Summary

This document describes the automatic generation of a Workbench Board Support Package (BSP) using Xilinx Platform Studio™ (XPS). The document contains the following sections.

- “Overview”
- “Generating the VxWorks 6.3 BSP”
- “The VxWorks 6.3 BSP”
- “Booting VxWorks”

Overview

One of the key embedded system development activities is the development of the BSP. The creation of a BSP can be a lengthy and tedious process that must be incurred when there is a change in the microprocessor complex which is comprised of the processor and associated peripherals. Although the management of these changes applies to any microprocessor-based project, now the changes can be accomplished more rapidly with the advent of programmable System-on-Chip (SoC) hardware.

This document describes automatic generation of a customized VxWorks 6.3 BSP for the IBM PowerPC™ 405 microprocessor and its peripherals as defined within a Xilinx FPGA. An automatically generated BSP enables embedded system designers to:

- Decrease substantially the development cycles, thereby decreasing the time-to-market
- Create a customized BSP to match the hardware and the application
- Eliminate BSP design bugs (automatically created based on certified components)
- Enable application software development by eliminating the wait for BSP development

The VxWorks 6.3 BSP is generated from XPS, an IDE delivered as part of the Xilinx Embedded Development Kit (EDK), used to create an embedded system within a Xilinx FPGA. The BSP contains all the necessary support software for a system, including boot code, device drivers, and RTOS initialization. The BSP is customized based on the peripherals chosen and configured by the user for the FPGA-based embedded system.

Experienced BSP designers should readily integrate a generated BSP into their target system. Conversely, less experience users may encounter difficulties because even though XPS can generate an operational BSP for a given set of IP hardware, there will always be some additional configuration and adjustments required to produce the best performance out of the target system. It is recommended that the user have available the Wind River VxWorks BSP Developer’s Guide and the VxWorks Application Programmer’s Guide or consider the Wind River classes on BSP design, available at an additional cost.

Requirements

The Wind River Workbench 2.5 development kit must be installed on the host computer. Because XPS generates relocatable BSPs that are compiled and configured outside the EDK environment, the host computer need not have both the Xilinx EDK and Workbench installed.
Microprocessor Library Definition

XPS supports a plug-in interface for 3rd party libraries and operating systems through the Microprocessor Library Definition (MLD) interface, thereby allowing 3rd party vendors to have their software available to XPS users. In addition, it provides the vendors a means for tailoring their libraries or BSPs to the FPGA-based embedded system created within XPS. Because the system can change easily, this capability is critical in properly supporting embedded systems in FPGAs.

Xilinx develops and maintains the VxWorks 6.3 MLD in its EDK releases. The MLD is used to automatically generate the VxWorks 6.3 BSP.

Template-Based Approach

A set of VxWorks 6.3 BSP template files are released with the EDK. These template files are used during automatic generation of the BSP and appropriate modifications are made based on the makeup of the FPGA-based embedded system.

These template files could be used as a reference for building a BSP from scratch if the user chooses not to automatically generate a BSP.

Device Drivers

A set of device driver source files are released with the EDK and reside in an installation directory. During creation of a customized BSP, device driver source code is copied from this installation directory to the BSP directory. Only the source code pertaining to the devices built into the FPGA-based embedded system are copied. This copy provides the user with a self-contained, standalone BSP directory which can be modified or relocated. If the user makes changes to the device driver source code for this BSP, then later wishes to undo the changes, the EDK tools can be used to regenerate the BSP. At that point, the device driver source files are recopied from the installation directory to the BSP.

Generating the VxWorks 6.3 BSP

Using XPS

XPS is available in the EDK and is a graphical design entry and implementation tool for a PowerPC 405-based or a MicroBlaze™-based embedded system. This section describes the steps needed to create a VxWorks 6.3 BSP using XPS. These steps are applicable when using EDK 8.2.2i and later. It is assumed that a valid hardware design has been entered into XPS.

1. Select the operating system
   In the Software Platform Settings dialog box, select VxWorks6_3 as the operating system.

2. Configure the VxWorks console device
   If a serial device such as a Uart is intended to be used as the VxWorks console, select or enter the instance name of the serial device as the STDIN/STDOUT peripheral in the OS and Libraries pane of the Software Platform Settings dialog box. It is important to enter the same device for both STDIN and STDOUT. Currently, only the Uart 16550/16450 and UartLite devices are supported as VxWorks console devices.

3. Integrate the device drivers
   a. Connect to VxWorks
      There is a connected_periphs dialog box available in the OS and Libraries pane. Peripherals have been pre-populated for user convenience. Use this dialog box to modify those peripherals to be tightly integrated with the OS, including the device that was selected as the STDIN/STDOUT peripheral. See the “Device Integration” section for more details on tight integration of devices.

4. Generate the VxWorks 6.3 BSP
   In the Software menu of XPS, select Generate Libraries and BSPs to generate the BSP. The output of this invocation is shown in the XPS output window. Once done, the resulting VxWorks 6.3 BSP will exist under the PowerPC 405 instance subdirectory of the user’s EDK project. For example, if
in XPS the user has named the PowerPC 405 instance, myppc405, the BSP will reside at <user
project>/myppc405/bsp_myppc405.

Backups
To prevent the inadvertent loss of changes made by the user to BSP source files, existing files in the
directory location of the BSP will be copied into a backup directory before being overwritten. The
backup directory resides within the BSP directory and is named backup<timestamp>, where
<timestamp> represents the current date and time. Because the BSP that is generated by XPS is
relocatable, it is recommended to relocate the BSP from the XPS project directory to an appropriate BSP
development directory as soon as the hardware platform is stable.

The VxWorks 6.3 BSP
This section describes the VxWorks 6.3 BSP output by XPS. It is assumed that the reader is familiar with
Wind River’s Workbench 2.5 IDE.

The automatically generated BSP is integrated into the Workbench IDE. The BSP can be compiled from
the command-line using the Workbench make tools, or from the Workbench Project facility (also
referred to as the Workbench IDE). Once the BSP has been generated, the user can type make
vxWorks from the command-line to compile a bootable RAM image. This assumes the Workbench
environment has been previously set up, which can be done via the command-line using the wrenv Wind
River environment utility on a a Windows platform. See the Wind River Workbench Command-Line
Users Guide: Creating a Development Shell With wrenv for more information on using the command-
line utilities. If using the Workbench Project facility, the user can create a project based on the newly
generated BSP, then use the build environment provided through the IDE to compile the BSP.

In Workbench 2.5, the diab compiler is supported in addition to the gnu compiler. The VxWorks 6.3 BSP
created by XPS has a Makefile that can be modified by the command-line user to use the diab compiler
instead of the gnu compiler. Look for the make variable named TOOLS and set the value to diab instead
of gnu. If using the Workbench Project facility, the user can select the desired tool when the project is
first created.
Driver Organization

This section briefly discusses how the Xilinx drivers are compiled and linked and eventually used by Workbench makefiles to be included into the VxWorks image.

Xilinx drivers are implemented in C programming language and can be distributed among several source files unlike traditional VxWorks drivers, which consist of single C header and implementation files.

There are up to three components for Xilinx drivers:

- Driver source inclusion.
- OS independent implementation
- OS dependent implementation (optional).

Driver source inclusion refers to how Xilinx drivers are compiled. For every driver, there is a file named `<procname>_drv_<dev>_version.c`. Using the `#include` command will include the source file(s) (*.c) for each driver for each given device.

This process is analogous to how the VxWorks sysLib.c `#include`'s source for Wind River supplied drivers. The reason why Xilinx files are not simply included in sysLib.c, as are the rest of the drivers, is because of namespace conflicts and maintainability issues. If all Xilinx files were part of a single compilation unit, static functions and data are no longer private. This places restrictions on the device drivers and would negate their operating system independence.

The OS independent part of the driver is designed for use with any operating system or any processor. It provides an API that utilizes the functionality of the underlying hardware. The OS dependent part of the driver adapts the driver for use with VxWorks. Such examples are SIO drivers for serial ports, or END drivers for ethernet adapters. Not all drivers require the OS dependent drivers, nor is it required to include the OS dependent portion of the driver in the VxWorks build.

Device Driver Location

The automatically generated BSP resembles most other Workbench BSPs except for the placement of device driver code. Off-the-shelf device driver code distributed with the Workbench IDE typically resides in the `vxworks-6.3/target/src/drv` directory in the Workbench installation directory. Device driver code for a BSP that is automatically generated resides in the BSP directory itself. This minor deviation is due to the dynamic nature of FPGA-based embedded system. Since the FPGA-based embedded system can be reprogrammed with new or changed IP, the device driver configuration can change, calling for a more dynamic placement of device driver source files.

The directory tree for the automatically generated BSP is shown in Figure 1.

```
<bsp_name>
  └── <csp_name>_csp
      └── xsrc
```

Figure 1: Driver Directory Location

The top-level directory is named according to the name of the processor instance in the XPS design project. The customized BSP source files reside in this directory. There is a subdirectory within the BSP directory named according to the processor instance with `_drv_csp` as a suffix. The driver directory contains two subdirectories. The `xsrec` subdirectory contains all the device driver related source files. If building from the Workbench Project facility, the files generated during the build process reside at `$PRJ_DIR/$BUILD_SPEC`.  

www.xilinx.com
Configuration

XPS BSPs are configured like any other VxWorks 6.3 BSP. There is little configurability to Xilinx drivers because the IP hardware has been pre-configured in most cases by XPS. The only configuration available generally is whether the driver is included in the VxWorks build at all. The process of including/excluding drivers depends on whether the Project facility or the command-line method is being used to perform the configuration activities.

Note that simply by including a Xilinx device driver does not mean that the driver will be automatically utilized. Most drivers with VxWorks adapters have initialization code. In some cases the user may be required to add the proper driver initialization function calls to the BSP.

