0	0x9	0, 3
0x2	1	0xA	1, 3
0x3	0, 1	0xB	0, 1, 3
0x4	2	0xC	2, 3
0x5	0, 2	0xD	0, 2, 3
0x6	1, 2	0xE	1, 2, 3
0x7	0 , 1, 2	0xF	0, 1, 2, 3

For the Y Processor:

Hex Value	LED Pattern	Hex Value	LED Pattern
0	none	0x8	7
0x1	4	0x9	4, 7
0x2	5	0xA	5, 7
0x3	4, 5	0xB	4, 5, 7
0x4	6	0xC	6, 7
0x5	4, 6	0xD	4, 6, 7
0x6	5, 6	0xE	5, 6, 7
0x7	4 , 5, 6	0xF	4, 5, 6, 7

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP