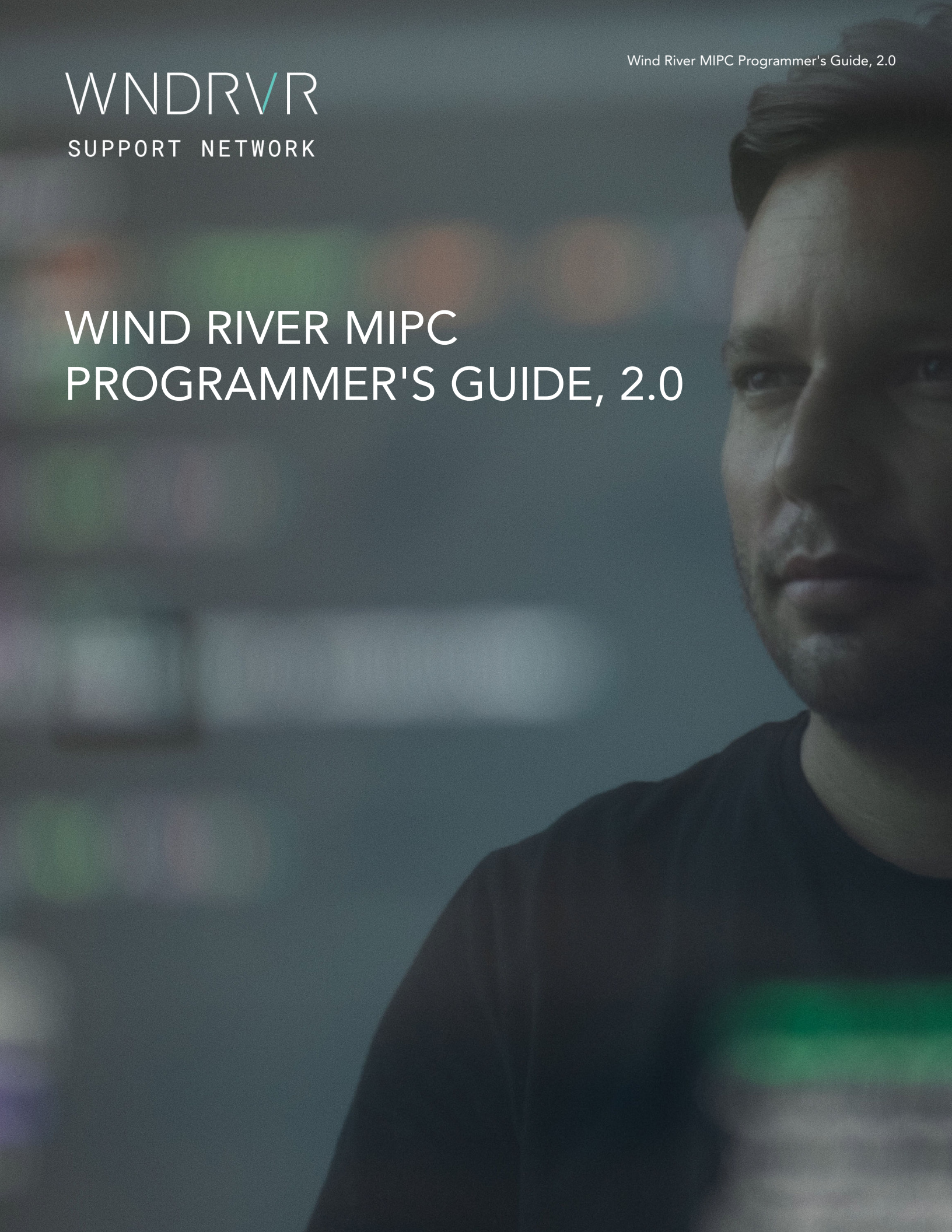
WNDRVR
SUPPORT NETWORK

WIND RIVER MIPC PROGRAMMER'S GUIDE, 2.0



Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Wind River MIPC Programmer's Guide, 2.0

4 February 2019

1. WIND RIVER MIPC PROGRAMMER'S GUIDE, 2.0

2. OVERVIEW

2.1. Introduction

This Guide describes the features and use of Wind River Multi-OS Interprocess Communication (MIPC) 2.0.

MIPC 2.0 provides socket-oriented communication between applications in the following contexts:

- A multicore environment in which VxWorks applications run on nodes configured for asymmetric multiprocessing (AMP) (see the VxWorks Kernel Programmer's Guide: Overview of VxWorks AMP).
- A Wind River Hypervisor environment in which applications run under guest operating systems, whether Linux or VxWorks. (For explanations of the terminology in this paragraph, see Hypervisor-Specific Terminology on page 3. For information on Wind River Hypervisor, see the Wind River Hypervisor User's Guide.)
- A multicore environment in which applications run on multiple CPUs under a single VxWorks operating system (SMP); For information on SMP, see the VxWorks Kernel Programmer's Guide: VxWorks SMP).
- A uniprocessor environment for communication between local processes.

MIPC 2.0 supports VxWorks and Linux kernel applications and Linux user-space applications. The current release does not support MIPC user-space (RTP) applications for VxWorks.

For Linux user-space applications, MIPC 2.0 supports the standard socket API defined by the Berkeley Software Distribution (BSD). The socket family for use with MIPC BSD sockets is **AF_MIPC** (see AF_MIPC API (Linux only) on page). In this document, the MIPC BSD socket API is referred to as the **AF_MIPC**socket API or, more simply, as the **AF_MIPC** API or **AF_MIPC**.

For kernel applications, MIPC provides a socket-like API in which names are prefixed with **mipc_**, as in **mipc_bind( )** (see mipc_ API for Kernel Applications on page). To distinguish it from the **AF_MIPC** API, this API is referred to as the **mipc_socket** API or, simply, the **mipc_** API. Special features of the **mipc_** API are:

- Zero-copy buffers

  The **mipc_** API provides routines for sending and receiving data in zero-copy buffers. This can result in higher performance, since it means that MIPC does not have to copy data into and out of buffers.

- Interrupt level handling of received events

  Events received at a socket, such as connection attempts and the arrival of packets, can be handled at interrupt level.

- Rapid transfer of short, 64-bit messages between sockets (see Express Data Transfer on page 48).

- Callback routines for handling events such as the following (the list is not exhaustive):
  - Nodes becoming available or unavailable on a bus (see MIPC_NODEJOIN_CALLBACK( ) on page 44 and MIPC_NODELEFT_CALLBACK( ) on page 45).
  - Non-blocking connection attempts and socket disconnections (see MIPC_CONNECTED_CALLBACK( ) on page 42 and MIPC_DISCONNECTED_CALLBACK( ) on page 43).
  - Arrival of zero-copy buffers at a socket. This allows you to immediately release a buffer, queue it for a MIPC receive routine, or hold it for further processing (see MIPC_RX_CALLBACK( ) on page 46).
  - Queuing of a buffer in the receive queue of a socket (see MIPC_RXQUEUED_CALLBACK( ) on page 45.

Both the **mipc_** and **AF_MIPC** APIs support connection-based and connectionless datagrams and connection-based byte streams.

2.2. Terminology

A number of terms used in this document may not be familiar, or may not be used in ways that are familiar to you. The following are some definitions and clarifications.

Nodes, CPUs, and Operating Systems

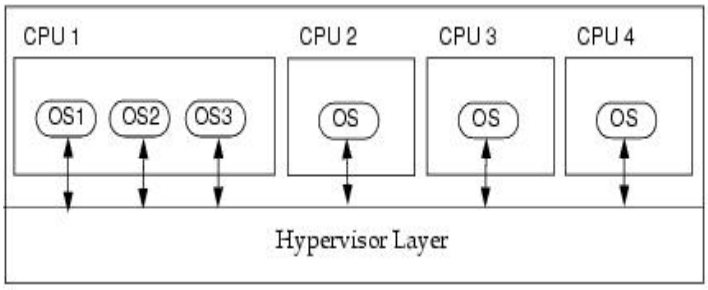
In this document, the term node applies to an instance of an operating system. In some cases there is a one-to-one correspondence between nodes and CPUs, but often there is no such correspondence. For example, under Wind River Hypervisor there may be two or more instances of an operating system--that is, two or more nodes--on a single physical CPU; in a multicore SMP configuration using three CPUs, the three CPUs together correspond to a single node.

Hypervisor-Specific Terminology

A hypervisor is a software layer that makes it possible for two or more operating systems to share the same CPU. The hypervisor provides scheduling and access to hardware for the operating systems. An operating system that uses the hypervisor layer is called a guest operating system. The term virtual board is used to refer to a guest operating system and the physical resources (memory, cores, devices) allocated to it.

Wind River Hypervisor--or, Hypervisor, with a capital H--is the Wind River implementation of a hypervisor. In a multicore environment, there can be multiple guest operating systems on any given CPU. Wind River Hypervisor can also regulate access to resources when there are CPUs with only a single guest operating system, as in Figure 1-1 on page 4.

Figure 1-1 : Multicore Hypervisor Configuration



In Figure 1-1 on page 4, CPU 1 has three guest operating-systems, and the remaining three CPUs each have one guest operating system. Wind River Hypervisor allows you to configure MIPC communication between operating systems so that, for example, all six instances of an operating system can communicate with each other, or so that OS1, OS2, and OS3 in CPU 1 can communicate with each other, but not with CPUs 2 through 4. For more information on Wind River Hypervisor, see the Wind River Hypervisor User's Guide.

2.3. Architectural Overview of MIPC Communication

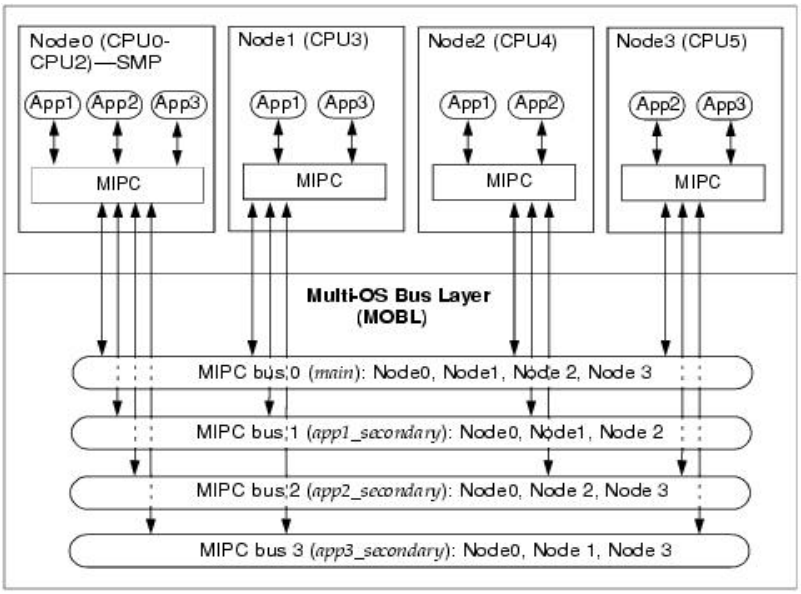
This section describes MIPC communication in the following environments:

- MIPC with VxWorks AMP and SMP in a multicore environment (see the next section).
- MIPC with two or more Hypervisor guest operating systems (see MIPC With Hypervisor on page 5).

2.3.1. MIPC with VxWorks AMP and SMP

Figure 1-2 on page 5 illustrates the basic framework for MIPC 2.0 communication between instances of the VxWorks operating system in a multicore environment.

Figure 1-2 : MIPC 2.0 Communication in a VxWorks Multicore Environment



In Figure 1-2 on page 5, applications run under VxWorks with communication between them carried out through MIPC. In the example, Node0 represents three CPUs used for symmetric multiprocessing (SMP), with VxWorks on CPU0 (this is a requirement of VxWorks 6.x). All four nodes are part of a VxWorks AMP configuration.

MIPC uses the Multi-OS Bus Layer (MOBL) to allocate shared memory and create communication links between nodes. Within the MOBL, shared memory can be partitioned into multiple virtual buses--MIPC buses. MIPC allows you to define up to 1024 such buses, with each bus given its own name and ID number.

In Figure 1-2 on page 5, there are four MIPC buses. The first MIPC bus has been assigned the name main and ID number **0**. All four nodes use this bus.

The second MIPC bus, app1_secondary, is dedicated to **App1**, which runs on **Node0**, **Node1**, and **Node2**. Similarly, the third and fourth MIPC buses are dedicated to **app2** and **app3**, respectively.

As the example indicates, a given node can be attached to more than one MIPC bus and multiple applications can use the same bus (bus 0, in Figure 1-2 on page 5).

2.3.2. MIPC With Hypervisor

Figure 1-3 on page 6 illustrates the basic framework for MIPC 2.0 communication between guest operating systems running under Wind River Hypervisor.

Figure 1-3 : MIPC 2.0 Communication Framework with Hypervisor

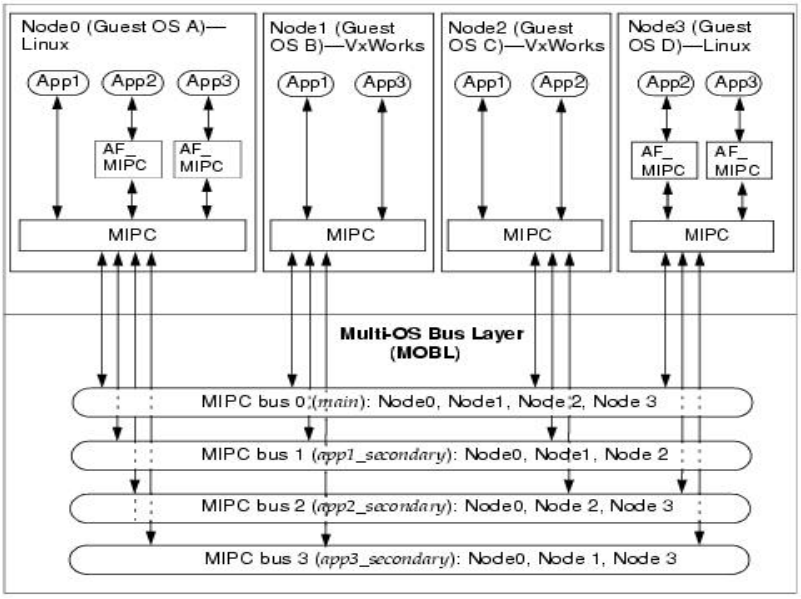


Figure 1-3 on page 6 shows four nodes with four guest operating systems, representing four virtual boards. The four nodes and their guest operating systems can be on four separate CPUs, on a single CPU, or configured on two or three CPUs.