When using XPS to generate a BSP, the resulting BSP files may contain “TODO” comments. These comments, many of which originate from the PowerPC 405 BSP template provided by Wind River, provides suggestions what the user must provide to configure the BSP for the target board. The VxWorks BSP Developer Guide and VxWorks Application Programmer’s Guide are very useful resources for BSP configuration.

Command-Line Driver Inclusion/Exclusion

Within the BSP, a set of constants (one for each driver) are defined in <procname>_drv_config.h and follow the format:

```c
#define INCLUDE_<XDRIVER>
```

This file is included near the top of config.h. By default all drivers are included in the build. To exclude a driver, add the following line in config.h after the inclusion of the <procname>_drv_config.h header file.

```c
#undef INCLUDE_<XDRIVER>
```

This exclusion will prevent the driver from being compiled and linked into the build. To re-instate the driver, remove the #undef line from config.h. Some care is required for certain drivers. For example, Ethernet may require that a DMA driver be present. Undefining the DMA driver will cause the build to fail.

Project Facility Driver Inclusion/Exclusion

The file 50<csp_name>.cdf resides in the BSP directory and is tailored during creation of the BSP. This file integrates the Xilinx device drivers into the Workbench IDE. The Xilinx device drivers are hooked into the IDE at the hardware/peripherals sub-folder of the components tab. Below this are individual device driver folders. An example of the GUI with Xilinx drivers is shown in Figure 2. To add or delete Xilinx drivers, include or exclude driver components as with any other VxWorks component.
The VxWorks 6.3 BSP

The automatically generated BSPs follow the standard Workbench conventions when it comes to creating VxWorks images. Refer to Workbench documentation on how to make a VxWorks image.

**Building VxWorks**

The Xilinx drivers are compiled/linked with the same toolchain VxWorks is built with. Minor additions to the Makefile were required to help Workbench find the location of driver source code files.

**Command-Line BSP Build Extensions**

The number of new files used to integrate the Xilinx device drivers into the Workbench build process can be seen in the <bsp_name> directory. As stated earlier, these files are automatically created by XPS. The user need only be aware of that the files exist. These files are prefixed with the instance name of the processor.

---

Figure 2: Workbench 2.5 Project IDE - VxWorks

Note that whatever configuration has been specified in <procname>_drv_config.h and config.h will be overridden by the project facility.
Device Integration

Devices in the FPGA-based embedded system have varying degrees of integration with the VxWorks operating system. The degree of integration is selectable by the XPS user in the Connected Peripherals dialog box of the Library/OS Parameters tab. Below is a list of currently supported devices and their level of integration.

- One or more UART 16450/16550/Lite devices can be integrated into the VxWorks Serial I/O (SIO) interface. This makes a UART available for file I/O and printf/stdio. Only one UART device can be selected as the console, where standard I/O (stdin, stdout, and stderr) is directed. A UART device, when integrated into the SIO interface, must be capable of generating an interrupt. Reference the `sysSerial.c` file of the BSP to see details of this integration.

- Ethernet 10/100 MAC, Ethernet Lite 10/100, Gigabit Ethernet MAC, and 10/100/1000 Tri-speed Ethernet MAC devices can be integrated into the VxWorks Enhanced Network Driver (END) interface. This makes the device available to the VxWorks network stack and thus socket-level applications. An Ethernet device, when integrated into the END interface, must be capable of generating interrupts. Reference the `configNet.h` and `sysNet.c` files of the BSP to see details of this integration. Also, the user might need to modify the default bootline values in `config.h` for the Ethernet device to be used as the boot device.

- An Interrupt controller can be connected to the VxWorks intLib exception handling and the PowerPC 405 external non-critical interrupt pin. The generated BSP does not currently handle interrupt controller integration for the critical interrupt pin of the PowerPC 405, nor does it support direct connection of a single interrupting device (other than the intc) to the processor. However, the user is always able to add manually this integration in the `sysInterrupt.c` file of the BSP.

- A System ACE™ controller can be connected to VxWorks as a block device, allowing the user to attach a filesystem to the CompactFlash device connected to the System ACE controller. The user must call manually the BSP functions to initialize the System ACE/CompactFlash as a block device and attach it to the DOS operating system. The functions currently available to the user are: `sysSystemAceInitFS()` and `sysSystemAceMount()`. A system ACE controller, when integrated into the block device interface, must be capable of generating an interrupt. Reference the file `sysSystemAce.c` in the BSP for more details. The BSP will mount the CF as a DOS FAT disk partition using Wind River’s DosFs2.0 add-on. To get the required VxWorks libraries into the image, the following packages must be defined in `config.h` or by the Project Facility:
  - `INCLUDE_DOSFS_MAIN`
  - `INCLUDE_DOSFS_FAT`
  - `INCLUDE_DISK_CACHE`
  - `INCLUDE_DISK_PART`
  - `INCLUDE_DOSFS_DIR_FIXED`
  - `INCLUDE_DOSFS_DIR_VFAT`
  - `INCLUDE_XBD_BLK_DEV`
  - `INCLUDE_XBD_PART_LIB`

