Processor IP Reference Guide

January 2003

XILINX®

Processor IP Reference Guide
January 2003

The following table shows the revision history for this document..

	Version	Revision
August 2002	1.0	Initial Xilinx release for EDK 3.1
October 2002	1.1	Add memory and peripheral cores
November 2002	1.2	Release for EDK 3.1 Service Pack 2
January 2003	1.3	Release for EDK 3.1 Service Pack 3

About This Manual

The Processor IP Reference Guide supports the Embedded systems Design Kit (EDK) for MicroBlaze™ and Virtex-II Pro™.

Note: For more information, refer to the *Embedded Software Tools Reference Guide* and *PowerPC 405 Processor Reference Guide*.

Manual Contents

This manual contains the following sections:

This page left intentionally blank

Part I: Embedded Processor IP

This section contains information on the following:

Chapter 1

OPB Usage in FPGAs

Overview

This chapter includes the following sections:

For detailed information on the IBM OPB, refer to IBM's *On-Chip Peripheral Bus, Architecture Specifications, Version 2.1*: **OpbBus.pdf**

The OPB is one element of IBM's CoreConnect architecture, and is a general-purpose synchronous bus designed for easy connection of on-chip peripheral devices. The OPB includes the following features:

- 32-bit or 64-bit data bus
- Up to 64-bit address
- Supports 8-bit, 16-bit, 32-bit, and 64-bit slaves
- Supports 32-bit and 64-bit masters
- Dynamic bus sizing with byte, halfword, fullword, and doubleword transfers
- Optional Byte Enable support
- Distributed multiplexer bus instead of 3-state drivers
- Single cycle transfers between OPB master and OPB slaves (not including arbitration)
- Support for sequential address protocol
- 16-cycle bus time-out (provided by arbiter)
- Slave time-out suppress capability
- Support for multiple OPB bus masters
- Support for bus parking
- Support for bus locking
- Support for slave-requested retry
- Bus arbitration overlapped with last cycle of bus transfers

The OPB is a full-featured bus architecture with many features that increase bus performance. You can use most of these features effectively in the FPGA architecture. However, some features can result in the inefficient use of FPGA resources or can lower system clock rates. Consequently, Xilinx uses an efficient *subset* of the OPB for Xilinx-developed OPB devices. However, because of the flexible nature of FPGAs, you can also implement systems utilizing OPB devices that are fully OPB V2.1 compliant.

Xilinx OPB Usage

OPB Options

Legacy Devices

Previous to OPB V2.0, there was a single signaling protocol for OPB data transfers. This protocol (which is also present in OPB V2.0 and later specifications) supports dynamic bus sizing through the use of transfer qualifiers and acknowledge signals. The transfer qualifiers denote the size of the transfer initiated by the master, and the acknowledge signals indicate the size of the transfer from the slave. Devices that support this type of dynamic bus sizing are called *legacy devices*.

Byte-enable Devices

Starting with OPB V2.0, IBM introduced an optional, alternate transfer protocol based on Byte Enables. In the byte-enable architecture, each byte lane of the data bus has an associated byte enable signal. For each transfer, the byte enable signals indicate which byte lanes have valid data. This eliminates the need for separate transfer qualifiers that indicate the transfer size since all size information is contained in the byte enable signals. The byte-enable architecture does not permit dynamic bus sizing, since there is only one acknowledge signal for each transfer. The OPB V2.0 specification (and later) allows you to build systems that are legacy-only, byte-enable only, or mixed. Devices that only support the byte-enable signaling are called *byte-enable devices*.

OPB V2.0 Devices

Devices that support both byte-enable signaling and legacy signaling are called *OPB V2.0 devices*. Systems that have both legacy signaling and byte-enable signaling can perform dynamic bus sizing. Note that legacy devices do not support byte-enable transfers.

Xilinx OPB Devices

These various transfer protocols have several implications for Xilinx OPB device implementations.

Conversion Cycles

Dynamic bus sizing (as supported by legacy devices) results in *conversion cycles*, which are extra transfer cycles that re-transfer data when the master-initiated transfer is larger than the slave response. For example, in a legacy system, if a master writes a 32-bit word to a slave, and the 8-bit device slave responds that it only accepted 8-bits of the transfer, then the master must perform three additional conversion cycles to transfer all of the data to the slave. Generating conversion cycles requires more logic, increases the complexity of the master, and is not an efficient use of FPGA resources. The byte-enable architecture provides a simple alternative to this problem, and is easier to implement in an FPGA.

Write Mirroring and Read Steering

Another consequence of supporting devices smaller than the bus size is *write mirroring* and *read steering*. In the OPB specification, devices smaller than the bus size are always left-justified (aligned toward the most significant side of the bus) so that the byte lanes associated with the smaller devices are easily determined. For example, a byte-wide peripheral is always located on the most-significant byte of the bus. The peripheral writes and reads data using this byte-lane. You can simplify the design of OPB masters by using a byte-enable only, no-write-mirroring architecture. A small degree of added complexity is required for peripherals that are smaller than the bus size if OPB masters do not mirror data.

Ideal FPGA Implementation of OPB-based System

The ideal FPGA implementation of an OPB-based system has the following features:

- Requires no conversion cycles
- Uses only the byte-enable architecture as specified in the OPB specification
- Does not require masters to mirror write data

These characteristics help determine how Xilinx-developed OPB devices are implemented. The detailed specifications that describe how the OPB is used in Xilinx intellectual property are provided in the next section

.

Specifications for OPB Usage in Xilinx-developed OPB Devices

Xilinx-developed OPB devices adhere to the following OPB usage rules:

- The width of the OPB data buses and address buses is 32 bits. Note that some peripherals may parameterize these widths, but currently only 32-bit buses are supported. Peripherals that are smaller than 32-bits can be attached to the OPB with a corresponding restriction in addressing. For example, an 8-bit peripheral at base address A can be attached to byte lane 0, but can only be addressed at A, A+4, A+8, and so on.
- All OPB devices (masters and slaves) are byte-enable devices. These devices do not support the legacy data transfer signals and therefore do not support dynamic bus sizing. OPB masters do not mirror data to unused byte lanes. See Figure 1-1 for the byte lane usage for aligned transfers.
- All OPB devices (masters and slaves) are required to output logic zero when they are inactive. This eliminates the need for the Mn_DBusEn and Sln_DBusEn signals external to the master or slave. The enable function is still implemented within the device.
- To obtain better timing in the FPGA implementation of the OPB, the OPB_timeout signal is registered. This means that all slaves must assert Sl_xferAck or Sl_retry on or before the rising edge of the 16th clock cycle after the assertion of OPB_select. If an OPB slave wishes to assert Sl_toutSup, Sl_toutSup must be asserted on or before the rising edge of the 15th clock after the assertion of OPB_select.
- The byte-enables and the least-significant address bits are driven by all masters and contain consistent information. Examples of byte lane usage for aligned transfers are

shown in the following figure:

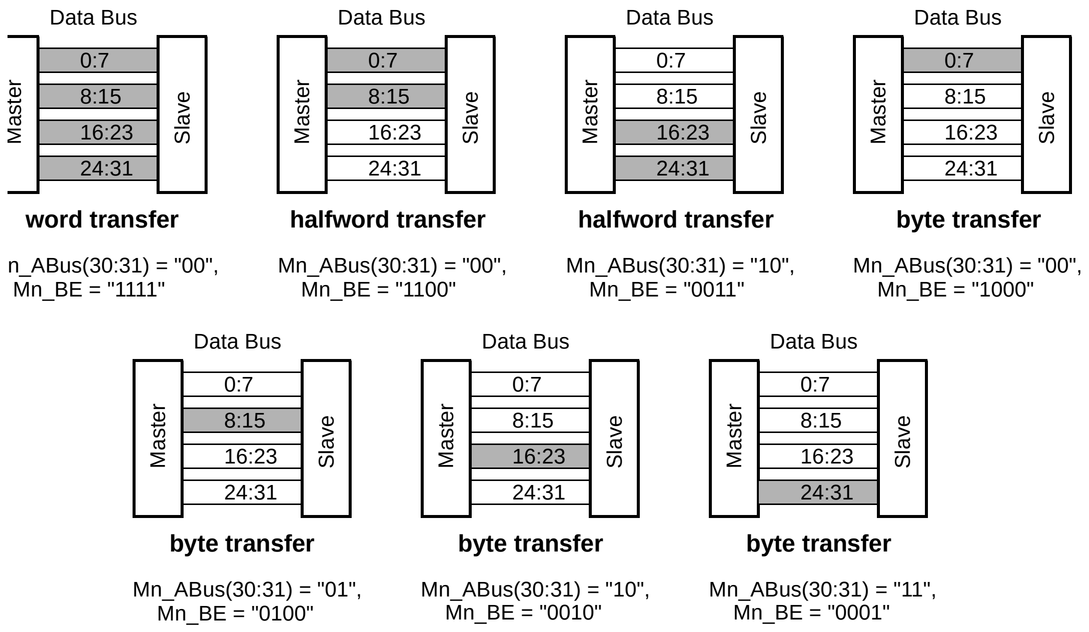

Data Bus
Master
Slave
0:7
8:15
16:23
24:31
word transfer
n_ABus(30:31) = "00", Mn_BE = "1111"
halfword transfer
Mn_ABus(30:31) = "00", Mn_BE = "1100"
halfword transfer
Mn_ABus(30:31) = "10", Mn_BE = "0011"
byte transfer
Mn_ABus(30:31) = "00", Mn_BE = "1000"
byte transfer
Mn_ABus(30:31) = "01", Mn_BE = "0100"
byte transfer
Mn_ABus(30:31) = "10", Mn_BE = "0010"
byte transfer
Mn_ABus(30:31) = "11", Mn_BE = "0001"

Figure 1-1: **Byte lane usage for aligned transfers**

- All OPB slave devices that require a continuous address space (use of all byte lanes) will implement an attachment to the OPB bus that is as wide as the OPB data width, regardless of device width. This eliminates the need for left justification on the OPB bus and eliminates the need for masters to mirror write data.

  As an example, consider an 8-bit memory device that must be addressed at consecutive byte addresses being attached to a 32-bit OPB. The 8-bit memory device must implement a 32-bit wide attachment to the OPB; in the bus attachment, data is steered from the proper byte lane into the 8-bit device for writes, and from the 8-bit device onto the proper byte lane for reads.

  The simplest way to accomplish this is with a multiplexer for steering the writes, and a connection from the 8-bit device to all byte lanes (essentially mirroring to all byte lanes) for reads.
- By convention, registers in all OPB slave devices are aligned to word boundaries (lowest two address bits are "00"), regardless of the size of the data in the register or the size of the peripheral.
- Master and Slave I/O: OPB masters adhere to the signal set shown in Table 1-1. OPB slaves adhere to the signal set shown in Table 1-2. Devices that are both master and slave adhere to the signal set shown in Table 1-3. Page numbers referenced in the tables apply to both the OPB V2.0 specification and the OPB V2.1 specification, both from IBM. All signals shown must be present, except for the one signal shown as optional (<Master>_DBus[0:31] for devices that are both master and slave). No additional signals for OPB interconnection may be added. The naming convention is as follows: <Master> represents a master name or acronym that starts with an upper-case letter, <Slave> represents a slave name or acronym that starts with an upper-case letter. <nOPB> represents an OPB identifier (for masters or slaves with more than OPB attachment) and must start with an uppercase letter and end with upper-case "OPB". For devices with a single OPB attachment, the <nOPB> identifier should default to "OPB" (for example, OPB_ABus). All other parts of the signal name must be

referenced exactly as shown (including case).

Table 1-1: **Summary of OPB master-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Master>_ABus[0:31]	O	Master address bus	OPB-11
<Master>_BE[0:3]	O	Master byte enables	OPB-16
<Master>_busLock	O	Master buslock	OPB-9
<Master>_DBus[0:31]	O	Master write data bus	OPB-13
<Master>_request	O	Master bus request	OPB-8
<Master>_RNW	O	Master read, not write	OPB-12
<Master>_select	O	Master select	OPB-12
<Master>_seqAddr	O	Master sequential address	OPB-13
<nOPB>_DBus[0:31]	I	OPB read data bus	OPB-13
<nOPB>_errAck	I	OPB error acknowledge	OPB-15
<nOPB>_MGrant	I	OPB bus grant	OPB-9
<nOPB>_retry	I	OPB bus cycle retry	OPB-10
<nOPB>_timeout	I	OPB timeout error	OPB-10
<nOPB>_xferAck	I	OPB transfer acknowledge	OPB-14

Table 1-2: **Summary of OPB Slave-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Slave>_DBus[0:31]	O	Slave data bus	OPB-11
<Slave>_errAck	O	Slave error acknowledge	OPB-15
<Slave>_retry	O	Slave retry	OPB-10
<Slave>_toutSup	O	Slave timeout suppress	OPB-15
<Slave>_xferAck	O	Slave transfer acknowledge	OPB-14
<nOPB>_ABus[0:31]	I	OPB address bus	OPB-11
<nOPB>_BE	I	OPB byte enable	OPB-16
<nOPB>_DBus[0:31]	I	OPB data bus	OPB-13
<nOPB>_RNW	I	OPB read/not write	OPB-12
<nOPB>_select	I	OPB select	OPB-12
<nOPB>_seqAddr	I	OPB sequential address	OPB-13

Table 1-3: **Summary of OPB Master/Slave Device I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nOPB>_Clk	I	OPB Clock	
<nOPB>_Rst	I	OPB Reset	
<Master>_ABus[0:31]	O	Master address bus	OPB-11
<Master>_BE[0:3]	O	Master byte enables	OPB-16
<Master>_busLock	O	Master buslock	OPB-9
<Master>_DBus[0:31]	O	Master write data bus (optional)	OPB-13
<Master>_request	O	Master bus request	OPB-8
<Master>_RNW	O	Master read, not write	OPB-12
<Master>_select	O	Master select	OPB-12
<Master>_seqAddr	O	Master sequential address	OPB-13
<nOPB>_DBus[0:31]	I	OPB read data bus	OPB-13
<nOPB>_errAck	I	OPB error acknowledge	OPB-15
<nOPB>_MGrant	I	OPB bus grant	OPB-9
<nOPB>_retry	I	OPB bus cycle retry	OPB-10
<nOPB>_timeout	I	OPB timeout error	OPB-10
<nOPB>_xferAck	I	OPB transfer acknowledge	OPB-14
<Slave>_DBus[0:31]	O	Slave data bus (may optionally function as master write data bus if <Master>_DBus not present)	OPB-11
<Slave>_errAck	O	Slave error acknowledge	OPB-15
<Slave>_retry	O	Slave retry	OPB-10
<Slave>_toutSup	O	Slave timeout suppress	OPB-15
<Slave>_xferAck	O	Slave transfer acknowledge	OPB-14
<nOPB>_ABus[0:31]	I	OPB address bus	OPB-11
<nOPB>_BE	I	OPB byte enable	OPB-16
<nOPB>_RNW	I	OPB read/not write	OPB-12
<nOPB>_select	I	OPB select	OPB-12
<nOPB>_seqAddr	I	OPB sequential address	OPB-13

Additional Notes on Signal Sets

- Xilinx-developed OPB devices do not support dynamic bus sizing and consequently *do not* use the following legacy signals: Mn_dwXfer, Mn_fwXfer, Mn_hwXfer, Sln_dwAck, Sln_fwAck, and Sln_hwAck.
- Since Xilinx-developed OPB devices are byte-enable only, the Mn_beXfer and Sln_beAck signals are not required and so are not used.

- The signals required for masters and slaves are separate from the signals present in the OPB interconnect. The OPB interconnect (the OR gates and other logic required to connect OPB devices) supports the full OPB V2.1 specification (i.e. all signals are present). Thus the OPB interconnect does not limit a design to byte-enable devices and supports designs in which a mix of byte-enable, legacy, and OPB V2.0 devices are present. The bus interconnect does not limit the use of any feature of the V2.1 specification.

Legacy OPB Devices

Although byte-enable devices are the preferred and most efficient OPB devices in Xilinx devices, some designs may also use legacy OPB devices or fully V2.0 compliant devices. However, a legacy device cannot communicate directly with a byte-enable device because they use different signal sets. An interface layer between the byte-enable device and the legacy device is required. This interface is called the Byte Enable Interface (BEIF) device.

Mixed Systems

The system shown in the following figure represents a design with a mix of byte-enable, legacy, and OPB V2.0 devices. The BEIF device converts the legacy-type signals to byte-enable-type signals and vice versa.

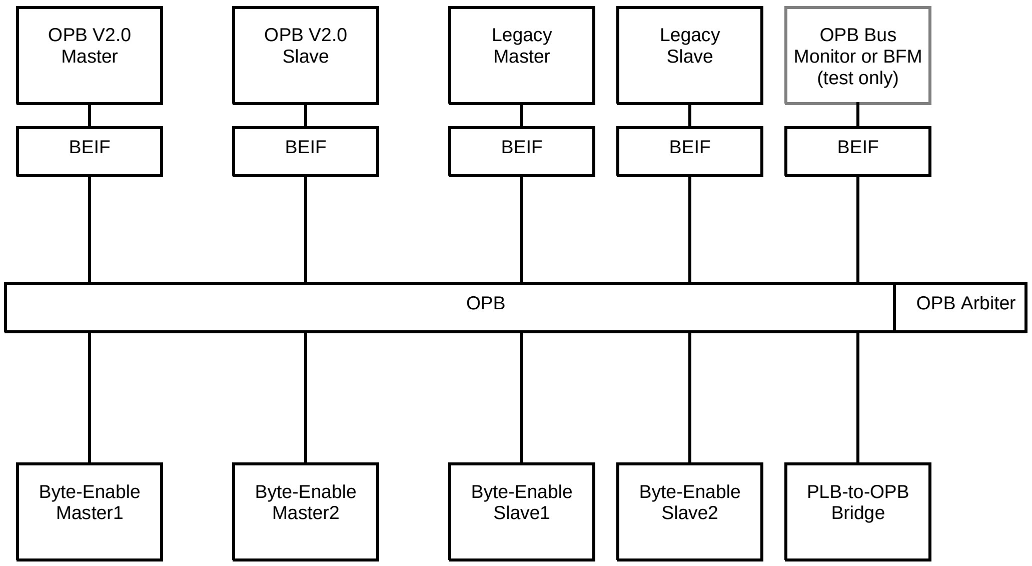


Figure 1-2: **OPB Interconnect with Mixed Device Types**

The BEIF device contains the following logic, not all of which must be used in all situations:

- Signal translation for byte-enable device to legacy device transfers: **<Master>_BE** is translated to the appropriate <Master>_hwXfer, <Master>_fwXfer, and <Master>_dwXfer. **<nOPB>_BE** is translated to the appropriate <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer. <Slave>_hwXfer, <Slave>_fwXfer, and <Slave>_dwXfer are translated to **<Slave>_xferAck**. <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer are translated to **<nOPB>_xferAck**. The correct lower address bits are also generated.
- Signal translation for legacy device to byte-enable device transfers: <Master>_hwXfer, <Master>_fwXfer, and <Master>_dwXfer are translated to **<Master>_BE** . <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer are translated to **<nOPB>_BE** . **<Slave>_xferAck** is translated to <Slave>_hwXfer, <Slave>_fwXfer, and <Slave>_dwXfer. **<nOPB>_xferAck** is translated to <nOPB>_hwXfer, <nOPB>_fwXfer, and <nOPB>_dwXfer.
- Mirroring and steering logic.
- Conversion cycle generator for byte-enable device to legacy device transfers.

With this architecture, systems that do not require full V2.1 features (for example, systems that contain only Xilinx IP) do not need to instantiate the BEIF and hence optimally use the available FPGA resources. Systems that require legacy or OPB V2.0 devices must instantiate the BEIF, although the most costly part of the BEIF (the conversion cycle generator) only needs to be instantiated if conversion cycles are possible (not all slaves will cause generation of conversion cycles).

OPB Usage Notes

The following are general notes on OPB usage that apply primarily to mixed systems:

- Conversion cycles are only required when a master generates a transfer request to a slave that is larger than the slave's width *and* the slave is capable of indicating that it accepted a smaller transfer than the master requested hence requiring with a conversion cycle.
- Byte-enable masters cannot directly generate conversion cycles. They require a conversion cycle generator in the Byte Enable Interface (BEIF) device. This is because byte-enable masters do not receive any size information in the acknowledge from the slave.
- Byte-enable slaves cannot cause generation of conversion cycles. A consequence of this is that any master accessing a byte-enable slave can only transfer data up to the size of the slave. Transfers larger than the slave size will result in either 1) no response from the slave (time-out), 2) an errAck from the slave, or 3) lost data; the actual result depends on how the decode and acknowledge logic is implemented in the slave.
- Conversion cycle generator logic in the BEIF is required only for byte-enable device to legacy/OPB V2.0 device transfers.
- Write mirroring and read steering in the V2.1 specification is based on left-justified peripherals. A more complex slave attachment can be used instead of left justification.

OPB Comparison

Table 1-4 illustrates the major embedded processor bus architectures used in Xilinx FPGAs and lists some of their characteristics. Each bus has different capabilities in terms of data transfer rates, multi-master capability, and data bursting. The use of a particular bus is dictated by the processor used, the data bandwidth required in the application, and availability of peripherals. The OPB is a general-purpose peripheral bus that can be effectively used in many design situations.

PLB - Processor Local Bus (IBM). PLB Reference

OPB - On-chip Peripheral Bus (IBM). OPB Reference

OCM - On-chip Memory interface (IBM). OCM Reference

LMB - Local Memory Bus (Xilinx). *MicroBlaze Processor Reference Guide*

DCR - Device Control Register bus (IBM). DCR Reference

Table 1-4: **Comparison of buses used in Xilinx Embedded Processor Systems**

Feature	CoreConnect Buses			Other Buses	
	PLB	OPB	DCR	OCM	LMB
Processor family	PPC405	PPC405, MicroBlaze	PPC405	PPC405	MicroBlaze
Data bus width	64	32	32	32	32
Address bus width	32	32	10	32	32
Clock rate, MHz (max)[1]	100	125	125	375	125
Masters (max)	16	16	1	1	1
Masters (typical)	2-8	2-8	1	1	1
Slaves (max)	limited only by hardware resources			1	1
Slaves (typical)	2-6	2-8	1-8	1	1
Data rate (peak)[2]	1600 MB/s	500 MB/s	500 MB/s	500 MB/s	500 MB/s

Table 1-4: **Comparison of buses used in Xilinx Embedded Processor Systems** *(Continued)*

Feature	CoreConnect Buses			Other Buses	
	PLB	OPB	DCR	OCM	LMB
Data rate (typical)[3]	533 MB/s[4]	167 MB/s[5]	100 MB/s[8]	333 MB/s[6]	333 MB/s[7]
Concurrent read/write	Yes	No	No	No	No
Address pipelining	Yes	No	No	No	No
Bus locking	Yes	Yes	No	No	No
Retry	Yes	Yes	No	No	No
Timeout	Yes	Yes	No	No	No
Fixed burst	Yes	No	No	No	No
Variable burst	Yes	No	No	No	No
Cache fill	Yes	No	No	No	No
Target word first	Yes	No	No	No	No
FPGA resource usage	High	Medium	Low	Low	Low
Compiler support for load/store	Yes	Yes	No	Yes	Yes

Notes:

1. Maximum clock rates are estimates and are presented for comparison only. The actual maximum clock rate for each bus is dependent on device family, device speed grade, design complexity, and other factors.
2. Peak data rate is the maximum theoretical data transfer rate at the clock rate shown for each bus.
3. The typical data rates are intended to illustrate data rates that are representative of actual system configurations. The typical data is highly dependent on the application software and system hardware configuration.
4. Assumes primarily cache-line fills, minimal read/write concurrency (66.7% bus utilization).
5. Assumes minimal use of sequential address capabilities and 3 clock cycles per OPB transfer.
6. The OCM controller operates at the PPC405 core clock rate, but its data transfer rate is limited by the access time of the on-chip memory. The typical data rate assumes 66.7% bus utilization.
7. Assumes 66.7% bus utilization.
8. Assumes DCR operates at same clock rate as PLB and each DCR access requires 5 clock cycles. The number of clock cycles per DCR transfer is dependent on how many DCR devices are present in the system. Each additional DCR device adds latency to all DCR transfers.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/17/01	1.0	Initial Xilinx version.
10/19/01	1.1	Minor editorial changes. Added links to bus references.
12/10/01	1.2	Changed Figure 2 and other minor edits.
3/20/02	1.3	Updated for MDK 2.2

PLB Usage in Xilinx FPGAs

Summary

This chapter describes how to use the IBM Processor Local Bus (PLB) in Xilinx FPGAs, and provides guidelines and simplifications for efficient FPGA implementations, and the set of signals used in Xilinx-developed PLB devices.