On Node0, there are three applications. App1 is a Linux kernel application that uses the **mipc_** API. App2 and App3 are both user-space applications that use the **AF_MIPC** API. Node1 and Node2 both have VxWorks guest operating systems that uses the **mipc_** API (the current release does not support use of the **AF_MIPC** API for VxWorks). Node3 has Linux **AF_MIPC** versions of App2 and App3.

The applications running on the four nodes communicate with each other using MIPC, which uses shared memory provided through the Multi-OS Bus Layer (MOBL). Within the MOBL, shared memory can be partitioned into multiple virtual buses--MIPC buses. MIPC allows you to define up to 1024 such buses, with each bus given its own name and ID number.

In Figure 1-3 on page 6, as in Figure 1-2 on page 5, there are four MIPC buses. The first MIPC bus has been assigned the name main and ID number **0**. All four nodes use this bus.

The second MIPC bus, app1_secondary, is dedicated to **App1**, which runs on **Node0**, **Node1**, and **Node2**. Similarly, the third and fourth MIPC buses are dedicated to **app2** and **app3**, respectively.

As the example indicates, a given node can be attached to more than one bus and multiple applications can use the same bus (bus 0, in Figure 1-3 on page 6).

2.4. Restrictions on the Use of MIPC

The current version of MIPC is subject to the following restriction:

- MIPC 2.0 is dependent on the use of shared memory and cannot be used for communication between networked processors.

2.5. Debugging a MIPC Target

Targets that use MIPC for communication between nodes typically do not have a full network stack and cannot communicate directly with Workbench debugging tools on a host. When this is the case, communication between target and host needs to go through a target system set up as a gateway between the target and the host. For information on setting up such a gateway and debugging over MIPC, see either the VxWorks Kernel Programmer's Guide: Configuring VxWorks for AMP or the Wind River Hypervisor User's Guide: Understanding the Debug Shell.

2.6. Organization of This Document

The remaining chapters in this book are organized as follows:

- VxWorks: Building MIPC on page covers VxWorks build components and configuration parameters.
- Linux: Building MIPC on page covers Linux build configuration for MIPC, Linux kernel configuration (Kconfig) components and parameters for MIPC, and loadable kernel modules and parameters for MIPC.
- mipc_ API for Kernel Applications on page describes the **mipc**_ API, which includes socket-like routines, routines for the use of zero-copy buffers, callback routines, and other features. In addition the chapter provides a sample application illustrating the use of the **mipc**_ API.
- AF_MIPC API (Linux only) on page describes the **AF_MIPC** socket API and its extensions. The API is restricted to Linux user-space applications.
- MIPC Demo on page describes how to run the MIPC demo application.
- MIPC Show Routines (VxWorks) on page provides information about the VxWorks show routines available with MIPC and gives sample output.

3. VXWORKS: BUILDING MIPC

3.1. Introduction

This chapter covers MIPC 2.0 build components and configuration parameters for VxWorks.

The basic steps in building and configuring MIPC are:

1. Create a VxWorks Image Project that includes the **MIPC over SM (INCLUDE_MIPC_SM**) build component (see VxWorks Build Components for MIPC on page 8, and the entry for **MIPC over SM** inTable 2-1 on page 8).

   This is the only required MIPC component. There are additional components for including MIPC show routines and a demo application (see Table 2-1 on page 8).

2. Set MIPC configuration parameters (see MIPC Configuration Parameters for VxWorks on page 9, and Table 2-2 on page 9).

3. After including and configuring all other VxWorks components, build your project.

3.2. VxWorks Build Components for MIPC

Table 2-1 on page 8 table lists the build components for VxWorks MIPC. To make it easier to use the table as a reference, components are listed alphabetically, based on the **Description** field in the Workbench Kernel Configuration Editor.010#f441fa34-6ffe-4cb1-9809-e554895ce7e7__wp161094 on page 9 Only one of the components, **MIPC over SM (INCLUDE_MIPC_SM**), contains configuration parameters MIPC Configuration Parameters for VxWorks on page 9.

Table 2-1 : Build Components Used in Configuring VxWorks for MIPC

Component (Workbench Description and Macro Name)	Description
This BSP does not have MIPC Multi-OS support [INCLUDE_MIPC_UNSUPPORTED]	This component appears in Workbench only if you have created a project based on a BSP that does not support MIPC.
MIPC demo [INCLUDE_MIPC_DEMO]	Includes the MIPC demo application (see 6. MIPC Demo on page).

MIPC over SM [**[INCLUDE_MIPC_SM]**	A required component that gives MIPC access to shared memory. This component contains a number of parameters (see 2.3 MIPC Configuration Parameters for VxWorks on page 9).
MIPC Show routines **[INCLUDE_MIPC_SHOW]**	Includes the following command-line show routines in the build: **mipcHelp( )** (see The mipcHelp Command on page 84) **mipcShow( )** (see The mipcShow Command on page 84) **mipcShowBus( )** (see The mipcShowBus Command on page 84) **mipcShowNode( )** (see The mipcShowNode Command on page 85) **mipcShowPort( )** (see The mipcShowPort Command on page 85)

[1] on page 8In the Workbench Kernel Configuration Editor, the components under a folder are listed alphabetically by **Description**. Even if you use **vxprj** for command-line configuration, viewing the Kernel Configuration Editor may make it easier to see the components and parameters required.

3.3. MIPC Configuration Parameters for VxWorks

All the configuration parameters for MIPC are contained in the **MIPC over SM** (**INCLUDE_MIPC_SM**) build component (see Table 2-1 on page 8). The parameters are listed in Table 2-2 on page 9. As in the case of build components, parameters are listed alphabetically, based on the **Description** field in the Workbench Kernel Configuration Editor.

Table 2-2 : MIPC over SM (INCLUDE_MIPC_SM) Parameters

Parameter(Workbench Description and Macro Name)	Default Value and Data Type	Description
Maximum buffers per bus **[MIPC_SM_BUFFERS]**	**64** UINT	The number of buffers allocated to this node on each bus used by the node. For example, the

		default value of 64 means that when the node attaches to a bus, the system allocates 64 buffers from shared memory to the node. You can override the value of this parameter for individual buses through the **buffers** parameter of the bus configuration string (see the table entry for **List of available MIPC buses**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
List of available MIPC buses [MIPC_SM_BUS_LIST]	**"#main=0"** **STRING**	A string that lists MIPC buses and allows you to set bus-specific parameter values that override the settings of parameters such as **Maximum buffers per bus** that assign node-wide default values to all buses that the node uses. • For nodes that cannot initialize shared memory (see the table entry for **Shared memory initialization mode**), you need to list all buses that the current node can use. • For nodes that can initialize shared memory, you need to list all buses available in the system, including buses that the current node is never going to use. This list defines the buses available to any node when the current node initializes shared memory (see Initializing Shared Memory on page 17). For additional information, the syntax of the string, and

		examples, see 2.3.1 MIPC Bus Configuration String on page 15). This parameter affects the allocation of shared memory. If you are making changes to the bus configuration string in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Event pool size per bus [MIPC_SM_EVENTS]	32 **INT**	Determines the number of express messages (see 4.3 Express Data Transfer on page 48) and connection handshaking events that can be handled on a bus on this node. In general, set this parameter to the maximum number of ports expected on a bus plus the maximum number of expected express messages on a bus on this node. You can override the value of this parameter for individual buses through the **events** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Heartbeat rate [MIPC_SM_HEARTBEAT_PERIOD]	50 **UINT**	The rate, in milliseconds, at which the node updates its heartbeat count in shared memory, letting other nodes know that it is active on a bus. It also indirectly determines the rate at which the node polls shared memory for the hearbeats

		of other nodes in order to find out which nodes are currently active on a bus. All nodes must be set to the same heartbeat rate. This parameter affects the way shared memory is initialized. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Shared memory initialization mode [MIPC_SM_INIT_MODE]	1 **UINT**	Determines whether the current node initializes shared memory at start up. There are three options: • 0 (**WAIT_FOR_INIT**)--Do not initialize. • 1 (**CAN_INIT**)-- Initialize, if shared memory has not already been initialized. • 2 (**ALWAYS_INIT**)--Always initialize at start up. For more information about the individual options, see 2.3.2 Initializing Shared Memory on page 17.
Maximum buffer size per bus [MIPC_SM_MTU]	1520 **UINT**	Specifies the maximum size, in bytes, of the buffers this node can use for transmission on a bus. You can override the value of this parameter for individual buses through the **mtu** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node,

		make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Maximum nodes per bus [MIPC_SM_NODES]	2 **UINT**	Specifies the maximum number of nodes a bus can support. You can override the value of this parameter for individual buses through the **nodes** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Interrupt request line [MIPC_SM_NODE_IRQ]	[Default depends on BSP] **INT**	The interrupt request line this node uses to detect incoming events on MIPC buses. Do not change the default value. The default value instructs MIPC to use the interrupt line specified by the BSP for the current project.
Maximum ports per bus [MIPC_SM_PORTS]	32 **UINT**	The maximum number of ports this node can have on a bus. You can override the value of this parameter for individual buses through the **ports** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current

		parameter value in order to rebuild the image for a node, make sure you are familiar with 2.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 18.
Quality of service used by buses [MIPC_SM_QOS]	41 **INT**	The processing mode a bus uses to handle incoming events, such as the arrival of incoming buffers. There are three modes. The following gives a brief description of each node. For more detailed information, see 2.3.4 QoS Processing Modes for Incoming Events on page 18. To specify a QoS processing mode, enter an integer, as follows: • N > 0 (ISR deferred mode) The interrupt handler schedules responses (such as MIPC callbacks--see mipc_ API Callback Routines on page 41) to events at task level N, rather than handling the events in an interrupt service routine (ISR). • 0 (ISR mode) ISRs respond to events and invoke callback routines at interrupt level. • -1 (user mode) For use only with kernel space applications. MIPC detects incoming events only when instructed to by a client application. You can override the value of this parameter for individual buses through the **qos** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**).

Receive task stack size [MIPC_SM_RXTASK_STACK]	4000 **UINT**	For buses operating in **ISR deferred** mode (see the table entry for **MIPC_SM_QOS**), the size, in bytes, of the stack available to tasks processing incoming events. If an application creates sockets that have callback routines that use a lot of stack space, it may be necessary to increase the parameter value above its default value.
Maximum number of sockets [MIPC_SM_SOCKETS]	-1 **INT**	The maximum number of MIPC sockets that can simultaneously exist on this node. The default value instructs MIPC to reserve enough memory to support a socket on every available port on each of the node's buses. For example, if there are three buses, each of which can have 32 ports, MIPC will reserve enough memory for 96 sockets.
Shared memory base address [MIPC_SM_SYSTEM_POOL_BASE]	[Default depends on BSP] **INT**	The base address of the shared-memory region for MIPC. Do not change the default value. The default value instructs MIPC to use the shared-memory base address specified by the BSP for the current project.
Shared memory size [MIPC_SM_SYSTEM_POOL_SIZE]	[Default depends on BSP] **INT**	The size of the shared memory region used by MIPC. Do not change the default value. The default value instructs MIPC to use the shared-memory base address specified by the BSP for the current project.

3.3.1. MIPC Bus Configuration String

When you configure a MIPC node, the **List of available MIPC buses** (**MIPC_SM_BUS_LIST**) parameter requires you to specify a bus configuration string that lists MIPC buses and allows you to set bus-specific

parameter values. The bus-specific values override the settings of parameters such as **Maximum ports per bus** (**MIPC_SM_PORTS**) that establish default values for all buses used by a node.

- For nodes that cannot initialize shared memory (see the entry for **Shared memory initialization mode** in Table 2-2 on page 9), you need to list all buses that the current node can use.
- For nodes that can initialize shared memory, you need to list all buses available in the system, including buses that the current node is never going to use. This list defines the buses available to a node when the current node initializes shared memory (see Initializing Shared Memory on page 17).
- The bus configuration string for a node can only specify bus-specific parameter values for its own node.

Syntax of the Bus Configuration String

The syntax of the bus configuration string is:

#busname**=**busid**[,**param**=**value**,**param**=**value**,...]#**busname**=**busid**[,**param**=**value**,**param**=**value**,...]**...

where

- busname is the name of the bus.
- busid is an integer ID for the bus.

  Bus names and IDs must be consistent across nodes.
- param is one of the bus-specific parameters in Table 2-3 on page 15. Table 2-3 on page 15 lists bus-specific parameters and gives cross-references to the corresponding node-wide parameters that they override.)
- value is a bus-specific parameter value.

Table 2-3 : Bus-Specific Parameters

Bus-Specific Parameter	Node-Wide Bus Parameter
activeonbus	[No corresponding node-wide bus parameter.] Determines whether a node can use (be active on) a bus: **1**: The node can use the bus. **0**: The node cannot use on the bus. By default, activeonbus is **1**. Typically, you only set **activeonbus** to **0** when a node that always initializes shared memory does not itself use a given bus (see 2.3.2 Initializing Shared Memory on page 17).

buffers	**Maximum buffers per bus** **[MIPC_SM_BUFFERS]** on page 9
events	**Event pool size per bus** **[MIPC_SM_EVENTS]** on page 11
mtu	**Maximum buffer size per bus** **[MIPC_SM_MTU]** on page 12
nodes	**Maximum nodes per bus** **[MIPC_SM_NODES]** on page 13
ports	**Maximum ports per bus** **[MIPC_SM_PORTS]** on page 13
qos	**Quality of service used by buses** **[MIPC_SM_QOS]** on page 14

Configuration String Examples

The following is an example of a configuration string that lists buses, but does not set any bus-specific parameters:

#main=0#app1_secondary=1#app2_secondary=2#app3_secondary=3

The following example sets bus-specific parameters for four buses:

#main=0,buffers=32,events=48,ports=64#app1_secondary=1#app2_secondary=2#app3_secondary=3,buffers=24

In the example, bus **main** is allocated 32 buffers and bus **app3_secondary** is allocated 24 buffers on the current node. The remaining two buses are allocated the number of buffers set in the **Maximum buffers per bus** (**MIPC_SM_BUFFERS**) parameter. In addition, bus **main** is configured for 48 events and 64 ports. All other buses are configured for whatever is set in the **Event pool size per bus** (**MIPC_SM_EVENTS**) and **Maximum ports per bus** (**MIPC_SM_PORTS**) parameters.

3.3.2. Initializing Shared Memory

Before any MIPC node can start communicating with other nodes, shared memory needs to be initialized. Initialization is performed by a single node. The **Shared memory initialization mode** (**MIPC_SM_INIT_MODE**) parameter, allows you to designate a specific node for shared-memory initialization or to specify multiple nodes any of which can initialize shared memory if it is started first. The parameter provides the following initialization options:

- 0 (**WAIT_FOR_INIT**)--The node never initializes shared memory
- 1 (**CAN_INIT**)-- The node initializes shared memory if shared memory hasn't already been initialized when it starts.

  This allows a node that can initialize shared memory to reboot without requiring all other nodes to reboot with it (see the next option, **ALWAYS_INIT**).
- 2 (**ALWAYS_INIT**)--The node always initializes shared memory.

  Use this option for a node only if:

  - This is always the first node up that is capable of initializing shared memory (all other nodes should be designated **WAIT_FOR_INIT**)
  - It is acceptable that whenever this node reboots, it forces all other nodes to to reboot.

3.3.3. Changing Parameters that Affect Shared Memory When Other Nodes are Running

A number of MIPC parameters affect shared memory (see Parameters that affect shared memory on page 18). If you find that you need to change the value of one or more of these parameters for a node that is currently active on a bus:

- Stop the node, make the changes, and re-image the node.
- Before restarting the node, stop all other nodes and then restart nodes according to your standard startup procedure.

If you re-image a node after making changes that affect shared memory and start it while other nodes are still running, it forces the nodes to reboot. This can have unwanted consequences.

Parameters that affect shared memory

The following parameters affect shared memory:

- **List of available MIPC buses** (**MIPC_SM_BUS_LIST**)

  The number of buses that you define in the bus configuration string affects the allocation of shared memory. In addition, the following parameters within the configuration string have an effect on shared-memory allocation:

  - **buffers**
  - **events**
  - **mtu**
  - **nodes**
  - **ports**
- **Maximum buffers per bus** (**MIPC_SM_BUFFERS**)
- **Maximum buffer size per bus** (**MIPC_SM_MTU**)
- **Maximum nodes per bus** (**MIPC_SM_NODES**)
- **Maximum ports per bus** (**MIPC_SM_PORTS**)
- **Event pool size per bus** (**MIPC_SM_EVENTS**)
- **Heartbeat rate** (**MIPC_SM_HEARTBEAT_PERIOD**)
- **Shared memory base address** (**MIPC_SM_SYSTEM_POOL_BASE**)
- **Shared memory size** (**MIPC_SM_SYSTEM_POOL_SIZE**)

3.3.4. QoS Processing Modes for Incoming Events

The **Quality of service used by buses** (**MIPC_SM_QOS**) parameter requires you to specify a Quality of Service (QoS) processing mode for incoming events such as connection attempts and incoming buffers. These events trigger internal callbacks or, if they have been implemented, MIPC callbacks routines (see mipc_ API Callback Routines on page 41). There are three processing modes. To specify a mode, enter an integer, as follows:

- N > 0 (ISR deferred mode)

  The interrupt handler schedules responses (such as MIPC callbacks--see mipc_ API Callback Routines on page 41) to events at task level rather than handling the events in an interrupt service routine (ISR). The worker task that handles an event is created with a VxWorks task priority level of N and a stack size of **MIPC_SM_NODE_RXTASK_STACK** bytes (see Table 2-2 on page 9**)**.

- 0 (ISR mode)

  ISRs respond to events and invoke callback routines at interrupt level. Do not choose this option if any of the callbacks that might be called contain blocking operations.

  This processing mode is not recommended for sustained levels of high throughput.

- -1 (user mode)

  For use only with kernel space applications. MIPC detects incoming events only when instructed to by a client application. This mode allows an application to poll for bus activity, rather than using interrupts.

4. LINUX: BUILDING MIPC

4.1. Introduction

This chapter covers including MIPC in a Linux build and loading MIPC kernel modules and their configuration parameters.

To include MIPC in a Linux build you need to:

1. Configure your build for MIPC (see Including MIPC in a Linux Build Configuration on page 20).
2. Configure the MIPC kernel configuration (Kconfig) component (see Configuring the MIPC Kconfig Component and Its Parameters on page 21).
3. Execute **make**.

At run time:

4. Load the MIPC kernel modules and set kernel module configuration parameters (see Loading the MIPC Kernel Modules on page 30).000#3f2fa1c1-5306-4e33-b26c-7189555ee44c__wp158862 on page 20

[1] on page 20If you are going to include the MIPC Serial Device (MSD) feature in your project, you need to make entries for MSD devices in your target's file system before you load MIPC (see VxWorks Kernel Programmer's Guide: MIPC Serial Device (MSD)).

4.2. Including MIPC in a Linux Build Configuration

You can build Linux to include MIPC from Workbench or from the command line. In either case, you need to include the **wrll-multicore** layer and the **feature/mipc** template in your build, as follows:

Workbench:

1. Create a new Linux Platform Project and enter a name for the project, then click **Next**.

   This brings up the Configuration options window.

2. In the Configuration options window, enter required settings and then click **Advanced**.

   This displays the **Layers** box and the **Templates and Profiles** box.

3. Click **Add**, next to the **Layers** box.

   This allows you to browse to the location of the **wrll-multicore** layer:

   > layers > wrll-multicore

4. Select **wrll-multicore** and click **OK**.

   The **Layers** box now displays the **wrll-multicore** layer.

5. Click **reload**.

6. Click **Template...** next to the **Templates and Profiles** box.

   This displays the **Add Templates** dialog.

7. Scroll down the template list, check **feature/mipc** and click **OK**.

   This returns you to the Configuration options window, with **fea**ture/mipc displayed in the **Templates and Profiles** box.

8. Click **Finish**.

Command Line

1. Configure your basic settings for kernel, file system, and tool set; set the **--with-layer** option to **wrll-multicore**, and add the **feature/mipc** template as in the following example:

   installDir/wrlinux-3.0/wrlinux/configure --enable-rootfs= glibc_small+debug --enable-board=fsl_8572ds --enable-kernel=standa

   Note that you need to set the **--enable-board** argument to match the BSP for your project.

4.3. Configuring the MIPC Kconfig Component and Its Parameters

There is one kernel configuration (Kconfig) component for MIPC, which you can configure either through Workbench or from the command line with the **linux.menuconfig** utility: **Multi-OS Inter-Processor Communication** (**MULTIOS_IPC**). The component is included in a MIPC project by default. The path to the component is:

Linux Kernel Configuration > Device Drivers > Multi-OS Inter-Processor Communication

Table 3-1 on page 21 lists the component's parameters.

Table 3-1 : MIPC Kconfig Parameters

Parameter(Workbench Description and Macro Name)	Default Value and Data Type	Description

Maximum buffers per bus [MIPC_SM_BUFFERS]	64 UINT	The number of buffers allocated to this node on each bus used by the node. For example, the default value of 64 means that when the node attaches to a bus, the system allocates 64 buffers from shared memory to the node. You can override the value of this parameter for individual buses through the **buffers** parameter of the bus configuration string (see the table entry for **List of available MIPC buses**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
List of available MIPC buses [MIPC_SM_BUS_LIST]	**"#main=0"** **STRING**	A string that lists MIPC buses and allows you to set bus-specific parameter values that override the settings of parameters such as **Maximum buffers per bus** that assign node-wide default values to all buses that the node uses. • For nodes that cannot initialize shared memory (see the table entry for **Shared memory initialization mode**), you need to list all buses that the current node can use. • For nodes that can initialize shared memory, you need to list all buses available in the system, including buses that the current node is never going to use. This list defines the buses available to any node when the current node initializes shared memory (see Initializing Shared Memory on page 28).

		For additional information, the syntax of the string, and examples, see 3.3.1 MIPC Bus Configuration String on page 27). This parameter affects the allocation of shared memory. If you are making changes to the bus configuration string in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
Event pool size per bus [MIPC_SM_EVENTS]	32 **INT**	Determines the number of express messages (see 4.3 Express Data Transfer on page 48) and connection handshaking events that can be handled on a bus on this node. In general, set this parameter to the maximum number of ports expected on a bus plus the maximum number of expected express messages sent on a bus by this node. You can override the value of this parameter for individual buses through the **events** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
Heartbeat ratex [MIPC_SM_HEARTBEAT_PERIOD]	50 **UINT**	The rate, in milliseconds, at which the node updates its heartbeat count in shared memory, letting other nodes know that it is active on a bus. It

		also indirectly determines the rate at which the node polls shared memory for the hearbeats of other nodes in order to find out which nodes are currently active on a bus. All nodes must be set to the same heartbeat rate. This parameter affects the way shared memory is initialized. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
Shared memory initialization mode [MIPC_SM_INIT_MODE]	1 **UINT**	Determines whether the current node initializes shared memory at start up. There are three options: • 0 (**WAIT_FOR_INIT**)--Do not initialize. • 1 (**CAN_INIT**)-- Initialize, if shared memory has not already been initialized. • 2 (**ALWAYS_INIT**)--Always initialize at start up. For more information about the individual options, see 3.3.2 Initializing Shared Memory on page 28.
Maximum buffer size per bus [MIPC_SM_MTU]	1520 **UINT**	Specifies the maximum size, in bytes, of the buffers this node can use for transmission on a bus. You can override the value of this parameter for individual buses through the **mtu** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST)**. This parameter affects the allocation of shared memory. If

		you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
Maximum nodes per bus [MIPC_SM_NODES]	2 **UINT**	Specifies the maximum number of nodes a bus can support. You can override the value of this parameter for individual buses through the **nodes** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.
Maximum ports per bus [MIPC_SM_PORTS]	32 **UINT**	The maximum number of ports this node can have on a bus. You can override the value of this parameter for individual buses through the **ports** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**). This parameter affects the allocation of shared memory. If you are changing the current parameter value in order to rebuild the image for a node, make sure you are familiar with 3.3.3 Changing Parameters that Affect Shared Memory When Other Nodes are Running on page 29.

Quality of service used by buses [MIPC_SM_QOS]	1 **INT**	The processing mode a bus uses to handle incoming events, such as the arrival of incoming buffers. There are three modes. The following gives a brief description of each node. For more detailed information, see 3.3.4 QOS Processing Modes for Incoming Events on page 30. To specify a QoS processing mode, enter an integer, as follows: • 1 (ISR deferred mode) The interrupt handler schedules responses (such as MIPC callbacks--see mipc_ API Callback Routines on page 41) to events using a Linux kernel work queue, rather than handling the events in an interrupt service routine (ISR). • 0 (ISR mode) ISRs respond to events and invoke callback routines at interrupt level. • -1 (user mode) For use only with kernel space applications. MIPC detects incoming events only when instructed to by a client application. You can override the value of this parameter for individual buses through the **qos** parameter of the bus configuration string (see the table entry for **List of available MIPC buses (MIPC_SM_BUS_LIST**).
Maximum number of sockets [MIPC_SM_SOCKETS]	-1 **INT**	The maximum number of MIPC sockets that can simultaneously exist on this node. The default value instructs MIPC to reserve enough memory to support a socket on every available port on each of the

		node's buses. For example, if there are three buses, each of which can have 32 ports, MIPC will reserve enough memory for 96 sockets.

4.3.1. MIPC Bus Configuration String

When you configure a MIPC node, the **List of available MIPC buses** (**MIPC_SM_BUS_LIST**) parameter requires you to specify a bus configuration string that lists MIPC buses and allows you to set bus-specific parameter values. The bus-specific values override the settings of parameters such as **Maximum ports per bus** (**MIPC_SM_PORTS**) that establish default values for all buses used by a node.

- For nodes that cannot initialize shared memory (see the entry for **Shared memory initialization mode** in Table 3-1 on page 21), you need to list all buses that the current node can use.
- For nodes that can initialize shared memory, you need to list all buses available in the system, including buses that the current node is never going to use. This list defines the buses available to a node when the current node initializes shared memory (see Initializing Shared Memory on page 28).
- The bus configuration string for a node can only specify bus-specific parameter values for its own node.

Syntax of the MIPC Bus Configuration String

The syntax of the MIPC bus configuration string is:

#busname**=**busid**[,**param**=**value**,**param**=**value**,...]#**busname**=**busid**[,**param**=**value**,**param**=**value**,...]**...

where

- busname is the name of the bus.
- busid is an integer ID for the bus.

  Bus names and IDs must be consistent across nodes.
- param is one of the bus-specific parameters in Table 3-2 on page 27. Table 3-2 on page 27 lists bus-specific parameters and gives cross-references to the corresponding node-wide parameters that they override.

Table 3-2 : Bus-Specific Parameters

Bus-Specific Parameter	Node-Wide Bus Parameter
activeonbus	[No corresponding node-wide bus parameter.] Determines whether a node can use (be active on) a bus:

	1: The node can use the bus. **0**: The node cannot use on the bus. By default, activeonbus is **1**. Typically, you only set **activeonbus** to **0** when a node that always initializes shared memory does not itself use a given bus (see 3.3.2 Initializing Shared Memory on page 28).
buffers	**Maximum buffers per bus** **[MIPC_SM_BUFFERS]** on page 21
events	**Event pool size per bus** **[MIPC_SM_EVENTS]** on page 23
mtu	**Maximum buffer size per bus** **[MIPC_SM_MTU]** on page 24
nodes	**Maximum nodes per bus** **[MIPC_SM_NODES]** on page 25
ports	**Maximum ports per bus** **[MIPC_SM_PORTS]** on page 25
qos	**Quality of service used by buses** **[MIPC_SM_QOS]** on page 25

- value is a bus-specific parameter value.

Examples

The following is an example of a configuration string that lists buses, but does not set any bus-specific parameters:

#main=0#app1_secondary=1#app2_secondary=2#app3_secondary=3

The following example sets bus-specific parameters for four buses:

#main=0,buffers=32,events=48,ports=64#app1_secondary=1#app2_secondary=2#app3_secondary=3,buffers=24

Based on the example, bus **main** is allocated 32 buffers and bus **app3_secondary** is allocated 24 buffers on the current node. The remaining two buses are allocated the number of buffers set in the **Maximum buffers per bus** (**MIPC_SM_BUFFERS**) parameter. In addition, bus **main** is configured for 48 events and 64 ports. All other buses are configured for whatever is set in the **Event pool size per bus** (**MIPC_SM_EVENTS**) and **Maximum ports per bus** (**MIPC_SM_PORTS**) parameters.

4.3.2. Initializing Shared Memory

Before any MIPC node can start communicating with other nodes, shared memory needs to be initialized. Initialization is performed by a single node. The **Shared memory initialization mode**

(**MIPC_SM_INIT_MODE**) parameter, allows you to designate a specific node for shared-memory initialization or to specify multiple nodes any of which can initialize shared memory if it is started first. The parameter provides the following initialization options:

- 0 (**WAIT_FOR_INIT**)--The node never initializes shared memory
- 1 (**CAN_INIT**)-- The node initializes shared memory if shared memory hasn't already been initialized when it starts.

  This allows a node that can initialize shared memory to reboot without requiring all other nodes to reboot with it (see the next option, **ALWAYS_INIT**).
- 2 (**ALWAYS_INIT**)--The node always initializes shared memory.

  Use this option for a node only if:

  – This is always the first node up that is capable of initializing shared memory (all other nodes should be designated **WAIT_FOR_INIT**)

  – It is acceptable that whenever this node reboots, it forces all other nodes to to reboot.

4.3.3. Changing Parameters that Affect Shared Memory When Other Nodes are Running

A number of MIPC parameters affect shared memory (see Parameters that affect shared memory on page 29). If you find that you need to change the value of one or more of these parameters for a node that is currently active on a bus:

- Stop the node, make the changes, and re-image the node.
- Before restarting the node, stop all other nodes and then restart nodes according to your standard startup procedure.

If you re-image a node after making changes that affect shared memory and start it while other nodes are still running, it forces the nodes to reboot. This can have unwanted consequences.

Parameters that affect shared memory

The following parameters affect shared memory:

- **List of available MIPC buses** (**MIPC_SM_BUS_LIST**)

  The number of buses that you define in the bus configuration string affects the allocation of shared memory. In addition, the following parameters within the configuration string have an effect on shared-memory allocation:

  – **buffers**

  – **events**

  – **mtu**

  – **nodes**

  – **ports**
- **Maximum buffers per bus** (**MIPC_SM_BUFFERS**)

- **Maximum buffer size per bus** (**MIPC_SM_MTU**)
- **Maximum nodes per bus** (**MIPC_SM_NODES**)
- **Maximum ports per bus** (**MIPC_SM_PORTS**)
- **Event pool size per bus** (**MIPC_SM_EVENTS**)
- **Heartbeat rate** (**MIPC_SM_HEARTBEAT_PERIOD**)
- **Shared memory base address** (**MIPC_SM_SYSTEM_POOL_BASE**)
- **Shared memory size** (**MIPC_SM_SYSTEM_POOL_SIZE**)

4.3.4. QOS Processing Modes for Incoming Events

The **Quality of service used by buses** (**MIPC_SM_QOS**) parameter requires you to specify a Quality of Service (QoS) processing mode for incoming events such as connection attempts and incoming buffers. These events trigger internal callbacks or, if they have been implemented, MIPC callbacks routines (see mipc_ API Callback Routines on page 41). There are three processing modes. To specify a mode, enter an integer, as follows:

- 1 (ISR deferred mode)

  The interrupt handler schedules responses (such as MIPC callbacks--see mipc_ API Callback Routines on page 41) to events at task level rather than handling the events in an interrupt service routine (ISR).

- 0 (ISR mode)

  ISRs respond to events and invoke callback routines at interrupt level. Do not choose this option if any of the callbacks that might be called contain blocking operations.

  This processing mode is not recommended for sustained levels of high throughput.

- -1 (user mode)

  For use only with kernel space applications. MIPC detects incoming events only when instructed to by a client application. This mode allows an application to poll for bus activity, rather than using interrupts.

4.4. Loading the MIPC Kernel Modules

When you build MIPC, it contains two loadable kernel modules (LKMs):

- **multios_ipc_lkm-prop.ko**

  This module contains proprietary MIPC code and must be loaded first. It contains a number of configuration parameters for MIPC (see Parameters in the multios_ipc_lkm-prop.ko Kernel Module on page 31).

- **multios_ipc_lkm-gpl.ko**

  This module contains open-source General Public License (GPL) code. It contains configuration parameters for the MIPC Network Device (MND) and MIPC Serial Device (MSD) features (see Parameters in the multios_ipc_lkm-prop.ko Kernel Module on page 32).

If you use the **mipcload** command to load the modules (see Loading the MIPC Kernel Modules and Setting Parameters with the mipcload Command on page 33), you do not need to specify the module names and it automatically loads both modules in the correct order.

NOTE: If you are going to be using the MIPC Serial Device (MSD) feature, you need to make entries for MSD devices in your target's file system before you load MIPC (see Wind River Linux Guest OS for Hypervisor 1.1 Programmer's Guide: MIPC Serial Device (MSD)).

4.4.1. Parameters in the multios_ipc_lkm-prop.ko Kernel Module

Each of the parameters in the **multios_ipc_lkm-prop.ko** kernel module corresponds to one of the Konfig parameters listed in Table 3-1 on page 21 and functions in the same way. Table 3-3 on page 31 gives cross-references from the kernel-module parameters to the Kconfig parameters.

Table 3-3 : Correspondence between MIPC Target Load-Time Parameters and Kconfig Parameters

Load-Time Parameter	Cross-Reference to Kconfig Parameter
buslist	List of available MIPC buses [MIPC_SM_BUS_LIST] on page 22
buffers	Maximum buffers per bus [MIPC_SM_BUFFERS] on page 21
mtu	Maximum buffer size per bus [MIPC_SM_MTU] on page 24
nodes	Maximum nodes per bus [MIPC_SM_NODES] on page 25
ports	Maximum ports per bus [MIPC_SM_PORTS] on page 25
events	Event pool size per bus [MIPC_SM_EVENTS] on page 23
qos	Quality of service used by buses [MIPC_SM_QOS] on page 25
	Maximum number of sockets [MIPC_SM_SOCKETS] on page 26

sockets	
initmode	Shared memory initialization mode [MIPC_SM_INIT_MODE] on page 24
heartbeat	Heartbeat ratex [MIPC_SM_HEARTBEAT_ PERIOD] on page 23

4.4.2. Parameters in the multios_ipc_lkm-prop.ko Kernel Module

The **multios_ipc_lkm-gpl.ko** kernel module contains parameters for both the MIPC Network Device (MND) and MIPC Serial Device (MSD) features. Table1 lists the MND parameters. Table2 lists the MSD parameters. For more information on these parameters, see Wind River Linux Guest OS for Hypervisor 1.1 Programmer's Guide: MND: Simulating an Ethernet Connection Between Nodes) and Wind River Linux Guest OS for Hypervisor 1.1 Programmer's Guide: MIPC Serial Device (MSD).

MND Parameters

Table 3-4 : MND Load-Time Parameters

MND Load-Time Parameters	Description
mnd_watchdog_timeo	The number of jiffies to wait for a notification that a send operation has completed. Default: 5.
mnd_napi_weight	The maximum number of packets to process in a poll. Default: 64.
mnd_mtu	The maximum size, in bytes, of the data portion (consisting of Ethernet frames) of an MND message. Default: 1520.
mnd_tx_bufs	The maximum number of transmission buffers allocated to MIPC control and data sockets. Default: -1.
mnd_rx_bufs	The number of receive buffers allocated to MIPC control and data sockets. Default: -1.
mnd_cfg_str	A configuration string that assigns one or more MND devices to the current node. Default: **"#unit=0,segment=0,port=23,bus=bus0"**.

MSD Parameters

Table 3-5 : MSD Load-Time Parameters

Load-Time Parameter	Description
msd_tx_bufs	The number of transmission buffers of the underlying MIPC socket (MSD uses a single socket per bus). Default: 8.
msd_mtu	The maximum size, in bytes, of the data that follows the MSD header. Default: -1.
msd_num_devs	The number of MSDs to create on this CPU. Default: 1.
msd_cfg_str	A configuration string that pairs each local MSD with a remote MSD. Default: **"#auto=y bus=main"**.

4.4.3. Loading the MIPC Kernel Modules and Setting Parameters with the mipcload Command

To load the MIPC kernel modules and set their parameters, you can use the **mipcload** command from the command line.020#4289d796-fe0e-493e-8a4b-e8ff4fdfdb57__wp163619 on page 33 The syntax for loading the kernel modules and setting parameters is:

mipcload [param1=value] [param2=value] ...[paramN=value]

If value is a string value, it must be enclosed in quotes, and no spaces are allowed. parameters listed in the string must be separated by commas. The following example sets the maximum number of nodes per bus to 8, the maximum number of ports to 48, and sets the maximum number of buffers on a bus named **main** to 100.

mipcload nodes=8 ports=48 buslist="#main=0,buffers=100"

If you use the **mipcload** command by itself, it loads the MIPC kernel modules and uses the configuration settings specified at build time.020#4289d796-fe0e-493e-8a4b-e8ff4fdfdb57__wp163531 on page 34

Note that you cannot use the **rmmod** command to unload the MIPC kernel modules.

[1] on page 33If your project includes MSD, before you load the MIPC kernel modules, you need to make entries for MSD devices in your target's file system (see Wind River Linux Guest OS for Hypervisor 1.1 Programmer's Guide: MIPC Serial Device (MSD)).

[2] on page 33If you have done a custom installation that may have altered the path to the MIPC kernel modules, you can use the **insmod** command (see the **insmod** man page) and specify the full path to the modules.

5. MIPC_ API FOR KERNEL APPLICATIONS

5.1. Introductions

For kernel applications, MIPC provides the **mipc**_ API. Two useful features of the mipc_ API are zero-copy buffers for sending messages from one node to another and express data transfer, for sending short messages between sockets (see Express Data Transfer on page 48)

The routines in the **mipc**_ API can be divided into three main groups:

- Routines that correspond to standard socket API calls (see mipc _Socket API on page 37).

  The **mipc**_ API provides functions for opening and closing sockets, creating ports for services and binding them to sockets, and for sending and receiving data on a socket.

- Extensions to the standard socket API for handling zero-copy buffers and performing other operations (see mipc_ Routines Outside of the Socket API on page 40).

- Callbacks for user implementation (see mipc_ API Callback Routines on page 41).

  The callbacks are for handling events such as the establishment or loss of a connection, availability of other nodes for communication, and availability of buffers for transmitting messages.

This chapter describes the **mipc**_ API, gives code examples showing the use of individual routines (see mipc_ API Code Examples on page 49), and provides a sample application (see mipc_ API Sample Application (VxWorks) on page 64).

5.2. The mipc_ API

This section presents the **mipc**_ API. It is organized as follows:

In all code that makes use of the **mipc**_ API, you need to include the **mipc.h** header file, as follows:

```
#include <multios_ipc/mipc.h>
```

5.2.1. mipc_ API Types

The **mipc**_ API defines the following data types:

- **mipc_sockaddr** structure (see The mipc_sockaddr Structure on page 36)
- **MIPC_ZBUFFER** (see The MIPC_ZBUFFER Structure on page 36)
- **mipc_stats**structure (see The mipc_stats Structure on page 36)
- Type definitions for callback routines.

  MIPC provides a number of type definitions for callbacks that are invoked through the **mipc_setsockopt( )** routine. For the individual type definitions, refer directly to the API reference entry for **mipc_setsockopt( )**. For information on the callbacks see mipc_ API Callback Routines on page 41.

The mipc_sockaddr Structure

The definition of the **mipc_sockaddr** structure is:

```
struct mipc_sockaddr    {

    unsigned short family; /* family number (MIPC_AF) */

    unsigned short busNum; /* bus number */

    unsigned short nodeNum; /* node number */

    unsigned short portNum; /* port number */

};
```

where

family is the socket address family for the **mipc_** API: **MIPC_AF**

busNum is the number of the bus that the socket is bound to.

nodeNum is the number of the node that the socket is bound to.

portNum is the number of the port that the socket is bound to.

For use with the **mipc_bind( )** routine, there are macro definitions for the busNum, nodeNum, and portNum fields (see Macros for Fields in the mipc_sockaddr Structure When Used with mipc_bind( ) on page 39).

The MIPC_ZBUFFER Structure

The definition of the **MIPC_ZBUFFER** structure is:

typedef void * MIPC_ZBUFFER

MIPC_ZBUFFER is used as an opaque handle to a zero-copy buffer (received or locally allocated) in MIPC zero-copy routines.