Programmatically, an application can mount the DOS file system using the following API calls:

```c
FILE *fp;
sysSystemAceInitFS();
if (sysSystemAceMount("/cf0", 1) != OK)
{
    /* handle error */
}
fp = fopen("/cf0/myfile.dat","r");
```

- A PCI bridge can be initialized and made available to the standard VxWorks PCI driver and configuration functions. The user is required to edit the config.h and `sysBusPci.c` BSP files to tailor the PCI memory addresses and configuration for their target system. Note that PCI interrupts are not automatically integrated into the BSP.
• All other devices and associated device drivers are not tightly integrated into a VxWorks interface. Instead, they are loosely integrated and access to these devices is available by directly accessing the associated device drivers from the user’s application.

• User cores and associated device drivers, if included in the EDK project, are supported through the BSP generation flow. The user core device drivers will be copied into the BSP in the same way the Xilinx device drivers are copied. This assumes the directory structure of the user core device driver matches the structure of the Xilinx device drivers. The /data and /build sub-directories of the device driver must exist and be formatted in the same way as the Xilinx device drivers. This includes the CDF snippet and xtag files in the /build/vxworks5_4 sub-directory. User device drivers are not automatically integrated into any OS interface (e.g., SIO), but they are available for direct access by an application.

Deviations

The following list summarizes the differences between XPS generated BSPs and traditional BSPs.

• An extra directory structure is added to the root BSP directory to contain the device driver source code files.

• To keep the BSP buildable while maintaining compatibility with the Workbench Project facility, a set of files named <procname>_drv_<driver>_<version>.c populate the BSP directory that simply #include the source code from the driver subdirectory of the BSP.

• The BSP Makefile has been modified so that the compiler can find the driver source code. The Makefile contains more information about this deviation and its implications.

• SystemACE usage may require changes to VxWorks source code files found in the Workbench distribution directory. See the “Bootrom with SystemACE as the Boot Device” section.

Limitations

The automatically generated BSP should be considered a good starting point for the user, but should not be expected to meet all the user’s needs right out of the box. Due to the potential complexities of a BSP, the variety of features that can be included in a BSP, and the support necessary for board devices external to the FPGA, the automatically generated BSP will likely require enhancements by the user. However, the generated BSP will be compilable and will contain the necessary device drivers represented in the FPGA-based embedded system. Some of the commonly used devices are also integrated with the operating system. Specific limitations are listed below.

• An interrupt controller connected to the PowerPC 405 critical interrupt pin is not automatically integrated into VxWorks’ interrupt scheme. Only the external interrupt is currently supported.

• Bus error detection from bus bridges or arbiters is not supported.

• The command-line VxWorks 6.3 BSP defaults to use the GNU compiler. The user must manually change the Makefile to use the DIAB compiler, or specify the DIAB compiler when creating a Workbench project based on the BSP.

• Memory address ranges might need to be tailored in config.h to match specific memory devices and their address ranges.

• PowerPC 405 caches are disabled by default. The user must enable caches manually through the config.h file or the Workbench project menu.

• When SystemACE is setup to download VxWorks images into RAM via JTAG, all boots are cold (i.e., no warm boots). This is because the System ACE controller resets the processor whenever it performs an ace file download. An effect of this could cause exception messages generated by VxWorks to not be printed on the console when the system is rebooted due to an exception in an ISR or a kernel panic.

Note: No compressed images can be used with SystemACE. This applies to standard compressed images created with Workbench such as bootrom. Compressed images cannot be placed on SystemACE as an ace file. SystemACE cannot decompress data as it writes it to RAM. Starting such an image will lead to a system crash.
Booting VxWorks

- A command-line build can’t initialize the network when SystemACE is the boot device. This requires that the application provide code to initialize the network when SystemACE is the boot device. To get around this issue see the discussion of $WIND_BASE/target/src/config/usrNetwork.c in the “Bootrom with SystemACE as the Boot Device” section.

- On the PowerPC 405 processor, the reset vector is at physical address 0xFFFFFC. There is a short time window where the processor will attempt to fetch and execute the instruction at this address while SystemACE processes the ace file. The processor needs to be given something to do during this time even if it is a spin loop:

  FFFFFFFC b.

If BRAM occupies this address range, then the designer who creates the bitstream should place instructions here with the elf to BRAM utility found in the Xilinx ISE tools.

- VxBus support is currently not available.

Booting VxWorks

VxWorks Bootup Sequence

There are many variations of VxWorks images with some based in RAM, some in ROM. Depending on board design, not all these images are supported. The following list discusses various image types:

- ROM compressed images - These images begin execution in ROM and decompress the BSP image into RAM, then transfer control to the decompressed image in RAM. This image type is not compatible with SystemACE because SystemACE doesn’t know the image is compressed and will dutifully place it in RAM at an address that will be overwritten by the decompression algorithm when it begins. It may be possible to get this type of image to work if modifications are made to the standard Workbench makefiles to handle this scenario.