This chapter includes the following sections:

Overview

For detailed information on the IBM PLB, refer to IBM's *64-bit Processor Local Bus, Architecture Specifications, Version 3.5*:

PlbBus.pdf

The PLB is one element of IBM's CoreConnect architecture, and is a high-performance synchronous bus designed for connection of processors to high-performance peripheral devices. The PLB includes the following features (from *64-bit Processor Local Bus, Architecture Specifications*):

- Overlapping of read and write transfers allows two data transfers per clock cycle for maximum bus utilization.
- Decoupled address and data buses support split-bus transaction capability for improved bandwidth.
- Address pipelining reduces overall bus latency by allowing the latency associated with a new request to be overlapped with an ongoing data transfer in the same direction.
- Late master request abort capability reduces latency associated with aborted requests.
- Hidden (overlapped) bus request/grant protocol reduces arbitration latency.
- Bus architecture supports sixteen masters and any number of slave devices.
- Four levels of request priority for each master allow PLB implementations with various arbitration schemes.
- Bus arbitration-locking mechanism allows for master-driven atomic operations.
- Support for 16-, 32-, and 64-byte line data transfers.
- Read word address capability allows slave devices to fetch line data in any order (that is, target word-first or sequential).
- Sequential burst protocol allows byte, halfword, and word burst data transfers in either direction.

- Guarded and unguarded memory transfers allow a slave device to enable or disable the pre-fetching of instructions or data.

The PLB is a full-featured bus architecture with many features that increase bus performance. Most of these features map well to the FPGA architecture, however, some can result in the inefficient use of FPGA resources or can lower system clock rates. Consequently, Xilinx uses an efficient *subset* of the PLB for Xilinx-developed PLB devices. However, because of the flexible nature of FPGAs, you can also implement systems utilizing PLB devices that are fully PLB V3.5 compliant.

Xilinx PLB Usage

Dynamic Bus Sizing

Dynamic bus sizing is a PLB architectural feature that allows a designer to mix 32 and 64-bit devices on the same 64-bit PLB. A master provides a master size signal, <Master>_MSize[0:1], that describes the data width of the master initiating a transaction. Slaves provide a similar signal, Sl_Mn_SSize(0:1), with the address acknowledge that describes the data width of the slave that is responding to the transaction. While dynamic bus sizing is a useful architectural feature, its use in FPGAs can result in inefficient implementations of PLB masters.

Conversion Cycles

Dynamic bus sizing results in *conversion cycles*, which are extra transfer cycles that re-transfer data when the master-initiated transfer is larger than the slave response. For example, if a master writes a 64-bit word to a slave, and the 32-bit device slave responds with a slave size of 32-bits, then the master must perform an additional conversion cycle to transfer all of the data to the slave. Generating conversion cycles requires more logic, increases the complexity of the master, and is typically not an efficient use of FPGA resources.

Write Mirroring and Read Steering

Another consequence of supporting devices smaller than the bus size is *write mirroring* and *read steering*. In the PLB specification, devices smaller than the bus size are always left-justified (aligned toward the most significant side of the bus) so that the byte lanes associated with the smaller devices are easily determined. For example, a word-wide peripheral is always located on the most-significant word of the 64-bit bus. The peripheral writes and reads data using only the four most significant byte lanes. You can simplify the design of PLB masters by using an architecture that requires no write mirroring and transfers data based on which byte enables are active. A small degree of added complexity is required in the bus attachment for peripherals that are smaller than the bus size if PLB masters do not mirror data. This additional logic is built into the parameterizable slave attachment in each Xilinx peripheral.

Xilinx PLB Devices

Ideal FPGA Implementation of PLB-based System

The ideal FPGA implementation of a PLB-based system has the following features:

- Requires no conversion cycles
- Does not require masters to mirror write data

These characteristics help determine how Xilinx-developed PLB devices are implemented. The detailed specifications that describe how the PLB is used in Xilinx intellectual property are provided in the next section.

Specifications for PLB Usage in Xilinx-developed PLB Devices

Xilinx-developed PLB devices adhere to the following PLB usage rules:

- The width of the PLB data buses is 64 bits and the width of address buses is 32 bits. Note that some peripherals may parameterize these widths, but currently only 64-bit data buses are supported. Peripherals that are smaller than 64-bits can be attached to the PLB with a corresponding restriction in addressing. For example, a 32-bit peripheral at base address A can be attached to byte lanes 0 – 4, but word-wide accesses can only be addressed at A, A+8, A+16, etc.
- PLB masters are not required to support dynamic bus sizing. PLB masters are not required to mirror data to unused byte lanes. See Figure 2-1 and Figure 2-2 for the byte lane usage for aligned transfers. PLB Masters are required to correctly drive the <Master>_MSize[0:1] signals. PLB slaves are required to correctly drive the <Slave>_SSize[0:1] signals for PLB masters that do provide conversion cycles (such as the PowerPC 405).
- All PLB slaves are required to output logic zero when they are inactive.
- The byte-enables and the least-significant address bits are driven by all masters and contain consistent information. Examples of byte lane usage for aligned transfers are shown in Figure 2-1 and Figure 2-2.

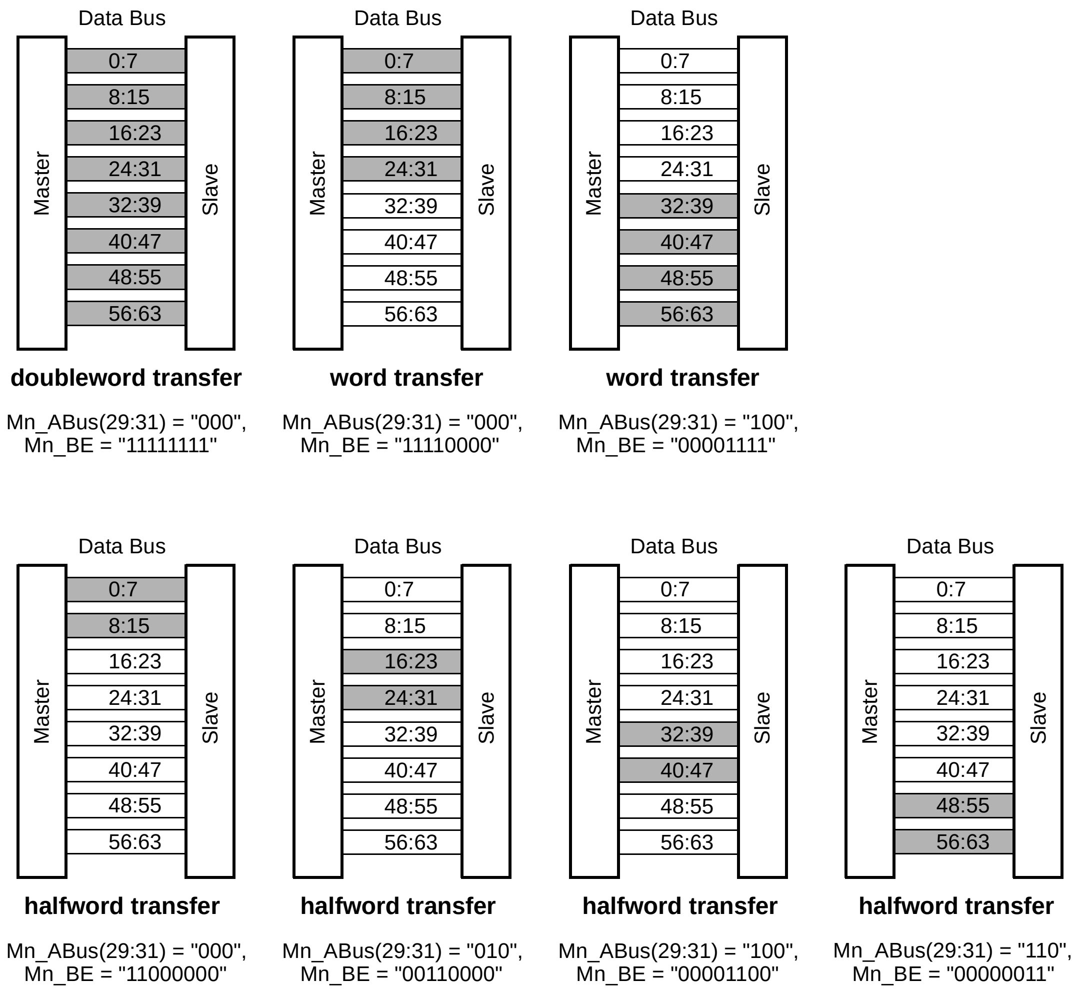

Data Bus
Master
Slave
0:7
8:15
16:23
24:31
32:39
40:47
48:55
56:63
doubleword transfer
Mn_ABus(29:31) = "000", Mn_BE = "11111111"
word transfer
Mn_ABus(29:31) = "000", Mn_BE = "11110000"
word transfer
Mn_ABus(29:31) = "100", Mn_BE = "00001111"
halfword transfer
Mn_ABus(29:31) = "000", Mn_BE = "11000000"
halfword transfer
Mn_ABus(29:31) = "010", Mn_BE = "00110000"
halfword transfer
Mn_ABus(29:31) = "100", Mn_BE = "00001100"
halfword transfer
Mn_ABus(29:31) = "110", Mn_BE = "00000011"

Figure 2-1: **Byte lane usage for aligned doubleword, word, and halfword transfers**

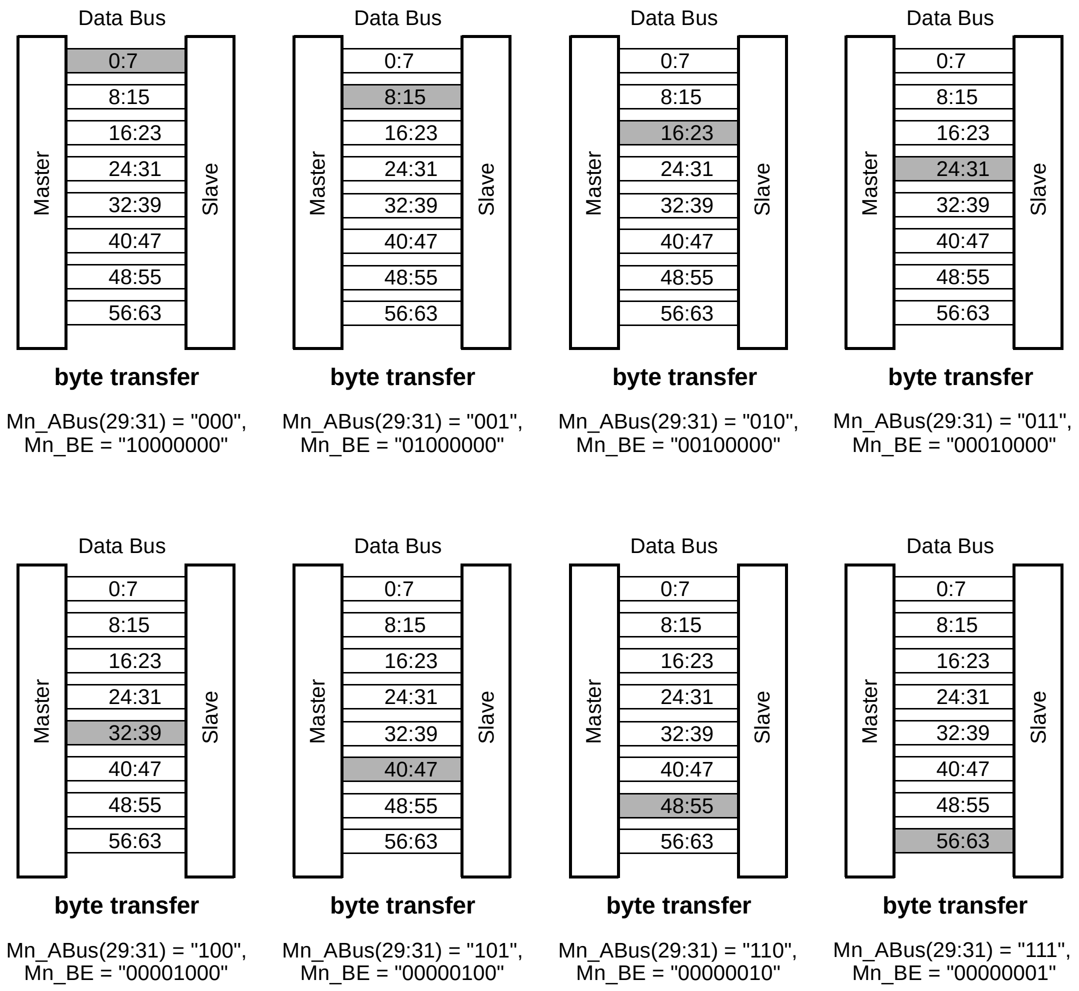

Data Bus
Master
Slave
0:7
8:15
16:23
24:31
32:39
40:47
48:55
56:63
byte transfer
Mn_ABus(29:31) = "000", Mn_BE = "10000000"
Mn_ABus(29:31) = "001", Mn_BE = "01000000"
Mn_ABus(29:31) = "010", Mn_BE = "00100000"
Mn_ABus(29:31) = "011", Mn_BE = "00010000"
Mn_ABus(29:31) = "100", Mn_BE = "00001000"
Mn_ABus(29:31) = "101", Mn_BE = "00000100"
Mn_ABus(29:31) = "110", Mn_BE = "00000010"
Mn_ABus(29:31) = "111", Mn_BE = "00000001"

Figure 2-2: **Byte lane usage for byte transfers**

- All PLB slave devices that require a continuous address space (i.e. use of all byte lanes) will implement an attachment to the PLB bus that is as wide as the PLB data width, regardless of device width. This eliminates the need for left justification on the PLB bus and eliminates the need for masters to mirror write data. As an example, consider a 32-bit memory device that must be addressed at consecutive byte addresses being attached to a 64-bit PLB. The 32-bit memory device must implement a 64-bit wide attachment to the PLB; in the bus attachment, data is steered from the proper byte lanes into the 32-bit device for writes, and from the 32-bit device onto the proper byte lanes for reads.
- By convention, registers in all PLB slave devices are aligned to word boundaries (lowest two address bits are "00"), regardless of the size of the data in the register or the size of the peripheral.
- Master and Slave I/O: PLB masters adhere to the signal set shown in Table 2-1. PLB slaves adhere to the signal set shown in Table 2-2. Page numbers referenced in the tables apply to the PLB V3.5 specification from IBM. All signals shown must be present. No additional signals for PLB interconnection may be added. The naming convention is as follows: <Master> represents a master name or acronym that starts with an upper-case letter, <Slave> represents a slave name or acronym that starts with an upper-case letter. <nPLB> represents an PLB identifier (for masters or slaves with more than one PLB attachment) and must start with an uppercase letter and end with upper-case "PLB". For devices with a single PLB attachment, the <nPLB> identifier

should default to "PLB" (for example, PLB_ABus). All other parts of the signal name must be referenced exactly as shown (including case).

Table 2-1: **Summary of PLB Master-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nPLB>_Clk	I	PLB Clock (SYS_plbClk)	PLB-11
<nPLB>_Rst	I	PLB Reset (SYS_plbReset)	PLB-11
<Master>_abort	O	Master abort bus request indicator	PLB-19
<Master>_ABus[0:31]	O	Master address bus	PLB-27
<Master>_BE[0:7]	O	Master byte enables	PLB-21
<Master>_busLock	O	Master buslock	PLB-13
<Master>_compress	O	Master compressed data transfer indicator	PLB-25
<Master>_guarded	O	Master guarded transfer indicator	PLB-26
<Master>_lockErr	O	Master lock error indicator	PLB-27
<Master>_MSize[0:1]	O	Master data bus size	PLB-40
<Master>_ordered	O	Master synchronize transfer indicator	PLB-26
<Master>_priority[0:1]	O	Master request priority	PLB-12
<Master>_rdBurst	O	Master burst read transfer indicator	PLB-34
<Master>_request	O	Master request	PLB-12
<Master>_RNW	O	Master read/not write	PLB-21
<Master>_size[0:3]	O	Master transfer size	PLB-24
<Master>_type[0:2]	O	Master transfer type	PLB-25
<Master>_wrBurst	O	Master burst write transfer indicator	PLB-29
<Master>_wrDBus[0:63]	O	Master write data bus	PLB-28
<nPLB>_<Master>_Busy	I	PLB master slave busy indicator	PLB-36
<nPLB>_<Master>_Err	I	PLB master slave error indicator	PLB-37
<nPLB>_<Master>_WrBTerm	I	PLB master terminate write burst indicator	PLB-30
<nPLB>_<Master>_WrDAck	I	PLB master write data acknowledge	PLB-29
<nPLB>_<Master>AddrAck	I	PLB master address acknowledge	PLB-18
<nPLB>_<Master>RdBTerm	I	PLB master terminate read burst indicator	PLB-36
<nPLB>_<Master>RdDAck	I	PLB master read data acknowledge	PLB-33

Table 2-1: **Summary of PLB Master-only I/O** ***(Continued)***

Signal	I/O	Description	Page (in Ref. 1)
<nPLB>_<Master>RdDBus[0:63]	I	PLB master read data bus	PLB-31
<nPLB>_<Master>RdWdAddr[0:3]	I	PLB master read word address	PLB-32
<nPLB>_<Master>Rearbitrate	I	PLB master bus re-arbitrate indicator	PLB-19
<nPLB>_<Master>SSize[0:1]	I	PLB slave data bus size	PLB-40

Table 2-2: **Summary of PLB Slave-only I/O**

Signal	I/O	Description	Page (in Ref. 1)
<nPLB>_Clk	I	PLB Clock (SYS_plbClk)	PLB-11
<nPLB>_Reset	I	PLB Reset (SYS_plbReset)	PLB-11
<Slave>_addrAck	O	Slave address acknowledge	PLB-18
<Slave>_MBusy[0:3]	O	Slave busy indicator	PLB-36
<Slave>_MErr[0:3]	O	Slave error indicator	PLB-37
<Slave>_rdBTerm	O	Slave terminate read burst transfer	PLB-36
<Slave>_rdComp	O	Slave read transfer complete indicator	PLB-34
<Slave>_rdDAck	O	Slave read data acknowledge	PLB-33
<Slave>_rdDBus[0:63]	O	Slave read data bus	PLB-31
<Slave>_rdWdAddr[0:3]	O	Slave read word address	PLB-32
<Slave>_rearbitrate	O	Slave re-arbitrate bus indicator	PLB-19
<Slave>_SSize[0:1]	O	Slave data bus size	PLB-40
<Slave>_wait	O	Slave wait indicator	PLB-18
<Slave>_wrBTerm	O	Slave terminate write burst transfer	PLB-30
<Slave>_wrComp	O	Slave write transfer complete indicator	PLB-29
<Slave>_wrDAck	O	Slave write data acknowledge	PLB-29
<nPLB>_abort	I	PLB abort request indicator	PLB-19
<nPLB>_ABus[0:31]	I	PLB address bus	PLB-27
<nPLB>_BE[0:7]	I	PLB byte enables	PLB-21
<nPLB>_busLock	I	PLB bus lock	PLB-13
<nPLB>_compress	I	PLB compressed data transfer indicator	PLB-25
<nPLB>_guarded	I	PLB guarded transfer indicator	PLB-26
<nPLB>_lockErr	I	PLB lock error indicator	PLB-27
<nPLB>_masterID[0:1]	I	PLB current master identifier	PLB-20

Table 2-2: **Summary of PLB Slave-only I/O** *(Continued)*

Signal	I/O	Description	Page (in Ref. 1)
<nPLB>_MSize[0:1]	I	PLB master data bus size	PLB-40
<nPLB>_ordered	I	PLB synchronize transfer indicator	PLB-26
<nPLB>_PAValid	I	PLB primary address valid indicator	PLB-13
<nPLB>_pendPri[0:1]	I	PLB pending request priority	PLB-20
<nPLB>_pendReq	I	PLB pending bus request indicator	PLB-20
<nPLB>_rdBurst	I	PLB burst read transfer indicator	PLB-34
<nPLB>_rdPrim	I	PLB secondary to primary read request indicator	PLB-36
<nPLB>_reqPri[0:1]	I	PLB current request priority	PLB-20
<nPLB>_RNW	I	PLB read/not write	PLB-21
<nPLB>_SAValid	I	PLB secondary address valid indicator	PLB-16
<nPLB>_size[0:3]	I	PLB transfer size	PLB-24
<nPLB>_type[0:2]	I	PLB transfer type	PLB-25
<nPLB>_wrBurst	I	PLB burst write transfer indicator	PLB-29
<nPLB>_wrDBus[0:63]	I	PLB write data bus	PLB-28
<nPLB>_wrPrim	I	PLB secondary to primary write request indicator	PLB-31

PLB Comparison

Table 2-3 illustrates the major embedded processor bus architectures used in Xilinx FPGAs and lists some of their characteristics. Each bus has different capabilities in terms of data transfer rates, multi-master capability, and data bursting. The use of a particular bus is dictated by the processor used, the data bandwidth required in the application, and availability of peripherals. The PLB is a high-performance local bus that can be effectively used in many design situations.