The mipc_stats Structure

The definition of the **mipc_stats** structure is:

```
typedef struct
{
    uint32_t bufs_sent;              /* # buffers sent */
    uint32_t bufs_rcvd;              /* # buffers received */
    uint32_t bytes_sent;             /* # buffer bytes sent */
    uint32_t bytes_rcvd;             /* # buffer bytes received */
    uint32_t bufs_allocated;         /* # buffers allocated */
    uint32_t bufs_freed;             /* # buffers freed */
    uint32_t buf_cong_wait;          /* buffer congestion (blocking) */
    uint32_t buf_cong_no_wait;       /* buffer congestion (non-blocking) */
    uint32_t buf_avail_sent;         /* buffer available events sent */
    uint32_t buf_avail_rcvd;         /* buffer available events received */
    uint32_t express_msgs_sent;      /* express messages sent */
    uint32_t express_msgs_rcvd;      /* express messages received */
    uint32_t buffer_not_avail;       /* couldn't get buffer from packet pool */
    uint32_t event_not_avail;        /* couldn't get event from event pool */
    uint32_t interrupts_sent;        /* # interrupts sent */
    uint32_t interrupts_rcvd;        /* # interrupts received */
    uint32_t interrupts_deferred;    /* # invocations of deferred work thead */
    uint32_t sockets_bound;          /* # sockets bound to bus */
    uint32_t sockets_closed;         /* # sockets closed after binding to bus */
    uint32_t socket_rx_queue_err;    /* # socket rx queue errors (full q) */
};
```

The **mipc_stats** structure receives values returned by the **mipc_getstats( )** routine.

5.2.2. mipc _Socket API

This section lists the routines that make up the **mipc_** socket API. In writing applications that use the API, the following points need to be considered:

- The **mipc_** API does not provide a mechanism for preventing two applications or two threads within an application from accessing a socket at the same time.

  As a result, applications need to access sockets in a single-threaded manner and ensure that there is no conflict with other applications attempting to access a socket.

- A MIPC socket can only be bound to a single port.

  In this document, when the term port is used, it can also refer to the socket bound to the port.

- Port 0 is reserved by MIPC. In addition, when MSD and MND are included in an image, port 1 is used for MSD (see the VxWorks AMP Programmer's Guide: MIPC Serial Devices (MSD)), and port 2 is used for WDB.

- A node, in the context of MIPC programming, is an instance of an operating system running in a CPU; node numbers do not always correspond to CPU numbers (see Terminology on page 3).

- An application should not close a MIPC socket if it is holding any zero-copy buffers, otherwise the buffers will become unavailable for re-use by other sockets. This prohibition applies both to buffers that the application has requested for sending (for example, buffers obtained using **mipc_zalloc( )**), and to buffers that have been sent to it (for example, buffers obtained using **mipc_zrecv( )**).

The **mipc_** socket API is summarized in Table 4-1 on page 37. For detailed information on individual routines, see the API reference.

Table 4-1 : MIPC Socket API

SocketRoutine	Description
mipc_accept( )	Accept a request for a connection to a MIPC socket.
mipc_bind( )	Bind an address to a MIPC socket. There are special macros for **mipc_bind( )** to use in filling in fields of the **mipc_sockaddr** structure (see Macros for Fields in the mipc_sockaddr Structure When Used with mipc_bind( ) on page 39).
mipc_close( )	Close a MIPC socket.
mipc_connect( )	Request a connection to a MIPC socket.
mipc_getpeername( )	Get the address of a peer MIPC socket.
mipc_getsockname( )	Get the address of a MIPC socket.
mipc_getsockopt( )	Get the value of an option associated with a MIPC socket.

mipc_listen( )	Enable a MIPC socket to receive connection requests.
mipc_recv( )	Receive data from a MIPC socket.
mipc_recvfrom( )	Receive data from a MIPC socket.
mipc_send( )	Send a message to a MIPC socket.
mipc_sendto( )	Send a message to a specified MIPC socket.
mipc_setsockopt( )	Set the value of an option associated with a MIPC socket.
mipc_shutdown( )	Shut down communication by a MIPC socket.
mipc_socket( )	Create a MIPC socket.

For code examples showing the use of socket and non-socket routines in the **mipc_** API, see mipc_ API Code Examples on page 49.

Macros for Fields in the mipc_sockaddr Structure When Used with mipc_bind( )

There are three macros for use in the busNum, nodeNum, and portNum fields of the **mipc_sockaddr** structure (see The mipc_sockaddr Structure on page 36) when it is used with the **mipc_bind( )** routine. In each case, the macro allows MIPC to fill in the field value, rather than requiring the application to find the value for the field.

Macro	Description
MIPC_BUS_ANY	MIPC maps this macro to the first bus it attaches to at startup and returns the corresponding bus number in the busNum field of the **mipc_sockaddr** structure. This allows an application running on a node configured for only one bus to get the bus number needed for calls to **mipc_connect( )**, **mipc_send_express( )**, **mipc_sendto( )**, and **mipc_zsendto( )** without needing to call **mipc_getbusbyname( )**.
MIPC_NODE_ANY	

	This macro tells MIPC to bind to the node calling **mipc_bind( )**, which means that the application does not need to find out its own node number. The node number is returned in the nodeNum field of the **mipc_sockaddr** structure.
MIPC_PORT_ANY	This macro tells MIPC to choose an available port number to bind to. The port number is returned in the portNum field of the **mipc_sockaddr** structure.

5.2.3. mipc_ Routines Outside of the Socket API

Table 4-2 on page 40 lists MIPC routines that do not correspond to routines in the standard socket API.

Table 4-2 : MIPC Routines That are Not Part of the Socket API

mipc_Routine	Description
mipc_addr2offset( )	Return an offset from the start of shared memory given a local pointer into shared memory.
mipc_clearstats( )	(Deprecated) Clear MIPC statistics.
mipc_clearstatsbybus( )	Clear MIPC statistics for a bus.
mipc_getactivebusesl( )	Get a bitfield of the buses a node can utilize.
mipc_getactivenodes( )	Get a bit field of the active MIPC nodes on a bus.
mipc_getactivenodesl( )	Get a bit field of the active MIPC nodes on a bus.
mipc_getbusbyname( )	Get the number of a MIPC bus, given its name.
mipc_getbusmaxnodes( )	Get the maximum number of nodes supported on a specified bus.
mipc_getnamebybus( )	Get the name of a MIPC bus, given its number.
mipc_getnodebybus( )	Given a bus number, get the node number of the current node.
mipc_getstats( )	(Deprecated) Get MIPC statistics.

mipc_getuserhandle( )	Get the user handle associated with a socket.
mipc_offset2addr( )	Return the local address given an offset in shared memory.
mipc_processbus( )	Read and process the incoming event queue on a bus.
mipc_sendexpress64( )	Send a short, 64-bit express data message to a MIPC socket.
mipc_setuserhandle( )	Specify a user handle that is to be associated with a socket.
mipc_zalloc( )	Allocate a zero-copy buffer.
mipc_zfree( )	Free a zero-copy buffer.
mipc_zfreerx( )	(Deprecated) Free a zero-copy buffer kept by a receive callback.
mipc_zrecv( )	Receive a zero-copy message from a MIPC socket.
mipc_zrecvfrom( )	Receive a zero-copy message from a MIPC socket.
mipc_zsend( )	Send a zero-copy message to a MIPC socket.
mipc_zsendto( )	Send a zero-copy message to a specified MIPC socket.

For code examples showing the use of socket and non-socket routines in the MIPC API, see mipc_ API Code Examples on page 49.

5.2.4. mipc_ API Callback Routines

MIPC provides a set of callback routines for users to implement. Implementation of the routines is optional. The callbacks are assigned to sockets through options in the **mipc_getsockopt( )** and **mipc_setsockopt( )** routines (see the API reference entry for **mipc_setsockopt( )**).

A socket's callback routines are invoked in the execution context of the bus it is on. The **Quality of service used by buses** (**MIPC_SM_QOS**) parameter (see the entry for **Quality of service used by buses** in Table 2-2 on page 9 (VxWorks) or Table 3-1 on page 21 (Linux) determines the way sockets bound to a bus process received events such as connection attempts and the arrival of packets. If the sockets on a bus are configured to process received events at interrupt level, any user callbacks associated with the sockets will execute at interrupt level and must be free of blocking operations. Blocking operations at interrupt level can result in system instability.

The following callbacks are available:

MIPC_CONNECTED_CALLBACK( ) on page 42, for responding to successful establishment of a connection.

MIPC_CONNECTREFUSED_CALLBACK( ) on page 42, for handling a connection refusal.

MIPC_CONNECTREQUEST_CALLBACK( ) on page 43, for responding to connection requests received at a listening socket.

MIPC_DISCONNECTED_CALLBACK( ) on page 43, for responding to a disconnection.

MIPC_EXPRESS_CALLBACK( ) on page 44, for handling express data received at a socket.

MIPC_NODEJOIN_CALLBACK( ) on page 44, for responding when a node becomes available on a bus.

MIPC_NODELEFT_CALLBACK( ) on page 45, for responding when a node leaves a bus and is unavailable for communication.

MIPC_RX_CALLBACK( ) on page 46, for handling zero-copy buffers received at a socket. A buffer can be immediately released, queued for a MIPC receive routine, or held for further processing.

MIPC_RXQUEUED_CALLBACK( ) on page 45, for handling buffers when they are enqueued in the receive queue of a socket.

MIPC_TXBUFAVAIL_CALLBACK( ) on page 47, for immediate response when a transmission buffer becomes available.

MIPC_CONNECTED_CALLBACK( )

The **MIPC_CONNECTED_CALLBACK( )** routine is called when a connection is established on the socket.

Syntax

The syntax of the **MIPC_CONNECTED_CALLBACK( )** routine is:

```
void MIPC_CONNECTED_CALLBACK
        (
        int sockfd              /* Socket descriptor */
        )
```

MIPC_CONNECTREFUSED_CALLBACK( )

The **MIPC_CONNECTREFUSED_CALLBACK( )** routine is called when a request to connect to a socket is refused.

Syntax

The syntax of the **MIPC_CONNECTREFUSED_CALLBACK( )** routine is:

```
void MIPC_CONNECTREFUSED_CALLBACK
        (
        int sockfd              /* Socket descriptor */
        )
```

MIPC_CONNECTREQUEST_CALLBACK( )

The **MIPC_CONNECTREQUEST_CALLBACK( )** routine is called when a connection request has been received on a listening MIPC socket.

Syntax

The syntax of the **MIPC_CONNECTREQUEST_CALLBACK( )** routine is:

```
void MIPC_CONNECTREQUEST_CALLBACK
        (
        int sockfd,                 /* Socket descriptor */
        uint16_t srcNode,           /* Node that sent request */
        uint16_t srcPort,           /* Port that sent request */
        uint64_t data               /* data */
        )
```

MIPC_DISCONNECTED_CALLBACK( )

The **MIPC_DISCONNECTED_CALLBACK( )** routine is called when the socket is disconnected.

Syntax

The syntax of the **MIPC_DISCONNECTED_CALLBACK( )** routine is:

```
void MIPC_DISCONNECTED_CALLBACK
```

```
        (
        int sockfd              /* Socket descriptor */
        )
```

MIPC_EXPRESS_CALLBACK( )

The **MIPC_EXPRESS_CALLBACK( )** routine is called when express data is received at a socket.

Syntax

The syntax of the **MIPC_EXPRESS_CALLBACK( )** routine is:

```
void MIPC_EXPRESS_CALLBACK
        (
        int sockfd                  /* Socket descriptor */
        unsigned short srcNode,     /* Node that sent data */
        unsigned short srcPort,     /* Port that sent data */
        unsigned short data         /* Data */
        )
```

MIPC_NODEJOIN_CALLBACK( )

The **MIPC_NODEJOIN_CALLBACK( )** routine is called for each node that joins the bus on which the socket is bound.

Syntax

The syntax of the **MIPC_NODEJOIN_CALLBACK( )** routine is:

```
void MIPC_NODEJOIN_CALLBACK
        (
        int sockfd,              /* Socket descriptor */
        unsigned short nodeNum   /* Node that joined the bus */
        )
```

MIPC_NODELEFT_CALLBACK( )

The **MIPC_NODELEFT_CALLBACK( )** routine is called for each node that leaves the bus on which the socket is bound.

Syntax

The syntax of the **MIPC_NODELEFT_CALLBACK( )** routine is:

```
void MIPC_NODELEFT_CALLBACK
        (
        int sockfd,              /* Socket descriptor */
        unsigned short nodeNum   /* Node that left the bus */
        )
```

MIPC_RXQUEUED_CALLBACK( )

The **MIPC_RXQUEUED_CALLBACK( )** routine is called for each zero-copy buffer that is added to the receive queue of a socket. When the callback is called, data is ready to be received and the application can call one of the following receive routines without blocking:

- For receiving data into a user-defined area:
  - **mipc_recv( )**
  - **mipc_recvfrom( )**
- For receiving data in a zero copy buffer:
  - **mipc_zrecv( )**
  - **mipc_zrecvfrom( )**

Syntax

The syntax of the **MIPC_RXQUEUED_CALLBACK( )** routine is:

```
int MIPC_RXQUEUED_CALLBACK
        (
        int sd,             /* socket that queued a message */
```

```
        )
```

In the current release, the return value must always be 0.

MIPC_RX_CALLBACK( )

The **MIPC_RX_CALLBACK**( ) routine is called for each zero-copy buffer received on the socket. It allows you to immediately release the buffer, queue it for one of the MIPC receive routines (**mipc_recv( )**, **mipc_recvfrom( )**, **mipc_zrecv( )**, **mipc_zrecvfrom( )**), or hold it for further processing.

Syntax

The syntax of the **MIPC_RX_CALLBACK( )** routine is:

```
int MIPC_RX_CALLBACK
        (
        int sockfd,                  /* socket that received message */
        const unsigned char *buff,   /* pointer to received buffer */
        size_t nbytes,               /* size of buffer */
        unsigned short srcNodeNum,   /* sender's node number */
        unsigned short srcPortNum,   /* sender's port number */
        MIPC_ZBUFFER zbuf            /* handle to buffer */
        )
```

If you keep the received zero-copy buffer for further processing, the zbuf parameter allows you to release the buffer using the **mipc_zfree( )** routine.