- RAM based images - These images are loaded into RAM by a bootloader, SystemACE, or an emulator. These images are fully supported.

- ROM based images - These images begin execution in ROM, copy themselves to RAM then transfer execution to RAM. In designs with SystemACE as the bootloader, the image is automatically copied to RAM. The handcoded BSP examples short-circuit the VxWorks copy operation so that the copy does not occur again after control is transferred to RAM by SystemACE (see romInit.s).

- ROM resident images - These images begin execution in ROM, copy the data section to RAM, and execution remains in ROM. In systems with only a SystemACE, this image is not supported. Theoretically BRAM could be used as a ROM, however, the current Virtex-II Pro parts being used in evaluation boards do not have the capacity to store a VxWorks image which could range in size from 200KB to over 700KB.

VxWorks Boot Sequence

This standard image is designed to be downloaded to the target RAM space by some device. Once downloaded, the processor is setup to begin execution at function _sysInit at address RAM_LOW_ADRS. (this constant is defined in config.h and Makefile). Most of the time, the device performing the download will do this automatically as it can extract the entry point from the image.

1. _sysInit: This assembly language function running out of RAM performs low level initialization. When completed, this function will setup the initial stack and invoke the first “C” function usrInit(). _sysInit is located in source code file <bspname>/sysALib.s.

2. usrInit(): This “C” function running out of RAM sets up the “C” runtime environment and performs pre-kernel initialization. It invokes sysHwInit() (implemented in sysLib.c) to place the HW in a quiescent state. When completed, this function will call kernelInit() to bring up the VxWorks kernel. This function will in turn invoke usrRoot() as the first task.

3. usrRoot(): Performs post-kernel initialization. Hooks up the system clock, initializes the TCP/IP stack, etc. It invokes sysHwInit2() (implemented in sysLib.c) to attach and enable HW
interrupts. When complete, usrRoot() invokes user application startup code usrAppInit() if so configured in the BSP.

Both usrInit() and usrRoot() are implemented by Wind River. The source code files they exist in are different depending on whether the command line or the Workbench Project facility is being used to compile the system. Under the command line interface, they are implemented at $WIND_BASE/target/config/all/usrConfig.c. Under the project facility, they are maintained in the user’s project directory.

"bootrom_uncmp" Boot Sequence

This standard image is ROM based but in reality it is linked to execute out of RAM addresses. While executing from ROM, this image uses relative addressing tricks to call functions for processing tasks before jumping to RAM.

1. Power on. Processor vectors to 0xFFFFFFFC where a jump instruction should be located that transfers control to the bootrom at address _romInit.

2. _romInit : This assembly language function running out of ROM notes that this is a cold boot then jumps to start. Both _romInit and start are located in source code file <bspname>/romInit.s.

3. start : This assembly language function running out of ROM sets up the processor, invalidates the caches, and prepares the system to operate out of RAM. The last operation is to invoke “C” function romStart() which is implemented by Wind River and is located in source code file $WIND_BASE/target/config/all/bootInit.c.

4. romStart() : This “C” function running out of ROM copies VxWorks to its RAM start address located at RAM_HIGH_ADRS (this constant is defined in config.h and Makefile). After copying VxWorks, control is transferred to function usrInit() in RAM.

5. Follows steps 2 & 3 of the "vxWorks" bootup sequence.

"bootrom_uncmp" Boot Sequence with SystemACE

This non-standard image is similar to the image discussed in the previous section except that SystemACE is used to load it. Several changes have to be made to the boot process. More information can be found in section <RD Red><BT BoldType>Bootrom with SystemACE as the Boot Device<RD Red>, page 12.

1. Power on. SystemACE loads the image into RAM at RAM_HIGH_ADRS (this constant is defined in config.h and Makefile) and sets the processor to begin fetching instructions at address _romInit.

2. _romInit : This assembly language function running out of RAM notes that this is a cold boot then jumps to start. Both _romInit and start are located in source code file <bspname>/romInit.s.

3. start : This assembly language function running out of RAM simply jumps to function _sysInit. The call to romStart() is bypassed because SystemACE has already loaded the bootrom into it’s destination RAM address.

4. Follow steps 1, 2, & 3 of the “vxWorks” bootup sequence.

Bootroms

The bootrom is a scaled down VxWorks image that operates in much the same way a PC BIOS does. Its primary job is to find and boot a full VxWorks image. The full VxWorks image may reside on disk, in flash memory, or on some host via the Ethernet. The bootrom must be compiled in such a way that it has the ability to retrieve the image. If the image is retrieved from an Ethernet network, then the bootrom must have the TCP/IP stack compiled in, if the image is on disk, then the bootrom must have disk access support compiled in, etc. The bootroms do little else than retrieve and start the full image and maintain a bootline. The bootline is a text string that sets certain user characteristics such as the target’s IP address if using Ethernet and the file path to the VxWorks image to boot.