PLB - Processor Local Bus (IBM). **PLB Reference**

OPB - On-chip Peripheral Bus (IBM). **OPB Reference**

OCM - On-chip Memory interface (IBM). **OCM Reference**

DCR - Device Control Register bus (IBM). **DCR Reference**

Table 2-3: **Comparison of buses used in Xilinx embedded processor systems**

Feature	CoreConnect Buses			Other Buses	
	PLB	OPB	DCR	OCM	LMB
Processor family	PPC405	PPC405, MicroBlaze	PPC405	PPC405	MicroBlaze
Data bus width	64	32	32	32	32
Address bus width	32	32	10	32	32
Clock rate, MHz (max)[1]	100	125	125	375	125
Masters (max)	16	16	1	1	1
Masters (typical)	2-8	2-8	1	1	1
Slaves (max)	limited only by hardware resources			1	1
Slaves (typical)	2-6	2-8	1-8	1	1
Data rate (MB/s, peak)[2]	1600	500	500	500	500
Data rate (MB/s, typical)[3]	533[4]	167[5]	100[8]	333[6]	333[7]
Concurrent read/write	Yes	No	No	No	No
Address pipelining	Yes	No	No	No	No
Bus locking	Yes	Yes	No	No	No
Retry	Yes	Yes	No	No	No
Timeout	Yes	Yes	No	No	No
Fixed burst	Yes	No	No	No	No
Variable burst	Yes	No	No	No	No
Cache fill	Yes	No	No	No	No
Target word first	Yes	No	No	No	No
FPGA resource usage	High	Medium	Low	Low	Low
Compiler support for load/store	Yes	Yes	No	Yes	Yes

Notes:

1. Maximum clock rates are estimates and are presented for comparison only. The actual maximum clock rate for each bus is dependent on device family, device speed grade, design complexity, and other factors.
2. Peak data rate is the maximum theoretical data transfer rate at the clock rate shown for each bus.
3. The typical data rates are intended to illustrate data rates that are representative of actual system configurations. The typical data is highly dependent on the application software and system hardware configuration.
4. Assumes primarily cache-line fills, minimal read/write concurrency (66.7% bus utilization).
5. Assumes minimal use of sequential address capabilities and 3 clock cycles per OPB transfer.
6. The OCM controller operates at the PPC405 core clock rate, but its data transfer rate is limited by the access time of the on-chip memory. The typical data rate assumes 66.7% bus utilization.
7. Assumes 66.7% bus utilization.
8. Assumes DCR operates at same clock rate as PLB and each DCR access requires 5 clock cycles. The number of clock cycles per DCR transfer is dependent on how many DCR devices are present in the system. Each additional DCR device adds latency to all DCR transfers.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
5/8/02	1.0	Initial Xilinx version.

Chapter 3

Bus Infrastructure Cores

This section of the reference guide contains information on the following bus infrastructure cores:

IPIF

This section of the reference guide contains information on the following:

Memory Interface Cores

This section of the reference guide contains information on the following memory interface cores:

Peripheral Cores

This section of the reference guide contains information on the following peripheral cores:

Part II: Software

This section contains information on the following:

Chapter 7

Device Driver Programmer Guide

Overview

This document describes the Xilinx device driver environment, and includes information on the following:

- Design and implementation details for using the drivers
- Device driver architecture
- Application Programmer Interface (API) conventions
- Scheme for configuring the drivers to work with reconfigurable hardware devices
- Infrastructure that is common to all device drivers.

Goals and Objectives

The Xilinx device drivers are designed to meet the following goals and objectives:

- Provide maximum portability

  The device drivers are provided as ANSI C source code. ANSI C was chosen to maximize portability across processors and development tools. Source code is provided both to aid customers in debugging their applications as well as allow customers to modify or optimize the device driver if necessary.

  A layered device driver architecture additionally separates device communication from processor and Real Time Operating System (RTOS) dependencies, thus providing portability of core device driver functionality across processors and operating systems.

- Support FPGA configurability

  Since FPGA-based devices can be parameterized to provide varying functionality, the device drivers must support this varying functionality. The configurability of device drivers should be supported at compile-time and at run-time. Run-time configurability provides the flexibility needed for future dynamic system reconfiguration.

  In addition, a device driver supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per instance basis.

- Support simple and complex use cases

  Device drivers are needed for simple tasks such as board bring-up and testing, as well as complex embedded system applications. A layered device driver architecture provides both simple device drivers with minimal memory footprints and more robust, full-featured device drivers with larger memory footprints.

- Ease of use and maintenance

Xilinx makes use of coding standards and provides well-documented source code in order to give developers (i.e., customers and internal development) a consistent view of source code that is easy to understand and maintain. In addition, the API for all device drivers is consistent to provide customers a similar look and feel between drivers.

Device Driver Architecture

The architecture of the device drivers is designed as a layered architecture as shown in the following figure. The layered architecture accommodates the many use cases of device drivers while at the same time providing portability across operating systems, toolsets, and processors. The layered architecture provides seamless integration with an RTOS (Layer 2), high-level device drivers that are full-featured and portable across operating systems and processors (Layer 1), and low-level drivers for simple use cases (Layer 0). The following paragraphs describe each of the layers. The user can choose to use any and all layers.

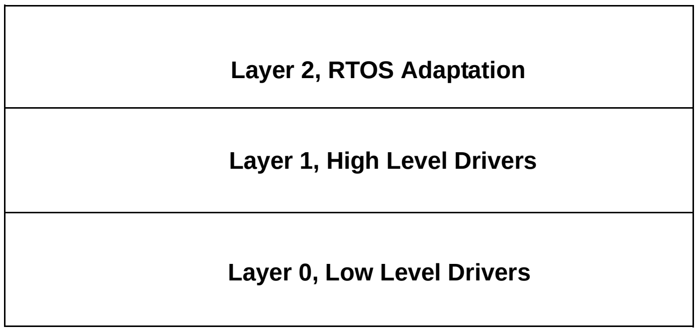


Figure 7-1: **Layered Architecture**

Layer 2, RTOS Adaptation

This layer consists of adapters for device drivers. An adapter converts a Layer 1 device driver interface to an interface that matches the requirements of the device driver scheme for an RTOS. Unique adapters may be necessary for each RTOS. Adapters typically have the following characteristics.

- Communicates directly to the RTOS and the Layer 1, high-level driver.
- References functions and identifiers specific to the RTOS. This layer is therefore not portable across operating systems.
- Can use memory management
- Can use RTOS services such as threading, inter-task communication, etc.
- Can be simple or complex depending on the RTOS interface and requirements for the device driver

Layer 1, High Level Drivers

This layer consists of high level device drivers . They are implemented as macros and functions and are designed to allow a developer to utilize all features of a device. These high-level drivers are independent of operating system and processor, making them highly portable. They typically have the following characteristics.

- Consistent and high-level (abstract) API that gives the user an "out-of-the-box" solution
- No RTOS or processor dependencies, making them highly portable

- Run-time error checking such as assertion of input arguments. Also provides the ability to compile away asserts.
- Comprehensive support of device features
- Abstract API that isolates the API from hardware device changes
- Supports device configuration parameters to handle FPGA-based parameterization of hardware devices.
- Supports multiple instances of a device while managing unique characteristics on a per instance basis.
- Polled and interrupt driven I/O
- Non-blocking function calls to aid complex applications
- May have a large memory footprint
- Typically provides buffer interfaces for data transfers as opposed to byte interfaces. This makes the API easier to use for complex applications.
- Does not communicate directly to Layer 2 adapters or application software. Utilizes asynchronous callbacks for upward communication.

Layer 0, Low Level Drivers

This layer consists of low level device drivers. They are implemented as macros and functions and are designed to allow a developer to create a small system, typically for internal memory of an FPGA. They typically have the following characteristics.

- Simple, low-level API
- Small memory footprint
- Little to no error checking is performed
- Supports primary device features only
- Minimal abstraction such that the API typically matches the device registers. The API is therefore less isolated from hardware device changes.
- No support of device configuration parameters
- Supports multiple instances of a device with base address input to the API
- None or minimal state is maintained
- Polled I/O only
- Blocking functions for simple use cases
- Typically provides byte interfaces but can provide buffer interfaces for packet-based devices.

Object-Oriented Device Drivers

In addition to the layered architecture, it is important that the user understand the underlying design of the device drivers. The device drivers are designed using an object-oriented methodology. The methodology is based upon components and is described in the following paragraphs. This approach pertains particularly to the Layer 1, high-level device drivers.

Component Definition

A component is a logical partition of the software which provides a functionality similar to one or more classes in C++. Each component provides a set of functions that operate on the internal data of the component. In general, components are not allowed access to the data of other components. A device driver is typically designed as a single component. A component may consist of one or more files.

Component Implementation

The component contains data variables which define the set of values that instances of that type can hold and a set of functions that operate on those data variables. Components must utilize the functions of other components in order to access the data of other components, rather than accessing component data directly. Components provide data abstraction and encapsulation by gathering the state of an object and the functions that operate on that object into a single unit and by denying direct access to its data members.

Component Data Variables

The primary mechanism for implementing a component in C is the structure. The data variables for a component are grouped in a single structure such that instances of the component each have their own data. The structure and the prototypes for all component functions are declared in the header file which is shared between the implementing component and other components which utilize it. A pointer to this structure, referred to as the instance pointer, is passed into each function of the component which operates on the instance data.

Component Interface

Each component has a set of functions which are collectively referred to as the component interface. Every function of a component which operates on the instance data utilizes a pointer, named InstancePtr, to an instance of a component as the first argument. This argument emulates the *this* pointer in C++ and allows the component function to manipulate the instance data.

Component Instance

An instance of a component is created when a variable is created using the component data type. An instance of a component maps to each physical hardware device. Each instance may have unique characteristics such as it's memory mapped address and specific device capabilities.

Component Example

The following code example illustrates a device driver component.

```
/* the device component data type */

typedef struct
{
    Xuint32 BaseAddress;  /* component data variables */
    Xuint32 IsReady;
    Xuint32 IsStarted;
} XDevice;

/* create an instance of a device */

XDevice DeviceInstance;

/* device component interfaces */

XStatus XDevice_Initialize(XDevice *InstancePtr, Xuint16 DeviceId);
XStatus XDevice_Start(XDevice *InstancePtr);
```

API and Naming Conventions

External Identifiers

External identifiers are defined as those items that are accessible to all other components in the system (global) and include functions, constants, typedefs, and variables.

An 'X' is prepended to each Xilinx external so it does not pollute the global name space, thus reducing the risk of a name conflict with application code. The names of externals are based upon the component in which they exist. The component name is prepended to each external name. An underscore character always separates the component name from the variable or function name.

External Name Pattern:

```
X<component name>_VariableName;
X<component name>_FunctionName(ArgumentType Argument)
X<component name>_TypeName;
```

Constants are typically defined as all uppercase and prefixed with an abbreviation of the component name. For example, a component named XUartLite (for the UART Lite device driver) would have constants that begin with XUL_, and a component named XEmac (for the Ethernet 10/100 device driver) would have constants that begin with XEM_. The abbreviation utilizes the first three uppercase letters of the component name, or the first three letters if there are only two uppercase letters in the component name.

File Naming Conventions

The file naming convention utilizes long file names and is not limited to 8 characters as imposed by the older versions of the DOS operating system.

Component Based Source File Names

Source file names are based upon the name of the component implemented within the source files such that the contents of the source file are obvious from the file name. All file names must begin with the lowercase letter "x" to differentiate Xilinx source files. File extensions .h and .c are utilized to distinguish between header source files and implementation source files.

Implementation Source Files (*.c)

The C source files contain the implementation of a component. A component is typically contained in multiple source files to allow parts of the component to be user selectable.

Source File Naming Pattern:

```
x<component name>.c                      main source file
x<component name>_functionality.c        secondary source file
```

Header Source Files (*.h)

The header files contain the interfaces for a component. There will always be external interfaces which is what an application that utilizes the component invokes.

- The external interfaces for the high level drivers (Layer 1) are contained in a header file with the file name format *x<component name>.h*.
- The external interfaces for the low level drivers (Layer 0) are contained in a header file with the file name format *x<component name>_l.h*.

In the case of multiple C source files which implement the class, there may also be a header file which contains internal interfaces for the class. The internal interfaces allow the functions within each source file to access functions in the another source file.

- The internal interfaces are contained in a header file with the file name format *x<component name>_i.h*.

Device Driver Layers

Layer 1 and Layer 0 device drivers (i.e., high-level and low-level drivers) are typically bundled together in a directory. The Layer 0 device driver files are named *x<component name>_l.h* and *x<component name>_l.c*. The "*_l*" indicates low-level driver. Layer 2 RTOS adapter files include the word "adapter" in the file name, such as *x<component name>_adapter.h* and *x<component name>_adapter.c*. These are typically stored in a different directory name (e.g., one specific to the RTOS) than the device driver files.

Example File Names

The following source file names illustrates an example which is complex enough to utilize multiple C source files.

```
xuartns550.c         Main implementation file
xuartns550_intr.c    Secondary implementation file for interrupt
handling
xuartns550.h         High level external interfaces header file
xuartns550_i.h       Internal identifiers header file
xuartns550_l.h       Low level external interfaces header file
xuartns550_l.c       Low level implementation file
xuartns550_g.c       Generated file controlling parameterized
instances

and,

xuartns550_sio_adapter.c VxWorks Serial I/O (SIO) adapter
```

High Level Device Driver API

High level device drivers are designed to have an API which includes a standard API together with functions that may be unique to that device. The standard API provides a consistent interface for Xilinx drivers such that the effort to use multiple device drivers is minimized. An example API follows.

Standard Device Driver API

Initialize

This function initializes an instance of a device driver. Initialization must be performed before the instance is used. Initialization includes mapping a device to a memory-mapped address and initialization of data structures. It maps the instance of the device driver to a physical hardware device. The user is responsible for allocating an instance variable using the driver's data type, and passing a pointer to this variable to this and all other API functions.

Reset

This function resets the device driver and device with which it is associated. This function is provided to allow recovery from exception conditions. This function resets the device and device driver to a state equivalent to after the Initialize() function has been called.

SelfTest

This function performs a self-test on the device driver and device with which it is associated. The self-test verifies that the device and device driver are functional.

Optional Functions

Each of the following functions may be provided by device drivers.

Start

This function is provided to start the device driver. Starting a device driver typically enables the device and enables interrupts. This function, when provided, must be called prior to other data or event processing functions.

Stop

This function is provided to stop the device driver. Stopping a device driver typically disables the device and disables interrupts.

GetStats

This function gets the statistics for the device and/or device driver.

ClearStats

This function clears the statistics for the device and/or device driver.

InterruptHandler

This function is provided for interrupt processing when the device must handle interrupts. It does not save or restore context. The user is expected to connect this interrupt handler to their system interrupt controller. Most drivers will also provide hooks, or callbacks, for the user to be notified of asynchronous events during interrupt processing (e.g., received data or device errors).

Configuration Parameters

Standard device driver API functions (of Layer 1, high-level drivers) such as Initialize() and Start() require basic information about the device such as where it exists in the system memory map or how many instances of the device there are. In addition, the hardware features of the device may change because of the ability to reconfigure the hardware within the FPGA. Other parts of the system such as the operating system or application may need to know which interrupt vector the device is attached to. For each device driver, this type of information is distributed across two files: *xparameters.h* and *x<component name>_g.c*.

Typically, these files are automatically generated by a system generation tool based on what the user has included in their system. However, these files can be hand coded to support internal development and integration activities. Note that the low-level drivers of Layer 0 do not require or make use of the configuration information defined in these two files. Other than the memory-mapped location of the device, the low-level drivers are typically fixed in the hardware features they support.

xparameters.h

This source file centralizes basic configuration constants for all drivers within the system. Browsing this file gives the user an overall view of the system architecture. The device drivers and Board Support Package (BSP) utilize the information contained here to configure the system at runtime. The amount of configuration information varies by device, but at a minimum the following items should be defined for each device:

- Number of device instances
- Device ID for each instance

A Device ID uniquely identifies each hardware device which maps to a device driver. A Device ID is used during initialization to perform the mapping of a device driver to a hardware device. Device IDs are typically assigned either by the user or by a system generation tool. It is currently defined as a 16-bit unsigned integer.

- Device base address for each instance
- Device interrupt assignment for each instance if interrupts can be generated.

File Format and Naming Conventions

Every device must have the following constant defined indicating how many instances of that device are present in the system (note that `<component name>` does not include the preceding "X"):

```
XPAR_X<component name>_NUM_INSTANCES
```

Each device instance will then have multiple, unique constants defined. The names of the constants typically match the hardware configuration parameters, but can also include other constants. For example, each device instance has a unique device identifier (DEVICE_ID), the base address of the device's registers (BASEADDR), and the end address of the device's registers (HIGHADDR).

```
XPAR_<component name>_<component instance>_DEVICE_ID
XPAR_<component name>_<component instance>_BASEADDR
XPAR_<component name>_<component instance>_HIGHADDR
```

<component instance> is typically a number between 0 and (`XPAR_X<component name>_NUM_INSTANCES` - 1). Note that the system generation tools may create these constants with a different convention than described here. Other device specific constants are defined as needed:

```
XPAR_<component name>_<component instance>_<item description>
```

When the device specific constant applies to all instances of the device:

```
XPAR_<component name>_<item description>
```

For devices that can generate interrupts, a separate section within *xparameters.h* is used to store interrupt vector information. While the device driver implementation files do not utilize this information, their RTOS adapters, BSP files, or user application code will require them to be defined in order to connect, enable, and disable interrupts from that device. The naming convention of these constants varies whether an interrupt controller is part of the system or the device hooks directly into the processor.

For the case where an interrupt controller is considered external and part of the system, the naming convention is as follows:

```
XPAR_INTC_<instance>_<component name>_<component instance>_VEC_ID
```

Where INTC is the name of the interrupt controller component, <instance> is the component instance of the INTC, <component name> and <component instance> is the name and instance number of the component connected to the controller. Of course XPAR_INTC must have the other required constants DEVICE_ID, BASEADDR, etc. This convention supports single and cascaded interrupt controller architectures.

For the case where an interrupt controller is considered internal to a processor, the naming convention changes:

```
XPAR_<proc name>_<component name>_<component instance>_VEC_ID
```

Where <proc name> is the name of the processor.

x<component name>_g.c

The header file *x<component name>.h* defines the type of a configuration structure. The type will contain all of the configuration information necessary for an instance of the device. The format of the data type is as follows:

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;

   /* Other device dependent data attributes */

} X<component name>_Config;
```

The implementation file *x<component name>_g.c* defines an array of structures of `X<component name>_Config` type. Each element of the array represents an instance of the device, and contains most of the per-instance XPAR constants from *xparameters.h*.

Example

To help illustrate the relationships between these configuration files, an example is presented that contains a single interrupt controller whose component name is INTC and a single UART whose component name is (UART). Only xintc.h and xintc_g.c are illustrated, but xuart.h and xuart_g.c would be very similar.

xparameters.h

```
/* Constants for INTC */
XPAR_INTC_NUM_INSTANCES     1
XPAR_INTC_0_DEVICE_ID       21
XPAR_INTC_0_BASEADDR        0xA0000100

/* Interrupt vector assignments for this instance */
XPAR_INTC_0_UART_0_VEC_ID   0

/* Constants for UART */
XPAR_UART_NUM_INSTANCES     1
XPAR_UART_0_DEVICE_ID       2
XPAR_UART_0_BASEADDR        0xB0001000
```

xintc.h

```
typedef struct
{
   Xuint16 DeviceID;
   Xuint32 BaseAddress;
} XIntc_Config;
```

xintc_g.c

```
static XintcConfig[XPAR_INTC_NUM_INSTANCES] =
{
  {
     XPAR_INTC_0_DEVICE_ID,
     XPAR_INTC_0_BASEADDR,
  }
}
```

Common Driver Infrastructure

Source Code Documentation

The comments in the device driver source code contain *doxygen* tags for *javadoc*-style documentation. *Doxygen* is a *javadoc*-like tool that works on C language source code. These tags typically start with "@" and provide a means to automatically generate HTML-based documentation for the device drivers. The HTML documentation contains a detailed description of the API for each device driver.

Driver Versions

Some device drivers may have multiple versions. Device drivers are usually versioned when the API changes, either due to a significant hardware change or simply restructuring of the device driver code. The version of a device driver is only indicated within the comment block of a device driver file. A modification history exists at the top of each file and contains the version of the driver. An example of a device driver version is "1.00b", where 1 is the major revision, 00 is the minor revision, and b is a subminor revision. The hardware device and its device driver must match major and minor revisions in order to be compatible.

Currently, the user is not allowed to link two versions of the same device driver into their application. The versions of a device driver use the same function and file names, thereby preventing them from being linked into the same link image. As multiple versions of drivers are supported, the version name will be included in the driver file names, as in *x<component>_v1_00_a.c*.

Primitive Data Types

The primitive data types provided by C are minimized by the device drivers because they are not guaranteed to be the same size across processor architectures. Data types which are size specific are utilized to provide portability and are contained in the header file *xbasic_types.h*.

Device I/O

The method by which I/O devices are accessed varies between processor architectures. In order for the device drivers to be portable, this difference is isolated such that the driver for a device will work for many microprocessor architectures with minimal changes. A device I/O component, XIo, in *xio.c* and *xio.h* source files, contains functions and/or macros which provide access to the device I/O and are utilized for portability.

Error Handling

Errors that occur within device drivers are propagated to the application. Errors can be divided into two classes, synchronous and asynchronous. Synchronous errors are those that are returned from function calls (either as return status or as a parameter), so propagation of the error occurs when the function returns. Asynchronous errors are those that occur during an asynchronous event, such as an interrupt and are handled through callback functions.

Return Status

In order to indicate an error condition, functions which include error processing return a status which indicates success or an error condition. Any other return values for such functions are returned as parameters. Error codes are standardized in a 32-bit word and the definitions are contained in the file *xstatus.h*.

Asserts

Asserts are utilized in the device drivers to allow better debugging capabilities. Asserts are used to test each input argument into a function. Asserts are also used to ensure that the component instance has been initialized.

Asserts may be turned off by defining the symbol NDEBUG before the inclusion of the header file *xbasic_types.h*.

The assert macro is defined in *xbasic_types.h* and calls the function XAssert when an assert condition fails. This function is designed to allow a debugger to set breakpoints to check for assert conditions when the assert macro is not connected to any form of I/O.

The XAssert function calls a user defined function and then enters an endless loop. A user may change the default behavior of asserts such that an assert condition which fails does return to the user by changing the initial value of the variable XWaitInAssert to XFALSE in *xbasic_types.c*. A user defined function may be defined by initializing the variable XAssertCallbackRoutine to the function in *xbasic_types.c*.

Communication with the Application

Communication from an application to a device driver is implemented utilizing standard function calls. Asynchronous communication from a device driver to an application is accomplished with callbacks using C function pointers. It should be noted that callback functions are called from an interrupt context in many drivers. The application function called by the asynchronous callback must minimize processing to communicate to the application thread of control.

Reentrancy and Thread Safety

The device drivers are designed to be reentrant, but may not be thread-safe due to shared resources.

Interrupt Management

The device drivers use device-specific interrupt management rather than processor-specific interrupt management.

Multi-threading & Dynamic Memory Management

The device drivers are designed without the use of mult-threading and dynamic memory management. This is expected to be accomplished by the application or by an RTOS adapter.

Cache & MMU Management

The device drivers are designed without the use of cache and MMU management. This is expected to be accomplished by the application or by an RTOS adapter.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/28/02	1.0	Xilinx initial release.
7/02/02	1.1	Made IP Spec # conditional text and removed ML reference.

ML300 Tornado 2.0 BSP User Guide

Overview

The purpose of this document is to provide an introduction to the Tornado 2.0 BSP as implemented on the ML300 reference board equipped with the Virtex II FPGA.

There are two BSPs associated with the ML300 reference board. The ML300seg BSP was created during EDK development before it was capable of generating IP bitstreams that supported all the hardware on the reference board. This BSP will eventually be phased out as the EDK matures in favor of the ML300 BSP. This document refers to both BSPs collectively as the ML300 BSP. Specific differences between the two BSPs will be noted.

The addition of the Chip Support Package (CSP) into a Tornado 2.0 BSP is a unique challenge because of the nature of how easily hardware is added and removed from the FPGA using System Build Generator and how difficult it is to accommodate this feature into a Tornado 2.0 BSP. The CSP is a part of the BSP in that it provides the software drivers for hardware IP utilized by the BSP and application code. The CSP is designed to be primarily operating system independent so in many respects it is segregated and independently configured from the BSP.

The reader is expected to understand how Tornado 2 BSPs operate in general.

Requirements

Tornado 2.0.2

The user should have Wind River Tornado 2.0.2 installed on their PC with the PPC405 libraries.

Patches required that can be found at Wind River's Windsurf technical support web site:

- SPR67953. Cumulative patch
- DosFs 2.0. Dos file system support. This package is required if you wish to use the SystemACE compact flash device as an external storage device.

SingleStep (XE)

The XE stands for Xilinx Edition. This version of the SingleStep debugger is VirtexII Pro aware. This debugger works in concert with the VisionProbe debugger pod which connects to the "CPU Debug" port of the ML300.

Installation

Copy the entire ML300 source tree to $WIND_BASE\config\ML300 and perform the following operations from the DOS command-line:

```
C:\> make clean
C:\> make release
```

When this process finishes, a new project is placed at `$WIND_BASE\proj\ML300_vx`. As an alternative to this procedure, the Tornado project facility can be used to create a bootable application using the ML300 as the basis BSP. When creating a project in this way, the BSP can be located anywhere. See Project Facility documentation from Wind River.

Compact Flash

A compress zipfile is provided in the ace subdirectory. This is a complete image containing bootrom and sample VxWorks images in ace file format. See the `README` in the ace directory for more information.

To install this image, do the following:

1. Make a backup of your microdrive then erase all files from it.
2. Uncompress the ace/compactFlash.zip file to the microdrive.
3. Insert the microdrive into the compact flash slot on the ML300.
4. Connect a serial port cable to the P106 connector on the evaluation board. Default comm settings are 115200, N, 8, 1.
5. Set the rotary switch on the ML300 to setting 6 and apply power. At this point, the VxWorks bootrom should be running and writing to the console serial port.
6. Set the bootrom boot line per your requirements. See **Bootrom Programming**, page 63 for more information.

Setting Ethernet MAC Address

To verify your MAC address is correct perform the following steps:

1. Set the rotary switch associated with the VxWorks bootrom and reboot the ML300.
2. Interrupt the countdown sequence to get the `[VxWorks Boot]:` prompt.
3. Enter the "N" command (case sensitive). The current MAC will be displayed and you will be prompted to enter a new MAC. The first three bytes of the MAC should be `000A35`.