Return Value

MIPC_RX_CALLBACK( ) returns one of the following values:

Table 4-3 : MIPC_RX_CALLBACK( ) Return **Values**

Return Value	Description
MIPC_RX_RELEASEBUF	Release the data buffer.

MIPC_RX_QUEUEBUF	Queue the data buffer, making it available to the **mipc_recv( )** and **mipc_recvfrom( )** routines.
MIPC_RX_KEEPBUF	Keep the buffer without queuing it. When the application is ready to release the buffer, it should call **mipc_zfree( )**.

MIPC_TXBUFAVAIL_CALLBACK( )

The **MIPC_TXBUFAVAIL_CALLBACK( )** routine is called when a socket that had no available transmission buffers once again has a buffer available for transmission.

Syntax

The syntax of the **MIPC_TXBUFAVAIL_CALLBACK( )** routine is:

```
void MIPC_TXBUFAVAIL_CALLBACK
        (
        int sockfd              /* Socket descriptor */
        )
```

5.2.5. Validating Parameters in mipc_ API Routines

For purposes of debugging during application development, you can turn on parameter checking for routines belonging to the **mipc_** API. To do this:

1. Open **mipc_sm_adapt.h** for editing.

   The location of the file is:

   For VxWorks:

installDir**/vxworks-6.**x**/target/h/multios_ipc/mipc_sm_adapt.h**

   For Linux:

projectDir**/build/linux/drivers/multios_ipc_lkm/multios_ipc/mipc_sm_adapt.h**

2. Define the compiler flag **MIPC_CHECK_ARGS**.

   By default, the flag is undefined in the file. Locate the flag in the file, and set it to:

   #define MIPC_CHECK_ARGS

3. Recompile the MIPC code, as follows:

For VxWorks:

cd target/src/multios_ipcmake CPU=cpu **TOOL=**toolchain

For Linux you need to rebuild the kernel:

make -C build linux.rebuild

Using parameter checking increases the size of the MIPC code and can impair performance. Once you have debugged and regression tested your code, you should make sure to turn off parameter checking, either by explicitly undefining **MIPC_CHECK_ARGS** in **mipc_sm_adapt.h** or by commenting it out, and then recompile your code, as in Step on page 47040#b5a0f003-0940-492c-aad4-65b5eab05884__wp168973 on page 47.

5.2.6. mipc_ API Error Codes

MIPC error codes have a one-to-one correspondence with POSIX error codes. For example, the **MIPC_EINVAL** error code directly corresponds to the POSIX **EINVAL** error code. Defines for the MIPC error codes are in:

For VxWorks:

installDir**/vxworks-6.**x**/target/h/multios_ipc/mipc_os_adapt.h**

For Linux:

projectDir**/build/linux/drivers/multios_ipc_lkm/multios_ipc/mipc_os_adapt.h**

5.3. Express Data Transfer

Express data transfer is a feature that allows MIPC kernel applications or kernel threads to send short, 64-bit messages to each other more rapidly than they could using either standard or zero-copy data transfer.

You can send express data from any bound socket to any port on the same bus as the socket. For example, you can send express data from a **MIPC_SOCK_STREAM** socket to a **MIPC_SOCK_RDM** socket. Note, however, that MIPC does not provide flow control to regulate express data messaging. If express data is sent too frequently, it can interfere with the handling of non-express messages and with connection handshaking.

To send express data, you need to use the following routines:

- **mipc_sendexpress64( )** (for information, see the API reference)

To receive express data, you need to use the following callback routine:

- **MIPC_EXPRESS_CALLBACK( )** (see MIPC_EXPRESS_CALLBACK( ) on page 44).

Usage Example: Exchanging Pointers to Locations in Shared Memory

Sockets can exchange data in shared memory by sending express data that contains pointers to data in shared memory. To implement this, a sender can allocate a shared-memory buffer using the **mipc_zalloc( )** routine. It then fills the buffer with information, converts a pointer to the buffer to an offset (see Routines for Shared-Memory Address Translation on page 49), and sends the offset to the receiver.

The receiving application gets the offset and, after converting for node addressing (see Routines for Shared-Memory Address Translation on page 49), directly accesses the shared-memory buffer to respond to the information in the buffer. It can then signal that it has completed its use of the buffer by sending an express-data acknowledgement back to the sender.

You can also use the exchange of pointers to buffers in shared memory to implement shared variables for applications on different nodes. In this case, to maintain the integrity of the variables, you can use the atomic operations provided for AMP (see VxWorks Kernel Programmer's Guide: Atomic Operators for AMP).

Routines for Shared-Memory Address Translation

Operating systems on separate nodes may address shared memory at different locations. The following two routines are available to simplify address translation between operating systems:

- **mipc_addr2offset( )** takes a shared-memory address and returns its offset.
- **mipc_offset2addr( )** takes an offset into shared memory and returns its address.

5.4. Mipc_ API Code Examples

This section provides code examples showing the use of individual routines in the MIPC API. The routines appear in alphabetical order. The following deprecated routines are excluded:

mipc_clearstats( )mipc_getstats( )mipc_zfreerx( )

The code illustrating a particular routine has error checking for that routine but, in order to keep examples short, there is no error checking for other routines called within the example. Thus, in the example for **mipc_accept( )**, the call to **mipc_accept( )** is validated, but there is no validation of calls to **mipc_socket( )** and **mipc_bind( )**.

The examples assume the following definitions and variable declarations:

```
#define TEST_BUSNAME "main"

#define TEST_PORTNUM 3

#define TEST_MSG "Hello"

#define TEST_MSG_REPLY "Hi Back"

#define TEST_MSG_SIZE (sizeof(TEST_MSG_REPLY))

#define TEST_MSG_REPLY_SIZE (sizeof(TEST_MSG))


int sd_rdm;                 /* rdm socket sd */

int sd_seqpkt;              /* sequential socket sd */

int sd_accept;              /* accepted socket sd */

int busnum;                 /* bus number we want to bind to */
```

```
int rc;                     /* return code */
int nbytes;                 /* number of bytes */
struct mipc_sockaddr addr;  /* binding address */
struct mipc_sockaddr peer;  /* peer address */
int addrlen;                /* length of a mipc_sockaddr structure */
uint8_t *buf;               /* pointer to a buffer */
char buffer[100];           /* buffer for sending or receiving */
MIPC_ZBUFFER zbuf;          /* zero copy buffer */
```

mipc_accept( )

```
char busname[MIPC_SM_MAX_NAME_LEN];       /* storage for bus name */
struct mipc_stats statsbuf;               /* storage for MIPC statistics */
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
mipc_listen(sd_seqpkt, 2);

sd_accept = mipc_accept(sd_seqpkt, &peer, &peerlen);
if (sd_accept < 0)
        {
        MIPC_PRINT("Could not accept a connection (err=%d)\n",sd_accept);
        }
```

mipc_bind( )

```
addr.family = MIPC_AF;
addr.busNum = busnum;
addr.nodeNum = MIPC_NODE_ANY;
addr.portNum = TEST_PORTNUM;
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
```

```
if (mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr)) < 0)
        {
        MIPC_PRINT("Failed to bind socket (%d).\n", sd_seqpkt);
        }
```

mipc_clearstatsbybus( )

```
/* clear the statistics */
mipc_getbusbyname(TEST_BUSNAME, &busnum);
if (mipc_clearstatsbybus(busnum) < 0)
        {
        MIPC_PRINT("Failed to clear stats (incorrect bus number?\n");
        }
```

mipc_close( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
addr.family = MIPC_AF;
addr.busNum = busnum;
addr.nodeNum = MIPC_NODE_ANY;
addr.portNum = TEST_PORTNUM;
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
if (mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr)) < 0)
        {
        MIPC_PRINT("Failed to bind socket (%d).\n", sd_rdm);
        mipc_close(sd_rdm);
        return;
        }
mipc_sendto(sd_rdm, sbuf, TEST_MSG_SIZE, 0, &peer, addrlen);
```

```
mipc_close (sd_rdm);
```

mipc_connect( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
addr.family = MIPC_AF;
addr.busNum = busnum;
addr.nodeNum = MIPC_NODE_ANY;
addr.portNum = TEST_PORTNUM;
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
peer.busNum = busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;

if (mipc_connect(sd_seqpkt, &peerAddr,
        sizeof(struct mipc_sockaddr)) < 0)
        {
        MIPC_PRINT("Failed to connect to peer socket.\n");
        }
```

mipc_getactivenodes( )

```
unsigned long long nodes;
mipc_getbusbyname(TEST_BUSNAME, &busnum);

if (mipc_getactivenodes(busnum, &nodes) < 0)
```

```
        {
        MIPC_PRINT("Failed to get active nodes.\n");
        }
```

mipc_getactivenodesl( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);

if (mipc_getactivenodesl(busnum, &buffer, 5 /* eg length */) < 0)
        {
        MIPC_PRINT("Failed to get active nodes.\n");
        }
```

mipc_getbusbyname( )

```
if (mipc_getbusbyname(TEST_BUSNAME, &busnum) < 0)
        {
        MIPC_PRINT("Failed to find bus '%s'.\n", TEST_BUSNAME);
        return -1;
        }
```

mipc_getnamebybus( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);

if (mipc_getnamebybus(busnum, busname, MIPC_SM_BUS_MAX_NAME_LEN) < 0)
        {
        MIPC_PRINT("Failed to get bus name for bus %d.\n", busnum);
        }
```

mipc_getnodebybus( )

```
if (mipc_getnodebybus(busnum) < 0)
        {
        MIPC_PRINT("Failed to get node number on invalid bus %d.\n", busnum);
        }
```

mipc_getpeername( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
addr.busNum = busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));

rc = mipc_getpeername(sd_seqpkt, &peeraddr, &addrlen);
if (rc != MIPC_OK)
        {
        MIPC_PRINT("mipc_getpeername() returned error %d.\n", rc);
        }
```

mipc_getsockname( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
addr.family = MIPC_AF;
addr.busNum = busnum;
```

```
addr.nodeNum = MIPC_NODE_ANY;
addr.portNum = TEST_PORTNUM;
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);

if (mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0)
        {
        MIPC_PRINT("Failed to get address of socket (%d).\n", sd_seqpkt);
        }
```

mipc_getsockopt( )

```
int optVal;
int optLen = sizeof(optVal);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);

rc = mipc_getsockopt(sd_seqpkt, MIPC_SOL, MIPC_SO_TXBUFS,
        (void *)&optVal, &optLen);
if (rc < 0)
        {
        MIPC_PRINT("Failed to get sock option MIPC_SO_TXBUFS
            (err=%d)\n", rc);
        }
```

mipc_getstatsbybus( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);

if (mipc_getstatsbybus(&statsbuf, sizeof(struct mipc_stats)) < 0)
        {
        MIPC_PRINT("Failed to get statistics on bus %d.  Is the MIPC_STATS_MODE set to 1?\n", busnum);
```

```
        }
```

mipc_listen( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);

if (mipc_listen(sd_seqpkt, 2) < 0)
        {
        MIPC_PRINT("Failed to set socket up to listen.\n");
        }
```

mipc_processbus( )

```
/* Process any events on a given bus */
if (mipc_processbus(busnum) < 0)
    {
    MIPC_PRINT("Failed to process unreachable bus %d.\n", busnum);
    }
```

mipc_recv( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
addr.family = MIPC_AF;
addr.busNum = busnum;
addr.nodeNum = MIPC_NODE_ANY;
addr.portNum = TEST_PORTNUM;
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
```

```
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));

nbytes = mipc_recv(sd_seqpkt, &buffer, sizeof(buffer), 0);
if (nbytes <= 0)
        {
        MIPC_PRINT("Could not receive a message from peer (err=%d)\n",
            nbytes);
        }
```

mipc_recvfrom( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);

nbytes = mipc_recvfrom(sd_rdm, &buffer, sizeof(buffer),0,&peer,
                        &addrlen);
if (nbytes <= 0)
        {
        MIPC_PRINT("Could not receive a message (err=%d)\n", nbytes);
        }
```

mipc_send( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
```

```
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));
strcpy(buffer, TEST_MSG);

nbytes = mipc_send(sd_seqpkt, buffer, TEST_MSG_SIZE, 0);
if (nbytes < TEST_MSG_SIZE)
        {
        MIPC_PRINT("Could not send the message to peer (err=%d)\n",
            nbytes);
        mipc_zfree(sd_seqpkt, zbuf);
        }
```

mipc_sendto( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_rdm, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
strcpy(sbuf, TEST_MSG);

nbytes = mipc_sendto(sd_rdm, sbuf, TEST_MSG_SIZE, 0, &peer, addrlen);
if (nbytes < TEST_MSG_SIZE)
        {
        MIPC_PRINT("Could not send the message to peer (err=%d)\n",
```

```
            nbytes);
        }
```

mipc_sendexpress64( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr));

peer.family = MIPC_AF;
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
if (mipc_sendexpress64(sd_rdm, &peer, data64) < 0)
        {
        MIPC_PRINT("Failed to send express data to peer.\n");
        }
```

mipc_setsockopt( )

```
int optVal = 8;
int optLen = sizeof(optVal);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);

rc = mipc_setsockopt(sd_seqpkt, MIPC_SOL, MIPC_SO_TXBUFS,
    (void *)&optVal, &optLen );
if (rc < 0)
        {
        MIPC_PRINT("Failed to set sock option MIPC_SO_TXBUFS
            (err=%d)\n", rc);
        }
```

mipc_shutdown( )

```
addr.family = MIPC_AF;
addr.nodeNum = MIPC_NODE_ANY;
addr.busNum = busnum;
addr.portNum = TEST_PORTNUM;
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
if (mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr)) < 0)
        {
        MIPC_PRINT("Failed to bind socket (%d).\n", sd_rdm);
        mipc_close(sd_rdm);
        return;
        }
mipc_sendto(sd_rdm, sbuf, TEST_MSG_SIZE, 0, &peer, addrlen);

mipc_shutdown(sd_rdm, MIPC_SHUT_RDWR);
```

mipc_socket( )