Bootroms are not a requirement. They are typically used in a development environment then replaced with a production VxWorks image.
Creating Bootroms

At a command shell in the BSP directory, issue the following command to create an uncompressed bootrom image (required for SystemACE):

```make
make bootrom_uncmp
```
or

```make
make bootrom
```
to create a compressed image suitable for placing in a flash memory array.

Bootrom Display

Upon cycling power, if the bootroms are working correctly, output similar to the following should be seen on the console serial port:

```
VxWorks System Boot

Copyright 1984-2006 Wind River Systems, Inc.

CPU: ppc405_0 VirtexII Pro PPC405
Version: VxWorks 6.3
BSP version: 1.2/0
Creation date: Aug 11, 2006, 16:40:32

Press any key to stop auto-boot...
3

[VxWorks Boot]:
```

Typing the **help** at this prompt lists the available commands.

Bootline

The bootline is a text string that defines user serviceable characteristics such as the IP address of the target board and how to find a VxWorks image to boot. The bootline is maintained at runtime by the bootrom and is typically kept in some non-volatile (NVRAM) storage area of the system such as an EEPROM or flash memory. If there is no NVRAM, or an error occurs reading it, then the bootline is hard-coded with DEFAULT_BOOT_LINE defined in the BSP's config.h source code file. In new systems where NVRAM has not been initialized, the bootline may be undefined data.

The bootline can be changed if the auto-boot countdown sequence is interrupted by entering a character on the console serial port. The "c" command can then be used to interactively edit the bootline. Enter "p" to view the bootline. On a non-bootrom image, the bootline can be changed by entering the bootChange command at a host or target shell prompt.

The bootline fields are defined below:

- **boot device**: Device to boot from. This could be Ethernet, or a local disk. Note that when changing the bootline, the unit number may be shown appended to this field ("xemac0" or "sysace=10) when prompting for the new boot device. This number can be ignored.
- **processor number**: Always 0 with single processor systems.
- **host name**: Name as needed.
- **file name**: The VxWorks image to boot.
- **inet on ethernet (e)**: The IP internet address of the target. If there is no network interface, then this field can be left blank.
Booting VxWorks

- **host inet (h)**: The IP internet address of the host. If there is no network interface, then this field can be left blank.
- **user (u)**: Username for host file system access. Pick whatever name suites you. Your ftp server must be setup to allow this user access to the host file system.
- **ftp password (pw)**: Password for host file system access. Pick whatever name suites you. Your ftp server must be setup to allow this user access to the host file system.
- **flags (f)**: For a list of options, enter the "help" command at the [VxWorks Boot]: prompt.
- **target name (tn)**: Name as needed. Set per network requirements.
- **other (o)**: This field is useful when you have a non-Ethernet device as the boot device. When this is the case, VxWorks will not start the network when it boots. Specifying an Ethernet device here will enable that device at boot time with the network parameters specified in the other bootline fields.
- **inet on backplane (b)**: Typically left blank if the target system is not on a VME or PCI backplane.
- **gateway inet (g)**: Enter an IP address here if you have to go through a gateway to reach the host computer. Otherwise leave blank.
- **startup script (s)**: Path to a file on the host computer containing shell commands to execute once bootup is complete. Leave blank if not using a script. Examples:
  - SystemACE resident script: /cf0/vxworks/scripts/myscript.txt
  - Host resident script: c:/temp/myscript.txt

Bootrom with SystemACE as the Boot Device

SystemACE enabled bootroms are capable of booting VxWorks images directly off the Compact Flash device either as a regular elf file or an ace file.

**Required Modifications to VxWorks Source**

While the EDK is capable of generating a BSP that uses SystemACE in a “vxWorks” image that can open and close files in a DOS filesystem, it cannot generate a BSP that uses SystemACE as a “bootrom” boot device. To use SystemACE in this way requires extensive modifications to bootrom code provided by Wind River. Wind River allows BSP developers to change Workbench source code files provided they keep the changes local to the BSP and leave the original code as is. The two files that have to be modified from their original version are:

1. `$WIND_BASE/target/config/all/bootConfig.c`: This file is overridden with one found in the `<bspname>` directory. The changes needed are to add code to properly parse the bootline and to initialize and use SystemACE as a JTAG and DOS boot device. To override the default bootConfig.c, the following line must be added to the BSP’s Makefile:
   ```make
   BOOTCONFIG = .:/bootConfig.c
   ```

2. `$WIND_BASE/target/config/comps/src/net/usrNetBoot.c`: This file is overridden with one found in the `<bspname>/net/usrNetBoot.c` directory. The changes needed are to add code to make VxWorks aware that SystemACE is a disk based system like IDE, SCSI, or floppy drives. This change allows a BSP built from the Workbench Project downloaded with a SystemACE enabled bootrom to properly process the **other** field of the bootline. The existence of the modified file in the BSP directory automatically overrides the original.

Neither of these files are provided by the EDK because they are maintained in their original form by Wind River. The ML300 reference BSP contains versions of these files with the modifications to support SystemACE as a boot device. Those versions can be copied to the developer’s BSP.