```
Press any key to stop auto-boot...
 1
[VxWorks Boot]: N
Current Ethernet Address is: 00:0a:35:00:03:20
Modify only the last 3 bytes (board unique portion) of Ethernet Address.
The first 3 bytes are fixed at manufacturer’s default address block.
00- 00
0a- 0a
35- 35
00-
```

4. If the MAC is valid, then enter return three times to accept the default. On new boards, the address may be all FFs. If this is the case, enter the last three bytes that are assigned to the serial number. If you are not sure of the numbers, then enter return three times. This will change the MAC to `00:0a:35:FF:FF:FF`. This will provide you with a

valid MAC until the correct number is obtained. Boards with a MAC of all FFs will not be capable of running the network stack. Multiple boards connected to the same network with the same MAC will not work either.

ML300 vs. ML300seg

As discussed in the overview of this document, there are two distinct BSPs associated with the ML300 evaluation board. This section will explain differences between them.

The ML300seg BSP is intended to be used with the handcoded IP bitstream developed before the EDK was released. This bitstream supports all the board HW. The ML300 BSP uses a bitstream created with the EDK. At the current time, this bitstream does not support all of the board HW. There are enough differences between the two bitstreams to warrant two different BSPs. The table below illustrates the differences between the bitstreams.

Even though the ML300seg bitstream includes all HW, its associated BSP does not necessarily support all of it. See the release notes at the end of this document for supported devices.

Table 8-1: **ML300 vs. ML300seg bitstreams**

Component	Difference
Supported HW	ML300 doesn't support all board HW while ML300seg does.
Memory map	OPB peripheral memory map differs substantially.
DCR	ML300 uses true DCR bus while ML300seg uses DCR to OPB bridge to make DCR registers memory mapped.
GPIO	Tristate register use opposite settings on the J10 header.
Interrupt controller	1) MER register bitmap differences. 2) ML300 uses DCR interrupt controller while ML300seg uses memory mapped controller.
PLB to OPB Bridge	Totally different bridges
EEPROM	ML300 uses a GPIO line to prevent writes.
PLB/OPB Bus Error LEDs	ML300seg's PLB/OPB bridges control these leds and require no software control. ML300's PLB/OPB bridges can turn the LEDs red but require SW to turn the LEDs back to green.

Files & Directories

While the root directory of the BSP can be placed anywhere, it is typically located at `$WIND_BASE/target/config/ML300`. The Tornado Project component of the BSP is located at `$WIND_BASE/target/proj/ML300_vx`.

The project component of the BSP is required if it will be configured/compiled with the Tornado Project Facility IDE. Normally, the Project Facility is utilized during application development and trivial BSP tweaks. The non-project component (also referred to as the command-line Tornado 1.0.1 BSP) is utilized during BSP development. Note that the methods of configuring and building the BSP differ greatly between the Project and command-line methods. See Tornado documentation for more information.

The CSP adds a directory structure not usually seen with VxWorks BSPs. It has been added to segregate BSP files from the CSP.

The following directories make up the ML300 BSP:

config/ML300

The traditional directory for Tornado 2.0 BSPs. Contains BSP library source code and the command-line makefile.

config/ML300/net

Contains Tornado Project "configlette" network source code that overrides configlettes located at `$WIND_BASE/target/config/comps/src/net`.

config/ML300/ace

Contains the bitstream and compact flash image which in itself contains the bootrom and other sample VxWorks ace and elf images.

config/ML300/ip_csp

The base directory for the CSP.

config/ML300/ip_csp/xsrc

Contains source code for the CSP.

proj/ML300_vx

The base directory for a Tornado project. All files here are maintained by the Project Facility.

proj/ML300_vx/<build spec>

A build specification maintained by the Project Facility. There is typically a "default" build spec here unless removed by the developer. Other build specifications can be added by the developer.

CSP Driver Organization

This section briefly discusses how the CSP is compiled and linked and eventually used by Tornado makefiles to include into the VxWorks image.

CSP drivers are implemented in "C" and can be distributed among several source files unlike traditional VxWorks drivers which consist of single "C" header and implementation files. For this reason, we place all CSP files in their own `xsrc` subdirectory.

There are up to three components for CSP drivers:

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

"Driver source inclusion" refers to how CSP drivers are compiled. For every CSP driver, there is a file named `ip_<dev>_<version>.c`. This file `#include`’s each CSP driver source file(s) (*.c) for the given device.

This process is analogous to how VxWorks’ sysLib.c `#include`’s source for Wind River supplied drivers. The reason why CSP files are not simply `#include`’d in sysLib.c like the rest of the drivers is due to namespace conflicts and maintainability issues. If all CSP files were part of a single compilation unit, static functions and data are no longer private. This places restrictions on the CSP device drivers and would take away from its 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 CSP build.

Configuration

The ML300 BSP is configured just like any other Tornado 2 BSP. There is not much configurability to CSP drivers since the IP hardware has been pre-configured in most cases by System Build Generator. The only configuration available generally is whether the driver is included in the CSP at all. How to go about 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 CSP device driver does not mean that driver will be automatically utilized. Most CSP 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.

Command-Line

A set of constants (one for each driver) are defined in `config/ML300/ip_config.h` and follow the format:

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

This file is included near the top of `config/ML300/config.h`. By default all drivers are included in the build. To exclude a driver, add the following line in `config.h` after the `#include "ip_config.h"` statement.

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

This 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

The Project Facility is part of the Tornado IDE. It is a GUI driven environment. To add/delete CSP drivers, go to the VxWorks pane in the workspace window (see figure below). Then add/delete driver components under IP_CSP just as you would with any other VxWorks component.

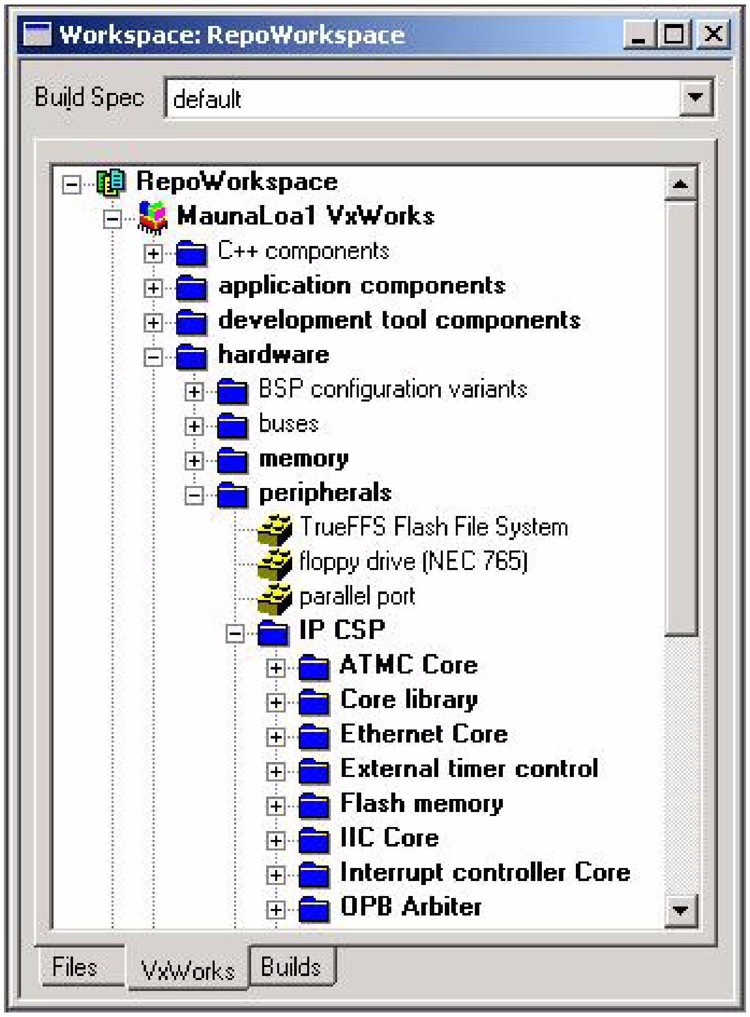

Build Spec: default

Figure 1: **Project Facility GUI Configuration**

Note that whatever configuration has been specified in `ip_config.h` and `config.h` will be overridden by the project facility.

Memory Map

Due to the nature of this evaluation board a full memory map is not given in this document. The user is instead referenced to "C" source code header file `xparameters.h`. This source file provides a memory map for all CSP devices. A partial map is given here that relates directly to BSP operation.

Table 8-2: **System Memory Map**

Device	Start (hex)	End (hex)	Size (bytes)
PLB DDR	00000000	07DFFFFF	126 MB
PLB DDR (ML300)	07E00000	07FFFFFF	2 MB
PLB LCD Frame Buffer (ML300seg)*	07E00000	07FFFFFF	2 MB
OPB Space (ML300Seg)	60000000	DFFFFFFF	2 GB
OPB Space (ML300)	60000000	60010000	64 KB
BRAM	FFFF8000	FFFFFFFF	32 KB

* LCD Frame buffer may be added to the ML300 BSP in a later release.

RAM Memory Map (includes DDR and BRAM)

RAM device contains the VxWorks runtime image and heap space. ML300 follows VxWorks conventions for RAM usage for PowerPC processors. Refer to Appendix F of the *VxWorks 5.4 Programmer's Guide*.

Table 8-3: **RAM Memory Map**

Physical Address Range (hex)	Usage
00000000..000000FF	(DDR) Unused & undefined
00000100..00002FFF	(DDR) Interrupt Vector table
00003000..00010000	(DDR) VxWorks usage. Exception reason message and other VxWorks constructs are at the bottom of this region. Initial stack is set at the top of this range and grows downward. Once VxWorks has switched to multi-tasking mode, this stack is no longer used.
00010000..00BFFFFF	(DDR) RAM_LOW_ADRS. VxWorks image, interrupt stack, host memory pool, and heap space.
00C00000..07BFFFFF	(DDR) RAM_HIGH_ADRS. Two possible uses. (1) VxWorks bootrom image and heap space. (2) VxWorks heap space.
07C00000..07CFFFFF	(DDR) USER_RESERVED_MEM. This 1MB is used for network data buffers and network DMA descriptor spaces.
07D00000..07DFFFFF	(DDR) USER_RESERVED_MEM. This 1 MB is not used by BSP. Available for application use
FFFF8000..FFFFFFFF	(BRAM) Address FFFFFFFC contains reset vector.

NVRAM

NVRAM support is provided by a Microchip Technology 24LC32A EEPROM on the IIC bus. This device provides 4KB of storage space. BSP source code file `24LC32aNvRam.c` is the driver for this device and provides the API interface required by VxWorks. The primary BSP related objects stored in NVRAM are the bootline and the Ethernet MAC address.

When there is no IIC bus support, the BSP will replace the EEPROM driver with `$WIND_BASE/src/drv/mem/nullNvRam.c` which provides only function stubs so that VxWorks will link. When this is the case, the default bootline is used (see `config.h`) and the Ethernet MAC address defaults to: `00:0a:35:00:00:00`.

Table 8-4: **NVRAM Memory Map**

Part Offset Range (hex)	sysNvRamGet/Set Offset	Usage
0000..07FF	N/A*	Reserved for board level objects such as the Ethernet MAC address
0800..08FF	0000..00FF	Reserved for VxWorks bootline
0900..0FEF	0100..07EF	Unused
0FF0..0FFF	07F0..07FF	Reserved

* `sysNvRamGet` and `sysNvRamSet` are the VxWorks required NVRAM interface functions. The interface they provide uses offsets relative to the bootline offset. Accessing part offsets 0000..07FF requires an alternate interface.

Caches

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

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

The caches are configured by the following constants in `ML300.h`. These constants map to the PPC cache control registers of the same name. See PPC405 documentation for further information on these registers:

- `ML300_ICCR_VAL` - Initial contents of the ICCR register (instruction cacheability attribute).
- `ML300_DCCR_VAL` - Initial contents of the DCCR register (data cacheability attribute).
- `ML300_DCWR_VAL` - Initial contents of the DCWR register (write back/through attribute).
- `ML300_SGR_VAL` - Initial contents of the SGR register (guarded attribute).

Table 8-5: **Cache Map**

Physical Address Range (hex)	I Cache	D Cache	Write Back/Through	Guarded
00000000..07FFFFFF	Y	Y[1]	Back	N
F8000000..FFFFFFFF	Y	N	N/A	N
everything else	N	N	N/A	N

[1] This region includes the LCD frame buffer. A data cache flush may be required to write all data to the buffer.

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

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

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 #define'd 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`. 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.

The PPC405 is capable of other attributes including zone protection, however, Wind River documentation is poor in this area and it is unclear whether the basic MMU package supports them. An add-on is available from Wind River (which is enabled by `INCLUDE_MMU_FULL`) for advanced MMU operations.

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 (see `ML300_ICCR_VAL` in previous section). This strategy can dramatically reduce the number of page faults while still keeping instructions in cache.

Exception Handling

There are two types of exceptions which are of importance to the BSP. The first type are internal exceptions such as machine check, illegal instruction, etc.. By default, the BSP configures VxWorks to trap these types of exceptions. When one occurs, the offending task is suspended and a descriptive message is displayed on the console. If the exception occurs in interrupt context, VxWorks will reboot itself.

The other type of exception are external asynchronous. The BSP initializes and handles these exceptions which are the result of an active signal on the external or critical interrupt pins of the processor.

There are two INTC IP devices within the FPGA, one connected to the processor's external interrupt and the other on the critical interrupt. Functions in BSP source code file `sysInterrupt.c` are responsible for initializing these two devices with the `XIntc` component driver and hooking them into VxWorks.

External Interrupts

Most IP peripherals that can generate interrupts are attached to the INTC component responsible for asserting the external interrupt processor exception. BSP initialization code hooks control of this device into the VxWorks `intLib` library.

External interrupt vectors are defined in `xparameters.h`. `ML300.h` may translate these vectors into `SYS_<device>_VEC_ID` to limit changes to BSP source code when device names change. These constants are utilized when invoking the VxWorks intLib functions. Example:

```
#include <intLib.h>

void foo(void)
{
   intEnable(SOME_DEVICE_VEC_ID);
}
```

Critical Interrupts

Since VxWorks does not define a critical interrupt API as it does for external interrupts, the user must utilize the API defined in `sysLibExtra.h`. Functions `sysIntCritConnect`, `sysIntCritEnable`, and `sysIntCritDisable` are designed to work identically to those for the external interrupt defined by the VxWorks `intLib.h` library. Example

```
#include "sysLibExtra.h"

void foo(void)
{
   sysIntCritEnable(SOME_CRITICAL_DEVICE_VEC_ID);
}
```

Note: PLB/OPB bridges & arbiters are wired to the critical interrupt handler in both the ML300 and ML300seg bitstreams. If these interrupt sources are enabled and the PPC machine check interrupt is enabled then VxWorks may reboot when an exception occurs. This is because the PLB/OPB bridge/arbiter will most likely interrupt when a transaction cannot complete. At the same time the PPC will detect a bad bus cycle and generate a machine check exception. This will lead to the VxWorks exception handler being interrupted. VxWorks architecturally does not allow this and will reboot the system when it occurs.

It is not recommended to `sysIntCritEnable()` one of these interrupt sources. Instead, use the VxWorks `excHookAdd()` function to use your own function perform custom exception processing (after VxWorks finishes its own processing). Here, the hook function can examine the bridges/arbiters and perform whatever task is required for the event.

IIC

There are several devices connected to the IIC bus with hardwired addresses. These addresses are defined for the BSP in the `ML300.h` header file. The BSP provides a polled interface to the IIC bus to access these devices. The interface includes a mutual exclusion semaphore that can be used to prevent more than one task from accessing the bus at a time. Before the operating system is up, the semaphore is not available and it is up to boot code to sequence access to the bus. This should not be an issue since the system is single-threaded at boot time and the only device accessed should be the NVRAM.

BSP file `sysIic.c` provides initialization, read/write primitives, and resource allocation functions.

System ACE

The System ACE controller is a device that provides a way to store multiple FPGA bitstream loads. These loads are stored on a compact flash (CF) device and downloaded by the System ACE controller into the FPGA when the system is powered up. Additionally, these bitstreams can contain a software load that is downloaded to RAM after the FPGA's IP cores have been programmed. These bitstream loads are stored in the CF device in a DOS filesystem. This means regular files can be accessed from the CF as well. Such files include VxWorks ELF images, application code & data, and text script files.

The BSP utilizes the SystemACE controller in two ways. First as a boot device and second as an external storage device. Both applications require the following constants be defined in `config.h` or by using the Tornado Project facility to change the constants of the same name:

- `SYS_SYSACE_DEV_ID` - Should be set to the `xparameters.h` XPAR constant associated with the System ACE controller device identifier.
- `SYS_SYSACE_BASEADDR` - Should be set to the `xparameters.h` XPAR constant associated with the System ACE controller base address.

DOS File System

When being used as a file storage device, 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 `#define`'d 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_CBIO

Automounting

To automatically mount the System ACE as a file system at boot time, `INCLUDE_XSYSACE_AUTOMOUNT` must be defined. In the Project facility, this is defined by enabling the automount feature in the System ACE folder. When defined, two more constants are utilized to mount the compact flash device: `SYSACE_AUTOMOUNT_POINT` and `SYSACE_AUTOMOUNT_PARTITION`. In the Project facility, these constants can be set by editing the System ACE properties folder. This relieves the application from having to initialize and mount the DOS File system. Note that this works only for Project builds. Command line builds require that the application invoke `sysSystemAceInitFS()` and `sysSystemAceMount()`. These functions are described in the Board API section below.

Board API

This section will not go over CSP device driver functions. Instead the user is directed to the appropriate `ip_csp/xsrc/<device>.c` file for documentation and usage.

There are a handful of "board level" BSP functions not implemented by the CSP device drivers. Prototypes for these functions are located in `config/ML300/sysLibExtra.h`.

Standard I/O

The BSP comes with stdin, stdout, and stderr directed through the UART on the P106 connector . The default UART baud rate is set to 115200, no parity, 8 data bits, and 1 stop bit. The secondary UART on P107 is enabled and ready for application usage. It defaults to 19200 baud, no parity, 8 data bits, and 1 stop bit.

GPIO

Two instances of GPIO can be included in the BSP. The first instance controls the momentary push button switches and their surrounding LEDs. The second controls the 32 GPIO lines on the J10 connector. Both instances require that `INCLUDE_XGPIO` constant be defined. Each instance can be enabled or disabled with constants `INCLUDE_GPIO_LED_SWITCHES` and `INCLUDE_GPIO_TEST_PORT`.

void sysLedOn(UINT32 mask)

Turns on LEDs in the mask. Bits set to one cause the associated LED to be illuminated. The mask is built using constants `GPIO_LED_DSxx` defined in `ML300.h` where `xx` is the LED number and `DSxx` is the LED label on the PCB. This function requires that both `INCLUDE_XGPIO` and `INCLUDE_GPIO_LED_SWITCHES` be defined.

void sysLedOff(UINT32 mask)

Turns off LEDs in the mask. Bits set to one cause the associated LED to be turned off. The mask is built using constants `GPIO_LED_DSxx` defined in `ML300.h` where `xx` is the LED number and `DSxx` is the LED label on the PCB. This function requires that both `INCLUDE_XGPIO` and `INCLUDE_GPIO_LED_SWITCHES` be defined.

UINT32 sysSwitchReadState(void)

Reads the state of all the push button switches. A mask is returned describing which switches are closed (i.e. being pushed). The mask is decoded using constants `GPIO_SWITCH_SWxx` defined in ML300.h where `xx` is the switch number and `SWxx` is the switch label on the PCB. This function requires that both `INCLUDE_XGPIO` and `INCLUDE_GPIO_LED_SWITCHES` be defined. Usage example:

```
UINT32 mask = sysSwitchReadState();

if (mask & GPIO_SWITCH_SW06)
{
    // handle switch 6 press
}
```

void sysGpioBankSetDataDirection(UINT32 mask)

Sets the output enable for the J10 32-bit GPIO header located adjacent to the LCD display. Bits in the mask set to "1" are inputs, "0" are outputs.

Note for ML300seg: SEG bitstreams reverse the mask, "1" is output, "0" is input. If you are using an EDK generated bitstream then disregard this note.

void sysGpioBankWriteDiscretes(UINT32 data)

Writes to the 32-bit GPIO J10 header.

UINT32 sysGpioBankReadDiscretes(void)

Reads the state of the pins of the 32-bit GPIO J10 header.

void sysLedBusErrClear(UINT32 ledMask)

(Not available in the ML300seg BSP)

Turns the PLB & OPB bus error LEDs from red (bus error occured) to green. This function does not clear the error condition. Parameter ledMask is formed from or'ing together `GPIO_LED_BUSERR` constants defined in ML300.h.

System ACE

These routines require that the `INCLUDE_XSYSACE` constant be defined. The command line BSP will not initialize the DOS file system resident on the compact flash. Application code will have to make function calls to initialize and mount:

```
FILE *fp;

sysSystemAceInitFS();
if (sysSystemAceMount("/cf0", 1) != OK)
{
   /* handle error */
}

fp = fopen("/cf0/myfile.dat","r");
         .
         .
```

STATUS sysSystemAceSetRebootAddr(unsigned configAddr)

Sets the reboot JTAG configuration address. This address is mapped to cfgaddr0..7 as defined in XILINX.SYS in the root directory of the CF device. If this function is never invoked, then the default address is used. The default address is the address selected by the rotary switch. The given address will be rebooted if `sysToMonitor()` or `reset()` is called.

The `configAddr` parameter range is 0..7 (i.e. cfgaddr0..7) or -1 to select the default address.

Returns ERROR if `configAddr` is out of range, OK otherwise.

void sysSystemAceInitFS(void)

Initializes the required Wind River DosFs 2.0 libraries. Application code is not required to call this function on a BSP built with the Project facility.

STATUS sysSystemAceMount(char* mountPoint, int partition)

Mount the compact flash as DOS file system volume. The `mountpoint` parameter is an arbitrary string labeling the device. Once mounted, refer to this mountpoint in all file accesses. The `partition` parameter specifies the partition to mount. If "0" is specified then the boot device is assumed to not contain a partition table (i.e. it is treated like a floppy disk).

Note: Before calling this routine, be sure to initialize the DOS file system with a call to `sysSystemAceInitFS()`.

Note: Application code is not required to call this function on a BSP built with the Project facility with `INCLUDE_XSYSACE_AUTOMOUNT` defined.

LCD (ML300seg)

These functions perform very basic operations useful for verifying the LCD is working. A more substantial library will be required if, say, someone wanted to port Quake. Screen geometry is defined in `ML300.h` with the constants `LCD_COLS`, `LCD_ROWS`, and `LCD_ROW_ALIGNMENT`.

These functions are not available in the ML300 BSP.

void sysLcdSetColor(UINT32 rgb)

Set the entire display to the color encoded by the rgb parameter. The rgb parameter is encoded as follows: 0x00RRGGB where RR is the red component, GG is the green component, and RR is the red component. A value of 0x00000000 is black, 0x00FFFFFF is white.

void sysLcdDisplayColorBars(void)

Writes a test display to the LCD screen that includes 8 bars in the top half of the display and a 256 grey-scale pattern in the lower half of the display.

void sysLcdSetPixels(int row, int col, unsigned numPixels, UINT32 rgb)

Starting at row and column, write the RGB encoded value to consecutive pixels as they exist in the LCD's frame buffer. This basically boils down to a horizontal line draw function. If `numPixels` is large enough, then the RGB color continues onto the next row. See `sysLcdSetColor` described above for information on the rgb parameter.

STATUS sysLcdSetBrightness(unsigned char value)

This routine sets the brightness level of the LCD display. Valid range is 0..255 with 0 being the dimmest setting. This setting is persistent in that the LCD keeps this setting even through power cycles.

Returns `ERROR` if unable to communicate with device, `OK` otherwise.

STATUS sysLcdGetBrightness(unsigned char *value)

This routine retrieves the brightness level of the LCD display. Valid returned range is 0..255 with 0 being the dimmest setting.

Returns `ERROR` if unable to communicate with device, `OK` otherwise.

Power & Temperature Monitor Functions

void sysPowerMonCpuGet(int *v1_8, int *v2_5, int *v3_3, int *v5, int *v12)

This routine reads the two power monitor devices on the IIC bus to determine the current voltage levels on the CPU board. All voltages are returned in units of milli-volts. If the voltage cannot be read for any reason, then that voltage level is returned as `SYS_MEASUREMENT-_ERROR`. Parameters are interpreted as follows: v1_8 = 1.8volt source, etc..

void sysPowerMonIoGet(int *v1_8, int *v2_5, int *v3_3, int *v5, int *v12)

This routine reads the two power monitor devices on the IIC bus to determine the current voltage levels on the IO board. All voltages are returned in units of milli-volts. If the voltage cannot be read for any reason, then that voltage level is returned as `SYS_MEASUREMENT-_ERROR`. Parameters are interpreted as follows: v1_8 = 1.8volt source, etc..

void sysPowerMonShow(void)

Print the voltages from all power monitor sources to the console. If errors are encountered while reading the voltage monitors, then "Err" is displayed next to the voltage. This function requires `INCLUDE_POWERMON_SHOW` be defined in `config.h` or in the project facility under *development tool components -> show routines.*

void sysTemperatureMonGet(int *cpu, int *ambient)

This routine reads the two temperature sensing devices on the IIC bus to determine the current temperature. If the temperature cannot be read for whatever reason, then that temperature is returned as `SYS_MEASUREMENT_ERROR`. Temperature is returned in units of deg C.

void sysPowerMonShow(void)

Print the temperature (in deg C) for the CPU and the ambient temperature. If errors are encountered while reading the temperature monitors, then "Err" is displayed next to the temperature. `INCLUDE_TEMPERATUREMON_SHOW` be defined in `config.h` or in the project facility under *development tool components -> show routines.*

Miscellaneous Functions

void sysMsDelay(UINT32 delay)

Delay the specified number of milliseconds. The delay is implemented as a busy loop that occupies the CPU. The delay can be pre-empted by a higher priority task or interrupts if tasking/interrupts are enabled causing loss of delay precision.

void sysUsDelay(UINT32 delay)

Delay the specified number of microseconds. The delay is implemented as a busy loop that occupies the CPU. The delay can be pre-empted by a higher priority task or interrupts if tasking/interrupts are enabled causing loss of delay precision.

This function not accurate for delay times below 20us due to system overhead. The overhead is more or less constant and can be negated by the use of `SYS_US_DELAY_BIAS` defined in `config.h`. Use this constant to calibrate to your system’s needs. As delivered with a 300 MHz CPU clock and a bias of -2, this function is accurate within +/-15% for a 20us delay. As the delay time increases, the accuracy increases.

void sysEepromWriteEnable(void)

(Not available in the ML300seg BSP)

Enables writes to the IIC EEPROM.

void sysEepromWriteDisable(void)

(Not available in the ML300seg BSP)

Disable writes to the IIC EEPROM.

ML300 Specific Options

This section discusses ML300 specific configuration options that can be set either in `config.h` or in the Project GUI. Unless otherwise stated, these options can be set by

#define'ing or #undef'ing them in config.h or by defining them in the Project GUI in the project workspace's macros settings in the build tab.

Table 8-6: **Custom BSP Options**

Option	Description
INCLUDE_XSYSACE_INSTALL_-RESET_VEC	Controls whether reset code is placed the processor's reset vector address. This reset code will trigger SystemACE to load the default configuration bitstream.
INCLUDE_XSYSACE_AUTOMOUNT	Controls whether the System ACE filesystem is mounted at boot time using the next two SYSACE_ constants defined in this table. This constant affects only Project builds.
SYSACE_AUTOMOUNT_POINT	Default mount point used when INCLUDE_-XSYSACE_AUTOMOUNT is defined in Project builds.
SYSACE_AUTOMOUNT_PARTITION	Default partition used when INCLUDE_XSYSACE_-AUTOMOUNT is defined in Project builds.
INCLUDE_GPIO_LED_SWITCHES	Controls whether GPIO support is present for the switches on top of the board and the LED in close proximity to those switches. If support is not included, then functions sysLedOn, sysLedOff, and sysSwitchReadState have no effect.
INCLUDE_GPIO_TEST_PORT	Controls whether GPIO support is present for the J10 I/O connector port. If support is not included, then sysGpioBank functions have no effect.
SYS_US_DELAY_BIAS	Adds the specified number of microseconds to the delay parameter in sysUsDelay(). This option can be used to cancel out overhead.
INCLUDE_LCD_CLEAR_AT_BOOT	(ML300seg BSP) Clears the LCD display at VxWorks boot time.
INCLUDE_LCD_BARS_AT_BOOT	(ML300seg BSP) Displays the color bar test screen on the LCD at VxWorks boot time. If both INCLUDE_LCD_BARS_-AT_BOOT and INCLUDE_LCD_CLEAR_AT_BOOT are defined, the LCD will first be cleared then the color bars will be drawn.
INCLUDE_EMAC_PHY_RESET_-AT_BOOT	Controls whether the Ethernet PHY is reset at boot time. In the Project GUI, this is a parameter under the emac component and can be found under *hardware->peripherals->IP CSP-> Ethernet Core*. Set to TRUE to enable, FALSE to disable.
INCLUDE_POWERMON_SHOW	Controls whether function sysPowerMonShow is compiled into the BSP.
INCLUDE_TEMPERATUREMON_-SHOW	Controls whether function sysTemperature-MonShow is compiled into the BSP.
SYS_GPIO_SWITCH_DEBOUNCE_TICKS	Sampling interval used by function sysSwitchReadState() when attempting to debounce switches. Units are in clock ticks.

Building VxWorks

The ML300 BSP follows the standard Tornado conventions when it comes to creating a VxWorks image. Refer to Tornado documentation on how to make a VxWorks image. This section discusses extensions made to the build process.

Command-Line BSP Build Extensions

The CSP is compiled/linked with the same toolchain VxWorks is created with. Only very minor additions are made to the Makefile to support the CSP build such as compiler directives telling the BSP where to find the CSP files and visa-versa.

Project BSP Build Extensions

There are no extensions to the Project build. The BSP should behave just like any other normal BSP.

Bootup Sequence

There are many variations of VxWorks images with some based in RAM, some in ROM. Not all these images are supported on the ML300 board. The following list discusses various image types:

- Compressed images - Not supported. These images begin execution in ROM and decompress the image into RAM. SystemACE has no knowledge of the compression algorithm being used by VxWorks.
- RAM based images - Fully supported.
- ROM based images - Fully supported. These images begin execution in ROM, copy themselves to RAM then transfer execution to and stay in RAM. System ACE performs the copy operation, so the BSP has been prepared to short circuit the VxWorks copy (see `romInit.s`).
- ROM resident images - Not supported. These images begin execution in ROM, copy the data section to RAM, and execution remains in ROM. There is no ROM device in ML300. Theoretically BRAM could be used as a ROM however the current VirtexII Pro parts being used in ML300s do not have the capacity to store a VxWorks image which could range in size from 200KB to over 700KB.

vxWorks

This image is meant to be downloaded to the target RAM space. Once downloaded, the PC should be set to function `_sysInit` (implemented in `sysALib.s`). Most of the time, the device performing the download will do this for you as it can extract the entry point from the image.

1. `_sysInit` : Low level initialization. Since this image is copied to RAM, the device that downloaded the image may have to perform manual system initialization to make RAM visible. When completed, this function will setup the initial stack and invoke the first "C" function `usrInit()`.
2. `usrInit()` : Performs pre-kernel initialization. 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. 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.

bootrom_uncmp

This 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 to perform tasks before jumping to RAM. This image behaves differently than a traditional bootrom due to the fact it is already in RAM when control is passed to it (via System ACE).

1. Power on. System ACE loads the bitstream into the FPGA then loads the bootrom image into RAM and passes control to assembly language function `_romInit` located in `romInit.s`.
2. `_romInit` : Traditionally this function would perform board level initialization then call `romInit()` which would copy the VxWorks image to RAM. Since the image is already in RAM, this function simply jumps to assembly function `_sysInit`.
3. Follows steps 1, 2 & 3 of the "vxWorks" bootup sequence.

Difference Between Command-Line & Project BSPs

Functions `usrInit()`, `usrRoot()`, and `romStart()` as explained in the boot sequence steps above are implemented by Tornado. In command line BSPs, these functions are defined in source code located at `$WIND_BASE/target/config/all`. In Project BSPs, the Project Facility generates this code in the user's project directory.

Functions `_sysInit`, `_romInit`, `sysHwInit()`, and `sysHwInit2()` are implemented by the BSP in `config/ML300`. These functions are utilized on both the command-line and project BSPs.

Bootrom Programming

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 full image. If the image is retrieved on the Ethernet, 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 set 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 development and replaced with the production VxWorks image.

Creating Bootroms

On a command line window, cd to the `config/ML300` directory. Issue a "`make bootrom_uncmp`". Run the batch file $WIND_BASE`\host\x86-win32\bin\torVars.bat` (if using Micro$oft Windows) to setup command line environment variables before building the bootroms.

The next step is to either test `bootrom_uncmp` by downloading it with an emulator or creating an .ace file out of it (combined with the IP core bitstream) for download by SystemACE. See VirtexII Pro documentation on how to create .ace files

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 1984-1998  Wind River Systems, Inc.

CPU: ML300 VirtexII Pro PPC405 Rev D
Version: 5.4.2
BSP version: 1.2/0
Creation date: July 26 2002, 12:51:32


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

[VxWorks Boot]:
```

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

Bootline

Non-volatile storage of the bootline requires NVRAM support which in itself requires IIC support. If NVRAM support is not present or an error occurs reading it, then the `DEFAULT_BOOT_LINE` is utilized. If NVRAM is uninitialized (such as it will be in new systems) then the bootline may be gibberish.

ML300 bootroms support the network interface and System ACE as the boot device. The bootline tells the bootrom how to find the vxWorks image. The bootline is maintained at runtime by the bootrom. 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 edit the bootline. Enter "p" to view the bootline. On a non-bootrom image, you can still change the bootrom by entering the `bootChange` command at a host or target shell prompt.

The following list goes over the meanings of the bootline fields:

- `boot device` : Choices are "xemac" or "sysace=x". When set to xemac, the BSP will boot over the network. When set to "sysace=x", the BSP will boot from a file resident on the System ACE device. See **Booting from SystemACE**, page 65 for further information on how to specify the System ACE boot device. 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.
- `host name` : Name as needed. Can be arbitrary.
- `file name` : The VxWorks image to boot. If the boot device is the network "xemac", then the file must be accessible on the host computer via ftp. See **Booting from SystemACE**, page 65 for specifying a System ACE file.
- `inet on ethernet (e)` : The IP internet address of the target. If there is no network interface, then this field can be left blank.
- `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)` : Whatever names suites you.
- `other (o)` : This field is not applicable when "xemac" is specified as the boot device. When "sysace" is the boot device, then this field should be set to "xemac". This will signal the VxWorks image specified in the `file name` field to start the network on the xemac device. (if network support was included)
- `inet on backplane (b)` : Leave blank. ML300 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`

Booting from SystemACE

The "boot device" field of the bootline is specified using the following syntax:

sysace=<partition number>

where `<partition number>` is the partition to boot from. Some CF devices do not have a partition table and are formatted as if they were a large floppy drive. 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 accessed.

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 file system partition. Simply build vxWorks using the Tornado tools then 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 `cf0` 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 DOS formatted disk, read the file into memory and begin execution.

2. Boot from an ace file. The ace file can contain HW only, SW only, 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 correct memory address ranges, 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. ace files are created with the Xilinx ISI tools and is beyond the scope of this manual.

The "file name" must have the following syntax:

```
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 selected by the rotary switch on the ML300 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.

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:/tornado/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.

```
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
```

This BSP on Other Boards

The ML300 BSP can be used directly on other development hardware such as the Xilinx AFX Evaluation board. Modifications will be required to the BSP to account for a differing list of CSP peripherals and the amount and type of RAM.

First step is to remove all unsupported peripheral drivers from the BSP. This can be done by following steps in the section titled **Configuration**, page 49.

Next step is to change the amount and type of RAM the BSP recognizes. This can be done by editing the constant `LOCAL_MEM_SIZE`. If not using the Tornado project facility, this constant can be modified by editing its definition in `config.h`. If using the Tornado Project facility, this constant can be modified by changing its property definition in the memory folder.

Other areas to watch out for are constants defined by `xparameters.h`. These constants must match the VirtexII Pro bitstream. If they do not then all kinds of problems can be expected such as bus errors. Key constants to watch out for:

- XPAR_CORE_CLOCK_FREQ_HZ
- XPAR_UARTNS550_0_CLOCK_HZ
- XPAR_<device>_BASEADDR
- XPAR_INTC_0_<device>_x_VEC_ID

Deviations

This section sums up the difference between garden variety BSPs and the ML300. The differences between the two fall roughly into key areas: CSP and System ACE support.

The CSP contains drivers for the Xilinx IP cores (see **CSP Driver Organization**, page 48). To keep the BSP buildable while maintaining compatibility with the Tornado Project facility, a set of files named `ip_<driver>_<version>.c` populate the BSP directory that simply `#include` the source code from the CSP.

The location of the CSP relative to the BSP directory causes problems because command line and Project facility differ in how BSP files are found during compilation. To address this issue, a key Project macro (BSP_DIR) is defined in the BSP's `Makefile`. Of all deviations, this one is the most dangerous because future versions of Tornado may cause builds to fail. The `Makefile` contains more information about this deviation.

System ACE, being a boot device and a DOS file system, has required that two VxWorks source code files found in the Tornado distribution be changed. Wind River allows BSP developers to change some source code files provided they follow set guidelines. The two files that have been modified from their original version are `bootConfig.c` and `net/usrNetBoot.c`.

`usrNetBoot.c`, used only by Project Facility builds, required a 1 line of code change to tell VxWorks that the System ACE device is a disk based system like IDE, SCSI, or floppy drives. This change allows the BSP to properly process the "other" field of the bootline (see

Bootline, page 64) when System ACE is the boot device. The "other" field allows the selection of a network device when booting from a disk based system.

`bootConfig.c`, used only by bootroms builds, required extensive modifications to support SystemACE as a boot device. These mods are bracketed by `INCLUDE_XSYSACE` preprocessor ifdefs. Another mod enabled the data cache when ethernet frames are copied from fifos instead of DMA. This change greatly increases the bootup times for the system but could cause problems if another device required for booting utilizes DMA or requires some sort of special cache coherency in the first 128MB of address space.

Limitations

This section goes over what this BSP cannot do and the reasons why. It also goes over what-if scenarios when key pieces of IP are not part of the FPGA load.

No WARM boots

All boots are cold. There is no distinction between warm, cold, or any other type of boot. This is because reboots are managed by the System ACE device which resets the processor whenever it performs an ace download.

This could cause an exception message generated by VxWorks to not be printed to the console when the system is rebooted due to an exception in an ISR or a kernel panic. See troubleshooting guide for tips to get at this exception message.

No compressed images

If you compile a compressed image then try to boot it as an ace file, results will be undetermined. This is because System ACE cannot decompress data as it writes it to ram.

Command line builds cannot initialize the network when System ACE is the boot device

This requires that the application provide code to initialize the network. Project builds can get around this because a modified `net/usrNetBoot.c` is provided in the BSP directory (see **Deviations**, page 67). The equivalent file for command line builds is located at `$WIND_BASE/target/src/config/usrNetwork.c`. The architecture of the command line build prevents us from overriding this file with a clone in the BSP directory.

Fixing `usrNetwork.c` requires changing the following code in function `usrNetInit()`:

```
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

```
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.

Reset Vector

On the PPC405 processor, the reset vector is at physical address 0xFFFFFFFC. There is a short time window where the processor will attempt to fetch and execute the instruction at this address.This window is between the time when System ACE has finished downloading the HW bit stream and before it begins to download the SW image. All VxWorks requires here is the following assembly instruction:

```
FFFFFFFC   b .
```

This is in effect a spin loop. This instruction encodes into 0x48000000. Be sure whoever writes the HW IP includes this instruction at this address which is typically a BRAM internal to the FPGA.

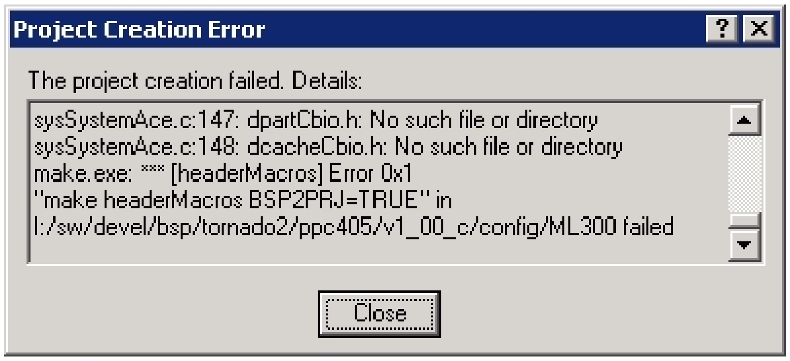
Trouble Shooting

Project Creation

Issues seen when creating a Tornado Project based on the ML300 BSP.

"Project Creation Error" Dialog Pop-up

Scroll to the end of the box and if it contains error messages complaining about missing header files `dpartCbio.h` and `dcacheCbio.h`, then you don't have the DosFS 2.0 libraries installed in your system.


Project Creation Error

The project creation failed. Details:

```
sysSystemAce.c:147: dpartCbio.h: No such file or directory
sysSystemAce.c:148: dcacheCbio.h: No such file or directory
make.exe: *** [headerMacros] Error 0x1
"make headerMacros BSP2PRJ=TRUE" in
I:/sw/devel/bsp/tornado2/ppc405/v1_00_c/config/ML300 failed
```

SingleStep

Issues seen when using SingleStep with this BSP.

Source browser not displaying source code at addresses where source code should be

Try to rebuild everything with the -gdwarf compiler option.

Tornado Crosswind debugger

Issues seen when using Tornado's IDE debugger with this BSP.

Source browser not displaying source code at addresses where source code should be

Is the -gdwarf option enabled in the compiler? Try to rebuild everything with the -g compiler option.

Target Shell Issues

Issues seen when using the built-in target shell.

Relocation value does not fit in 24 bits message from Loader

This is seen when a system contains more than 32MB of memory. Recompile your source code using the -mlongcall compiler option.

Ethernet Issues

Issues seen when integrating/using the XEmac Ethernet adapter

Network interface xemac unknown. Message from console at boot

There are multiple causes to this problem.

1. Did you compile the XEmac component into the BSP? In the Tornado Project facility, check that both *hardware->peripherals->IP CSP->Ethernet EMAC* and *EMAC END* components are included. On command line BSP builds, is `INCLUDE_XEMAC` and `INCLUDE_XEMAC_END` declared in `ip_config.h` and not `#undef`'d anywhere.
2. In the Tornado Project facility, if you remove then later restore network support, Project fails to restore "END" driver support. Check folder *network components->network devices* and verify that both *END attach interface* and *END interface support* components are included.

BSP Release History

ML300 1.2/0 - September 26, 2002

First pre-release for Tornado 2.0. This is a beta release that does not include support for all HW. Note that testing has been done on the ML3 evaluation board as opposed to the ML300. In other words, this version of the BSP has never been run on the ML300 hardware. The SEG IP bitstream is used for this load.

HW Supported

- 16550 UART on outside edge connector (VxWorks console)
- EMAC Ethernet
- 16MB DDR RAM
- 32KB BRAM
- System ACE
- PPC Instruction cache

HW Not supported

- PPC Data cache
- PPC MMU
- PLB/OPB Bridge register access (no access to BEAR, BESR registers).
- LCD display
- PCI
- GPIO
- Parallel Port
- PS2 Ports

Usage Notes

1. Bootroms: The bootroms are integrated into the FPGA bitstream and downloaded by System ACE at powerup and reset. There are two different types of bootroms stored in the ace subdirectory. Each one uses serial port #1 as the console at 38400 baud, N,8,1.

   `top_vxboot.ace`: This bootrom has a hardcoded bootline of "`sysace=1(0,0):/cf0/ vxworks/vxWorks.st`". It will mount the compact flash device using the MPU interface of the System ACE as an external DOS volume. The given VxWorks image will be loaded and started. If a `/vxworks` directory is not in your compact flash device then create one and place your `vxWorks.st` image there. This bootrom has no network support.

   `top_vxbootnet.ace`: This bootrom has a hardcoded bootline of "`xemac(0,0)host:c:/tornado/ target/config/ML300/vxWorks h=192.168.0.1 e=192.168.0.2 u=xemhost pw=slurm`". The network is started and the vxWorks image is downloaded via ftp.
2. When using SystemACE as the boot device for bootroms, make sure macro `INCLUDE_NET_INIT` is undefined and any macro that causes it to be defined such as `WDB_COMM_TYPE=WDB_COMM_END`.
3. Reset vector issue. The PPC405 reset vector is at physical address `FFFFFFFC`. With SystemACE at powerup, the processor will be prevented from executing an instruction at this address in some cases. With a VxWorks bootrom in the bitstream, SystemACE will load the FPGA IP cores, then the bootrom, place the PC at the bootrom entry point `romInit` and release it to begin fetching instructions. So what happens when you press the reset button? The answer is the processor will vector to `FFFFFFFC` and something had better be there. The BSP can be configured to initialize this reset vector with code to cause a System ACE jtag reboot. Defining `INCLUDE_XSYSACE_INSTALL_RESET_VEC` will install this code but a better solution would be to place something at the reset vector with Data2Bram in the Xilinx design flow toolchain.

Errata

1. Since there is no NVRAM support, the bootline is hardcoded in the `DEFAULT_BOOT_LINE` macro defined in `config.h`. The Ethernet MAC address is hardcoded in `sysNet.c`.
2. Pressing the CPU reset button will not reset the system and reload the bootrom. Users must press the System ACE reset button or place code in the BRAM at the end of the memory map that triggers system ACE to reset itself.

3. Ethernet 100Base-T may have poor performance with some host computers. If this is the case, then switch your host computer's ethernet adapter to 10Base-T.
4. When creating a Tornado Project using this BSP as the "basis BSP", ensure the data cache is disabled. Go to the "enable caches" component of the memory folder and set `USER_D_CACHE_ENABLE` to no value.

ML300 1.2/1 - November 13, 2002

The first release using an EDK/platgen generated bitstream.

HW Supported

- 16550 UART on P107 (VxWorks console)
- 16550 UART on P106
- EMAC Ethernet
- 128MB DDR RAM
- 32KB BRAM
- PPC MMU & Instruction & Data caches
- PLB/OPB Bridge
- GPIO (LEDs & Switches)
- IIC including NVRAM, temperature & power monitors.

HW Not supported

- System ACE
- LCD Display
- PCI
- Parallel Port
- PS2 Ports
- USB Ports
- Audio Ports
- SPI

Usage Notes

At the time this document was updated, this BSP is largely untested on a real EDK/platgen bitstream load.

Errata

1. System mode debugging through the END connection does not work.
2. Serial port usage as the WDB target connection does not work. Serial port polling mode does not seem to work.

ML300 1.2/2 - January 10, 2003

Tested using the EDK/platgen reference design bitstream.

Added support for SystemACE HW.

Usage Notes

1.

Errata

1. System mode debugging through the END connection does not work.
2. Serial port usage as the WDB target connection does not work. Serial port polling mode does not seem to work.

ML300seg 1.2/0 (seg 092402) - November 13, 2002

Written using the ML300 HW as a testbed and the 9/24/02 hand-coded version of the SEG IP load. This release supports more HW including the data cache, LCD, IIC, GPIO LEDs and switches.

This is in reality a renamed ML300 1.2/0 BSP with additional HW support and other refinements. The ML300 1.2/x BSP will be based on EDK/platgen bitstreams.

HW Supported

- 16550 UART on P107 (VxWorks console)
- 16550 UART on P106
- EMAC Ethernet
- 126MB DDR RAM
- 32KB BRAM
- LCD Display
- System ACE
- PPC MMU & Instruction & Data caches
- PLB/OPB Bridge BEAR & BESR register access (seg version)
- GPIO (LEDs & Switches)
- IIC including NVRAM, temperature & power monitors, and LCD brightness.

HW Not supported

The SEG IP load used by this BSP does map control registers for the following devices, however there is no BSP support for them.

- PCI
- Parallel Port
- PS2 Ports
- USB Ports
- Audio Ports
- SPI

Usage Notes

None.

Errata

1. System mode debugging through the END connection does not work.
2. Serial port usage as the WDB target connection does not work. Serial port polling mode does not seem to work.

References

- VxWorks 5.4 Programmer's Guide
- *Tornado 2.0 User's Guide*

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
08/01/02	1.0	Xilinx internal release.
08/21/02	1.1	Added alternate evaluation board usage notes. Minor touch ups.
09/19/02	1.2	Changes to accommodate ML300 vs. ML3. Changed memory map to 128MB RAM space from 256MB. Added LCD support, config.h options and annotated mmu description. Documented usage notes for BSP version 1.2/1 seg 08xx02. This version of the document was released as part of the world-wide fae training conference held in October of 2002.
10/22/02	1.3	Added documentation for additional HW support of IIC devices. More detail of bootrom operations. Updates for ML300seg 1.2/0.
01/10/03	1.4	Updates for ML300 BSP 1.2/2.

Device Driver Summary

Summary

A summary of each device driver is provided with a link to its main header file. In addition, building block components are described. A hardware-to-software driver cross-reference table is also provided.

Device Driver Reference

ATM Controller

The Asynchronous Transfer Mode (ATM) Controller driver resides in the atmc subdirectory. Details of the driver can be found in the xatmc.h header file

Ethernet 10/100 MAC

The Ethernet 10/100 MAC driver resides in the emac subdirectory. Details of the driver can be found in the xemac.h header file.

Ethernet 10/100 MAC Lite

The Ethernet 10/100 MAC Lite driver resides in the *emaclite* subdirectory. Details of the driver can be found in the xemaclite.h header file.

External Memory Controller

The External Memory Controller driver resides in the *emc* subdirectory. Details of the driver can be found in the xemc.h header file.

General Purpose I/O

The General Purpose I/O driver resides in the gpio subdirectory. Details of the driver can be found in the xgpio.h header file.

HDLC

The HDLC driver resides in the *hdlc* subdirectory. Details of the driver can be found in the xhdlc.h header file.

Intel StrataFlash

The Intel StrataFlash driver resides in the flash subdirectory. Details of the driver can be found in the xflash.h header file.

Inter-Integrated Circuit (IIC)

The IIC driver resides in the iic subdirectory. Details of the driver can be found in the xiic.h header file.

Interrupt Controller

The Interrupt Controller driver resides in the intc subdirectory. Details of the driver can be found in the xintc.h header file.

OPB Arbiter

The OPB Arbiter driver resides in the opb_arbiter subdirectory. Details of the driver can be found in the xopb_arbiter.h header file.

OPB to PLB Bridge

The OPB to PLB bridge driver resides in the *opb2plb* subdirectory. Details of the driver can be found in the xopb2plb.h header file.

PLB Arbiter

The PLB arbiter driver resides in the *plbarb* subdirectory. Details of the driver can be found in the xplbarb.h header file.

PLB to OPB Bridge

The PLB to OPB bridge driver resides in the *plb2opb* subdirectory. Details of the driver can be found in the xplb2opb.h header file.

Rapid I/O

The Rapid I/O driver resides in the rapidio subdirectory. Details of the 0 low leve driver can be found in the xrapidio_l.h header file

Serial Peripheral Interface (SPI)

The SPI driver resides in the spi subdirectory. Details of the driver can be found in the xspi.h header file.

System ACE

The System ACE driver resides in the *sysace* subdirectory. Details of the driver can be found in the xsysace.h header file.

Timer/Counter

The Timer/Counter driver resides in the tmrctr subdirectory. Details of the driver can be found in the xtmrctr.h header file.

UART Lite

The UART Lite driver resides in the uartlite subdirectory. Details of the driver can be found in the UART Lite Driver Datasheet and in the xuartlite.h header file.

UART 16450/16550

The UART 16450/16550 driver resides in the uartns550 subdirectory. Details of the driver can be found in the xuartns550.h header file.

Watchdog Timer/Timebase

The Watchdog Timer/Timebase driver resides in the wdttb subdirectory. Details of the driver can be found in the xwdttb.h header file.

Building Block Components

Common

Common components reside in the common subdirectory and comprise a collection of header files and ".c" files that are commonly used by all device drivers and application code. Included in this collection are: xstatus.h, which contains the identifiers for Xilinx status codes; xparameters.h, which contains the identifiers for the driver configurations and memory map; and xbasic_types.h, which contains identifiers for primitive data types and commonly used constants.

CPU/CPU_PPC405

CPU components reside in the cpu[_ppc405] subdirectory and comprise I/O functions specific to a processor. These I/O functions are defined in xio.h. These functions are used by drivers and are not intended for external use.

IPIF

IPIF components reside in the ipif subdirectory and comprise functions related to the IP Interface (IPIF) interrupt control logic. Since most devices are built with IPIF, drivers utilize this common source code to prevent duplication of code within the drivers. These functions are used by drivers and are not intended for external use.

DMA

DMA components reside in the dma subdirectory and comprise functions used for Direct Memory Access (DMA). Both simple DMA and scatter-gather DMA are supported.

Packet FIFO

Packet FIFO components reside in the packet_fifo subdirectory and comprise functions used for packet FIFO control. Packet FIFOs are typically used by devices that process and potentially retransmit packets, such as Ethernet and ATM. These functions are used by drivers and are not intended for external use.

Hardware/Software Cross Reference

Table 9-1: **Hardware and Software Cross Reference**

Hardware Device	Software Driver
DCR Bus Structure	XIo
DCR Interrupt Controller (INTC)	XIntc
OCM Packet Processing Engine	
OPB <-> PCI Full Bridge	XPci
OPB 10/100M Ethernet Controller	XEmac

Table 9-1: **Hardware and Software Cross Reference *(Continued)***

Hardware Device	Software Driver
OPB 10/100M Ethernet Controller - Lite	XEmacLite
OPB 16450 UART Controller	XUartNs550
OPB 16550 UART Controller	XUartNs550
OPB Arbiter and Bus Structure	XOpbArb
OPB ATM Utopia Level 2 Master	XAtmc
OPB ATM Utopia Level 2 Slave	XAtmc
OPB External Memory Controller (EMC)	XEmc
OPB GPIO Controller	XGpio
OPB IIC Master and Slave Bus Controller	XIic
OPB Interrupt Controller (INTC)	XIntc
OPB IPIF	XIpIf
OPB JTAG UART	XUartLite
OPB PS/2 Controller	
OPB Single Channel HDLC Controller	XHdlc
OPB SPI Master and Slave Bus Controller	XSpi
OPB TimeBase / WatchDog Timer	XWdtTb
OPB Timer / Counter	XTmrCtr
OPB Touchscreen Controller	
OPB UART - Lite	XUartLite
OPB2PLB Bridge	XOpb2Plb
PLB 1Gb Ethernet Controller	
PLB Arbiter and Bus Structure	XPlbArb
PLB External Memory Controller (EMC)	XEmc
PLB IPIF	XIpIf
PLB Packet Processing Engine	
PLB TFT VGA LCD Controller	
PLB UART-16450	XUartNs550
PLB UART-16550	XUartNs550
PLB2OPB Bridge	XPlb2Opb
RAPID IO	Xrapidio

Chapter 10

Automatic Generation of Tornado 2.0 (VxWorks 5.4) Board Support Packages

Overview

One of the key embedded system development activities is the development of the Board Support Package (BSP). Creation of a BSP can be a lengthy and tedious process that must be incurred every time the microprocessor complex (processor plus associated peripherals) changes. While managing these changes applies to any microprocessor-based project, the changes can come about more rapidly than ever with the advent of programmable System-on-Chip (SoC) hardware.

This document describes a tool, BSP Generator (*BSPgen*), which automatically generates a customized BSP for various microprocessor, peripheral, and RTOS combinations. This tool enables embedded system designers to:

- Substantially decrease development cycles (decrease time-to-market)
- Save years of development effort
- Create a BSP which matches the application (customized BSP)
- Eliminate BSP design bugs (automatically created based on certified components)
- Allow inclusion of customer-specific device drivers (provides a standard interface)
- Enable application software developers (don't have to wait for BSP development)

BSPgen is currently used in conjunction with the VirtexII-Pro and MicroBlaze system generation tools. Through these tools, the user can choose to automatically create a BSP based on embedded system just created. 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 type of operating system, processor, and peripherals chosen by the user for the FPGA-based embedded system.

The only type of BSP currently supported by *BSPgen* is for the WindRiver VxWorks 5.4 operating system and Tornado 2.0.2 IDE, in conjunction with the IBM PowerPC 405 microprocessor core.

The system generation tools provide a description of the embedded system to *BSPgen*. Using this system description and a set of template files pertaining to the operating system and processor selected, *BSPgen* generates a customized BSP.

Generating the BSP

User Interface

BSPgen supports a command-line interface and an Application Programmer Interface (API) using a Java class package. The command-line interface is specifically geared for the Xilinx Embedded Development Kit (EDK) tools. It requires system description files in the form of .mss/.mhs files that are output by the EDK tools. The command-line usage syntax is as follows:

Usage: bspgen -h <mhsfile> -s <mssfile> -p <project_path>

where:

-h <mhsfile>

Specifies the name of the .mhs file created by the MDT toolset. The .mhs file describes the hardware selected by the user for the embedded system.

-s <mssfile>

Specifies the name of the .mss file created by the MDT toolset. The .mss file describes the software, or device drivers, selected by the user and corresponding to the system hardware.

-p <project_path>

The absolute path of the user's MDT project directory.

BSPgen makes use of the XILINX_EDK environment variable. It should be set to the installation directory of the EDK.

Note that the end user does not typically invoke *BSPgen*. Instead, the EDK tools invoke *BSPgen* using the appropriate interface.

User Input

When choosing to automatically generate a BSP, the user is required to enter the following information:

- Type of operating system

  The WindRiver VxWorks 5.4 operating system is the only operating system currently supported. This implies the use of the Tornado 2.0.2 IDE. This section of the user's guide pertains only to a VxWorks 5.4/Tornado 2.0.2 Board Support Package.

- Directory location where the BSP will reside

  In the Tornado 2.0.2 case, this directory location typically resides in the standard *target/config* directory within the Tornado distribution directory tree. However, the user is free to choose another location for the BSP.

- Name of the BSP

  The name chosen by the user to identify the board on which the FPGA-based system resides. This name will be used throughout the generated BSP source files.

Template-Based Approach

A set of BSP template files will be released with *BSPgen*. Every operating system supported will have a corresponding set of template files. These template files are used during creation of the BSP, making appropriate modifications based on the makeup of the FPGA-based embedded system.

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

Device Drivers

A set of device driver source files will be released with the EDK tools and will 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 by the user if necessary and/or relocated if necessary. If the user makes changes to the device driver source code for this BSP and sometime later wishes to back those changes out, the user can use the EDK tools to regenerate the BSP. Device driver source files are then recopied from the installation directory to the BSP.

Backups

If the directory location of the BSP contains existing files, these files are copied into a backup directory before being overwritten. This prevents the inadvertent loss of changes made by the user to BSP source files. The backup directory will reside within the BSP directory and will be named *backup<timestamp>*, where *<timestamp>* represents the current date and time.

The Tornado 2.0 BSP

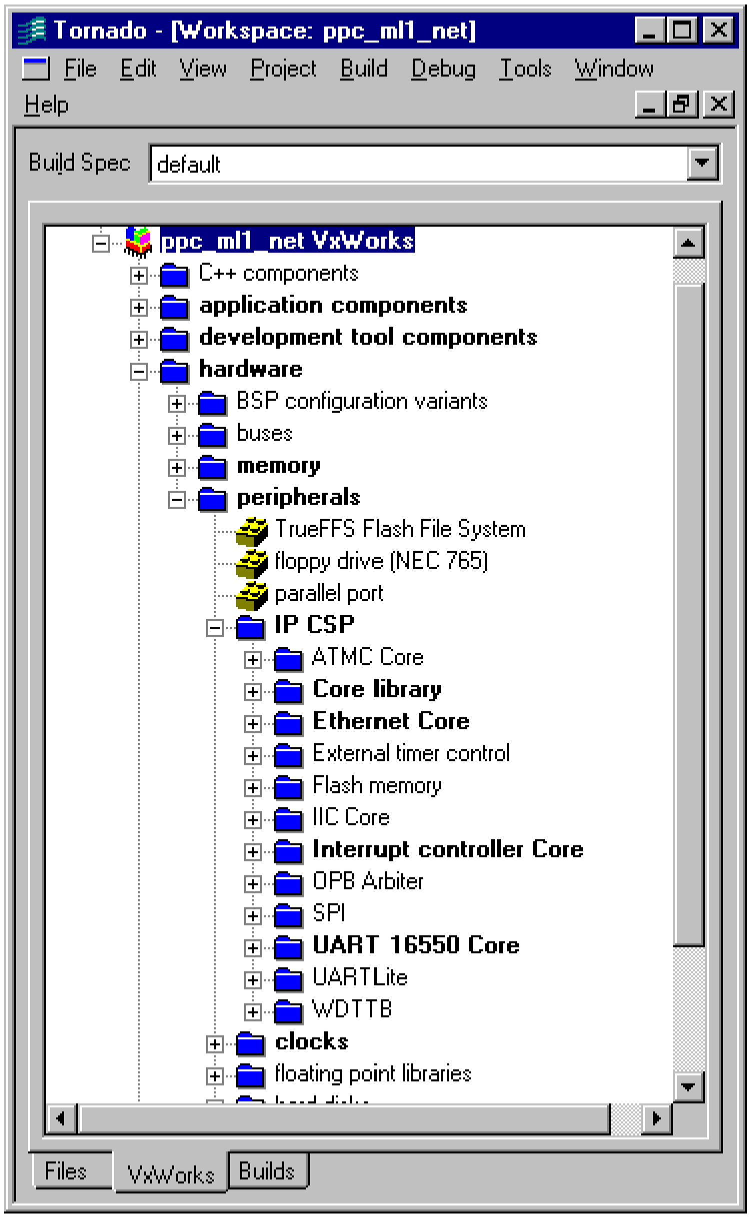
This section assumes the reader is familiar with WindRiver's Tornado 2.0.2 IDE.

Capabilities and Features

Integration with IDE

The automatically generated BSP is integrated into the Tornado 2.0.2 IDE and Project facility. The BSP can be compiled from the command-line using the Tornado make tools, or from the Tornado Project facility (also referred to as the Tornado GUI). Once the BSP has been generated, the user can simply type *make vxWorks* from the command-line to compile a bootable RAM image. This assumes the Tornado environment has been previously set up. If using the Tornado Project facility, the user can create a project based on the newly generated BSP, then use the build environment provided through the GUI to compile the BSP.

The file *50<csp_name>.cdf* resides in the BSP directory and is tailored during creation of the BSP. This file integrates the CSP device drivers into the Tornado GUI. CSPs hook themselves into the BSP at the **hardware/peripherals** sub-folder. Below this is a **Core library** folder and individual device driver folders. Figure 1 shows the look of the GUI given the CSP name "IP".

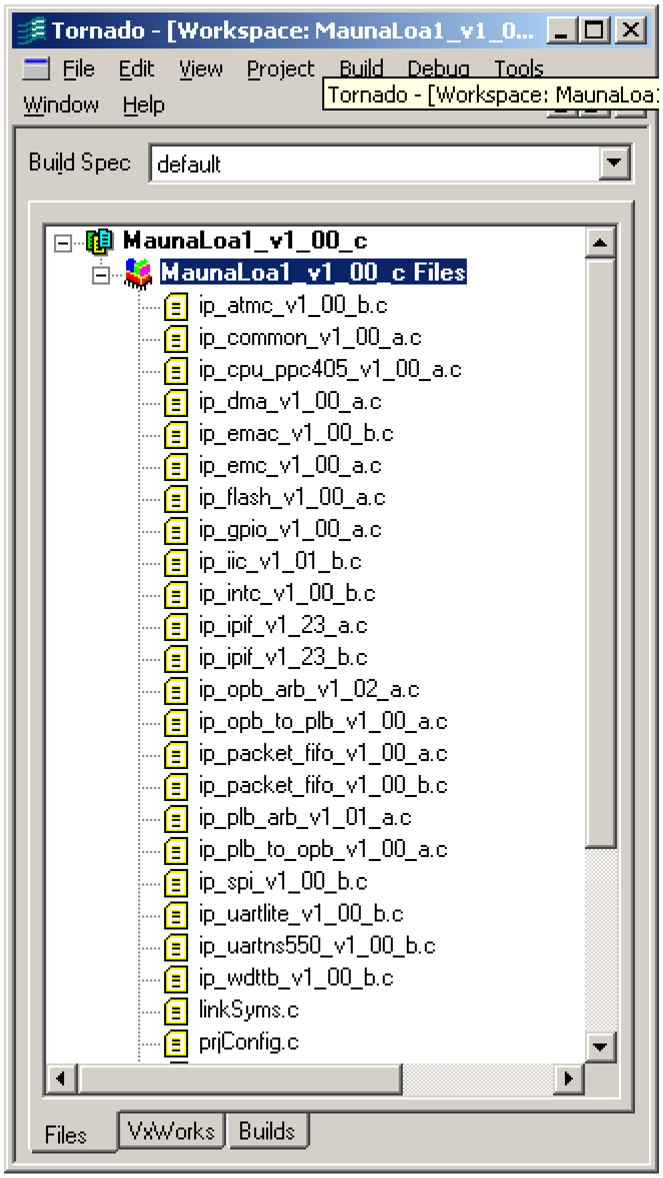

Tornado - [Workspace: ppc_ml1_net]
File Edit View Project Build Debug Tools Window
Help
Files
VxWorks
Builds

Build Spec: default

```
ppc_ml1_net VxWorks
  C++ components
  application components
  development tool components
  hardware
    BSP configuration variants
    buses
    memory
    peripherals
      TrueFFS Flash File System
      floppy drive (NEC 765)
      parallel port
      IP CSP
        ATMC Core
        Core library
        Ethernet Core
        External timer control
        Flash memory
        IIC Core
        Interrupt controller Core
        OPB Arbiter
        SPI
        UART 16550 Core
        UARTLite
        WDTTB
      clocks
      floating point libraries
```

Figure 1: **Tornado 2.0 Project GUI - VxWorks**

The "Files" tab of the Tornado Project GUI will also show a number of new files used to integrate the CSP device drivers into the Tornado build process. Once again, these files are automatically created by *BSPgen*. The user need only be aware of that the files exist. These files are prefixed with the name of the CSP. Figure 2 shows an example of the CSP build files.


Tornado - [Workspace: MaunaLoa1_v1_0...

Build Spec: default