```
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
if (sd_seqpkt < 0)
        {
        MIPC_PRINT("Could not create a MIPC socket (error = %d)\n",
            sd_seqpkt);
        return -1;
        }
```

mipc_zalloc( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
```

```
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));

if (mipc_zalloc(sd_seqpkt, 0 /* MTU size */, &buf, &zbuf, 0) < 0)
        {
        MIPC_PRINT("mipc_zalloc could not allocate a buffer (possibly would block
            on a non-blocking socket)\n");
        }
```

mipc_zallocbuf( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));
if (mipc_zallocbuf(sd_seqpkt, &buf, &zbuf, 0) < 0)
        {
        MIPC_PRINT("mipc_zallocbuf could not allocate a buffer (possibly would block on a non-blocking soc
        }
```

mipc_zfree( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));
mipc_zrecv(sd_seqpkt, &buf, &zbuf, 0);

mipc_zfree(sd_seqpkt, zbuf);
```

mipc_zrecv( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));

nbytes = mipc_zrecv(sd_seqpkt, &buf, &zbuf, 0);
if (nbytes < 0)
        {
        MIPC_PRINT("Could not recv a peer response (err=%d)\n", nbytes);
        }
```

mipc_zrecvfrom( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);
mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
```

```
nbytes = mipc_zrecvfrom(sd_rdm, &buf, &zbuf, 0, &peer, &addrlen);
if (nbytes < 0)
        {
        MIPC_PRINT("Could not recv a peer response (err=%d)\n", nbytes);
        }
```

mipc_zsend( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);
sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
mipc_bind(sd_seqpkt, &addr, sizeof(struct mipc_sockaddr));
addrlen = sizeof(struct mipc_sockaddr);
mipc_getsockname(sd_seqpkt, &addr, &addrlen) < 0);
peer.family = MIPC_AF;
peer.busNum = addr.busnum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;
mipc_connect(sd_seqpkt, &peerAddr, sizeof(struct mipc_sockaddr));
mipc_zalloc(sd_seqpkt, 0 /* MTU size */, &buf, &zbuf, 0);
strcpy(&buf, TEST_MSG);  /* assume MTU bigger than TEST_MSG */

nbytes = mipc_zsend(sd_seqpkt, zbuf, TEST_MSG_SIZE, 0);
if (nbytes < TEST_MSG_SIZE)
        {
        MIPC_PRINT("Could not send the message to peer (err=%d)\n",
            nbytes);
        mipc_zfree(sd_seqpkt, zbuf);
        }
```

mipc_zsendto( )

```
mipc_getbusbyname(TEST_BUSNAME, &busnum);

sd_rdm = mipc_socket(0, MIPC_SOCK_RDM, 0);

mipc_bind(sd_rdm, &addr, sizeof(struct mipc_sockaddr));

addrlen = sizeof(struct mipc_sockaddr);

mipc_getsockname(sd_rdm, &addr, &addrlen) < 0);

peer.family = MIPC_AF;

peer.busNum = addr.busnum;

peer.portNum = TEST_PORTNUM;

peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;

mipc_zalloc(sd_rdm, 0 /* MTU size */, &buf, &zbuf, 0);

strcpy(&buf, TEST_MSG); /* assume MTU bigger than TEST_MSG */


nbytes = mipc_zsendto(sd_rdm, zbuf, TEST_MSG_SIZE, 0, &peer,

                        addrlen);

if (nbytes < TEST_MSG_SIZE)

        {

        MIPC_PRINT("Could not send the message to peer (err=%d)\n",

            nbytes);

        mipc_zfree(sd_rdm, zbuf);

        }
```

5.5. Mipc_ API Sample Application (VxWorks)

The following application for VxWorks illustrates the use of the **mipc_** API with zero-copy buffers. It uses **mipc_zsend( )** to send a message and **mipc_zrecv( )** to receive a message. The sender and receiver are hardcoded to use nodes 0 and 1, but either node can be a sender or a receiver.

By default, the application uses MIPC bus **main**, but you can specify a different bus from the command line.

To run the application using **main**, from the command line, enter:

Receiving node:**mipc_demo_zrecv**

Sending node:**mipc_demo_zsend**

To run the application using a virtual bus other than **main**, specify the bus on the command line, as in the following example for **app1_secondary**:

Receiving node:**mipc_demo_zrecv "app1_secondary"**

Sending node:**mipc_demo_zsend "app1_secondary"**

Sample Application

```
/* Copyright (c) 2008-2009 Wind River Systems, Inc. */


#include <multios_ipc/mipc.h>

#include <string.h>


static int mycpu = 0;        /* this cpu number */

static int peercpu = 0;        /* my peer's cpu number */

#define TEST_BUSNAME "main"

#define TEST_PORTNUM 3

#define TEST_SEND_PORTNUM 0    /* use any port */

#define TEST_MSG "Hello"

#define TEST_MSG_REPLY "Hi Back"

#define TEST_MSG_SIZE sizeof(TEST_MSG)

#define TEST_MSG_REPLY_SIZE sizeof(TEST_MSG_REPLY)


/*************************************************************************

* mipc_demo_zsend - send a message to a peer using mipc_zsend.

*/

int mipc_demo_zsend (char * busname)

{

    int sd_seqpkt;            /* sequential socket sd */

    int busnum;            /* bus number we want to bind to */

    struct mipc_sockaddr addr;    /* binding address */

    struct mipc_sockaddr peer;    /* peer address */
```

```
size_t addrlen;            /* address structure length */
uint8_t *buf;             /* pointer to a buffer */
int nbytes;             /* number of bytes */
int rc;                 /* return code for operations */
MIPC_ZBUFFER zbuf;        /* buffer handle */

MIPC_PRINT("mipc_demo_zsend: Sequential Packet test with zero-copy.\n");
if ((rc = mipc_getbusbyname((!busname) ? TEST_BUSNAME : busname,
             &busnum)) < 0) {
    MIPC_PRINT("Failed to find bus '%s' (err=%d).\n",
     TEST_BUSNAME, rc);
    return -1;
}

sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
if (sd_seqpkt < 0) {
    MIPC_PRINT("Could not create a MIPC socket (error = %d)\n",
     sd_seqpkt);
    return -1;
}

addr.busNum = busnum;
addr.portNum = TEST_PORTNUM;

if ((rc = mipc_bind(sd_seqpkt, &addr,
          sizeof(struct mipc_sockaddr))) < 0) {
    MIPC_PRINT("Failed to bind socket (%d) (err=%d).\n", sd_seqpkt,
     rc);
    goto send_close;
}
```

```
addrlen = sizeof(struct mipc_sockaddr);
if ((rc = mipc_getsockname(sd_seqpkt, &addr, &addrlen)) < 0) {
    MIPC_PRINT("Failed to get address of socket (%d) (err=%d).\n",
     sd_seqpkt, rc);
    goto send_close;
}

peer.busNum = addr.busNum;
peer.portNum = TEST_PORTNUM;
peer.nodeNum = (addr.nodeNum == 0) ? 1 : 0;

if ((rc = mipc_connect(sd_seqpkt, &peer,
          sizeof(struct mipc_sockaddr))) < 0) {
    MIPC_PRINT("Failed to connect to peer socket (err=%d).\n", rc);
    goto send_close;
}

/* our socket blocks by default, so mipc_zalloc will always work */
mipc_zalloc(sd_seqpkt, 0 /* MTU size */, &buf, &zbuf, 0);

strcpy((char *)buf, TEST_MSG);        /* assume MTU > TEST_MSG */
MIPC_PRINT("We are sending '%s' to peer.\n", buf);

nbytes = mipc_zsend(sd_seqpkt, zbuf, TEST_MSG_SIZE, 0);
if (nbytes < TEST_MSG_SIZE) {
    MIPC_PRINT("Could not send the message to peer (err=%d)\n",
     nbytes);
    mipc_zfree(sd_seqpkt, zbuf);
    goto send_close;
```

```
    }
    nbytes = mipc_zrecv(sd_seqpkt, &buf, &zbuf, 0);
    if (nbytes < 0) {
        MIPC_PRINT("Could not recv a peer response (err=%d)\n", nbytes);
        goto send_close;
    }

    MIPC_PRINT("Peer replied with '%s'\n", buf);

    mipc_zfree(sd_seqpkt, zbuf);
    mipc_close (sd_seqpkt);
    return 0;

send_close:
    mipc_close (sd_seqpkt);
    return -1;
}

/*****************************************************************************
* mipc_demo_zrecv - recv a message from a peer using mipc_zrecv.
*/
int mipc_demo_zrecv (char * busname)
{
    int sd_seqpkt;            /* sequential socket sd */
    int sd_accept;            /* sequential socket sd */
    int busnum;            /* bus number we want to bind to */
    struct mipc_sockaddr addr;    /* binding address */
    struct mipc_sockaddr peer;    /* peer address */
    size_t addrlen;            /* address structure length */
    uint8_t *buf;            /* buffer */
```

```
int nbytes;             /* number of bytes */
int rc;                 /* return code for operations */
MIPC_ZBUFFER zbuf;         /* buffer handle */

MIPC_PRINT("mipc_demo_zrecv: Sequential Packet test with zero-copy.\n");

if ((rc = mipc_getbusbyname((!busname) ? TEST_BUSNAME : busname,
             &busnum)) < 0) {
    MIPC_PRINT("Failed to find bus '%s' (err=%d).\n", TEST_BUSNAME,
     rc);
    return -1;
}

sd_seqpkt = mipc_socket(0, MIPC_SOCK_SEQPACKET, 0);
if (sd_seqpkt < 0) {
    MIPC_PRINT("Could not create a MIPC socket (error = %d)\n",
     sd_seqpkt);
    return -1;
}

addr.busNum = busnum;
addr.portNum = TEST_PORTNUM;

if ((rc = mipc_bind(sd_seqpkt, &addr,
          sizeof(struct mipc_sockaddr))) < 0) {
    MIPC_PRINT("Failed to bind socket (%d) (err=%d).\n", sd_seqpkt,
     rc);
    goto recv_close;
}
```

```
addrlen = sizeof(struct mipc_sockaddr);
if ((rc = mipc_getsockname(sd_seqpkt, &addr, &addrlen)) < 0) {
    MIPC_PRINT("Failed to get address of socket (%d) (err=%d).\n",
     sd_seqpkt, rc);
    goto recv_close;
}

if ((rc = mipc_listen(sd_seqpkt, 2)) < 0) {
    MIPC_PRINT("Failed to set socket up to listen (err=%d).\n", rc);
    goto recv_close;
}

sd_accept = mipc_accept(sd_seqpkt, &peer, &addrlen);
if (sd_accept < 0) {
    MIPC_PRINT("Could not accept a connection (err=%d)\n", sd_accept);
    goto recv_close;
}

nbytes = mipc_zrecv(sd_accept, &buf, &zbuf, 0);
if (nbytes < 0) {
    MIPC_PRINT("Could not recv a peer response (err=%d)\n", nbytes);
    goto recv_close2;
}

MIPC_PRINT("Peer sent us '%s'\n", buf);

mipc_zfree(sd_accept, zbuf);

/* our socket is blocking by default, so mipc_zalloc always works */
mipc_zalloc(sd_accept, 0 /* MTU size */, &buf, &zbuf, 0);
```

```
    strcpy((char *)buf, TEST_MSG_REPLY);    /* assume MTU > TEST_MSG_REPLY */
    MIPC_PRINT("We are sending back '%s'\n", buf);

    nbytes = mipc_zsend(sd_accept, zbuf, TEST_MSG_REPLY_SIZE, 0);
    if (nbytes < TEST_MSG_REPLY_SIZE) {
        MIPC_PRINT("Could not send the message to peer (err=%d)\n",
         nbytes);
        mipc_zfree(sd_accept, zbuf);
        goto recv_close2;
    }

    mipc_close (sd_accept);
    mipc_close (sd_seqpkt);
    return 0;

recv_close2:
    mipc_close (sd_accept);
recv_close:
    mipc_close (sd_seqpkt);
    return -1;
}
```

6. AF_MIPC API (LINUX ONLY)

6.1. Introduction

For Linux user-space applications, MIPC 2.0 provides the **AF_MIPC** API, which supports most features of the standard socket API defined by the Berkeley Software Distribution (BSD) (see AF_MIPC Socket API on page 73). The socket family for use with the **AF_MIPC** API is **AF_MIPC**.

Note that the current release does not support user-space (RTP) applications for VxWorks.

6.2. AF_MIPC Socket Address Structure

The **AF_MIPC** API has its own socket address structure:

```
struct sockaddr_mipc {
    unsigned short family;          /* address family (AF_MIPC) */
    unsigned short busNum;          /* bus number */
    unsigned short nodeNum;         /* node number */
    unsigned short portNum;         /* port number */
};
```

where:

family is **AF_MIPC**.

busNum is the number of the bus that the socket is bound to. To obtain the bus number, you can use the **mipc_getbusbyname( )** routine.

nodeNum is the number of the node that the socket is bound to.

portNum is the number of the port that the socket is bound to.

For use with the **bind( )** routine, there are macro definitions for the busNum, nodeNum, and portNum fields (see Macros for Fields in the mipc_sockaddr When Used with mipc_bind( ) on page 75).

6.3. AF_MIPC Socket API

This section presents the **AF_MIPC** socket API. In the current release, which limits the API to Linux applications, you need to include the **mipc.h** header file, as follows:

```
#include <linux/mipc.h>
```

Table 5-1 on page 73lists the BSD socket calls supported by **AF_MIPC**. Unless otherwise noted, the listed routines conform to POSIX (Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition).

Table 5-1 : AF_MIPC Support for BSD Socket Routines

Socket Routine	Comment
accept( )	
bind( )	There are special macros for **bind( )** to use in filling in fields of the **sockaddr_mipc** structure (see Macros for Fields in the mipc_sockaddr When Used with mipc_bind( ) on page 75).
close( )	
connect( )	
getpeername( )	
getsockname( )	
getsockopt( )	See **setsockopt( )** for a list of supported options.
listen( )	
poll( )	
read( )	
recv( )	
recvfrom( )	

recvmsg( )	For some protocols, the **recvmsg( )** routine returns ancillary data items in a **msg_control** array. MIPC does not use this array, which can therefore be set to NULL. The **recvmsg( )** routine supports the following flags: • **MSG_DONTWAIT** • **MSG_PEEK** • **MSG_TRUNC** • **MSG_WAITALL**
select( )	
send( )	
sendmsg( )	The following flag is supported: **MSG_DONTWAIT**
sendto( )	
setsockopt( )	The **setsockopt( )** routine for MIPC does not support standard **SOL_SOCKET** options. The following MIPC-specific **SOL_MIPC** options are supported: • **MIPC_SO_MTU** • **MIPC_SO_RXBUFS** • **MIPC_SO_TXBUFS** • **MIPC_SO_CONN_TIMEOUT** • **MIPC_SO_RECEIVE_TIMEOUT** • **MIPC_SO_SEND_TIMEOUT** • (Deprecated) **MIPC_SO_SHUTDOWN_TIMEOUT** For information on individual options, see the API reference entry for **mipc_setsockopt( )** or **mipc_getsockopt**.
shutdown( )	
socket( )	The following socket types are supported: • **SOCK_DGRAM** • **SOCK_RDM**

	• **SOCK_SEQPACKET** • **SOCK_STREAM**
write( )	

Macros for Fields in the mipc_sockaddr When Used with mipc_bind( )

There are three macros for use in the busNum, nodeNum, and portNum fields of the **sockaddr_mipc** structure (see AF_MIPC Socket Address Structure on page 72) when it is used with the **mipc_bind( )** routine. In each case, the macro allows MIPC to fill in the field value, rather than requiring the application to find the value for the field.

Macro	Description
MIPC_BUS_ANY	MIPC maps this macro to the first bus it attaches to at startup and returns the corresponding bus number in the busNum field of the **mipc_sockaddr** structure. This allows an application running on a node configured for only one bus to get the bus number needed for calls to **mipc_connect( )**, **mipc_send_express( )**, **mipc_sendto( )**, and **mipc_zsendto( )** without needing to call **mipc_getbusbyname( )**.
MIPC_NODE_ANY	This macro tells MIPC to bind to the node calling **mipc_bind( )**, which means that the application does not need to find out its own node number. The node number is returned in the nodeNum field of the **mipc_sockaddr** structure.
MIPC_PORT_ANY	This macro tells MIPC to choose an available port number to bind to. The port number is returned in the portNum field of the **mipc_sockaddr** structure.

6.4. Informational Routines Added to the AF_MIPC Socket API

The routines in Table 5-2 on page 75 provide useful information about MIPC nodes and buses. They are part of the **mipc_** API, but can also be used with the **AF_MIPC** API.

Table 5-2 : Routines Added to the AF_MIPC Socket API

mipc_Routine	Description
mipc_getactivebusesl( )	Get a bitfield of the buses a node can utilize.
mipc_getactivenodesl( )	Get a bitfield of the active MIPC nodes on a bus.
mipc_getbusbyname( )	Get the number of a MIPC bus, given its name.
mipc_getnamebybus( )	Get the name of a MIPC bus, given its number.
mipc_getnodebybus( )	Given a bus number, get the node number of the current node.

6.5. AF_MIPC Symbols for MIPC Version Numbers

The **AF_MIPC** API provides the following symbols that make it possible for an application to find out which version of MIPC it is using:

- **MIPC_VERSION_MAJOR**

  The first number in a MIPC version number given as X.Y. For example, the number 2 in MIPC version 2.0.

- **MIPC_VERSION_MINOR**

  The second value in a MIPC version number given as X.Y. For example, the value 0 in MIPC version 2.0.

6.6. Features of BSD Sockets Not Supported by AF_MIPC

This section covers features of BSD sockets that are not supported by **AF_MIPC**.

Unsupported BSD Socket Routines

The following BSD socket routines are not supported:

- **sockatmark( )**
- **socketpair( )**

Unsupported BSD Send and Receive Flags

The following send and receive flags are not supported:

- **MSG_BCAST**
- **MSG_CTRUNC**
- **MSG_DONTROUTE**
- **MSG_EOR**
- **MSG_MCAST**
- **MSG_OOB**

Unsupported SOL_SOCKET Options

AF_MIPC does not support any SOL_SOCKET options.

Unsupported ioctl( ) Requests

The following BSD **ioctl( )** requests are not supported:

- **SIOCATMARK**
- **SIOCSPGRP**
- **SIOCGPRGRP**
- **FIOASYNC**
- **FIONREAD**
- **FIOSETOWN**
- **SIOGETOWN**

Implicit Binding is Not Supported

AF_MIPC does not support implicit binding of a port name.

Asynchronous connect( ) is Not Supported

AF_MIPC does not support the asynchronous form of the **connect( )** routine.

O_ASYNC File Status Flag is Not Supported

AF_MIPC does not support the **O_ASYNC** file status flag for signal-driven I/O.

Partial Closing of a Connection with the shutdown( ) Routine is Not Supported

AF_MIPC supports full closure of a connection using the **shutdown( )** routine:

```
shutdown(fd,SHUT_RDWR)
```

It does not support partial shutdown of a connection using the **shutdown( )** routine with either **SHUT_RD** or **SHUT_WR**.

Receiving Data into More than One iovec with recvmsg( ) is Not Supported

AF_MIPC does not support receiving data into more than one **iovec** when the **recvmsg( )** routine is used.

The Use of connect( ) with Connectionless Sockets is Not Supported

AF_MIPC does not support the use of the **connect( )** routinewith connectionless sockets--**SOCK_DGRAM** and **SOCK_RDM**.

6.7. Differences Between AF_MIPC and mipc_ Sockets

The **AF_MIPC** APIuses the BSD socket API; the **mipc_** API has its own proprietary API. This section lists significant differences between the two sets of APIs.

- The **AF_MIPC** API uses the **sockaddr_mipc** structure for socket addresses (see AF_MIPC Socket Address Structure on page 72); the **mipc_** API uses its own **mipc_sockaddr** structure for socket addresses (see The mipc_sockaddr Structure on page 36).
- **AF_MIPC** sockets cannot send or receive data at interrupt level.
- **AF_MIPC** sockets cannot send or receive data using zero-copy buffers.
- The following **SOL_MIPC** socket options are not supported by **AF_MIPC**:
  - **MIPC_SO_NBIO**

    This **mipc_** socket option allows you to configure a socket for non-blocking operation. You can configure an **AF_MIPC** socket for non-blocking operation using standard socket mechanisms, such as the **ioctl( ) FIONBIO** request or the **fcntl( ) O_NONBLOCK** flag.
  - **MIPC_SO_CONNECTED_CALLBACK**
  - **MIPC_SO_CONNECTREFUSED_CALLBACK**

  - **MIPC_SO_DISCONNECTED_CALLBACK**
  - **MIPC_SO_NODEJOIN_CALLBACK**
  - **MIPC_SO_NODELEFT_CALLBACK**
  - **MIPC_SO_RX_CALLBACK**
  - **MIPC_SO_TXBUFAVAIL_CALLBACK**
  - MIPC_SO_RXQUEUED_CALLBACK
  - **MIPC_SO_TXBUFAVAIL_THRESHOLD**
- **AF_MIPC** does not provide asynchronous notification of nodes joining or leaving a bus.

  For **mipc_**, this is provided through the **MIPC_SO_NODEJOIN_CALLBACK** and **MIPC_SO_NODELEFT_CALLBACK** routines.

7. MIPC DEMO

7.1. Introduction

MIPC provides a demo application that illustrates MIPC's ability to send and receive data using different types of sockets and connections. In addition, the source code for the application gives software designers who are new to MIPC an example of how to write programs using MIPC's **AF_MIPC** socket API (for the Linux demo) or the **mipc_** socket API (for the VxWorks demo). The source code for the application is located at:

For VxWorks:

installDir**/vxworks-6.8/target/src/multios_ipc/demo/mipcdemo.c**

For Linux:

installDir**/wrlinux-3.0/layers/wrll-multicore/dist/multios_tools/src/mipcdemo.c**

The demo application contains three demos in which a client and a server exchange data. The demos illustrate:

- Connectionless data exchange using **DGRAM**-type sockets (Demo 1).
- Datagram connections using **SEQPACKET**-type sockets (Demo 2).
- Byte stream connections using **STREAM**-type sockets (Demo 3).

As each demo runs, the client and server print out informational messages that allow the user to follow along with the operations being performed (for sample output see Sample Demo Output on page 82).

7.2. Including the Demo Application in a Project

If MIPC is included in a Linux Platform Project, the demo application is included with it by default.

To include the demo application in a VxWorks Image Project, you need to include the **MIPC demo** (**INCLUDE_MIPC_DEMO**) component in your project.

7.3. Running the Demo Application

Before you run the demo application, you should note the following:

- The demo application assumes that nodes running the application are on a bus called **main** (**#define MIPC_DEMO_DEFAULT_BUS "main"**). If you want to run the demo on nodes that are not on **main**, you will need to edit the file **mipcdemo.c** and replace **main** with the name of the bus that the nodes are on (for the path to **mipcdemo.c**, see Introduction on page 80).
- The demo application allows only a single server and a single client to be active at the same time. Running more than one client or server at a time is likely to make the demo fail.

To run the demo application and then terminate it when you are done with it:

1. Start the server on a node with one of the following commands:

   mipcdemo (VxWorks)**mipcdemo &** (Linux)

2. Start the client on the same or a different node and run a specific demo or all three demos with the one of the following commands:

   For VxWorks:

   mipcdemo "all" (runs all three demos)

   mipcdemo "N**"** or **mipcdemo** N, where N is the number of the demo.

   For Linux:

   mipcdemo all (runs all three demos)

   mipcdemo N, where N is the number of the demo.

   The demos are numbered as follows:

   1: MIPC **DGRAM** demo.

   2: MIPC **SEQPACKET** demo.

   3: MIPC **STREAM** demo.

3. Terminate the demo with the following command on either the client's node or the server's node:

   For VxWorks:

   mipcdemo "stop"

   For Linux:

   mipcdemo stop

For **mipcdemo** usage information, enter the following:

For VxWorks:

mipcdemo "help" or **mipcdemo help**

For Linux:

mipcdemo help

This displays information equivalent to that in the preceding steps.

Error During Operation of Demo

If the demo server or client detects a problem while running a demo, it immediately issues an error message and terminates operation. When this happens:

For VxWorks:

Reboot the node or nodes that the client and server are running on--you cannot simply restart the client and server.

For Linux:

Terminate any remaining demo, if you used **mipcdemo** with **all**, and then restart both the demo server and client.

7.4. Sample Demo Output

The following is sample output for a client and server running on the same node.

MIPC DGRAM Demo

The following is sample output for the following command:

mipcdemo "1" `(VxWorks)` **mipcdemo 1** `(Linux)`

```
MIPC DGRAM demo started    client: sending 10 packets of 100 bytes    server: received 10 packets of
MIPC DGRAM demo finished
```

MIPC SEQPACKET Demo

The following is sample output for the following command:

mipcdemo "2" `(VxWorks)` **mipcdemo 2** `(Linux)`

```
MIPC SEQPACKET demo started     client: connecting to server     server: waiting for connection reque
```

MIPC STREAM Demo

The following is sample output for the following command:

mipcdemo "3" `(VxWorks)` **mipcdemo 3** `(Linux)`

```
MIPC STREAM demo started    client: connecting to server     server: waiting for connection request fi
```

8. MIPC SHOW ROUTINES (VXWORKS)

8.1. Introduction

MIPC provides the following command-line show routines:

mipcHelp (see The mipcHelp Command on page 84)

mipcShow (see The mipcShow Command on page 84)

mipcShowBus (see The mipcShowBus Command on page 84)

mipcShowNode (see The mipcShowNode Command on page 85)

mipcShowPort (see The mipcShowPort Command on page 85)

8.2. The mipcHelp Command

The **mipcHelp( )** command displays information about the other show commands:

```
-> mipcHelp


MIPC    {     installed version: 2.0      available show commands:      mipcHelp - this command      m
```

8.3. The mipcShow Command

The **mipcShow( )** command displays general information about your MIPC system. The command does not take any arguments. The following is sample output:

```
-> mipcShow      MIPC-SM SYSTEM     {      base address of SM: 0x84000000      SM memory pool region: s
```

8.4. The mipcShowBus Command

The **mipcShowBus( )** command displays information about a specified bus. The syntax of the command is:

mipcShowBus busNumber

The node on which the command is issued must be attached to the specified bus.

The following is sample output:

```
-> mipcShowBus 0        MIPC-SM BUS 0    {      bus name: "main"     maximum nodes: 2     my node numbe
```

8.5. The mipcShowNode Command

The **mipcShowNode( )** command displays information about a node on a specified bus. The syntax of the command is:

mipcShowNodebusNumber**,**nodeNumber

The node on which the command is issued must be attached to the specified bus.

The following is sample output:

```
-> mipcShowNode 0,0         MIPC-SM NODE 0.0
{
 current state: active (signature=466, heartbeat=36)
 number of ports: 32
 qos: 41 (ISR deferred mode)
 packet pool
 packet size: 1520
 number of buffers: 100
 available buffers: yes
 event pool
 number of events: 32
 available events: yes
 active ports: { 1, 15 }
}
```

8.6. The mipcShowPort Command

The **mipcShowPort( )** command displays information about a specified port. The syntax of the command is:

mipcShowPort busNumber**,**nodeNumber**,**portNumber

The node on which the command is issued must be attached to the specified bus.

The following is sample output:

```
-> mipcShowPort 0,1,4       MIPC-SM PORT 0.1.4
{
 port active: yes
 available buffers: yes
 buffer allocation: 8
}
```