A third file, `$WIND_BASE/target/src/config/usrNetwork.c`, cannot be overridden because of the architecture of the command line BSP build process. This affects network capable BSPs built from the command line that are downloaded with a SystemACE enabled bootrom. Without modifying usrNetwork.c, affected BSPs are unable to initialize their network device and must rely on application
code to perform this function. If the user desires, they can make the change to this file in their Workbench installation. There are disadvantages to this approach because any edits made to this file affect all users of that installation and may be lost if the user upgrades or re-installs Workbench. The change to usrNetwork.c occurs in function usrNetDevStart().

from:

```c
if ((strncmp (params.bootDev, "scsi", 4) == 0) ||
    (strncmp (params.bootDev, "ide", 3) == 0) ||
    (strncmp (params.bootDev, "ata", 3) == 0) ||
    (strncmp (params.bootDev, "fd", 2) == 0)  ||
    (strncmp (params.bootDev, "tffs", 4) == 0))
```

to

```c
if ((strncmp (params.bootDev, "scsi", 4) == 0) ||
    (strncmp (params.bootDev, "ide", 3) == 0) ||
    (strncmp (params.bootDev, "ata", 3) == 0) ||
    (strncmp (params.bootDev, "fd", 2) == 0)  ||
    (strncmp (params.bootDev, "sysace", 6) == 0) ||
    (strncmp (params.bootDev, "tffs", 4) == 0))
```

Edit this code at your own risk.

**Special Configuration**

Preparing a bootrom_uncmp image downloadable by SystemACE as an ace file requires special configuration. These changes are required because the bootrom is linked to begin running out of a non-volatile memory device, copy itself to RAM, then transfer control to the RAM copy. The changes will prevent the copy operation since SystemACE has already placed the bootrom into a RAM device at reset.

a. Change the definitions of ROM_TEXT_ADRS and ROM_WARM_ADRS to a value equivalent to RAM_HIGH_ADRS in both config.h and Makefile.

b. Change the assembly language code at the start label in romInit.s to jump to function _sysInit (Note: an example of this usage is available in the romInit.s source code file in the ML300 reference BSP):

```assembly
start:
    LOADPTR (r1, _sysInit)
    mtlr r1
    blrl
```

**Bootline Format**

The *boot device* field 6_3 of the bootline is specified using the following syntax:

```
sysace=<partition number>
```

where `<partition number>` is the partition to boot from. Normally, this value is set to 1, but some CF devices do not have a partition table and are formatted as if they were a large floppy disk. In this case, specify 0 as the partition number. Failure to get the partition number correct will lead to errors being reported by VxWork’s dosFS libraries when the drive is mounted.

The *file name* field of the bootline is set depending on how the System ACE is to boot the system. There are two boot methods:

1. Boot from a regular file. This is similar to network booting in that the vxWorks image resides in the SystemACE compact flash storage device instead of the host file system. The compact flash device is a DOS FAT file system partition. Build vxWorks using the Workbench tools, copy the resulting image file to the compact flash device using a USB card reader or similar tool, then specify that file in the *file name* field of the boot rom. The file name must have the following syntax:

```
/cf0/<path/to/vxWorks/image>
```
where c0 is the mount point. <path/to/vxWorks/image> should provide the complete path to the VxWorks image to boot. When being specified in this way, the bootrom will mount the drive as a FAT formatted disk, load the file into memory and begin execution.

2. Boot from an ace file. The ace file can contain HW only, SW only, or HW + SW. When booting from an ace file with HW, the FPGA is reprogrammed. If the ace file contains SW, then it is loaded into the memory, the processor’s PC is set to the entry point and released to begin fetching instructions. This boot method is flexible in that a totally different HW profile can be booted from a VxWorks bootrom. The file name must have the following syntax:

\[ \text{cfgaddr}[x] \]

where \([X]\) is a number between 0 and 7 that corresponds to one of the configuration directories specified in the XILINX.SYS file resident in the root directory of the compact flash device. If \([X]\) is omitted, then the default configuration is used. The default configuration is typically selected by a rotary switch mounted somewhere on the evaluation board. The bootrom will trigger a JTAG download of the ace file pointed to by the specified config address. There should be only a single file with an .ace extension in the selected configuration directory.

In either boot scenario, if the Ethernet device is to be started when the downloaded VxWorks starts, the "other" field of the bootline must be modified to contain the name of the network device.

**Bootrom with 10/100 Ethernet (EMAC) as the Boot Device**

XPS will generate a BSP that is capable of being built as a bootrom using the EMAC as a boot device. Very little user configuration is required. The MAC address is hard coded in the source file sysNet.c. The BSP can be used with the default MAC as long as the target is on a private network and there is no more than one target on that network with the same default MAC address. Otherwise the designer should replace this MAC with a function to retrieve one from a non-volatile memory device on their target board.