```
MaunaLoa1_v1_00_c
  MaunaLoa1_v1_00_c Files
    ip_atmc_v1_00_b.c
    ip_common_v1_00_a.c
    ip_cpu_ppc405_v1_00_a.c
    ip_dma_v1_00_a.c
    ip_emac_v1_00_b.c
    ip_emc_v1_00_a.c
    ip_flash_v1_00_a.c
    ip_gpio_v1_00_a.c
    ip_iic_v1_01_b.c
    ip_intc_v1_00_b.c
    ip_ipif_v1_23_a.c
    ip_ipif_v1_23_b.c
    ip_opb_arb_v1_02_a.c
    ip_opb_to_plb_v1_00_a.c
    ip_packet_fifo_v1_00_a.c
    ip_packet_fifo_v1_00_b.c
    ip_plb_arb_v1_01_a.c
    ip_plb_to_opb_v1_00_a.c
    ip_spi_v1_00_b.c
    ip_uartlite_v1_00_b.c
    ip_uartns550_v1_00_b.c
    ip_wdttb_v1_00_b.c
    linkSyms.c
    prjConfig.c
```

Figure 2: **Tornado 2.0 Project GUI - Files**

Device Integration

Devices in the FPGA-based embedded system have varying degrees of integration with the VxWorks operating system. The degree of integration is currently fixed, but may be selectable by the user in the future. Below is a list of currently supported devices and their level of integration.

- A UART 16450/16550/Lite is integrated into the VxWorks Serial I/O (SIO) interface. This makes the UART available for file I/O and printf. Only one UART device can be selected as the console, where standard I/O (stdin, stdout, and stderr) is directed.
- An Ethernet 10/100 MAC is integrated into the VxWorks Enhanced Network Driver (END) interface. This makes it available to the VxWorks network stack and thus socket-level applications.
- An Interrupt controller is connected to the VxWorks exception handling and the PowerPC 405 external non-critical interrupt pin.

- All other devices and associated device drivers are not tightly integrated into a VxWorks interface. Access to these devices is available through direct access to the associated device drivers.

Device Driver Location and BSP Directory Tree

The automatically generated BSP contains boot code, device driver code, and initialization code. The BSP resembles most other Tornado BSPs except for the placement of device driver code. Off-the-shelf device driver code distributed with the Tornado IDE typically resides in the *target/src/drv* directory in the Tornado distribution 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 below.

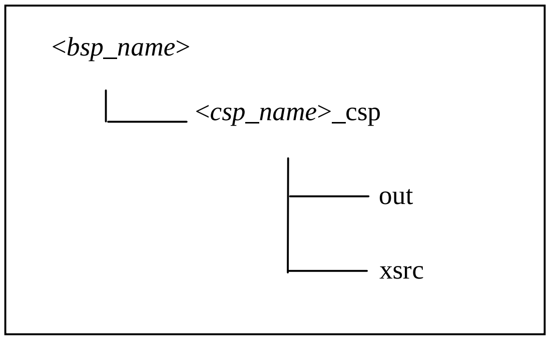


Figure 3: **BSP directory tree**

The top-level directory is named according to the name of the BSP the user provides. The customized BSP source files reside in this directory. There is a subdirectory within the BSP directory named according to the name of the CSP the user provides. The CSP directory contains two subdirectories. The *xsrc* subdirectory contains all the device driver related source files. The *out* subdirectory is created during the build process and only exists if building from the command-line. It contains files generated during the compilation or build process (e.g., the .o files for each driver source file). If building from the Project facility, the files generated during the build process reside at $PRJ_DIR/$BUILD_SPEC/<*csp_name*>_csp.

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. 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 all the necessary device drivers represented in the FPGA-based embedded system. Some of the devices are also integrated to some degree with the operating system.

Chapter 8

Insight MDFG456 Tornado 2.0 BSP User's Guide

Overview

The purpose of this document is to provide an introduction to the Tornado 2.0 BSP as implemented on the Insight MDFG456 reference board equipped with the Virtex II-Pro FPGA.

The addition of the Chip Support Package (CSP) into a Tornado 2.0 BSP is a unique challenge because of the nature of how easily hardware is added and removed from the FPGA using System Build Generator and how difficult it is to accommodate this feature into a Tornado 2.0 BSP. The CSP is a part of the BSP in that it provides the software drivers for hardware IP utilized by the BSP and application code. The CSP is designed to be primarily operating system independent so in many respects it is segregated and independently configured from the BSP.

The reader is expected to understand how Tornado 2 BSPs operate in general.

Requirements

Tornado 2.0.2

The user should have Wind River Tornado 2.0.2 installed on their PC with the PPC405 libraries.

Patches required that can be found at Wind River's Windsurf technical support web site:

- SPR67953. Cumulative patch
- DosFs 2.0. Dos file system support. This package is required if you wish to use the SystemACE compact flash device as an external storage device.

SingleStep (XE)

The XE stands for Xilinx Edition. This version of the SingleStep debugger is VirtexII Pro aware. This debugger works in concert with the VisionProbe debugger pod which connects to the "CPU Debug" port of the reference board.

Installation

Copy the entire MDFG456 source tree to $WIND_BASE\config\MDFG456 and perform the following operations from the DOS command-line:

```
C:\> make clean
C:\> make release
```

When this process finishes, a new project is placed at `$WIND_BASE\proj\MDFG456_vx`. As an alternative to this procedure, the Tornado project facility can be used to create a bootable application using the MDFG456 as the basis BSP. When creating a project in this way, the BSP can be located anywhere. See Project Facility documentation from Wind River.

Compact Flash

A compress zipfile is provided in the ace subdirectory. This is a complete image containing bootrom and sample VxWorks images in ace file format. See the `README` in the ace directory for more information.

To use the compact flash / System ACE solution, the

To install this image, do the following:

1. Make a backup of your microdrive then erase all files from it.
2. Uncompress the ace/compactFlash.zip file to the microdrive.
3. Insert the microdrive into the compact flash slot.
4. Connect a serial port cable to the P106 connector on the evaluation board. Default comm settings are 19200, N, 8, 1.
5. Set the rotary switch on the system ace adapter board to setting 6 and apply power. At this point, the VxWorks bootrom should be running and writing to the console serial port.
6. Set the bootrom boot line per your requirements. See **Bootrom Programming, page 100** for more information.

Setting Ethernet MAC Address

To verify your MAC address is correct, perform the following steps:

1. Set the rotary switch associated with the VxWorks bootrom and reboot the MDFG456.
2. Interrupt the countdown sequence to get the `[VxWorks Boot]:` prompt.
3. Enter the "N" command (case sensitive). The current MAC will be displayed and you will be prompted to enter a new MAC. The first three bytes of the MAC should be `000A35`.

```
Press any key to stop auto-boot...
 1
[VxWorks Boot]: N
Current Ethernet Address is: 00:0a:35:00:03:20
Modify only the last 3 bytes (board unique portion) of Ethernet Address.
The first 3 bytes are fixed at manufacturer's default address block.
00- 00
0a- 0a
35- 35
00-
```

4. If the MAC is valid, then enter return three times to accept the default. On new boards, the address may be all FFs. If this is the case, enter the last three bytes that are assigned to the serial number. If you are not sure of the numbers, then enter return three times. This will change the MAC to `00:0a:35:FF:FF:FF`. This will provide you with a valid MAC until the correct number is obtained. Boards with a MAC of all FFs will not

be capable of running the network stack. Multiple boards connected to the same network with the same MAC will not work either.

Files & Directories

While the root directory of the BSP can be placed anywhere, it is typically located at `$WIND_BASE/target/config/MDFG456`. The Tornado Project component of the BSP is located at `$WIND_BASE/target/proj/MDFG456_vx`.

The project component of the BSP is required if it will be configured/compiled with the Tornado Project Facility IDE. Normally, the Project Facility is utilized during application development and trivial BSP tweaks. The non-project component (also referred to as the command-line Tornado 1.0.1 BSP) is utilized during BSP development. Note that the methods of configuring and building the BSP differ greatly between the Project and command-line methods. See Tornado documentation for more information.

The CSP adds a directory structure not usually seen with VxWorks BSPs. It has been added to segregate BSP files from the CSP.

The following directories make up the MDFG456 BSP:

config/MDFG456

The traditional directory for Tornado 2.0 BSPs. Contains BSP library source code and the command-line makefile.

config/MDFG456/net

Contains Tornado Project "configlette" network source code that overrides configlettes located at `$WIND_BASE/target/config/comps/src/net`.

config/MDFG456/ace

Contains the bitstream and compact flash image which in itself contains the bootrom and other sample VxWorks ace and elf images.

config/MDFG456/ip_csp

The base directory for the CSP.

config/MDFG456/ip_csp/xsrc

Contains source code for the CSP.

proj/MDFG456_vx

The base directory for a Tornado project. All files here are maintained by the Project Facility.

proj/MDFG456_vx/<build spec>

A build specification maintained by the Project Facility. There is typically a "default" build spec here unless removed by the developer. Other build specifications can be added by the developer.

CSP Driver Organization

This section briefly discusses how the CSP is compiled and linked and eventually used by Tornado makefiles to include into the VxWorks image.

CSP drivers are implemented in "C" and can be distributed among several source files unlike traditional VxWorks drivers which consist of single "C" header and implementation files. For this reason, we place all CSP files in their own `xsrc` subdirectory.

There are up to three components for CSP drivers:

- Driver source inclusion.
- OS independent implementation

- OS dependent implementation (optional).

"Driver source inclusion" refers to how CSP drivers are compiled. For every CSP driver, there is a file named `ip_<dev>_<version>.c`. This file `#include`'s each CSP driver source file(s) (*.c) for the given device.

This process is analogous to how VxWorks' sysLib.c `#include`'s source for Wind River supplied drivers. The reason why CSP files are not simply `#include`'d in sysLib.c like the rest of the drivers is due to namespace conflicts and maintainability issues. If all CSP files were part of a single compilation unit, static functions and data are no longer private. This places restrictions on the CSP device drivers and would take away from its 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 CSP build.

Configuration

This BSP is configured just like any other Tornado 2 BSP. There is not much configurability to CSP drivers since the IP hardware has been pre-configured in most cases by System Build Generator. The only configuration available generally is whether the driver is included in the CSP at all. How to go about 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 CSP device driver does not mean that driver will be automatically utilized. Most CSP 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.

Command-Line

A set of constants (one for each driver) are defined in `config/MDFG456/ip_config.h` and follow the format:

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

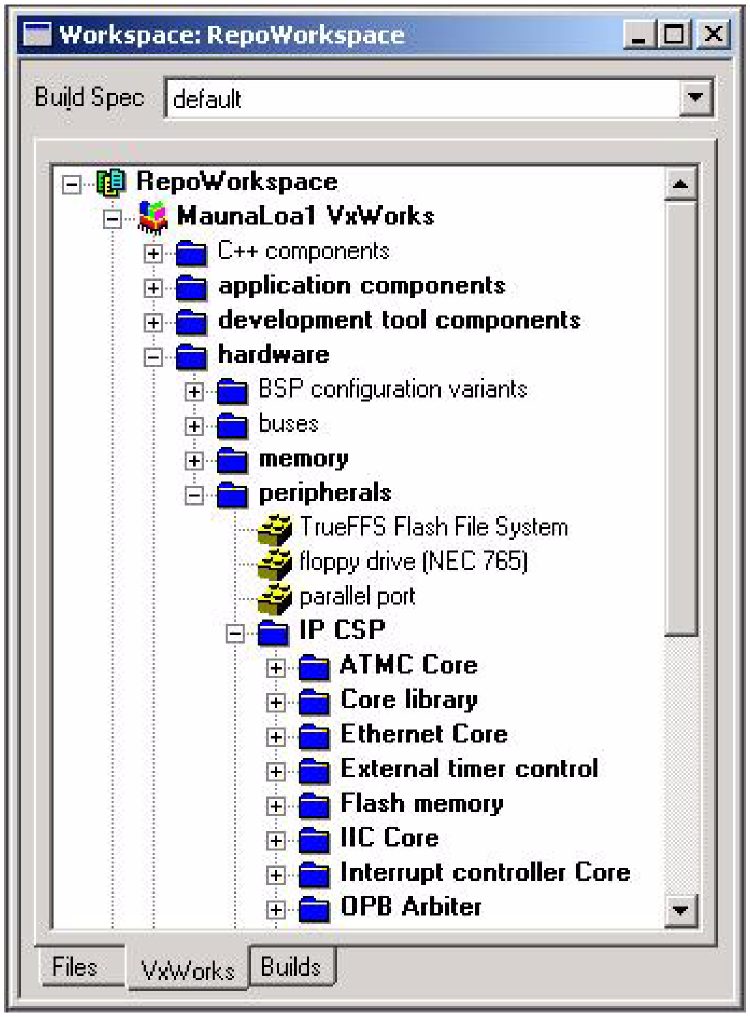
This file is included near the top of `config/MDFG456/config.h`. By default all drivers are included in the build. To exclude a driver, add the following line in `config.h` after the `#include "ip_config.h"` statement.

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

This 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

The Project Facility is part of the Tornado IDE. It is a GUI driven environment. To add/delete CSP drivers, go to the VxWorks pane in the workspace window (see figure below). Then add/delete driver components under IP_CSP just as you would with any other VxWorks component.


Workspace: RepoWorkspace

Build Spec: default

- RepoWorkspace
  - MaunaLoa1 VxWorks
    - C++ components
    - application components
    - development tool components
    - hardware
      - BSP configuration variants
      - buses
      - memory
      - peripherals
        - TrueFFS Flash File System
        - floppy drive (NEC 765)
        - parallel port
        - IP CSP
          - ATMC Core
          - Core library
          - Ethernet Core
          - External timer control
          - Flash memory
          - IIC Core
          - Interrupt controller Core
          - OPB Arbiter

Figure 1: **Project Facility GUI Configuration**

Note that whatever configuration has been specified in `ip_config.h` and `config.h` will be overridden by the project facility.

Memory Map

Due to the nature of this evaluation board a full memory map is not given in this document. The user is instead referenced to "C" source code header file `xparameters.h`. This source file provides a memory map for all CSP devices. A partial map is given here that relates directly to BSP operation.

Table 8-1: **System Memory Map**

Device	Start (hex)	End (hex)	Size (bytes)
PLB SDRAM	`00000000`	`01FFFFFF`	`32 MB`
OPB Space	`40000000`	`DFFFFFFF`	`2.5 GB`
PLB BRAM	`FFFF8000`	`FFFFFFFF`	`32 KB`

RAM Memory Map

RAM device contains the VxWorks runtime image and heap space. MDFG456 follows VxWorks conventions for RAM usage for PowerPC processors. Refer to Appendix F of the *VxWorks 5.4 Programmer's Guide*.

Table 8-2: **RAM Memory Map**

Physical Address Range (hex)	Usage
00000000..000000FF	(SDRAM) Unused & undefined
00000100..00002FFF	(SDRAM) Interrupt Vector table
00003000..00010000	(SDRAM) VxWorks usage. Exception reason message and other VxWorks constructs are at the bottom of this region. Initial stack is set at the top of this range and grows downward. Once VxWorks has switched to multi-tasking mode, this stack is no longer used.
00010000..00BFFFFF	(SDRAM) RAM_LOW_ADRS. VxWorks image, interrupt stack, host memory pool, and heap space.
00C00000..01FFFFFF	(SDRAM) RAM_HIGH_ADRS. Two possible uses. (1) VxWorks bootrom image and heap space. (2) VxWorks heap space.
48000000..480FFFFF	(SDRAM) This memory is resident on the P160 communications add-on module and is part of the Toshiba flash memory device. This memory area is used for network buffers.
FFFF8000..FFFFFFFF	(BRAM) Address FFFFFFFC contains reset vector.

Flash Memory Map

The P160 communications module contains 8MB of flash memory implemented on two Toshiba TH50VSF2581 parts. These parts are wired together to form a 32 bit memory width. The flash blocks are not of uniform size. Smaller "parameter" blocks reside at the beginning of the addressing range of these parts. The BSP uses the first one of these eight parameter blocks. The remaining parameter blocks are not used. The "main" blocks can be used to store a VxWorks image or any other user data. The "hidden" block is not utilized by this BSP.

Table 8-3: **Flash Memory Map**

Offset Byte Address Range (hex)	Usage
00000000..00003FFF	First parameter block used for NVRAM storage VxWorks boot line at offset `[0..255]` Ethernet mac address at offset `[3FFA-3FFF]`
00004000..0001FFFF	Remaining parameter blocks unused
00020000..007FFFFF	VxWorks images or other user data

OPB Memory Map

The P160 communications module contains 8MB of flash memory implemented on two Toshiba TH50VSF2581 parts. These parts are wired together to form a 32 bit memory width. The flash blocks are not of uniform size. Smaller "parameter" blocks reside at the beginning of the addressing range of these parts. The BSP uses the first one of these eight parameter blocks. The remaining parameter blocks are not used. The "main" blocks can be used to store a VxWorks image or any other user data. The "hidden" block is not utilized by this BSP.

Table 8-4: **OPB Memory Map (unused areas not shown)**

Physical Address Range (hex)	Usage
48000000..480FFFFF	SRAM
5FFFFF00..5FFFFFFF	EMC control
60000000..6000FFFF	OPB peripherals
DE000000..DE7FFFFF	Flash memory

NVRAM

NVRAM support is provided by the Toshiba flash memory. The first parameter block of this flash array is reserved for NVRAM. A special NVRAM to flash driver is utilized by the BSP at $WIND_BASE/src/drv/mem/nvRamToFlash.c. This driver uses functions in sysFlash.c to read/write parameters to NVRAM.

When there is no flash support, the BSP will replace the NVRAM driver with `$WIND_BASE/src/drv/mem/nullNvRam.c` which provides only function stubs so that VxWorks will link. When this is the case, the default bootline is used (see `config.h`) and the Ethernet MAC address defaults to: `00:0a:35:00:00:00`.

Table 8-5: **NVRAM Memory Map**

Part Offset Range (hex)	sysNvRamGet/Set Offset	Usage
0000..00FF	`0000..00FF`	Reserved for VxWorks bootline
0100..3FF9	`0100..3FF9`	Unused
3FFA..3FFF	`3FFA..3FFF`	Ethernet MAC address

Caches

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

- `INCLUDE_CACHE_SUPPORT` - If `#define`'d, the VxWorks cache libraries are linked into the image. If caching is not desired, then `#undef` this constant.
- `USER_I_CACHE_ENABLE` - If `#define`'d, VxWorks will enable the instruction cache at boottime. Requires `INCLUDE_CACHE_SUPPORT` be `#define`'d to have any effect.

- `USER_D_CACHE_ENABLE` - If `#define`'d, VxWorks will enable the data cache at boottime. Requires `INCLUDE_CACHE_SUPPORT` be `#define`'d to have any effect.

The caches are configured by the following constants in `MDFG456.h`. These constants map to the PPC cache control registers of the same name. See PPC405 documentation for further information on these registers:

- `MDFG456_ICCR_VAL` - Initial contents of the ICCR register (instruction cacheability attribute).
- `MDFG456_DCCR_VAL` - Initial contents of the DCCR register (data cacheability attribute).
- `MDFG456_DCWR_VAL` - Initial contents of the DCWR register (write back/through attribute).
- `MDFG456_SGR_VAL` - Initial contents of the SGR register (guarded attribute).

Table 8-6: **Cache Map**

Physical Address Range (hex)	I Cache	D Cache	Write Back/Through	Guarded
00000000..01FFFFFF	Y	Y	Back	N
F8000000..FFFFFFFF	Y	Y	N/A	N
everything else	N	N	N/A	N

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

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

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 #define'd 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`. 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.

The PPC405 is capable of other attributes including zone protection, however, Wind River documentation is poor in this area and it is unclear whether the basic MMU package supports them. An add-on is available from Wind River (which is enabled by `INCLUDE_MMU_FULL`) for advanced MMU operations.

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 (see `MDFG456_ICCR_VAL` in previous section). This strategy can dramatically reduce the number of page faults while still keeping instructions in cache.

Exception Handling

There are two types of exceptions which are of importance to the BSP. The first type are internal exceptions such as machine check, illegal instruction, etc.. By default, the BSP configures VxWorks to trap these types of exceptions. When one occurs, the offending task is suspended and a descriptive message is displayed on the console. If the exception occurs in interrupt context, VxWorks will reboot itself.

The other type of exception are external asynchronous. The BSP initializes and handles these exceptions which are the result of an active signal on the external or critical interrupt pins of the processor.

There are two INTC IP devices within the FPGA, one connected to the processor's external interrupt and the other on the critical interrupt. Functions in BSP source code file `sysInterrupt.c` are responsible for initializing these two devices with the `XIntc` component driver and hooking them into VxWorks.

External Interrupts

Most IP peripherals that can generate interrupts are attached to the INTC component responsible for asserting the external interrupt processor exception. BSP initialization code hooks control of this device into the VxWorks `intLib` library.

External interrupt vectors are defined in `xparameters.h`. `MDFG456.h` may translate these vectors into `SYS_<device>_VEC_ID` to limit changes to BSP source code when device names change. These constants are utilized when invoking the VxWorks intLib functions. Example:

```
#include <intLib.h>

void foo(void)
{
   intEnable(SOME_DEVICE_VEC_ID);
}
```

Critical Interrupts

Since VxWorks does not define a critical interrupt API as it does for external interrupts, the user must utilize the API defined in `sysLibExtra.h`. Functions `sysIntCritConnect`, `sysIntCritEnable`, and `sysIntCritDisable` are designed to work identically to those for the external interrupt defined by the VxWorks `intLib.h` library. Example

```
#include "sysLibExtra.h"

void foo(void)
{
   sysIntCritEnable(SOME_CRITICAL_DEVICE_VEC_ID);
}
```

Note: PLB/OPB bridges & arbiters are wired to the critical interrupt handler. If these interrupt sources are enabled and the PPC machine check interrupt is enabled then VxWorks may reboot when an exception occurs. This is because the PLB/OPB bridge/arbiter will most likely interrupt when a transaction cannot complete. At the same time the PPC will detect a bad bus cycle and generate a machine check exception. This will lead to the VxWorks exception handler being interrupted. VxWorks architecturally does not allow this and will reboot the system when it occurs.

It is not recommended to `sysIntCritEnable()` one of these interrupt sources. Instead, use the VxWorks `excHookAdd()` function to use your own function perform custom exception processing (after VxWorks finishes its own processing). Here, the hook function can examine the bridges/arbiters and perform whatever task is required for the event.

System ACE

The System ACE controller is a device that provides a way to store multiple FPGA bitstream loads. These loads are stored on a compact flash (CF) device and downloaded by the System ACE controller into the FPGA when the system is powered up. Additionally, these bitstreams can contain a software load that is downloaded to RAM after the FPGA's IP cores have been programmed. These bitstream loads are stored in the CF device in a DOS filesystem. This means regular files can be accessed from the CF as well. Such files include VxWorks ELF images, application code & data, and text script files.

The BSP utilizes the SystemACE controller in two ways. First as a boot device and second as an external storage device. Both applications require the following constants be defined in `config.h` or by using the Tornado Project facility to change the constants of the same name:

- `SYS_SYSACE_DEV_ID` - Should be set to the `xparameters.h` XPAR constant associated with the System ACE controller device identifier.
- `SYS_SYSACE_BASEADDR` - Should be set to the `xparameters.h` XPAR constant associated with the System ACE controller base address.

Note that System ACE is not supported in Rev 0 of the BSP.

DOS File System

When being used as a file storage device, 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 `#define`'d 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_CBIO

Automounting

To automatically mount the System ACE as a file system at boot time, `INCLUDE_XSYSACE_AUTOMOUNT` must be defined. In the Project facility, this is defined by enabling the automount feature in the System ACE folder. When defined, two more constants are utilized to mount the compact flash device: `SYSACE_AUTOMOUNT_POINT` and `SYSACE_AUTOMOUNT_PARTITION`. In the Project facility, these constants can be set by editing the System ACE properties folder. This relieves the application from having to initialize and mount the DOS File system. Note that this works only for Project builds. Command line builds require that the application invoke `sysSystemAceInitFS()` and `sysSystemAceMount()`. These functions are described in the Board API section below.

Board API

This section will not go over CSP device driver functions. Instead the user is directed to the appropriate `ip_csp/xsrc/<device>.c` file for documentation and usage.

There are a handful of "board level" BSP functions not implemented by the CSP device drivers. Prototypes for these functions are located in `config/MDFG456/sysLibExtra.h`.

Standard I/O

The BSP comes with stdin, stdout, and stderr directed through the UART on the P106 connector . The default UART baud rate is set to 115200, no parity, 8 data bits, and 1 stop bit. The secondary UART on P107 is enabled and ready for application usage. It defaults to 19200 baud, no parity, 8 data bits, and 1 stop bit.

GPIO

Two instances of GPIO can be included in the BSP. The first instance controls the momentary push button switches, their surrounding LEDs, and the bank of 8 DIP switches. The second GPIO instance controls the LCD display. Both instances require that `INCLUDE_XGPIO` constant be defined.

DIP switches 1-4 are reserved for the bitstream and for the BSP. Application code may use switches 5-8.

void sysLedOn(UINT32 mask)

Turns on LEDs in the mask. Bits set to one cause the associated LED to be illuminated. The mask is built using constants `GPIO_OUT_LEDx` defined in `MDFG456.h` where `x` is the LED number on the PCB. This function requires `INCLUDE_XGPIO` be defined.

void sysLedOff(UINT32 mask)

Turns off LEDs in the mask. Bits set to one cause the associated LED to be turned off. The mask is built using constants `GPIO_OUT_LEDx` defined in `MDFG456.h` where `x` is the LED number on the PCB. This function requires `INCLUDE_XGPIO` be defined.

UINT32 sysSwitchReadState(void)

Reads the state of all the push button switches. A mask is returned describing which switches are closed (i.e. being pushed). The mask is decoded using constants `GPIO_IN_PUSHx` defined in MDFG456.h where `x` is the switch on the PCB. This function requires `INCLUDE_XGPIO` be defined. Usage example:

```
UINT32 mask = sysSwitchReadState();
```

```
if (mask & GPIO_IN_PUSH3)
{
    // handle switch 3 press
}
```

UINT32 sysDipReadState(void)

Reads the state of all the DIP switches. A mask is returned describing which switches are in the "ON" position. The mask is decoded using constants `GPIO_IN_DIPx` defined in MDFG456.h where `x` is the DIP switch position on the PCB. This function requires `INCLUDE_XGPIO` be defined. Usage example:

```
UINT32 mask = sysDipReadState();

if (mask & GPIO_IN_DIP5)
{
    // handle DIP #5 being "on"
}
```

void sysLcdWriteInstruction(UINT8 data)

This function clocks in an instruction command to the LCD device. Application code can use this function to perform low level operations to the LCD display.

void sysLcdWriteData(UINT8 data)

This function clocks in data to the LCD device's internal RAM. This function is typically used to place a character somewhere in the device's memory. Application code can use this function to perform low level operations to the LCD display.

System ACE

These routines require that the `INCLUDE_XSYSACE` constant be defined. The command line BSP will not initialize the DOS file system resident on the compact flash. Application code will have to make function calls to initialize and mount:

```
FILE *fp;

sysSystemAceInitFS();
if (sysSystemAceMount("/cf0", 1) != OK)
{
   /* handle error */
}

fp = fopen("/cf0/myfile.dat","r");
         .
         .
```

STATUS sysSystemAceSetRebootAddr(unsigned configAddr)

Sets the reboot JTAG configuration address. This address is mapped to cfgaddr0..7 as defined in XILINX.SYS in the root directory of the CF device. If this function is never invoked, then the default address is used. The default address is the address selected by the rotary switch. The given address will be rebooted if `sysToMonitor()` or `reset()` is called.

The `configAddr` parameter range is 0..7 (i.e. cfgaddr0..7) or -1 to select the default address.

Returns ERROR if `configAddr` is out of range, OK otherwise.

void sysSystemAceInitFS(void)

Initializes the required Wind River DosFs 2.0 libraries. Application code is not required to call this function on a BSP built with the Project facility.

STATUS sysSystemAceMount(char* mountPoint, int partition)

Mount the compact flash as DOS file system volume. The `mountpoint` parameter is an arbitrary string labeling the device. Once mounted, refer to this mountpoint in all file accesses. The `partition` parameter specifies the partition to mount. If "0" is specified then the boot device is assumed to not contain a partition table (i.e. it is treated like a floppy disk).

Note: Before calling this routine, be sure to initialize the DOS file system with a call to `sysSystemAceInitFS()`.

Note: Application code is not required to call this function on a BSP built with the Project facility with `INCLUDE_XSYSACE_AUTOMOUNT` defined.

LCD

This board contains a LCD character display capable of displaying 1 or 2 lines of characters. Control of the display is handled using GPIO lines. The BSP sets up the LCD display for two lines using the 5x8 character matrix with no cursor. This setup provides a 2x16 character window.

Low level functions are available to write instructions and data to the device (see `sysLcdWriteInstruction()` and `sysLcdWriteData()` in section **GPIO, page 95**).

void sysLcdWriteString(char* string)

This function writes the given string to the LCD display. Before writing, the LCD display is cleared. The string parameter may truncate in the display if too long. A null string will clear the display. If string contains the newline character "\n", then characters following it will be placed in the 2nd line of the display. If more than one newline is present, then the text following the last newline will be written to the 2nd line of the display.

This function assumes LCD display characteristics have not been changed since sysLcdInit() was invoked at boot time. If display font or line mode have been changed, then the string may look scrambled in the display.

Miscellaneous Functions

void sysMsDelay(UINT32 delay)

Delay the specified number of milliseconds. The delay is implemented as a busy loop that occupies the CPU. The delay can be pre-empted by a higher priority task or interrupts if tasking/interrupts are enabled causing loss of delay precision.

void sysUsDelay(UINT32 delay)

Delay the specified number of microseconds. The delay is implemented as a busy loop that occupies the CPU. The delay can be pre-empted by a higher priority task or interrupts if tasking/interrupts are enabled causing loss of delay precision.

This function not accurate for delay times below 20us due to system overhead. The overhead is more or less constant and can be negated by the use of `SYS_US_DELAY_BIAS` defined in `config.h`. Use this constant to calibrate to your system's needs. As delivered with a 300 MHz CPU clock and a bias of -2, this function is accurate within +/-15% for a 20us delay. As the delay time increases, the accuracy increases.

Custom Options

This section discusses MDFG456 specific configuration options that can be set either in `config.h` or in the Project GUI. Unless otherwise stated, these options can be set by `#define`'ing or `#undef'`ing them in `config.h` or by defining them in the Project GUI in the project workspace's macros settings in the build tab.

Table 8-7: **Custom BSP Options**

Option	Description
INCLUDE_P160_COMM_MODULE	Controls whether code used to control peripherals is eligible to be compiled into the BSP. P160 has an ethernet controller, flash/SRAM memories, and a UART amongst other things.
INCLUDE_XSYSACE_INSTALL_-RESET_VEC	Controls whether reset code is placed the processor's reset vector address. This reset code will trigger SystemACE to load the default configuration bitstream.
INCLUDE_XSYSACE_AUTOMOUNT	Controls whether the System ACE filesystem is mounted at boot time using the next two `SYSACE_` constants defined in this table. This constant affects only Project builds.
SYSACE_AUTOMOUNT_POINT	Default mount point used when `INCLUDE_-XSYSACE_AUTOMOUNT` is defined in Project builds.
SYSACE_AUTOMOUNT_PARTITION	Default partition used when `INCLUDE_XSYSACE_-AUTOMOUNT` is defined in Project builds.
INCLUDE_BOOT_FLASH	Sets the BSP up to boot from Flash memory. If not defined, then SystemACE is assumed to be the bootstrap device.
SYS_US_DELAY_BIAS	Adds the specified number of microseconds to the delay parameter in `sysUsDelay()`. This option can be used to cancel out overhead.

Table 8-7: **Custom BSP Options**

Option	Description
INCLUDE_EMAC_PHY_RESET_-AT_BOOT	Controls whether the Ethernet PHY is reset at boot time. In the Project GUI, this is a parameter under the emac component and can be found under *hardware->peripherals->IP CSP-> Ethernet Core*. Set to TRUE to enable, FALSE to disable.
SYS_GPIO_SWITCH_DEBOUNCE_TICKS	Sampling interval used by function `sysSwitchReadState()` when attempting to debounce switches. Units are in clock ticks.
SYS_LCD_INIT_DISPLAY	If this character constant is defined, then it's value will be written to the LCD display at boot time. Requires GPIO support (`INCLUDE_XGPIO`).

Building VxWorks

The MDFG456 BSP follows the standard Tornado conventions when it comes to creating a VxWorks image. Refer to Tornado documentation on how to make a VxWorks image. This section discusses extensions made to the build process.

Command-Line BSP Build Extensions

The CSP is compiled/linked with the same toolchain VxWorks is created with. Only very minor additions are made to the Makefile to support the CSP build such as compiler directives telling the BSP where to find the CSP files and visa-versa.

Project BSP Build Extensions

There are no extensions to the Project build. The BSP should behave just like any other normal BSP.

Bootup Sequence

There are many variations of VxWorks images with some based in RAM, some in ROM. Not all these images are supported on the reference board when using System ACE. The following list discusses various image types:

- Compressed images - Not supported when using System ACE as the bootstrap loader. These images begin execution in ROM and decompress the image into RAM. SystemACE has no knowledge of the compression algorithm being used by VxWorks. Compressed images are allowed if flash memory is the bootstrap device.
- RAM based images - Fully supported.
- ROM based images - Fully supported. These images begin execution in ROM, copy themselves to RAM then transfer execution to and stay in RAM. System ACE performs the copy operation, so the BSP has been prepared to short circuit the VxWorks copy (see `romInit.s`).
- ROM resident images - Not supported when using System ACE as the loader. These images begin execution in ROM, copy the data section to RAM, and execution remains in ROM. Theoretically BRAM could be used as a ROM however the current VirtexII Pro parts being used in MDFG456s do not have the capacity to store a VxWorks image which could range in size from 200KB to over 700KB. ROM resident images are allowed if flash memory is the bootstrap device.

vxWorks

This image is meant to be downloaded to the target RAM space. Once downloaded, the PC should be set to function `_sysInit` (implemented in `sysALib.s`). Most of the time, the device performing the download will do this for you as it can extract the entry point from the image.

1. `_sysInit` : Low level initialization. Since this image is copied to RAM, the device that downloaded the image may have to perform manual system initialization to make RAM visible. When completed, this function will setup the initial stack and invoke the first "C" function `usrInit()`.
2. `usrInit()` : Performs pre-kernel initialization. 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. 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.

bootrom_uncmp

This 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 to perform tasks before jumping to RAM. This image behaves differently than a traditional bootrom due to the fact it is already in RAM when control is passed to it (via System ACE).

1. Power on. System ACE loads the bitstream into the FPGA then loads the bootrom image into RAM and passes control to assembly language function `_romInit` located in `romInit.s`.
2. `_romInit` : Traditionally this function would perform board level initialization then call `romInit()` which would copy the VxWorks image to RAM. Since the image is already in RAM, this function simply jumps to assembly function `_sysInit`.
3. Follows steps 1, 2 & 3 of the "vxWorks" bootup sequence.

Difference Between Command-Line & Project BSPs

Functions `usrInit()`, `usrRoot()`, and `romStart()` as explained in the boot sequence steps above are implemented by Tornado. In command line BSPs, these functions are defined in source code located at `$WIND_BASE/target/config/all`. In Project BSPs, the Project Facility generates this code in the user's project directory.

Functions `_sysInit`, `_romInit`, `sysHwInit()`, and `sysHwInit2()` are implemented by the BSP in `config/MDFG456`. These functions are utilized on both the command-line and project BSPs.

Bootrom Programming

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 full image. If the image is retrieved on the Ethernet, 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 set 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 development and replaced with the production VxWorks image.

Creating Bootroms

On a command line window, cd to the `config/MDFG456` directory. Issue a "`make bootrom_uncmp`". Run the batch file $WIND_BASE`\host\x86-win32\bin\torVars.bat` (if using Micro$oft Windows) to setup command line environment variables before building the bootroms.

The next step is to either test `bootrom_uncmp` by downloading it with an emulator or creating an .ace file out of it (combined with the IP core bitstream) for download by SystemACE. See VirtexII Pro documentation on how to create .ace files

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 1984-1998  Wind River Systems, Inc.

 CPU: MDFG456 VirtexII Pro PPC405
 Version: 5.4.2
 BSP version: 1.2/0
 Creation date: January 10 2003, 11:59:00


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

 [VxWorks Boot]:
```

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

Bootline

Non-volatile storage of the bootline requires NVRAM support which in itself requires IIC support. If NVRAM support is not present or an error occurs reading it, then the `DEFAULT_BOOT_LINE` is utilized. If NVRAM is uninitialized (such as it will be in new systems) then the bootline may be gibberish.

MDFG456 bootroms support the network interface and System ACE as the boot device. The bootline tells the bootrom how to find the vxWorks image. The bootline is maintained at runtime by the bootrom. 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 edit the bootline. Enter "p" to view the bootline. On a non-bootrom image, you can still change the bootrom by entering the `bootChange` command at a host or target shell prompt.

The following list goes over the meanings of the bootline fields:

- `boot device` : Choices are "xemac" or "sysace=x". When set to xemac, the BSP will boot over the network. When set to "sysace=x", the BSP will boot from a file resident on the System ACE device. See **Booting from SystemACE**, page 102 for further information on how to specify the System ACE boot device. 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.
- `host name` : Name as needed. Can be arbitrary.
- `file name` : The VxWorks image to boot. If the boot device is the network "xemac", then the file must be accessible on the host computer via ftp. See **Booting from SystemACE**, page 102 for specifying a System ACE file.
- `inet on ethernet (e)` : The IP internet address of the target. If there is no network interface, then this field can be left blank.
- `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)` : Whatever names suites you.
- `other (o)` : This field is not applicable when "xemac" is specified as the boot device. When "sysace" is the boot device, then this field should be set to "xemac". This will signal the VxWorks image specified in the `file name` field to start the network on the xemac device. (if network support was included)
- `inet on backplane (b)` : Leave blank. MDFG456 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`

Booting from SystemACE

The "boot device" field of the bootline is specified using the following syntax:

sysace=<partition number>

where `<partition number>` is the partition to boot from. Some CF devices do not have a partition table and are formatted as if they were a large floppy drive. 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 accessed.

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 file system partition. Simply build vxWorks using the Tornado tools then 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 `cf0` 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 DOS formatted disk, read the file into memory and begin execution.

2. Boot from an ace file. The ace file can contain HW only, SW only, 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 correct memory address ranges, 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. ace files are created with the Xilinx ISI tools and is beyond the scope of this manual.

   The "file name" must have the following syntax:

   ```
   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 selected by the rotary switch on the MDFG456 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.

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:/tornado/target/config/MDFG456/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.

```
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
```

Deviations

This section sums up the difference between garden variety BSPs and the MDFG456. The differences between the two fall roughly into key areas: CSP and System ACE support.

The CSP contains drivers for the Xilinx IP cores (see **CSP Driver Organization**, page 87). To keep the BSP buildable while maintaining compatibility with the Tornado Project facility, a set of files named `ip_<driver>_<version>.c` populate the BSP directory that simply `#include` the source code from the CSP.

The location of the CSP relative to the BSP directory causes problems because command line and Project facility differ in how BSP files are found during compilation. To address this issue, a key Project macro (BSP_DIR) is defined in the BSP's `Makefile`. Of all deviations, this one is the most dangerous because future versions of Tornado may cause builds to fail. The `Makefile` contains more information about this deviation.

System ACE, being a boot device and a DOS file system, has required that two VxWorks source code files found in the Tornado distribution be changed. Wind River allows BSP developers to change some source code files provided they follow set guidelines. The two files that have been modified from their original version are `bootConfig.c` and `net/usrNetBoot.c`.

`usrNetBoot.c`, used only by Project Facility builds, required a 1 line of code change to tell VxWorks that the System ACE device is a disk based system like IDE, SCSI, or floppy drives. This change allows the BSP to properly process the "other" field of the bootline (see **Bootline**, page 101) when System ACE is the boot device. The "other" field allows the selection of a network device when booting from a disk based system.

`bootConfig.c`, used only by bootroms builds, required extensive modifications to support SystemACE as a boot device. These mods are bracketed by `INCLUDE_XSYSACE` preprocessor ifdefs. Another mod enabled the data cache when ethernet frames are copied from fifos instead of DMA. This change greatly increases the bootup times for the system but could cause problems if another device required for booting utilizes DMA or requires some sort of special cache coherency in the first 128MB of address space.

Limitations

This section goes over what this BSP cannot do and the reasons why. It also goes over what-if scenarios when key pieces of IP are not part of the FPGA load.

No WARM boots when System ACE is the bootstrapping device

All boots are cold. There is no distinction between warm, cold, or any other type of boot. This is because reboots are managed by the System ACE device which resets the processor whenever it performs an ace download.

This could cause an exception message generated by VxWorks to not be printed to the console when the system is rebooted due to an exception in an ISR or a kernel panic. See troubleshooting guide for tips to get at this exception message.

No compressed images when System ACE is the bootstrapping device

If you compile a compressed image then try to boot it as an ace file, results will be undetermined. This is because System ACE cannot decompress data as it writes it to ram.

Command line builds cannot initialize the network when System ACE is the boot device

This requires that the application provide code to initialize the network. Project builds can get around this because a modified `net/usrNetBoot.c` is provided in the BSP directory (see **Deviations**, page 104). The equivalent file for command line builds is located at `$WIND_BASE/target/src/config/usrNetwork.c`. The architecture of the command line build prevents us from overriding this file with a clone in the BSP directory.

Fixing `usrNetwork.c` requires changing the following code in function `usrNetInit()`:

```
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

```
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.

Reset Vector

On the PPC405 processor, the reset vector is at physical address 0xFFFFFFFC. There is a short time window where the processor will attempt to fetch and execute the instruction at this address.This window is between the time when System ACE has finished downloading the HW bit stream and before it begins to download the SW image. All VxWorks requires here is the following assembly instruction:

```
FFFFFFFC   b .
```

This is in effect a spin loop. This instruction encodes into 0x48000000. Be sure whoever writes the HW IP includes this instruction at this address which is typically a BRAM internal to the FPGA.

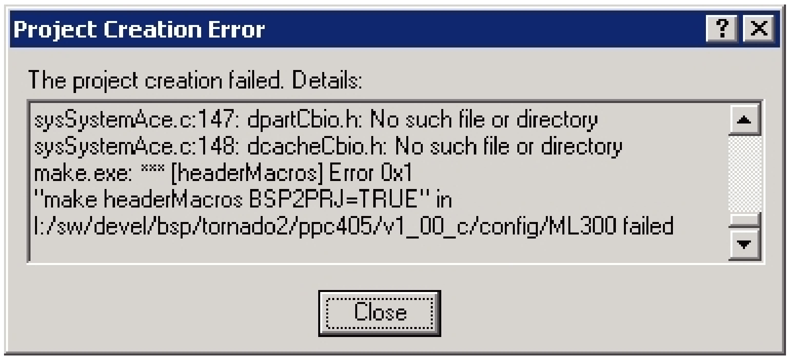
Trouble Shooting

Project Creation

Issues seen when creating a Tornado Project based on the MDFG456 BSP.

"Project Creation Error" Dialog Pop-up

Scroll to the end of the box and if it contains error messages complaining about missing header files `dpartCbio.h` and `dcacheCbio.h`, then you don't have the DosFS 2.0 libraries installed in your system.



```
sysSystemAce.c:147: dpartCbio.h: No such file or directory
sysSystemAce.c:148: dcacheCbio.h: No such file or directory
make.exe: *** [headerMacros] Error 0x1
"make headerMacros BSP2PRJ=TRUE" in
l:/sw/devel/bsp/tornado2/ppc405/v1_00_c/config/ML300 failed
```

SingleStep

Issues seen when using SingleStep with this BSP.

Source browser not displaying source code at addresses where source code should be

Try to rebuild everything with the -gdwarf compiler option.

Tornado Crosswind debugger

Issues seen when using Tornado's IDE debugger with this BSP.

Source browser not displaying source code at addresses where source code should be

Is the -gdwarf option enabled in the compiler? Try to rebuild everything with the -g compiler option.

Target Shell Issues

Issues seen when using the built-in target shell.

Relocation value does not fit in 24 bits message from Loader

This is seen when a system contains more than 32MB of memory. Recompile your source code using the -mlongcall compiler option.

Ethernet Issues

Issues seen when integrating/using the XEmac Ethernet adapter

Network interface xemac unknown. Message from console at boot

There are multiple causes to this problem.

1. Did you compile the XEmac component into the BSP? In the Tornado Project facility, check that both *hardware->peripherals->IP CSP->Ethernet EMAC* and *EMAC END* components are included. On command line BSP builds, is `INCLUDE_XEMAC` and `INCLUDE_XEMAC_END` declared in `ip_config.h` and not `#undef`'d anywhere.
2. In the Tornado Project facility, if you remove then later restore network support, Project fails to restore "END" driver support. Check folder *network components->network devices* and verify that both *END attach interface* and *END interface support* components are included.

BSP Release History

MDFG456 1.2/0 - January 10, 2003

First pre-release for Tornado 2.0. This is a beta release that does not include support for all HW. Note that testing has been done on the ML3 evaluation board as opposed to the MDFG456. In other words, this version of the BSP has never been run on the MDFG456 hardware. The SEG IP bitstream is used for this load.

HW Supported

- Main board UART (console). IP core is UartLite.
- Main board SDRAM 32MB
- Main board LEDs (1-4)
- Main board push button switches (1-3)
- Main board DIP switch (1-8)
- Main board LCD display
- P160 daughter board UART. IP core is a UartLite.
- P160 daughter board Flash/SRAM.
- P160 daughter board Ethernet. IP core is a Emac.

HW Not supported

- System ACE daughter board.
- P160 daughter board IIC header
- P160 daughter board SPI header
- P160 daughter board USB port.
- P160 daughter board PS/2 port.

Errata

1. System mode debugging through the END connection does not work.
2. Serial port usage as the WDB target connection does not work. Serial port polling mode does not seem to work.
3. Rev1 evaluation boards have mis-wired SDRAM that requires all accesses occur in 32 bit divisible quantities. This problem is worked around by enabling the data cache at bootstrap time and leaving it on. Instruction fetches always occur in 32 bit divisible quantities.
4. Since System ACE is not supported in this revision (it will be in future revisions), the only way to get a VxWorks image downloaded into the target is via an emulator.
5. Ethernet links have been iffy on two sample Rev1 boards this BSP has been tested on. If the link LED on the P160 board does not illuminate after applying power to the target (assuming there is a cable connected between the RJ45 jack and another network device), then try reseating the P160, emulator connector, and RS232 connectors. Once the link is established, it seems to remain that way.
6. Use non null-modem cabling for the RS-232 serial ports.
7. System not tested with MMU enabled.

References

- VxWorks 5.4 Programmer's Guide
- *Tornado 2.0 User's Guide*

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/10/03	1.0	First release.