To specify the EMAC as the boot device in the bootrom, change the **boot device** field in the bootline to **xemac**. If there is a single EMAC, set the unit number to **0**.

**Bootrom with 1 Gigabit Ethernet (GEMAC) as the Boot Device**

XPS will generate a BSP that is capable of being built as a bootrom using the GEMAC as a boot device. Very little user configuration is required. The MAC address is hard coded in the source file sysNet.c. The BSP can be used with the default MAC as long as the target is on a private network and there is no more than one target on that network with the same default MAC address. Otherwise the designer should replace this MAC with a function to retrieve one from a non-volatile memory device on their target board.

To specify the GEMAC as the boot device in the bootrom, change the **boot device** field in the bootline to **xgemac**. If there is a single GEMAC, set the unit number to **0**.

**Bootline Examples**

The following example boots from the ethernet using the Xilinx xemac as the boot device. The image booted is on the host file system on drive C.

```
boot device          : xemac
unit number          : 0
processor number     : 0
host name            : host
file name            : c:/WindRiver/vxworks-6.3/target/config/ML300/vxWorks
inet on ethernet (e) : 192.168.0.2
host inet (h)        : 192.168.0.1
user (u)             : xemhost
ftp password (pw)    : whatever
flags (f)            : 0x0
target name (tn)     : vxtarget
other (o)            : 
```
The following example boots from a file resident on the first partition of the SystemACE’s compact flash device. If the file booted from `/cf0/vxworks/images/vxWorks` utilizes the network, then the `xemac` device is initialized.

```
boot device          : sysace=1
unit number          : 0
processor number     : 0
host name            : host
file name            : /cf0/vxworks/images/vxWorks
inet on ethernet (e) : 192.168.0.2
host inet (h)        : 192.168.0.1
user (u)             : xemhost
ftp password (pw)    : whatever
flags (f)            : 0x0
target name (tn)     : vxtarget
other (o)            : xemac
```

The following example boots from an ace file resident on the first partition of the SystemACE’s compact flash device. The location of the ace file is set by `XILINX.SYS` located in the root directory of the compact flash device. If the ace file contains a VxWorks SW image that utilizes the network, then the `xemac` device is initialized for that image.

```
boot device          : sysace=1
unit number          : 0
processor number     : 0
host name            : host
file name            : cfgaddr2
inet on ethernet (e) : 192.168.0.2
host inet (h)        : 192.168.0.1
user (u)             : xemhost
ftp password (pw)    : whatever
flags (f)            : 0x0
target name (tn)     : vxtarget
other (o)            : xemac
```

### Caches

The instruction and data caches are managed by VxWorks proprietary libraries. They are enabled by modifying the following constants in `config.h` or by using the Workbench Project facility to change the constants of the same name:

- **INCLUDE_CACHE_SUPPORT**: If defined, the VxWorks cache libraries are linked into the image. If caching is not desired, then `#undef` this constant.
- **USER_I_CACHE_ENABLE**: If defined, VxWorks will enable the instruction cache at boottime. Requires `INCLUDE_CACHE_SUPPORT` to have any effect.
- **USER_D_CACHE_ENABLE**: If defined, VxWorks will enable the data cache at boottime. Requires `INCLUDE_CACHE_SUPPORT` to have any effect.

### MMU

If the MMU is enabled, then the cache control discussed in the previous section may not have any effect. The MMU is managed by VxWorks proprietary libraries but the initial setup is defined in the BSP. To enable the MMU, the constant `INCLUDE_MMU_BASIC` should be defined in `config.h` or by using the Project Facility. The constant `USER_D_MMU_ENABLE` and `USER_I_MMU_ENABLE` control whether the instruction and/or data MMU is utilized.

VxWorks initializes the MMU based on data in the `sysPhysMemDesc` structure defined in `sysCache.c`. User reserved memory and ED&R (when `INCLUDE_EDR_PM` is enabled) reserved memory is
included in this table. Amongst other things, this table configures memory areas with the following attributes:

- Whether instruction execution is allowed.
- Whether data writes are allowed
- Instruction & data cacheability attributes.
- Translation offsets used to form virtual addresses.

When VxWorks initializes the MMU, it takes the definitions from sysPhysMemDesc and creates page table entries (PTEs) in RAM. Each PTE describes 4KB of memory area (even though the processor is capable of representing up to 16MB per PTE). Beware that specifying large areas of memory uses substantial amounts of RAM to store the PTEs. To map 4MB of contiguous memory space takes 8KB of RAM to store the PTEs.

To increase performance with the VxWorks basic MMU package for the PPC405 processor, it may be beneficial to not enable the instruction MMU and rely on the cache control settings in the ICCR register. This strategy can dramatically reduce the number of page faults while still keeping instructions in cache. The initial setting of the ICCR is defined in the `<bspname>.h` header file.

Without the MMU enabled, the following rules apply to configuring memory access attributes and caching:

- There is no address translation, all effective addresses are physical.
- Cache control granularity is 128MB.
- The guarded attribute applies only to speculative instruction fetches on the PPC405.